CUDA PARALLEL IMPLEMENTATION OF AIRSPACE CONFLICT DETECTION

A Thesis
Submitted to the Faculty
of
Purdue University
by
Nathan Joseph Clem

In Partial Fulfillment of the
Requirements for the Degree
of
Master of Science in Engineering

May 2015
Purdue University
Fort Wayne, Indiana
For my family, who are always there for me.
ACKNOWLEDGEMENTS

I want to thank my principle advisor, Dr. Elizabeth Thompson, for taking the time to meet with on a weekly basis and contributing significantly to the research in this work. Her efforts in getting funding for this work and pursuing solutions to the many encounters we had cannot be overstated.

This work could not have happened without the backing from Raytheon. Whether it is guiding me along the research or providing funds through the Security and Software Engineering Research Center (S²ERC), Raytheon continuously showed support for this work. I also appreciate the partial support provided by the National Science Foundation (NSF) through the grant DGE-SMP 1010908 titled “Science Master’s Program: Concentration in Wireless Technology and Systems Engineering.”

By accepting the grant provided by S²ERC, I was required to write unpublished status reports on this research every six months. Portions of those unpublished status reports were incorporated into this paper.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>List</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>LIST OF ABBREVIATIONS</td>
<td>xii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xiii</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. CONCEPT OF AIRSPACE CONFLICT DETECTION</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Conflict Detection Overview</td>
<td>3</td>
</tr>
<tr>
<td>2.1.1 Airspace domain</td>
<td>5</td>
</tr>
<tr>
<td>2.1.2 Track development</td>
<td>7</td>
</tr>
<tr>
<td>2.1.3 Geometry development</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Violation Equations</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Violation Comparisons</td>
<td>13</td>
</tr>
<tr>
<td>2.3.1 Planar patch comparisons</td>
<td>13</td>
</tr>
<tr>
<td>2.3.2 Directional comparisons</td>
<td>16</td>
</tr>
<tr>
<td>2.3.3 Buffer zone comparisons</td>
<td>18</td>
</tr>
<tr>
<td>2.3.4 Current track position comparisons</td>
<td>19</td>
</tr>
<tr>
<td>2.3.5 Summary of comparisons</td>
<td>20</td>
</tr>
<tr>
<td>3. SEQUENTIAL VERSION</td>
<td>22</td>
</tr>
<tr>
<td>3.1 Hardware and Software Used on Sequential Version</td>
<td>22</td>
</tr>
<tr>
<td>3.2 Sequential Algorithm</td>
<td>22</td>
</tr>
<tr>
<td>3.2.1 Sequential algorithm part 1: data storage</td>
<td>23</td>
</tr>
<tr>
<td>3.2.2 Sequential algorithm part 2: conflict detection</td>
<td>24</td>
</tr>
<tr>
<td>3.3 Sequential Version Results</td>
<td>26</td>
</tr>
<tr>
<td>3.4 Updated Sequential Version</td>
<td>28</td>
</tr>
<tr>
<td>4. PARALLEL PROCESSING</td>
<td>31</td>
</tr>
</tbody>
</table>
4.1 Graphical Processing Unit (GPU) ................................................................. 31
4.2 GPU Architecture .......................................................................................... 32
4.3 CUDA Programming Overview .................................................................... 33

5. CUDA PERFORMANCE ....................................................................................... 40
5.1 CUDA Libraries............................................................................................. 40
5.1.1 NPP (NVIDIA Performance Primitives)................................................... 41
5.1.2 cuBLAS (NVIDIA CUDA Basic Linear Algebra Subroutines) ............... 49
5.1.3 Thrust ........................................................................................................ 63
5.2 Asynchronous Programming ......................................................................... 65

6. PARALLEL VERSION .......................................................................................... 86
6.1 Synchronous Single GPU .............................................................................. 86
6.2 Issues with Synchronous Single GPU Version.............................................. 96
6.2.1 NPP library grid dimension limitation ...................................................... 96
6.2.2 GPU global memory limitation .................................................................. 99
6.2.3 Transfer overhead limitation ................................................................... 100
6.3 Asynchronous Single GPU .......................................................................... 101
6.4 Asynchronous Multiple GPUs ..................................................................... 107

7. CONCLUSION ..................................................................................................... 111

LIST OF REFERENCES ................................................................................................ 113

A. CUDA LIBRARY TEST PROGRAMS AND RAW DATA ............................... 115

A.1 NPP Test Programs and Raw Data ................................................................. 115
A.1.1 nppiCompare_32f_C1R test program .................................................... 116
A.1.2 nppiCompare_32f_C1R raw data............................................................ 120
A.1.3 nppiCompareC_32f_C1R test program.................................................... 121
A.1.4 nppiCompareC_32f_C1R raw data........................................................... 124
A.1.5 nppsAdd_8u_ISfs test program .............................................................. 125
A.1.6 nppsAdd_8u_ISfs raw data .................................................................... 129
A.1.7 nppsAdd_8u_Sfs test program ............................................................... 130
A.1.8 nppsAdd_8u_Sfs raw data ...................................................................... 134
A.1.9 nppsAdd_32f_I test program .................................................................. 135
A.1.10 nppsAdd_32f_I raw data ...................................................................... 139
A.1.11 nppsAddC_32f_I test program ............................................................. 140
A.1.12 nppsAddC_32f_I raw data.................................................................... 143
A.1.13 nppsCopy_32f test program ................................................................. 144
A.1.14 nppsCopy_32f raw data ....................................................................... 147
A.1.15  nppsDiv_32f test program .......................................................... 148
A.1.16  nppsDiv_32f raw data ................................................................. 152
A.1.17  nppsDiv_32f_I test program ...................................................... 153
A.1.18  nppsDiv_32f_I raw data ............................................................. 157
A.1.19  nppsDivC_32f_I test program .................................................... 157
A.1.20  nppsDivC_32f_I raw data ......................................................... 161
A.1.21  nppsMul_32f_I test program ...................................................... 157
A.1.22  nppsMul_32f_I raw data ............................................................. 165
A.1.23  nppsMulC_32f_I test program .................................................... 166
A.1.24  nppsMulC_32f_I raw data ........................................................... 169
A.1.25  nppsSqr_32f_I test program ...................................................... 170
A.1.26  nppsSqr_32f_I raw data ............................................................. 173
A.1.27  nppsSqrt_32f_I test program .................................................... 174
A.1.28  nppsSqrt_32f_I raw data ........................................................... 177
A.1.29  nppsSub_32f test program ........................................................ 178
A.1.30  nppsSub_32f raw data ............................................................... 182
A.1.31  nppsSub_32f_I test program ...................................................... 183
A.1.32  nppsSub_32f_I raw data ............................................................. 187
A.1.33  nppsSubC_32f_I test program .................................................... 187
A.1.34  nppsSubC_32f_I raw data ........................................................... 191
A.2  cuBLAS Test Programs and Raw Data .............................................. 192
  A.2.1  cublasSgemm execution time test program ................................... 192
  A.2.2  cublasSgemm execution time raw data ....................................... 199
  A.2.3  cuBLAS only cublasSgemm test program .................................... 200
  A.2.4  NPP only cublasSgemm test program ....................................... 203
A.3  Thrust Test Programs and Raw Data .............................................. 206
  A.3.1  Thrust count_if test program .................................................... 206
  A.3.2  Thrust count_if raw data .......................................................... 209
  A.3.3  Thrust remove_copy_if test program ........................................ 211
  A.3.4  Thrust remove_copy_if raw data .............................................. 215
  A.3.5  Thrust sequence test program ................................................ 216
  A.3.6  Thrust sequence raw data ....................................................... 219
B.  ASYNCHRONOUS TEST PROGRAMS AND RAW DATA ...................... 221
  B.1  Synchronous Single GPU Test Program ......................................... 221
  B.2  Synchronous Single GPU Raw Data .............................................. 224
  B.3  Asynchronous Single GPU Depth First Test Program ....................... 225
  B.4  Asynchronous Single GPU Depth First Raw Data ............................ 227
  B.5  Asynchronous Single GPU Breadth First Test Program .................... 228
  B.6  Asynchronous Single GPU Breadth First Raw Data .......................... 231
  B.7  Synchronous Multiple GPU Test Program ..................................... 232
  B.8  Synchronous Multiple GPU Raw Data .......................................... 236
Page

B.9  Asynchronous Multiple GPU Depth First Test Program............................. 237
B.10  Asynchronous Multiple GPU Depth First Raw Data................................ 241
B.11  Asynchronous Multiple GPU Breadth First Test Program ....................... 242
B.12  Asynchronous Multiple GPU Breadth First Raw Data ............................. 246

C.  SEQUENTIAL VERSION PROGRAM AND RAW DATA............................... 248
C.1  Sequential Version Program ........................................................................ 248
C.2  Sequential Version Raw Data ...................................................................... 261

D.  PARALLEL VERSION PROGRAMS AND RAW DATA ................................ 267
D.1  Synchronous Parallel Version Program......................................................... 267
D.2  Synchronous Parallel Version Raw Data....................................................... 291
D.3  Asynchronous Single GPU Parallel Version Program................................. 291
D.4  Asynchronous Single GPU Parallel Version Raw Data............................... 305
D.5  Asynchronous Multiple GPU Parallel Version Program............................. 305
D.6  Asynchronous Multiple GPU Parallel Version Raw Data........................... 326
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Grid and block limitations for CUDA devices with compute capability of 1.3</td>
<td>37</td>
</tr>
<tr>
<td>5.1 List of NPP kernels tested</td>
<td>42</td>
</tr>
<tr>
<td>5.2 Execution time performance results for NPP kernels</td>
<td>46</td>
</tr>
<tr>
<td>5.3 NPP result discrepancy percentages and average margin of errors</td>
<td>48</td>
</tr>
<tr>
<td>5.4 Execution time performance results for Thrust kernels</td>
<td>65</td>
</tr>
<tr>
<td>A.1 Column representation for majority of the NPP raw data results</td>
<td>115</td>
</tr>
<tr>
<td>A.2 Column representation for some of the NPP raw data results</td>
<td>115</td>
</tr>
<tr>
<td>A.3 Column representation of cuBLAS raw data results</td>
<td>192</td>
</tr>
<tr>
<td>A.4 Column representation for Thrust raw data results</td>
<td>206</td>
</tr>
<tr>
<td>B.1 Column representation for single GPU raw data results</td>
<td>221</td>
</tr>
<tr>
<td>B.2 Column representation for multiple GPU raw data results</td>
<td>221</td>
</tr>
<tr>
<td>C.1 Column representation for sequential version raw data results</td>
<td>248</td>
</tr>
<tr>
<td>D.1 Column representation for parallel version raw data results</td>
<td>267</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Example of violations</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Airspace domain axis orientation</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Track representation</td>
<td>7</td>
</tr>
<tr>
<td>2.4 Geometry representation</td>
<td>9</td>
</tr>
<tr>
<td>2.5 Graphical representation of the variables of Equation 2.2</td>
<td>11</td>
</tr>
<tr>
<td>2.6 Planar patch comparison example</td>
<td>15</td>
</tr>
<tr>
<td>2.7 Relative placement and sign of $T_{hit}$</td>
<td>18</td>
</tr>
<tr>
<td>2.8 Track locations relative to geometry borders</td>
<td>20</td>
</tr>
<tr>
<td>3.1 Sequential version flowchart</td>
<td>25</td>
</tr>
<tr>
<td>3.2 Sequential version execution times</td>
<td>28</td>
</tr>
<tr>
<td>3.3 Updated sequential version execution times</td>
<td>30</td>
</tr>
<tr>
<td>4.1 GPU architecture</td>
<td>33</td>
</tr>
<tr>
<td>4.2 Required CUDA programming steps</td>
<td>34</td>
</tr>
<tr>
<td>4.3 Visual representation of thread organization</td>
<td>36</td>
</tr>
<tr>
<td>4.4 Example of warp distribution</td>
<td>39</td>
</tr>
<tr>
<td>5.1 Flowchart for NPP test programs</td>
<td>44</td>
</tr>
<tr>
<td>5.2 Execution times for cublasSgemm</td>
<td>52</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>5.3 Visual Profiler results for <code>cublasSgemm</code> kernel</td>
<td>54</td>
</tr>
<tr>
<td>5.4 Visual Profiler results for NPP equivalent of <code>cublasSgemm</code></td>
<td>55</td>
</tr>
<tr>
<td>5.5 Properties session for <code>cublasSgemm</code> kernel</td>
<td>60</td>
</tr>
<tr>
<td>5.6 Properties session for <code>nppsMulC_32f_I</code> kernel</td>
<td>62</td>
</tr>
<tr>
<td>5.7 Properties session for <code>nppsAddC_32f_I</code> kernel</td>
<td>62</td>
</tr>
<tr>
<td>5.8 Depth first method of asynchronous programming</td>
<td>67</td>
</tr>
<tr>
<td>5.9 Breadth first method of asynchronous programming</td>
<td>68</td>
</tr>
<tr>
<td>5.10 NVIDIA Visual Profiler screenshot of a program running on the default stream</td>
<td>69</td>
</tr>
<tr>
<td>5.11 NVIDIA Visual Profiler screenshot of a program running four streams depth first</td>
<td>70</td>
</tr>
<tr>
<td>5.12 NVIDIA Visual Profiler screenshot of a program running four streams breadth first</td>
<td>72</td>
</tr>
<tr>
<td>5.13 NVIDIA Visual Profiler screenshot of a program running four streams breadth first with 20 kernels per stream</td>
<td>73</td>
</tr>
<tr>
<td>5.14 NVIDIA Visual Profiler screenshot of a program running four streams breadth first with 30 kernels per stream</td>
<td>74</td>
</tr>
<tr>
<td>5.15 Execution times for a single GPU running asynchronously</td>
<td>76</td>
</tr>
<tr>
<td>5.16 NVIDIA Visual Profiler screenshot of a program using the default stream and alternating execution between two GPU devices</td>
<td>79</td>
</tr>
<tr>
<td>5.17 NVIDIA Visual Profiler results for a program running on two GPUs using the depth first method of asynchronous programming</td>
<td>80</td>
</tr>
<tr>
<td>5.18 NVIDIA Visual Profiler results for a program running on two GPUs using the breadth first method of asynchronous programming</td>
<td>81</td>
</tr>
<tr>
<td>5.19 Breadth first method running on two GPUs with 20 kernels per stream</td>
<td>82</td>
</tr>
<tr>
<td>5.20 Breadth first method running on two GPUs with 50 kernels per stream</td>
<td>83</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>5.21</td>
<td>84</td>
</tr>
<tr>
<td>6.1</td>
<td>93</td>
</tr>
<tr>
<td>6.2</td>
<td>94</td>
</tr>
<tr>
<td>6.3</td>
<td>95</td>
</tr>
<tr>
<td>6.4</td>
<td>102</td>
</tr>
<tr>
<td>6.5</td>
<td>106</td>
</tr>
<tr>
<td>6.6</td>
<td>107</td>
</tr>
<tr>
<td>6.7</td>
<td>109</td>
</tr>
<tr>
<td>6.8</td>
<td>110</td>
</tr>
</tbody>
</table>
LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD</td>
<td>Advanced Micro Devices</td>
</tr>
<tr>
<td>cuBLAS</td>
<td>NVIDIA CUDA Basic Linear Algebra Subroutines</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Processing Unit</td>
</tr>
<tr>
<td>HSA</td>
<td>Heterogeneous System Architecture</td>
</tr>
<tr>
<td>ID</td>
<td>Identification</td>
</tr>
<tr>
<td>NPP</td>
<td>NVIDIA Performance Primitives</td>
</tr>
<tr>
<td>NVCC</td>
<td>NVIDIA CUDA Compiler</td>
</tr>
<tr>
<td>OpenGL</td>
<td>Open Graphics Library</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SM</td>
<td>Streaming Multiprocessor</td>
</tr>
<tr>
<td>UAV</td>
<td>Unmanned Aerial Vehicle</td>
</tr>
</tbody>
</table>
The maintaining of separation between aircraft or any air bound projectile is an important concept in the daily lives of many people. Whether it is air traffic controllers in public transportation or military use in the battlefield, maintaining projectile separation can be the difference between success and tragedy. To prevent and resolve failures and operational errors that may occur, automated tools are being developed to assist in controlling traffic in airspaces. These automated tools not only need to process logistical data for each aircraft or projectile in a particular airspace, but they also need to provide collision-avoidance techniques such as airspace deconfliction. Developing these kinds of automated tools may be too computationally demanding for typical Central Processing Units (CPUs). Solutions to handling such computational stresses may come in the form of supercomputers or a series of computers executing in parallel. Though these come at a price and require additional resources, a solution involving low cost Graphical Processing Units (GPUs) may provide the processing power required by the air traffic control systems because of the parallel nature in which it is able to perform computations.
1. INTRODUCTION

The research in this work involves converting a CPU version of a conflict detection algorithm into a GPU equivalent. It is expected that the parallelization of the algorithm will outperform the CPU baseline version because of the greatly increased amount of parallel throughput capability of a GPU. This work focuses only on the conflict detection algorithm and not that of an airspace deconfliction algorithm. The method of parallelization could still be used as a blueprint to achieve the goal of having a fully automated tool executing both conflict detections and airspace deconflictions in parallel.

Even though a GPU comes standard on laptops and personal computers (PCs), developing a program to execute on a GPU is not as simple as developing one for the CPU. There are many constraints and factors a programmer must abide by in order to fully utilize the capabilities of a GPU [1]. GPU manufacturers such as NVIDIA have developed software development kits (SDKs) that allow programmers to easily communicate with a GPU. The SDK developed by NVIDIA is called Compute Unified Device Architecture (CUDA), which is used to communicate with NVIDIA GPUs [2]. For this particular work, CUDA will be used to develop and test a parallel conflict detection algorithm, since the HP Z800 Workstation located in ET 324 on the campus of IPFW has two CUDA capable devices on board.
The CUDA SDK is a free download provided by NVIDIA that comes with several program libraries [2]. The version of CUDA used in the implementation of the parallel programs developed in this work is CUDA v5.5 [3], although, as of this writing, CUDA v7.0 is now available as a free download [2]. The performance of the libraries in CUDA will be tested and compared to the CPU equivalent operations. By testing the performance of these libraries along with researching the capabilities of GPUs, a parallel version of the conflict detection algorithm can be developed. The results and recommendations of this work could be used for possible solutions for similar algorithms in future projects.
2. CONCEPT OF AIRSPACE CONFLICT DETECTION

2.1 Conflict Detection Overview

Airspace conflict detection is not limited to only aircraft-to-aircraft spatial avoidance. The ability to find conflict violations between aircrafts and designated airspaces is also vital for recognizing clear path trajectories for said aircrafts. For purposes of this research, the developed airspace conflict detection programs check if a projectile or an aircraft is in or will be in conflict with an object or a designated area. The term track will be used to describe the projectile or aircraft, and the term geometry will be used to describe the object or designated airspace.

There are two types of conflicts that can occur within a given airspace. The first type of conflict is called an imminent violation. This occurs when a track is outside a geometry but is within a predetermined buffer zone. This buffer zone can either be the time it takes to reach the geometry or a set distance away from the geometry. The second type of conflict to be determined is strictly called a violation. This occurs when the location of a track is within the actual boundaries of a geometry.

Figure 2.1 shows examples of the various types of violations. The cubes represent the geometries, whereas the circle represents the buffer zone. The points with the arrows represent the tracks and their direction of travel. Take note of the example
shown in Figure 2.1 (a). Even though the track is within the buffer zone, it is not considered an imminent violation since the direction of the track does not point towards the geometry.

It is important to note that all values generated and calculated for every variable used in the developed programs are unit measurements and do not represent any real world measurement units such as meters or feet. This is not to say that the developed programs cannot handle real world measurements. Unit values were used to provide a template that would allow for conversions into real world values.
Fig. 2.1 Example of violations

2.1.1 Airspace domain

The simulated airspace used to contain all the tracks and all the geometries is represented by a three-dimensional Cartesian system. Each axis represents a direction in space. The x-axis represents altitude where the positive x direction is of higher elevation and the negative x direction is of lower elevation. The y-axis represents the east and west
directions of the three-dimensional space. The positive y direction points east and the negative y direction points west. The z-axis represents north and south with the positive z direction pointing north and the negative z direction pointing south. Figure 2.2 shows the three-dimensional airspace domain along with the corresponding directions that each axis represents. In this work, the airspace is limited to ±500.0 units. This limit is purely arbitrary and can be changed to any value the user deems necessary.

![Fig. 2.2 Airspace domain axis orientation](image-url)
2.1.2 Track development

Information regarding a track consists of its current position and its previous position. The Cartesian coordinate system is used to represent those positions. The values used for this representation are stored as floats in the programs and are created by using the random number generator that is provided by the C++ standard library. The values for the track position’s coordinates must fall within the range of ±500.0 due to the restrictions of the experimental airspace described in Section 2.1.1. The distance between the tracks’ previous positions and current positions are also randomized and limited to five spatial units. This simulates tracks traveling at different velocities and limits their velocities. Figure 2.3 displays a representation of a track along with pertinent information about that track.

Fig. 2.3 Track representation
2.1.3 Geometry development

For this work, the geometries will be represented by three-dimensional rectangles. This is done for simplicity. To further simplify the math required to find violations, all sides of all geometries are assumed to be parallel to their corresponding coordinate axes.

The data needed to describe a geometry requires four coordinate locations and two scalar values which represent elevation. The creation of a geometry begins with randomly generating four Cartesian point positions using the random number generator that is available in C++. These four coordinate positions represent the corners of a finite rectangular planar patch that is parallel to the yz-coordinate plane. Reference to these four positions can be determined by their location in relation to the directions corresponding to the three-dimensional Cartesian axes. For example, the coordinate located closest to the northwest is the northwest coordinate.

The next step is to create two scalar values called maximum height and minimum height. They represent the elevation relative to the newly created planar patch. By adding these values to the x coordinates of the four created points, the upper and lower bounds of the geometry can be found. Adding the maximum height to the x coordinates yields the upper planar patch of the geometry and subtracting the minimum height to the x coordinates yields the lower planar patch. Figure 2.4 displays a graphical representation of a geometry along with its necessary components and labels.
Fig. 2.4 Geometry representation

The actual size of a geometry can be anything the user wants. For this work, the size of the geometries is limited to prevent the scenario in which a newly created geometry encompasses the entire airspace domain or a large majority of it. Unlike the track value limitations, the geometry limitations are dependent on the domain of the overall experimental airspace. The sides of the geometry that lie parallel to the y-axis and the z-axis range in length from one to 26 units. In other words, the maximum length for these particular sides is roughly 2.5% of the overall airspace range of 1,000 units along those axes. The sides of the geometry lying parallel to the x-axis range in length from one to 52 units. This range is roughly 5% of the x-axis range of 1,000 units. These ranges are purely arbitrary and can be modified to the user’s specific preference.
2.2 Violation Equations

The equations used to find potential violations are inherently sequential. This means the input of one equation is dependent on the output of a preceding equation. The first step in finding a violation requires solving the intersection point between the track’s trajectory and the geometry. To begin this inherently sequential process, the first thing to do is to find the trajectory of the track. This is done using Equation 2.1, where \( P \) is the coordinates for the track’s previous position and \( C \) is the coordinates for the track’s current position. Subtracting the previous position from the current position will yield the track’s directional vector.

\[
\vec{b} = C - P \tag{2.1}
\]

The resulting directional vector \( \vec{b} \) is then used in the calculation of \( T_{hit} \) described in Equation 2.2 below [4].

\[
T_{hit} = \frac{\vec{n} \cdot (A-C)}{\vec{n} \cdot \vec{b}} \tag{2.2}
\]

In Equation 2.2, \( \vec{n} \) is an outward normal to a side (a.k.a. plane) of the geometry, \( \vec{b} \) is the vector obtained from Equation 2.1, and the denominator represents the dot product between these two entities. Furthermore, in Equation 2.2, \( A \) is a point on the plane, and \( C \) is the track’s current position on the trajectory, \( \vec{b} \). The variable \( T_{hit} \) represents a position in space along the track trajectory at which it intersects with the plane. The variables of Equations 2.1 and 2.2 are shown in Figure 2.5.
The actual intersection point of the trajectory and the plane is then calculated using Equation 2.3 below.

\[
\text{Intersection Point} = C + \vec{b}T_{\text{hit}}
\]  

(2.3)

In Equation 2.3, \( \vec{b} \), \( C \), and \( T_{\text{hit}} \) are the same as those defined in Equation 2.2. This intersection point is the point of intersection between the trajectory and an infinite plane.

Having determined the intersection point of the trajectory with an infinite plane, it must be determined whether or not this intersection point lies within the bounds of the rectangular side of the geometry of interest and if that side is in front of the track. These factors are determined using simple comparisons which will be discussed in Section 2.3.

Once the intersection point has been determined, several pieces of information about the track and its relation to the geometry are also known. Such information includes the track’s distance to the geometry (Equation 2.7), its velocity (Equation 2.9),
and the time it would take for the track to reach the geometry (Equation 2.10). The distance is first found by computing the difference between the x, y, and z coordinates of the intersection point found in Equation 2.3 and those of the current position of the track, \( C \). Those equations are shown in Equations 2.4 – 2.6.

\[
\begin{align*}
\text{Dist}_x &= \text{Intersection Point}_x - C_x \quad (2.4) \\
\text{Dist}_y &= \text{Intersection Point}_y - C_y \quad (2.5) \\
\text{Dist}_z &= \text{Intersection Point}_z - C_z \quad (2.6)
\end{align*}
\]

The resulting component differences are then inserted into the distance formula of Equation 2.7, which utilizes \( \text{Dist}_x, \text{Dist}_y, \text{and} \text{Dist}_z \) computed in Equations 2.4 – 2.6. The resulting value, \( \text{DistInt} \), represents the distance from the current position of the track to the side of the geometry that it is intersecting.

\[
\text{DistInt} = \sqrt{\text{Dist}_x^2 + \text{Dist}_y^2 + \text{Dist}_z^2} \quad (2.7)
\]

The velocity of an individual track can be calculated using the vector \( \vec{b} \) of Equation 2.1. Inputting the x, y, and z components of \( \vec{b} \) into Equation 2.8 yields the distance between the previous position and the current position of the track, \( \text{CoorDist} \). The variable \( \text{CoorDist} \) in Equation 2.8 can subsequently be used to calculate the velocity \( \text{Vel} \) simply by dividing the distance \( \text{CoorDist} \) by the time it took for the track to reach its current position from its previous position. Since the track data is simulated and not a part of any real world data set, an arbitrary time difference value was defined to simulate the difference in time between the track’s current position and its previous position. This arbitrary time variable, called \( \text{TimeDiff} \), is used in Equation 2.9 to find the velocity of the track.
\[ CoorDist = \sqrt{b_x^2 + b_y^2 + b_z^2} \]  
\[ Vel = \frac{CoorDist}{TimeDiff} \]  

The final piece of information useful for understanding the situation surrounding a potential violation is the predicted time it would take for the track to reach the point of intersection with the side (a.k.a. plane) of a geometry. This is calculated as \( TimeInt \) in Equation 2.10, which utilizes \( Vel \) from Equation 2.9 and \( DistInt \) from Equation 2.7.

\[ TimeInt = \frac{DistInt}{Vel} \]  

2.3 Violation Comparisons

Four comparisons are used to determine what kind of violation, if any, is occurring between a track and a geometry. The first comparison determines if the intersection point between a track trajectory and the side of a geometry is within the bounds of the planar patch that represents the side of the geometry. The second determines the direction in which the track is traveling. The third comparison determines if the track is within the designated buffer zone or not. The final comparison determines if a track is inside the borders of a geometry or not.

2.3.1 Planar patch comparisons

As discussed in Section 2.2, the equations used to find the intersection point of a trajectory and a plane assumes the plane is an infinite plane. Yet, the planes used to
represent the sides of a three-dimensional rectangle are finite in size. To determine if an intersection point is within the bounds of the finite planes of a geometry, the coordinates at the outer boundaries of the geometry are compared to the coordinates of the intersection point.

Figure 2.6 shows an example highlighting one side of a geometry. As described in Section 2.1.3, each plane that represents a side of a geometry is parallel to its corresponding coordinate plane. The planar patch in Figure 2.6 is parallel to the xy-coordinate plane. In this figure, A, B, C, and D represent points located on the boundaries of the planar patch. Point I is the calculated intersection point. The y coordinates of points A and B, Ay and By, respectively, are compared to the y coordinate of I, Iy. If the value of Iy is greater than Ay and less than By, then the y coordinate of the intersection point is within the range of y values of the planar patch. Similarly, the x coordinates of points C and D are compared to those of I. IfIx is greater than Dx and less than Cx, then the x coordinate of the intersection point is within the range of x values of the planar patch.
Since the planar patch is parallel to the xy-coordinate plane, then all z coordinates located on the plane have the same value. Therefore, there is no need to perform equivalent calculations on the coordinate $I_z$. Both $I_x$ and $I_y$ have to be within their respective ranges in order to conclude that the intersection point is actually intersecting the finite plane of a geometry. If one or both coordinates are not within range, then the intersection point lies outside the bounds of the planar patch. Thus, in order for a track’s trajectory to truly intersect with a side of a geometry that is parallel to the xy-coordinate plane, Equations 2.11 – 2.13 must be true.

\[
Min_x < I_x < Max_x \quad (2.11)
\]

\[
Min_y < I_y < Max_y \quad (2.12)
\]
\[ I_z = \text{Plane's z coordinate} \]  
(2.13)

The values \textit{Max} and \textit{Min} in Equations 2.11 and 2.12 correspond with the maximum \( x \) and \( y \) values and the minimum \( x \) and \( y \) values of the planar patch of a geometry that is parallel to the \( xy \)-coordinate plane. Since the geometries vary in size and are located in various positions across the experimental airspace, the maximum and minimum values and their corresponding ranges will be unique for each geometry. If the side of a geometry is parallel to the \( yz \)-coordinate plane, then the conditions of Equations 2.14 – 2.16 determine whether an intersection has occurred.

\[ \text{Min}_y < I_y < \text{Max}_y \]  
(2.14)

\[ \text{Min}_z < I_z < \text{Max}_z \]  
(2.15)

\[ I_x = \text{Plane's x coordinate} \]  
(2.16)

Finally, if the side of a geometry is parallel to the \( xz \)-coordinate plane, then the following equations must hold true for an intersection with a geometry to take place.

\[ \text{Min}_x < I_x < \text{Max}_x \]  
(2.17)

\[ \text{Min}_z < I_z < \text{Max}_z \]  
(2.18)

\[ I_y = \text{Plane's y coordinate} \]  
(2.19)

### 2.3.2 Directional comparisons

The second comparison required for establishing whether or not a violation has occurred involves determining whether the geometry is in front of or behind the track trajectory. Equation 2.1 in Section 2.2 calculated the directional vector for the track, but the direction of the track trajectory in relation to the geometry remains unclear without
certain spatial parameters. One such spatial parameter is the value calculated by Equation 2.2. As described in Section 2.2, the variable $T_{hit}$ represents the location of the intersection point along its trajectory. If $T_{hit}$ is positive, then the location of the intersection point is in front of the track trajectory. If $T_{hit}$ is negative, then the location of the intersection point is behind the track. If $T_{hit}$ equals zero, then the track is located exactly at the intersection point. In the current application, having knowledge that the intersection point is behind the track is useless since the intersection with the geometry already happened in the past. Knowing what geometries the track is going to intersect in the future is necessary in determining violations. Thus, only positive values of $T_{hit}$ are of interest, which can be ascertained with a simple sign check.

Figure 2.7 displays the relationship of $T_{hit}$ with a track and surrounding geometries. Since the track is headed away from Geometry A, the negative $T_{hit}$ that corresponds with the intersection point at Geometry A is deemed useless. In contrast the positive value of $T_{hit}$ with respect to Geometry B is an indication that the track will eventually intersect with Geometry B assuming it continues on its current trajectory.
2.3.3 Buffer zone comparisons

The third comparison needed for determining what type of violation, if any, is occurring is whether or not the current track location is within the buffer zone of a geometry. The buffer zone acts as a warning system that tells the user that a potential violation may occur if the track continues on its current trajectory towards a geometry. The buffer zone can be defined as either the distance from a geometry or the time to reach the geometry. The distance or time can be adjusted according to user specifications. For the simulations recorded here, the buffer zone was defined as the time it would take for a track to reach a geometry, and an arbitrary value of 180 time units was utilized for...
this purpose. Thus, tracks within 180 time units of an intersection point were deemed to be within the buffer zone. However, a track within the buffer zone does not necessarily imply an imminent violation -- the trajectory of a track must also be intersecting the geometry, as demonstrated in Figure 2.1 of Section 2.1.

By comparing the value of $T_{int}$ in Equation 2.10 to the arbitrary time value defining the buffer zone limit, it can be determined if the track is within the buffer zone.

2.3.4 Current track position comparisons

The final comparison needed for the violation check sequence is determining whether or not the current position of the track is within the borders of a geometry. This is done by counting the number of times the trajectory intersects a geometry. If the trajectory intersects twice, once on entry and once on exit, then the track is outside the geometry. If the track only intersects with a geometry once, only on exit, then the track is located inside the geometry.

Figure 2.8 illustrates these concepts. Geometry A is intersected twice at points $l_1$ (entry point) and $l_2$ (exit point) by the trajectory of the track located at $C_1$. On the other hand, Geometry B has only a single intersection point. Thus, $C_1$ is outside Geometry A and $C_2$ is inside Geometry B.
2.3.5 Summary of comparisons

In summary, all four comparisons are performed to determine which type of violation is occurring, if any. First, all intersection points between the tracks and the geometries are calculated (Section 2.3.1). The result of this calculation also includes those intersection points that are occurring behind the track trajectories. To eliminate those past intersection points, it must be determined which intersection points are in front of the track trajectories (Section 2.3.2). It must then be found which of those remaining intersection points are occurring within the buffer zone (Section 2.3.3). After this point, all remaining tracks and geometries fall under one of two categories of violations. They will either be an imminent violation or an actual violation. All track and geometry pairs that were eliminated by the previous comparisons are assumed not to be any type of violation. For the remaining track and geometry pairs, it must be determined whether the track is within the bounds of a geometry (Section 2.3.4).
Having determined which tracks will be or are currently in violation of a geometry, information about these violations can be relayed to the user. Such information may include which track and geometry pairs are in conflict. Other information can include the distance of the track from a geometry, its speed, and the predicted time to intersect.
3. SEQUENTIAL VERSION

3.1 Hardware and Software Used on Sequential Version

A sequential version of the conflict detection algorithm was developed to be used as a basis of comparison for the parallel implementation. The sequential version was written with C++ on an HP Z800 Workstation located in room 324 of the Engineering, Technology, and Computer Science Building on the campus of Indiana University Purdue University Fort Wayne (IPFW). The HP Z800 Workstation is equipped with 12 Intel Xeon x5660 2.8 GHz processors, but only one processor was used for the sequential execution. The operating system used to develop and test the program was CentOS Linux 2.6.32-71.29.1e16.x86-64. The text editor gedit was used to write the program, and the Linux terminal was used to compile and run the sequential program.

3.2 Sequential Algorithm

The sequential program is composed of a single C++ file which can be logically divided into two parts. In the first part of the program, all the simulated data needed for subsequent conflict detection calculations is created. Conflict detection calculations and comparisons occur in the second part of the program. Only the execution time of the
second part of the program is recorded and used for comparison to the parallel implementation of conflict detection. The rationale is that in the actual implementation of the algorithm, the use of real data would eliminate the need for simulated data.

3.2.1 Sequential algorithm part 1: data storage

The track and geometry data used for conflict detection are declared as floats and stored in arrays. The data used for the program are random but also have certain restrictions which are described in Sections 2.1.2 and 2.1.3. The track data consists of the coordinates for the previous and current positions of the track, stored in six arrays – three for the x, y, and z coordinates of the previous position and three for those of the current position. Each of these arrays contains T elements, where T equals the number of tracks in the currently defined airspace.

The geometry data is stored in 14 arrays of length G, where G is total number of geometries in the currently defined airspace. Twelve of the 14 arrays contain the x, y, and z coordinates of the four points that make the initial planar patch as described in Section 2.1.3. The final two arrays contain the maximum height and the minimum height of the corresponding geometries also described in Section 2.1.3. In addition, the outward normals for each side of each geometry are stored in 18 more arrays. Six groups of three arrays contain the x, y, and z coordinates of each outward normal for each side of each geometry.
3.2.2 Sequential algorithm part 2: conflict detection

The second portion of the program contains the computations for conflict detection. Figure 3.1 displays the flowchart for the sequential implementation of the code. The algorithm consists of a nested loop containing two for loops. The outer for loop increments through the tracks while the inner loop iterates through the geometries. Within the inner loop, a single track is sequentially compared with every side of every geometry. When the track has been compared to the last geometry, the inner loop is exited, and the outer loop increments to the next track. This process repeats until every track is compared with every side of every geometry. Thus, the algorithm sequentially compares one track with one side of a geometry until all tracks have been compared to all sides of all geometries.
The program outputs relevant information about a violation or an imminent violation during program execution. This information is not saved for subsequent reporting. Instead, once a violation has been discovered, the program calculates information about the conflict and immediately records it to a text file. Since the number of violations is initially unknown, this scheme was thought to be less complicated than an alternative involving linked lists or other types of dynamic memory allocation of data structures to temporarily store this information for subsequent print out. Furthermore, there were no foreseeable advantages to dynamically storing the conflict data for retrieval at the end of the program.

3.3 Sequential Version Results

To test the performance of the sequential version of the conflict detection algorithm, the execution time of the second part of the program (Section 3.2.2) was recorded. The execution time was found by inserting the C++ time function `clock()` immediately before and after the portion of the program that handles conflict detection. By taking the difference of these two values and dividing it by the constant value of `CLOCKS_PER_SEC`, the execution time in seconds can be calculated. The C++ function `clock()` returns the number of clock cycles that have occurred on the CPU since the start of the program. Taking the difference between the number of clock cycles obtained from two different locations in the program yields the number of clock cycles for the portion of the program between those locations. Dividing this value by the constant value
CLOCKS_PER_SEC, which contains the number of clock cycles per second for the CPU, will yield the execution time in seconds for that portion of the program.

For testing purposes, the code will initially execute using 500 tracks and 50 geometries. To eliminate the need for the tester to manually increase the number of tracks and geometries after each recording of the execution time, the entire main function of the program was placed within an infinite while loop. After each iteration of the infinite while loop, the number of tracks and geometries were automatically increased by 500 and 50, respectively, and a new set of simulated data was created.

Within each iteration of the while loop, the conflict detection portion of the program was repeated 5 times using the same simulated data set. Execution times for each of these 5 repetitions of the algorithm were recorded, and then an average value was obtained for comparison to the execution time of the parallel version of the code. An average was found to account for any slight variations in the execution time that may occur.

Having finalized the algorithm, the program was executed continuously for five days without failure before it was manually stopped. At this point, it had reached 230,500 tracks and 23,050 geometries, which were determined to be sufficient for comparison to a parallel version. The results of the test are shown in Figure 3.2. This plot displays the ever increasing average execution times for the sequential version of the algorithm. As the number of tracks and geometries increase, the execution time also increases. The slightly quadratic shaped curve is expected since the number of geometries and the number of tracks are both increasing.
It was soon discovered after the test run of the sequential version that there was an oversimplification in the program that skewed the execution time results shown in Figure 3.2. The oversimplification occurred when calculating the dot products in Equation 2.2. Since each side of a geometry is parallel to a coordinate axis, two of the three vector coordinates of the outward normals will always have a value of zero. The variables representing the zero value vector coordinates were not written in to the program since the mathematical result of the dot product would be the same as if they were
implemented into the code. This oversimplification underestimated the dot product execution time for the general case in which the geometry is not parallel to the coordinate axis and was determined to be not sufficiently representative of this general case.

To correct this oversimplification, the two excluded zero value vector coordinates for the outward normals were included in the program to more accurately represent the execution time needed to calculate a dot product in the general case. After fixing the sequential version, it was tested again using the same technique described in Section 3.3. The result of this test is shown in Figure 3.3 along with the original sequential version result. It can be seen that these additional program instructions significantly increase the execution time for the sequential version. This increase is large enough to now consider the updated version of the sequential program to be the new baseline to which the parallel version will be compared.
It should be noted that the maximum number of tracks and geometries executed in the updated sequential version was 200,000 and 20,000, respectively (compare to the original sequential version that was stopped at 230,500 tracks and 23,050 geometries). The program was manually ended after 6 days of continuous execution without failure. Although the updated version executed an entire day longer than the original, the final number of tracks and geometries at end time were significantly less than those of the original. Nevertheless, the numbers were still deemed sufficient for comparison to results of the parallel version.
4. PARALLEL PROCESSING

4.1 Graphical Processing Unit (GPU)

There are several options in which parallel processing can be achieved. Such options are massive supercomputers or multiple workstations working collaboratively. These options are expensive and require resources that are unavailable to the average programmer. Graphical Processing Units, or GPUs, are an alternative that are relatively inexpensive and accessible to most programmers. These processing units are standard components of laptops and desktops. That is because GPUs are housed on graphic cards which are standard hardware attachments for PCs, laptops, and desktops. GPUs were originally designed for image rendering in video games and other applications. Recently, programmers have been realizing the benefits of GPUs for parallelization of scientific computations.

Finding violations for multiple tracks and geometries is a natural fit for parallel processing. Rather than computing a single violation at a time as in the sequential CPU version, multiple violations can be found in parallel using GPUs. At each step of the inherently sequential process of finding violations, data for multiple tracks and multiple geometries can be computed in parallel. For example, Equation 2.1 can be used to find
the directional vectors for all the tracks at once. However, not all scientific computational algorithms are suitable for parallel processing, particularly if dependencies exist. For example, calculating Equation 2.1 concurrently with Equation 2.2 would pose challenges since some of the inputs for Equation 2.2 are dependent on the results of Equation 2.1.

4.2 GPU Architecture

The architecture of a GPU enables efficient parallel processing. A GPU contains a series of multiprocessors, called Streaming Multiprocessors (SMs), each containing a number of processors called cores. Figure 4.1 shows a diagram of a generic GPU architecture and illustrates how the SMs and processor cores are related. The number of SMs and cores vary for each GPU. The GPUs used in this work are the Quadro FX 5800 and the Tesla C1060, which both have 30 SMs and 240 cores [5][6]. Each core contains a number of threads where each thread controls a single calculation [1]. With each thread handling a single calculation, there could potentially be tens of thousands of instructions executed in parallel.
Fig. 4.1 GPU architecture

4.3 CUDA Programming Overview

All GPU architectures are not the same. Intel makes GPUs with architectures different from those of NVIDIA. Software Development Kits (SDKs) were developed by the various manufacturers for ease in writing programs that utilize their GPUs. An example of a SDK used to communicate with a GPU is the Open Graphics Library (OpenGL), which is a universal toolkit that can be used to program GPUs made by different manufactures [7]. There are, however, other toolkits that are restricted to
specific GPU manufactures. One such toolkit is the NVIDIA Compute Unified Device Architecture (CUDA) [2]. CUDA was specifically developed by NVIDIA to enable access to NVIDIA GPUs, which makes it an ideal starting point for controlling NVIDIA GPUs. A NVIDIA GPU that is compatible with CUDA is called a CUDA device.

When accessing a CUDA device, three basic steps are required [1][8]. The first step is to transfer data from the host (CPU) to the device (GPU). The next step is to execute a kernel. A kernel is a program function that runs on a GPU. The last step is retrieving the relevant data from the device (GPU) back to the host (CPU). Figure 4.2 show a block diagram of these basic steps with the relevant portions highlighted. For CUDA devices, the GPU cannot access memory outside its own device memory [1]. Thus, the transferring of data is required when a programmer decides to use a GPU to manipulate data [1]. The transferring of data also creates program execution overhead.

![Fig. 4.2 Required CUDA programming steps](image-url)
When programming using CUDA, it is necessary to understand how the threads of a GPU are organized. In a GPU, a group of threads is called a block and a group of blocks is called a grid, as shown in Figure 4.3 [1]. NVIDIA uses three-dimensional coordinates to relay the size and shape of the grids and blocks. The notation NVIDIA uses to define both grids and blocks is \( <<X, Y, Z>> \), where \( X \) is the number of columns, \( Y \) is the number of rows, and \( Z \) is the number of frames [1]. The product of the \( X \), \( Y \), and \( Z \) dimensions of a block yields the number of threads contained in each block. The product of the \( X \), \( Y \), and \( Z \) dimensions of a grid yields the number of blocks contained in that grid. For example, a grid with dimensions \( <<3, 2, 1>> \) and block dimensions \( <<4, 5, 3>> \), contains 6 blocks and 60 threads per block, for a total of 360 threads. Figure 4.3 shows a visual representation of how grids, blocks, and thread are organized in the CUDA architecture. In general, when developing kernels to run on the GPU, the programmer must define the grid and block dimensions. However, the libraries provided by the CUDA toolkit, such as NPP, cuBLAS, and Thrust, automatically set the grid and block dimensions for the programmer [9]. These libraries will be discussed in more detail in Chapter 5.
The values for the grid and block dimensions are limited, as are the number of grids and blocks themselves. The limitations vary from GPU to GPU and are defined by the GPU architecture designed by the manufacturer. The version, or compute capability, of the CUDA device determines the grid and block limitations for a particular GPU [1]. Since there are hundreds of different types of GPUs, only the limitations of the GPUs used in this work will be discussed. As a reminder, the GPUs used in this work are the Quadro FX 5800 and the Tesla C1060. Both of these GPUs have a compute capability of
1.3; thus they have the same specifications. Table 1 shows the various limitations for a GPU of compute capacity 1.3 [1].

Table 4.1
Grid and block limitations for CUDA devices with compute capability of 1.3

<table>
<thead>
<tr>
<th>Specifications</th>
<th>Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of dimensions for grid</td>
<td>2</td>
</tr>
<tr>
<td>Maximum X-dimension of grid</td>
<td>65535</td>
</tr>
<tr>
<td>Maximum Y-dimension of grid</td>
<td>65535</td>
</tr>
<tr>
<td>Maximum Z-dimension of grid</td>
<td>1</td>
</tr>
<tr>
<td>Maximum number of dimensions for block</td>
<td>3</td>
</tr>
<tr>
<td>Maximum X-dimension of block</td>
<td>512</td>
</tr>
<tr>
<td>Maximum Y-dimension of block</td>
<td>512</td>
</tr>
<tr>
<td>Maximum Z-dimension of block</td>
<td>64</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
</tr>
<tr>
<td>Maximum number of resident blocks per multi-processor</td>
<td>8</td>
</tr>
<tr>
<td>Maximum number of resident threads per multi-processor</td>
<td>1024</td>
</tr>
</tbody>
</table>

All of these specifications cannot be set to their maximum limits at the same time, since some of these limitations depend on the values of other specifications listed in Table 1. For example, a programmer cannot define the number of blocks in a SM as 8 and concurrently specify the number of threads per block to be 512. This would result in 4,096 threads per SM, which exceeds the maximum limit of 1,024 threads per SM. A programmer can fix this problem by either specifying 2 blocks with 512 threads per block,
or 8 blocks with 128 threads per block. Since there is not a unique combination, determining the optimum mix can be challenging to the novice CUDA programmer.

A critical factor in efficiently defining grid and block dimensions involves the warp size. Once the block dimensions have been specified, the GPU divides the threads within a block into groups of 32 called warps [1]. A warp is a group of threads that execute a specified instruction. Virtually all CUDA devices have a warp size of 32 [1]. To fully utilize the capability of a GPU, all threads within a warp must be active, meaning they are executing the same instructions [1]. When the number of threads in a block is not a multiple of 32, some of the threads will be inactive. Thus, performance of the GPU is most efficient when the number of threads within a block is a multiple of 32.

Figure 4.4 shows two examples of how warps are organized in a CUDA device depending on the specified block and grid dimensions. In Figure 4.4 (a), the block dimension is <<8, 8, 1>>, and the grid dimension is <<2, 1, 1>>. The total number of threads in this case is 128 with each of the 2 blocks containing 64 threads. Since the number of threads per block, 64, is a multiple of 32, all the threads within the block are utilized for computation. The thread to warp efficiency is 100%. In contrast, the configuration of Figure 4.4 (b) utilizes a block dimension of <<8, 5, 1>> and a grid dimension of <<2, 2, 1>>. There are 160 total threads specified with each of the 4 blocks containing 40 threads. In this case, since the 40 threads assigned to each block are not a multiple of 32, the first 32 threads of each block are assigned to a warp. The remaining 8 threads in each block are assigned to a second warp of size 32. In this second warp, only 8 of the threads are active; the remaining 24 threads are inactive. In Figure 4.4 (b), these inactive threads within each block are depicted in red.
Fig. 4.4 Example of warp distribution

The result of the situation in Figure 4.4 (b) is a total of 256 threads incorporated into eight warps, but only 160 of those threads are active. The other 96 threads are inactive yielding a thread to warp efficiency of 62.5%. Because the number of threads per block is not a multiple of 32 in Figure 4.4 (b), about 37.5% of the GPU’s resources are idle. This is highly inefficient and demonstrates the importance of having the number of threads within each block be a multiple of 32.

NVIDIA provides various CUDA libraries that help a programmer efficiently write code for a CUDA device [9]. CUDA libraries provide kernels that automatically define the block and grid dimensions in an efficient manner in order to fully utilize the resources of the GPU. However, Chapter 5 will demonstrate that this is not always the case.
5. CUDA PERFORMANCE

5.1 CUDA Libraries

For CUDA v5.5, three main libraries are included in the free download of the CUDA SDK [3]. These are NPP (NVIDIA Performance Primitives), cuBLAS (NVIDIA CUDA Basic Linear Algebra Subroutines), and Thrust. These libraries provide efficiently constructed kernels (functions) that make it easy for the programmer to seamlessly incorporate CUDA commands without knowledge of the underlying architecture. To gain understanding of the functionality of these libraries, a series of test programs was written and executed. The kernels tested included only those that were used or could be used in the final parallel version of the conflict detection algorithm. Kernels chosen for testing included simple arithmetic operations and other kernels similar to the operations performed in the sequential version of the conflict detection algorithm.

All test programs were executed on the HP Z800 Workstation located in Room 324 of the Engineering, Technology, and Computer Science Building on the IPFW campus. All test programs were compiled in the Linux terminal using the command NVCC (NVIDIA CUDA Compiler) after saving in the CUDA file .cu format. The text editor used to write the test programs was gedit.
5.1.1 NPP (NVIDIA Performance Primitives)

A vast majority of kernels used in the final parallel version of the conflict detection algorithm use operations provided by the NPP library. That is because the kernels provided by the NPP library most represented the basic arithmetic operations needed to find violations in the conflict detection algorithm [10]. Excluding the data transfers to and from the GPU, a single line of code is all that is needed to execute an NPP kernel. This is far simpler than Thrust which requires a template data structure for most of its kernel calls [11]. The NPP library also executes element-by-element array calculations which are a preferred method of calculating results for the conflict detection algorithm. The cuBLAS library is structured more for vector and matrix multiplication; thus most of its kernels may not be as suitable to perform basic operations [12].

Seventeen separate test programs were written to measure the performance of specific NPP kernels. The 17 kernels were chosen based on how close they are to the operations used in the sequential version and how likely they are to be used in the final parallel version of the conflict detection algorithm. The NPP kernels used for testing are listed in Table 5.1 along with a mathematical description of each. All kernels used in the NPP test programs execute element-by-element calculations of the input arrays. A detailed description of the inputs for the NPP kernels can be found in the NPP User Guide [10].
### Table 5.1
List of NPP kernels tested

<table>
<thead>
<tr>
<th>NPP Kernel</th>
<th>Mathematical Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nppiCompare_32f_C1R</td>
<td>If A &quot;$\ll\text{Comparison}\gg&quot; B, Then C = 255; Else C = 0;</td>
</tr>
<tr>
<td>nppiCompareC_32f_C1R</td>
<td>If A &quot;$\ll\text{Comparison}\gg&quot; \text{Constant}, Then C = 255; Else C = 0;</td>
</tr>
<tr>
<td>nppsAdd_8u_ISfs</td>
<td>B = A + B</td>
</tr>
<tr>
<td>nppsAdd_8u_Sfs</td>
<td>C = A + B</td>
</tr>
<tr>
<td>nppsAdd_32f_I</td>
<td>B = A + B</td>
</tr>
<tr>
<td>nppsAddC_32f_I</td>
<td>A = A + \text{Constant}</td>
</tr>
<tr>
<td>nppsCopy_32f</td>
<td>B = A</td>
</tr>
<tr>
<td>nppsDiv_32f</td>
<td>C = \frac{A}{B}</td>
</tr>
<tr>
<td>nppsDiv_32f_I</td>
<td>A = \frac{A}{B}</td>
</tr>
<tr>
<td>nppsDivC_32f_I</td>
<td>A = \frac{A}{\text{Constant}}</td>
</tr>
<tr>
<td>nppsMul_32f_I</td>
<td>B = A \cdot B</td>
</tr>
<tr>
<td>nppsMulC_32f_I</td>
<td>A = A \cdot \text{Constant}</td>
</tr>
<tr>
<td>nppsSqr_32f_I</td>
<td>A = A^2</td>
</tr>
<tr>
<td>nppsSqrt_32f_I</td>
<td>A = \sqrt{A}</td>
</tr>
<tr>
<td>nppsSub_32f</td>
<td>C = A – B</td>
</tr>
<tr>
<td>nppsSub_32f_I</td>
<td>A = A – B</td>
</tr>
<tr>
<td>nppsSubC_32f_I</td>
<td>A = A – \text{Constant}</td>
</tr>
</tbody>
</table>
The NPP library uses a standard notation for naming its kernels [10]. Information on the data type to which the function applies appears in the suffix. For example, the kernel \textit{nppsSub\_32f} requires 32 bit floats (32f) for its inputs, whereas the kernel \textit{nppsAdd\_8u\_Sfs} takes 8 bit unsigned characters (8u) as inputs. Another piece of information provided by the kernel name is whether or not one of the inputs is a constant. This is designated by a capital letter \textit{C}, as in \textit{nppsMulC\_32f\_I}, which requires a constant for one of its inputs. In contrast, the kernel \textit{nppsMul\_32f\_I} does not. A third piece of information that can be determined from the NPP function name is the destination of the output. The capital letter \textit{I} in the kernel name stands for in-place, meaning the input variable is also the destination variable. Function names without an \textit{I} require an additional input variable to specify the destination of the operation output. The NPP prefixes \textit{npps} and \textit{nppi} indicate functions for signal processing or image processing, respectively. Since the research reported here is not designed for signal or image processing, these prefixes are not relevant here. In summary, the name of most NPP kernels supplies the information required for its execution. For example, the kernel \textit{nppsSubC\_32f\_I} is a signal processing function (\textit{npps}) that subtracts (\textit{Sub}) a constant (\textit{C}) from a 32 bit float (32f) and stores the output of that operation back into the input variable (\textit{I}).

Each of the 17 NPP test programs follow the same overall structure with each program divided into three parts. The first part of the program executes the CPU version of the NPP kernel, which serves as a baseline for comparison. The second part of the test programs executes the NPP kernel on the Quadro FX 5800. The last part of the test
programs executes the NPP kernel on the Tesla C1060. Figure 5.1 shows a generic flowchart that each NPP test program follows.

Fig. 5.1 Flowchart for NPP test programs

Each test program uses its own randomly generated data set since each kernel requires a different number of variables and different data types. However, within each program, the same random data set is used for all three portions. All data sets used in all the NPP test programs are \( N \times N \) matrices, unless the input requires a constant. All three
portions of a NPP test program reside within an infinite while loop. After each iteration of the infinite while loop, the number of rows and columns of the input matrices increase by 500. The program continuously executes until either the GPU or CPU runs out of memory. The data transfers to and from the GPUs are included in the execution times since CUDA requires the data to be uploaded and downloaded to and from the GPU.

For each data set in each NPP test program, the percent difference between the execution times of the GPUs and the CPU were calculated. The average of these percent differences for each NPP test program was then determined for both the Quadro FX 5800 and the Tesla C1060. The percent differences for each of the NPP test program are listed in Table 5.2. A positive percent difference means the GPU executed the operation on average faster than the CPU and by what percent faster. A negative percentage means the GPU executed the operation slower and by what percent when compared to the CPU execution times.
Table 5.2
Execution time performance results for NPP kernels

<table>
<thead>
<tr>
<th>NPP Kernel</th>
<th>Quadro FX 5800 Percent Difference</th>
<th>Tesla C1060 Percent Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>nppiCompare_32f_C1R</td>
<td>67.57%</td>
<td>67.18%</td>
</tr>
<tr>
<td>nppiCompareC_32f_C1R</td>
<td>82.42%</td>
<td>82.78%</td>
</tr>
<tr>
<td>nppsAdd_8u_ISfs</td>
<td>78.93%</td>
<td>82.70%</td>
</tr>
<tr>
<td>nppsAdd_8u_Sfs</td>
<td>79.38%</td>
<td>76.53%</td>
</tr>
<tr>
<td>nppsAdd_32f_I</td>
<td>30.45%</td>
<td>34.42%</td>
</tr>
<tr>
<td>nppsAddC_32f_I</td>
<td>42.69%</td>
<td>45.42%</td>
</tr>
<tr>
<td>nppsCopy_32f</td>
<td>36.55%</td>
<td>39.17%</td>
</tr>
<tr>
<td>nppsDiv_32f</td>
<td>34.42%</td>
<td>31.23%</td>
</tr>
<tr>
<td>nppsDiv_32f_I</td>
<td>35.57%</td>
<td>38.16%</td>
</tr>
<tr>
<td>nppsDivC_32f_I</td>
<td>71.25%</td>
<td>72.81%</td>
</tr>
<tr>
<td>nppsMul_32f_I</td>
<td>28.74%</td>
<td>37.27%</td>
</tr>
<tr>
<td>nppsMulC_32f_I</td>
<td>45.91%</td>
<td>50.07%</td>
</tr>
<tr>
<td>nppsSqr_32f_I</td>
<td>53.91%</td>
<td>54.24%</td>
</tr>
<tr>
<td>nppsSqrt_32f_I</td>
<td>68.65%</td>
<td>72.74%</td>
</tr>
<tr>
<td>nppsSub_32f</td>
<td>32.39%</td>
<td>31.03%</td>
</tr>
<tr>
<td>nppsSub_32f_I</td>
<td>32.44%</td>
<td>37.43%</td>
</tr>
<tr>
<td>nppsSubC_32f_I</td>
<td>43.90%</td>
<td>47.64%</td>
</tr>
</tbody>
</table>
The results in Figure 5.2 show an overall improvement in execution time for all NPP kernels tested. The percent improvement varies from kernel to kernel with the kernel *nppiCompareC_32f_C1R* showing the most improvement. Also, the improvements vary for each GPU. Table 5.2 shows the Tesla C1060 having higher improvement percentages than the Quadro FX 5800. This makes sense since the Tesla C1060 was built specifically for scientific computations and not graphics rendering.

In addition to comparing the execution times, the numerical results of the calculations were also compared. This was to determine if there were any discrepancies in accuracy between the two CUDA devices and the CPU. A simple inequality was used to compare the computational results of the CPU to those of both of the GPUs. Using the CPU results as the “ground truth”, the total number of discrepancies was found. The total number of discrepancies was then divided by the total number of calculations. This yields the discrepancy percentage between the GPU calculations and the CPU calculations. This percentage describes the probability that a discrepancy may occur, but it does not describe the margin of error between a GPU calculation and a CPU calculation. Thus, an average margin of error was calculated by summing the values of each discrepancy and dividing that number by the total number of discrepancies. Both the resulting discrepancy percentages and the average margin of errors for all NPP kernels tested are listed in Table 5.3.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Tesla C1060 Improv.</th>
<th>Quadro FX 5800 Improv.</th>
</tr>
</thead>
<tbody>
<tr>
<td>nppiCompareC_32f_C1R</td>
<td>10%</td>
<td>5%</td>
</tr>
<tr>
<td>nppiCompareC_16f_C1R</td>
<td>7%</td>
<td>3%</td>
</tr>
<tr>
<td>nppiCompareC_8f_C1R</td>
<td>4%</td>
<td>2%</td>
</tr>
</tbody>
</table>

Table 5.2: Comparison of Improvement Percentages for NPP Kernels
Table 5.3
NPP result discrepancy percentages and average margin of errors

<table>
<thead>
<tr>
<th>NPP Kernel</th>
<th>Discrepancy Percentage</th>
<th>Average Margin of Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>npipCompare_32f_C1R</td>
<td>0%</td>
<td>N/A</td>
</tr>
<tr>
<td>npipCompareC_32f_C1R</td>
<td>0%</td>
<td>N/A</td>
</tr>
<tr>
<td>nppsAdd_8u_ISfs</td>
<td>0%</td>
<td>N/A</td>
</tr>
<tr>
<td>nppsAdd_8u_Sfs</td>
<td>0%</td>
<td>N/A</td>
</tr>
<tr>
<td>nppsAdd_32f_I</td>
<td>0%</td>
<td>N/A</td>
</tr>
<tr>
<td>nppsAddC_32f_I</td>
<td>0%</td>
<td>N/A</td>
</tr>
<tr>
<td>nppsCopy_32f</td>
<td>0%</td>
<td>N/A</td>
</tr>
<tr>
<td>nppsDiv_32f</td>
<td>28.36%</td>
<td>10^{-6}</td>
</tr>
<tr>
<td>nppsDiv_32f_I</td>
<td>28.36%</td>
<td>1.02 x 10^{-6}</td>
</tr>
<tr>
<td>nppsDivC_32f_I</td>
<td>16.13%</td>
<td>&lt; 10^{-6}</td>
</tr>
<tr>
<td>nppsMul_32f_I</td>
<td>0%</td>
<td>N/A</td>
</tr>
<tr>
<td>nppsMulC_32f_I</td>
<td>0%</td>
<td>N/A</td>
</tr>
<tr>
<td>nppsSqr_32f_I</td>
<td>0%</td>
<td>N/A</td>
</tr>
<tr>
<td>nppsSqrt_32f_I</td>
<td>37.46%</td>
<td>2.7 x 10^{-7}</td>
</tr>
<tr>
<td>nppsSub_32f</td>
<td>0%</td>
<td>N/A</td>
</tr>
<tr>
<td>nppsSub_32f_I</td>
<td>0%</td>
<td>N/A</td>
</tr>
<tr>
<td>nppsSubC_32f_I</td>
<td>0%</td>
<td>N/A</td>
</tr>
</tbody>
</table>
According to the results depicted in Table 5.3, there are four kernels tested that had discrepancies in their results when compared to the CPU results. Though the discrepancy percentages are relatively high, the average margin of error for each case is miniscule. At this point, it is unclear whether these small errors will have any effect on the overall calculated results of the conflict detection algorithm, but considering how small the errors are, one could assume that they will have little to no effect at all. If by chance the calculations in the algorithm are affected by these miniscule errors, the results can then be translated as a rounding error.

The results in this section show that the NPP library is not a perfect replacement for the CPU operations due to the calculation errors that occur. However, the tradeoff of these very small errors is a relatively large reduction in execution time. This low risk high reward scenario makes the NPP library a viable option for the parallel implementation of the conflict detection algorithm.

5.1.2 cuBLAS (NVIDIA CUDA Basic Linear Algebra Subroutines)

The cuBLAS library is a vector and matrix driven library [12]. One kernel in particular is the cublasSgemm kernel call. The cublasSgemm kernel performs the matrix multiplication and addition operations described in Equation 5.1, where $\alpha$ and $\beta$ are constants and $A$, $B$, and $C$ are matrices. It is assumed that the setup of the rows and columns of each matrix is mathematically consistent with the rules of matrix multiplication and matrix addition. The matrices are uploaded to the GPU beforehand using the cuBLAS function cublasSetMatrix, specifying the number of rows and columns
in each matrix. The number of rows and columns must be specified in the \textit{cublasSgemm} kernel as well. The resulting matrix $C$ can be retrieved from the GPU using the cuBLAS function \textit{cublasGetMatrix}.

\[
C = \alpha AB + \beta C
\] (5.1)

A test program was developed to compare execution times of the \textit{cublasSgemm} kernel to those of equivalent NPP and CPU operations. The program was set up to perform the CPU operations, followed by the \textit{cublasSgemm} operations, and then the NPP operations, using the same data, in sequential order. For the CPU calculations, three matrices were generated and populated with random values. For this test, the constants $\alpha$ and $\beta$ from Equation 5.1 were both equal to 1. Matrix $A$ was of size $1 \times 3$, $B$ was $3 \times \text{columns}$, and $C$ was $1 \times \text{columns}$. The variable \textit{columns} was varied from 1,000,000 to 100,000,000 in increments of 1,000,000 and execution times to compute the matrix operations of Equation 5.1 were recorded for each value of \textit{columns}. The execution times along with the resulting computed values of the CPU portion of the test program were used as the baseline standard to which the cuBLAS and NPP kernel results were compared.

The \textit{cublasSgemm} portion of the test program utilized the same data and the same three matrices but used the cuBLAS kernel \textit{cublasSgemm} to perform the operations of Equation 5.1. For the NPP version of the test code, the NPP kernels \textit{nppsMulC_32f_I} and \textit{nppsAdd_32f_I} were used. The \textit{nppsMulC_32f_I} kernel performs an in-place multiply of a constant by a matrix, and the \textit{nppsAdd_32f_I} kernel performs an in-place addition of two matrices. Matrix $A$ was defined as a $1 \times 3$ constant matrix that was not uploaded to the GPU, but remained on the CPU since its elements were used as constants in the NPP
kernels. Matrix $B$ was represented as three $1 \times \text{columns}$ matrices, representing the first, second, and third rows of the $3 \times \text{columns}$ $B$ matrix used in the CPU and \textit{cublasSgemm} versions of the test code. Matrix $C$ was defined as a $1 \times \text{columns}$ matrix. Each of the three elements of matrix $A$ was multiplied by the corresponding three $1 \times \text{column}$ matrices that compose matrix $B$, in three sequential operations. The next two operations using \textit{nppsAdd\_32f\_I} were used to sum the resulting three $1 \times \text{columns}$ matrices into a single $1 \times \text{columns}$ matrix. This $1 \times \text{columns}$ matrix was equivalent to the results of the matrix multiplication of $A$ and $B$ in the \textit{cublasSgemm} kernel. Next, two more \textit{nppsMulC\_32f\_I} kernels were called to multiply the constants $\alpha$ and $\beta$ to the resulting $1 \times \text{columns}$ matrix from the previous step and matrix $C$, respectively. The final step was to add those results together yielding a final $1 \times \text{columns}$ matrix consistent with the CPU and \textit{cublasSgemm} results.

The Tesla C0160 was used for the test program and the execution times are shown in Figure 5.2. Surprisingly, the cuBLAS kernel was the slowest of all three methods, even slower than the CPU version of Equation 5.1. The average percent difference in execution time for the \textit{cublasSgemm} kernel was found to be -17.71\% whereas the average percent difference for the NPP equivalent was found to be 47.04\%. This means the \textit{cublasSgemm} kernel on average executed about 17\% slower than the CPU equivalent and the NPP executed about 47\% faster than the CPU equivalent.
Fig. 5.2 Execution times for \textit{cublasSgemm}

The program analyzer NVIDIA Compute Visual Profiler, which comes free with the download of the CUDA SDK, was used to help determine why the cuBLAS kernel had such high runtimes in these tests [3]. The NVIDIA Compute Visual Profiler is a profiling tool that can be used to measure the performance of programs on an NVIDIA GPU platform. It provides information on kernel occupancy, instruction throughput, and memory access. It displays statistics on the amount of time used to transfer variables to and from the GPU compared to the time spent in kernels, as well as a timeline of the execution of a program.
Two separate programs were developed for analysis on the NVIDIA Visual Profiler. The first program only contains the minimum required code to allow the cuBLAS kernel, \texttt{cublasSgemm}, to function properly. The second program only contains the require coding to allow the NPP equivalent of the \texttt{cublasSgemm} kernel to function properly. These two programs were separately evaluated using the NVIDIA Visual Profiler. Figure 5.3 is a screenshot of the NVIDIA Visual Profiler results for the \texttt{cublasSgemm} test program. The particular segment shown in Figure 5.3 displays the timeline of the GPU operations when the variable \textit{column} is equal to 80,000,000.

The left side of Figure 5.3 contains the session frame which describes what each line of the results represents. The right side of Figure 5.3 contains the workspace frame which shows the timeline of the different operations being performed on the GPU in regards to the \texttt{cublasSgemm} test program. Within the session frame, the 8th line from the top titled “MemCpy (HtoD)” corresponds to the amount of time used to transfer data from the CPU (host) to the GPU (device). The corresponding execution time for this operation is displayed in bar form horizontally along the 8th line within the workspace frame. Similarly, the 9th line from the top within the session frame is titled “MemCpy (DtoH)” corresponds to the time used to transfer data from the GPU to the CPU. Also within the session frame, the 10th line from the top titled “Compute” corresponds to the execution time used to perform computations on the GPU. The “Compute” line contains all the kernels being executed on the GPU. The Visual Profiler allows for different kernels to be displayed on their own lines which are sub-lines of the “Compute” line. If there are multiple different kernels being executed on a program, each kernel would have its own line located directly below the “Compute” line. Since only the \texttt{cublasSgemm}
kernel is being executed in this test program, there is only one sub-line (line 11) within “Compute”.

**Fig. 5.3 Visual Profiler results for cublasSgemm kernel**

In Figure 5.3, the execution time of the *cublasSgemm* kernel when the variable *columns* is equal to 80,000,000 is about 0.144 seconds. The NPP equivalent was also analyzed using the NVIDIA Visual Profiler using the same exact data and the same value for *columns* (80,000,000). The result of this analysis is shown in Figure 5.4. Note that there are two sub-lines within “Compute”. Lines 11 and 12 represent the timelines for the kernels *nppsMulC_32f_I* and *nppsAdd_32f_I*, respectively. Line 10 contains the timeline of both kernels. The kernel execution time for the NPP equivalent encompasses all eight kernel calls; five kernel calls for *nppsMulC_32f_I* and three kernel calls for *nppsAdd_32f_I*. For the particular case depicted in Figure 5.4, the kernel execution time for the NPP equivalent equals about 0.019 seconds. To explain why eight kernel calls
execute the same data set much faster than one kernel call, an understanding of the kernel property occupancy is beneficial.

Fig. 5.4 Visual Profiler results for NPP equivalent of cublasSgemm

Occupancy is a property used to describe the number of active warps divided by the total available warps on a multiprocessor [13]. The occupancy of a kernel is directly proportional to the number of active warps in a GPU. The more warps that are active, the higher the occupancy. It is common to associate higher performance with higher occupancy since a higher occupancy means more threads are executing in parallel, but this is not always the case as described by Vasily Volkov in his presentation “Better Performance at Lower Occupancy” [14].

Occupancy is found using Equation 5.2, where $W_{active}$ is the number of active warps in a multiprocessor and $W_{total}$ is the total number of available warps in a multiprocessor [13]. For the GPUs used in this work the total number of resident warps per multiprocessor is 32 [1].

$$Occupancy = \frac{W_{active}}{W_{total}}$$  \hspace{1cm} (5.2)
However, the number of active warps, $W_{active}$, is determined by the number of active blocks in a multiprocessor. The number of active warps is found by multiplying the number of active blocks, $B_{active}$, by the number of warps per block, $W_{block}$, as described in Equation 5.3.

$$W_{active} = B_{active} \cdot W_{block} \quad (5.3)$$

The number of warps per block, $W_{block}$, is found by dividing the number of threads in each block by the number of threads required for a warp, $T_{warp}$, and rounding up to the nearest integer. For GPUs used in this work, the number of threads in a warp is 32 [1]. The number of threads in each block is found by calculating the product of the dimensions of the block. Equation 5.4 describes the calculations needed to find the number of warps per block, where $xDim$, $yDim$, and $zDim$ are the dimensions of a block. Note that the notation $\text{ceil}(x, y)$ means that $x$ is rounded up to the nearest integer multiple of $y$.

$$W_{block} = \text{ceil} \left( \frac{xDim \cdot yDim \cdot zDim}{T_{warp}}, 1 \right) \quad (5.4)$$

The most important variable when determining occupancy is the number of active blocks, $B_{active}$. The number of active blocks on a multiprocessor is determined by one of three limiting factors. These limiting factors include block dimensions, GPU register usage, and GPU shared memory usage [13][15]. Each of these methods calculates a different theoretical number of active blocks in a multiprocessor. The smallest result of the three methods is the limiting factor of occupancy. Thus, the smallest of the three calculated values for $B_{active}$ will be used in Equation 5.3 when determining the occupancy. Note that for each limiter, $B_{active}$ cannot exceed the maximum number of blocks per
multiprocessor since that maximum is a hardware limitation set by the manufactures of the GPU.

The first of three active block limiters involves the use of block dimensions [13]. The theoretical number of active blocks based on block dimensions, $B_{active,1}$, is determined by Equation 5.5. In this equation, $T_{MP}$ is the total number of available threads per multiprocessor and $T_{block}$ is the number of threads per block. The number of threads per block is found by calculating the numerator in Equation 5.4. The variable $MaxBlocks$ represents the maximum number of resident blocks per multiprocessor for a particular GPU, thus is the maximum allowable value for $B_{active,1}$. Recall from Table 4.1 that the GPUs in this work have a maximum of 1,024 resident threads per multiprocessor and that the maximum number of resident blocks per multiprocessor is 8 [1].

$$B_{active,1} = \min(\text{MaxBlocks}, \, \text{ceil}(\frac{T_{MP}}{T_{block}}, \, 1)) \quad (5.5)$$

The second active block limiter is based on the register usage of a kernel [13][15]. This method for finding the theoretical number of active blocks consists of two parts. The first part requires finding the number of registers being used per block, $R_{block}$, in Equation 5.6. The second part involves taking the result of Equation 5.6 and using it in Equation 5.7. In Equation 5.6, $W_{block}$ and $T_{warp}$ are the same variables used in Equation 5.4. The variable $R_{thread}$ is the number of registers used per thread. $G_{warp}$ and $G_{thread}$ are the allocation granularity of the warps and threads, respectively. Allocation granularity means that the resources allocated for warps are in multiples $G_{warp}$, and the resources allocated for threads are in multiples of $G_{thread}$, respectively. For example, CUDA devices of compute capability 1.3 have a warp allocation granularity of 2 and a thread
allocation granularity of 512 [15]. This means GPU resources are allocated for 2 warps at a time and for 512 threads at a time. Even if there are only 5 active warps, resources for 6 warps would be allocated. The allocation of resources for that extra inactive warp would be wasted causing lower performance. The same can be said for thread resource allocation. If 513 threads are active, then the GPU would allocate enough resources to apply to 1,024 threads. Resources allocated for 511 threads would be wasted on inactive threads, thus lowering performance.

\[
R_{\text{block}} = \text{ceil}(\text{ceil}(W_{\text{block}}, G_{\text{warp}}) \cdot T_{\text{warp}} \cdot R_{\text{thread}}, G_{\text{thread}})
\]  \hspace{1cm} (5.6)

Having calculated the number of registers per block, it can be used in a simple division to solve the second limiter for the number of active blocks on a multiprocessor, \(B_{\text{active},2}\). Equation 5.7 is used to find that second limiter, where \(R_{MP}\) is the number of registers available on each multiprocessor and \(R_{\text{block}}\) is the result of Equation 5.6. For GPUs used in this work the number of registers per multiprocessors is 16,384 [1].

\[
B_{\text{active},2} = \min \left( \text{MaxBlocks}, \text{floor} \left( \frac{R_{MP}}{R_{\text{block}}}, 1 \right) \right)
\]  \hspace{1cm} (5.7)

The final of the three limiters depends on the amount of shared memory being used by the kernel [13][15]. Each multiprocessor has dedicated memory called shared memory that allows threads to quickly access memory that other threads in the same block are using [1]. Similar to the register limiter, this limiter also involves two steps. First, Equation 5.8 is used to calculate the amount of shared memory per block, \(S_{\text{block}}\), by rounding up the initially required shared memory per block, \(S_{\text{int}}\), to the nearest integer multiple of \(G_{\text{share}}\). \(G_{\text{share}}\) is the shared memory allocation granularity of a GPU. For the
Quadro FX 5800 and the Tesla C1060, $G_{\text{share}}$ equals 512 Bytes [15]. This means the shared memory is allocated in 512 Byte segments.

\[
S_{\text{block}} = \text{ceil}(S_{\text{int}}, G_{\text{share}})
\]  

(5.8)

Next, the result of Equation 5.8 is inserted into Equation 5.9 to find the theoretical number of active blocks in a multiprocessor based on the usage of shared memory, $B_{\text{active},3}$. In Equation 5.9, $S_{\text{MP}}$ is the total amount of shared memory per multiprocessor. The GPUs used in this work have 16 kilobytes (KB) per multiprocessor [1].

\[
B_{\text{active},3} = \min\left(\text{MaxBlocks}, \text{floor}\left(\frac{S_{\text{MP}}}{S_{\text{block}}}, 1\right)\right)
\]  

(5.9)

Once $B_{\text{active},1}$, $B_{\text{active},2}$, and $B_{\text{active},3}$ have been calculated, the smallest of those three results will be inserted into Equation 5.3. The result of that is then used for Equation 5.2 which results in occupancy of the kernel.

By clicking on a kernel within the workspace frame of a NVIDIA Visual Profiler result, information about that particular kernel is displayed in a separate properties frame. The properties frame displays information such as grid dimensions, block dimensions, and occupancy. Figure 5.5 shows the properties frame for the `cublasSgemm` kernel depicted in Figure 5.3.
In Figure 5.5 occupancy of the \textit{cublasSgemm} kernel is only 31%. This means only 31% of the warps available on the GPU are active when executing the \textit{cublasSgemm} kernel. This is highly inefficient and could explain why the \textit{cublasSgemm} kernel is executing so slowly compared to the CPU and NPP versions. By following the guidelines established for Equations 5.2 – 5.9 and taking the information gathered in the properties frame, it was found that the allocation of the registers per block was the reason for the 31.2% occupancy. This is because there are 3,072 registers per block (Equation 5.6). In Equation 5.6, the variables $G_{\text{warp}}$, $T_{\text{warp}}$, and $G_{\text{thread}}$ are 2, 32, and 512, respectively. These values are already known based on the architecture of a GPU with compute capability of 1.3 [15]. The variable $R_{\text{thread}}$ was provided in Figure 5.5 and has a value of 42 registers/thread. However, the variable $W_{\text{block}}$ was calculated using Equation 5.4. By using the block size of $<<16, 4, 1>>$ provided in Figure 5.5 for $x\text{Dim}$, $y\text{Dim}$, and $z\text{Dim}$, respectively, and 32 for $T_{\text{warp}}$, Equation 5.4 yields a value of 2 warps per block.
These values are plugged into Equation 5.6 which yields a value of 3072 registers per thread. Since there are 16,384 registers per multiprocessor, $R_{MP}$, and MaxBlocks is 8, Equation 5.7 yields a result of 5 active blocks per multiprocessor. This is the smallest of the three limiters when compared to the 8 active blocks of the block dimension limiter (Equation 5.5) and the 8 active blocks of the shared memory limiter (Equation 5.9). The result of 8 active blocks is obtained from Equation 5.5 using the value of 1024 threads per multiprocessor for $T_{MP}$ and calculating a value of 64 for $T_{block}$ by multiplying the values of $xDim$, $yDim$, and $zDim$ provided in Figure 5.5. The result of 8 active blocks is obtained from Equation 5.9 by first calculating a value of 1536 for $S_{block}$ in Equation 5.8. $S_{int}$ of Equation 5.8 is determined to be 1187.84 bytes, since the shared memory/block is reported as 1.16 KB in Figure 5.5. Combining this with a $G_{share}$ value of 512 yields the value of 1536 for $S_{block}$ in Equation 5.8. By utilizing 16,384 for the amount of shared memory per multiprocessor, $S_{MP}$, and 1536 for $S_{block}$, the result of Equation 5.9 yields 8 active blocks. With the smallest of the three limiters being 5 active blocks per multiprocessor and each block containing only 2 warps (Equation 5.4), there are only 10 active warps per multiprocessor (Equation 5.3). Having only 10 warps active means the occupancy is only 31.2% (Equation 5.2 with $W_{total} = 32$), which is in agreement with the theoretical occupancy reported in Figure 5.5.

However, Figures 5.6 and 5.7 show that the two kernels used to executing the NPP equivalent test program have occupancies of 100%. This means every available warp in the GPU is active when executing the kernels. This may explain why the NPP functions $nppsMulC_32f_I$ and $nppsAddC_32f_I$ execute much faster than does $cublasSgemm$ in this study.
To assist in determining the occupancy of a kernel, NVIDIA created an Occupancy Calculator using Microsoft Excel [16]. This free download calculates the occupancy of a kernel using the same calculations depicted in Equations 5.2 – 5.9.

Fig. 5.6 Properties session for nppsMulC_32f_I kernel

Fig. 5.7 Properties session for nppsAddC_32f_I kernel
Occupancy is not the only problem with cublasSgemm. There is also an issue with its computational accuracy. Similar to the values calculated in Table 5.3, the discrepancy percentage and average margin of error were calculated. For cublasSgemm, the average discrepancy percentage is 43.07% and the average margin of error is 0.000411 when compared to the CPU results. Not only is the discrepancy percentage much higher than any of the NPP kernels tested in this work, but the margin of error is several magnitudes higher. When 43% of the time there is an error averaging 0.000411, there is high probability that the results of the parallel conflict detection algorithm will not be the same as those of the sequential version if cublasSgemm is used as one of the kernels. Because of the long execution times and the computational inaccuracies, it was decided that the cuBLAS library was not a viable option for this particular work.

5.1.3 Thrust

The third CUDA library tested for this work is the Thrust library. This library comes free with the CUDA SDK download. Thrust is a template structured library based off of the C++ Standard Template Library (STL) [11]. Thrust provides kernels that allow for scanning, sorting, and reducing of data. These kernels could be useful in locating violations in track and geometry pairs in lieu of a computationally intensive loop alternative. Three Thrust kernels were tested for this work. They are sequence, count_if, and remove_copy_if.

The Thrust kernel sequence creates a sequence of numbers from 0 to N – 1, where N equals the number of elements in the sequence. The count_if kernel counts the number
of elements within a data set that match a specified value. However, the remove_copy_if kernel is more complicated. Three arrays are required for the inputs to this kernel. For this explanation, the three arrays are named A, B, and C. This kernel also requires a Thrust predicate. A Thrust predicate calls a user defined C programming structure that implements a comparison. The remove_copy_if kernel searches for all the elements in array B that do not comply with the Thrust predicate. Next, the kernel copies to array C the elements in array A corresponding to the noncomplying elements in B. An issue with this kernel is that the size of C must be known beforehand since memory allocation for kernel input data is required when implementing GPU computations. The kernel count_if can be used to find the number of elements that do not comply with the Thrust predicate before the implementation of remove_copy_if. By knowing the number of elements that do not comply, the programmer can then allocate the correct amount of memory for array C, and the kernel remove_copy_if can function properly without issue.

Three test programs were developed to compare the accuracy and execution times of the aforementioned Thrust kernels to their CPU equivalents. These test programs follow the same method of testing used to test the NPP kernels described in Section 5.1.1. Similar to the program structure outlined in Figure 5.1, the CPU equivalent of these kernels is implemented first, followed by the implementation of the kernels on the Quadro FX 5800 and then the Tesla C1060 with each iteration of the infinite while loop increasing the number of rows and columns in the data sets by 500 elements each. Table 5.4 lists the Thrust kernels along with the average percent difference between the CPU equivalent and the GPU implementation of the kernels.
Table 5.4
Execution time performance results for Thrust kernels

<table>
<thead>
<tr>
<th>Thrust Kernel</th>
<th>Quadro FX 5800 Percent Difference</th>
<th>Tesla C1060 Percent Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>thrust::sequence</td>
<td>44.68%</td>
<td>49.65%</td>
</tr>
<tr>
<td>thrust::count_if</td>
<td>94.59%</td>
<td>94.06%</td>
</tr>
<tr>
<td>thrust::remove_copy_if</td>
<td>38.85%</td>
<td>44.04%</td>
</tr>
</tbody>
</table>

The results in Table 5.4 show an overall increase in performance speed over the CPU implementation, especially for *count_if* which is about 94% faster. Furthermore, the accuracy of the GPU Thrust programs were in 100% agreement with those of the CPU counterparts. With these positive results, it is easy to declare the Thrust library as a viable option when implementing the parallel version of the conflict detection algorithm. However, the issue of asynchronous programming may be a cause for concern since the version of Thrust available with CUDA v5.5 SDK does not allow for asynchronous programming. Asynchronous programming will be discussed in further detail in the next section, Section 5.2.

5.2 Asynchronous Programming

Asynchronous programming allows some concurrent CUDA operations [17]. For example, some CUDA enabled machines support asynchronous memory copy from host to device or device to host while executing a kernel [18]. Some devices can execute
multiple kernels concurrently [18]. Advantages of this approach include hiding some of
the overhead associated with the use of the GPU and introducing more parallelism to
improve execution time. The use of streams is required to utilize asynchronous capability
on the GPU. CUDA allows creation of streams. A stream is a sequence of commands
that execute in order. Each of the streams requires a separate kernel invocation, but they
can potentially be used to hide some of the associated overhead. Streams can be
explicitly synchronized so that a second stream does not start until the first is completed.
This would be necessary if dependencies exist, i.e., if a stream operation depends on the
output of another. By default, a CUDA program has only one stream called the default
stream. Research was conducted to determine if savings in execution time could be
realized by introducing streams into the parallel version of conflict detection code.

Streams are created using the CUDA call `cudaStreamCreate` for each stream. To
enable concurrency, `cudaMemcpyAsync` must be used in conjunction with this for
asynchronous data transfers in both uploading and downloading of data. Furthermore,
the host memory to be transferred to the GPU must be allocated using the CUDA
function `cudaMallocHost`. NPP has built-in functions to enable the use of streams, thus
eliminating the need for programmer-defined kernels, which are often inefficient. The
NPP function `nppSetStream` is invoked prior to assigning NPP kernels to a particular
stream.

An important issue in the use of multiple streams involves the sequence in which
data is uploaded and downloaded and the order in which kernels are invoked. There are
two basic ways in which the order of the code can be written. The first approach is called
depth first [17]. In this method, all the operations of a given stream are grouped together.
Code to upload data for the first stream occurs first in the program, followed by the operations to be executed by the first stream, and then by operations to download data for the first stream. This sequence of operations is repeated for the second stream, then the third, etc. Figure 5.8 below provides an example of the depth first method. In this code snippet, every loop represents a stream. In the first loop, all of the kernels assigned to the first stream are performed; in the next loop all of the kernels associated with the second stream are executed and so on until all streams have completed.

```c
//Depth First Programming: Loop {Upload, Kernel, Download}
for (i = 0; i < nStreams; i++)
{
    //Memory offset
    offset = i * StreamSize;
    //Upload to device (GPU)
    cudaMemcpyAsync(&d_A[offset], &h_A[offset], nstreambytes, cudaMemcpyHostToDevice, s[i]);
    //Tell GPU which stream to run NPP function
    nppSetStream(s[i]);
    //Add a constant i to device variable
    //Represents kernel
    nppsAddC_32f_I1(1, &d_A[offset], StreamSize);
    //Download to host (CPU)
    cudaMemcpyAsync(&h_A[offset], &d_A[offset], nstreambytes, cudaMemcpyDeviceToHost, s[i]);
}
```

Fig. 5.8 Depth first method of asynchronous programming

The second technique for creating programs involving streams is called breadth first [17]. In this method, all uploading to the GPU for all of the streams occurs first in the program, followed by all of the kernel calls for all of the streams in sequence. Finally, downloading for all of the streams occurs last. Figure 5.9 below demonstrates code using the breadth first method.
Three simple programs were written to test these methods. The purpose was to determine which method of asynchronous programming, depth first or breadth first, provided the most benefit, i.e., savings in execution time, on the GPU platform on which our parallel conflict detection code is executing. Thus, one test program was written as a baseline program using the default stream. The other two programs were written with four streams each. One of these utilized the depth first method; the other used breadth first. Each stream contained one fourth of the total data allocated, which was incremented each iteration, up to a maximum of approximately 4 gigabytes (GB). All three programs uploaded and downloaded three variables. To introduce kernels, five NPP addition kernels (*nppsAdd_32f*) were called in succession. Thrust kernels are unable to execute asynchronously in CUDA v5.5 so those kernels were not used in these tests.
In the two asynchronous test programs, each stream called the NPP kernel five times. Since there were four streams in each program, the NPP kernel was called in the program a total of 20 times in order to be equivalent to the default case which used a single stream.

The NPP addition kernel performed an element by element addition on two input arrays and placed the results into a third variable. Figure 5.10 shows the NVIDIA Visual Profiler results for a single GPU running on the default stream. The number of streams being executed can be verified by looking at line 11 titled “Streams” in the session frame. This line displays all the streams that are active. So in the case of Figure 5.10, only the default stream is active as it should be for this particular test. As expected, the screenshot shows that there is no overlap in time between uploading data, kernel execution, and downloading data. Rather, each of these operations is sequential and all occur on the single existing stream.

![Fig. 5.10 NVIDIA Visual Profiler screenshot of a program running on the default stream](image)

Figure 5.11 shows the screenshot of the NVIDIA Visual Profiler results for the depth first method of asynchronous programming using four streams. It displays the operations being performed on each of the four streams, including uploading and
downloading of data and kernel execution. For this particular test, the program was set up to cycle through the program four times in order to isolate CUDA’s initialization process from the results of Figure 5.11. Thus, streams 8 – 15 are not shown because they occurred within the first two cycles of the test program. Only the third cycle, which contains streams 16 – 19, is shown in Figure 5.11. Note that since there is no time overlap between the data transfer and the kernel blocks, none of the four streams executed concurrently. Rather, each stream executed sequentially, with the next stream not starting until the previous had completed. Therefore, this program exhibited no benefit over that running on the default stream.

Fig. 5.11 NVIDIA Visual Profiler screenshot of a program running four streams depth first
The reason for this lack of concurrency in the depth first method is related to how the operations are scheduled to execute and by the number of task engines on the GPU. Graphical Processing Units are equipped with devices, called engines, used to transfer data to and from the GPU (copy engine) or execute a kernel (kernel engine) [17]. These engines can run concurrently if multiple streams are in use. When a CUDA program is compiled, a copy engine stores the order of execution of all data transfers based on the sequence in which they occur in the code. Similarly, the kernel engine stores the order at which the kernels are programmed. The engines are equivalent to a first-in-first-out queue. The number of engines available on a GPU device varies. Some GPUs have two copy engines; one for uploading and one for downloading plus a kernel engine [18]. In this case, three operations could be executed concurrently on three different streams – uploading, kernel execution, and downloading. Using different device engines on different streams is what enables uploading and downloading to be accomplished concurrently on a single GPU. The Quadro FX 5800 and the Tesla C1060 both have only a single copy engine responsible for all data transfers (uploading and downloading) and one kernel engine responsible for all kernels, thus prohibiting concurrent uploading and downloading on different streams when operating on a single GPU.

Due to the structure of the depth first method and the architecture of the Quadro FX 5800 and the Tesla C1060, concurrent execution is not possible on this platform using the depth first method. Since this platform has only one copy engine and one kernel engine, the order in which the code is written using the depth first method does not allow the kernels to run concurrently with the data transfers because the kernels have already performed their respective operations before the second stream begins its uploading of
data. Note however, that concurrent execution could be attainable using the depth first method if the GPU device contained two copy engines. On the other hand, the breadth first method does allow for concurrency on the available GPU platform, as shown in Figure 5.12.

![NVIDIA Visual Profiler screenshot of a program running four streams breadth first](image)

Fig. 5.12 NVIDIA Visual Profiler screenshot of a program running four streams breadth first

In Figure 5.12, the NPP kernels (turquoise blocks) of one stream are able to run at the same time as the memory transfer (brown blocks) of another stream. Due to the structure of the breadth first method, once the copy engine uploads the data of the first stream, the kernel engine can start executing it while the copy engine is uploading data from the second stream. This method for writing a program provides the ability to perform true concurrency. Unfortunately, in this case, due to the architecture of the available GPU devices, the downloading of data cannot begin until the last stream has
finished upload because the Quadro FX 5800 and the Tesla C1060 contain only one copy engine.

Figure 5.12 also shows that the total execution time required for all the NPP kernels is hidden behind the data transfer overhead which can be seen in lines 8 – 10 in the workspace frame. That is, the data transfers to and from the GPU define the start and stop times of this program. In fact, in this program, the kernel execution time could be substantially increased without impacting the total program execution time. Figure 5.13 shows what happens when instead of 5 kernels per stream, as in Figure 5.12; there are 20 NPP calls (kernels) per stream, thus increasing kernel execution time.

---

Fig. 5.13 NVIDIA Visual Profiler screenshot of a program running four streams breadth first with 20 kernels per stream

In Figure 5.13, even though the kernel execution time has substantially increased over that of Figure 5.12, total execution time is still driven by the data transfer time. In Figure 5.13, all the kernels are still able to run concurrently with the data transfers. For this particular case the NPP addition kernel is invoked 80 times (4 streams x 20 NPP
kernels) and every one of those calls are hidden from the overall execution time. Figure 5.14 shows what would happen if the number of kernels per stream were to increase to 30.

![NVIDIA Visual Profiler screenshot](image)

**Fig. 5.14 NVIDIA Visual Profiler screenshot of a program running four streams breadth first with 30 kernels per stream**

In this case, total execution time is no longer dictated solely by the data transfer time. Rather, the increase in execution time has resulted in delays in download data transfers. There is still obvious concurrency, but there is limit on how many kernels can be run on a stream before kernel execution time starts to impact the total execution time.

This analysis illustrates the complexities involved in optimizing the performance of the GPU. Its performance is dependent on a number of factors, including the amount of data being transferred compared to the kernel execution time, the number of copy and kernel engines on the GPU, and the design of the software implementation.
It is important to note that the method used to record the execution times for all asynchronous programs in this work is different from the \texttt{clock()} function method used in other cases. In all asynchronous programs in this work, including the final asynchronous versions of the conflict detection algorithm, the CUDA function \texttt{cudaMemcpyAsync} is used to upload and download data to the GPU. The CUDA function \texttt{cudaMemcpyAsync} is an asynchronous function so it will immediately return control to the CPU unlike the synchronous CUDA function \texttt{cudaMemcpy} which will return control as soon as the data has finished transferring [17]. Since control is returning to the CPU before the GPU finishes executing the called instructions, a method was used to allow the GPU to finish all instructions on all streams before control is returned to the CPU for recording. By allowing the GPU to finish executing all asynchronous CUDA calls, a proper execution time can be recorded.

The method used to properly record the execution times of an asynchronous program involves CUDA event timers [19]. Two CUDA events are created using the CUDA function \texttt{cudaEventCreate}, which will serve as inputs for the CUDA function \texttt{cudaEventRecord}. Similar to the \texttt{clock()} function in C++, a \texttt{cudaEventRecord} function will be placed at the beginning and at the end of the part of the program that will be measured for execution time. The key difference here is that the CUDA function \texttt{cudaEventSynchronize} will be called directly after the second \texttt{cudaEventRecord}. The CUDA function \texttt{cudaEventSynchronize} will ensure all GPU instructions have finished executing before returning control back to the CPU, thus ensuring all instructions executed on the GPU are accounted for when recording the execution times.
Figure 5.15 shows the execution times of the programs depicted in Figures 5.10, 5.11, and 5.12. The test programs were executed on the Tesla C1060. For these tests, the asynchronous breadth first method exhibits the fastest execution times.

![Graph](image)

**Fig. 5.15** Execution times for a single GPU running asynchronously

The average percent difference between the execution times of the asynchronous programs and the baseline synchronous program were calculated. The depth first method shows an average speed up of 9.59% and the breadth first method shows an average speed up of 27.03%. Even though there is no concurrency happening in the depth first method there is still a slight increase in performance. The obvious best option for
increased performance occurs when the program is executing the breadth first method of asynchronous programming.

Another approach to improve the performance of the parallel version of the conflict detection algorithm involved the use of multiple GPUs. Multiple GPUs may be introduced using multiple CUDA devices on the host system or using CUDA devices containing more than one GPU. An example of the latter includes the Quadro Plex 2200 D2, which is a single unit consisting of two Quadro FX 5800 devices. In this case, the former approach was used since the Z800 Workstation located in ET 324 at IPFW is equipped with two GPU devices, the Quadro FX 5800 and the Tesla C1060.

When running CUDA programs, the system is able to detect the number of CUDA-enabled devices residing on the host. These devices can be enumerated, their properties queried, and one of them can be selected for kernel executions. On the Z800 Workstation, the system assigned the identification number (ID) 0 to the Quadro FX 5800 and 1 to the Tesla C1060. On systems with multiple GPU devices, kernels are executed on device 0 by default. Thus, on the Z800 Workstation, the Quadro FX5800, is the default for kernel executions. The Tesla C1060 can be designated by calling \texttt{cudaSetDevice(1)} prior to the segment of code to be executed on that device. Similarly, the Quadro FX 5800 may be explicitly selected for kernel executions by first calling \texttt{cudaSetDevice(0)}. To switch execution back and forth between the two devices simply requires the appropriate call to \texttt{cudaSetDevice}.

To test the behavior of multiple GPUs in the conflict detection algorithm, the same three programs used to test the behavior of a single GPU were modified to run on multiple GPUs. In particular, these were the programs whose single GPU results are
displayed in Figures 5.10, 5.11, and 5.12. To maintain consistency with the single GPU approach, the data was split in half, with one half uploaded to the Quadro FX 5800 and the other half sent to the Tesla C1060.

Figure 5.16 shows the Visual Profiler results of a program designed to run five NPP additions on both the Quadro GPU and the Tesla GPU using the default stream, analogous to the baseline single GPU approach of Figure 5.10 that used a single stream. Thus, this program demonstrates the behavior of multiple GPUs without the use of multiple streams per CUDA device. The program was written such that every CUDA or NPP kernel call was preceded with a call to `cudaSetDevice` that alternated between the two GPUs. Figure 5.16 indicates that both GPUs were executing, but not completely concurrently. Note that only the execution of the kernels (NPP additions) occurred concurrently. The uploading of data from the host to the Tesla C1060 occurred after the uploading of data from the host to the Quadro FX 5800. Similarly, the downloading of data from these two devices occurred sequentially rather than concurrently. The reason for this is due to the difference in the handling of data transfer versus kernel executions by CUDA-enabled devices. Unless the data is being uploaded and downloaded asynchronously, a CUDA data transfer will not execute until the previous CUDA data transfer has completed its data transfer regardless if the transfer is on a separate GPU [17].

Kernel execution differs from that of CUDA data transfers in that for kernel launches, control is returned to the host thread before the device has completed the requested task [17]. Since control is returned to the host thread after a kernel launch, another kernel could be launched immediately if another kernel engine is available. Such is the case if
the second GPU is specified for the subsequent kernel launch, thus enabling concurrent kernel execution on the two GPU devices.

![Fig. 5.16 NVIDIA Visual Profiler screenshot of a program using the default stream and alternating execution between two GPU devices](image)

The depth first method of asynchronous programming, the results of which are depicted in Figure 5.11, was also modified to incorporate multiple GPUs. The multiple GPU version executed the depth first method of asynchronous programming using four streams on each of the two GPUs. The purpose was to ascertain whether additional benefit from the depth first method could be obtained using multiple GPUs. The multiple GPU behavior was similar to that on the single GPU as shown in NVIDIA Visual Profiler results of Figure 5.17. In contrast to Figure 5.16, the results of Figure 5.17 indicate concurrent data transfers on the two devices, in addition to the concurrent kernel executions observed in Figure 5.17. This is due to the asynchronous data transfers using the CUDA function `cudaMemcpyAsync`. However, as in the depth first results of Figure 5.11, within each of the two GPU devices, none of the four streams executed
concurrently. Rather, each stream executed sequentially, with the next stream not starting until the previous had completed. Thus, the only concurrency that occurred in the test program depicted in Figure 5.17 is that both GPUs operated at the same time but no concurrency occurred among the streams.

Fig. 5.17 NVIDIA Visual Profiler results for a program running on two GPUs using the depth first method of asynchronous programming

Finally, the breadth first method of asynchronous programming, the results of which are depicted in Figure 5.12, was modified to incorporate multiple GPUs. The
multiple GPU version executed the breadth first method of asynchronous programming using four streams on each of the two GPUs. The results are displayed in Figure 5.18. Figure 5.18 exhibits improvement over the results of Figure 5.17 in that not only are both GPUs running concurrently, but there is also concurrency among the streams within both GPUs.

Fig. 5.18 NVIDIA Visual Profiler results for a program running on two GPUs using the breadth first method of asynchronous programming
Analogous to the single GPU scenarios depicted in Figures 5.13 and 5.14, Figures 5.19 and 5.20 display results of increasing the kernel execution time using the breadth first method of asynchronization on a multiple GPU platform. Figure 5.19 is the result of 20 kernels per stream; Figure 5.20 is for 50 kernels per stream. Figure 5.19 demonstrates that kernel execution time can be increased substantially without affecting overall execution time, but Figure 5.20 shows that after an upper limit of kernel time is reached, the total execution time is impacted.

Fig. 5.19 Breadth first method running on two GPUs with 20 kernels per stream
By comparing the single GPU execution times of Figure 5.15 to those of the multiple GPU versions of the programs, the optimum approach in terms of fastest execution can be determined. Figure 5.21 provides the execution times for all six test programs, that is, those whose results are displayed in Figures 5.10 – 5.12, 5.16 – 5.18. Each program ran until hitting the memory limitations of the GPUs. Note from Figure 5.21 that the results from the synchronous version of the multiple GPUs test (results of which are displayed in Figure 5.16) appears to be cut off prematurely signifying the program ended before it reached the memory limitations of the GPUs. Such was not the
case. Rather, the results for matrices of size 22,500 x 22,500 and above were purposely thrown out for use in Figure 5.21. This was done because the execution times for those size matrices took minutes to perform and not seconds.

Also of note in Figure 5.21 is the fluctuation in execution times for the program running asynchronously on multiple GPUs. Even though there is a clear advantage to the amount of memory available with the additional GPU, this oscillation may prove to be a hindrance.

Fig. 5.21 Execution times for all versions of the GPU behavior tests
Based on these results, the most optimal approach for the parallelization of the conflict detection algorithm would be to implement the breadth first method on multiple GPUs. Other versions of the algorithm were implemented as well, such as a synchronous single GPU version and an asynchronous breadth first single GPU version. These versions were created for comparison purposes and are described in more detail in Chapter 6.
6. PARALLEL VERSION

6.1 Synchronous Single GPU

The synchronous single GPU version of the conflict detection algorithm was the first attempt at parallelizing the code. It was written and executed on the HP Z800 Workstation located in room 324 of the Engineering, Technology, and Computer Science Building on the campus of Indiana University Purdue University Fort Wayne (IPFW). The HP Z800 Workstation is equipped with both a Quadro FX 5800 and a Tesla C1060 CUDA device. Only the Quadro FX 5800 was used to execute this particular version of the conflict detection algorithm. The operating system used to develop and test the program was CentOS Linux 2.6.32-71.29.1e16.x86-64. The text editor gedit was used to write the program, and the Linux terminal was used to execute the parallel program. The program was compiled using the NVCC command since the program file was saved as a .cu file.

The first major difference between the sequential version and this synchronous single GPU parallel version involves the organization of the data sets used for testing. The randomly created data in the parallel version is exactly the same as the data created in the sequential version, but the size and structure of the arrays used to store this information are different. The arrays were reconstructed in order to take advantage of the
parallel capability of CUDA. The data was arranged in order for all tracks to be checked for the violations against all sides of every geometry in parallel. For the sequential version of the algorithm, two sets of randomly generated x, y, z Cartesian coordinates were created for each track – one to represent the track’s current position; the other representing its previous location. The coordinates for these two sets of track data were stored into 6 arrays, each of length $T$, where $T$ is the number of tracks. For the parallel version, the track data was stored into 6 arrays, each of length $6 \cdot T \cdot G$, where $T$ is the number of tracks and $G$ the number of geometries. The $T$ number of elements in each of the six arrays in the sequential version is repeated $6G$ times in the parallel version. This allows for every possible combination of each track and each side of a geometry to be compared in parallel.

In addition to storing the current and previous track position, information about the geometries was also stored in arrays in preparation for running the parallel version of the algorithm. This information included the normal for each side of each geometry, the maximum and minimum points for each geometry, and a point on each side of the geometry. The following includes a brief explanation of how the data for each of these arrays was arranged.

**Normal for Each Side of Each Geometry:** Three separate arrays were used, one for each coordinate of the normal. The array starts with the normal of the north side of Geometry 1 repeated $T$ times, where $T = \text{total number of tracks}$. The normal for the east side of Geometry 1 follows, also replicated $T$ times. This continues until each side (6 total sides) of Geometry 1 has been stored $T$ times in succession. Then, the normal for each side of Geometry 2 is replicated $T$ times and stored in the array following the
information for the first geometry. This is repeated for all $G$ geometries yielding three arrays (one for each coordinate $x$, $y$, and $z$), each of length $6 \cdot T \cdot G$.

**Geometry High Points:** Three separate arrays were used, one for each coordinate of the maximum value of each side of each geometry. The values were stored in a manner similar to that described above for the normals, i.e., the maximum value of the north side of Geometry 1 was repeated $T$ times, followed by the maximum value for the east side of Geometry 1, repeated $T$ times, etc. The values for the remaining geometries were constructed in the same manner and concatenated to those of the first, resulting in three arrays, each of length $6 \cdot T \cdot G$.

**Geometry Low Points:** Three separate arrays were constructed in the same manner as those for the Geometry High Points, resulting in three arrays, each of length $6 \cdot T \cdot G$.

**Point on the Plane:** Three separate arrays were constructed in the same manner as those for the Geometry High and Low Points, resulting in three arrays, each of length $6 \cdot T \cdot G$.

Another major distinction between the synchronous single GPU parallel version and the sequential version is that this parallel version of the conflict detection algorithm does not contain program loops. However, for testing purposes, the program does contain an infinite `while` loop for reasons similar to that of the sequential version. As in the sequential version, each iteration of the infinite `while` loop will increase the number of tracks and the number of geometries by 500 elements and 50 elements, respectively.
The equations used to find violations in the synchronous single GPU parallel version are the same as those used in the sequential version. However, in the GPU version, NPP kernels replaced the simple operands used in the sequential version to perform arithmetic operations. For example, the NPP kernel \textit{nppsSub\_32f\_I} was used instead of a simple subtract operand when calculating the directional vector between the current position of a track and its previous position. Rather than having to loop the simple subtraction instruction \( T \) times, the NPP kernel enabled computation of every directional vector using one program instruction call.

In the synchronous single GPU parallel version, the \textit{nppiCompare\_32f\_C1R} and the \textit{nppiCompareC\_32f\_C1R} kernels from the NVIDIA NPP library replaced all of the comparison checks of the sequential version, enabling \( 6 \cdot T \cdot G \) comparisons to be accomplished in parallel. The NPP library kernel \textit{nppiCompare\_32f\_C1R} performs an element-by-element comparison of two sets of data, and outputs the Boolean results to a third array. The \textit{nppiCompareC\_32f\_C1R} kernel performs an element-by-element comparison of a data set to a constant value and also outputs the Boolean results to a third array. In each case, the output array of these compare kernels consists of 8-bit unsigned character types, of value 0 or 255. The value of 255 indicates a true condition; 0 indicates false. These kernels were used to find the results of the comparisons depicted in Section 2.3.

The output arrays for each comparison check were summed into one final composite array. For this particular program, if a 0 was found in the final comparison array, it meant that an actual or imminent violation had occurred. The locations of the 0s within this array corresponded to locations in other arrays that provided the desired
information regarding the tracks and geometries in violation. Thus, it was necessary to determine the locations of these 0s. One option was to download this array to the CPU and perform the search on the CPU. It was believed that the CPU approach would not be as efficient as that on the GPU, due to excessive looping. Furthermore, since the full array was very large and would require a large transfer at this point, it was decided to extract the 0s and their corresponding locations on the GPU and download to the CPU a condensed array containing only this information. To do this, the NVIDIA Thrust library kernels \textit{count\_if} and \textit{remove\_copy\_if} were used to perform an operation on the GPU similar to that of the combination of Matlab \textit{find} and \textit{nonzeros} commands. First, an array of index values had to be created. An array, \textit{index}, of length $6 \cdot T \cdot G$ and containing numbers ranging in sequence from 0 to $6 \cdot T \cdot G - 1$ was created to represent the indices of the final comparison array. Then the Thrust kernel \textit{count\_if} was employed to count the number of 0s in the final comparison array. If there were no 0s, indicating no violations, then the program ended. If there were 0s in the array, then the Thrust kernel \textit{remove\_copy\_if} was implemented with a Thrust predicate that compares if an element in the final comparison array is equal to zero or not. The kernel \textit{remove\_copy\_if} will search the final comparison array for all values not equal to zero. The elements within the array \textit{index} that correspond to the elements that are equal to zero in the final comparison array are then copied over to a third array called \textit{index2}. Since the number of violations is unknown at the start of the program, the result of the kernel \textit{count\_if} was used to determine the number of elements that will be contained in the array \textit{index2}. The data in the resulting \textit{index2} was then transferred from the GPU to the CPU. On the CPU, \textit{index2} was used to compute the track Identifications (IDs) and the geometry IDs of the
violations and to extract corresponding measurements such as distance to the intersection point and velocity of the track. The pseudocode used to find the actual Track ID and Geometry ID for the corresponding index position of the located 0 is shown below.

\[
\begin{align*}
if \ (\text{Index} + 1) \mod (T) = 0 \\
Track \ ID &= T \\
else \\
Track \ ID &= (\text{Index} + 1) \mod (T)
\end{align*}
\]

\[
Geometry \ ID = \frac{\text{Index}}{6 \cdot T} + 1
\]

In addition to downloading the index2 array, several other large arrays were also downloaded to the CPU. These included arrays containing distance to intersection point, time to intersection point, track velocity, and the actual intersection point. This latter information was downloaded to create an output report consistent with that of the sequential version of the algorithm.

One last major difference between the sequential version and the synchronous single GPU parallel version of the algorithm involves the continuous uploading and downloading of data to and from the CPU and the GPU. The CUDA architecture requires data used in the computation to be transferred from the CPU to the GPU. The desired computations are then performed on the GPU, and the result must be transferred back to the CPU. This transferring of data takes time and creates overhead, which impacts execution time. A GPU also has a limited amount of global memory. In order for the global memory of the CUDA device to not be consumed too quickly, the data transfers are scattered throughout the algorithm. For a particular calculation in the
algorithm, only the required data is uploaded to the GPU. Once the data is no longer needed for the rest of the program, the data is immediately deallocated from the GPU in order to make room for data needed in future computations. However, the transfer of data from the GPU back to the CPU is limited to a one-time transfer of final results at the completion of the program.

A total of 18 floating point arrays of length $6 \cdot T \cdot G$, where $T$ is the number of tracks and $G$ the number of geometries, are allocated and transferred from the CPU to the GPU. However, 11 additional floating point arrays and 9 unsigned character arrays of length $6 \cdot T \cdot G$ are allocated on the GPU but data in these arrays are not transferred to the GPU. That is because these additional arrays serve as empty sets that are used to store results for certain calculations throughout the program. Since these arrays are initially empty, there is no data to be transferred to the GPU. Thus, there is no need to transfer those arrays to the GPU. Each value in the floating point array requires 4 bytes of storage; each unsigned character uses 1 byte of memory. Thus, the total number of bytes transferred to the GPU is as given in Equation 6.1. However, the total amount of data allocated on the GPU is given in Equation 6.2. To reiterate, these data allocations and transfers are not occurring all at once. They are being scattered throughout the program.

$$Total \, Bytes \, Transferred = (18 \cdot 4) \cdot 6 \cdot T \cdot G$$ \hspace{1cm} (6.1)

$$Total \, Bytes \, Allocated = (29 \cdot 4 + 9 \cdot 1) \cdot 6 \cdot T \cdot G$$ \hspace{1cm} (6.2)

In addition, at least 6 floating point arrays of length $6 \cdot T \cdot G$ are downloaded from the GPU to the CPU at the completion of the program, corresponding to $24 \cdot 6 \cdot T \cdot G$ bytes. This transfer of data to and from the GPU severely impacts the performance of the parallel version of the algorithm.
Figure 6.1 shows the flowchart of the synchronous single GPU version of the conflict detection algorithm. Note that there are no program loops in the flowchart since the data in each step is being computed in parallel.

Figure 6.2 displays the NVIDIA Visual Profiler results for the synchronous single GPU version of the conflict detection algorithm. This particular screenshot was taken when the number of tracks equaled 1,500 and the number of geometries equaled 150. As expected, all CUDA operations are occurring in the default stream. However, in between most of the operations there appears to be portions of time where the GPU is idle. It is unknown at this time as to why the GPU is idling at those positions which is unfortunate since they appear to have some effect to the overall execution of the algorithm.
Fig. 6.2 NVIDIA Visual Profiler results for the synchronous single GPU version of the conflict detection algorithm

To see the effects of those idle times along with the overhead caused by the data transfers, Figure 6.3 shows the execution times for the synchronous single GPU version of the algorithm along with the sequential version results gathered in Section 3.4. The program is written to continuously run with the track and geometry values increasing by 500 and 50, respectively, after each execution cycle. Similar to the sequential version, only the execution time of the portion of the program that is executing the conflict detection algorithm is being recorded. The same \texttt{clock()} function used to measure the execution times of the sequential version was used to measure this version of the algorithm. The initial spike in execution time for the parallel version of the algorithm is due to the initialization that occurs when executing an NPP kernel for the first time. This only happens once at the first time an \texttt{npps} or an \texttt{nppi} prefixed NPP kernel is executed. Also note in Figure 6.3 the limited number of tracks and geometries that were successfully executed on the program. Whereas the sequential version of the code was able to handle at least 200,000 tracks and 20,000 geometries, the parallel version at this point is only able to process 3,500 tracks and 350 geometries. One reason for this is due to the way the NPP library compare kernel calculates the grid and block dimensions. This will be explained further in the following section of this report. Note that the
limited number of tracks and geometries able to be processed at this point is not due to the 4 GB memory limitation of the GPU. Per Equation 6.2, 3500 tracks and 350 geometries are using only around 0.856 GB of memory. Due to this limitation, only the corresponding executions times of the sequential version of the program are being used for comparison purposes. For this limited amount of data, the parallel version of the algorithm does not exhibit improved performance over that of the sequential counterpart.

Fig. 6.3 Synchronous single GPU and CPU execution time results
6.2 Issues with Synchronous Single GPU Version

Several issues were identified that contributed to the poor performance of the synchronous single GPU parallel version of the algorithm. These issues include limited GPU global memory, data transfer overhead, and grid dimension allocation in the NPP compare kernels.

6.2.1 NPP library grid dimension limitation

The first issue contributing to the poor performance of the synchronous single GPU version of the algorithm involves the way the NPP compare kernels define the grid and block dimensions. The NPP compare kernels used in this parallel version of the algorithm are the \textit{nppiCompare\_32f\_C1R} and the \textit{nppiCompareC\_32f\_C1R} kernels. Only those two NPP compare kernels were thoroughly examined.

The maximum allowable grid size and number of threads per block depends on the compute capability of the CUDA device and are critical parameters in the speed of execution of the code. Recall from Table 4.1 that the Quadro FX 5800 has a maximum grid dimension of $\langle\langle65535, 65535, 1\rangle\rangle$ and a maximum block dimension is $\langle\langle512, 512, 64\rangle\rangle$. Below is a pseudocode describing how the NPP compare kernels determine the grid dimensions. This was determined through trial and error and confirmed using the CUDA Debugger and Performance Analyzer.

$$Grid \ rows = ceil\left(\frac{\# \ of \ rows \ in \ input \ matrix}{8}\right)$$
\[
\text{If } (\text{# of columns in input matrix}) \mod (4) = 0
\]

\[
\text{Grid columns} = \text{ceil} \left( \frac{\text{# of columns in input matrix}}{128} \right)
\]

\text{else}

\[
\text{Grid columns} = \text{ceil} \left( \frac{\text{# of columns in input matrix}}{32} \right)
\]

The NPP compare kernels utilized in the parallel version of the conflict detection algorithm allocate grid dimensions corresponding to the number of rows and columns of the input matrix. If the number of columns in the input matrix is divisible by four, the maximum number of allowable input columns to the NPP compare kernels before exceeding grid size limitations is 8,388,480 (65,535 X 128). If the number of columns in the input matrix is not divisible by four, the maximum number of allowable input columns to the NPP compare kernel is 2,097,120 (65,535 X 32). Therefore, using the NPP compare kernel with the input matrix dimension of 1 x 6GT posed a limitation to its use well below the 4 GB memory limitation of the GPU. In particular, test results were limited to 3,500 tracks and 350 geometries, resulting in a grid column dimension of 57,422, which is below the 65,535 maximum allowable. Increasing to 4,000 tracks and 400 geometries would result in a grid column dimension of 75,000, which exceeds the maximum allowable number. As a result, the kernels do not launch, but due to an apparent glitch in the kernel, the program does not exit. It does yield an error of “CUDA Kernel Execution Error” when queried, but the program continues on.

These apparent limitations to the NPP comparison kernels can easily be fixed by focusing on the two specific input variables used in both the \textit{nppiCompare\_32f\_C1R} kernel and the \textit{nppiCompareC\_32f\_C1R} kernel. The two variables include “Line Step”
and “NppiSize.” “Line Step” is an integer representing the amount of memory allocated to each row of the input matrix. “NppiSize” is a variable containing two values—the number of rows and the number of columns in the input matrix. It was discovered that “Line Step” and “NppiSize” do not have to be the same as the actual dimensions of the uploaded matrices. Rather, they can be set to whatever the programmer desires as long as they are consistent with each other within the confines of the NPP comparison kernels. By simply changing the values of “Line Step” and “NppiSize” so that the number of columns never exceeds the limitations of the NPP comparison kernels, the parallel program can exceed the 3,500 track and 350 geometry limitation. This eliminates the need to physically rearrange the data every time the number of columns exceeds the grid dimension limitation. If the number of columns in the matrix to be passed to the NPP compare kernels will result in a grid dimension that exceeds 65,535, then “NppiSize” can be defined to have twice the number of rows and half the number of columns of the actual input matrix. The variable “Line Step” would have to be changed accordingly to account for the change in the variable “NppiSize.” The new “Line Step” would be the amount of memory in half of a row of the actual data matrix. “Line Step” and “NppiSize” of both input matrices must be defined identically. Defining the input variables in this way does not change the order in which the variables are stored in memory, but simply regroups them for the purposes of the element-by-element comparisons.
6.2.2 GPU global memory limitation

Limited global memory may also be an issue in the performance of the synchronous single GPU version of the algorithm. The Quadro FX 5800 contains 4 GB of global memory. If the amount of memory allocated exceeds the 4 GB limit, then the program exits with a memory allocation error. If the NPP compare kernel were to execute properly regardless of the number of columns in the matrix, the synchronous single GPU program would only be able to handle 10,000 tracks and 1,000 geometries. The program fails when 10,500 tracks and 1,050 geometries are introduced. This is because the global memory limitation of the GPU has been reached. By using Equation 6.2, the total amount of memory allocated to the GPU for 10,000 tracks and 1,000 geometries would be about 6.98 GB. This clearly exceeds the global memory limit of 4 GB of the Quadro FX 5800, yet that much data was able to be processed on the GPU. This is due to the GPU memory continually being allocated and deallocated throughout the program. However, that staggered nature of the allocations and deallocations will not prevent a bottleneck scenario where so much memory is allocated at one specific part of the program that the program will fail due to exceeding the memory limit. This is what occurred when there were 10,500 tracks and 1,050 geometries.

For this particular parallel algorithm, the bottleneck occurs at the portion of the program where memory is being allocated for use in the NPP comparison kernels that check to see if the trajectories of the tracks are intersecting within the bounds of the planar patches that represent the sides of the geometries. The global memory limitation of the Quadro FX 5800 will be exceeded at that part of the program when the number of
tracks equals 10,500 and the number of geometries equals 1,050. At that part of the program, there needs to be enough memory allocated for 16 floating point arrays and 6 unsigned character arrays. With that many arrays for 10,500 tracks and 1,050 geometries, the amount of memory that needs to be allocated on the GPU is about 4.31 GB which is greater than the 4 GB limit.

Several solutions can alleviate this bottleneck. One solution is to completely reorganize the data sets so that they are smaller, though utilization of full parallelism may be lost due to the possible introduction of program loops. Another is solution would be to use a second GPU, though the same bottleneck issue would eventually occur again. A third solution would involve asynchronous programming to “hide” memory usage by uploading data at the same time memory at those same locations is being downloaded. However, this would require a GPU device with more than one copy engine which are not available at this time on the campus of IPFW.

6.2.3 Transfer overhead limitation

The overhead created by the need for large data transfers to the GPU is a bottleneck inherent in the synchronous single GPU version of the algorithm. The large increase in the size of the matrices over those of the sequential version, due to replication of data, to accommodate parallel operations further exacerbates the problem. As depicted in Figure 6.2, large portions of the program are performing these data transfers which increases the overall execution time of the algorithm. This can be alleviated by structuring the program to execute asynchronous. In Section 5.2, it was found that the
only way to write a program asynchronously with the current available GPUs at IPFW was to use the breadth first method, which would mitigate the issue of data transfer overhead. Using a GPU with multiple copy engines and kernels engines would help “hide” even more CUDA operations. One long term solution to transfer overhead is to use a different GPU architecture altogether. A possible option would be to use the Advanced Micro Devices (AMD) Heterogeneous System Architecture (HSA) [20], which allows shared memory between the CPU and GPU without memory copies.

6.3 Asynchronous Single GPU

The asynchronous single GPU program was created to address some of the issues with the synchronous single GPU version identified in Section 6.2 above. The asynchronous single GPU version was modeled after the breadth first method of asynchronous programming, in which all data for all streams is uploaded first to the GPU, followed by the execution of all the kernels on each stream, and then finishes by downloading all the data on each stream back to the CPU. To enable the program to execution asynchronously, the CUDA function `cudaMemcpyAsync` was used to upload and download the data and `cudaStreamCreate` was used to create streams. Also, the NPP function `nppSetStream` is used to specify the streams on which subsequent NPP kernels will be executed. This version of the algorithm was tested using the same method as that of the synchronous and sequential versions on the same computer. However, the asynchronous single GPU version was run on the Tesla C1060 GPU card rather than the Quadro FX 5800 which was used for the synchronous version.
The flowchart for the asynchronous single GPU version of the algorithm is shown in Figure 6.4. The structure of the algorithm in this version is quite different from that of synchronous version depicted in Figure 6.1.

Fig. 6.4 Asynchronous single GPU flowchart
The first major difference between the asynchronous single GPU version and the synchronous single GPU version is that all the data used in the asynchronous single GPU program is allocated and uploaded beforehand and not staggered as in the synchronous program. This was done to prevent idle times due to streams having to wait for completion of the initial round of data uploads before starting a second round of uploads. Also, a smaller number of floating point arrays and unsigned character arrays were allocated in the asynchronous single GPU version compared to the synchronous single GPU. The reduction in the number of arrays was accomplished by using the equivalent in-place version of certain NPP kernels used in the synchronous program. This allowed for the elimination of arrays that were created for the sole purpose of storing interim results. The number of floating point arrays allocated in the asynchronous signal GPU version is 23 and the number of unsigned character arrays is 2, whereas the synchronous single GPU version had 29 and 9, respectively. The number of floating point arrays transferred, 18, is the same for both GPU versions since the exact same data is being used. The total amount of data transferred and allocated to the GPU for the asynchronous single GPU version is found using Equations 6.3 and 6.4, respectively.

\[ \text{Total Bytes Transferred} = (18 \cdot 4) \cdot 6 \cdot T \cdot G \]  \hspace{1cm} (6.3)

\[ \text{Total Bytes Allocated} = (23 \cdot 4 + 2 \cdot 1) \cdot 6 \cdot T \cdot G \]  \hspace{1cm} (6.4)

Another difference between the synchronous version and the asynchronous version is the inclusion of loops in the asynchronous version due to the introduction of streams. Since the asynchronous version of the parallel program uses the breadth first method, each iteration of a loop represents the tasks occurring on a single stream. Four
streams were chosen for testing through trial and error. The number of streams to use depends on what the particular program is trying to accomplish. There is no set method for selecting the number of streams so it is basically up to the programmer to decide how many streams would work for a particular program. Each stream contains one fourth of the total data. This means the first stream contains data from the first quarter of each array, and the second stream contains data from the second quarter of each array and so on for all four streams.

The last major difference is the placement of the GPU downloads. For the asynchronous parallel version, the information regarding possible violations are being downloaded before it is determined if a violation has occurred. In the synchronous version, the information is downloaded after. This decision can be explained using the NVIDIA Visual Profiler results for the synchronous single GPU version of the parallel algorithm shown in Figure 6.5. This figure shows the concurrency among the four streams that is occurring in the program. This particular screenshot was taken when the program was running 1,500 tracks and 150 geometries. Even at this amount of data, every calculation is shown to be hidden by the uploading and downloading of data except for the calculations used by Thrust. The row labeled “Default” represents the default stream and contains the Thrust kernels that were used to calculate the corresponding indices of the matrix that represents the found violations. Essentially, the only kernels that are contributing to the overall execution time are those of the Thrust kernels which are relatively minimal when compared to the rest of the program. One thing to note is the decision to download the possible violation information before it was known by the program that there were any violations. Slightly before the 1.74 second mark on the row
labeled “Memcpy (DtoH)” is where the violation information begins downloading. The Thrust kernels begin right after the downloading ends on the row labeled “Default” which is where it is determined if any violations have occurred or not. To be clear, the Thrust kernels are not finding violations, they are finding the number of violations, if any. The decision to download information before it was deemed necessary to download that information was done on purpose to allow for the downloads to run concurrently with the NPP kernels that are calculating violations. There would be no concurrency between the NPP kernels and the CUDA downloads if the downloading would occur after the Thrust kernels. Even if violations are rare occurrences, the amount of time added on to the overall execution time would still be minimal since most of the downloading would be hidden by the essential calculations performed by the NPP kernels located in the streams. In the case of Figure 6.5, most of the downloading is hidden by the calculation of stream 19. Having the downloads occur before the Thrust kernels basically makes those downloads invisible to begin with regardless if there are zero violations or several hundred violations.
Fig. 6.5 NVIDIA Visual Profiler results for the asynchronous single GPU version of the conflict detection algorithm

The execution times for the asynchronous single GPU program along with those of the synchronous single GPU version and the sequential CPU version are plotted in Figure 6.6. Not only did writing the program asynchronously improve its performance over that of the synchronous version, but the asynchronous single GPU version showed improvement over the sequential CPU version. However, the limited amount of global memory on the GPU is still an issue since the asynchronous version of the algorithm fails after 8,500 tracks and 850 geometries. Using a second GPU would help to overcome this obstacle, enabling the parallel version to further increase the number of tracks and geometries.
Asynchronous Multiple GPUs

Using multiple GPUs would not only create an additional layer of parallelism, but it could also help alleviate the restriction due to the limited global memory on GPUs. Thus, a third GPU parallel version of the algorithm was created to use two GPUs. This version is exactly the same as the asynchronous single GPU version, but this version has the first half of each array uploaded and executed on the Quadro FX 5800 and the last half of each array uploaded and executed on the Tesla C1060. Thus, the amount of data
being uploaded and allocated on each CUDA device is found by dividing the results of Equations 6.3 and 6.4 in half. The structure for this algorithm is essentially the same as that of the asynchronous single GPU version depicted in Figure 6.4 except that at each major portion of the algorithm (uploading, kernel execution, and downloading), the program will alternate between the two CUDA devices using the CUDA function \textit{cudaSetDevice}.

The execution times were recorded using the same method as that of all other asynchronous programs in this work, which involves CUDA event timers. The execution times for all versions are shown in Figure 6.7. From these results, it appears that the asynchronous multiple GPU version of the algorithm has the best performance, though the program fails after 12,000 tracks and 1,200 geometries. The extra GPU has enabled an increase in the number of tracks and geometries that the algorithm can process. However, it is still significantly lower than that of the sequential version. Note that starting around 8,000 tracks, the execution times begin exhibiting behavior similar to that of the asynchronous multiple GPU test programs depicted in Figure 5.21. Even with this fluctuation in execution times, the multiple GPU approach still outperforms the other versions of the conflict detection algorithm.
Fig. 6.7 Execution times for all versions of the conflict detection algorithm

Figure 6.8 displays a Visual Profiler screenshot of the asynchronous multiple GPU version of the parallel program. Surprisingly, this screenshot shows that the NPP kernels on one GPU are not running concurrently with the NPP kernels on the other GPU. This program was written in the same manner as that of the multiple GPU test program that uses the breadth first method, so it is unclear why concurrency in not taking place as in Figure 5.19. Although more research is needed to determine the underlying case, it seems to not be a disadvantage when compared to other versions of the conflict detection algorithm used in this report. Even though the NPP kernel executions on the Quadro FX 5800 do not occur concurrently with those on the Tesla C1060, the multiple GPU version
of the final program still appears to be the best performing version based on the results in Figure 6.7.

![Fig. 6.8 NVIDIA Visual Profiler results for the asynchronous multiple GPU version of the conflict detection algorithm](image)

Fig. 6.8 NVIDIA Visual Profiler results for the asynchronous multiple GPU version of the conflict detection algorithm
7. CONCLUSION

A parallel implementation of a conflict detection algorithm for application to airspace deconfliction has been developed and its performance compared to that of a sequential equivalent. The parallel implementation utilized GPUs residing on the NVIDIA Quadro FX 5800 and the Tesla C1060 graphics cards. The synchronous single GPU parallel version of the algorithm utilized data replication to enable the intersections of all tracks with all faces of all geometries to be computed in parallel. However, it had limitations regarding the amount of data that could be processed and did not exhibit performance improvement over the sequential counterpart in terms of execution time, due in large part to the overhead in data transfers to and from the GPU and the limited amount of memory on the GPU.

The synchronous single GPU parallel version was subsequently revised to enable the use of additional parallelism in an attempt to improve its performance. Specifically, the use of asynchronous programming using streams resulted in a significant performance improvement.

Another major performance improvement resulted from the use of multiple GPUs. Two GPUs were used in this analysis. The benefits included additional memory availability and added parallelism in computations. It is anticipated that further increasing the number of GPUs will accelerate this trend.
Optimizing the performance of the GPU is complex and involves a number of factors. The architecture of the GPU device is a critical factor in the equation. GPUs with compute capability of 2.0 or higher may be equipped with more than one copy engine, enabling uploading of data to the GPU to be done concurrently with downloading of data to the CPU. In addition, devices with compute capability 2.0 have higher overall resources, e.g., larger number of multiprocessors and larger maximum number of threads per block, thus enabling improved computing performance. Thus, upgrading to a GPU of compute capability 2.0 or higher (from the current 1.3) has the potential to drastically improve the results of the parallel version of the conflict detection code.

Sometimes the use of one library is preferable over another. In this study, the NPP library kernels were found to exhibit improved performance over those of CUBLAS kernels; however, further investigation is warranted.

The asynchronous multiple GPU parallel version of the conflict detection algorithm is not currently capable of running NPP kernels on one GPU concurrently with those on another. Fixing this bug in the program will also improve the execution times.
LIST OF REFERENCES
LIST OF REFERENCES


APPENDICES
A. CUDA LIBRARY TEST PROGRAMS AND RAW DATA

A.1 NPP Test Programs and Raw Data

Results for the NPP test programs were analyzed in Section 5.1.1. The columns in the raw data are represented by the titles listed in Table A.1.

Table A.1
Column representation for majority of the NPP raw data results

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
<th>Column 5</th>
<th>Column 6</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Rows and Columns</td>
<td>CPU Time in seconds</td>
<td>Quadro Time in seconds</td>
<td>Tesla Time in seconds</td>
<td>Quadro Percent Error</td>
<td>Tesla Percent Error</td>
</tr>
</tbody>
</table>

There are four NPP test programs that stored the raw data in a different format. For the NPP kernels `nppsDiv_32f`, `nppsDiv_32f_I`, `nppDivC_32f_I`, and `nppsSqrt_32f_I`, the columns in the raw data are represented by the titles listed in Table A.2.

Table A.2
Column representation for some of the NPP raw data results

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
<th>Column 5</th>
<th>Column 6</th>
<th>Column 7</th>
<th>Column 8</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Rows and Columns</td>
<td>CPU Time in seconds</td>
<td>Quadro Time in seconds</td>
<td>Tesla Time in seconds</td>
<td>Quadro Percent Error</td>
<td>Quadro Average Margin of Error</td>
<td>Tesla Percent Error</td>
<td>Tesla Average Margin of Error</td>
</tr>
</tbody>
</table>
A.1.1 nppiCompare_32f_C1R test program

// Compile and Execute (Make sure in correct directory):
// nvcc -I/usr/local/cuda-5.5/include -L/usr/local/cuda-5.5/lib64
// NPPICompare32fC1R_CPUCheck_Time.cu -lnpcc -lnpil -lnpps -lm

#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

int rows = 500;
int cols = 500;
int step = 500;
int PointBound = 100;
signal(SIGINT, signalHandler);
int i;
clock_t Start, End;
double Time;
int Number_of_Errors;
float Percent_Error;
while(1)
{
    srand(time(0));
    NppiSize sz;
    sz.height = rows;
    sz.width = cols;
    printf("Rows: %i\n", rows);
    fprintf(TimeFile, "%i ", rows);
    float* h_A;
    float* h_B;
    unsigned char* h_C;
    float* h_A0;
    float* h_B0;
    unsigned char* h_C0;
    float* h_A1;
float* h_B1;
unsigned char* h_C1;

//HOST ONLY VARIABLES.
h_A = (float*)malloc(rows*cols*sizeof(float));
if (h_A == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_A)\n");
    return EXIT_FAILURE;
}
h_B = (float*)malloc(rows*cols*sizeof(float));
if (h_B == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_B)\n");
    return EXIT_FAILURE;
}
h_C = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
if (h_C == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_C)\n");
    return EXIT_FAILURE;
}

//HOST VARIABLES USED FOR DEVICE 0.
h_A0 = (float*)malloc(rows*cols*sizeof(float));
if (h_A0 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_A0)\n");
    return EXIT_FAILURE;
}
h_B0 = (float*)malloc(rows*cols*sizeof(float));
if (h_B0 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_B0)\n");
    return EXIT_FAILURE;
}
h_C0 = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
if (h_C0 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_C0)\n");
    return EXIT_FAILURE;
}

//HOST VARIABLES USED FOR DEVICE 1.
h_A1 = (float*)malloc(rows*cols*sizeof(float));
if (h_A1 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_A1)\n");
    return EXIT_FAILURE;
}
h_B1 = (float*)malloc(rows*cols*sizeof(float));
if (h_B1 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_B1)\n");
    return EXIT_FAILURE;
}
h_C1 = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
if (h_C1 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_C1)\n");
    return EXIT_FAILURE;
}

//Creation of random values.
for (i=0; i < (rows*cols); i++)
{
    h_A[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_B[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
}

for (i=0; i < (rows*cols); i++)
{
    h_A0[i] = h_A[i];
    h_B0[i] = h_B[i];
    h_A1[i] = h_A[i];
    h_B1[i] = h_B[i];
Start Time for CPU Calculations.

```c
Start = clock();
```

//CPU Calculations

```c
//h_A <<Compare>> h_B
//  If TRUE, then h_C = 255
//  If FALSE, then h_C = 0
//for (i=0; i < (rows*cols); i++)
//  if (h_A[i] > h_B[i])
//    h_C[i] = 255;
//  else
//    h_C[i] = 0;
```

End Time for CPU Calculations.
```c
End = clock();
```
```c
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
```

Start Time for GPU Quadro FX 5800 Calculations.
```c
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();
```
```c
float* d_A0;
float* d_B0;
unsigned char* d_C0;
```
```c
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A0, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_B0, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_C0, rows*cols*sizeof(unsigned char)) );
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B0, h_B0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
```
```c
//Compare Elements of Two Matrices (nppiCompare_32f_C1R)
//
//d_A <<Compare>> d_B
//  If TRUE, then d_C = 255
//  If FALSE, then d_C = 0
//nppiCompare_32f_C1R(d_A0, cols*sizeof(float), d_B0, cols*sizeof(float), d_C0, cols*sizeof(unsigned char), sz, NPP_CMP_GREATER);
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_C0, d_C0, rows*cols*sizeof(unsigned char), cudaMemcpyDeviceToHost) );
```

End Time for GPU Calculations.
```c
cudaDeviceSynchronize();
End = clock();
```
```c
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
```

Start Time for GPU Tesla C1060 Calculations.
```c
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();
```
float* d_A1;
float* d_B1;
unsigned char* d_C1;

// Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A1, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_B1, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_C1, rows*cols*sizeof(unsigned char)) );

// Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B1, h_B1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );

// Compare Elements of Two Matrices (nppiCompare_32f_C1R)
//
// d_A <<Compare>> d_B
//  If TRUE, then d_C = 255
//  If FALSE, then d_C = 0
//
nppiCompare_32f_C1R(d_A1, cols*sizeof(float), d_B1, cols*sizeof(float), d_C1,
// cols*sizeof(unsigned char), sz, NPP_CMP_GREATER);

// Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_C1, d_C1, rows*cols*sizeof(unsigned char), cudaMemcpyDeviceToHost) );

// End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-------------------------------------------------------------------------------------//
// Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
    if (h_C1[i] != h_C0[i])
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\n", Percent_Error);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
    if (h_C[i] != h_C0[i])
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f\n", Percent_Error);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
printf("Tesla C1060 Error Percentage: %.3f\n\n", Percent_Error);
// HOST MEMORY CLEAN UP.
free(h_A);
free(h_B);
free(h_C);
free(h_A0);
free(h_B0);
free(h_C0);
free(h_A1);
free(h_B1);
free(h_C1);
// DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
CudaErrCheck( cudaFree(d_B0) );
CudaErrCheck( cudaFree(d_C0) );

// DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
CudaErrCheck( cudaFree(d_B1) );
CudaErrCheck( cudaFree(d_C1) );

//-----------------------------------------------------------------------------//
// Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;
//-----------------------------------------------------------------------------//

} // END PROGRAM

A.1.2  nppiCompare_32f_C1R raw data

500  0.000  1.190  1.130  0.000  0.000
1000  0.010  0.010  0.000  0.000  0.000
1500  0.020  0.000  0.010  0.000  0.000
2000  0.040  0.010  0.010  0.000  0.000
2500  0.060  0.010  0.020  0.000  0.000
3000  0.090  0.030  0.020  0.000  0.000
3500  0.110  0.030  0.040  0.000  0.000
4000  0.150  0.040  0.040  0.000  0.000
4500  0.190  0.040  0.060  0.000  0.000
5000  0.250  0.060  0.070  0.000  0.000
5500  0.280  0.090  0.080  0.000  0.000
6000  0.340  0.100  0.090  0.000  0.000
6500  0.400  0.120  0.110  0.000  0.000
7000  0.480  0.120  0.130  0.000  0.000
7500  0.530  0.150  0.140  0.000  0.000
8000  0.610  0.160  0.160  0.000  0.000
8500  0.690  0.190  0.180  0.000  0.000
9000  0.770  0.220  0.200  0.000  0.000
9500  0.860  0.240  0.230  0.000  0.000
10000  0.960  0.250  0.250  0.000  0.000
10500  1.050  0.380  0.390  0.000  0.000
11000  1.150  0.420  0.430  0.000  0.000
11500  1.260  0.470  0.470  0.000  0.000
12000  1.380  0.510  0.510  0.000  0.000
12500  1.490  0.560  0.560  0.000  0.000
13000  1.620  0.600  0.610  0.000  0.000
13500  1.730  0.660  0.660  0.000  0.000
14000  1.870  0.690  0.690  0.000  0.000
14500  2.010  0.760  0.760  0.000  0.000
15000  2.150  0.810  0.810  0.000  0.000
15500  2.290  0.870  0.870  0.000  0.000
16000  2.450  0.900  0.900  0.000  0.000
16500  2.600  0.930  0.930  0.000  0.000
17000  2.750  0.980  0.980  0.000  0.000
17500  2.920  1.050  1.040  0.000  0.000
18000  3.090  1.090  1.070  0.000  0.000
18500  3.260  1.170  1.150  0.000  0.000
19000  3.440  1.210  1.210  0.000  0.000
19500  3.610  1.260  1.340  0.000  0.000
20000  3.820  1.380  1.610  0.000  0.000
20500  4.010  1.650  1.740  0.000  0.000
21000  4.220  1.770  1.770  0.000  0.000
A.1.3 nppiCompareC_32f_C1R test program

A.1.3 nppiCompareC_32f_C1R test program

//NPPICompareC32fC1R_CPUCheck_Time.cu
//Nathan Clem

//Compile and Execute (Make sure in correct directory):
// nvcc -I/usr/local/cuda-5.5/include -L/usr/local/cuda-5.5/lib64
//NPPICompareC32fC1R_CPUCheck_Time.cu -lnppc -lnppi -lnpps -lm
// ./a.out

#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

//Defined Constant.
#define Constant 0

//File to store CPU and GPU run times.
FILE *TimeFile = fopen("NPPICompareC32fC1R_Times.txt", "w");

//Use to stop program in case of infinite loop
//Enter: "CTRL + C"
void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

//Cuda Error Check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
    if (code != cudaSuccess)
    {
        fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}

//Main
int main(int argc, char** argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    int PointBound = 100;
    signal(SIGINT, signalHandler);
    int i;
    clock_t Start, End;
    double Time;
    int Number_of_Errors;
    float Percent_Error;
    while(1)
    {
        srand(time(0));
        NppiSize sz;
        sz.height = rows;
        sz.width = cols;
        printf("Rows: %i\n", rows);
        printf("Columns: %i\n", cols);
        fprintf(TimeFile, "%i\n", rows);
        float* h_A;
        unsigned char* h_C;
        float* h_A0;
        unsigned char* h_C0;
float* h_A1;
unsigned char* h_C1;

// HOST ONLY VARIABLES.
float* h_A = (float*)malloc(rows*cols*sizeof(float));
if (h_A == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_A)\n");
    return EXIT_FAILURE;
}

unsigned char* h_C = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
if (h_C == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_C)\n");
    return EXIT_FAILURE;
}

// HOST VARIABLES USED FOR DEVICE 0.
float* h_A0 = (float*)malloc(rows*cols*sizeof(float));
if (h_A0 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_A0)\n");
    return EXIT_FAILURE;
}

unsigned char* h_C0 = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
if (h_C0 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_C0)\n");
    return EXIT_FAILURE;
}

// HOST VARIABLES USED FOR DEVICE 1.
float* h_A1 = (float*)malloc(rows*cols*sizeof(float));
if (h_A1 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_A1)\n");
    return EXIT_FAILURE;
}

unsigned char* h_C1 = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
if (h_C1 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_C1)\n");
    return EXIT_FAILURE;
}

// Creation of random values.
for (i=0; i < (rows*cols); i++)
{
    h_A[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
}

for (i=0; i < (rows*cols); i++)
{
    h_A0[i] = h_A[i];
    h_A1[i] = h_A[i];
}

// Start Time for CPU Calculations.
Start = clock();

// CPU Calculations

// h_A <<Compare>> Constant
// If TRUE, then h_C = 255
// If FALSE, then h_C = 0

for (i=0; i < (rows*cols); i++)
{
    if (h_A[i] > Constant)
    {
        h_C[i] = 255;
    }
    else
    {
        h_C[i] = 0;
    }
End Time for CPU Calculations.
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));

Start Time for GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();
float* d_A0;
unsigned char* d_C0;

Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A0, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_C0, rows*cols*sizeof(unsigned char)) );

//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );

//Compare Elements of One Matrix Against a Constant (nppiCompareC_32f_C1R)

//d_A <<Compare>> Constant
// If TRUE, then d_C = 255
// If FALSE, then d_C = 0
nppiCompareC_32f_C1R(d_A0, cols*sizeof(float), Constant, d_C0, cols*sizeof(unsigned char), sz, NPP_CMP_GREATER);

//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_C0, d_C0, rows*cols*sizeof(unsigned char), cudaMemcpyDeviceToHost) );

End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));

Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();
float* d_A1;
unsigned char* d_C1;

Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A1, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_C1, rows*cols*sizeof(unsigned char)) );

//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );

//Compare Elements of One Matrix Against a Constant (nppiCompareC_32f_C1R)

//d_A <<Compare>> Constant
// If TRUE, then d_C = 255
// If FALSE, then d_C = 0
nppiCompareC_32f_C1R(d_A1, cols*sizeof(float), Constant, d_C1, cols*sizeof(unsigned char), sz, NPP_CMP_GREATER);

//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_C1, d_C1, rows*cols*sizeof(unsigned char), cudaMemcpyDeviceToHost) );

End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------//
//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
  if (h_C[i] != h_C0[i])
  {
    Number_of_Errors++;
  }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\%
", Percent_Error);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
  if (h_C[i] != h_C1[i])
  {
    Number_of_Errors++;
  }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f\n", Percent_Error);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
printf("Tesla C1060 Error Percentage: %.3f\%

", Percent_Error);
//-----------------------------------------------------------------------------//
//HOST MEMORY CLEAN UP.
free(h_A);
free(h_C);
free(h_A0);
free(h_C0);
free(h_A1);
free(h_C1);
//DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
CudaErrCheck( cudaFree(d_C0) );
//DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
CudaErrCheck( cudaFree(d_C1) );
//-----------------------------------------------------------------------------//
//Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;
//-----------------------------------------------------------------------------//
A.1.4 nppiCompareC_32f_C1R raw data

500 0.010 1.210 1.120 0.000 0.000
1000 0.010 0.000 0.000 0.000 0.000
1500 0.020 0.010 0.000 0.000 0.000
2000 0.040 0.000 0.010 0.000 0.000
2500 0.060 0.010 0.010 0.000 0.000
3000 0.090 0.010 0.010 0.000 0.000
3500 0.110 0.010 0.020 0.000 0.000
4000 0.150 0.020 0.030 0.000 0.000
4500 0.180 0.030 0.020 0.000 0.000
A.1.5  nppsAdd_8u_ISfs test program

//----------------------------------------------------------------------------------------------------------------------------------------------
//NPPSAdd8uISfs_CPUCheck_Time.cu
//Nathan Clem
//
//Compile and Execute (Make sure in correct directory):
// nvcc -I/usr/local/cuda-5.5/include -L/usr/local/cuda-5.5/lib64
//NPPSAdd8uISfs_CPUCheck_Time.cu -lnppc -lnppi -lnpps -lm
// ./a.out
//----------------------------------------------------------------------------------------------------------------------------------------------
#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>
//----------------------------------------------------------------------------------------------------------------------------------------------
//File to store CPU and GPU run times.
FILE *TimeFile = fopen("NPPSAdd8uISfs_Times.txt", "w");
//----------------------------------------------------------------------------------------------------------------------------------------------
//Use to stop program in case of infinite loop
//Enter: "CTRL + C"
void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

//Cuda Error Check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
    if (code != cudaSuccess)
    {
        fprintf(stderr,"GPUassert: %s %s \d\n", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}

static const char alphanum[] =
"0123456789"
"!@#$%^&*"
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
"abcdefghijklmnopqrstuvwxyz";

int stringLength = sizeof(alphanum) - 1;
char genRandom() // Random string generator function.
{
    return alphanum[rand() % stringLength];
}

//Main
int main(int argc, char** argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    signal(SIGINT, signalHandler);
    int i;
    clock_t Start, End;
    double Time;
    int Number_of_Errors;
    float Percent_Error;
    while(1)
    {
        srand(time(0));
        printf("Rows: %i\n", rows);
        printf("Columns: %i\n", cols);
        fprintf(TimeFile, "%i \n", rows);
        unsigned char* h_A;
        unsigned char* h_B;
        unsigned char* h_A0;
        unsigned char* h_B0;
        unsigned char* h_A1;
        unsigned char* h_B1;
        //
        //HOST ONLY VARIABLES.
        h_A = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
        if (h_A == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A)\n");
            return EXIT_FAILURE;
        }
        h_B = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
        if (h_B == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_B)\n");
            return EXIT_FAILURE;
        }
HOST VARIABLES USED FOR DEVICE 0.

\[ h_{A0} = (\text{unsigned char}*) \text{malloc}(\text{rows} \times \text{cols} \times \text{sizeof(unsigned char)}) \]
if (h_{A0} == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_{A0})\n");
    return EXIT_FAILURE;
}

\[ h_{B0} = (\text{unsigned char}*) \text{malloc}(\text{rows} \times \text{cols} \times \text{sizeof(unsigned char)}) \]
if (h_{B0} == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_{B0})\n");
    return EXIT_FAILURE;
}

HOST VARIABLES USED FOR DEVICE 1.

\[ h_{A1} = (\text{unsigned char}*) \text{malloc}(\text{rows} \times \text{cols} \times \text{sizeof(unsigned char)}) \]
if (h_{A1} == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_{A1})\n");
    return EXIT_FAILURE;
}

\[ h_{B1} = (\text{unsigned char}*) \text{malloc}(\text{rows} \times \text{cols} \times \text{sizeof(unsigned char)}) \]
if (h_{B1} == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_{B1})\n");
    return EXIT_FAILURE;
}

for (i=0; i<(\text{rows} \times \text{cols}); i++)
{
    h_{A}[i] = \text{genRandom()};
    h_{B}[i] = \text{genRandom()};
}

for (i=0; i < (\text{rows} \times \text{cols}); i++)
{
    h_{A0}[i] = h_{A}[i];
    h_{B0}[i] = h_{B}[i];
    h_{A1}[i] = h_{A}[i];
    h_{B1}[i] = h_{B}[i];
}

---

Start Time for CPU Calculations.

Start = \text{clock();}

//CPU Calculations

\[ \text{h}_B = \text{h}_A + \text{h}_B \]
//
for (i=0; i < (\text{rows} \times \text{cols}); i++)
{
    \text{h}_B[i] = \text{h}_A[i] + \text{h}_B[i];
}

End Time for CPU Calculations.

End = \text{clock();}

Time = (\text{End-Start})/\text{CLOCKS_PER_SEC};

\text{fprintf(TimeFile, "%.3f ", \text{fabs(Time);}}

---

Start Time for GPU Quadro FX 5800 Calculations.

cudaSetDevice(0);
cudaDeviceSynchronize();

Start = \text{clock();}

unsigned char* \text{d}_A0;
unsigned char* \text{d}_B0;

Allocate Device Memory (GPU).

cudaMemcpy(d_{A0}, \text{h}_{A0}, \text{rows} \times \text{cols} \times \text{sizeof(unsigned char)});
cudaMemcpy(d_{B0}, \text{h}_{B0}, \text{rows} \times \text{cols} \times \text{sizeof(unsigned char)});

Transfer Data from HOST to DEVICE.

cudaMemcpy(d_{A0}, \text{h}_{A0}, \text{rows} \times \text{cols} \times \text{sizeof(unsigned char)});
cudaMemcpyHostToDevice);
CudaErrCheck( cudaMemcpy(d_B0, h_B0, rows*cols*sizeof(unsigned char), cudaMemcpyHostToDevice) );
//
//Addition Inplace 8 Bit (nppsAdd_8u_ISfs)
//
//d_B = d_A + d_B
//nppsAdd_8u_ISfs(d_A0, d_B0, rows*cols, 0);
//
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_B0, d_B0, rows*cols*sizeof(unsigned char), cudaMemcpyDeviceToHost) );
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//---------------------------------------------------------------
//Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();
unsigned char* d_A1;
unsigned char* d_B1;
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A1, rows*cols*sizeof(unsigned char)) );
CudaErrCheck( cudaMalloc((void **)&d_B1, rows*cols*sizeof(unsigned char)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(unsigned char), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B1, h_B1, rows*cols*sizeof(unsigned char), cudaMemcpyHostToDevice) );
//
//Addition Inplace 8 Bit (nppsAdd_8u_ISfs)
//
//d_B = d_A + d_B
//nppsAdd_8u_ISfs(d_A1, d_B1, rows*cols, 0);
//
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_B1, d_B1, rows*cols*sizeof(unsigned char), cudaMemcpyDeviceToHost) );
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//---------------------------------------------------------------
//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
   if ((h_B[i] - h_B0[i]) != 0)
   {
      Number_of_Errors++;
   }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\n", Percent_Error);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
   if ((h_B[i] - h_B1[i]) != 0)
A.1.6  nppsAdd_8u_ISfs raw data

<table>
<thead>
<tr>
<th>Value</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
<th>Column 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1500</td>
<td>0.010</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2000</td>
<td>0.010</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2500</td>
<td>0.030</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3000</td>
<td>0.040</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3500</td>
<td>0.060</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>4000</td>
<td>0.060</td>
<td>0.020</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>4500</td>
<td>0.090</td>
<td>0.020</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>5000</td>
<td>0.110</td>
<td>0.020</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>5500</td>
<td>0.130</td>
<td>0.020</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>6000</td>
<td>0.150</td>
<td>0.020</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>6500</td>
<td>0.180</td>
<td>0.030</td>
<td>0.040</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>7000</td>
<td>0.200</td>
<td>0.040</td>
<td>0.050</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>7500</td>
<td>0.230</td>
<td>0.040</td>
<td>0.050</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>8000</td>
<td>0.270</td>
<td>0.040</td>
<td>0.050</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>8500</td>
<td>0.300</td>
<td>0.060</td>
<td>0.080</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>9000</td>
<td>0.340</td>
<td>0.060</td>
<td>0.080</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>9500</td>
<td>0.370</td>
<td>0.080</td>
<td>0.080</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10000</td>
<td>0.410</td>
<td>0.070</td>
<td>0.080</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10500</td>
<td>0.460</td>
<td>0.080</td>
<td>0.080</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>11000</td>
<td>0.510</td>
<td>0.100</td>
<td>0.100</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>11500</td>
<td>0.550</td>
<td>0.110</td>
<td>0.110</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>12000</td>
<td>0.600</td>
<td>0.110</td>
<td>0.110</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>12500</td>
<td>0.650</td>
<td>0.120</td>
<td>0.120</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>13000</td>
<td>0.710</td>
<td>0.130</td>
<td>0.130</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>13500</td>
<td>0.760</td>
<td>0.140</td>
<td>0.140</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>14000</td>
<td>0.810</td>
<td>0.150</td>
<td>0.160</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>14500</td>
<td>0.870</td>
<td>0.160</td>
<td>0.160</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>15000</td>
<td>0.930</td>
<td>0.180</td>
<td>0.180</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>15500</td>
<td>1.000</td>
<td>0.180</td>
<td>0.180</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>16000</td>
<td>1.060</td>
<td>0.200</td>
<td>0.190</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>16500</td>
<td>1.120</td>
<td>0.220</td>
<td>0.190</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>Value</td>
<td>Time</td>
<td>Width</td>
<td>Height</td>
<td>Depth</td>
<td>Width</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
<td>--------</td>
<td>--------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>1700</td>
<td>1.190</td>
<td>0.230</td>
<td>0.220</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1750</td>
<td>1.260</td>
<td>0.240</td>
<td>0.220</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1800</td>
<td>1.340</td>
<td>0.250</td>
<td>0.220</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1850</td>
<td>1.410</td>
<td>0.270</td>
<td>0.220</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1900</td>
<td>1.490</td>
<td>0.280</td>
<td>0.240</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1950</td>
<td>1.580</td>
<td>0.290</td>
<td>0.250</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2000</td>
<td>1.650</td>
<td>0.310</td>
<td>0.270</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2050</td>
<td>1.740</td>
<td>0.320</td>
<td>0.290</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2100</td>
<td>1.830</td>
<td>0.340</td>
<td>0.310</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2150</td>
<td>1.920</td>
<td>0.360</td>
<td>0.320</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2200</td>
<td>2.010</td>
<td>0.380</td>
<td>0.330</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2250</td>
<td>2.090</td>
<td>0.400</td>
<td>0.350</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2300</td>
<td>2.190</td>
<td>0.410</td>
<td>0.360</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2350</td>
<td>2.280</td>
<td>0.430</td>
<td>0.370</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2400</td>
<td>2.380</td>
<td>0.450</td>
<td>0.390</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2450</td>
<td>2.490</td>
<td>0.470</td>
<td>0.410</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2500</td>
<td>2.600</td>
<td>0.490</td>
<td>0.430</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2550</td>
<td>2.690</td>
<td>0.500</td>
<td>0.450</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2600</td>
<td>2.800</td>
<td>0.520</td>
<td>0.470</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2650</td>
<td>2.910</td>
<td>0.540</td>
<td>0.490</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2700</td>
<td>3.020</td>
<td>0.560</td>
<td>0.510</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2750</td>
<td>3.130</td>
<td>0.580</td>
<td>0.530</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2800</td>
<td>3.250</td>
<td>0.600</td>
<td>0.550</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2850</td>
<td>3.380</td>
<td>0.630</td>
<td>0.570</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2900</td>
<td>3.470</td>
<td>0.670</td>
<td>0.590</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2950</td>
<td>3.620</td>
<td>0.690</td>
<td>0.610</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3000</td>
<td>3.720</td>
<td>0.710</td>
<td>0.630</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3050</td>
<td>3.850</td>
<td>0.730</td>
<td>0.650</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3100</td>
<td>3.980</td>
<td>0.750</td>
<td>0.670</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3150</td>
<td>4.100</td>
<td>1.060</td>
<td>1.060</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>

### A.1.7 nppsAdd_8u_Sfs test program

```c
#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

// File to store CPU and GPU run times.
FILE *TimeFile = fopen("NPPSAdd8uSfs_Times.txt", "w");
```

```c
void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}
```

```c
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__);} 
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
    if (code != cudaSuccess)
    {
        fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}```
static const char alphanum[] =
"0123456789"
"!@#$%^&*"
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
"abcdefghijklmnopqrstuvwxyz";

int stringLength = sizeof(alphanum) - 1;

char genRandom() // Random string generator function.
{
    return alphanum[rand() % stringLength];
}

//Main
int main(int argc, char** argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    signal(SIGINT, signalHandler);
    int i;
    clock_t Start, End;
    double Time;
    int Number_of_Errors;
    float Percent_Error;
    while(1)
    {
        srand(time(0));
        printf("Rows: %i\n", rows);
        printf("Columns: %i\n", cols);
        fprintf(TimeFile, "%i ", rows);
        //HOST ONLY VARIABLES.
        h_A = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
        if (h_A == 0)
        {
            fprintf(stderr, "!!!!! host memory allocation error (h_A)\n");
            return EXIT_FAILURE;
        }
        h_B = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
        if (h_B == 0)
        {
            fprintf(stderr, "!!!!! host memory allocation error (h_B)\n");
            return EXIT_FAILURE;
        }
        h_C = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
        if (h_C == 0)
        {
            fprintf(stderr, "!!!!! host memory allocation error (h_C)\n");
            return EXIT_FAILURE;
        }
        //HOST VARIABLES USED FOR DEVICE 0.
        h_A0 = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
        if (h_A0 == 0)
        {
            fprintf(stderr, "!!!!! host memory allocation error (h_A0)\n");
            return EXIT_FAILURE;
        }
        h_B0 = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
        if (h_B0 == 0)
fprintf (stderr, "!!!! host memory allocation error (h_C0)\n");
return EXIT_FAILURE;
}

h_C0 = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
if (h_C0 == 0)
{
fprintf (stderr, "!!!! host memory allocation error (h_C0)\n");
return EXIT_FAILURE;
}

//HOST VARIABLES USED FOR DEVICE 1.

h_A1 = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
if (h_A1 == 0)
{
fprintf (stderr, "!!!! host memory allocation error (h_A1)\n");
return EXIT_FAILURE;
}

h_B1 = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
if (h_B1 == 0)
{
fprintf (stderr, "!!!! host memory allocation error (h_B1)\n");
return EXIT_FAILURE;
}

h_C1 = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
if (h_C1 == 0)
{
fprintf (stderr, "!!!! host memory allocation error (h_C1)\n");
return EXIT_FAILURE;
}

for (i=0; i<(rows*cols); i++)
{
    h_A[i] = genRandom();
    h_B[i] = genRandom();
}

for (i=0; i < (rows*cols); i++)
{
    h_A0[i] = h_A[i];
    h_B0[i] = h_B[i];
    h_A1[i] = h_A[i];
    h_B1[i] = h_B[i];
}

ATTERY FOR GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();

unsigned char* d_A0;
unsigned char* d_B0;
unsigned char* d_C0;

//Allocate Device Memory (GPU).
CudaErrCheck( cudaMemcpy(void **&d_A0, rows*cols*sizeof(unsigned char)) );
CudaErrCheck( cudaMemcpy(void **&d_B0, rows*cols*sizeof(unsigned char)) );
CudaErrCheck( cudaMemcpy(void **&d_C0, rows*cols*sizeof(unsigned char)) );
// Transfer Data from HOST to DEVICE.
CudaErrCheck(cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(unsigned char), cudaMemcpyHostToDevice));
CudaErrCheck(cudaMemcpy(d_B0, h_B0, rows*cols*sizeof(unsigned char), cudaMemcpyHostToDevice));

// Addition 8 Bit (nppsAdd_8u_Sfs)
//
// d_C = d_A + d_B
//
nppsAdd_8u_Sfs(d_A0, d_B0, d_C0, rows*cols, 0);

// Transfer Data from DEVICE to HOST.
CudaErrCheck(cudaMemcpy(h_C0, d_C0, rows*cols*sizeof(unsigned char), cudaMemcpyDeviceToHost));

// End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));

// Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();

unsigned char* d_A1;
unsigned char* d_B1;
unsigned char* d_C1;

// Allocate Device Memory (GPU).
CudaErrCheck(cudaMalloc((void**)&d_A1, rows*cols*sizeof(unsigned char)));
CudaErrCheck(cudaMalloc((void**)&d_B1, rows*cols*sizeof(unsigned char)));
CudaErrCheck(cudaMalloc((void**)&d_C1, rows*cols*sizeof(unsigned char)));

// Transfer Data from HOST to DEVICE.
CudaErrCheck(cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(unsigned char), cudaMemcpyHostToDevice));
CudaErrCheck(cudaMemcpy(d_B1, h_B1, rows*cols*sizeof(unsigned char), cudaMemcpyHostToDevice));

// Addition 8 Bit (nppsAdd_8u_Sfs)
//
// d_C = d_A + d_B
//
nppsAdd_8u_Sfs(d_A1, d_B1, d_C1, rows*cols, 0);

// Transfer Data from DEVICE to HOST.
CudaErrCheck(cudaMemcpy(h_C1, d_C1, rows*cols*sizeof(unsigned char), cudaMemcpyDeviceToHost));

// End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));

// Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
    if (((h_C[i] - h_C0[i]) != 0))
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\%
", Percent_Error);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{    
    if ((h_C[i] - h_C1[i]) != 0)
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f\n", Percent_Error);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
printf("Tesla C1060 Error Percentage: %.3f\%\n\n", Percent_Error);
//-----------------------------------------------------------------------------//
//HOST MEMORY CLEAN UP.
free(h_A);
free(h_B);
free(h_C);
free(h_A0);
free(h_B0);
free(h_C0);
free(h_A1);
free(h_B1);
free(h_C1);
//DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
CudaErrCheck( cudaFree(d_B0) );
CudaErrCheck( cudaFree(d_C0) );
//DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
CudaErrCheck( cudaFree(d_B1) );
CudaErrCheck( cudaFree(d_C1) );
//-----------------------------------------------------------------------------//
//Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;
//-----------------------------------------------------------------------------//
///END PROGRAM
return EXIT_SUCCESS;
//-----------------------------------------------------------------------------//

A.1.8  nppsAdd_8u_Sfs raw data

500 0.000 0.070 0.050 0.000 0.000
1000 0.010 0.000 0.000 0.000 0.000
1500 0.010 0.000 0.020 0.000 0.000
2000 0.020 0.000 0.000 0.000 0.000
2500 0.030 0.010 0.010 0.000 0.000
3000 0.030 0.020 0.010 0.000 0.000
3500 0.050 0.010 0.010 0.000 0.000
4000 0.080 0.010 0.020 0.000 0.000
4500 0.090 0.010 0.030 0.000 0.000
5000 0.110 0.020 0.030 0.000 0.000
5500 0.130 0.030 0.030 0.000 0.000
6000 0.160 0.040 0.020 0.000 0.000
6500 0.190 0.030 0.040 0.000 0.000
7000 0.210 0.040 0.040 0.000 0.000
7500 0.250 0.050 0.050 0.000 0.000
8000 0.280 0.060 0.060 0.000 0.000
8500 0.320 0.060 0.060 0.000 0.000
9000 0.350 0.070 0.060 0.000 0.000
9500 0.390 0.090 0.070 0.000 0.000
10000 0.440 0.090 0.090 0.000 0.000
10500 0.480 0.100 0.100 0.000 0.000
A.1.9 nppsAdd_32f_I test program

// File to store CPU and GPU run times.
FILE *TimeFile = fopen("NPPSAdd32fI_Times.txt", "w");
// Use to stop program in case of infinite loop
// Enter: "CTRL + C"
void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
}
exit(signum);
}  
// Cuda Error Check  
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }  
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)  
{  
    if (code != cudaSuccess)  
    {  
        fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);  
        if (abort) exit(code);  
    }  
}  
// Main  
int main(int argc, char** argv)  
{  
    int rows = 500;  
    int cols = 500;  
    int step = 500;  
    int PointBound = 100;  
    signal(SIGINT, signalHandler);  
    int i;  
    clock_t Start, End;  
    double Time;  
    int Number_of_Errors;  
    float Percent_Error;  
    while(1)  
    {  
        srand(time(0));  
        printf("Rows: %i\n", rows);  
        printf("Columns: %i\n", cols);  
        fprintf(TimeFile, "%i \n", rows);  
    // HOST ONLY VARIABLES.  
    h_A = (float*)malloc(rows*cols*sizeof(float));  
    if (h_A == 0)  
    {  
        fprintf (stderr, "!!!! host memory allocation error (h_A)\n");  
        return EXIT_FAILURE;  
    }  
    h_B = (float*)malloc(rows*cols*sizeof(float));  
    if (h_B == 0)  
    {  
        fprintf (stderr, "!!!! host memory allocation error (h_B)\n");  
        return EXIT_FAILURE;  
    }  
    // HOST VARIABLES USED FOR DEVICE 0.  
    h_A0 = (float*)malloc(rows*cols*sizeof(float));  
    if (h_A0 == 0)  
    {  
        fprintf (stderr, "!!!! host memory allocation error (h_A0)\n");  
        return EXIT_FAILURE;  
    }  
    h_B0 = (float*)malloc(rows*cols*sizeof(float));  
    if (h_B0 == 0)  
    {  
        fprintf (stderr, "!!!! host memory allocation error (h_B0)\n");  
        return EXIT_FAILURE;  
    }  
    // HOST VARIABLES USED FOR DEVICE 1.  
    h_A1 = (float*)malloc(rows*cols*sizeof(float));  
    if (h_A1 == 0)  
    {  
        fprintf (stderr, "!!!! host memory allocation error (h_A1)\n");  
        return EXIT_FAILURE;  
    }  
    // HOST VARIABLES USED FOR DEVICE 2.  
    h_A2 = (float*)malloc(rows*cols*sizeof(float));  
    if (h_A2 == 0)  
    {  
        fprintf (stderr, "!!!! host memory allocation error (h_A2)\n");  
        return EXIT_FAILURE;  
    }  
}  
// Main
return EXIT_FAILURE;
}
h_B1 = (float*)malloc(rows*cols*sizeof(float));
if (h_B1 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_B1)\n");
    return EXIT_FAILURE;
}
//Creation of random values.
for (i=0; i < (rows*cols); i++)
{
    h_A[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_B[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
}
for (i=0; i < (rows*cols); i++)
{
    h_A0[i] = h_A[i];
    h_B0[i] = h_B[i];
    h_A1[i] = h_A[i];
    h_B1[i] = h_B[i];
}
//-----------------------------------------------------------------------------//
//Start Time for CPU Calculations.
Start = clock();
//
//CPU Calculations
//
//h_B = h_A + h_B
//
for (i=0; i < (rows*cols); i++)
{
    h_B[i] = h_A[i] + h_B[i];
}
//
//End Time for CPU Calculations.
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------//
//Start Time for GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A0;
float* d_B0;
//
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A0, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_B0, rows*cols*sizeof(float)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B0, h_B0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
//
//Addition Inplace (nppsAdd_32f_I)
//
//d_B = d_A + d_B
//
nppsAdd_32f_I(d_A0, d_B0, rows*cols);
//
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_B0, d_B0, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();

float* d_A1;
float* d_B1;

Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A1, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_B1, rows*cols*sizeof(float)) );

Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B1, h_B1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );

Addition Inplace (nppsAdd_32f_I)
   d_B = d_A + d_B

Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_B1, d_B1, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );

End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));

Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
   if ((h_B[i] - h_B0[i]) != 0)
   {
      Number_of_Errors++;
   }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\n", Percent_Error);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
   if ((h_B[i] - h_B1[i]) != 0)
   {
      Number_of_Errors++;
   }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f\n", Percent_Error);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
printf("Tesla C1060 Error Percentage: %.3f\n\n", Percent_Error);

HOST MEMORY CLEAN UP.
free(h_A);
free(h_B);
free(h_A0);
free(h_B0);
free(h_A1);
free(h_B1);

DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
CudaErrCheck( cudaFree(d_B0) );
//DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
CudaErrCheck( cudaFree(d_B1) );
//-----------------------------------------------------------------------------//
//Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;
//-----------------------------------------------------------------------------//
}
////////////////////////////////////////////////////////////////////////
//END PROGRAM
return EXIT_SUCCESS;
////////////////////////////////////////////////////////////////////////

A.1.10 nppsAdd_32f_I raw data

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.000</td>
<td>0.060</td>
<td>0.070</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>0.000</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1500</td>
<td>0.010</td>
<td>0.010</td>
<td>0.020</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2000</td>
<td>0.020</td>
<td>0.010</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2500</td>
<td>0.030</td>
<td>0.020</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3000</td>
<td>0.040</td>
<td>0.030</td>
<td>0.020</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3500</td>
<td>0.050</td>
<td>0.040</td>
<td>0.040</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>4000</td>
<td>0.070</td>
<td>0.050</td>
<td>0.040</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>4500</td>
<td>0.090</td>
<td>0.060</td>
<td>0.050</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>5000</td>
<td>0.110</td>
<td>0.070</td>
<td>0.070</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>5500</td>
<td>0.130</td>
<td>0.090</td>
<td>0.080</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>6000</td>
<td>0.160</td>
<td>0.090</td>
<td>0.090</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>6500</td>
<td>0.180</td>
<td>0.120</td>
<td>0.110</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>7000</td>
<td>0.210</td>
<td>0.140</td>
<td>0.120</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>7500</td>
<td>0.250</td>
<td>0.150</td>
<td>0.150</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>8000</td>
<td>0.280</td>
<td>0.180</td>
<td>0.160</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>8500</td>
<td>0.310</td>
<td>0.210</td>
<td>0.180</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>9000</td>
<td>0.350</td>
<td>0.220</td>
<td>0.220</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>9500</td>
<td>0.400</td>
<td>0.250</td>
<td>0.240</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10000</td>
<td>0.440</td>
<td>0.270</td>
<td>0.260</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10500</td>
<td>0.480</td>
<td>0.310</td>
<td>0.290</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>11000</td>
<td>0.530</td>
<td>0.330</td>
<td>0.320</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>11500</td>
<td>0.580</td>
<td>0.370</td>
<td>0.340</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>12000</td>
<td>0.630</td>
<td>0.400</td>
<td>0.380</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>12500</td>
<td>0.700</td>
<td>0.430</td>
<td>0.400</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>13000</td>
<td>0.750</td>
<td>0.460</td>
<td>0.450</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>13500</td>
<td>0.800</td>
<td>0.500</td>
<td>0.480</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>14000</td>
<td>0.860</td>
<td>0.550</td>
<td>0.500</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>14500</td>
<td>0.920</td>
<td>0.580</td>
<td>0.550</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>15000</td>
<td>0.990</td>
<td>0.610</td>
<td>0.590</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>15500</td>
<td>1.050</td>
<td>0.670</td>
<td>0.630</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>16000</td>
<td>1.120</td>
<td>0.710</td>
<td>0.680</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>16500</td>
<td>1.200</td>
<td>0.760</td>
<td>0.730</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>17000</td>
<td>1.280</td>
<td>0.820</td>
<td>0.790</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>17500</td>
<td>1.340</td>
<td>0.880</td>
<td>0.860</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>18000</td>
<td>1.430</td>
<td>0.940</td>
<td>0.910</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>18500</td>
<td>1.510</td>
<td>0.990</td>
<td>0.980</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>19000</td>
<td>1.600</td>
<td>1.060</td>
<td>1.030</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>19500</td>
<td>1.680</td>
<td>1.110</td>
<td>1.110</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>20000</td>
<td>1.760</td>
<td>1.200</td>
<td>1.170</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>20500</td>
<td>1.860</td>
<td>1.250</td>
<td>1.240</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>21000</td>
<td>1.940</td>
<td>1.320</td>
<td>1.310</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>21500</td>
<td>2.030</td>
<td>1.410</td>
<td>1.380</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>22000</td>
<td>2.140</td>
<td>1.470</td>
<td>1.450</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>22500</td>
<td>2.270</td>
<td>2.170</td>
<td>2.380</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>
A.1.11 nppsAddC_32f_I test program

// Compile and Execute (Make sure in correct directory):
// nvcc -I/usr/local/cuda-5.5/include -L/usr/local/cuda-5.5/lib64
// NPPSAddC32fI_CPUCheck_Time.cu -lnppc -lnppi -lnpps -lm
// ./a.out

#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <csignal>
#include <ctime>

#define Constant 10

FILE *TimeFile = fopen("NPPSAddC32fI_Times.txt", "w");

void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.
", signum);
    fclose(TimeFile);
    exit(signum);
}

#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
    if (code != cudaSuccess)
    {
        fprintf(stderr,"GPUassert: %s %s %d
", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}

int main(int argc, char** argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    int PointBound = 100;
    signal(SIGINT, signalHandler);
    int i;
    clock_t Start, End;
    double Time;
    int Number_of_Errors;
    float Percent_Error;
    while(1)
    {
        srand(time(0));
        printf("Rows: %i
", rows);
        printf("Columns: %i
", cols);
        fprintf(TimeFile, "%i ", rows);
        float* h_A;
        float* h_A0;
        float* h_A1;
        //HOST ONLY VARIABLES.
        h_A = (float*)malloc(rows*cols*sizeof(float));
        if (h_A == 0)


```c
{
    fprintf (stderr, "!!!! host memory allocation error (h_A)\n");
    return EXIT_FAILURE;
}

//HOST VARIABLES USED FOR DEVICE 0.
h_A0 = (float*)malloc(rows*cols*sizeof(float));
if (h_A0 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_A0)\n");
    return EXIT_FAILURE;
}

//HOST VARIABLES USED FOR DEVICE 1.
h_A1 = (float*)malloc(rows*cols*sizeof(float));
if (h_A1 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_A1)\n");
    return EXIT_FAILURE;
}

//Creation of random values.
for (i=0; i < (rows*cols); i++)
{
    h_A[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
}
for (i=0; i < (rows*cols); i++)
{
    h_A0[i] = h_A[i];
    h_A1[i] = h_A[i];
}

//-----------------------------------------------------------------------------
//Start Time for CPU Calculations.
Start = clock();
//
//CPU Calculations
//
//h_A = h_A + Constant
//
for (i=0; i < (rows*cols); i++)
{
    h_A[i] = h_A[i] + Constant;
}

//End Time for CPU Calculations.
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f\n", fabs(Time));

//-----------------------------------------------------------------------------
//Start Time for GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A0;
//
Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void**)&d_A0, rows*cols*sizeof(float)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
//
//Add Constant Inplace (nppsAddC_32f_I)
//
//d_A = d_A + Constant
//
nppsAddC_32f_I(Constant, d_A0, rows*cols);  
//
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_A0, d_A0, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
```
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------//
//Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A1;
//
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A1, rows*cols*sizeof(float)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
//
//Add Constant Inplace (nppsAddC_32f_I)
//
//d_A = d_A + Constant
//
nppsAddC_32f_I(Constant, d_A1, rows*cols);
//
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_A1, d_A1, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------//
//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
    if ( ((h_A[i] - h_A0[i]) != 0) )
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
    if ( ((h_A[i] - h_A1[i]) != 0) )
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f\n", Percent_Error);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
Number_of_Errors = 0;
Percent_Error = 0;
//-----------------------------------------------------------------------------//
//HOST MEMORY CLEAN UP.
free(h_A);
free(h_A0);
free(h_A1);
//DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
//DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
A.1.12 nppsAddC_32f_I raw data

<table>
<thead>
<tr>
<th>Value of</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.070</td>
</tr>
<tr>
<td></td>
<td>0.070</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>0.010</td>
</tr>
<tr>
<td>1500</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.010</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>2000</td>
<td>0.020</td>
</tr>
<tr>
<td></td>
<td>0.030</td>
</tr>
<tr>
<td></td>
<td>0.010</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>2500</td>
<td>0.020</td>
</tr>
<tr>
<td>3000</td>
<td>0.040</td>
</tr>
<tr>
<td></td>
<td>0.010</td>
</tr>
<tr>
<td></td>
<td>0.020</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>3500</td>
<td>0.050</td>
</tr>
<tr>
<td>4000</td>
<td>0.060</td>
</tr>
<tr>
<td></td>
<td>0.030</td>
</tr>
<tr>
<td></td>
<td>0.030</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>4500</td>
<td>0.070</td>
</tr>
<tr>
<td>5000</td>
<td>0.090</td>
</tr>
<tr>
<td></td>
<td>0.050</td>
</tr>
<tr>
<td></td>
<td>0.040</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>5500</td>
<td>0.110</td>
</tr>
<tr>
<td>6000</td>
<td>0.130</td>
</tr>
<tr>
<td></td>
<td>0.060</td>
</tr>
<tr>
<td></td>
<td>0.080</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>6500</td>
<td>0.150</td>
</tr>
<tr>
<td>7000</td>
<td>0.180</td>
</tr>
<tr>
<td></td>
<td>0.110</td>
</tr>
<tr>
<td></td>
<td>0.130</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>7500</td>
<td>0.200</td>
</tr>
<tr>
<td>8000</td>
<td>0.230</td>
</tr>
<tr>
<td></td>
<td>0.110</td>
</tr>
<tr>
<td></td>
<td>0.120</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>8500</td>
<td>0.260</td>
</tr>
<tr>
<td>9000</td>
<td>0.290</td>
</tr>
<tr>
<td></td>
<td>0.160</td>
</tr>
<tr>
<td></td>
<td>0.150</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>9500</td>
<td>0.320</td>
</tr>
<tr>
<td>10000</td>
<td>0.360</td>
</tr>
<tr>
<td></td>
<td>0.190</td>
</tr>
<tr>
<td></td>
<td>0.170</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>10500</td>
<td>0.400</td>
</tr>
<tr>
<td>11000</td>
<td>0.440</td>
</tr>
<tr>
<td></td>
<td>0.230</td>
</tr>
<tr>
<td></td>
<td>0.220</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>11500</td>
<td>0.480</td>
</tr>
<tr>
<td>12000</td>
<td>0.520</td>
</tr>
<tr>
<td></td>
<td>0.270</td>
</tr>
<tr>
<td></td>
<td>0.260</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>12500</td>
<td>0.570</td>
</tr>
<tr>
<td>13000</td>
<td>0.610</td>
</tr>
<tr>
<td></td>
<td>0.320</td>
</tr>
<tr>
<td></td>
<td>0.300</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>13500</td>
<td>0.660</td>
</tr>
<tr>
<td>14000</td>
<td>0.710</td>
</tr>
<tr>
<td></td>
<td>0.370</td>
</tr>
<tr>
<td></td>
<td>0.340</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>14500</td>
<td>0.750</td>
</tr>
<tr>
<td>15000</td>
<td>0.810</td>
</tr>
<tr>
<td></td>
<td>0.420</td>
</tr>
<tr>
<td></td>
<td>0.400</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>15500</td>
<td>0.870</td>
</tr>
<tr>
<td>16000</td>
<td>0.920</td>
</tr>
<tr>
<td></td>
<td>0.490</td>
</tr>
<tr>
<td></td>
<td>0.450</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>16500</td>
<td>0.980</td>
</tr>
<tr>
<td>17000</td>
<td>1.040</td>
</tr>
<tr>
<td></td>
<td>0.540</td>
</tr>
<tr>
<td></td>
<td>0.510</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>17500</td>
<td>1.110</td>
</tr>
<tr>
<td>18000</td>
<td>1.160</td>
</tr>
<tr>
<td></td>
<td>0.620</td>
</tr>
<tr>
<td></td>
<td>0.570</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>18500</td>
<td>1.230</td>
</tr>
<tr>
<td>19000</td>
<td>1.300</td>
</tr>
<tr>
<td></td>
<td>0.690</td>
</tr>
<tr>
<td></td>
<td>0.620</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>19500</td>
<td>1.370</td>
</tr>
<tr>
<td>20000</td>
<td>1.440</td>
</tr>
<tr>
<td></td>
<td>0.770</td>
</tr>
<tr>
<td></td>
<td>0.700</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>20500</td>
<td>1.510</td>
</tr>
<tr>
<td>21000</td>
<td>1.590</td>
</tr>
<tr>
<td></td>
<td>0.830</td>
</tr>
<tr>
<td></td>
<td>0.780</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>21500</td>
<td>1.660</td>
</tr>
<tr>
<td>22000</td>
<td>1.750</td>
</tr>
<tr>
<td></td>
<td>0.930</td>
</tr>
<tr>
<td></td>
<td>0.860</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>22500</td>
<td>1.820</td>
</tr>
<tr>
<td>23000</td>
<td>1.910</td>
</tr>
<tr>
<td></td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>0.960</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>23500</td>
<td>1.980</td>
</tr>
<tr>
<td>24000</td>
<td>2.060</td>
</tr>
<tr>
<td></td>
<td>1.130</td>
</tr>
<tr>
<td></td>
<td>1.100</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>24500</td>
<td>2.160</td>
</tr>
<tr>
<td>25000</td>
<td>2.250</td>
</tr>
<tr>
<td></td>
<td>1.260</td>
</tr>
<tr>
<td></td>
<td>1.210</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>25500</td>
<td>2.340</td>
</tr>
<tr>
<td>26000</td>
<td>2.440</td>
</tr>
<tr>
<td></td>
<td>1.310</td>
</tr>
<tr>
<td></td>
<td>1.260</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>26500</td>
<td>2.530</td>
</tr>
<tr>
<td>27000</td>
<td>2.630</td>
</tr>
<tr>
<td></td>
<td>1.810</td>
</tr>
<tr>
<td></td>
<td>1.830</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>27500</td>
<td>2.730</td>
</tr>
<tr>
<td>28000</td>
<td>2.840</td>
</tr>
</tbody>
</table>
A.1.13 nppsCopy_32f test program

```c
#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

FILE *TimeFile = fopen("NPPSCopy32f_Times.txt", "w");

void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

int main(int argc, char** argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    int PointBound = 100;
    signal(SIGINT, signalHandler);
    int i;
    clock_t Start, End;
    double Time;
    int Number_of_Errors;
    float Percent_Error;
    while(1)
    {
        srand(time(0));
        printf("Rows: %i\n", rows);
        printf("Columns: %i\n", cols);
        fprintf(TimeFile, "%i ", rows);
        //---
    }
    return 0;
}
```
float* h_A;
float* h_B;
float* h_A0;
float* h_B0;
float* h_A1;
float* h_B1;
// HOST ONLY VARIABLES.
h_A = (float*)malloc(rows*cols*sizeof(float));
if (h_A == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_A)\n");
    return EXIT_FAILURE;
}
h_B = (float*)malloc(rows*cols*sizeof(float));
if (h_B == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_B)\n");
    return EXIT_FAILURE;
}
//HOST VARIABLES USED FOR DEVICE 0.
h_A0 = (float*)malloc(rows*cols*sizeof(float));
if (h_A0 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_A0)\n");
    return EXIT_FAILURE;
}
h_B0 = (float*)malloc(rows*cols*sizeof(float));
if (h_B0 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_B0)\n");
    return EXIT_FAILURE;
}
//HOST VARIABLES USED FOR DEVICE 1.
h_A1 = (float*)malloc(rows*cols*sizeof(float));
if (h_A1 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_A1)\n");
    return EXIT_FAILURE;
}
h_B1 = (float*)malloc(rows*cols*sizeof(float));
if (h_B1 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_B1)\n");
    return EXIT_FAILURE;
}
//Creation of random values.
for (i=0; i < (rows*cols); i++)
{
    h_A[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
}
for (i=0; i < (rows*cols); i++)
{
    h_A0[i] = h_A[i];
    h_A1[i] = h_A[i];
}
//-----------------------------------------------------------------------------
//Start Time for CPU Calculations.
Start = clock();
//CPU Calculations
//
//h_B = h_A
//for (i=0; i < (rows*cols); i++)
//{
//    h_B[i] = h_A[i];
//}
//End Time for CPU Calculations.
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%3f ", fabs(Time));
//------------------------------------------------------------------------------
//Start Time for GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A0;
float* d_B0;
//
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A0, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_B0, rows*cols*sizeof(float)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
//
//Copy (nppsCopy_32f)
//
//d_B = d_A
//
nppsCopy_32f(d_A0, d_B0, rows*cols);
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%3f ", fabs(Time));
//------------------------------------------------------------------------------
//Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A1;
float* d_B1;
//
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A1, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_B1, rows*cols*sizeof(float)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
//
//Copy (nppsCopy_32f)
//
//d_B = d_A
//
nppsCopy_32f(d_A1, d_B1, rows*cols);
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%3f ", fabs(Time));
//------------------------------------------------------------------------------
//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
  if ((h_B[i] - h_B0[i]) != 0)
A.1.14 nppsCopy_32f raw data

500 0.000 0.000 0.070 0.000 0.000 0.000
1000 0.010 0.000 0.000 0.000 0.000 0.000
1500 0.010 0.000 0.000 0.000 0.000 0.000
2000 0.020 0.000 0.010 0.000 0.000 0.000
2500 0.020 0.020 0.020 0.000 0.000 0.000
3000 0.030 0.030 0.020 0.000 0.000 0.000
3500 0.030 0.030 0.030 0.000 0.000 0.000
4000 0.040 0.030 0.030 0.000 0.000 0.000
4500 0.050 0.050 0.050 0.000 0.000 0.000
5000 0.070 0.070 0.070 0.000 0.000 0.000
5500 0.100 0.070 0.070 0.000 0.000 0.000
6000 0.140 0.090 0.080 0.000 0.000 0.000
6500 0.170 0.090 0.080 0.000 0.000 0.000
7000 0.200 0.110 0.110 0.000 0.000 0.000
7500 0.220 0.130 0.130 0.000 0.000 0.000
8000 0.240 0.150 0.140 0.000 0.000 0.000
8500 0.260 0.160 0.170 0.000 0.000 0.000
9000 0.320 0.200 0.170 0.000 0.000 0.000
9500 0.350 0.220 0.190 0.000 0.000 0.000
A.1.15 nppsDiv_32f test program

```c
#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

FILE *TimeFile = fopen("NPPSDiv32f_Times.txt", "w");

void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

Cuda Error Check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }

Cuda Error Check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }

int main(int argc, char** argv)
```
{ int rows = 500; 
int cols = 500; 
int step = 500; 
int PointBound = 100; 
signal(SIGINT, signalHandler); 
int i; 
clock_t Start, End; 
double Time; 
int Number_of_Errors; 
float Percent_Error; 
float Margin_of_Error_Ave; 
while(1) 
{
    srand(time(0)); 
    printf("Rows: %i\n", rows); 
    printf("Columns: %i\n", cols); 
    fprintf(TimeFile, "%i ", rows); 
    //------------------------------------------------------------------------------------
    float* h_A; 
    float* h_B; 
    float* h_C; 
    float* h_A0; 
    float* h_B0; 
    float* h_C0; 
    float* h_A1; 
    float* h_B1; 
    float* h_C1; 
    // //HOST ONLY VARIABLES. 
    h_A = (float*)malloc(rows*cols*sizeof(float)); 
    if (h_A == 0) 
    { 
        fprintf(stderr, "!!!! host memory allocation error (h_A)\n"); 
        return EXIT_FAILURE; 
    } 
    h_B = (float*)malloc(rows*cols*sizeof(float)); 
    if (h_B == 0) 
    { 
        fprintf(stderr, "!!!! host memory allocation error (h_B)\n"); 
        return EXIT_FAILURE; 
    } 
    h_C = (float*)malloc(rows*cols*sizeof(float)); 
    if (h_C == 0) 
    { 
        fprintf(stderr, "!!!! host memory allocation error (h_C)\n"); 
        return EXIT_FAILURE; 
    } 
    //HOST VARIABLES USED FOR DEVICE 0. 
    h_A0 = (float*)malloc(rows*cols*sizeof(float)); 
    if (h_A0 == 0) 
    { 
        fprintf(stderr, "!!!! host memory allocation error (h_A0)\n"); 
        return EXIT_FAILURE; 
    } 
    h_B0 = (float*)malloc(rows*cols*sizeof(float)); 
    if (h_B0 == 0) 
    { 
        fprintf(stderr, "!!!! host memory allocation error (h_B0)\n"); 
        return EXIT_FAILURE; 
    } 
    h_C0 = (float*)malloc(rows*cols*sizeof(float)); 
    if (h_C0 == 0) 
    { 
        fprintf(stderr, "!!!! host memory allocation error (h_C0)\n"); 
        return EXIT_FAILURE; 
    } 
    //HOST VARIABLES USED FOR DEVICE 1. 
    h_A1 = (float*)malloc(rows*cols*sizeof(float)); 
    if (h_A1 == 0) 
    { 
        fprintf(stderr, "!!!! host memory allocation error (h_A1)\n"); 
    }
return EXIT_FAILURE;
} h_B1 = (float*)malloc(rows*cols*sizeof(float));
if (h_B1 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_B1)\n");
    return EXIT_FAILURE;
} h_C1 = (float*)malloc(rows*cols*sizeof(float));
if (h_C1 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_C1)\n");
    return EXIT_FAILURE;
}
//Creation of random values.
for (i=0; i < (rows*cols); i++)
{
    h_A[i] = 2*PointBound*((float)rand()/RAND_MAX);
    h_B[i] = 2*PointBound*((float)rand()/RAND_MAX);
}
for (i=0; i < (rows*cols); i++)
{
    h_A0[i] = h_A[i];
    h_B0[i] = h_B[i];
    h_A1[i] = h_A[i];
    h_B1[i] = h_B[i];
}
//-----------------------------------------------------------------------------//
//Start Time for CPU Calculations.
Start = clock();
//
//CPU Calculations
//
//h_C = h_A / h_B
//
for (i=0; i < (rows*cols); i++)
{
    h_C[i] = h_A[i] / h_B[i];
}
//-----------------------------------------------------------------------------//
//End Time for CPU Calculations.
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------//
//Start Time for GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A0;
float* d_B0;
float* d_C0;
//
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void**)&d_A0, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void**)&d_B0, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void**)&d_C0, rows*cols*sizeof(float)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B0, h_B0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
//
//Divide (nppsDiv_32f)
//
//d_C = d_A / d_B
//
nppsDiv_32f(d_B0, d_A0, d_C0, rows*cols);
//
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_C0, d_C0, rows*cols*sizeof(float), cudaMemcpyDeviceToHost));

//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));

//Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();

float* d_A1;
float* d_B1;
float* d_C1;

//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A1, rows*cols*sizeof(float)));
CudaErrCheck( cudaMalloc((void **)&d_B1, rows*cols*sizeof(float)));
CudaErrCheck( cudaMalloc((void **)&d_C1, rows*cols*sizeof(float)));

//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(float), cudaMemcpyHostToDevice));
CudaErrCheck( cudaMemcpy(d_B1, h_B1, rows*cols*sizeof(float), cudaMemcpyHostToDevice));

//Divide (nppsDiv_32f)
//d_C = d_A / d_B

nppsDiv_32f(d_B1, d_A1, d_C1, rows*cols);

//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_C1, d_C1, rows*cols*sizeof(float), cudaMemcpyDeviceToHost));

//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));

//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
Margin_of_Error_Ave = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_C[i] - h_C0[i]) != 0)
    {
        Number_of_Errors++;
        Margin_of_Error_Ave = Margin_of_Error_Ave + abs(h_C[i] - h_C0[i]);
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
fprintf(TimeFile, "%f ", Margin_of_Error_Ave/Number_of_Errors);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\n", Percent_Error);
printf("Quadro FX 5800 Error Margin: %f\n", Margin_of_Error_Ave/Number_of_Errors);
Number_of_Errors = 0;
Percent_Error = 0;
Margin_of_Error_Ave = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_C[i] - h_C1[i]) != 0)
    {
        Number_of_Errors++;
        Margin_of_Error_Ave = Margin_of_Error_Ave + abs(h_C[i] - h_C1[i]);
    }
}
A.1.16 nppsDiv_32f raw data

<table>
<thead>
<tr>
<th>Value</th>
<th>nppsDiv_32f raw data</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.000 0.000 0.070 0.080 28.312 0.000000 28.312 0.000000</td>
</tr>
<tr>
<td>1000</td>
<td>0.010 0.000 0.010 0.010 28.343 0.000001 28.343 0.000001</td>
</tr>
<tr>
<td>1500</td>
<td>0.020 0.010 0.010 0.010 28.373 0.000001 28.373 0.000001</td>
</tr>
<tr>
<td>2000</td>
<td>0.030 0.010 0.010 0.010 28.352 0.000001 28.352 0.000001</td>
</tr>
<tr>
<td>2500</td>
<td>0.030 0.030 0.010 0.010 28.374 0.000001 28.374 0.000001</td>
</tr>
<tr>
<td>3000</td>
<td>0.050 0.030 0.020 0.020 28.355 0.000001 28.355 0.000001</td>
</tr>
<tr>
<td>3500</td>
<td>0.070 0.040 0.040 0.040 28.353 0.000001 28.353 0.000001</td>
</tr>
<tr>
<td>4000</td>
<td>0.080 0.050 0.060 0.060 28.353 0.000001 28.353 0.000001</td>
</tr>
<tr>
<td>4500</td>
<td>0.100 0.070 0.080 0.080 28.358 0.000001 28.358 0.000001</td>
</tr>
<tr>
<td>5000</td>
<td>0.130 0.080 0.090 0.090 28.357 0.000001 28.357 0.000001</td>
</tr>
<tr>
<td>5500</td>
<td>0.150 0.100 0.100 0.100 28.357 0.000001 28.357 0.000001</td>
</tr>
<tr>
<td>6000</td>
<td>0.190 0.120 0.120 0.120 28.355 0.000001 28.355 0.000001</td>
</tr>
<tr>
<td>6500</td>
<td>0.210 0.140 0.140 0.140 28.361 0.000001 28.361 0.000001</td>
</tr>
<tr>
<td>7000</td>
<td>0.250 0.160 0.160 0.160 28.351 0.000001 28.351 0.000001</td>
</tr>
<tr>
<td>7500</td>
<td>0.280 0.180 0.180 0.180 28.369 0.000001 28.369 0.000001</td>
</tr>
<tr>
<td>8000</td>
<td>0.330 0.200 0.200 0.200 28.360 0.000001 28.360 0.000001</td>
</tr>
<tr>
<td>8500</td>
<td>0.370 0.240 0.240 0.240 28.361 0.000001 28.361 0.000001</td>
</tr>
<tr>
<td>9000</td>
<td>0.420 0.260 0.260 0.260 28.354 0.000001 28.354 0.000001</td>
</tr>
<tr>
<td>9500</td>
<td>0.460 0.290 0.290 0.290 28.360 0.000001 28.360 0.000001</td>
</tr>
<tr>
<td>10000</td>
<td>0.510 0.320 0.320 0.320 28.371 0.000001 28.371 0.000001</td>
</tr>
<tr>
<td>10500</td>
<td>0.560 0.360 0.330 0.330 28.354 0.000001 28.354 0.000001</td>
</tr>
<tr>
<td>11000</td>
<td>0.620 0.390 0.370 0.370 28.359 nan 28.359 nan</td>
</tr>
<tr>
<td>11500</td>
<td>0.670 0.430 0.410 0.410 28.360 0.000001 28.360 0.000001</td>
</tr>
<tr>
<td>12000</td>
<td>0.730 0.470 0.430 0.430 28.362 0.000001 28.362 0.000001</td>
</tr>
<tr>
<td>12500</td>
<td>0.800 0.520 0.490 0.490 28.357 0.000001 28.357 0.000001</td>
</tr>
<tr>
<td>13000</td>
<td>0.870 0.550 0.620 0.620 28.360 0.000001 28.360 0.000001</td>
</tr>
<tr>
<td>13500</td>
<td>0.940 0.630 0.700 0.700 28.360 0.000001 28.360 0.000001</td>
</tr>
</tbody>
</table>
A.1.17 nppsDiv_32f_I test program

```c
#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

FILE *TimeFile = fopen("NPPSDiv32fI_Times.txt", "w");

void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

int main(int argc, char** argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    int PointBound = 100;
    srand(time(0));
    printf("Rows: %i\n", rows);
    int i;
    clock_t Start, End;
    double Time;
    int Number_of_Errors;
    float Percent_Error;
    float Margin_of_Error_Ave;
    while(1)
    {
        srand(time(0));
        printf("Rows: %i\n", rows);
```
printf("Columns: %i\n", cols);
printf(TimeFile, "%i ", rows);
//------------------------------------------------------------------------------
float* h_A;
float* h_B;
float* h_A0;
float* h_B0;
float* h_A1;
float* h_B1;
//
//HOST ONLY VARIABLES.
h_A = (float*)malloc(rows*cols*sizeof(float));
if (h_A == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_A)\n");
    return EXIT_FAILURE;
}
h_B = (float*)malloc(rows*cols*sizeof(float));
if (h_B == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_B)\n");
    return EXIT_FAILURE;
}
//HOST VARIABLES USED FOR DEVICE 0.
h_A0 = (float*)malloc(rows*cols*sizeof(float));
if (h_A0 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_A0)\n");
    return EXIT_FAILURE;
}
h_B0 = (float*)malloc(rows*cols*sizeof(float));
if (h_B0 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_B0)\n");
    return EXIT_FAILURE;
}
//HOST VARIABLES USED FOR DEVICE 1.
h_A1 = (float*)malloc(rows*cols*sizeof(float));
if (h_A1 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_A1)\n");
    return EXIT_FAILURE;
}
h_B1 = (float*)malloc(rows*cols*sizeof(float));
if (h_B1 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_B1)\n");
    return EXIT_FAILURE;
}
//Creation of random values.
for (i=0; i < (rows*cols); i++)
{
    h_A[i] = 2*PointBound*((float)rand()/RAND_MAX);
    h_B[i] = 2*PointBound*((float)rand()/RAND_MAX);
}
for (i=0; i < (rows*cols); i++)
{
    h_A0[i] = h_A[i];
    h_B0[i] = h_B[i];
    h_A1[i] = h_A[i];
    h_B1[i] = h_B[i];
}
//------------------------------------------------------------------------------
//Start Time for CPU Calculations.
Start = clock();
//
//CPU Calculations
//
//h_A = h_A / h_B
//
for (i=0; i < (rows*cols); i++)
{

h_A[i] = h_A[i] / h_B[i];
}
//End Time for CPU Calculations.
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%3f ", fabs(Time));
//-------------------------------------------------------------------------------//
//Start Time for GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A0;
float* d_B0;
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A0, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_B0, rows*cols*sizeof(float)) );
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B0, h_B0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
//Divide Inplace (nppsDiv_32f_I)
//
//d_A = d_A / d_B
//nppsDiv_32f_I(d_B0, d_A0, rows*cols);
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_A0, d_A0, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%3f ", fabs(Time));
//-------------------------------------------------------------------------------//
//Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A1;
float* d_B1;
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A1, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_B1, rows*cols*sizeof(float)) );
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B1, h_B1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
//Divide Inplace (nppsDiv_32f_I)
//
//d_A = d_A / d_B
//nppsDiv_32f_I(d_B1, d_A1, rows*cols);
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_A1, d_A1, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------//
//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
Margin_of_Error_Ave = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_A[i] - h_A0[i]) != 0)
    {
        Number_of_Errors++;
        Margin_of_Error_Ave = Margin_of_Error_Ave + abs(h_A[i] - h_A0[i]);
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
fprintf(TimeFile, "%f\n", Margin_of_Error_Ave/Number_of_Errors);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\%\n", Percent_Error);
printf("Quadro FX 5800 Error Margin: %f\n", Margin_of_Error_Ave/Number_of_Errors);
Number_of_Errors = 0;
Percent_Error = 0;
Margin_of_Error_Ave = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_A[i] - h_A1[i]) != 0)
    {
        Number_of_Errors++;
        Margin_of_Error_Ave = Margin_of_Error_Ave + abs(h_A[i] - h_A1[i]);
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
fprintf(TimeFile, "%f\n", Margin_of_Error_Ave/Number_of_Errors);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
printf("Tesla C1060 Error Percentage: %.3f\%\n", Percent_Error);
printf("Tesla C1060 Error Margin: %f\n\n", Margin_of_Error_Ave/Number_of_Errors);
//-----------------------------------------------------------------------------//
//HOST MEMORY CLEAN UP.
free(h_A);
free(h_B);
free(h_A0);
free(h_B0);
free(h_A1);
free(h_B1);
//DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
CudaErrCheck( cudaFree(d_B0) );
//DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
CudaErrCheck( cudaFree(d_B1) );
//-----------------------------------------------------------------------------//
//Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;
//--------------------------------------------------------------------------------------//
//END PROGRAM
return EXIT_SUCCESS;
//--------------------------------------------------------------------------------------//
### A.1.18 nppsDiv_32f_I raw data

<table>
<thead>
<tr>
<th>Value</th>
<th>0.000</th>
<th>0.000</th>
<th>0.000</th>
<th>0.060</th>
<th>0.000</th>
<th>0.000</th>
<th>0.000</th>
<th>28.265</th>
<th>0.000</th>
<th>0.000</th>
<th>28.265</th>
<th>0.000</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.000</td>
<td>0.000</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.341</td>
<td>0.000</td>
<td>0.000</td>
<td>28.341</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>0.010</td>
<td>0.010</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.381</td>
<td>0.000</td>
<td>0.000</td>
<td>28.381</td>
<td>0.000</td>
</tr>
<tr>
<td>1500</td>
<td>0.030</td>
<td>0.020</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.372</td>
<td>0.000</td>
<td>0.000</td>
<td>28.372</td>
<td>0.000</td>
</tr>
<tr>
<td>2000</td>
<td>0.040</td>
<td>0.020</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.365</td>
<td>0.000</td>
<td>0.000</td>
<td>28.365</td>
<td>0.000</td>
</tr>
<tr>
<td>2500</td>
<td>0.060</td>
<td>0.030</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.360</td>
<td>0.000</td>
<td>0.000</td>
<td>28.360</td>
<td>0.000</td>
</tr>
<tr>
<td>3000</td>
<td>0.070</td>
<td>0.040</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.356</td>
<td>0.000</td>
<td>0.000</td>
<td>28.356</td>
<td>0.000</td>
</tr>
<tr>
<td>3500</td>
<td>0.090</td>
<td>0.050</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.361</td>
<td>0.000</td>
<td>0.000</td>
<td>28.361</td>
<td>0.000</td>
</tr>
<tr>
<td>4000</td>
<td>0.110</td>
<td>0.060</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.356</td>
<td>0.000</td>
<td>0.000</td>
<td>28.356</td>
<td>0.000</td>
</tr>
<tr>
<td>4500</td>
<td>0.130</td>
<td>0.070</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.361</td>
<td>0.000</td>
<td>0.000</td>
<td>28.361</td>
<td>0.000</td>
</tr>
<tr>
<td>5000</td>
<td>0.150</td>
<td>0.080</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.357</td>
<td>0.000</td>
<td>0.000</td>
<td>28.357</td>
<td>0.000</td>
</tr>
<tr>
<td>5500</td>
<td>0.170</td>
<td>0.090</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.362</td>
<td>0.000</td>
<td>0.000</td>
<td>28.362</td>
<td>0.000</td>
</tr>
<tr>
<td>6000</td>
<td>0.190</td>
<td>0.100</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.357</td>
<td>0.000</td>
<td>0.000</td>
<td>28.357</td>
<td>0.000</td>
</tr>
<tr>
<td>6500</td>
<td>0.210</td>
<td>0.110</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.362</td>
<td>0.000</td>
<td>0.000</td>
<td>28.362</td>
<td>0.000</td>
</tr>
<tr>
<td>7000</td>
<td>0.230</td>
<td>0.120</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.357</td>
<td>0.000</td>
<td>0.000</td>
<td>28.357</td>
<td>0.000</td>
</tr>
<tr>
<td>7500</td>
<td>0.250</td>
<td>0.130</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.362</td>
<td>0.000</td>
<td>0.000</td>
<td>28.362</td>
<td>0.000</td>
</tr>
<tr>
<td>8000</td>
<td>0.270</td>
<td>0.140</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.357</td>
<td>0.000</td>
<td>0.000</td>
<td>28.357</td>
<td>0.000</td>
</tr>
<tr>
<td>8500</td>
<td>0.290</td>
<td>0.150</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.362</td>
<td>0.000</td>
<td>0.000</td>
<td>28.362</td>
<td>0.000</td>
</tr>
<tr>
<td>9000</td>
<td>0.310</td>
<td>0.160</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.357</td>
<td>0.000</td>
<td>0.000</td>
<td>28.357</td>
<td>0.000</td>
</tr>
<tr>
<td>9500</td>
<td>0.330</td>
<td>0.170</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.362</td>
<td>0.000</td>
<td>0.000</td>
<td>28.362</td>
<td>0.000</td>
</tr>
<tr>
<td>10000</td>
<td>0.350</td>
<td>0.180</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.357</td>
<td>0.000</td>
<td>0.000</td>
<td>28.357</td>
<td>0.000</td>
</tr>
<tr>
<td>10500</td>
<td>0.370</td>
<td>0.190</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.362</td>
<td>0.000</td>
<td>0.000</td>
<td>28.362</td>
<td>0.000</td>
</tr>
<tr>
<td>11000</td>
<td>0.390</td>
<td>0.200</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.357</td>
<td>0.000</td>
<td>0.000</td>
<td>28.357</td>
<td>0.000</td>
</tr>
<tr>
<td>11500</td>
<td>0.410</td>
<td>0.210</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>28.362</td>
<td>0.000</td>
<td>0.000</td>
<td>28.362</td>
<td>0.000</td>
</tr>
</tbody>
</table>

### A.1.19 nppsDivC_32f_I test program

```c
#include <npps.h>
#include <npi.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

//Defined Constant.
#define Constant 13.49
```
FILE *TimeFile = fopen("NPPSDivC32fI_Times.txt", "w");

void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

//-----------------------------------------------------------------------------------------//
//Cuda Error Check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
    if (code != cudaSuccess)
    {
        fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}

//-----------------------------------------------------------------------------------------//
//Main
int main(int argc, char** argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    int PointBound = 100;
    signal(SIGINT, signalHandler);
    int i;
    clock_t Start, End;
    double Time;
    int Number_of_Errors;
    float Percent_Error;
    float Margin_of_Error_Ave;
    while(1)
    {
        srand(time(0));
        printf("Rows: %i\n", rows);
        printf("Columns: %i\n", cols);
        printf(TimeFile, "%i \n", rows);
        float* h_A;
        float* h_A0;
        float* h_A1;
        //HOST ONLY VARIABLES.
        h_A = (float*)malloc(rows*cols*sizeof(float));
        if (h_A == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A)\n");
            return EXIT_FAILURE;
        }
        //HOST VARIABLES USED FOR DEVICE 0.
        h_A0 = (float*)malloc(rows*cols*sizeof(float));
        if (h_A0 == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A0)\n");
            return EXIT_FAILURE;
        }
        //HOST VARIABLES USED FOR DEVICE 1.
        h_A1 = (float*)malloc(rows*cols*sizeof(float));
        if (h_A1 == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A1)\n");
            return EXIT_FAILURE;
        }
        //Creation of random values.
        for (i=0; i < (rows*cols); i++)
h_A[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
}
for (i=0; i < (rows*cols); i++)
{
    h_A0[i] = h_A[i];
    h_A1[i] = h_A[i];
}

//Start Time for CPU Calculations.
Start = clock();

//CPU Calculations

//h_A = h_A / Constant
for (i=0; i < (rows*cols); i++)
{
    h_A[i] = h_A[i] / Constant;
}

//End Time for CPU Calculations.
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));

//Start Time for GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();

float* d_A0;

//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **) &d_A0, rows*cols*sizeof(float)) );

//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );

//Divide Constant Inplace (nppsDivC_32f_I)
//d_A = d_A / Constant
nppsDivC_32f_I(Constant, d_A0, rows*cols);

//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_A0, d_A0, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );

//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));

//Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();

float* d_A1;

//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **) &d_A1, rows*cols*sizeof(float)) );

//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );

//Divide Constant Inplace (nppsDivC_32f_I)
//d_A = d_A / Constant
// npssDivC_32f_I(Constant, d_A1, rows*cols);
//
// Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_A1, d_A1, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
//
// End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------//
// Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
Margin_of_Error_Ave = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_A[i] - h_A0[i]) != 0)
    {
        Number_of_Errors++;
        Margin_of_Error_Ave = Margin_of_Error_Ave + abs(h_A[i] - h_A0[i]);
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
fprintf(TimeFile, "%.3f\n", Margin_of_Error_Ave/Number_of_Errors);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
Number_of_Errors = 0;
Percent_Error = 0;
Margin_of_Error_Ave = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_A[i] - h_A1[i]) != 0)
    {
        Number_of_Errors++;
        Margin_of_Error_Ave = Margin_of_Error_Ave + abs(h_A[i] - h_A1[i]);
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
fprintf(TimeFile, "%.3f\n", Margin_of_Error_Ave/Number_of_Errors);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
//HOST MEMORY CLEAN UP.
free(h_A);
free(h_A0);
free(h_A1);
//DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
//DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
// Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;
//-----------------------------------------------------------------------------//
// END PROGRAM
return EXIT_SUCCESS;
//-----------------------------------------------------------------------------//
161
A.1.20 nppsDivC_32f_I raw data
500 0.000 0.070 0.070 16.018 0.000000 16.018 0.000000
1000 0.010 0.000 0.010 16.094 0.000000 16.094 0.000000
1500 0.010 0.010 0.000 16.103 0.000000 16.103 0.000000
2000 0.030 0.010 0.010 16.106 0.000000 16.106 0.000000
2500 0.040 0.020 0.010 16.117 0.000000 16.117 0.000000
3000 0.060 0.010 0.020 16.127 0.000000 16.127 0.000000
3500 0.090 0.030 0.020 16.131 0.000000 16.131 0.000000
4000 0.110 0.030 0.030 16.128 0.000000 16.128 0.000000
4500 0.140 0.050 0.030 16.130 0.000000 16.130 0.000000
5000 0.180 0.040 0.050 16.129 0.000000 16.129 0.000000
5500 0.210 0.060 0.050 16.132 0.000000 16.132 0.000000
6000 0.250 0.080 0.060 16.126 0.000000 16.126 0.000000
6500 0.300 0.080 0.080 16.124 0.000000 16.124 0.000000
7000 0.350 0.080 0.090 16.126 0.000000 16.126 0.000000
7500 0.400 0.100 0.090 16.136 0.000000 16.136 0.000000
8000 0.440 0.120 0.110 16.124 0.000000 16.124 0.000000
8500 0.500 0.140 0.130 16.133 0.000000 16.133 0.000000
9000 0.570 0.140 0.140 16.128 0.000000 16.128 0.000000
9500 0.630 0.170 0.150 16.134 0.000000 16.134 0.000000
10000 0.700 0.190 0.170 16.122 0.000000 16.122 0.000000
10500 0.770 0.210 0.190 16.132 0.000000 16.132 0.000000
11000 0.840 0.230 0.200 16.128 0.000000 16.128 0.000000
11500 0.920 0.240 0.230 16.131 0.000000 16.131 0.000000
12000 1.010 0.270 0.240 16.126 0.000000 16.126 0.000000
12500 1.090 0.290 0.270 16.129 0.000000 16.129 0.000000
13000 1.180 0.310 0.290 16.126 0.000000 16.126 0.000000
13500 1.270 0.330 0.320 16.129 0.000000 16.129 0.000000
14000 1.360 0.360 0.340 16.128 0.000000 16.128 0.000000
14500 1.460 0.390 0.360 16.131 0.000000 16.131 0.000000
15000 1.570 0.410 0.400 16.128 0.000000 16.128 0.000000
15500 1.670 0.440 0.420 16.130 0.000000 16.130 0.000000
16000 1.780 0.470 0.450 16.129 0.000000 16.129 0.000000
16500 1.900 0.500 0.470 16.126 0.000000 16.126 0.000000
17000 2.020 0.530 0.510 16.128 0.000000 16.128 0.000000
17500 2.140 0.560 0.530 16.129 0.000000 16.129 0.000000
18000 2.260 0.590 0.570 16.129 0.000000 16.129 0.000000
18500 2.380 0.630 0.600 16.126 0.000000 16.126 0.000000
19000 2.520 0.650 0.640 16.126 0.000000 16.126 0.000000
19500 2.650 0.700 0.660 16.128 0.000000 16.128 0.000000
20000 2.790 0.730 0.700 16.126 0.000000 16.126 0.000000
20500 2.930 0.770 0.740 16.127 0.000000 16.127 0.000000
21000 3.070 0.810 0.770 16.128 0.000000 16.128 0.000000
21500 3.220 0.850 0.810 16.129 0.000000 16.129 0.000000
22000 3.370 0.900 0.850 16.130 0.000000 16.130 0.000000
22500 3.520 0.950 0.900 16.130 0.000000 16.130 0.000000
23000 3.700 1.000 0.960 16.127 0.000000 16.127 0.000000
23500 3.850 1.050 1.020 16.127 0.000000 16.127 0.000000
24000 4.020 1.110 1.080 16.126 0.000000 16.126 0.000000
24500 4.180 1.170 1.130 16.129 0.000000 16.129 0.000000
25000 4.350 1.230 1.200 16.128 0.000000 16.128 0.000000
25500 4.530 1.290 1.260 16.129 0.000000 16.129 0.000000
26000 4.720 1.350 1.310 16.130 0.000000 16.130 0.000000
26500 4.890 1.410 1.390 16.127 0.000000 16.127 0.000000
27000 5.080 1.480 1.440 16.126 0.000000 16.126 0.000000
27500 5.270 1.550 1.510 16.127 0.000000 16.127 0.000000
28000 5.470 1.610 1.580 16.128 0.000000 16.128 0.000000
28500 5.670 1.680 1.650 16.128 0.000000 16.128 0.000000
29000 5.860 1.750 1.730 16.130 0.000000 16.130 0.000000
29500 6.070 1.820 1.800 16.127 0.000000 16.127 0.000000
30000 6.270 1.900 1.860 16.127 0.000000 16.127 0.000000
30500 6.480 1.960 1.940 16.129 0.000000 16.129 0.000000
31000 6.710 2.360 2.350 16.127 0.000000 16.127 0.000000
31500 6.980 2.480 2.460 16.128 0.000000 16.128 0.000000
32000 8.850 2.660 2.640 16.128 0.000000 16.128 0.000000


A.1.21 nppsMul_32f_I test program

#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <ctime>

FILE *TimeFile = fopen("NPPSMul32fI_Times.txt", "w");

int main(int argc, char** argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    int PointBound = 100;
    signal(SIGINT, signalHandler);
    int i;
    clock_t Start, End;
    double Time;
    int Number_of_Errors;
    float Percent_Error;
    while(1)
    {
        srand(time(0));
        printf("Rows: %i\n", rows);
        printf("Columns: %i\n", cols);
        printf(TimeFile, "%i\n", row);
        float* h_A;
        float* h_B;
        float* h_A0;
        float* h_B0;
        float* h_A1;
        float* h_B1;
        //HOST ONLY VARIABLES.
        h_A = (float*)malloc(rows*cols*sizeof(float));
        if (h_A == 0)
{  
    fprintf (stderr, "!!!! host memory allocation error (h_A)\n");  
    return EXIT_FAILURE;
}

h_B = (float*)malloc(rows*cols*sizeof(float));
if (h_B == 0)  
{  
    fprintf (stderr, "!!!! host memory allocation error (h_B)\n");  
    return EXIT_FAILURE;
}

//HOST VARIABLES USED FOR DEVICE 0.

h_A0 = (float*)malloc(rows*cols*sizeof(float));
if (h_A0 == 0)  
{  
    fprintf (stderr, "!!!! host memory allocation error (h_A0)\n");  
    return EXIT_FAILURE;
}

h_B0 = (float*)malloc(rows*cols*sizeof(float));
if (h_B0 == 0)  
{  
    fprintf (stderr, "!!!! host memory allocation error (h_B0)\n");  
    return EXIT_FAILURE;
}

//HOST VARIABLES USED FOR DEVICE 1.

h_A1 = (float*)malloc(rows*cols*sizeof(float));
if (h_A1 == 0)  
{  
    fprintf (stderr, "!!!! host memory allocation error (h_A1)\n");  
    return EXIT_FAILURE;
}

h_B1 = (float*)malloc(rows*cols*sizeof(float));
if (h_B1 == 0)  
{  
    fprintf (stderr, "!!!! host memory allocation error (h_B1)\n");  
    return EXIT_FAILURE;
}

//Creation of random values.

for (i=0; i < (rows*cols); i++)  
{  
    h_A[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_B[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
}

for (i=0; i < (rows*cols); i++)  
{  
    h_A0[i] = h_A[i];
    h_B0[i] = h_B[i];
    h_A1[i] = h_A[i];
    h_B1[i] = h_B[i];
}

Start = clock();  
//CPU Calculations

h_B = h_A * h_B

for (i=0; i < (rows*cols); i++)  
{  
    h_B[i] = h_A[i] * h_B[i];
}

End = clock();  
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f \n", fabs(Time));  
//GPU Calculations

cudaSetDevice(0);

Start = clock();  
//GPU Calculations

cudaDeviceSynchronize();
Start = clock();  
//
float* d_A0;
float* d_B0;
//
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)d_A0, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)d_B0, rows*cols*sizeof(float)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B0, h_B0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
//
//Multiply Inplace (nppsMul_32f_I)
//
//d_B = d_A * d_B
nppsMul_32f_I(d_A0, d_B0, rows*cols);
//
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_B0, d_B0, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-------------------------------------------------------------------------------
//Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A1;
float* d_B1;
//
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)d_A1, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)d_B1, rows*cols*sizeof(float)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B1, h_B1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
//
//Multiply Inplace (nppsMul_32f_I)
//
//d_B = d_A * d_B
nppsMul_32f_I(d_A1, d_B1, rows*cols);
//
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_B1, d_B1, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-------------------------------------------------------------------------------
//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_B[i] - h_B0[i]) != 0)
        { 
            Number_of_Errors++;
        }
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
  if ((h_B[i] - h_B1[i]) != 0)
  {
    Number_of_Errors++;
  }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f\n", Percent_Error);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
//HOST MEMORY CLEAN UP.
free(h_A);
free(h_B);
free(h_A0);
free(h_B0);
free(h_A1);
free(h_B1);
//DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
CudaErrCheck( cudaFree(d_B0) );
//DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
CudaErrCheck( cudaFree(d_B1) );
//Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;
//-----------------------------------------------------------------------------//
//END PROGRAM
return EXIT_SUCCESS;
//-----------------------------------------------------------------------------//

A.1.22 nppsMul_32f_I raw data

500 0.000 0.070 0.070 0.000 0.000
1000 0.010 0.010 0.010 0.000 0.000
1500 0.010 0.010 0.010 0.000 0.000
2000 0.010 0.020 0.010 0.000 0.000
2500 0.030 0.020 0.010 0.000 0.000
3000 0.030 0.040 0.020 0.000 0.000
3500 0.060 0.030 0.030 0.000 0.000
4000 0.070 0.040 0.030 0.000 0.000
4500 0.080 0.060 0.050 0.000 0.000
5000 0.080 0.070 0.070 0.000 0.000
5500 0.130 0.090 0.070 0.000 0.000
6000 0.160 0.090 0.100 0.000 0.000
6500 0.190 0.110 0.120 0.000 0.000
7000 0.210 0.140 0.150 0.000 0.000
7500 0.240 0.150 0.160 0.000 0.000
8000 0.280 0.180 0.170 0.000 0.000
8500 0.320 0.200 0.180 0.000 0.000
9000 0.350 0.230 0.210 0.000 0.000
9500 0.390 0.250 0.240 0.000 0.000
10000 0.440 0.270 0.270 0.000 0.000
10500 0.480 0.300 0.290 0.000 0.000
11000 0.530 0.330 0.310 0.000 0.000
A.1.23 nppsMulC_32f_I test program

```c
#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

//Defined Constant.
#define Constant -1

//File to store CPU and GPU run times.
FILE *TimeFile = fopen("NPPSMulC32fI_Times.txt", "w");

//Use to stop program in case of infinite loop
//Enter: "CTRL + C"
void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

//CUDA Error Check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
    if (code != cudaSuccess) {
        fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}

int main(int argc, char** argv)
```
int rows = 500;
int cols = 500;
int step = 500;
int PointBound = 100;
signal(SIGINT, signalHandler);
int i;
clock_t Start, End;
double Time;
int Number_of_Errors;
float Percent_Error;
while(1)
{
    srand(time(0));
    printf("Rows: %i\n", rows);
    printf("Columns: %i\n", cols);
    fprintf(TimeFile, "%i ", rows);
    //------------------------------------------------------------------
    float* h_A;
    float* h_A0;
    float* h_A1;
    //------------------------------------------------------------------
    //HOST ONLY VARIABLES.
    h_A = (float*)malloc(rows*cols*sizeof(float));
    if (h_A == 0)
    {
        fprintf(stderr, "!!!! host memory allocation error (h_A)\n");
        return EXIT_FAILURE;
    }
    //HOST VARIABLES USED FOR DEVICE 0.
    h_A0 = (float*)malloc(rows*cols*sizeof(float));
    if (h_A0 == 0)
    {
        fprintf(stderr, "!!!! host memory allocation error (h_A0)\n");
        return EXIT_FAILURE;
    }
    //HOST VARIABLES USED FOR DEVICE 1.
    h_A1 = (float*)malloc(rows*cols*sizeof(float));
    if (h_A1 == 0)
    {
        fprintf(stderr, "!!!! host memory allocation error (h_A1)\n");
        return EXIT_FAILURE;
    }
    //------------------------------------------------------------------
    //Creation of random values.
    for (i=0; i < (rows*cols); i++)
    {
        h_A[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    }
    for (i=0; i < (rows*cols); i++)
    {
        h_A0[i] = h_A[i];
        h_A1[i] = h_A[i];
    }
    //------------------------------------------------------------------
    //Start Time for CPU Calculations.
    Start = clock();
    //------------------------------------------------------------------
    //CPU Calculations
    //------------------------------------------------------------------
    //h_A = Constant * h_A
    for (i=0; i < (rows*cols); i++)
    {
        h_A[i] = Constant * h_A[i];
    }
    //------------------------------------------------------------------
    //End Time for CPU Calculations.
    End = clock();
    Time = (End-Start)*1.0/CLOCKS_PER_SEC;
    fprintf(TimeFile, "%.3f ", fabs(Time));
    //------------------------------------------------------------------
    //Start Time for GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();  
//  
float* d_A0;  
//  
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A0, rows*cols*sizeof(float)) );  
//  
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );  
//  
//Multiply Constant Inplace (nppsMulC_32f_I)  
//  
//d_A = Constant * d_A  
//  
nppsMulC_32f_I(Constant, d_A0, rows*cols);  
//  
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_A0, d_A0, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );  
//  
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));  
//  
//Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();  
//  
float* d_A1;  
//  
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A1, rows*cols*sizeof(float)) );  
//  
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );  
//  
//Multiply Constant Inplace (nppsMulC_32f_I)  
//  
//d_A = Constant * d_A  
//  
nppsMulC_32f_I(Constant, d_A1, rows*cols);  
//  
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_A1, d_A1, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );  
//  
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));  
//  
//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
  if ((h_A1[i] - h_A0[i]) != 0)
  {
    Number_of_Errors++;
  }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\n", Percent_Error);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_A[i] - h_A1[i]) != 0)
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%f\n", Percent_Error);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
printf("Tesla C1060 Error Percentage: %.3f\n\n", Percent_Error);
//-----------------------------------------------------------------------------//
//HOST MEMORY CLEAN UP.
free(h_A);
free(h_A0);
free(h_A1);
//DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
//DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
//-----------------------------------------------------------------------------//
//Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;
//-----------------------------------------------------------------------------//
//END PROGRAM
return EXIT_SUCCESS;
//-----------------------------------------------------------------------------//

A.1.24 nppsMulC_32f_I raw data

500 0.000 0.070 0.060 0.000 0.000
1000 0.000 0.000 0.000 0.000 0.000
1500 0.010 0.010 0.010 0.000 0.000
2000 0.020 0.020 0.020 0.000 0.000
2500 0.040 0.040 0.040 0.000 0.000
3000 0.060 0.060 0.060 0.000 0.000
3500 0.080 0.080 0.080 0.000 0.000
4000 0.100 0.100 0.100 0.000 0.000
4500 0.120 0.120 0.120 0.000 0.000
5000 0.140 0.140 0.140 0.000 0.000
5500 0.160 0.160 0.160 0.000 0.000
6000 0.180 0.180 0.180 0.000 0.000
6500 0.200 0.200 0.200 0.000 0.000
7000 0.220 0.220 0.220 0.000 0.000
7500 0.240 0.240 0.240 0.000 0.000
8000 0.260 0.260 0.260 0.000 0.000
8500 0.280 0.280 0.280 0.000 0.000
9000 0.300 0.300 0.300 0.000 0.000
9500 0.320 0.320 0.320 0.000 0.000
10000 0.340 0.340 0.340 0.000 0.000
10500 0.360 0.360 0.360 0.000 0.000
11000 0.380 0.380 0.380 0.000 0.000
11500 0.400 0.400 0.400 0.000 0.000
12000 0.420 0.420 0.420 0.000 0.000
12500 0.440 0.440 0.440 0.000 0.000
13000 0.460 0.460 0.460 0.000 0.000
13500 0.480 0.480 0.480 0.000 0.000
14000 0.500 0.500 0.500 0.000 0.000
14500 0.520 0.520 0.520 0.000 0.000
15000 0.540 0.540 0.540 0.000 0.000
15500 0.560 0.560 0.560 0.000 0.000
A.1.25 nppsSqr_32f_I test program

#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

FILE *TimeFile = fopen("NPPSSqr32fI_Times.txt", "w");
fprintf(stderr,"GPUassert: %s %s \%d\n", cudaGetErrorString(code), file, line);
if (abort) exit(code);
}

int main(int argc, char** argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    int PointBound = 100;
    signal(SIGINT, signalHandler);
    int i;
    clock_t Start, End;
    double Time;
    int Number_of_Errors;
    float Percent_Error;
    while(1)
    {  
        srand(time(0));
        printf("Rows: %i\n", rows);
        printf("Columns: %i\n", cols);
        fprintf(TimeFile, "%i ", rows);
        //-------------------------------
        float* h_A;
        float* h_A0;
        float* h_A1;
        //HOST ONLY VARIABLES.
        h_A = (float*)malloc(rows*cols*sizeof(float));
        if (h_A == 0)
        {  
            fprintf (stderr, "!!!! host memory allocation error (h_A)\n");
            return EXIT_FAILURE;
        }
        //HOST VARIABLES USED FOR DEVICE 0.
        h_A0 = (float*)malloc(rows*cols*sizeof(float));
        if (h_A0 == 0)
        {  
            fprintf (stderr, "!!!! host memory allocation error (h_A0)\n");
            return EXIT_FAILURE;
        }
        //HOST VARIABLES USED FOR DEVICE 1.
        h_A1 = (float*)malloc(rows*cols*sizeof(float));
        if (h_A1 == 0)
        {  
            fprintf (stderr, "!!!! host memory allocation error (h_A1)\n");
            return EXIT_FAILURE;
        }
        //Creation of random values.
        for (i=0; i < (rows*cols); i++)
        {  
            h_A[i] = 2*PointBound*(((float)rand())/(RAND_MAX)-PointBound);
        }
        for (i=0; i < (rows*cols); i++)
        {  
            h_A0[i] = h_A[i];
            h_A1[i] = h_A[i];
        }
        //-------------------------------
        //Start Time for CPU Calculations.
        Start = clock();
        //CPU Calculations
        //h_A = (d_A)^2
        for (i=0; i < (rows*cols); i++)
        {  
            h_A[i] = h_A[i] * h_A[i];
        }
// End Time for CPU Calculations.
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------
// Start Time for GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A0;
// Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A0, rows*cols*sizeof(float)) );
// Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
// Squared Inplace (nppsSqr_32f_I)
//
// d_A = (d_A)^2
// nppsSqr_32f_I(d_A0, rows*cols);
// Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_A0, d_A0, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
// End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------
// Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A1;
// Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A1, rows*cols*sizeof(float)) );
// Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
// Squared Inplace (nppsSqr_32f_I)
//
// d_A = (d_A)^2
// nppsSqr_32f_I(d_A1, rows*cols);
// Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_A1, d_A1, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
// End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------
// Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{ if ((h_A[i] - h_A0[i]) != 0)
Number_of_Errors++;
}

Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\%
", Percent_Error);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_A[i] - h_A1[i]) != 0)
    {
        Number_of_Errors++;
    }
}

Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
printf("Tesla C1060 Error Percentage: %.3f\%

", Percent_Error);
//-----------------------------------------------------------------------------//
//HOST MEMORY CLEAN UP.
free(h_A);
free(h_A0);
free(h_A1);
//DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
//DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
//-----------------------------------------------------------------------------//
//Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;
//-----------------------------------------------------------------------------//

A.1.26 nppsSqr_32f_I raw data

500 0.010 0.070 0.070 0.000 0.000
1000 0.010 0.000 0.000 0.000 0.000
1500 0.010 0.010 0.010 0.000 0.000
2000 0.020 0.010 0.010 0.000 0.000
2500 0.020 0.020 0.020 0.000 0.000
3000 0.040 0.020 0.010 0.000 0.000
3500 0.060 0.020 0.020 0.000 0.000
4000 0.070 0.040 0.030 0.000 0.000
4500 0.090 0.040 0.040 0.000 0.000
5000 0.110 0.040 0.050 0.000 0.000
5500 0.120 0.060 0.050 0.000 0.000
6000 0.150 0.070 0.070 0.000 0.000
6500 0.180 0.080 0.070 0.000 0.000
7000 0.210 0.090 0.090 0.000 0.000
7500 0.240 0.110 0.100 0.000 0.000
8000 0.270 0.110 0.120 0.000 0.000
8500 0.320 0.130 0.120 0.000 0.000
9000 0.350 0.150 0.150 0.000 0.000
9500 0.390 0.160 0.170 0.000 0.000
10000 0.430 0.190 0.180 0.000 0.000
10500 0.470 0.210 0.200 0.000 0.000
11000 0.510 0.230 0.210 0.000 0.000
11500 0.560 0.240 0.240 0.000 0.000
12000 0.620 0.260 0.260 0.000 0.000
A.1.27 nppsSqrt_32f_I test program

//Compile and Execute (Make sure in correct directory):
// nvcc -I/usr/local/cuda-5.5/include -L/usr/local/cuda-5.5/lib64
// NPPSSqrt32fI_CPUCheck_Time.cu -lnppc -lnppi -lnpps -lm
// ./a.out

#include <npps.h>
#include <nppi.h>
#include <cstio.h>
#include <ctime>

FILE *TimeFile = fopen("NPPSSqrt32fI_Times.txt", "w");

void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}
//Cuda Error Check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true) 
{
    if (code != cudaSuccess)
    {
        fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}

//-----------------------------------------------------------------------------------------//
//Main
int main(int argc, char** argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    int PointBound = 100;
signal(SIGINT, signalHandler);
    int i;
    clock_t Start, End;
    double Time;
    int Number_of_Errors;
    float Percent_Error;
    float Margin_of_Error_Ave;
    while(1)
    {
        srand(time(0));
        printf("Rows: %i\n", rows);
        printf("Columns: %i\n", cols);
        fprintf(TimeFile, "%i ", rows);
        //HOST VARIABLES.
        float* h_A;
        float* h_A0;
        float* h_A1;
        h_A = (float*)malloc(rows*cols*sizeof(float));
        if (h_A == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A)\n");
            return EXIT_FAILURE;
        }
        h_A0 = (float*)malloc(rows*cols*sizeof(float));
        if (h_A0 == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A0)\n");
            return EXIT_FAILURE;
        }
        h_A1 = (float*)malloc(rows*cols*sizeof(float));
        if (h_A1 == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A1)\n");
            return EXIT_FAILURE;
        }
        //Creation of random values.
        for (i=0; i < (rows*cols); i++)
        {
            h_A[i] = 2*PointBound*((float)rand()/RAND_MAX);
        }
        for (i=0; i < (rows*cols); i++)
        {
            h_A0[i] = h_A[i];
            h_A1[i] = h_A[i];
        }
        //Start Time for CPU Calculations.
        Start = clock();

        //GPU Calculations...
        //End Time for CPU Calculations.
        Time = (double) (clock() - Start) / CLOCKS_PER_SEC;
        Number_of_Errors = 0;
        Percent_Error = 0.0;
        Margin_of_Error_Ave = 0.0;
        for (i=0; i < (rows*cols); i++)
        {
            if (h_A[i] != h_A0[i] || h_A[i] != h_A1[i])
                Number_of_Errors++;
            Percent_Error += ((h_A[i] - h_A0[i])/PointBound);
            Margin_of_Error_Ave += ((h_A[i] - h_A0[i])/PointBound);
        }
        Percent_Error /= (float)Number_of_Errors;
        Margin_of_Error_Ave /= (float)Number_of_Errors;
        fprintf(TimeFile, "%f %f %f\n", Time, Percent_Error, Margin_of_Error_Ave);
    }
}

//CUDA Error Check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
//CPU Calculations
//
//h_A = sqrt(d_A)
//
//for (i=0; i < (rows*cols); i++)
//{
//    h_A[i] = sqrt(h_A[i]);
//}
//
//End Time for CPU Calculations.
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//------------------------------------------------------------------------------
//Start Time for GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A0;
//
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A0, rows*cols*sizeof(float)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
//
//Square Root Inplace (nppsSqrt_32f_I)
//
//d_A = sqrt(d_A)
//
nppsSqrt_32f_I(d_A0, rows*cols);
//
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_A0, d_A0, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//------------------------------------------------------------------------------
//Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A1;
//
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A1, rows*cols*sizeof(float)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
//
//Square Root Inplace (nppsSqrt_32f_I)
//
//d_A = sqrt(d_A)
//
nppsSqrt_32f_I(d_A1, rows*cols);
//
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_A1, d_A1, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
Margin_of_Error_Ave = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_A[i] - h_A0[i]) != 0)
    {
        Number_of_Errors++;
        Margin_of_Error_Ave = Margin_of_Error_Ave + abs(h_A[i] - h_A0[i]);
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
fprintf(TimeFile, "%f ", Margin_of_Error_Ave/Number_of_Errors);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\n", Percent_Error);
printf("Quadro FX 5800 Error Margin: %f\n", Margin_of_Error_Ave/Number_of_Errors);
Number_of_Errors = 0;
Percent_Error = 0;
Margin_of_Error_Ave = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_A[i] - h_A1[i]) != 0)
    {
        Number_of_Errors++;
        Margin_of_Error_Ave = Margin_of_Error_Ave + abs(h_A[i] - h_A1[i]);
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
fprintf(TimeFile, "%f ", Margin_of_Error_Ave/Number_of_Errors);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
printf("Tesla C1060 Error Percentage: %.3f\n", Percent_Error);
printf("Tesla C1060 Error Margin: %f\n", Margin_of_Error_Ave/Number_of_Errors);
//HOST MEMORY CLEAN UP.
free(h_A);
free(h_A0);
free(h_A1);
//DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
//DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );

//Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;

A.1.28 nppsSqrt_32f_I raw data
A.1.29 nppsSub_32f test program

#include <npps.h>
#include <nppi.h>

// Compile and Execute (Make sure in correct directory):
// nvcc -I /usr/local/cuda-5.5/include -L /usr/local/cuda-5.5/lib64 NPPSSub32f_CPUCheck_Time.cu
// -lnppc -lnppi -lnpps -lm
// ./a.out
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

//File to store CPU and GPU run times.
FILE *TimeFile = fopen("NPPSSub32f_Times.txt", "w");

//Use to stop program in case of infinite loop
//Enter: "CTRL + C"
void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

//Cuda Error Check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
    if (code != cudaSuccess)
    {
        fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}

//Main
int main(int argc, char** argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    int PointBound = 100;
    signal(SIGINT, signalHandler);
    int i;
    clock_t Start, End;
    double Time;
    int Number_of_Errors;
    float Percent_Error;
    while(1)
    {
        srand(time(0));
        printf("Rows: %i\n", rows);
        printf("Columns: %i\n", cols);
        fprintf(TimeFile, "%i ", rows);
        //HOST ONLY VARIABLES.
        float* h_A;
        float* h_B;
        float* h_C;
        float* h_A0;
        float* h_B0;
        float* h_C0;
        float* h_A1;
        float* h_B1;
        float* h_C1;
        //HOST ONLY VARIABLES.
        h_A = (float*)malloc(rows*cols*sizeof(float));
        if (h_A == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A)\\n");
            return EXIT_FAILURE;
        }
        h_B = (float*)malloc(rows*cols*sizeof(float));
        if (h_B == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_B)\\n");
            return EXIT_FAILURE;
        }
        h_C = (float*)malloc(rows*cols*sizeof(float));
if (h_C == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_C)\n");
    return EXIT_FAILURE;
}

//HOST VARIABLES USED FOR DEVICE 0.

h_A0 = (float*)malloc(rows*cols*sizeof(float));
if (h_A0 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_A0)\n");
    return EXIT_FAILURE;
}

h_B0 = (float*)malloc(rows*cols*sizeof(float));
if (h_B0 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_B0)\n");
    return EXIT_FAILURE;
}

h_C0 = (float*)malloc(rows*cols*sizeof(float));
if (h_C0 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_C0)\n");
    return EXIT_FAILURE;
}

//HOST VARIABLES USED FOR DEVICE 1.

h_A1 = (float*)malloc(rows*cols*sizeof(float));
if (h_A1 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_A1)\n");
    return EXIT_FAILURE;
}

h_B1 = (float*)malloc(rows*cols*sizeof(float));
if (h_B1 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_B1)\n");
    return EXIT_FAILURE;
}

h_C1 = (float*)malloc(rows*cols*sizeof(float));
if (h_C1 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_C1)\n");
    return EXIT_FAILURE;
}

//Creation of random values.

for (i=0; i < (rows*cols); i++)
{
    h_A[i] = 2*PointBound *((float)rand()/RAND_MAX)-PointBound;
    h_B[i] = 2*PointBound *((float)rand()/RAND_MAX)-PointBound;
}

for (i=0; i < (rows*cols); i++)
{
    h_A0[i] = h_A[i];
    h_B0[i] = h_B[i];
    h_A1[i] = h_A[i];
    h_B1[i] = h_B[i];
}

//------------------------------\
//Start Time for CPU Calculations.

Start = clock();
#endif

//CPU Calculations

//h_C = h_A - h_B

for (i=0; i < (rows*cols); i++)
{
    h_C[i] = h_A[i] - h_B[i];
}

//End Time for CPU Calculations.

End = clock();

Time = (End-Start)*1.8/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------//
// Start Time for GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A0;
float* d_B0;
float* d_C0;
//
_ALLOCATE Device Memory (GPU).
CudaErrCheck( cudaMalloc((void**)&d_A0, rows*cols*sizeof(float)));
CudaErrCheck( cudaMalloc((void**)&d_B0, rows*cols*sizeof(float)));
CudaErrCheck( cudaMalloc((void**)&d_C0, rows*cols*sizeof(float)));
//
// Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float),
cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B0, h_B0, rows*cols*sizeof(float),
cudaMemcpyHostToDevice) );
//
// Subtract (nppsSub_32f)
//
// d_C = d_A - d_B
// nppsSub_32f(d_B0, d_A0, d_C0, rows*cols);
//
// Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_C0, d_C0, rows*cols*sizeof(float),
cudaMemcpyDeviceToHost ) );
//
// End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------//
// Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A1;
float* d_B1;
float* d_C1;
//
_ALLOCATE Device Memory (GPU).
CudaErrCheck( cudaMalloc((void**)&d_A1, rows*cols*sizeof(float)));
CudaErrCheck( cudaMalloc((void**)&d_B1, rows*cols*sizeof(float)));
CudaErrCheck( cudaMalloc((void**)&d_C1, rows*cols*sizeof(float)));
//
// Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(float),
cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B1, h_B1, rows*cols*sizeof(float),
cudaMemcpyHostToDevice) );
//
// Subtract (nppsSub_32f)
//
// d_C = d_A - d_B
// nppsSub_32f(d_B1, d_A1, d_C1, rows*cols);
//
// Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_C1, d_C1, rows*cols*sizeof(float),
cudaMemcpyDeviceToHost ) );
//
// End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
if ((h_C[i] - h_C0[i]) != 0)
{
    Number_of_Errors++;
}
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\%
", Percent_Error);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_C[i] - h_C1[i]) != 0)
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f\n", Percent_Error);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
printf("Tesla C1060 Error Percentage: %.3f\%

", Percent_Error);
//HOST MEMORY CLEAN UP.
free(h_A);
free(h_B);
free(h_C);
free(h_A0);
free(h_B0);
free(h_C0);
free(h_A1);
free(h_B1);
free(h_C1);
//DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
CudaErrCheck( cudaFree(d_B0) );
CudaErrCheck( cudaFree(d_C0) );
//DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
CudaErrCheck( cudaFree(d_B1) );
CudaErrCheck( cudaFree(d_C1) );
//Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;
//--------------------------------------------------------------------------------------
//END PROGRAM
return EXIT_SUCCESS;
//--------------------------------------------------------------------------------------

A.1.30 nppsSub_32f raw data

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.000</td>
<td>0.070</td>
<td>0.070</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>0.010</td>
<td>0.000</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1500</td>
<td>0.010</td>
<td>0.000</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2000</td>
<td>0.020</td>
<td>0.020</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2500</td>
<td>0.030</td>
<td>0.030</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3000</td>
<td>0.050</td>
<td>0.030</td>
<td>0.020</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>
A.1.31 nppsSub_32f_I test program

#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

FILE *TimeFile = fopen("NPPSSub32fI_Times.txt", "w");

void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
    if (code != cudaSuccess)
    {
        fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}

//CUDA Error Check
#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <cstddef>
#include <ctime>

FILE *TimeFile;
```c
int main(int argc, char** argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    int PointBound = 100;
    signal(SIGINT, signalHandler);
    int i;
    clock_t Start, End;
    double Time;
    int Number_of_Errors;
    float Percent_Error;
    while(1)
    {
        srand(time(0));
        printf("Rows: %i\n", rows);
        printf("Columns: %i\n", cols);
        fprintf(TimeFile, "%i ", rows);
        
        float* h_A;
        float* h_B;
        float* h_A0;
        float* h_B0;
        float* h_A1;
        float* h_B1;
        
        //HOST ONLY VARIABLES.
        h_A = (float*)malloc(rows*cols*sizeof(float));
        if (h_A == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A)\n");
            return EXIT_FAILURE;
        }
        h_B = (float*)malloc(rows*cols*sizeof(float));
        if (h_B == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_B)\n");
            return EXIT_FAILURE;
        }
        //HOST VARIABLES USED FOR DEVICE 0.
        h_A0 = (float*)malloc(rows*cols*sizeof(float));
        if (h_A0 == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A0)\n");
            return EXIT_FAILURE;
        }
        h_B0 = (float*)malloc(rows*cols*sizeof(float));
        if (h_B0 == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_B0)\n");
            return EXIT_FAILURE;
        }
        //HOST VARIABLES USED FOR DEVICE 1.
        h_A1 = (float*)malloc(rows*cols*sizeof(float));
        if (h_A1 == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A1)\n");
            return EXIT_FAILURE;
        }
        h_B1 = (float*)malloc(rows*cols*sizeof(float));
        if (h_B1 == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_B1)\n");
            return EXIT_FAILURE;
        }
        //Creation of random values.
        for (i=0; i < (rows*cols); i++)
        {
```
h_A[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
h_B[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
}
for (i=0; i < (rows*cols); i++)
{
    h_A0[i] = h_A[i];
    h_B0[i] = h_B[i];
    h_A1[i] = h_A[i];
    h_B1[i] = h_B[i];
}

//*************************************************************************/

//Start Time for CPU Calculations.
Start = clock();
//
//CPU Calculations
//
//h_A = h_A - h_B
//
for (i=0; i < (rows*cols); i++)
{
    h_A[i] = h_A[i] - h_B[i];
}
//
//End Time for CPU Calulations.
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//*************************************************************************/

//Start Time for GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A0;
float* d_B0;
//
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A0, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_B0, rows*cols*sizeof(float)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B0, h_B0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
//
//Subtract Inplace (nppsSub_32f_I)
//
//d_A = d_A - d_B
//
nppsSub_32f_I(d_B0, d_A0, rows*cols);
//
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_B0, d_B0, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//*************************************************************************/

//Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();
//
float* d_A1;
float* d_B1;
//
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A1, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_b1, rows*cols*sizeof(float)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_a1, h_a1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_b1, h_b1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
//
//Subtract Inplace (nppsSub_32f_I)
//
//d_a = d_a - d_b
// nppsSub_32f_I(d_b1, d_a1, rows*cols);
//
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_b1, d_b1, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%f\n", fabs(Time));
//------------------------------------------------------------------------------------
//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
   if ((h_b[i] - h_b0[i]) != 0)
   {  
      Number_of_Errors++;
      }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%3.3f\n", Percent_Error);
printf("Quadro FX 5800 Error Percentage: %.3f\n", Percent_Error);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
   if ((h_b[i] - h_b1[i]) != 0)
   {  
      Number_of_Errors++;
      }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%3.3f\n", Percent_Error);
printf("Tesla C1060 Error Percentage: %.3f\n\n", Percent_Error);
//------------------------------------------------------------------------------------
//HOST MEMORY CLEAN UP.
free(h_a);
free(h_b);
free(h_b0);
free(h_a0);
free(h_a1);
free(h_b1);
//DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_a0) );
CudaErrCheck( cudaFree(d_b0) );
//DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_a1) );
CudaErrCheck( cudaFree(d_b1) );
//------------------------------------------------------------------------------------
//Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;
//------------------------------------------------------------------------------------
A.1.32 nppsSub_32f_I raw data

500 0.000 0.000 0.000 0.000 0.000
1000 0.010 0.000 0.000 0.000 0.000
1500 0.010 0.010 0.010 0.000 0.000
2000 0.010 0.010 0.010 0.000 0.000
2500 0.020 0.010 0.010 0.000 0.000
3000 0.040 0.020 0.020 0.000 0.000
3500 0.050 0.040 0.030 0.000 0.000
4000 0.070 0.040 0.040 0.000 0.000
4500 0.090 0.070 0.050 0.000 0.000
5000 0.110 0.070 0.060 0.000 0.000
5500 0.130 0.080 0.080 0.000 0.000
6000 0.150 0.110 0.100 0.000 0.000
6500 0.190 0.120 0.110 0.000 0.000
7000 0.210 0.140 0.130 0.000 0.000
7500 0.240 0.150 0.160 0.000 0.000
8000 0.280 0.170 0.180 0.000 0.000
8500 0.320 0.200 0.190 0.000 0.000
9000 0.360 0.220 0.220 0.000 0.000
9500 0.390 0.250 0.240 0.000 0.000
10000 0.440 0.270 0.270 0.000 0.000
10500 0.480 0.310 0.280 0.000 0.000
11000 0.530 0.330 0.310 0.000 0.000
11500 0.570 0.370 0.340 0.000 0.000
12000 0.630 0.400 0.380 0.000 0.000
12500 0.680 0.440 0.400 0.000 0.000
13000 0.750 0.460 0.450 0.000 0.000
13500 0.790 0.500 0.480 0.000 0.000
14000 0.860 0.550 0.510 0.000 0.000
14500 0.920 0.590 0.540 0.000 0.000
15000 0.990 0.620 0.590 0.000 0.000
15500 1.050 0.660 0.620 0.000 0.000
16000 1.120 0.710 0.690 0.000 0.000
16500 1.190 0.770 0.730 0.000 0.000
17000 1.260 0.810 0.800 0.000 0.000
17500 1.340 0.880 0.850 0.000 0.000
18000 1.420 0.940 0.910 0.000 0.000
18500 1.490 1.000 0.970 0.000 0.000
19000 1.580 1.060 1.040 0.000 0.000
19500 1.660 1.130 1.100 0.000 0.000
20000 1.740 1.200 1.170 0.000 0.000
20500 1.840 1.260 1.240 0.000 0.000
21000 1.930 1.330 1.310 0.000 0.000
21500 2.040 1.390 1.360 0.000 0.000
22000 2.130 1.480 1.460 0.000 0.000
22500 2.290 2.030 2.330 0.000 0.000

A.1.33 nppsSubC_32f_I test program

//-----------------------------------------------------------------------------------------//
//NPPSSubC32f_I_CPUCheck_Time.cu
//Nathan Clem
//Compile and Execute (Make sure in correct directory):
// nvcc -I/usr/local/cuda-5.5/include -L/usr/local/cuda-5.5/lib64
//NPPSSubC32f_I_CPUCheck_Time.cu -lnppc -lnppi -lnpps -lm
// ./a.out
////-----------------------------------------------------------------------------------------//
#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

//Defined Constant.
#define Constant 10

//File to store CPU and GPU run times.
FILE *TimeFile = fopen("NPPSSubC32fI_Times.txt", "w");

//Use to stop program in case of infinite loop
//Enter: "CTRL + C"
void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

//Cuda Error Check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
    if (code != cudaSuccess)
    {
        fprintf(stderr,"GPuassert: %s %s %d\n", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}

int main(int argc, char** argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    int PointBound = 100;
    signal(SIGINT, signalHandler);
    int i;
    clock_t Start, End;
    double Time;
    int Number_of_Errors;
    float Percent_Error;
    while(1)
    {
        srand(time(0));
        printf("Rows: %i\n", rows);
        printf("Columns: %i\n", cols);
        fprintf(TimeFile, "%i \n", rows);
        //--------------------------------------------------
        float* h_A;
        float* h_A0;
        float* h_A1;
        //HOST ONLY VARIABLES.
        h_A = (float*)malloc(rows*cols*sizeof(float));
        if (h_A == 0)
        {
            fprintf(stderr, "!!! host memory allocation error (h_A)\n");
            return EXIT_FAILURE;
        }
        //HOST VARIABLES USED FOR DEVICE 0.
        h_A0 = (float*)malloc(rows*cols*sizeof(float));
        if (h_A0 == 0)
        {
            fprintf(stderr, "!!! host memory allocation error (h_A0)\n");
            return EXIT_FAILURE;
        }
        //HOST VARIABLES USED FOR DEVICE 1.
        h_A1 = (float*)malloc(rows*cols*sizeof(float));
        if (h_A1 == 0)
        {

```
fprintf(stderr, "!!!! host memory allocation error (h_A1)\n");
return EXIT_FAILURE;
}
//Creation of random values.
for (i=0; i < (rows*cols); i++)
{
    h_A[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
}
for (i=0; i < (rows*cols); i++)
{
    h_A0[i] = h_A[i];
    h_A1[i] = h_A[i];
}
//-----------------------------------------------------------------------------//
//Start Time for CPU Calculations.
Start = clock();
// //CPU Calculations // //h_A = h_A - Constant // for (i=0; i < (rows*cols); i++)
{
    h_A[i] = h_A[i] - Constant;
} // //End Time for CPU Calculations.
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------//
//Start Time for GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();
// float* d_A0; // Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void**)&d_A0, rows*cols*sizeof(float)) );
// //Transfer Data from HOST to DEVICE. 
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
// //Subtract Constant Inplace (nppsSubC_32f_I) // //d_A = d_A - Constant // nppsSubC_32f_I(Constant, d_A0, rows*cols);
// //Transfer Data from DEVICE to HOST. 
CudaErrCheck( cudaMemcpy(h_A0, d_A0, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
// //End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------//
//Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();
// float* d_A1; // Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void**)&d_A1, rows*cols*sizeof(float)) );
// //Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );

// Subtract Constant Inplace (nppsSubC_32f_I)
// d_A = d_A - Constant
nppsSubC_32f_I(Constant, d_A1, rows*cols);

// Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_A1, d_A1, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );

// End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
// Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_A[i] - h_A0[i]) != 0) 
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\%\n", Percent_Error);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_A[i] - h_A1[i]) != 0) 
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f\n", Percent_Error);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
printf("Tesla C1060 Error Percentage: %.3f\%
", Percent_Error);
//HOST MEMORY CLEAN UP.
free(h_A);
free(h_A0);
free(h_A1);
//DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
//DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
// Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;
//-------------------------------
//END PROGRAM
return EXIT_SUCCESS;
//-------------------------------
# A.1.34 nppsSubC_32f_I raw data

<table>
<thead>
<tr>
<th>nppsSubC_32f_I</th>
<th>0.000</th>
<th>0.070</th>
<th>0.060</th>
<th>0.000</th>
<th>0.000</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1500</td>
<td>0.010</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2000</td>
<td>0.030</td>
<td>0.010</td>
<td>0.020</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2500</td>
<td>0.040</td>
<td>0.020</td>
<td>0.020</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3000</td>
<td>0.050</td>
<td>0.020</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3500</td>
<td>0.060</td>
<td>0.030</td>
<td>0.030</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>4000</td>
<td>0.070</td>
<td>0.040</td>
<td>0.040</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>4500</td>
<td>0.090</td>
<td>0.050</td>
<td>0.050</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>5000</td>
<td>0.110</td>
<td>0.050</td>
<td>0.050</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>5500</td>
<td>0.130</td>
<td>0.060</td>
<td>0.060</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>6000</td>
<td>0.150</td>
<td>0.070</td>
<td>0.070</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>6500</td>
<td>0.170</td>
<td>0.090</td>
<td>0.080</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>7000</td>
<td>0.200</td>
<td>0.110</td>
<td>0.090</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>7500</td>
<td>0.230</td>
<td>0.120</td>
<td>0.110</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>8000</td>
<td>0.250</td>
<td>0.140</td>
<td>0.120</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>8500</td>
<td>0.280</td>
<td>0.150</td>
<td>0.140</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>9000</td>
<td>0.300</td>
<td>0.170</td>
<td>0.160</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>9500</td>
<td>0.350</td>
<td>0.200</td>
<td>0.170</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10000</td>
<td>0.390</td>
<td>0.220</td>
<td>0.200</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10500</td>
<td>0.430</td>
<td>0.250</td>
<td>0.220</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>11000</td>
<td>0.470</td>
<td>0.280</td>
<td>0.250</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>11500</td>
<td>0.510</td>
<td>0.300</td>
<td>0.280</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>12000</td>
<td>0.560</td>
<td>0.320</td>
<td>0.300</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>12500</td>
<td>0.610</td>
<td>0.340</td>
<td>0.320</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>13000</td>
<td>0.650</td>
<td>0.360</td>
<td>0.340</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>13500</td>
<td>0.700</td>
<td>0.390</td>
<td>0.360</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>14000</td>
<td>0.750</td>
<td>0.420</td>
<td>0.380</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>14500</td>
<td>0.810</td>
<td>0.450</td>
<td>0.400</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>15000</td>
<td>0.880</td>
<td>0.500</td>
<td>0.420</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>15500</td>
<td>0.960</td>
<td>0.560</td>
<td>0.450</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>16000</td>
<td>1.040</td>
<td>0.620</td>
<td>0.480</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>16500</td>
<td>1.060</td>
<td>0.700</td>
<td>0.500</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>17000</td>
<td>1.130</td>
<td>0.760</td>
<td>0.550</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>17500</td>
<td>1.180</td>
<td>0.830</td>
<td>0.600</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>18000</td>
<td>1.240</td>
<td>0.900</td>
<td>0.650</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>18500</td>
<td>1.290</td>
<td>0.980</td>
<td>0.700</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>19000</td>
<td>1.340</td>
<td>1.060</td>
<td>0.750</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>19500</td>
<td>1.390</td>
<td>1.140</td>
<td>0.800</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>20000</td>
<td>1.440</td>
<td>1.220</td>
<td>0.850</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>20500</td>
<td>1.500</td>
<td>1.300</td>
<td>0.900</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>21000</td>
<td>1.560</td>
<td>1.380</td>
<td>0.950</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>21500</td>
<td>1.620</td>
<td>1.460</td>
<td>1.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>22000</td>
<td>1.680</td>
<td>1.540</td>
<td>1.050</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>22500</td>
<td>1.740</td>
<td>1.620</td>
<td>1.100</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>23000</td>
<td>1.800</td>
<td>1.700</td>
<td>1.150</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>23500</td>
<td>1.860</td>
<td>1.780</td>
<td>1.200</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>24000</td>
<td>1.920</td>
<td>1.860</td>
<td>1.250</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>24500</td>
<td>1.980</td>
<td>1.940</td>
<td>1.300</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>25000</td>
<td>2.040</td>
<td>2.020</td>
<td>1.350</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>25500</td>
<td>2.100</td>
<td>2.100</td>
<td>1.400</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>26000</td>
<td>2.160</td>
<td>2.180</td>
<td>1.450</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>26500</td>
<td>2.220</td>
<td>2.260</td>
<td>1.500</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>27000</td>
<td>2.280</td>
<td>2.320</td>
<td>1.550</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>27500</td>
<td>2.340</td>
<td>2.380</td>
<td>1.600</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>28000</td>
<td>2.400</td>
<td>2.440</td>
<td>1.650</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>28500</td>
<td>2.460</td>
<td>2.500</td>
<td>1.700</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>29000</td>
<td>2.520</td>
<td>2.560</td>
<td>1.750</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>29500</td>
<td>2.580</td>
<td>2.640</td>
<td>1.800</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>30000</td>
<td>2.640</td>
<td>2.700</td>
<td>1.850</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>30500</td>
<td>2.700</td>
<td>2.760</td>
<td>1.900</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>31000</td>
<td>2.760</td>
<td>2.820</td>
<td>1.960</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>31500</td>
<td>2.820</td>
<td>2.880</td>
<td>2.020</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>32000</td>
<td>2.880</td>
<td>2.940</td>
<td>2.080</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>
A.2 cuBLAS Test Programs and Raw Data

Results for the cuBLAS test programs were analyzed in Section 5.1.2. The columns in the raw data are represented by the titles listed in Table A.3.

Table A.3
Column representation of cuBLAS raw data results

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
<th>Column 5</th>
<th>Column 6</th>
<th>Column 7</th>
<th>Column 8</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Columns in Final Result</td>
<td>CPU Time in seconds</td>
<td>cuBLAS Time in seconds</td>
<td>NPP Time in seconds</td>
<td>cuBLAS Percent Error</td>
<td>cuBLAS Average Margin of Error</td>
<td>NPP Percent Error</td>
<td>NPP Average Margin of Error</td>
</tr>
</tbody>
</table>

A.2.1 cublasSgemm execution time test program

```c
#include <cublas.h>
#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

FILE *TimeFile = fopen("CublasSgemm_Times.txt", "w");

void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

int main(int argc, char** argv)
{
    cudaSetDevice(1);
    int cols = 1000000;
    int step = 1000000;
    int PointBound = 100;
    float alpha = 1;
    float beta = 1;
    float prod;
    signal(SIGINT, signalHandler);
    int i, j;
    ```
clock_t Start, End;
double Time;
int Number_of_Errors;
float Percent_Error;
float Margin_of_Error_Ave;
while(cols <= 81000000)
while(1)
{
    srand(time(0));
    printf("Columns: %i\n", cols);
    fprintf(TimeFile, "%i\n", cols);
    float* h_A;
    float* h_B;
    float* h_C;
    float* h_A_Cublas;
    float* h_B_Cublas;
    float* h_C_Cublas;
    float* h_A_NPP;
    float* h_B1_NPP;
    float* h_B2_NPP;
    float* h_B3_NPP;
    float* h_C_NPP;
    //
    //HOST ONLY VARIABLES.
    h_A = (float*)malloc(3*sizeof(float));
    if (h_A == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error (h_A)\n");
        return EXIT_FAILURE;
    }
    h_B = (float*)malloc(3*cols*sizeof(float));
    if (h_B == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error (h_B)\n");
        return EXIT_FAILURE;
    }
    h_C = (float*)malloc(cols*sizeof(float));
    if (h_C == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error (h_C)\n");
        return EXIT_FAILURE;
    }
    //HOST VARIABLES USED FOR CUBLAS SGEMM FUNCTION.
    h_A_Cublas = (float*)malloc(3*sizeof(float));
    if (h_A_Cublas == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error (h_A_Cublas)\n");
        return EXIT_FAILURE;
    }
    h_B_Cublas = (float*)malloc(3*cols*sizeof(float));
    if (h_B_Cublas == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error (h_B_Cublas)\n");
        return EXIT_FAILURE;
    }
    h_C_Cublas = (float*)malloc(cols*sizeof(float));
    if (h_C_Cublas == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error (h_C_Cublas)\n");
        return EXIT_FAILURE;
    }
    //HOST VARIABLES USED FOR NPP FUNCTIONS.
    h_A_NPP = (float*)malloc(3*sizeof(float));
    if (h_A_NPP == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error (h_A_NPP)\n");
        return EXIT_FAILURE;
    }
    h_B1_NPP = (float*)malloc(cols*sizeof(float));
    if (h_B1_NPP == 0)
fprintf (stderr, "!!!! host memory allocation error (h_B1_NPP)\n");
    return EXIT_FAILURE;
}    
    h_B2_NPP = (float*)malloc(cols*sizeof(float));
    if (h_B2_NPP == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error (h_B2_NPP)\n");
        return EXIT_FAILURE;
    }
    h_B3_NPP = (float*)malloc(cols*sizeof(float));
    if (h_B3_NPP == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error (h_B3_NPP)\n");
        return EXIT_FAILURE;
    }
    h_C_NPP = (float*)malloc(cols*sizeof(float));
    if (h_C_NPP == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error (h_C_NPP)\n");
        return EXIT_FAILURE;
    }
    //Creation of random values.
    for (i=0; i < 3; i++)
    {
        h_A[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    }
    for (i=0; i < (3*cols); i++)
    {
        h_B[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    }
    for (i=0; i < cols; i++)
    {
        h_C[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    }
    //Copy values to Cublas host variables
    for (i=0; i < 3; i++)
    {
        h_A_Cublas[i] = h_A[i];
    }
    for (i=0; i < (3*cols); i++)
    {
        h_B_Cublas[i] = h_B[i];
    }
    for (i=0; i < cols; i++)
    {
        h_C_Cublas[i] = h_C[i];
    }
    //Copy values to NPP host variables
    for (i=0; i < 3; i++)
    {
        h_A_NPP[i] = h_A[i];
    }
    j = 0;
    for (i=0; i < (3*cols); i=i+3)
    {
        h_B1_NPP[j] = h_B[i];
        h_B2_NPP[j] = h_B[i+1];
        h_B3_NPP[j] = h_B[i+2];
        j++;
    }
    for (i=0; i < cols; i++)
    {
        h_C_NPP[i] = h_C[i];
    }
    //Start Time for CPU Version of Matrix Multiplication.
    Start = clock();
    //Matrix Multiplication (CPU)
    //h_C = alpha * h_A * h_B + beta * h_C
j = 0;
for (i = 0; i < (3*cols); i=i+3)
{
    prod = 0;
    prod = alpha*(h_A[0]*h_B[i] + h_A[1]*h_B[i+1] + h_A[2]*h_B[i+2]);
    h_C[j] = prod + beta*h_C[j];
    j++;
}

// End Time for CPU Calculations.
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%3.3f ", fabs(Time));
//-----------------------------------------------------------------------------//
//Initialize CUBLAS.
cublasStatus status;
status = cublasInit();
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! CUBLAS initialization error\n");
    return EXIT_FAILURE;
}
//-----------------------------------------------------------------------------//
//Start Time for CUBLAS Version of Matrix Multiplication.
cudaDeviceSynchronize();
Start = clock();
// float* d_A_Cublas;
float* d_B_Cublas;
float* d_C_Cublas;
// Allocate DEVICE memory (GPU) for Matrix A_Cublas.
status = cublasAlloc(3, sizeof(float), (void**)&d_A_Cublas);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (A_Cublas)\n");
    return EXIT_FAILURE;
}
// Allocate DEVICE memory (GPU) for Matrix B_Cublas.
status = cublasAlloc(3*cols, sizeof(float), (void**)&d_B_Cublas);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (B_Cublas)\n");
    return EXIT_FAILURE;
}
// Allocate DEVICE memory (GPU) for Matrix C_Cublas.
status = cublasAlloc(cols, sizeof(float), (void**)&d_C_Cublas);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (C_Cublas)\n");
    return EXIT_FAILURE;
}
// Initialize the DEVICE Matrix A_Cublas with the HOST Matrix A_Cublas.
status = cublasSetMatrix(1, 3, sizeof(float), h_A_Cublas, 1, d_A_Cublas, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write A_Cublas)\n");
    return EXIT_FAILURE;
}
// Initialize the DEVICE Matrix B_Cublas with the HOST Matrix B_Cublas.
status = cublasSetMatrix(3, cols, sizeof(float), h_B_Cublas, 1, d_B_Cublas, 3);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write B_Cublas)\n");
    return EXIT_FAILURE;
}
// Initialize the DEVICE Matrix C_Cublas with the HOST Matrix C_Cublas.
status = cublasSetMatrix(1, cols, sizeof(float), h_C_Cublas, 1, d_C_Cublas, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write C_Cublas)\n");
}
return EXIT_FAILURE;
}

// Matrix Multiplication (cublasSgemm)
//
// h_C = alpha * h_A * h_B + beta * h_C
//
cublasSgemm('N', 'N', 1, cols, 3, alpha, d_A_Cublas, 1, d_B_Cublas, 3, beta,
// d_C_Cublas, 1);

// Retrieve DEVICE Matrix C_Cublas and place it back to HOST Matrix C_Cublas.
status = cublasGetMatrix(1, cols, sizeof(float), d_C_Cublas, 1, h_C_Cublas, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! device access error (C_Cublas)\n");
    return EXIT_FAILURE;
}

// End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));

// DEVICE MEMORY CLEAN UP (CUBLAS SGEMM)
status = cublasFree(d_A_Cublas);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! memory free error (A_Cublas)\n");
    return EXIT_FAILURE;
}
status = cublasFree(d_B_Cublas);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! memory free error (B_Cublas)\n");
    return EXIT_FAILURE;
}
status = cublasFree(d_C_Cublas);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! memory free error (C_Cublas)\n");
    return EXIT_FAILURE;
}

//-----------------------------------------------------------------------------
// Start Time for NPP Version of Matrix Multiplication

//Allocate DEVICE memory (GPU) for Matrix B1_NPP.
status = cublasAlloc(cols, sizeof(float), (void**)&d_B1_NPP);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! device memory allocation error (B1_NPP)\n");
    return EXIT_FAILURE;
}
//Allocate DEVICE memory (GPU) for Matrix B2_NPP.
status = cublasAlloc(cols, sizeof(float), (void**)&d_B2_NPP);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! device memory allocation error (B2_NPP)\n");
    return EXIT_FAILURE;
}
//Allocate DEVICE memory (GPU) for Matrix B3_NPP.
status = cublasAlloc(cols, sizeof(float), (void**)&d_B3_NPP);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! device memory allocation error (B3_NPP)\n");
    return EXIT_FAILURE;
}
return EXIT_FAILURE;
}
//Allocate DEVICE memory (GPU) for Matrix C_NPP.
status = cublasAlloc(cols, sizeof(float), (void**)&d_C_NPP);
if (status != CUBLAS_STATUS_SUCCESS)
{
  fprintf (stderr, "!!!!! device memory allocation error (C_NPP)\n");
  return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Matrix B_NPP.
status = cublasAlloc(cols, sizeof(float), (void**)&d_B_NPP);
if (status != CUBLAS_STATUS_SUCCESS)
{
  fprintf (stderr, "!!!!! device memory allocation error (B_NPP)\n");
  return EXIT_FAILURE;
}

//Matrix Multiplication (NPP)
//
//h_C = alpha * h_A * h_B + beta * h_C
//nppsMulC_32f_I(h_A_NPP[0], d_B1_NPP, cols);
nppsMulC_32f_I(h_A_NPP[1], d_B2_NPP, cols);
nppsMulC_32f_I(h_A_NPP[2], d_B3_NPP, cols);

//nppsAdd_32f_I(d_B1_NPP, d_B2_NPP, cols);
nppsAdd_32f_I(d_B2_NPP, d_B3_NPP, cols);

//nppsMulC_32f_I(alpha, d_B3_NPP, cols);
nppsMulC_32f_I(beta, d_C_NPP, cols);

//Retrieve DEVICE Matrix C_NPP and place it back to HOST Matrix C_NPP.
status = cublasGetMatrix(1, cols, sizeof(float), d_C_NPP, 1, h_C_NPP, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
  fprintf (stderr, "!!!!! device access error (C_NPP)\n");
  return EXIT_FAILURE;
}

//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, ".%3f ", fabs(Time));
//DEVICE MEMORY CLEAN UP (NPP)
status = cublasFree(d_B1_NPP);
if (status != CUBLAS_STATUS_SUCCESS)
status = cublasFree(d_B2_NPP);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (B2_NPP)\n");
    return EXIT_FAILURE;
}
status = cublasFree(d_B3_NPP);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (B3_NPP)\n");
    return EXIT_FAILURE;
}
status = cublasFree(d_C_NPP);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (C_NPP)\n");
    return EXIT_FAILURE;
}
//-----------------------------------------------------------------------------//
//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
Margin_of_Error_Ave = 0;
for (i=0; i < cols; i++)
{
    if ((h_C[i] - h_C_Cublas[i]) != 0)
    {
        Number_of_Errors++;
        Margin_of_Error_Ave = Margin_of_Error_Ave + abs(h_C[i] - h_C_Cublas[i]);
    }
}
Percent_Error = 100.0*Number_of_Errors/cols;
fprintf(TimeFile, "%3f ", Percent_Error);
printf("Number of Calculation Errors for Cublas Sgemm: %i\n", Number_of_Errors);
printf("Cublas Sgemm Error Percentage: %.3f\n", Percent_Error);
printf("Cublas Sgemm Error Margin: %f\n", Margin_of_Error_Ave/Number_of_Errors);
Number_of_Errors = 0;
Percent_Error = 0;
Margin_of_Error_Ave = 0;
for (i=0; i < cols; i++)
{
    if ((h_C[i] - h_C_NPP[i]) != 0)
    {
        Number_of_Errors++;
        Margin_of_Error_Ave = Margin_of_Error_Ave + abs(h_C[i] - h_C_NPP[i]);
    }
}
Percent_Error = 100.0*Number_of_Errors/cols;
fprintf(TimeFile, "%3f ", Percent_Error);
printf("Number of Calculation Errors for NPP: %i\n", Number_of_Errors);
printf("NPP Error Percentage: %.3f\n", Percent_Error);
printf("NPP Error Margin: %f\n\n", Margin_of_Error_Ave/Number_of_Errors);
//-----------------------------------------------------------------------------//
//HOST MEMORY CLEAN UP.
free(h_A);
free(h_B);
free(h_C);
free(h_A_Cublas);
free(h_B_Cublas);
free(h_C_Cublas);
free(h_A_NPP);
free(h_B1_NPP);
free(h_B2_NPP);
free(h_B3_NPP);
free(h_C_NPP);
//-------------------------------------------------------------------------------
//Shutdown CUBLAS
status = cublasShutdown();
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! shutdown error\n");
    return EXIT_FAILURE;
}
//-------------------------------------------------------------------------------
//Increasing of Matrix Dimensions.
cols = cols + step;
//-------------------------------------------------------------------------------
//END PROGRAM
return EXIT_SUCCESS;
//-------------------------------------------------------------------------------

A.2.2  cublasSgemm execution time raw data

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>0.010</td>
<td>0.010</td>
<td>0.180</td>
<td>48.492</td>
<td>0.000642</td>
</tr>
<tr>
<td>2000000</td>
<td>0.020</td>
<td>0.020</td>
<td>0.020</td>
<td>41.458</td>
<td>0.000516</td>
</tr>
<tr>
<td>3000000</td>
<td>0.020</td>
<td>0.030</td>
<td>0.020</td>
<td>41.445</td>
<td>0.000517</td>
</tr>
<tr>
<td>4000000</td>
<td>0.040</td>
<td>0.060</td>
<td>0.020</td>
<td>41.475</td>
<td>0.000516</td>
</tr>
<tr>
<td>5000000</td>
<td>0.050</td>
<td>0.060</td>
<td>0.020</td>
<td>49.991</td>
<td>0.000330</td>
</tr>
<tr>
<td>6000000</td>
<td>0.060</td>
<td>0.070</td>
<td>0.040</td>
<td>41.662</td>
<td>0.000307</td>
</tr>
<tr>
<td>7000000</td>
<td>0.070</td>
<td>0.080</td>
<td>0.020</td>
<td>41.669</td>
<td>0.000307</td>
</tr>
<tr>
<td>8000000</td>
<td>0.070</td>
<td>0.100</td>
<td>0.020</td>
<td>45.586</td>
<td>0.000415</td>
</tr>
<tr>
<td>9000000</td>
<td>0.090</td>
<td>0.100</td>
<td>0.050</td>
<td>32.137</td>
<td>0.000549</td>
</tr>
<tr>
<td>10000000</td>
<td>0.100</td>
<td>0.120</td>
<td>0.040</td>
<td>41.807</td>
<td>0.000278</td>
</tr>
<tr>
<td>11000000</td>
<td>0.100</td>
<td>0.120</td>
<td>0.060</td>
<td>36.177</td>
<td>0.000415</td>
</tr>
<tr>
<td>12000000</td>
<td>0.110</td>
<td>0.140</td>
<td>0.070</td>
<td>44.415</td>
<td>0.000580</td>
</tr>
<tr>
<td>13000000</td>
<td>0.130</td>
<td>0.150</td>
<td>0.070</td>
<td>41.259</td>
<td>0.000462</td>
</tr>
<tr>
<td>14000000</td>
<td>0.130</td>
<td>0.160</td>
<td>0.070</td>
<td>44.298</td>
<td>0.000705</td>
</tr>
<tr>
<td>15000000</td>
<td>0.140</td>
<td>0.160</td>
<td>0.070</td>
<td>46.575</td>
<td>0.000593</td>
</tr>
<tr>
<td>16000000</td>
<td>0.150</td>
<td>0.190</td>
<td>0.070</td>
<td>31.972</td>
<td>0.000398</td>
</tr>
<tr>
<td>17000000</td>
<td>0.170</td>
<td>0.190</td>
<td>0.090</td>
<td>33.449</td>
<td>0.000399</td>
</tr>
<tr>
<td>18000000</td>
<td>0.170</td>
<td>0.210</td>
<td>0.090</td>
<td>47.393</td>
<td>0.000409</td>
</tr>
<tr>
<td>19000000</td>
<td>0.180</td>
<td>0.230</td>
<td>0.080</td>
<td>45.296</td>
<td>0.000654</td>
</tr>
<tr>
<td>20000000</td>
<td>0.190</td>
<td>0.220</td>
<td>0.110</td>
<td>41.380</td>
<td>0.000553</td>
</tr>
<tr>
<td>21000000</td>
<td>0.210</td>
<td>0.240</td>
<td>0.100</td>
<td>44.782</td>
<td>0.000607</td>
</tr>
<tr>
<td>22000000</td>
<td>0.210</td>
<td>0.250</td>
<td>0.110</td>
<td>33.951</td>
<td>0.000294</td>
</tr>
<tr>
<td>23000000</td>
<td>0.220</td>
<td>0.270</td>
<td>0.110</td>
<td>21.659</td>
<td>0.000187</td>
</tr>
<tr>
<td>24000000</td>
<td>0.230</td>
<td>0.270</td>
<td>0.130</td>
<td>47.017</td>
<td>0.000561</td>
</tr>
<tr>
<td>25000000</td>
<td>0.250</td>
<td>0.290</td>
<td>0.120</td>
<td>47.809</td>
<td>0.000420</td>
</tr>
<tr>
<td>26000000</td>
<td>0.250</td>
<td>0.290</td>
<td>0.120</td>
<td>14.132</td>
<td>0.000166</td>
</tr>
<tr>
<td>27000000</td>
<td>0.250</td>
<td>0.310</td>
<td>0.130</td>
<td>42.643</td>
<td>0.000570</td>
</tr>
<tr>
<td>28000000</td>
<td>0.260</td>
<td>0.320</td>
<td>0.140</td>
<td>45.726</td>
<td>0.000458</td>
</tr>
<tr>
<td>29000000</td>
<td>0.290</td>
<td>0.330</td>
<td>0.140</td>
<td>47.172</td>
<td>0.000586</td>
</tr>
<tr>
<td>30000000</td>
<td>0.290</td>
<td>0.340</td>
<td>0.150</td>
<td>36.120</td>
<td>0.000314</td>
</tr>
<tr>
<td>31000000</td>
<td>0.300</td>
<td>0.350</td>
<td>0.160</td>
<td>44.887</td>
<td>0.000542</td>
</tr>
<tr>
<td>32000000</td>
<td>0.300</td>
<td>0.370</td>
<td>0.160</td>
<td>33.421</td>
<td>0.000425</td>
</tr>
<tr>
<td>33000000</td>
<td>0.330</td>
<td>0.380</td>
<td>0.170</td>
<td>44.821</td>
<td>0.000530</td>
</tr>
<tr>
<td>34000000</td>
<td>0.320</td>
<td>0.390</td>
<td>0.170</td>
<td>49.226</td>
<td>0.000591</td>
</tr>
<tr>
<td>35000000</td>
<td>0.330</td>
<td>0.400</td>
<td>0.180</td>
<td>49.904</td>
<td>0.000380</td>
</tr>
<tr>
<td>36000000</td>
<td>0.340</td>
<td>0.420</td>
<td>0.180</td>
<td>36.750</td>
<td>0.000369</td>
</tr>
<tr>
<td>37000000</td>
<td>0.380</td>
<td>0.420</td>
<td>0.180</td>
<td>48.386</td>
<td>0.000518</td>
</tr>
<tr>
<td>38000000</td>
<td>0.360</td>
<td>0.440</td>
<td>0.190</td>
<td>50.346</td>
<td>0.000530</td>
</tr>
<tr>
<td>39000000</td>
<td>0.370</td>
<td>0.440</td>
<td>0.190</td>
<td>46.487</td>
<td>0.000682</td>
</tr>
<tr>
<td>40000000</td>
<td>0.380</td>
<td>0.440</td>
<td>0.200</td>
<td>45.455</td>
<td>0.000445</td>
</tr>
<tr>
<td>41000000</td>
<td>0.420</td>
<td>0.460</td>
<td>0.210</td>
<td>47.783</td>
<td>0.000443</td>
</tr>
<tr>
<td>42000000</td>
<td>0.400</td>
<td>0.470</td>
<td>0.220</td>
<td>43.559</td>
<td>0.000435</td>
</tr>
<tr>
<td>43000000</td>
<td>0.410</td>
<td>0.470</td>
<td>0.220</td>
<td>50.182</td>
<td>0.000440</td>
</tr>
<tr>
<td>44000000</td>
<td>0.420</td>
<td>0.500</td>
<td>0.220</td>
<td>46.468</td>
<td>0.000644</td>
</tr>
<tr>
<td>45000000</td>
<td>0.460</td>
<td>0.510</td>
<td>0.230</td>
<td>32.601</td>
<td>0.000206</td>
</tr>
<tr>
<td>46000000</td>
<td>0.440</td>
<td>0.520</td>
<td>0.230</td>
<td>36.363</td>
<td>0.000482</td>
</tr>
<tr>
<td>47000000</td>
<td>0.450</td>
<td>0.530</td>
<td>0.240</td>
<td>47.873</td>
<td>0.000382</td>
</tr>
<tr>
<td>48000000</td>
<td>0.450</td>
<td>0.540</td>
<td>0.250</td>
<td>46.726</td>
<td>0.000196</td>
</tr>
</tbody>
</table>
//-----------------------------------------------------------------------------------------//
//CublasSgemm_Only.cu
//Nathan Clem
//
//Compile and Execute (Make sure in correct directory):
// nvcc -I/usr/local/cuda-5.5/include -L/usr/local/cuda-5.5/lib64 CublasSgemm_Only.cu -lm -lcublas
// ./a.out
//-----------------------------------------------------------------------------------------//
#include <cublas.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
//Use to stop program in case of infinite loop

A.2.3 cuBLAS only cublasSgemm test program
//Enter: "CTRL + C"
void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    exit(signum);
}

//Main
int main(int argc, char** argv)
{
    cudaMemcpy1(1);
    int cols = 10000000;
    int step = 1000000;
    int PointBound = 100;
    float alpha = 1;
    float beta = 1;
    signal(SIGINT, signalHandler);
    int i;

    while(cols <= 21000000)
    {
        srand(time(0));
        printf("Columns: %i\n", cols);
        //HOST VARIABLES USED FOR CUBLAS SGEMM FUNCTION.
        float* h_A_Cublas;
        float* h_B_Cublas;
        float* h_C_Cublas;
        h_A_Cublas = (float*)malloc(3*sizeof(float));
        if (h_A_Cublas == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A_Cublas)\n");
            return EXIT_FAILURE;
        }
        h_B_Cublas = (float*)malloc(3*cols*sizeof(float));
        if (h_B_Cublas == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_B_Cublas)\n");
            return EXIT_FAILURE;
        }
        h_C_Cublas = (float*)malloc(cols*sizeof(float));
        if (h_C_Cublas == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_C_Cublas)\n");
            return EXIT_FAILURE;
        }
        //Creation of random values.
        for (i=0; i < 3*cols; i++)
        {
            h_A_Cublas[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
        }
        for (i=0; i < 3*cols; i++)
        {
            h_B_Cublas[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
        }
        for (i=0; i < cols; i++)
        {
            h_C_Cublas[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
        }
        //Initialize CUBLAS.
        cublasStatus status;
        status = cublasInit();
        if (status != CUBLAS_STATUS_SUCCESS)
        {
            fprintf (stderr, "!!!! CUBLAS initialization error\n");
            return EXIT_FAILURE;
        }
        //Cublas Version of Matrix Multiplication.
float* d_A_Cublas;
float* d_B_Cublas;
float* d_C_Cublas;

// Allocate DEVICE memory (GPU) for Matrix A_Cublas.
status = cublasAlloc(3, sizeof(float), (void**)&d_A_Cublas);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (A_Cublas)\n");
    return EXIT_FAILURE;
}

// Allocate DEVICE memory (GPU) for Matrix B_Cublas.
status = cublasAlloc(cols, sizeof(float), (void**)&d_B_Cublas);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (B_Cublas)\n");
    return EXIT_FAILURE;
}

// Allocate DEVICE memory (GPU) for Matrix C_Cublas.
status = cublasAlloc(cols, sizeof(float), (void**)&d_C_Cublas);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (C_Cublas)\n");
    return EXIT_FAILURE;
}

// Initialize the DEVICE Matrix A_Cublas with the HOST Matrix A_Cublas.
status = cublasSetMatrix(1, 3, sizeof(float), h_A_Cublas, 1, d_A_Cublas, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write A_Cublas)\n");
    return EXIT_FAILURE;
}

// Initialize the DEVICE Matrix B_Cublas with the HOST Matrix B_Cublas.
status = cublasSetMatrix(cols, 3, sizeof(float), h_B_Cublas, 3, d_B_Cublas, 3);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write B_Cublas)\n");
    return EXIT_FAILURE;
}

// Initialize the DEVICE Matrix C_Cublas with the HOST Matrix C_Cublas.
status = cublasSetMatrix(1, cols, sizeof(float), h_C_Cublas, 1, d_C_Cublas, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write C_Cublas)\n");
    return EXIT_FAILURE;
}

// Matrix Multiplication (cublasSgemm)
//
h_C = alpha * h_A + beta * h_C
// cublasSgemm('N', 'N', 1, cols, 3, alpha, d_A_Cublas, 1, d_B_Cublas, 3, beta, d_C_Cublas, 1);

// Retrieve DEVICE Matrix C_Cublas and place it back to HOST Matrix C_Cublas.
status = cublasGetMatrix(1, cols, sizeof(float), d_C_Cublas, 1, h_C_Cublas, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (C_Cublas)\n");
    return EXIT_FAILURE;
}

// DEVICE MEMORY CLEAN UP (CUBLAS SGEMM)
status = cublasFree(d_A_Cublas);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (A_Cublas)\n");
    return EXIT_FAILURE;
}
status = cublasFree(d_B_Cublas);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!! memory free error (B_Cublas)\n");
    return EXIT_FAILURE;
}
status = cublasFree(d_C_Cublas);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!! memory free error (C_Cublas)\n");
    return EXIT_FAILURE;
}
//-----------------------------------------------------------------------------\n//HOST MEMORY CLEAN UP.
free(h_A_Cublas);
free(h_B_Cublas);
free(h_C_Cublas);
//-----------------------------------------------------------------------------\n//Shutdown CUBLAS
status = cublasShutdown();
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!! shutdown error\n");
    return EXIT_FAILURE;
}
//-----------------------------------------------------------------------------\n//Increasing of Matrix Dimensions.
cols = cols + step;
//-----------------------------------------------------------------------------\n
}ORAGE OF PROGRAM
return EXIT_SUCCESS;
//-----------------------------------------------------------------------------\n
A.2.4 NPP only cublasSgemm test program

//CublasSgemm_NPP_Only.cu
//Nathan Clem
//Compile and Execute (Make sure in correct directory):
// nvcc -I/usr/local/cuda-5.5/include -L/usr/local/cuda-5.5/lib64 CublasSgemm_NPP_Only.cu -lnppc -lnppi -lnpps -lm
// ./a.out

#include <npps.h>
#include <npi.h>
#include <csignal>

#include "npps.h"
#include "npi.h"
#include "stdio.h"
#include "stdlib.h"
#include "signal.h"
#include <npps.h>
#include <npi.h>
#include <csignal>

//Use to stop program in case of infinite loop
//Enter: "CTRL + C"
void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    exit(signum);
}

#include <npps.h>
>Loading libraries... done.

CUDA Error Check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
    if (code != cudaSuccess)
    {
        fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}
int main(int argc, char** argv)
{
    cudaSetDevice(1);
    int cols = 10000000;
    int step = 1000000;
    int PointBound = 100;
    float alpha = 1;
    float beta = 1;
    signal(SIGINT, signalHandler);
    int i;
    while(cols <= 21000000)
    {
        srand(time(0));
        printf("Columns: %i\n", cols);
        float* h_A_NPP;
        float* h_B1_NPP;
        float* h_B2_NPP;
        float* h_B3_NPP;
        float* h_C_NPP;
        //HOST VARIABLES USED FOR NPP FUNCTIONS.
        h_A_NPP = (float*)malloc(3*sizeof(float));
        if (h_A_NPP == 0)
        {
            fprintf(stderr, "!!!! host memory allocation error (h_A_NPP)\n");
            return EXIT_FAILURE;
        }
        h_B1_NPP = (float*)malloc(cols*sizeof(float));
        if (h_B1_NPP == 0)
        {
            fprintf(stderr, "!!!! host memory allocation error (h_B1_NPP)\n");
            return EXIT_FAILURE;
        }
        h_B2_NPP = (float*)malloc(cols*sizeof(float));
        if (h_B2_NPP == 0)
        {
            fprintf(stderr, "!!!! host memory allocation error (h_B2_NPP)\n");
            return EXIT_FAILURE;
        }
        h_B3_NPP = (float*)malloc(cols*sizeof(float));
        if (h_B3_NPP == 0)
        {
            fprintf(stderr, "!!!! host memory allocation error (h_B3_NPP)\n");
            return EXIT_FAILURE;
        }
        h_C_NPP = (float*)malloc(cols*sizeof(float));
        if (h_C_NPP == 0)
        {
            fprintf(stderr, "!!!! host memory allocation error (h_C_NPP)\n");
            return EXIT_FAILURE;
        }
        //Creation of random values.
        for (i=0; i < 3; i++)
        {
            h_A_NPP[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
        }
        for (i=0; i < cols; i++)
        {
            h_B1_NPP[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
            h_B2_NPP[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
            h_B3_NPP[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
        }
        for (i=0; i < cols; i++)
        {
            h_C_NPP[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
        }
    }
}

//NPP Version of Matrix Multiplication

//Main

float* d_B1_NPP;
float* d_B2_NPP;
float* d_B3_NPP;
float* d_C_NPP;

// Cuda Allocation of device memory.
CudaErrCheck( cudaMalloc((void **)&d_B1_NPP, cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_B2_NPP, cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_B3_NPP, cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_C_NPP, cols*sizeof(float)) );

// Transferring from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_B1_NPP, h_B1_NPP, cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B2_NPP, h_B1_NPP, cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B3_NPP, h_B1_NPP, cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_C_NPP, h_B1_NPP, cols*sizeof(float), cudaMemcpyHostToDevice) );

// Matrix Multiplication (NPP)
//
// h_C = alpha * h_A * h_B + beta * h_C
// nppsMulC_32f_I(h_A_NPP[0], d_B1_NPP, cols);
nppsMulC_32f_I(h_A_NPP[1], d_B2_NPP, cols);
nppsMulC_32f_I(h_A_NPP[2], d_B3_NPP, cols);
//
nppsAdd_32f_I(d_B1_NPP, d_B2_NPP, cols);
nppsAdd_32f_I(d_B2_NPP, d_B3_NPP, cols);
//
nppsMulC_32f_I(alpha, d_B3_NPP, cols);
nppsMulC_32f_I(beta, d_C_NPP, cols);
//
nppsAdd_32f_I(d_B3_NPP, d_C_NPP, cols);

// Transferring from DEVICE d_Cuda to HOST h_Cuda.
CudaErrCheck( cudaMemcpy(h_C_NPP, d_C_NPP, cols*sizeof(float), cudaMemcpyDeviceToHost) );

// DEVICE MEMORY CLEAN UP.
CudaErrCheck( cudaFree(d_B1_NPP) );
CudaErrCheck( cudaFree(d_B2_NPP) );
CudaErrCheck( cudaFree(d_B3_NPP) );
CudaErrCheck( cudaFree(d_C_NPP) );

// HOST MEMORY CLEAN UP.
free(h_A_NPP);
free(h_B1_NPP);
free(h_B2_NPP);
free(h_B3_NPP);
free(h_C_NPP);

// Increasing of Matrix Dimensions.
cols = cols + step;

}

// END PROGRAM
return EXIT_SUCCESS;


A.3 Thrust Test Programs and Raw Data

Results for the Thrust test programs were analyzed in Section 5.1.3. The columns in the raw data are represented by the titles listed in Table A.4.

Table A.4
Column representation for Thrust raw data results

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
<th>Column 5</th>
<th>Column 6</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Rows and Columns</td>
<td>CPU Time in seconds</td>
<td>Quadro Time in seconds</td>
<td>Tesla Time in seconds</td>
<td>Quadro Percent Error</td>
<td>Tesla Percent Error</td>
</tr>
</tbody>
</table>

A.3.1 Thrust count_if test program

```cpp
#include <thrust/device_ptr.h>
#include <thrust/count.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

//File to store CPU and GPU run times.
FILE *TimeFile = fopen("ThrustCountIf_Times.txt", "w");

//Structure predicate used in Thrust functions to find values equal to zero.
struct equal_to_zero
{
    __host__ __device__
    bool operator()(const float x)
    {
        return x == 0;
    }
};

//Structures error check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
    if (code != cudaSuccess) {
        CudaErrCheck(code);
        printf("Error: %s
", cudaGetErrorString(code));
    }
}
```
if (code != cudaSuccess)
{
    fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
    if (abort) exit(code);
}

//-----------------------------------------------------------------------------------------//
//Main
int main(int argc, char** argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    signal(SIGINT, signalHandler);
    int i;
    clock_t Start, End;
    double Time;
    int Number_of_Errors;
    float Percent_Error;
    while(1)
    {
        srand(time(0));
        printf("Rows: %i\n", rows);
        printf("Columns: %i\n", cols);
        fprintf(TimeFile, "%i \n", rows);
        //---------------------------------------------------------------------//
        float Temp;
        unsigned char* h_A;
        int result = 0;
        unsigned char* h_A0;
        int result0 = 0;
        unsigned char* h_A1;
        int result1 = 0;
        // //HOST ONLY VARIABLES.
        h_A = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
        if (h_A == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A)\n");
            return EXIT_FAILURE;
        }
        //HOST VARIABLES USED FOR DEVICE 0.
        h_A0 = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
        if (h_A0 == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A0)\n");
            return EXIT_FAILURE;
        }
        //HOST VARIABLES USED FOR DEVICE 1.
        h_A1 = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
        if (h_A1 == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A1)\n");
            return EXIT_FAILURE;
        }
        for (i=0; i<(rows*cols); i++)
        {
            Temp = rand() % 2;
            if (Temp == 0)
            {
                h_A[i] = 0;
            }
            else
            {
                h_A[i] = 255;
            }
        }
        for (i=0; i < (rows*cols); i++)
        {
            h_A0[i] = h_A[i];
            h_A1[i] = h_A[i];
        }
    }
}
//Start Time for CPU Calculations.
Start = clock();
//
//CPU Calculations
//
//Count if element is equal to 0.
//
for (i=0; i < (rows*cols); i++)
{
    if (h_A[i] == 0)
    {
        result++;
    }
}
//
//End Time for CPU Calculations.
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));

//Start Time for GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();
//
unsigned char* d_A0;
//
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void**)&d_A0, rows*cols*sizeof(unsigned char)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(unsigned char), cudaMemcpyHostToDevice) );
//
//Count if element is equal to 0. (thrust::count_if)
//
thrust::device_ptr<unsigned char> dev_ptr_A0(d_A0);
result0 = thrust::count_if(dev_ptr_A0, dev_ptr_A0 + (rows*cols), equal_to_zero());
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));

//Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();
//
unsigned char* d_A1;
//
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void**)&d_A1, rows*cols*sizeof(unsigned char)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, rows*cols*sizeof(unsigned char), cudaMemcpyHostToDevice) );
//
//Count if element is equal to 0. (thrust::count_if)
//
thrust::device_ptr<unsigned char> dev_ptr_A1(d_A1);
result1 = thrust::count_if(dev_ptr_A1, dev_ptr_A1 + (rows*cols), equal_to_zero());
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));

//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
if (result == result0) {
    Number_of_Errors = 0;
    Percent_Error = 0;
} else {
    Number_of_Errors = abs(result - result0);
    Percent_Error = 100.0*(Number_of_Errors/result);
}
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\%
", Percent_Error);
Number_of_Errors = 0;
Percent_Error = 0;
if (result == result1) {
    Number_of_Errors = 0;
    Percent_Error = 0;
} else {
    Number_of_Errors = abs(result - result1);
    Percent_Error = 100.0*(Number_of_Errors/result);
}
fprintf(TimeFile, "%.3f\n", Percent_Error);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
printf("Tesla C1060 Error Percentage: %.3f\%

", Percent_Error);
//-----------------------------------------------------------------------------//
//HOST MEMORY CLEAN UP.
free(h_A);
free(h_A0);
free(h_A1);
//DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
//DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
//-----------------------------------------------------------------------------//
//Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;
//-----------------------------------------------------------------------------//
}  
------------------------------------------------------------------------------------
//END PROGRAM
return EXIT_SUCCESS;
------------------------------------------------------------------------------------

A.3.2 Thrust count_if raw data

500 0.010 0.000 0.000 0.000 0.000 0.000
1000 0.000 0.000 0.010 0.000 0.000 0.000
1500 0.010 0.000 0.000 0.000 0.000 0.000
2000 0.030 0.000 0.000 0.000 0.000 0.000
2500 0.040 0.000 0.000 0.000 0.000 0.000
3000 0.060 0.000 0.000 0.000 0.000 0.000
3500 0.070 0.000 0.000 0.000 0.000 0.000
4000 0.100 0.010 0.000 0.000 0.000 0.000
4500 0.120 0.010 0.000 0.000 0.000 0.000
5000 0.150 0.010 0.010 0.000 0.000 0.000
5500 0.190 0.010 0.000 0.000 0.000 0.000
6000 0.230 0.010 0.010 0.000 0.000 0.000
6500 0.270 0.010 0.020 0.000 0.000 0.000
7000 0.310 0.010 0.020 0.000 0.000 0.000
7500 0.350 0.020 0.020 0.000 0.000 0.000
A.3.3 Thrust remove_copy_if test program

#include <thrust/device_ptr.h>
#include <thrust/remove.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

//File to store CPU and GPU run times.
FILE *TimeFile = fopen("ThrustRemoveCopyIf_Times.txt", "w");

struct equal_to_zero
{
    __host__ __device__
    bool operator()(const float x)
    {
        return x == 0;
    }
};

void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

int main(int argc, char** argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    int whole;
    int half;
    signal(SIGINT, signalHandler);
    int i;
    clock_t Start, End;
    double Time;
    int Number_of_Errors;

float Percent_Error;
while(1) {
    srand(time(0));
    printf("Rows: %i\n", rows);
    printf("Columns: %i\n", cols);
    fprintf(TimeFile, "%i ", rows);
    whole = rows*cols;
    half = (rows*cols)/2;
    //------------------------------------------------------------------
    int* h_A;
    unsigned char* h_B;
    int* h_C;
    int* h_A0;
    unsigned char* h_B0;
    int* h_C0;
    int* h_A1;
    unsigned char* h_B1;
    int* h_C1;
    //HOST ONLY VARIABLES.
    h_A = (int*)malloc(whole*sizeof(int));
    if (h_A == 0) {
        fprintf (stderr, "!!!! host memory allocation error (h_A)\n");
        return EXIT_FAILURE;
    }
    h_B = (unsigned char*)malloc(whole*sizeof(unsigned char));
    if (h_B == 0) {
        fprintf (stderr, "!!!! host memory allocation error (h_B)\n");
        return EXIT_FAILURE;
    }
    h_C = (int*)malloc(half*sizeof(int));
    if (h_C == 0) {
        fprintf (stderr, "!!!! host memory allocation error (h_C)\n");
        return EXIT_FAILURE;
    }
    //HOST VARIABLES USED FOR DEVICE 0.
    h_A0 = (int*)malloc(whole*sizeof(int));
    if (h_A0 == 0) {
        fprintf (stderr, "!!!! host memory allocation error (h_A0)\n");
        return EXIT_FAILURE;
    }
    h_B0 = (unsigned char*)malloc(whole*sizeof(unsigned char));
    if (h_B0 == 0) {
        fprintf (stderr, "!!!! host memory allocation error (h_B0)\n");
        return EXIT_FAILURE;
    }
    h_C0 = (int*)malloc(half*sizeof(int));
    if (h_C0 == 0) {
        fprintf (stderr, "!!!! host memory allocation error (h_C0)\n");
        return EXIT_FAILURE;
    }
    //HOST VARIABLES USED FOR DEVICE 1.
    h_A1 = (int*)malloc(whole*sizeof(int));
    if (h_A1 == 0) {
        fprintf (stderr, "!!!! host memory allocation error (h_A1)\n");
        return EXIT_FAILURE;
    }
    h_B1 = (unsigned char*)malloc(whole*sizeof(unsigned char));
    if (h_B1 == 0) {
        fprintf (stderr, "!!!! host memory allocation error (h_B1)\n");
        return EXIT_FAILURE;
    }
    h_C1 = (int*)malloc(half*sizeof(int));
if (h_C1 == 0)
{
    fprintf(stderr, "!!! host memory allocation error (h_C1)\n" );
    return EXIT_FAILURE;
}

// Create sequence for h_A of integers from 0 to rows*cols
for (i=0; i < whole; i++)
{
    h_A[i] = i;
}

// Creates array where half the values are non-zero
for (i=0; i < half; i++)
{
    h_B[i] = 1;
}
for (i=half; i < whole; i++)
{
    h_B[i] = 0;
}

// Copy values to variables used for GPU calculations
for (i=0; i < whole; i++)
{
    h_A0[i] = h_A[i];
    h_B0[i] = h_B[i];
    h_A1[i] = h_A[i];
    h_B1[i] = h_B[i];
}

//-----------------------------------------------------------------------------
// Start Time for CPU Calculations.
Start = clock();

// CPU Calculations

// Finds the location index values where the elements are non-zero.
for (i=0; i < whole; i++)
{
    if (h_B[i] != 0)
    {
        h_C[i] = h_A[i];
    }
}

// End Time for CPU Calculations.
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));

//-----------------------------------------------------------------------------
// Start Time for GPU Quadro FX 5800 Calculations.
cudaSetDevice(0);
cudaDeviceSynchronize();
Start = clock();

// Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A0, whole*sizeof(int)) );
CudaErrCheck( cudaMalloc((void **)&d_B0, whole*sizeof(unsigned char)) );
CudaErrCheck( cudaMalloc((void **)&d_C0, half*sizeof(int)) );

// Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, whole*sizeof(int), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B0, h_B0, whole*sizeof(unsigned char), cudaMemcpyHostToDevice) );

// Finds all elements in B that hold true for the Thrust predicate and removes those values. The elements in A that correspond to what is left in B is copied over to C (thrust::remove_copy_if)

thrust::device_ptr<int> dev_ptr_A0(d_A0);
thrust::device_ptr<
unsigned char> dev_ptr_B0(d_B0);
thrust::device_ptr<int> dev_ptr_C0(d_C0);
thrust::remove_copy_if(dev_ptr_A0, dev_ptr_A0 + whole, dev_ptr_B0, dev_ptr_C0,
equal_to_zero());
//
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_C0, d_C0, half*sizeof(int), cudaMemcpyDeviceToHost) );
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%3.3f ", fabs(Time));
//Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();
//
int* d_A1;
unsigned char* d_B1;
int* d_C1;
//
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A1, whole*sizeof(int)) );
CudaErrCheck( cudaMalloc((void **)&d_B1, whole*sizeof(unsigned char)) );
CudaErrCheck( cudaMalloc((void **)&d_C1, half*sizeof(int)) );
//
//Transfer Data from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A1, h_A1, whole*sizeof(int), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B1, h_B1, whole*sizeof(unsigned char), cudaMemcpyHostToDevice) );
//
//Finds all elements in B that hold true for the Thrust predicate and removes
//those values. The elements in A that correspond to what is left in B is copied
//over to C (thrust::removecopyif)
//
thrust::device_ptr<int> dev_ptr_A1(d_A1);
thrust::device_ptr<int> dev_ptr_B1(d_B1);
thrust::device_ptr<int> dev_ptr_C1(d_C1);
thrust::remove_copy_if(dev_ptr_A1, dev_ptr_A1 + whole, dev_ptr_B1, dev_ptr_C1,
equal_to_zero());
//
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_C1, d_C1, half*sizeof(int), cudaMemcpyDeviceToHost) );
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%3.3f ", fabs(Time));
//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < half; i++)
{
    if ((h_C[i] - h_C0[i]) != 0)
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%3.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\n", Percent_Error);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < half; i++)
{
    if ((h_C[i] - h_C1[i]) != 0)
    {
Number_of_Errors++;
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f\n", Percent_Error);
printf("Number of Calculation Errors for Tesla C1060: %i
Number of Calculation Errors for Tesla C1060: %i

A.3.4 Thrust remove_copy_if raw data

500 0.000 0.000 0.000 0.000 0.000 0.000
1000 0.000 0.010 0.010 0.000 0.000 0.000
1500 0.010 0.010 0.000 0.000 0.000 0.000
2000 0.020 0.010 0.010 0.000 0.000 0.000
2500 0.020 0.020 0.010 0.000 0.000 0.000
3000 0.030 0.030 0.010 0.000 0.000 0.000
3500 0.040 0.030 0.030 0.000 0.000 0.000
4000 0.060 0.030 0.040 0.000 0.000 0.000
4500 0.080 0.040 0.040 0.000 0.000 0.000
5000 0.090 0.050 0.050 0.000 0.000 0.000
5500 0.110 0.070 0.060 0.000 0.000 0.000
6000 0.130 0.080 0.080 0.000 0.000 0.000
6500 0.150 0.100 0.080 0.000 0.000 0.000
7000 0.170 0.110 0.090 0.000 0.000 0.000
7500 0.200 0.120 0.100 0.000 0.000 0.000
8000 0.230 0.130 0.120 0.000 0.000 0.000
8500 0.260 0.150 0.130 0.000 0.000 0.000
9000 0.300 0.160 0.140 0.000 0.000 0.000
9500 0.320 0.180 0.150 0.000 0.000 0.000
10000 0.350 0.200 0.170 0.000 0.000 0.000
10500 0.390 0.220 0.200 0.000 0.000 0.000
11000 0.440 0.240 0.220 0.000 0.000 0.000
11500 0.470 0.260 0.240 0.000 0.000 0.000
12000 0.520 0.290 0.270 0.000 0.000 0.000
12500 0.560 0.310 0.300 0.000 0.000 0.000
13000 0.610 0.330 0.330 0.000 0.000 0.000
13500 0.660 0.370 0.340 0.000 0.000 0.000
14000 0.700 0.390 0.370 0.000 0.000 0.000
A.3.5 Thrust sequence test program

```cpp
#include <thrust/device_ptr.h>
#include <thrust/sequence.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

FILE *TimeFile = fopen("ThrustSequence_Times.txt", "w");

void signalHandler( int signum )
{
  printf("Interrupt signal (%i) received.\n", signum);
  fclose(TimeFile);
  exit(signum);
}

int main(int argc, char** argv)
{
  int rows = 500;
  int cols = 500;
  int step = 500;

  signal(SIGINT, signalHandler);

  int i;
```
clock_t Start, End;
double Time;
int Number_of_Errors;
float Percent_Error;
while(1)
{
  srand(time(0));
  printf("Rows: %i\n", rows);
  printf("Columns: %i\n", cols);
  fprintf(TimeFile, "%i ", rows);
  //-------------------------------------------------------------------------------
  int* h_A;
  int* h_A0;
  int* h_A1;
  //
  //HOST ONLY VARIABLES.
  h_A = (int*)malloc(rows*cols*sizeof(int));
  if (h_A == 0)
  {
    fprintf (stderr, "!!!! host memory allocation error (h_A)\n");
    return EXIT_FAILURE;
  }
  //HOST VARIABLES USED FOR DEVICE 0.
  h_A0 = (int*)malloc(rows*cols*sizeof(int));
  if (h_A0 == 0)
  {
    fprintf (stderr, "!!!! host memory allocation error (h_A0)\n");
    return EXIT_FAILURE;
  }
  //HOST VARIABLES USED FOR DEVICE 1.
  h_A1 = (int*)malloc(rows*cols*sizeof(int));
  if (h_A1 == 0)
  {
    fprintf (stderr, "!!!! host memory allocation error (h_A1)\n");
    return EXIT_FAILURE;
  }
  //-------------------------------------------------------------------------------
  //Start Time for CPU Calculations.
  Start = clock();
  //
  //CPU Calculations
  //
  //Create sequence from 0 to rows*cols-1
  //
  for (i=0; i < (rows*cols); i++)
  {
    h_A[i] = i;
  }
  //
  //End Time for CPU Calculations.
  End = clock();
  Time = (End-Start)*1.0/CLOCKS_PER_SEC;
  fprintf(TimeFile, ";%.3f ", fabs(Time));
  //-------------------------------------------------------------------------------
  //Start Time for GPU Quadro FX 5800 Calculations.
  cudaSetDevice(0);
  cudaDeviceSynchronize();
  Start = clock();
  //
  int* d_A0;
  //
  //Allocate Device Memory (GPU).
  CudaErrCheck( cudaMalloc((void**)&d_A0, rows*cols*sizeof(int)) );
  //
  //Create sequence from 0 to rows*cols-1. (thrust::sequence)
  //
  thrust::device_ptr<int> dev_ptr_A0(d_A0);
  thrust::sequence(dev_ptr_A0, dev_ptr_A0 + (rows*cols));
  //
  //Transfer Data from DEVICE to HOST.
  CudaErrCheck( cudaMemcpy(h_A0, d_A0, rows*cols*sizeof(int), cudaMemcpyDeviceToHost) );
End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------//
//Start Time for GPU Tesla C1060 Calculations.
cudaSetDevice(1);
cudaDeviceSynchronize();
Start = clock();
//
int* d_A1;
//
//Allocate Device Memory (GPU).
CudaErrCheck( cudaMalloc((void **)&d_A1, rows*cols*sizeof(int)) );
//
//Create sequence from 0 to rows*cols-1. (thrust::sequence)
//
thrust::device_ptr<int> dev_ptr_A1(d_A1);
thrust::sequence(dev_ptr_A1, dev_ptr_A1 + (rows*cols));
//
//Transfer Data from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_A1, d_A1, rows*cols*sizeof(int), cudaMemcpyDeviceToHost) );
//
//End Time for GPU Calculations.
cudaDeviceSynchronize();
End = clock();
Time = (End-Start)*1.0/CLOCKS_PER_SEC;
fprintf(TimeFile, "%.3f ", fabs(Time));
//-----------------------------------------------------------------------------//
//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_A[i] - h_A0[i]) != 0)
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f ", Percent_Error);

Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_A[i] - h_A1[i]) != 0)
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f\n", Percent_Error);

HOST MEMORY CLEAN UP.
free(h_A);
free(h_A0);
free(h_A1);
//DEVICE 0 MEMORY CLEAN UP (QUADRO FX 5800)
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
//DEVICE 1 MEMORY CLEAN UP (TESLA C1060)
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
//-----------------------------------------------------------------------------//
//Increasing of Matrix Dimensions.

```plaintext
A.3.6  Thrust sequence raw data

<table>
<thead>
<tr>
<th>Time</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1500</td>
<td>0.010</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2000</td>
<td>0.010</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2500</td>
<td>0.020</td>
<td>0.010</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3000</td>
<td>0.020</td>
<td>0.020</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3500</td>
<td>0.040</td>
<td>0.020</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>4000</td>
<td>0.050</td>
<td>0.040</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>4500</td>
<td>0.060</td>
<td>0.040</td>
<td>0.030</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>5000</td>
<td>0.070</td>
<td>0.040</td>
<td>0.040</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>5500</td>
<td>0.090</td>
<td>0.060</td>
<td>0.040</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>6000</td>
<td>0.110</td>
<td>0.060</td>
<td>0.050</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>6500</td>
<td>0.140</td>
<td>0.060</td>
<td>0.060</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>7000</td>
<td>0.150</td>
<td>0.080</td>
<td>0.060</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>7500</td>
<td>0.190</td>
<td>0.080</td>
<td>0.080</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>8000</td>
<td>0.200</td>
<td>0.090</td>
<td>0.080</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>8500</td>
<td>0.230</td>
<td>0.120</td>
<td>0.090</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>9000</td>
<td>0.250</td>
<td>0.130</td>
<td>0.110</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>9500</td>
<td>0.290</td>
<td>0.130</td>
<td>0.130</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10000</td>
<td>0.320</td>
<td>0.160</td>
<td>0.130</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10500</td>
<td>0.350</td>
<td>0.160</td>
<td>0.150</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>11000</td>
<td>0.390</td>
<td>0.180</td>
<td>0.170</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>11500</td>
<td>0.420</td>
<td>0.190</td>
<td>0.180</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>12000</td>
<td>0.460</td>
<td>0.210</td>
<td>0.190</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>12500</td>
<td>0.500</td>
<td>0.230</td>
<td>0.210</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>13000</td>
<td>0.530</td>
<td>0.250</td>
<td>0.240</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>13500</td>
<td>0.570</td>
<td>0.280</td>
<td>0.240</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>14000</td>
<td>0.620</td>
<td>0.290</td>
<td>0.260</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>14500</td>
<td>0.660</td>
<td>0.320</td>
<td>0.290</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>15000</td>
<td>0.720</td>
<td>0.340</td>
<td>0.300</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>15500</td>
<td>0.770</td>
<td>0.360</td>
<td>0.320</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>16000</td>
<td>0.810</td>
<td>0.370</td>
<td>0.350</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>16500</td>
<td>0.870</td>
<td>0.400</td>
<td>0.370</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>17000</td>
<td>0.910</td>
<td>0.430</td>
<td>0.390</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>17500</td>
<td>0.970</td>
<td>0.460</td>
<td>0.410</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>18000</td>
<td>1.030</td>
<td>0.480</td>
<td>0.440</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>18500</td>
<td>1.090</td>
<td>0.500</td>
<td>0.470</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>19000</td>
<td>1.150</td>
<td>0.530</td>
<td>0.490</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>19500</td>
<td>1.210</td>
<td>0.560</td>
<td>0.520</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>20000</td>
<td>1.260</td>
<td>0.600</td>
<td>0.530</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>20500</td>
<td>1.330</td>
<td>0.620</td>
<td>0.570</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>21000</td>
<td>1.390</td>
<td>0.670</td>
<td>0.630</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>21500</td>
<td>1.470</td>
<td>0.680</td>
<td>0.710</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>22000</td>
<td>1.540</td>
<td>0.710</td>
<td>0.810</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>22500</td>
<td>1.600</td>
<td>0.750</td>
<td>0.880</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>23000</td>
<td>1.670</td>
<td>0.790</td>
<td>0.980</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>23500</td>
<td>1.760</td>
<td>0.820</td>
<td>1.050</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>24000</td>
<td>1.830</td>
<td>0.850</td>
<td>1.150</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>24500</td>
<td>1.900</td>
<td>0.890</td>
<td>1.240</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>25000</td>
<td>1.990</td>
<td>0.920</td>
<td>1.340</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>25500</td>
<td>2.060</td>
<td>0.990</td>
<td>1.410</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>26000</td>
<td>2.150</td>
<td>1.040</td>
<td>1.480</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>26500</td>
<td>2.220</td>
<td>1.140</td>
<td>1.530</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>27000</td>
<td>2.330</td>
<td>1.210</td>
<td>1.590</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>27500</td>
<td>2.390</td>
<td>1.290</td>
<td>1.650</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>28000</td>
<td>2.490</td>
<td>1.370</td>
<td>1.720</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>28500</td>
<td>2.580</td>
<td>1.450</td>
<td>1.780</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>29000</td>
<td>2.720</td>
<td>1.800</td>
<td>1.320</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th>Value</th>
<th>2.800</th>
<th>1.960</th>
<th>1.310</th>
<th>0.000</th>
<th>0.000</th>
</tr>
</thead>
<tbody>
<tr>
<td>30000</td>
<td>2.910</td>
<td>2.020</td>
<td>1.540</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>30500</td>
<td>3.000</td>
<td>2.100</td>
<td>1.960</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>31000</td>
<td>3.050</td>
<td>1.840</td>
<td>2.500</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>31500</td>
<td>3.210</td>
<td>2.200</td>
<td>2.050</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>32000</td>
<td>3.250</td>
<td>1.960</td>
<td>2.700</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>
B. ASYNCHRONOUS TEST PROGRAMS AND RAW DATA

Results for the synchronous and asynchronous test programs were analyzed in Section 5.2. The columns in the single GPU raw data results are represented by the titles listed in Table B.1, whereas the columns in the multiple GPU raw data results are shown in Table B.2.

Table B.1
Column representation for single GPU raw data results

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Rows and Columns</td>
<td>Tesla Time in milliseconds</td>
<td>Tesla Percent Error</td>
</tr>
</tbody>
</table>

Table B.2
Column representation for multiple GPU raw data results

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Rows and Columns</td>
<td>Overall GPU Time in milliseconds</td>
<td>Quadro Percent Error</td>
<td>Tesla Percent Error</td>
</tr>
</tbody>
</table>

B.1 Synchronous Single GPU Test Program

//-----------------------------------------------------------------------------------------//
//SingleGPU_Sync_TimeTest.cu
//Nathan Clem
//
//Compile and Execute (Make sure in correct directory):
// nvcc -I/usr/local/cuda-5.5/include -L/usr/local/cuda-5.5/lib64 SingleGPU_Sync_TimeTest.cu
//-lnppc -lnppi -lnpps -lm
// ./a.out
//-----------------------------------------------------------------------------------------//
#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

//File to store CPU and GPU run times.
FILE *TimeFile = fopen("SingleGPU_Sync_Times.txt", "w");
//Use to stop program in case of infinite loop
//Enter: "CTRL + C"
void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

//Cuda Error Check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
    if (code != cudaSuccess)
    {
        fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}

int main(int argc, char **argv)
{
    cudaSetDevice(0);
    int rows = 500;
    int cols = 500;
    int step = 500;
    int PointBound = 100;
    signal(SIGINT, signalHandler);
    int i;
    int nFunctions = 5;
    int Number_of_Errors;
    float Percent_Error;
    while(1)
    {
        srand(time(0));
        printf("Rows: %i\n", rows);
        printf("Columns: %i\n", cols);
        fprintf(TimeFile, "%i \n", rows);
        float msec;
        cudaEvent_t StartEvent, StopEvent;
        CudaErrCheck( cudaEventCreate(&StartEvent) );
        CudaErrCheck( cudaEventCreate(&StopEvent) );
        float* h_A;
        float* h_B;
        float* h_C;
        float* h_A0;
        float* h_B0;
        float* h_C0;
        h_A = (float*)malloc(rows*cols*sizeof(float));
        if (h_A == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_A)\n");
            return EXIT_FAILURE;
        }
        h_B = (float*)malloc(rows*cols*sizeof(float));
        if (h_B == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_B)\n");
            return EXIT_FAILURE;
        }
        h_C = (float*)malloc(rows*cols*sizeof(float));
        if (h_C == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_C)\n");
            return EXIT_FAILURE;
        }
        h_A0 = (float*)malloc(rows*cols*sizeof(float));
        if (h_A0 == 0)
        {
            CudaErrCheck (cudaMemcpy(h_A0, h_A, rows*cols*sizeof(float), cudaMemcpyHostToDevice));
            CudaErrCheck (cudaEventRecord(StopEvent, 0));
            CudaErrCheck (cudaEventSynchronize(StopEvent));
            CudaErrCheck (cudaEventElapsedTime(&msec, StartEvent, StopEvent));
            cout << "A: time = " << (1000 * msec) << " ms\n";
        }
    }
}
fprintf(stderr, "!!!! host memory allocation error (h_A0)\n");
return EXIT_FAILURE;
}
h_B0 = (float*)malloc(rows*cols*sizeof(float));
if (h_B0 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_B0)\n");
    return EXIT_FAILURE;
}
h_C0 = (float*)malloc(rows*cols*sizeof(float));
if (h_C0 == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_C0)\n");
    return EXIT_FAILURE;
}

//Creation of random values.
for (i=0; i < (rows*cols); i++)
{
    h_A[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_B[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
}
for (i=0; i < (rows*cols); i++)
{
    h_A0[i] = h_A[i];
    h_B0[i] = h_B[i];
}memset(h_C0, 0, rows*cols*sizeof(float));
//-----------------------------------------------------------------------------//
//CPU Calculations
//h_C = h_A + h_B
for (i=0; i < (rows*cols); i++)
{
    h_C[i] = h_A[i] + h_B[i];
}
//-----------------------------------------------------------------------------//
//Start Time for Single GPU Sync Implementation.
CudaErrCheck( cudaEventRecord(StartEvent, 0) );
float* d_A0;
float* d_B0;
float* d_C0;
//Allocation of DEVICE memory.
CudaErrCheck( cudaMalloc((void **)&d_A0, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_B0, rows*cols*sizeof(float)) );
CudaErrCheck( cudaMalloc((void **)&d_C0, rows*cols*sizeof(float)) );
//Transferring from HOST to DEVICE.
CudaErrCheck( cudaMemcpy(d_A0, h_A0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_B0, h_B0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
CudaErrCheck( cudaMemcpy(d_C0, h_C0, rows*cols*sizeof(float), cudaMemcpyHostToDevice) );
//Kernel Computations
for (i=0; i < nFunctions; i++)
{
    nppsAdd_32f(d_A0, d_B0, d_C0, rows*cols);
}
//Transferring from DEVICE to HOST.
CudaErrCheck( cudaMemcpy(h_A0, d_A0, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
CudaErrCheck( cudaMemcpy(h_B0, d_B0, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
CudaErrCheck( cudaMemcpy(h_C0, d_C0, rows*cols*sizeof(float), cudaMemcpyDeviceToHost) );
//Freeing DEVICE Memory.
CudaErrCheck( cudaFree(d_A0) );
CudaErrCheck( cudaFree(d_B0) );
CudaErrCheck( cudaFree(d_C0) );
//End Time for GPU Calculations.
CudaErrCheck( cudaEventRecord(StopEvent, 0) );
CudaErrCheck( cudaEventSynchronize(StopEvent) );
CudaErrCheck( cudaEventElapsedTime(&msec, StartEvent, StopEvent) );
fprintf(TimeFile, ".%.3f ", msec);
//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
_percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
    if ((h_C[i] - h_C0[i]) != 0)
    {
        Number_of_Errors++;
    }
}
_percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f\n", _percent_Error);
printf("Number of Calculation Errors for GPU: %i\n", Number_of_Errors);
printf("GPU Error Percentage: %.3f\n", _percent_Error);

//HOST MEMORY CLEAN UP.
free(h_A);
free(h_B);
free(h_C);
free(h_A0);
free(h_B0);
free(h_C0);
//Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;

//END PROGRAM
return EXIT_SUCCESS;

B.2  Synchronous Single GPU Raw Data

500  68.495  0.000
1000 8.299  0.000
1500 15.859  0.000
2000 25.175  0.000
2500 37.942  0.000
3000 53.541  0.000
3500 72.308  0.000
4000 93.370  0.000
4500 117.927  0.000
5000 144.789  0.000
5500 174.643  0.000
6000 207.596  0.000
6500 243.568  0.000
7000 281.986  0.000
7500 322.839  0.000
8000 368.072  0.000
8500 415.183  0.000
9000 466.193  0.000
9500 518.717  0.000
10000 574.315  0.000
10500 638.816  0.000
11000 696.518  0.000
11500 759.644  0.000
12000 829.120  0.000
12500 897.846  0.000
13000 971.183  0.000
13500 1048.114 0.000
14000 1128.632 0.000
14500 1209.236 0.000
15000 1292.185 0.000
15500 1402.848 0.000
16000 1528.383 0.000
16500 1658.529 0.000
17000 1795.779 0.000
17500 1930.516 0.000
18000 2078.058 0.000
18500 2222.291 0.000
B.3  Asynchronous Single GPU Depth First Test Program

#include <npps.h>
#include <nppi.h>
#include <nppcore.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

FILE *TimeFile = fopen("SingleGPU_Async_Depth_Times.txt", "w");
//Use to stop program in case of infinite loop
//Enter: "CTRL + C"
void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
    if (code != cudaSuccess)
    {
        fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}

int main(int argc, char **argv)
{
    cudaSetDevice(0);
    int rows = 500;
    int cols = 500;
    int step = 500;
    int PointBound = 100;
    signal(SIGINT, signalHandler);
    int i, j;
    int nFunctions = 5;
    int Number_of_Errors;
    float Percent_Error;
    int nElements;
    int nStreams;
    int StreamSize;
    int nBytesF;
    int nStreamBytesF;
    int offset;
    while(1)
    {
        srand(time(0));
        printf("Rows: %i\n", rows);
        printf("Columns: %i\n", cols);
        fprintf(TimeFile, "%i \n", rows);
        float msec;
        cudaEvent_t StartEvent, StopEvent;
        CudaErrCheck( cudaEventCreate(&StartEvent) );
        CudaErrCheck( cudaEventCreate(&StopEvent) );
        nElements = rows*cols;
        nStreams = 4;
        StreamSize = nElements/nStreams;
        nBytesF = nElements*sizeof(float);
        nStreamBytesF = StreamSize*nBytesF;
        nBytesF = nElements*sizeof(float);
nStreamBytesF = StreamSize*sizeof(float);
float* h_A;
float* h_B;
float* h_C;
float* h_A0;
float* h_B0;
float* h_C0;
h_A = (float*)malloc(nBytesF);
if (h_A == 0)
{  
    fprintf (stderr, "!!!! host memory allocation error (h_A)\n");
    return EXIT_FAILURE;
}

h_B = (float*)malloc(nBytesF);
if (h_B == 0)
{  
    fprintf (stderr, "!!!! host memory allocation error (h_B)\n");
    return EXIT_FAILURE;
}

h_C = (float*)malloc(nBytesF);
if (h_C == 0)
{  
    fprintf (stderr, "!!!! host memory allocation error (h_C)\n");
    return EXIT_FAILURE;
}

CudaErrCheck( cudaMallocHost((void**)&h_A0, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_B0, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_C0, nBytesF) );

//Creation of random values.
for (i=0; i < (rows*cols); i++)
{  
    h_A[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_B[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
}

for (i=0; i < (rows*cols); i++)
{  
    h_A0[i] = h_A[i];
    h_B0[i] = h_B[i];
}

memset(h_C0, 0, nBytesF);

//CPU Calculations
h_C = h_A + h_B
for (i=0; i < (rows*cols); i++)
{  
    h_C[i] = h_A[i] + h_B[i];
}

//Start Time for Single GPU Aync Depth Implementation.
CudaErrCheck( cudaEventRecord(StartEvent,0) );
CudaErrCheck( cudaStreamCreate(&s[i]) );

float* d_A0;
float* d_B0;
float* d_C0;

//Allocation of DEVICE memory.
CudaErrCheck( cudaMalloc((void**)&d_A0, nBytesF) );
CudaErrCheck( cudaMalloc((void**)&d_B0, nBytesF) );
CudaErrCheck( cudaMalloc((void**)&d_C0, nBytesF) );

//Cuda Depth First Streams
for (i = 0; i < nStreams; i++)
{  
    offset = i * StreamSize;
    CudaErrCheck( cudaMemcpyAsync(&d_A0[offset], &h_A0[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]) );
    CudaErrCheck( cudaMemcpyAsync(&d_B0[offset], &h_B0[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]) );
    CudaErrCheck( cudaMemcpyAsync(&d_C0[offset], &h_C0[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]) );
}
B.4 Asynchronous Single GPU Depth First Raw Data

<table>
<thead>
<tr>
<th>Value</th>
<th>CPU Time</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>67.571</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>6.629</td>
<td>0.000</td>
</tr>
<tr>
<td>1500</td>
<td>13.327</td>
<td>0.000</td>
</tr>
<tr>
<td>2000</td>
<td>23.045</td>
<td>0.000</td>
</tr>
<tr>
<td>2500</td>
<td>34.953</td>
<td>0.000</td>
</tr>
<tr>
<td>3000</td>
<td>49.536</td>
<td>0.000</td>
</tr>
<tr>
<td>3500</td>
<td>66.461</td>
<td>0.000</td>
</tr>
<tr>
<td>4000</td>
<td>86.134</td>
<td>0.000</td>
</tr>
</tbody>
</table>
Asynchronous Single GPU Breadth First Test Program

//SingleGPU_Async_Breadth_TimeTest.cu
//Nathan Clem

//Compile and Execute (Make sure in correct directory):
// nvcc -I/usr/local/cuda-5.5/include -L/usr/local/cuda-5.5/lib64
//SingleGPU_Async_Breadth_TimeTest.cu -lnppc -lnppl -lnpps -lm
//./a.out

#include <npps.h>
#include <nppi.h>
#include <nppcore.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

//File to store CPU and GPU run times.
FILE *TimeFile = fopen("SingleGPU_Async_Breadth_Times.txt", "w");

//Use to stop program in case of infinite loop
//Enter: "CTRL + C"
void signalHandler(int signum)
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

//Cuda Error Check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); } inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
    if (code != cudaSuccess) {
        fprintf(stderr,"GPUassert: %s %s %d\n", code, file, line);
        if (abort) exit(code);
    }
}

int main(int argc, char **argv)
{
cudaSetDevice(0);
int rows = 500;
int cols = 500;
int step = 500;
int PointBound = 100;
signal(SIGINT, signalHandler);
int i, j;
int nFunctions = 5;
int Number_of_Errors;
float Percent_Error;
int nElements;
int nStreams;
int StreamSize;
int nBytesF;
int nStreamBytesF;
int offset;
while(1)
{
    srand(time(0));
    printf("Rows: %i\n", rows);
    printf("Columns: %i\n", cols);
    fprintf(TimeFile, "%i ", rows);
    float msec;
    cudaEvent_t StartEvent, StopEvent;
    CudaErrCheck( cudaEventCreate(&StartEvent) );
    CudaErrCheck( cudaEventCreate(&StopEvent) );
    nElements = rows*cols;
    nStreams = 4;
    StreamSize = nElements/nStreams;
    nBytesF = nElements*sizeof(float);
    nStreamBytesF = StreamSize*sizeof(float);
    float* h_A;
    float* h_B;
    float* h_C;
    float* h_A0;
    float* h_B0;
    float* h_C0;
    h_A = (float*)malloc(nBytesF);
    if (h_A == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error (h_A)\n");
        return EXIT_FAILURE;
    }
    h_B = (float*)malloc(nBytesF);
    if (h_B == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error (h_B)\n");
        return EXIT_FAILURE;
    }
    h_C = (float*)malloc(nBytesF);
    if (h_C == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error (h_C)\n");
        return EXIT_FAILURE;
    }
    CudaErrCheck( cudaMallocHost((void**)&h_A0, nBytesF) );
    CudaErrCheck( cudaMallocHost((void**)&h_B0, nBytesF) );
    CudaErrCheck( cudaMallocHost((void**)&h_C0, nBytesF) );
    //Creation of random values.
    for (i=0; i < (rows*cols); i++)
    {
        h_A[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
        h_B[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    }
    for (i=0; i < (rows*cols); i++)
    {
        h_A0[i] = h_A[i];
        h_B0[i] = h_B[i];
    }
    memset(h_C0, 0, nBytesF);
    //CPU Calculations
//h_C = h_A + h_B
for (i=0; i < (rows*cols); i++)
{
    h_C[i] = h_A[i] + h_B[i];
}

CudaErrCheck( cudaEventRecord(StartEvent,0) );

//Creation of Streams.
cudaStream_t s[nStreams];
for(i=0; i < nStreams; i++)
{
    CudaErrCheck( cudaStreamCreate(&s[i]) );
}

float* d_A0;
float* d_B0;
float* d_C0;

//Allocation of DEVICE memory.
CudaErrCheck( cudaMalloc((void **)&d_A0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)&d_B0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)&d_C0, nBytesF) );

//Cuda Breadth First Streams
for (i = 0; i < nStreams; i++)
{
    offset = i * StreamSize;
    CudaErrCheck( cudaMemcpyAsync(&d_A0[offset], &h_A0[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]) );
    CudaErrCheck( cudaMemcpyAsync(&d_B0[offset], &h_B0[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]) );
    CudaErrCheck( cudaMemcpyAsync(&d_C0[offset], &h_C0[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]) );
}

//Freeing DEVICE Memory.
CudaErrCheck( cudaFree(d_A0) );
CudaErrCheck( cudaFree(d_B0) );
CudaErrCheck( cudaFree(d_C0) );

//Destruction of Streams.
for (i = 0; i < nStreams; i++)
{
    CudaErrCheck( cudaStreamDestroy(s[i]) );
}

//End Time for GPU Calculations.
CudaErrCheck( cudaEventRecord(StopEvent, 0) );
CudaErrCheck( cudaEventSynchronize(StopEvent) );
CudaErrCheck( cudaEventElapsedTime(&msec, StartEvent, StopEvent) );
fprintf(TimeFile, "%.3f ", msec);

//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < (rows*cols); i++)
{
if ((h_C[i] - h_C0[i]) != 0)
{
    Number_of_Errors++;
}

Percent_Error = 100.0*Number_of_Errors/(rows*cols);
fprintf(TimeFile, "%.3f\n", Percent_Error);
printf("Number of Calculation Errors for GPU: %i\n", Number_of_Errors);
printf("GPU Error Percentage: %.3f%\n", Percent_Error);
//HOST MEMORY CLEAN UP.
free(h_A);
free(h_B);
free(h_C);
CudaErrCheck( cudaFreeHost(h_A0) );
CudaErrCheck( cudaFreeHost(h_B0) );
CudaErrCheck( cudaFreeHost(h_C0) );
//Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;
}

//END PROGRAM
return EXIT_SUCCESS;

B.6  Asynchronous Single GPU Breadth First Raw Data

500 67.797 0.000
1000 5.610 0.000
1500 11.347 0.000
2000 19.155 0.000
2500 28.802 0.000
3000 40.680 0.000
3500 54.520 0.000
4000 70.391 0.000
4500 88.430 0.000
5000 108.611 0.000
5500 130.452 0.000
6000 154.912 0.000
6500 181.606 0.000
7000 208.915 0.000
7500 240.870 0.000
8000 273.595 0.000
8500 308.456 0.000
9000 344.963 0.000
9500 384.144 0.000
10000 425.170 0.000
10500 468.356 0.000
11000 513.774 0.000
11500 561.075 0.000
12000 610.425 0.000
12500 662.067 0.000
13000 716.047 0.000
13500 771.342 0.000
14000 829.362 0.000
14500 889.472 0.000
15000 951.998 0.000
15500 1015.646 0.000
16000 1083.961 0.000
16500 1150.240 0.000
17000 1224.008 0.000
17500 1293.521 0.000
18000 1370.230 0.000
18500 1444.985 0.000
B.7  Synchronous Multiple GPU Test Program

//Compile and Execute (Make sure in correct directory):
// nvcc -I/usr/local/cuda-5.5/include -L/usr/local/cuda-5.5/lib64 MultiGPU_Sync_TimeTest.cu -
// lnnpc -lnppi -lnpps -lm
// ./a.out

#include <npps.h>
#include <nppi.h>
#include <cstdlib>
#include <ctime>

FILE *TimeFile = fopen("MultiGPU_Sync_Times.txt", "w");

int main(int argc, char **argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    int PointBound = 100;
    signal(SIGINT, signalHandler);
    int i;
    int nFunctions = 5;
    int Number_of_Errors;
    float Percent_Error;
    int nElements;
    int nElementsDevice;
    int nBytesF;
    while(1)
    {
        srand(time(0));
        printf("Rows: %i\n", rows);
        printf("Columns: %i\n", cols);
        printf(TimeFile, "%i ", rows);
        float msec;
        cudaEvent_t StartEvent, StopEvent;
        cudaSetDevice(0);
        CudaErrCheck( cudaEventCreate(&StartEvent) );
        CudaErrCheck( cudaEventCreate(&StopEvent) );
        nElements = rows*cols;
        nElementsDevice = nElements/2;
        nBytesF = nElementsDevice*sizeof(float);
        float* h_A0Host;
        float* h_B0Host;
        float* h_C0Host;
        float* h_A1Host;
        float* h_B1Host;
        float* h_C1Host;
float* h_A0;
float* h_B0;
float* h_C0;
float* h_A1;
float* h_B1;
float* h_C1;

h_A0Host = (float*)malloc(nBytesF);
if (h_A0Host == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_A0Host)\n");
    return EXIT_FAILURE;
}

h_B0Host = (float*)malloc(nBytesF);
if (h_B0Host == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_B0Host)\n");
    return EXIT_FAILURE;
}

h_C0Host = (float*)malloc(nBytesF);
if (h_C0Host == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_C0Host)\n");
    return EXIT_FAILURE;
}

h_A1Host = (float*)malloc(nBytesF);
if (h_A1Host == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_A1Host)\n");
    return EXIT_FAILURE;
}

h_B1Host = (float*)malloc(nBytesF);
if (h_B1Host == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_B1Host)\n");
    return EXIT_FAILURE;
}

h_C1Host = (float*)malloc(nBytesF);
if (h_C1Host == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_C1Host)\n");
    return EXIT_FAILURE;
}

//HOST VARIABLES USED FOR DEVICE 0.
h_A0 = (float*)malloc(nBytesF);
if (h_A0 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_A0)\n");
    return EXIT_FAILURE;
}

h_B0 = (float*)malloc(nBytesF);
if (h_B0 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_B0)\n");
    return EXIT_FAILURE;
}

h_C0 = (float*)malloc(nBytesF);
if (h_C0 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_C0)\n");
    return EXIT_FAILURE;
}

//HOST VARIABLES USED FOR DEVICE 1.
h_A1 = (float*)malloc(nBytesF);
if (h_A1 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_A1)\n");
    return EXIT_FAILURE;
}

h_B1 = (float*)malloc(nBytesF);
if (h_B1 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_B1)\n");
}
return EXIT_FAILURE;
}
h_C1 = (float*)malloc(nBytesF);
if (h_C1 == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_C1)\n");
    return EXIT_FAILURE;
}
//Creation of random values.
for (i=0; i < nElementsDevice; i++)
{
    h_A0Host[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_B0Host[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_A1Host[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_B1Host[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
}
for (i=0; i < nElementsDevice; i++)
{
    h_A0[i] = h_A0Host[i];
    h_B0[i] = h_B0Host[i];
    h_A1[i] = h_A1Host[i];
    h_B1[i] = h_B1Host[i];
}
memset(h_C0, 0, nBytesF);
memset(h_C1, 0, nBytesF);
//-----------------------------------------------------------------------------//
//CPU Calculations
//h_C = h_A + h_B
for (i=0; i < nElementsDevice; i++)
{
    h_C0Host[i] = h_A0Host[i] + h_B0Host[i];
    h_C1Host[i] = h_A1Host[i] + h_B1Host[i];
}
//-----------------------------------------------------------------------------//
//Start Time For Multi GPU Sync Implementation.
CudaErrCheck( cudaEventRecord(StartEvent,0) );
float* d_A0;
float* d_A1;
float* d_B0;
float* d_B1;
float* d_C0;
float* d_C1;
//Allocation of DEVICE memory.
CudaErrCheck( cudaMalloc((void **)&d_A0, nBytesF) );
cudaSetDevice(1);
CudaErrCheck( cudaMalloc((void **)&d_A1, nBytesF) );
cudaSetDevice(0);
CudaErrCheck( cudaMalloc((void **)&d_B0, nBytesF) );
cudaSetDevice(1);
CudaErrCheck( cudaMalloc((void **)&d_B1, nBytesF) );
cudaSetDevice(0);
CudaErrCheck( cudaMalloc((void **)&d_C0, nBytesF) );
cudaSetDevice(1);
CudaErrCheck( cudaMalloc((void **)&d_C1, nBytesF) );
//Transferring from HOST to DEVICE.
cudaSetDevice(0);
CudaErrCheck( cudaMemcpy(d_A0, h_A0, nBytesF, cudaMemcpyHostToDevice) );
cudaSetDevice(1);
CudaErrCheck( cudaMemcpy(d_A1, h_A1, nBytesF, cudaMemcpyHostToDevice) );
cudaSetDevice(0);
CudaErrCheck( cudaMemcpy(d_B0, h_B0, nBytesF, cudaMemcpyHostToDevice) );
cudaSetDevice(1);
CudaErrCheck( cudaMemcpy(d_B1, h_B1, nBytesF, cudaMemcpyHostToDevice) );
cudaSetDevice(0);
CudaErrCheck( cudaMemcpy(d_C0, h_C0, nBytesF, cudaMemcpyHostToDevice) );
cudaSetDevice(1);
CudaErrCheck( cudaMemcpy(d_C1, h_C1, nBytesF, cudaMemcpyHostToDevice) );
//Kernel Computations
for (i=0; i < nFunctions; i++)
{
    cudaSetDevice(0);
    nppsAdd_32f(d_A0, d_B0, d_C0, nElementsDevice);
cudaSetDevice(1);
nppsAdd_32f(d_A1, d_B1, d_C1, nElementsDevice);
}
//Transferring from DEVICE to HOST.
cudaSetDevice(0);
CudaErrCheck(cudaMemcpy(h_A0, d_A0, nBytesF, cudaMemcpyDeviceToHost));
cudaSetDevice(1);
CudaErrCheck(cudaMemcpy(h_A1, d_A1, nBytesF, cudaMemcpyDeviceToHost));
cudaSetDevice(0);
CudaErrCheck(cudaMemcpy(h_B0, d_B0, nBytesF, cudaMemcpyDeviceToHost));
cudaSetDevice(1);
CudaErrCheck(cudaMemcpy(h_B1, d_B1, nBytesF, cudaMemcpyDeviceToHost));
cudaSetDevice(0);
CudaErrCheck(cudaMemcpy(h_C0, d_C0, nBytesF, cudaMemcpyDeviceToHost));
cudaSetDevice(1);
CudaErrCheck(cudaMemcpy(h_C1, d_C1, nBytesF, cudaMemcpyDeviceToHost));
//Freeing DEVICE Memory.
cudaSetDevice(0);
CudaErrCheck(cudaFree(d_A0));
cudaSetDevice(1);
CudaErrCheck(cudaFree(d_A1));
cudaSetDevice(0);
CudaErrCheck(cudaFree(d_B0));
cudaSetDevice(1);
CudaErrCheck(cudaFree(d_B1));
cudaSetDevice(0);
CudaErrCheck(cudaFree(d_C0));
cudaSetDevice(1);
CudaErrCheck(cudaFree(d_C1));
//End Time for GPU Calculations.
cudaSetDevice(0);
CudaErrCheck(cudaEventRecord(StopEvent, 0));
CudaErrCheck(cudaEventSynchronize(StopEvent));
fprintf(TimeFile, "%.3f ", msec);
//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < nElementsDevice; i++)
{
    if (((h_C0Host[i] - h_C0[i]) != 0)
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols/2);
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\n", Percent_Error);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < nElementsDevice; i++)
{
    if (((h_C1Host[i] - h_C1[i]) != 0)
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols/2);
fprintf(TimeFile, "%.3f\n", Percent_Error);
printf("Number of Calculation Errors for Tesla C1060: %\n", Number_of_Errors);
printf("Tesla C1060 Error Percentage: %.3f\n\n", Percent_Error);
//HOST MEMORY CLEAN UP.
free(h_A0Host);
free(h_B0Host);
free(h_C0Host);
free(h_A1Host);
free(h_B1Host);
free(h_C1Host);
free(h_A0);
free(h_B0);
free(h_C0);
cudaSetDevice(1);
free(h_A1);
free(h_B1);
free(h_C1);
//Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;

} //END PROGRAM
return EXIT_SUCCESS;

B.8  Synchronous Multiple GPU Raw Data

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>166.031</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>8.867</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1500</td>
<td>15.833</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2000</td>
<td>25.483</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2500</td>
<td>37.400</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3000</td>
<td>52.464</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3500</td>
<td>70.261</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>4000</td>
<td>90.872</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>4500</td>
<td>114.566</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>5000</td>
<td>140.824</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>5500</td>
<td>169.526</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>6000</td>
<td>201.241</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>6500</td>
<td>235.998</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>7000</td>
<td>273.765</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>7500</td>
<td>313.137</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>8000</td>
<td>355.866</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>8500</td>
<td>401.763</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>9000</td>
<td>450.820</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>9500</td>
<td>502.208</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10000</td>
<td>555.412</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10500</td>
<td>612.103</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>11000</td>
<td>675.231</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>11500</td>
<td>733.999</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>12000</td>
<td>799.142</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>12500</td>
<td>867.558</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>13000</td>
<td>937.922</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>13500</td>
<td>1014.144</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>14000</td>
<td>1087.668</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>14500</td>
<td>1169.484</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>15000</td>
<td>1247.659</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>15500</td>
<td>1334.052</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>16000</td>
<td>1418.327</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>16500</td>
<td>1510.745</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>17000</td>
<td>1603.456</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>17500</td>
<td>1726.523</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>18000</td>
<td>1866.467</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>18500</td>
<td>2000.611</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>19000</td>
<td>2135.926</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>19500</td>
<td>2277.989</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>20000</td>
<td>2427.760</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>20500</td>
<td>2578.075</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>21000</td>
<td>2731.809</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>21500</td>
<td>2890.717</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>22000</td>
<td>3053.824</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>22500</td>
<td>22554.619</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>23000</td>
<td>11210.952</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>23500</td>
<td>65390.887</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>24000</td>
<td>62942.340</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>24500</td>
<td>240594.594</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>25000</td>
<td>195697.250</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>25500</td>
<td>247241.828</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>26000</td>
<td>256912.453</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>
B.9  Asynchronous Multiple GPU Depth First Test Program

//Compile and Execute (Make sure in correct directory):
// nvcc -I/usr/local/cuda-5.5/include -L/usr/local/cuda-5.5/lib64
//MultiGPU_Async_Depth_TimeTest.cu -lnppc -lnppi -lnpps -lm
// ./a.out

#include <npps.h>
#include <nppi.h>
#include <nppcore.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

//File to store CPU and GPU run times.
FILE *TimeFile = fopen("MultiGPU_Async_Depth_Times.txt", "w");
//Use to stop program in case of infinite loop
//Enter: “CTRL + C”
void signalHandler( int signum )
{
  printf("Interrupt signal (%i) received.\n", signum);
  fclose(TimeFile);  
  exit(signum);
}

//Cuda Error Check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
  if (code != cudaSuccess)
  {
    fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
    if (abort) exit(code);
  }
}

int main(int argc, char **argv)
{
  int rows = 500;
  int cols = 500;
  int step = 500;
  int PointBound = 100;
  signal(SIGINT, signalHandler);
  int i, j;
  int nFunctions = 5;
  int Number_of_Errors;
  float Percent_Error;
  int nElements;
  int nElementsDevice;
  int nStreams;
  int StreamSize;
  int nBytesF;
  int nStreamBytesF;
  int offset;
  while(1)
  {
    srand(time(0));
    printf("Rows: %i\n", rows);
    printf("Columns: %i\n", cols);
    fprintf(TimeFile, "%i ", rows);
    float msec;
    cudaEvent_t StartEvent, StopEvent;
    cudaSetDevice(0);
    CudaErrCheck( cudaEventCreate(&StartEvent) );
    CudaErrCheck( cudaEventCreate(&StopEvent) );
    nElements = rows*cols;
    nStreams = 4;
StreamSize = nElementsDevice/nStreams;
nStreamBytesF = StreamSize*sizeof(float);
float* h_A0Host;
float* h_B0Host;
float* h_C0Host;
float* h_A1Host;
float* h_B1Host;
float* h_C1Host;
float* h_A0;
float* h_B0;
float* h_C0;
float* h_A1;
float* h_B1;
float* h_C1;
h_A0Host = (float*)malloc(nBytesF);
if (h_A0Host == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_A0Host)\n");
    return EXIT_FAILURE;
}
h_B0Host = (float*)malloc(nBytesF);
if (h_B0Host == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_B0Host)\n");
    return EXIT_FAILURE;
}
h_C0Host = (float*)malloc(nBytesF);
if (h_C0Host == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_C0Host)\n");
    return EXIT_FAILURE;
}
h_A1Host = (float*)malloc(nBytesF);
if (h_A1Host == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_A1Host)\n");
    return EXIT_FAILURE;
}
h_B1Host = (float*)malloc(nBytesF);
if (h_B1Host == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_B1Host)\n");
    return EXIT_FAILURE;
}
h_C1Host = (float*)malloc(nBytesF);
if (h_C1Host == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_C1Host)\n");
    return EXIT_FAILURE;
}
//HOST VARIABLES USED FOR DEVICE 0.
CudaErrCheck( cudaMallocHost((void**)&h_A0, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_B0, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_C0, nBytesF) );
//HOST VARIABLES USED FOR DEVICE 1.
cudaSetDevice(1);
CudaErrCheck( cudaMallocHost((void**)&h_A1, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_B1, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_C1, nBytesF) );
//Creation of random values.
for (i=0; i < nElementsDevice; i++)
{
    h_A0Host[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_B0Host[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_A1Host[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_B1Host[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
}
for (i=0; i < nElementsDevice; i++)
{
    h_A0[i] = h_A0Host[i];
    h_B0[i] = h_B0Host[i];
    h_A1[i] = h_A1Host[i];
h_B1[i] = h_B1Host[i];
}
memset(h_C0, 0, nBytesF);
memset(h_C1, 0, nBytesF);
//CPU Calculations
//h_C = h_A + h_B
for (i = 0; i < nElementsDevice; i++)
{
    h_C0Host[i] = h_A0Host[i] + h_B0Host[i];
    h_C1Host[i] = h_A1Host[i] + h_B1Host[i];
}
//Start Time for Multi GPU Async Depth Implementation.
cudaSetDevice(0);
CudaErrCheck( cudaEventRecord(StartEvent,0) );
//Creation of Streams.
cudaStream_t s0[nStreams];
cudaStream_t s1[nStreams];
for(i=0; i < nStreams; i++)
{
    cudaStreamCreate(&s0[i]);
}
cudaSetDevice(1);
for(i=0; i < nStreams; i++)
{
    cudaStreamCreate(&s1[i]);
}
float* d_A0;
float* d_A1;
float* d_B0;
float* d_B1;
float* d_C0;
float* d_C1;
//Allocation of DEVICE memory.
cudaSetDevice(0);
CudaErrCheck( cudaMalloc((void **)&d_A0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)&d_B0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)&d_C0, nBytesF) );
cudaSetDevice(1);
CudaErrCheck( cudaMalloc((void **)&d_A1, nBytesF) );
CudaErrCheck( cudaMalloc((void **)&d_B1, nBytesF) );
CudaErrCheck( cudaMalloc((void **)&d_C1, nBytesF) );
//Cuda Depth First Streams
cudaSetDevice(0);
for (i = 0; i < nStreams; i++)
{
    offset = i * StreamSize;
    CudaErrCheck( cudaMemcpyAsync(&d_A0[offset], &h_A0[offset], nStreamBytesF,
cudaMemcpyHostToDevice, s0[i]) );
    CudaErrCheck( cudaMemcpyAsync(&d_B0[offset], &h_B0[offset], nStreamBytesF,
cudaMemcpyHostToDevice, s0[i]) );
    CudaErrCheck( cudaMemcpyAsync(&d_C0[offset], &h_C0[offset], nStreamBytesF,
cudaMemcpyHostToDevice, s0[i]) );
    nppSetStream(s0[i]);
    for (j = 0; j < nFunctions; j++)
    {
        nppsAdd_32f(&d_A0[offset], &d_B0[offset], &d_C0[offset],
        StreamSize);
    }
    CudaErrCheck( cudaMemcpyAsync(&h_A0[offset], &d_A0[offset], nStreamBytesF,
cudaMemcpyDeviceToHost, s0[i]) );
    CudaErrCheck( cudaMemcpyAsync(&h_B0[offset], &d_B0[offset], nStreamBytesF,
cudaMemcpyDeviceToHost, s0[i]) );
    CudaErrCheck( cudaMemcpyAsync(&h_C0[offset], &d_C0[offset], nStreamBytesF,
cudaMemcpyDeviceToHost, s0[i]) );
}
cudaSetDevice(1);
for (i = 0; i < nStreams; i++)
{
    offset = i * StreamSize;
```c
CudaErrCheck( cudaMemcpyAsync(&d_A1[offset], &h_A1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i]));
CudaErrCheck( cudaMemcpyAsync(&d_B1[offset], &h_B1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i]));
CudaErrCheck( cudaMemcpyAsync(&d_C1[offset], &h_C1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i]));
nppSetStream(s1[i]);
for (j=0; j < nFunctions; j++)
{
    nppsAdd_32f(&d_A1[offset], &d_B1[offset], &d_C1[offset], StreamSize);
}
CudaErrCheck( cudaMemcpyAsync(&h_A1[offset], &d_A1[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s1[i]));
CudaErrCheck( cudaMemcpyAsync(&h_B1[offset], &d_B1[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s1[i]));
CudaErrCheck( cudaMemcpyAsync(&h_C1[offset], &d_C1[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s1[i]));

//Freeing DEVICE Memory.
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
CudaErrCheck( cudaFree(d_B0) );
CudaErrCheck( cudaFree(d_C0) );
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
CudaErrCheck( cudaFree(d_B1) );
CudaErrCheck( cudaFree(d_C1) );

//Destruction of Streams.
cudaSetDevice(0);
for (i = 0; i < nStreams; i++)
{
    CudaErrCheck( cudaStreamDestroy(s0[i]) );
}
cudaSetDevice(1);
for (i = 0; i < nStreams; i++)
{
    CudaErrCheck( cudaStreamDestroy(s1[i]) );
}

//End Time for GPU Calculations.
cudaSetDevice(0);
CudaErrCheck( cudaEventRecord(StopEvent, 0) );
CudaErrCheck( cudaEventSynchronize(StopEvent) );
fprintf(TimeFile, "%.3f ", msec);

//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < nElementsDevice; i++)
{
    if ((h_C0Host[i] - h_C0[i]) != 0)
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols/2);
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\n", Percent_Error);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < nElementsDevice; i++)
{
    if ((h_C1Host[i] - h_C1[i]) != 0)
    {
        Number_of_Errors++;
    }
}
```
Percent_Error = 100.0*Number_of_Errors/(rows*cols/2);
fprintf(TimeFile, "%.3f\n", Percent_Error);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
printf("Tesla C1060 Error Percentage: %.3f\n\n", Percent_Error);
//-----------------------------------------------------------------------------//
//HOST MEMORY CLEAN UP.
free(h_A0Host);
free(h_B0Host);
free(h_C0Host);
free(h_A1Host);
free(h_B1Host);
free(h_C1Host);
CudaErrCheck( cudaFreeHost(h_A0) );
CudaErrCheck( cudaFreeHost(h_B0) );
CudaErrCheck( cudaFreeHost(h_C0) );
cudaSetDevice(1);
CudaErrCheck( cudaFreeHost(h_A1) );
CudaErrCheck( cudaFreeHost(h_B1) );
CudaErrCheck( cudaFreeHost(h_C1) );
//Increasing of Matrix Dimensions.
rows = rows + step;
Cols = cols + step;
}
//END PROGRAM
return EXIT_SUCCESS;

B.10 Asynchronous Multiple GPU Depth First Raw Data

<table>
<thead>
<tr>
<th>Value</th>
<th>Time</th>
<th>Error</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>359.723</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1000</td>
<td>7.325</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>1500</td>
<td>11.256</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2000</td>
<td>17.713</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2500</td>
<td>26.301</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3000</td>
<td>35.909</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>3500</td>
<td>50.403</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>4000</td>
<td>62.641</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>4500</td>
<td>82.113</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>5000</td>
<td>96.491</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>5500</td>
<td>121.559</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>6000</td>
<td>137.780</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>6500</td>
<td>169.349</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>7000</td>
<td>187.032</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>7500</td>
<td>225.610</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>8000</td>
<td>242.857</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>8500</td>
<td>289.725</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>9000</td>
<td>307.559</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>9500</td>
<td>360.931</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10000</td>
<td>377.917</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10500</td>
<td>441.109</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>11000</td>
<td>458.290</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>11500</td>
<td>530.952</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>12000</td>
<td>542.943</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>12500</td>
<td>627.567</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>13000</td>
<td>641.095</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>13500</td>
<td>728.334</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>14000</td>
<td>738.234</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>14500</td>
<td>840.721</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>15000</td>
<td>854.478</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>15500</td>
<td>902.592</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>16000</td>
<td>963.613</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>16500</td>
<td>977.031</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>17000</td>
<td>1098.591</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>17500</td>
<td>1038.105</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>18000</td>
<td>1219.514</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>18500</td>
<td>1373.079</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>19000</td>
<td>1458.203</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>19500</td>
<td>1537.079</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>20000</td>
<td>1585.172</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>20500</td>
<td>1153.241</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>
B.11 Asynchronous Multiple GPU Breadth First Test Program

```
#include <npps.h>
#include <nppi.h>
#include <nppcore.h>
#include <cstdio>
#include <cstdlib>
#include <csignal>
#include <ctime>

FILE *TimeFile = fopen("MultiGPU_Async_Breadth_Times.txt", "w");

void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(TimeFile);
    exit(signum);
}

int main(int argc, char **argv)
{
    int rows = 500;
    int cols = 500;
    int step = 500;
    int PointBound = 100;
    signal(SIGINT, signalHandler);
    int i, j;
    int nFunctions = 5;
    int Number_of_Errors;
    float Percent_Error;
    int nStreams;
    int nStreamsDevice;
    int nBytesF;
    int nStreamBytesF;
    int offset;
    while(1)
```

```
srand(time(0));
printf("Rows: %i\n", rows);
printf("Columns: %i\n", cols);
fprintf(TimeFile, "%i ", rows);
float msec;
cudaEvent_t StartEvent, StopEvent;
cudaSetDevice(0);
CudaErrCheck( cudaEventCreate(&StartEvent) );
CudaErrCheck( cudaEventCreate(&StopEvent) );
nElements = rows*cols;
nElementsDevice = nElements/2;
nBytesF = nElementsDevice*sizeof(float);
nStreams = 4;
StreamSize = nElementsDevice/nStreams;
nStreamBytesF = StreamSize*sizeof(float);
float* h_A0Host;
float* h_B0Host;
float* h_C0Host;
float* h_A1Host;
float* h_B1Host;
float* h_C1Host;
float* h_A0;
float* h_B0;
float* h_C0;
float* h_A1;
float* h_B1;
float* h_C1;
h_A0Host = (float*)malloc(nBytesF);
if (h_A0Host == 0) 
{  
    fprintf (stderr, "!!!! host memory allocation error (h_A0Host)\n");
    return EXIT_FAILURE;
}
h_B0Host = (float*)malloc(nBytesF);
if (h_B0Host == 0) 
{  
    fprintf (stderr, "!!!! host memory allocation error (h_B0Host)\n");
    return EXIT_FAILURE;
}
h_C0Host = (float*)malloc(nBytesF);
if (h_C0Host == 0) 
{  
    fprintf (stderr, "!!!! host memory allocation error (h_C0Host)\n");
    return EXIT_FAILURE;
}
h_A1Host = (float*)malloc(nBytesF);
if (h_A1Host == 0) 
{  
    fprintf (stderr, "!!!! host memory allocation error (h_A1Host)\n");
    return EXIT_FAILURE;
}
h_B1Host = (float*)malloc(nBytesF);
if (h_B1Host == 0) 
{  
    fprintf (stderr, "!!!! host memory allocation error (h_B1Host)\n");
    return EXIT_FAILURE;
}
h_C1Host = (float*)malloc(nBytesF);
if (h_C1Host == 0) 
{  
    fprintf (stderr, "!!!! host memory allocation error (h_C1Host)\n");
    return EXIT_FAILURE;
}
/*HOST VARIABLES USED FOR DEVICE 0.
CudaErrCheck( cudaMallocHost((void**)&h_A0, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_B0, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_C0, nBytesF) );
//HOST VARIABLES USED FOR DEVICE 1.
cudaSetDevice(1);
CudaErrCheck( cudaMallocHost((void**)&h_A1, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_B1, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_C1, nBytesF) );
//Creation of random values.
for (i=0; i < nElementsDevice; i++)
{
    h_A0Host[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_B0Host[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_A1Host[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_B1Host[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
}
for (i=0; i < nElementsDevice; i++)
{
    h_A0[i] = h_A0Host[i];
    h_B0[i] = h_B0Host[i];
    h_A1[i] = h_A1Host[i];
    h_B1[i] = h_B1Host[i];
}
memset(h_C0, 0, nBytesF);
memset(h_C1, 0, nBytesF);
//-----------------------------------------------------------------------------//
//CPU Calculations
//h_C = h_A + h_B
for (i=0; i < nElementsDevice; i++)
{
    h_C0Host[i] = h_A0Host[i] + h_B0Host[i];
    h_C1Host[i] = h_A1Host[i] + h_B1Host[i];
}
//-----------------------------------------------------------------------------//
//Start Time for Multi GPU Async Breath Implementation.
cudaSetDevice(0);
CudaErrCheck( cudaEventRecord(StartEvent,0) );
//Creation of Streams.
cudaStream_t s0[nStreams];
cudaStream_t s1[nStreams];
for(i=0; i < nStreams; i++)
{
    cudaStreamCreate(&s0[i]);
}
cudaSetDevice(1);
for(i=0; i < nStreams; i++)
{
    cudaStreamCreate(&s1[i]);
}
float* d_A0;
float* d_A1;
float* d_B0;
float* d_B1;
float* d_C0;
float* d_C1;
//Allocation of DEVICE memory.
cudaSetDevice(0);
CudaErrCheck( cudaMalloc((void **)d_A0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_B0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_C0, nBytesF) );
cudaSetDevice(1);
CudaErrCheck( cudaMalloc((void **)d_A1, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_B1, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_C1, nBytesF) );
//CUDA Breadth First Streams
cudaSetDevice(0);
for (i = 0; i < nStreams; i++)
{
    offset = i * StreamSize;
    CudaErrCheck( cudaMemcpyAsync(&d_A0[offset], &h_A0[offset], nStreamBytesF,
        cudaMemcpyHostToDevice, s0[i]) );
    CudaErrCheck( cudaMemcpyAsync(&d_B0[offset], &h_B0[offset], nStreamBytesF,
        cudaMemcpyHostToDevice, s0[i]) );
    CudaErrCheck( cudaMemcpyAsync(&d_C0[offset], &h_C0[offset], nStreamBytesF,
        cudaMemcpyHostToDevice, s0[i]) );
}
cudaSetDevice(1);
for (i = 0; i < nStreams; i++)
{
    offset = i * StreamSize;
CudaErrCheck( cudaMemcpyAsync(&d_A1[offset], &h_A1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i]) );
CudaErrCheck( cudaMemcpyAsync(&d_B1[offset], &h_B1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i]) );
CudaErrCheck( cudaMemcpyAsync(&d_C1[offset], &h_C1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i]) );
}
cudaSetDevice(0);
for (i = 0; i < nStreams; i++)
{
    offset = i * StreamSize;
nppSetStream(s0[i]);
    for (j=0; j < nFunctions; j++)
    {
        nppsAdd_32f(&d_A0[offset], &d_B0[offset], &d_C0[offset], StreamSize);
    }
}
cudaSetDevice(1);
for (i = 0; i < nStreams; i++)
{
    offset = i * StreamSize;
nppSetStream(s1[i]);
    for (j=0; j < nFunctions; j++)
    {
        nppsAdd_32f(&d_A1[offset], &d_B1[offset], &d_C1[offset], StreamSize);
    }
}
cudaSetDevice(0);
for (i = 0; i < nStreams; i++)
{
    offset = i * StreamSize;
    CudaErrCheck( cudaMemcpyAsync(&h_A0[offset], &d_A0[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s0[i]) );
    CudaErrCheck( cudaMemcpyAsync(&h_B0[offset], &d_B0[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s0[i]) );
    CudaErrCheck( cudaMemcpyAsync(&h_C0[offset], &d_C0[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s0[i]) );
}
cudaSetDevice(1);
for (i = 0; i < nStreams; i++)
{
    offset = i * StreamSize;
    CudaErrCheck( cudaMemcpyAsync(&h_A1[offset], &d_A1[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s1[i]) );
    CudaErrCheck( cudaMemcpyAsync(&h_B1[offset], &d_B1[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s1[i]) );
    CudaErrCheck( cudaMemcpyAsync(&h_C1[offset], &d_C1[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s1[i]) );
}
//Freeing DEVICE Memory.
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_A0) );
CudaErrCheck( cudaFree(d_B0) );
CudaErrCheck( cudaFree(d_C0) );
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_A1) );
CudaErrCheck( cudaFree(d_B1) );
CudaErrCheck( cudaFree(d_C1) );
//Destruction of Streams.
cudaSetDevice(0);
for (i = 0; i < nStreams; i++)
{
    CudaErrCheck( cudaStreamDestroy(s0[i]) );
}
cudaSetDevice(1);
for (i = 0; i < nStreams; i++)
{
    CudaErrCheck( cudaStreamDestroy(s1[i]) );
}
//End Time for GPU Calculations.
cudaSetDevice(0);
CudaErrCheck( cudaEventRecord(StopEvent, 0) );
CudaErrCheck( cudaEventSynchronize(StopEvent) );
CudaErrCheck( cudaEventElapsedTime(&msec, StartEvent, StopEvent) );

//Comparison to check if CPU values are the same as the GPU values.
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < nElementsDevice; i++)
{
    if ((h_C0Host[i] - h_C0[i]) != 0)
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols/2);
fprintf(TimeFile, "%.3f ", Percent_Error);
printf("Number of Calculation Errors for Quadro FX 5800: %i\n", Number_of_Errors);
printf("Quadro FX 5800 Error Percentage: %.3f\%
", Percent_Error);
Number_of_Errors = 0;
Percent_Error = 0;
for (i=0; i < nElementsDevice; i++)
{
    if ((h_C1Host[i] - h_C1[i]) != 0)
    {
        Number_of_Errors++;
    }
}
Percent_Error = 100.0*Number_of_Errors/(rows*cols/2);
fprintf(TimeFile, "%.3f\n", Percent_Error);
printf("Number of Calculation Errors for Tesla C1060: %i\n", Number_of_Errors);
printf("Tesla C1060 Error Percentage: %.3f\%
\n", Percent_Error);

//HOST MEMORY CLEAN UP.
free(h_A0Host);
free(h_B0Host);
free(h_C0Host);
free(h_A1Host);
free(h_B1Host);
free(h_C1Host);
CudaErrCheck( cudaFreeHost(h_A0) );
CudaErrCheck( cudaFreeHost(h_B0) );
CudaErrCheck( cudaFreeHost(h_C0) );
cudaSetDevice(1);
CudaErrCheck( cudaFreeHost(h_A1) );
CudaErrCheck( cudaFreeHost(h_B1) );
CudaErrCheck( cudaFreeHost(h_C1) );

//Increasing of Matrix Dimensions.
rows = rows + step;
cols = cols + step;
}

//END PROGRAM
return EXIT_SUCCESS;

B.12 Asynchronous Multiple GPU Breadth First Raw Data

| 500 315.600 0.000 0.000 |
| 1000 7.846 0.000 0.000 |
| 1500 10.362 0.000 0.000 |
| 2000 15.522 0.000 0.000 |
| 2500 23.566 0.000 0.000 |
| 3000 32.284 0.000 0.000 |
| 3500 43.411 0.000 0.000 |
| 4000 56.097 0.000 0.000 |
| 4500 70.860 0.000 0.000 |
| 5000 86.911 0.000 0.000 |
| 5500 105.548 0.000 0.000 |
| 6000 124.956 0.000 0.000 |
C. SEQUENTIAL VERSION PROGRAM AND RAW DATA

Results for the sequential version of the conflict detection algorithm were analyzed in Chapter 3. The columns in the raw data results are represented by the titles listed in Table C.1.

Table C.1
Column representation for sequential version raw data results

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Tracks</td>
<td># of Geometries</td>
<td>Execution Time in seconds</td>
</tr>
</tbody>
</table>

C.1 Sequential Version Program

//-----------------------------------------------//
//Sequential.cpp
//Nathan Clem
//!!!!!!SEQUENTIAL VERSION!!!!!!
//To compile on Linux terminal
//g++ Sequential_R2.cpp
//./a.out
//-----------------------------------------------//

//Included header files and libraries used in program
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <new>
#include <ctime>
#include <csignal>
using namespace std;
//Defined values for use in program.
#define Tracks 500 //Number of tracks.
#define Geometries 50 //Number of Geometries.
#define TimeDiff 2
#define Buffer 180 //Simulates the buffer time.
#define PointBound 500 //Sets 3D space limit.
#define MaxDist 5 //Sets limit of the initial distance to next point.
#define Length 0.05*PointBound //Length tolerance of geometry (along the z-axis).
#define Width 0.05*PointBound //Width tolerance of geometry (along the y-axis).
#define High 0.05*PointBound //High tolerance of geometry (along the x-axis).
define Low 0.05*PointBound //Low tolerance of geometry (along the x-axis).
//Text files used to store results.
FILE *SequentialViolations = fopen("Sequential_R2_Violations.txt", "w"); //Stores violation results
FILE *SequentialExecutionTimes = fopen("Sequential_R2_Execution_Times.txt", "w"); //Stores runtimes
//Function used to print which side was violated
void PrintSide(int a);
//Function used to catch the user input "ctrl+c" so that data stored in file does
//not get erased when user quits manual quits program.
void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(SequentialExecutionTimes);
    exit(signum);
}
//Main function
int main()
{
    //Register signal and handler.
    signal(SIGINT, signalHandler);
    //Variables used for loops and program runtimes.
    int n, i, j, k;
    double AveTime = 0;
    //Variables used in the creation of the randomly generated tracks.
    int MaxTracks;
    int MaxTracksCount = Tracks;
    float D;
    //Variables used in the creation of the randomly generated geometries.
    int MaxGeometries;
    int MaxGeometriesCount = Geometries;
    float y2, z2, H;
    //Variables for the violation calculations.
    float Bx, By, Bz;
    float NB;
    float N_XPP;
    float T hit;
    float Intersect_PointX[6], Intersect_PointY[6], Intersect_PointZ[6];
    int WithinBound[6];
    float Dist[3];
    float Velocity;
    float Intersect_Time;
    int Total_WithinBound;
    float Intersect_Time_Out[6];
    int Side[6];
    float TimeSmall, TimeLarge;
    //Infinite While Loop used for testing
    while(1)
    {
        //Storage of the number of tracks and geometries.
        //After each iteration of the "while" loop, the
        //number of tracks and/or geometries are increased.
        MaxTracks = MaxTracksCount;
        MaxGeometries = MaxGeometriesCount;
        //Prints out in console (or terminal in Linux) the
        //current number of tracks and geometries that the
        //program is currently undertaking.
        printf("\nTracks: %i\n", MaxTracks);
        printf("Geometries: %i\n", MaxGeometries);
        //Resets to default the random number generator seed so that each time
        //track and geometries are created the values start from the same
        //place. This is to provide consistency in the results.
        srand(1);
        //Since the number of tracks and/or geometries change after each
        //iteration of the "while" loop, the arrays used to store this data
        //needs to be dynamic, thus memory is dynamically allocated.
        //Creation of dynamic arrays used to store the TRACK'S PREVIOUS POINT (location).
        float * PPx;
        PPx = new (nothrow) float[MaxTracks];
        float * PPy;
        PPy = new (nothrow) float[MaxTracks];
        float * PPz;
        PPz = new (nothrow) float[MaxTracks];
        //Creation of dynamic arrays used to store the TRACK'S CURRENT POINT (location).
float * CPx;
CPx = new (nothrow) float[MaxTracks];
float * CPy;
CPy = new (nothrow) float[MaxTracks];
float * CPz;
CPz = new (nothrow) float[MaxTracks];

//"For" loop used to CREATE and STORE PREVIOUS POINT and CURRENT POINT.
for (i = 0; i < MaxTracks; i++)
{
    PPx[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    D = 2*MaxDist*((float)rand()/RAND_MAX)-MaxDist;
    CPx[i] = PPx[i]+D;

    PPy[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    D = 2*MaxDist*((float)rand()/RAND_MAX)-MaxDist;
    CPy[i] = PPy[i]+D;

    PPz[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    D = 2*MaxDist*((float)rand()/RAND_MAX)-MaxDist;
    CPz[i] = PPz[i]+D;
}

//Creation of dynamic arrays used to store the GEOMETRY’S NORTHEAST POINT.
float * NEx;
NEx = new (nothrow) float[MaxGeometries];
float * NEy;
NEy = new (nothrow) float[MaxGeometries];
float * NEz;
NEz = new (nothrow) float[MaxGeometries];

//Creation of dynamic arrays used to store the GEOMETRY’S NORTHWEST POINT.
float * NWx;
NWx = new (nothrow) float[MaxGeometries];
float * NWy;
NWy = new (nothrow) float[MaxGeometries];
float * NWz;
NWz = new (nothrow) float[MaxGeometries];

//Creation of dynamic arrays used to store the GEOMETRY’S SOUTHWEST POINT.
float * SWx;
SWx = new (nothrow) float[MaxGeometries];
float * SWy;
SWy = new (nothrow) float[MaxGeometries];
float * SWz;
SWz = new (nothrow) float[MaxGeometries];

//Creation of dynamic arrays used to store the MAX HEIGHT and MIN HEIGHT.
float * MaxH;
MaxH = new (nothrow) float[MaxGeometries];
float * MinH;
MinH = new (nothrow) float[MaxGeometries];

//Creation of dynamic arrays used to store the GEOMETRY’S NORTH SIDE NORMAL.
float * NormNx;
NormNx = new (nothrow) float[MaxGeometries];
float * NormNy;
NormNy = new (nothrow) float[MaxGeometries];
float * NormNz;
NormNz = new (nothrow) float[MaxGeometries];

//Creation of dynamic arrays used to store the GEOMETRY’S EAST SIDE NORMAL.
float * NormEx;
NormEx = new (nothrow) float[MaxGeometries];
float * NormEy;
NormEy = new (nothrow) float[MaxGeometries];
float * NormEz;
NormEz = new (nothrow) float[MaxGeometries];

//Creation of dynamic arrays used to store the GEOMETRY’S TOP SIDE NORMAL.
float * NormTx;
NormTx = new (nothrow) float[MaxGeometries];
float * NormTy;
NormTy = new (nothrow) float[MaxGeometries];
float * NormTz;

NormTz = new (nothrow) float[MaxGeometries];
//Creation of dynamic arrays used to store the GEOMETRY’S SOUTH SIDE NORMAL.
float * NormSx;
NormSx = new (nothrow) float[MaxGeometries];
float * NormSy;
NormSy = new (nothrow) float[MaxGeometries];
float * NormSz;
NormSz = new (nothrow) float[MaxGeometries];
//Creation of dynamic arrays used to store the GEOMETRY’S WEST SIDE NORMAL.
float * NormWx;
NormWx = new (nothrow) float[MaxGeometries];
float * NormWy;
NormWy = new (nothrow) float[MaxGeometries];
float * NormWz;
NormWz = new (nothrow) float[MaxGeometries];
//Creation of dynamic arrays used to store the GEOMETRY’S BOTTOM SIDE NORMAL.
float * NormBx;
NormBx = new (nothrow) float[MaxGeometries];
float * NormBy;
NormBy = new (nothrow) float[MaxGeometries];
float * NormBz;
NormBz = new (nothrow) float[MaxGeometries];
//"For" loop used to CREATE and STORE GEOMETRIES and their corresponding NORMALS.
for (i = 0; i < MaxGeometries; i++)
{
    //Data storage for geometries.
    //NORTHEAST coordinates of Geometry.
    NEx[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    NEy[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    NEz[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    y2 = NEy[i]-((Width*((float)rand()/RAND_MAX))+1);
    z2 = NEz[i]-((Length*((float)rand()/RAND_MAX))+1);
    //NORTHWEST coordinates of Geometry.
    NWx[i] = NEx[i];
    NWy[i] = y2;
    NWz[i] = NEz[i];
    //SOUTHWEST coordinates of Geometry.
    SWx[i] = NEx[i];
    SWy[i] = y2;
    SWz[i] = z2;
    //SOUTHEAST coordinates of Geometry.
    SEx[i] = NEx[i];
    SEy[i] = NEy[i];
    SEz[i] = z2;
    //MAX HEIGHT of geometry.
    H = (High*((float)rand()/RAND_MAX))+1;
    MaxH[i] = NEx[i] + H;
    //MIN HEIGHT of geometry.
    MinH[i] = NEx[i]-((Low*((float)rand()/RAND_MAX))+1);
    //Data storage for Normals
    //Normal vector for NORTH SIDE of Geometry.
    NormNx[i] = 0;
    NormNy[i] = 0;
    Norm Nz[i] = -H*(y2-NEy[i]);
    //Normal vector for EAST SIDE of Geometry.
    NormEx[i] = 0;
    NormEy[i] = H*(NEz[i]-z2);
    NormEz[i] = 0;
    //Normal vector for TOP SIDE of Geometry.
    NormTx[i] = (y2-NEy[i])*(z2-NEz[i]);
    NormTy[i] = 0;
    NormTz[i] = 0;
    //Normal vector for SOUTH SIDE of Geometry.
    NormSx[i] = 0;
    NormSy[i] = 0;
    NormSz[i] = -H*(NEy[i]-y2);
    //Normal vector for WEST SIDE of Geometry.
    NormWx[i] = 0;
    NormWy[i] = H*(z2-NEz[i]);
    NormWz[i] = 0;
//Normal vector for BOTTOM SIDE of Geometry.
NormBx[i] = -(NEz[i] - z2)*(NEy[i] - y2);
NormBy[i] = 0;
NormBz[i] = 0;

//The purpose of this "for" loop is to have the violation check process run five
//times for each data set of tracks and geometries. The runtimes for each loop
//will be summed together and divided by 5 in order to find the average runtime
//for a set of tracks and geometries. These average runtimes will be used to
//compare the average runtimes of the parallel version of the violation checks.
//This loop will not affect the actual calculations used to find violations.
for (n = 0; n < 5; n++)
{
//Loop number will be printed out in to the console
//in order to keep track of where the program is at while running.
printf("Loop: %i\n", n+1);
//The starting of the clock
clock_t nStart, nEnd;
 nStart = clock();

//!!!!!!!!!!!!!!!!!!!!STARTING VIOLATION CHECK PROCESS!!!!!!!!!!!!!!!!!!!!
//The actual starting point of the violation check process of the program.
//The violation check process is a nested loop consisting of two "for"
//loops. The outer "for" loop represents the tracks and the inner "for"
//loop represents the geometries. Within the inner "for" loop, one track
//will be checked for a violation against each geometry, one geometry at
//a time. Once one track has been checked for a violation against each
// geometry, the program ends the inner "for" loop and starts the process
//over for the next track in line by means of the outer "for" loop. This
//process continues until all tracks have been check for violation
//against all geometries. One track is compared against one geometry at a
//time.
//Outer "for" loop.
for (i = 0; i < MaxTracks; i++)
{
    //Calculate the difference between the current point and previous
    //point
    Bx = CPx[i] - PPx[i];
    By = CPy[i] - PPy[i];
    Bz = CPz[i] - PPz[i];

//Inner "for" loop.
    for (j = 0; j < MaxGeometries; j++)
    {
        //Checks if current location of the track is inside or
        //outside of a geometry. It passes this if statement if
        //the track is INSIDE of the geometry, else the program
        //will skip this if statement.
        if ((CPx[i] >= MinH[j]) && (CPx[i] <= MaxH[j]) && (CPy[i] >= NWy[j]) && (CPy[i] <= NEy[j]) && (CPz[i] >= SEz[j]) && (CPz[i] <= NEz[j])
        {
            //INSIDE: Track is inside geometry.
            //NORTH SIDE violation check.
            //Calculation of intersection point.
            NB = (NormNx[j]*Bx) + (NormNy[j]*By) + (NormNz[j]*Bz);
            N_XPP = (NormNx[j] * (NEx[j] - CPx[i])) + (NormNy[j] * (NEy[j] - CPy[i])) + (NormNz[j] * (NEz[j] - CPz[i]));
            T_hit = N_XPP / NB;
            Intersect_PointX[0] = CPx[i] + (Bx * T_hit);
            Intersect_PointY[0] = CPy[i] + (By * T_hit);
            Intersect_PointZ[0] = CPz[i] + (Bz * T_hit);
            //Checks to see if the intersection point is
            //within the bounds of the plane that represents
            //the side of the geometry. If the intersection
            //point is within the bounds of the plane, then a
            //1 is stored to show that a hit was made.
            if ((Intersect_PointX[0] <= MaxH[j]) && (Intersect_PointX[0] >= MinH[j]) &&
            (Intersect_PointY[0] <= MaxH[j]) && (Intersect_PointY[0] >= MinH[j]) &&
            (Intersect_PointZ[0] <= MaxH[j]) && (Intersect_PointZ[0] >= MinH[j])
            )
        }
(Intersect_PointY[0] >= NWy[j]) &&
(Intersect_PointY[0] <= NEy[j]) && (T_hit >= 0))
{
    WithinBound[0] = 1;
}
else
{
    WithinBound[0] = 0;
}

//EAST SIDE violation check.
NB = (NormEx[j]*Bx)+(NormEy[j]*By)+(NormEz[j]*Bz);
N_XPP = (NormEx[j] * (Ex[j] - CPx[i])) +
(NormEy[j] * (Ey[j] - CPy[i])) + (NormEz[j] *
(Ex[j] - CPx[i]));
T_hit = N_XPP / NB;
Intersect_PointX[1] = CPx[i] + (Bx * T_hit);
Intersect_PointY[1] = CPy[i] + (By * T_hit);
Intersect_PointZ[1] = CPz[i] + (Bz * T_hit);
if ((Intersect_PointX[1] <=
MaxH[j])&&(Intersect_PointX[1] >= MinH[j]) &&
(Intersect_PointZ[1] >= SEz[j]) &&
(Intersect_PointZ[1] <= NEz[j]) && (T_hit >= 0))
{
    WithinBound[1] = 1;
}
else
{
    WithinBound[1] = 0;
}

//TOP SIDE violation check.
NB = (NormTx[j]*Bx)+(NormTy[j]*By)+(NormTz[j]*Bz);
N_XPP = (NormTx[j] * (MaxH[j] - CPx[i])) +
(NormTy[j] * (Ey[j] - CPy[i])) + (NormTz[j] *
(Ex[j] - CPx[i]));
T_hit = N_XPP / NB;
Intersect_PointX[2] = CPx[i] + (Bx * T_hit);
Intersect_PointY[2] = CPy[i] + (By * T_hit);
Intersect_PointZ[2] = CPz[i] + (Bz * T_hit);
if ((Intersect_PointY[2] <=
SEy[j])&&(Intersect_PointY[2] >= SWy[j]) &&
(Intersect_PointZ[2] >= SWz[j]) &&
(Intersect_PointZ[2] <= NWz[j]) && (T_hit >= 0))
{
    WithinBound[2] = 1;
}
else
{
    WithinBound[2] = 0;
}

//SOUTH SIDE violation check.
NB = (NormSx[j]*Bx)+(NormSy[j]*By)+(NormSz[j]*Bz);
N_XPP = (NormSx[j] * (SEx[j] - CPx[i])) +
(NormSy[j] * (SEy[j] - CPy[i])) + (NormSz[j] *
(SEz[j] - CPz[i]));
T_hit = N_XPP / NB;
Intersect_PointX[3] = CPx[i] + (Bx * T_hit);
Intersect_PointY[3] = CPy[i] + (By * T_hit);
Intersect_PointZ[3] = CPz[i] + (Bz * T_hit);
if ((Intersect_PointX[3] <=
MaxH[j])&&(Intersect_PointX[3] >= MinH[j]) &&
(Intersect_PointY[3] >= SWy[j]) &&
(Intersect_PointY[3] <= NWy[j]) && (T_hit >= 0))
{
    WithinBound[3] = 1;
}
else
{
    WithinBound[3] = 0;
}

//WEST SIDE violation check.
NB = (NormWx[j]*Bx)+(NormWy[j]*By)+(NormWz[j]*Bz);
N_XPP = (NormWx[j] * (NWx[j] - CPx[i])) + (NormWy[j] * (NWy[j] - CPy[i])) + (NormWz[j] * (NWz[j] - CPz[i]));
T_hit = N_XPP / NB;
Intersect_PointX[4] = CPx[i] + (Bx * T_hit);
Intersect_PointY[4] = CPy[i] + (By * T_hit);
Intersect_PointZ[4] = CPz[i] + (Bz * T_hit);
{ WithinBound[4] = 1; }
else
{ WithinBound[4] = 0; }

//BOTTOM SIDE violation check.
NB = (NormBx[j]*Bx)+(NormBy[j]*By)+(NormBz[j]*Bz);
N_XPP = (NormBx[j] * (MinH[j] - CPx[i])) + (NormBy[j] * (NEy[j] - CPy[i])) + (NormBz[j] * (NEz[j] - CPz[i]));
T_hit = N_XPP / NB;
Intersect_PointX[5] = CPx[i] + (Bx * T_hit);
Intersect_PointY[5] = CPy[i] + (By * T_hit);
Intersect_PointZ[5] = CPz[i] + (Bz * T_hit);
{ WithinBound[5] = 1; }
else
{ WithinBound[5] = 0; }

//Tells user that there is a violation meaning the current location of the track is inside the geometry.
fprintf(SequentialViolations, "VIOLATION: INSIDE\n");
fprintf(SequentialViolations, "--------------\n");
//Violation information
for (k = 0; k < 6; k++)
{ if (WithinBound[k] == 1)
  { }
//Violation process jumps to here if current position of
//a track is outside the boundaries of a geometry. The
//same logic used when the track is within the bounds of
//the geometry are used here expect it is now known that
//the track is outside geometry, thus finding the
//specific sides of intersection is slightly more
//complicated.

else
{

    //OUTSIDE: track is outside geometry.
    //NORTH SIDE violation check.
    NB = (NormNx[j]*Bx)+(NormNy[j]*By)+(NormNz[j]*Bz);
    N_XPP = (NormNx[j] * (NEx[j] - CPx[i])) +
            (NormNy[j] * (NEY[j] - CPy[i])) + (NormNz[j] *
            (NEz[j] - CPz[i]));
    T_hit = N_XPP / NB;
    Intersect_PointX[0] = CPx[i] + (Bx * T_hit);
    Intersect_PointY[0] = CPy[i] + (By * T_hit);
    Intersect_PointZ[0] = CPz[i] + (Bz * T_hit);
    if ((Intersect_PointX[0] <=
         MaxH[j])&&(Intersect_PointX[0] >= MinH[j]) 
        &&
        (Intersect_PointY[0] >= NWy[j]) 
        &&
        (Intersect_PointY[0] <= NEy[j]) 
        &&
        (T_hit >= 0))
    {
        WithinBound[0] = 1;
    }
    else
    {
        WithinBound[0] = 0;
    }

    //EAST SIDE violation check.
    NB = (NormEx[j]*Bx)+(NormEy[j]*By)+(NormEz[j]*Bz);
    N_XPP = (NormEx[j] * (NEx[j] - CPx[i])) +
            (NormEy[j] * (NEY[j] - CPy[i])) + (NormEz[j] *
            (NEz[j] - CPz[i]));
    T_hit = N_XPP / NB;
    Intersect_PointX[1] = CPx[i] + (Bx * T_hit);
    Intersect_PointY[1] = CPy[i] + (By * T_hit);
    Intersect_PointZ[1] = CPz[i] + (Bz * T_hit);
    if ((Intersect_PointX[1] <=
         MaxH[j])&&(Intersect_PointX[1] >= MinH[j]) 
        &&
        (Intersect_PointZ[1] >= SEz[j]) 
        &&
        (Intersect_PointZ[1] <= NEz[j]) 
        &&
        (T_hit >= 0))
    {
        WithinBound[1] = 1;
    }
    else
    {
        WithinBound[1] = 0;
    }

    //TOP SIDE violation check.
    NB = (NormTx[j]*Bx)+(NormTy[j]*By)+(NormTz[j]*Bz);
    N_XPP = (NormTx[j] * (MaxH[j] - CPx[i])) +
            (NormTy[j] * (NEY[j] - CPy[i])) + (NormTz[j] *
            (NEz[j] - CPz[i]));
    T_hit = N_XPP / NB;
    Intersect_PointX[2] = CPx[i] + (Bx * T_hit);
    Intersect_PointY[2] = CPy[i] + (By * T_hit);
    Intersect_PointZ[2] = CPz[i] + (Bz * T_hit);
}
Intersect_PointZ[2] = CPz[i] + (Bz * T_hit);
if ((Intersect_PointY[2] <=
  (Intersect_PointZ[2] >= SWz[j]) &&
  (Intersect_PointZ[2] <= NWz[j]) && (T_hit >= 0))
  { WithinBound[2] = 1; }
else
  { WithinBound[2] = 0; }

//SOUTH SIDE violation check.
NB = (NormSx[j] * Bx) + (NormSy[j] * By) + (NormSz[j] * Bz);
N_XPP = (NormSx[j] * (Sex[j] - Cpx[i])) +
  (NormSy[j] * (Sy[i] - Cpy[i])) + (NormSz[j] *
  (Ssz[j] - CPz[i]));
T_hit = N_XPP / NB;
Intersect_PointX[3] = Cpx[i] + (Bx * T_hit);
Intersect_PointY[3] = Cpy[i] + (By * T_hit);
Intersect_PointZ[3] = CPz[i] + (Bz * T_hit);
if ((Intersect_PointX[3] <=
  (Intersect_PointY[3] >= SWy[j]) &&
  (Intersect_PointY[3] <= SEy[j]) && (T_hit >= 0))
  { WithinBound[3] = 1; }
else
  { WithinBound[3] = 0; }

//WEST SIDE violation check.
NB = (NormWx[j] * Bx) + (NormWy[j] * By) + (NormWz[j] * Bz);
N_XPP = (NormWx[j] * (NWx[j] - Cpx[i])) +
  (NormWy[j] * (NWy[j] - Cpy[i])) + (NormWz[j] *
  (NWz[j] - CPz[i]));
T_hit = N_XPP / NB;
Intersect_PointX[4] = Cpx[i] + (Bx * T_hit);
Intersect_PointY[4] = Cpy[i] + (By * T_hit);
Intersect_PointZ[4] = CPz[i] + (Bz * T_hit);
if ((Intersect_PointX[4] <=
  (Intersect_PointY[4] >= SWy[j]) &&
  (Intersect_PointY[4] <= NWy[j]) && (T_hit >= 0))
  { WithinBound[4] = 1; }
else
  { WithinBound[4] = 0; }

//BOTTOM SIDE violation check.
NB = (NormBx[j] * Bx) + (NormBy[j] * By) + (NormBz[j] * Bz);
N_XPP = (NormBx[j] * (MinH[j] - Cpx[i])) +
  (NormBy[j] * (NEy[j] - Cpy[i])) + (NormBz[j] *
  (NEz[j] - CPz[i]));
T_hit = N_XPP / NB;
Intersect_PointX[5] = Cpx[i] + (Bx * T_hit);
Intersect_PointY[5] = Cpy[i] + (By * T_hit);
Intersect_PointZ[5] = CPz[i] + (Bz * T_hit);
if ((Intersect_PointY[5] <=
  (Intersect_PointZ[5] >= SWz[j]) &&
  (Intersect_PointZ[5] <= NWz[j]) && (T_hit >= 0))
  { WithinBound[5] = 1; }
else
  { WithinBound[5] = 0; }
// Calculates the total number of intersection points and if any occur then the process of finding the intersection points information will occur. If nor such intersection point occurs then it is known that no violation occurs.
if (Total_WithinBound > 0)
{
    int Index = 0;
    for (k = 0; k < 6; k++)
    {
        if (WithinBound[k] == 1)
        {
            Dist[0] = Intersect_PointX[k] - CPx[i];
            Dist[1] = Intersect_PointY[k] - CPy[i];
            Dist[2] = Intersect_PointZ[k] - CPz[i];
            Velocity = sqrt((Bx * Bx) + (By * By) + (Bz * Bz)) / TimeDiff;
            Side[Index] = k;
            Index++;
        }
    }
    // "For" loop used to reorder the intersection time from smallest to largest. This means it organizes the intersection points from nearest to furthest. This is so confusion will not be present when reviewing the data is case an intersection point occurs at an exact corner of the geometry.
    for (k = 0; k < Index; k++)
    {
        if (Intersect_Time_Out[0] != Intersect_Time_Out[k])
        {
            if (Intersect_Time_Out[0] < Intersect_Time_Out[k])
            {
                TimeSmall = Intersect_Time_Out[0];
                TimeLarge = Intersect_Time_Out[k];
            }
            else
            {
                TimeSmall = Intersect_Time_Out[k];
                TimeLarge = Intersect_Time_Out[0];
            }
            break;
        }
    }
}
else {
    TimeSmall = Intersect_Time_Out[0];
    TimeLarge = Intersect_Time_Out[k];
}

int tag = 0;
//Now that the intersection points are
//arranged from nearest to furthest, the
//information on said intersection
//points can be printed out to a text
//file for a user to review in an
//organized fashion.
for (k = 0; k < Index; k++)
{
    //Prints out intersection
    //information it is the nearer
    //of the intersection points AND
    //if the track is within the
    //buffer time.
    if ((TimeSmall == Intersect_Time_Out[k]) &&
        (Intersect_Time_Out[k] <= Buffer))
    {
        if (tag == 0)
        {
            fprintf(SequentialViolations, "VIOLATION: OUTSIDE\n");
            fprintf(SequentialViolations, "------------------\n");
            tag++;
        }
        fprintf(SequentialViolations, "ENTERING: ");
        PrintSide(Side[k]);
        fprintf(SequentialViolations, "Track ID: %i
", i+1);
        fprintf(SequentialViolations, "Geometry ID: %i
", j+1);
        fprintf(SequentialViolations, "Distance: %.6f
",
            Velocity * Intersect_Time_Out[k]);
        fprintf(SequentialViolations, "Velocity: %.6f
", Velocity);
        fprintf(SequentialViolations, "Intersect Time: %.6f
", Intersect_Time_Out[k]);
        fprintf(SequentialViolations, "Intersect Point: (%.6f, %.6f, %.6f)\n",
            Intersect_PointX[Side[k]],
            Intersect_PointY[Side[k]],
            Intersect_PointZ[Side[k]]);  
    }
}

for (k = 0; k < Index; k++)
{
    //Prints out intersection
    //information if it is farther
//of the intersection points AND
//if the track is within the
//buffer time.
if ((TimeLarge ==
Intersect_Time_Out[k]) &&
(Intersect_Time_Out[k] <=
Buffer))
{
    fprintf(SequentialViolat
ions, "EXIT: ");
    PrintSide(Side[k]);
    fprintf(SequentialViolat
ions, "\nTrack ID: %i\n", i+1);
    fprintf(SequentialViolat
ions, "Geometry
ID: %i\n", j+1);
    fprintf(SequentialViolat
ions, "Distance: %.6f\n", Velocity *
Intersect_Time_Out[k]);
    fprintf(SequentialViolat
ions, "Velocity: %.6f\n", Velocity);
    fprintf(SequentialViolat
ions, "Intersect
Time: %.6f\n",
Intersect_Time_Out[k]);
    fprintf(SequentialViolat
ions, "Intersect Point:
(%.6f, %.6f, %.6f)\n", Intersect_PointX[Side[k]
], Intersect_PointY[Side[k]
], Intersect_PointZ[Side[k]
]);
}

//This is the actual end of the violation check process. This process is
//repeated 4 more times for a total of 5 runs for each set of tracks and
//geometries. This is the point the clock stops and the time ran can be
//calculated.
End = clock();
double time = (End-Start)*1.0/CLOCKS_PER_SEC;
//The summing of the runtimes in order to find the total run time
//for the 5 loops.
AveTime = AveTime + fabs(time);
//At the end of the final loop the average time is calculated and the
//number of tracks and/or geometries are increased.
if (n == 4)
{
    AveTime = AveTime/5;
    fprintf(SequentialExecutionTimes, "%i %i %.3f\n", MaxTracks,
MaxGeometries, AveTime);
    MaxTracksCount = MaxTracksCount + 500;
    MaxGeometriesCount = MaxGeometriesCount + 50;
    AveTime = 0;
}
//The file "Sequential_Violations.txt" is closed and reopened after each
//loop so that file will not become too large because data from previous
//loops will still be in it if it were not closed and reopened. This
//file's main purpose is to simulate the outputting of information from
//the violation checks and not necessarily the actual information itself.
//Though the information is correct, saving the information is not need.
Only the average runtimes are needed when comparing the sequential version with the parallel version of the violation checks.

fclose(SequentialViolations);
SequentialViolations = fopen("Sequential_R2_Violations.txt", "w");

The deletion of the allocated memory used in the program.
delete [] PPx;
delete [] PPy;
delete [] PPz;
delete [] CPx;
delete [] CPy;
delete [] CPz;
delete [] NEx;
delete [] NEy;
delete [] NEz;
delete [] Nnx;
delete [] Nny;
delete [] Nnz;
delete [] Snx;
delete [] Sny;
delete [] Snz;
delete [] Sex;
delete []sey;
delete [] sez;
delete [] maxH;
delete [] minH;
delete [] normNx;
delete [] normNy;
delete [] normNz;
delete [] normEx;
delete [] normEy;
delete [] normEz;
delete [] normTx;
delete [] normTy;
delete [] normTz;
delete [] normSx;
delete [] normSy;
delete [] normSz;
delete [] normNx;
delete [] normNy;
delete [] normNz;
delete [] normBx;
delete [] normBy;
delete [] normBz;

The closing of all files used.
fclose(SequentialViolations);
fclose(SequentialExecutionTimes);
return 0;

//Function used to print out specific side of geometry when displaying information about the violation.
void PrintSide(int a)
{
    if (a == 0)
    {
        fprintf(SequentialViolations, "NorthSide ");
    }
    if (a == 1)
    {
        fprintf(SequentialViolations, "EastSide ");
    }
    if (a == 2)
    {
        fprintf(SequentialViolations, "TopSide ");
    }
    if (a == 3)
    {
        fprintf(SequentialViolations, "SouthSide ");
    }
    if (a == 4)
{  fprintf(SequentialViolations, "WestSide ");
}
if (a == 5)
{
  fprintf(SequentialViolations, "BottomSide ");
}

C.2  Sequential Version Raw Data

500 50  0.006
1000 100  0.018
1500 150  0.040
2000 200  0.064
2500 250  0.104
3000 300  0.144
3500 350  0.198
4000 400  0.256
4500 450  0.324
5000 500  0.402
5500 550  0.480
6000 600  0.576
6500 650  0.674
7000 700  0.780
7500 750  0.900
8000 800  1.022
8500 850  1.154
9000 900  1.292
9500 950  1.436
10000 1000 1.592
10500 1050 1.752
11000 1100 1.928
11500 1150 2.108
12000 1200 2.292
12500 1250 2.486
13000 1300 2.688
13500 1350 2.908
14000 1400 3.116
14500 1450 3.346
15000 1500 3.592
15500 1550 3.832
16000 1600 4.076
16500 1650 4.362
17000 1700 4.620
17500 1750 4.874
18000 1800 5.192
18500 1850 5.460
19000 1900 5.754
19500 1950 6.064
20000 2000 6.394
20500 2050 6.772
21000 2100 7.082
21500 2150 7.456
22000 2200 7.792
22500 2250 8.190
23000 2300 8.502
23500 2350 8.942
24000 2400 9.300
24500 2450 9.738
25000 2500 10.072
25500 2550 10.468
26000 2600 10.934
26500 2650 11.358
27000 2700 11.848
27500 2750 12.324
28000 2800 12.696
28500 2850 13.192
29000 2900 13.608
29500 2950 14.048
<table>
<thead>
<tr>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>3000</td>
<td>14.652</td>
</tr>
<tr>
<td>3050</td>
<td>3050</td>
<td>15.140</td>
</tr>
<tr>
<td>3100</td>
<td>3100</td>
<td>15.432</td>
</tr>
<tr>
<td>3150</td>
<td>3150</td>
<td>15.988</td>
</tr>
<tr>
<td>3200</td>
<td>3200</td>
<td>16.688</td>
</tr>
<tr>
<td>3250</td>
<td>3250</td>
<td>17.632</td>
</tr>
<tr>
<td>3300</td>
<td>3300</td>
<td>17.970</td>
</tr>
<tr>
<td>3350</td>
<td>3350</td>
<td>18.172</td>
</tr>
<tr>
<td>3400</td>
<td>3400</td>
<td>18.812</td>
</tr>
<tr>
<td>3450</td>
<td>3450</td>
<td>19.362</td>
</tr>
<tr>
<td>3500</td>
<td>3500</td>
<td>19.714</td>
</tr>
<tr>
<td>3550</td>
<td>3550</td>
<td>20.262</td>
</tr>
<tr>
<td>3600</td>
<td>3600</td>
<td>21.086</td>
</tr>
<tr>
<td>3650</td>
<td>3650</td>
<td>21.748</td>
</tr>
<tr>
<td>3700</td>
<td>3700</td>
<td>22.150</td>
</tr>
<tr>
<td>3750</td>
<td>3750</td>
<td>22.734</td>
</tr>
<tr>
<td>3800</td>
<td>3800</td>
<td>23.466</td>
</tr>
<tr>
<td>3850</td>
<td>3850</td>
<td>24.096</td>
</tr>
<tr>
<td>3900</td>
<td>3900</td>
<td>24.582</td>
</tr>
<tr>
<td>3950</td>
<td>3950</td>
<td>25.108</td>
</tr>
<tr>
<td>4000</td>
<td>4000</td>
<td>25.898</td>
</tr>
<tr>
<td>4050</td>
<td>4050</td>
<td>27.200</td>
</tr>
<tr>
<td>4100</td>
<td>4100</td>
<td>28.198</td>
</tr>
<tr>
<td>4150</td>
<td>4150</td>
<td>28.168</td>
</tr>
<tr>
<td>4200</td>
<td>4200</td>
<td>28.688</td>
</tr>
<tr>
<td>4250</td>
<td>4250</td>
<td>29.484</td>
</tr>
<tr>
<td>4300</td>
<td>4300</td>
<td>29.768</td>
</tr>
<tr>
<td>4350</td>
<td>4350</td>
<td>30.586</td>
</tr>
<tr>
<td>4400</td>
<td>4400</td>
<td>31.522</td>
</tr>
<tr>
<td>4450</td>
<td>4450</td>
<td>32.270</td>
</tr>
<tr>
<td>4500</td>
<td>4500</td>
<td>32.796</td>
</tr>
<tr>
<td>4550</td>
<td>4550</td>
<td>33.532</td>
</tr>
<tr>
<td>4600</td>
<td>4600</td>
<td>34.422</td>
</tr>
<tr>
<td>4650</td>
<td>4650</td>
<td>35.318</td>
</tr>
<tr>
<td>4700</td>
<td>4700</td>
<td>35.746</td>
</tr>
<tr>
<td>4750</td>
<td>4750</td>
<td>36.292</td>
</tr>
<tr>
<td>4800</td>
<td>4800</td>
<td>37.582</td>
</tr>
<tr>
<td>4850</td>
<td>4850</td>
<td>38.530</td>
</tr>
<tr>
<td>4900</td>
<td>4900</td>
<td>39.166</td>
</tr>
<tr>
<td>4950</td>
<td>4950</td>
<td>39.694</td>
</tr>
<tr>
<td>5000</td>
<td>5000</td>
<td>40.682</td>
</tr>
<tr>
<td>5050</td>
<td>5050</td>
<td>41.574</td>
</tr>
<tr>
<td>5100</td>
<td>5100</td>
<td>41.848</td>
</tr>
<tr>
<td>5150</td>
<td>5150</td>
<td>42.722</td>
</tr>
<tr>
<td>5200</td>
<td>5200</td>
<td>44.014</td>
</tr>
<tr>
<td>5250</td>
<td>5250</td>
<td>44.884</td>
</tr>
<tr>
<td>5300</td>
<td>5300</td>
<td>45.456</td>
</tr>
<tr>
<td>5350</td>
<td>5350</td>
<td>46.374</td>
</tr>
<tr>
<td>5400</td>
<td>5400</td>
<td>48.634</td>
</tr>
<tr>
<td>5450</td>
<td>5450</td>
<td>50.752</td>
</tr>
<tr>
<td>5500</td>
<td>5500</td>
<td>49.914</td>
</tr>
<tr>
<td>5550</td>
<td>5550</td>
<td>49.716</td>
</tr>
<tr>
<td>5600</td>
<td>5600</td>
<td>51.098</td>
</tr>
<tr>
<td>5650</td>
<td>5650</td>
<td>51.954</td>
</tr>
<tr>
<td>5700</td>
<td>5700</td>
<td>52.628</td>
</tr>
<tr>
<td>5750</td>
<td>5750</td>
<td>53.518</td>
</tr>
<tr>
<td>5800</td>
<td>5800</td>
<td>54.716</td>
</tr>
<tr>
<td>5850</td>
<td>5850</td>
<td>55.668</td>
</tr>
<tr>
<td>5900</td>
<td>5900</td>
<td>56.866</td>
</tr>
<tr>
<td>5950</td>
<td>5950</td>
<td>56.998</td>
</tr>
<tr>
<td>6000</td>
<td>6000</td>
<td>58.548</td>
</tr>
<tr>
<td>6050</td>
<td>6050</td>
<td>59.538</td>
</tr>
<tr>
<td>6100</td>
<td>6100</td>
<td>60.258</td>
</tr>
<tr>
<td>6150</td>
<td>6150</td>
<td>61.448</td>
</tr>
<tr>
<td>6200</td>
<td>6200</td>
<td>62.522</td>
</tr>
<tr>
<td>6250</td>
<td>6250</td>
<td>63.510</td>
</tr>
<tr>
<td>6300</td>
<td>6300</td>
<td>63.868</td>
</tr>
<tr>
<td>6350</td>
<td>6350</td>
<td>64.814</td>
</tr>
<tr>
<td>6400</td>
<td>6400</td>
<td>66.658</td>
</tr>
<tr>
<td>6450</td>
<td>6450</td>
<td>67.634</td>
</tr>
<tr>
<td>6500</td>
<td>6500</td>
<td>69.860</td>
</tr>
<tr>
<td>6550</td>
<td>6550</td>
<td>72.266</td>
</tr>
<tr>
<td>Value</td>
<td>Value</td>
<td>Percentage</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------------</td>
</tr>
<tr>
<td>66000</td>
<td>6600</td>
<td>71.162</td>
</tr>
<tr>
<td>66500</td>
<td>6650</td>
<td>71.886</td>
</tr>
<tr>
<td>67000</td>
<td>6700</td>
<td>72.240</td>
</tr>
<tr>
<td>67500</td>
<td>6750</td>
<td>73.284</td>
</tr>
<tr>
<td>68000</td>
<td>6800</td>
<td>75.180</td>
</tr>
<tr>
<td>68500</td>
<td>6850</td>
<td>76.272</td>
</tr>
<tr>
<td>69000</td>
<td>6900</td>
<td>77.074</td>
</tr>
<tr>
<td>69500</td>
<td>6950</td>
<td>78.110</td>
</tr>
<tr>
<td>70000</td>
<td>7000</td>
<td>80.364</td>
</tr>
<tr>
<td>70500</td>
<td>7050</td>
<td>81.166</td>
</tr>
<tr>
<td>71000</td>
<td>7100</td>
<td>81.680</td>
</tr>
<tr>
<td>71500</td>
<td>7150</td>
<td>82.288</td>
</tr>
<tr>
<td>72000</td>
<td>7200</td>
<td>84.288</td>
</tr>
<tr>
<td>72500</td>
<td>7250</td>
<td>85.618</td>
</tr>
<tr>
<td>73000</td>
<td>7300</td>
<td>86.416</td>
</tr>
<tr>
<td>73500</td>
<td>7350</td>
<td>87.392</td>
</tr>
<tr>
<td>74000</td>
<td>7400</td>
<td>89.018</td>
</tr>
<tr>
<td>74500</td>
<td>7450</td>
<td>90.448</td>
</tr>
<tr>
<td>75000</td>
<td>7500</td>
<td>90.534</td>
</tr>
<tr>
<td>75500</td>
<td>7550</td>
<td>91.710</td>
</tr>
<tr>
<td>76000</td>
<td>7600</td>
<td>93.012</td>
</tr>
<tr>
<td>76500</td>
<td>7650</td>
<td>95.108</td>
</tr>
<tr>
<td>77000</td>
<td>7700</td>
<td>95.862</td>
</tr>
<tr>
<td>77500</td>
<td>7750</td>
<td>97.120</td>
</tr>
<tr>
<td>78000</td>
<td>7800</td>
<td>99.046</td>
</tr>
<tr>
<td>78500</td>
<td>7850</td>
<td>100.260</td>
</tr>
<tr>
<td>79000</td>
<td>7900</td>
<td>100.370</td>
</tr>
<tr>
<td>79500</td>
<td>7950</td>
<td>101.686</td>
</tr>
<tr>
<td>80000</td>
<td>8000</td>
<td>104.152</td>
</tr>
<tr>
<td>80500</td>
<td>8050</td>
<td>105.664</td>
</tr>
<tr>
<td>81000</td>
<td>8100</td>
<td>109.500</td>
</tr>
<tr>
<td>81500</td>
<td>8150</td>
<td>112.198</td>
</tr>
<tr>
<td>82000</td>
<td>8200</td>
<td>115.996</td>
</tr>
<tr>
<td>82500</td>
<td>8250</td>
<td>115.898</td>
</tr>
<tr>
<td>83000</td>
<td>8300</td>
<td>112.326</td>
</tr>
<tr>
<td>83500</td>
<td>8350</td>
<td>112.310</td>
</tr>
<tr>
<td>84000</td>
<td>8400</td>
<td>114.766</td>
</tr>
<tr>
<td>84500</td>
<td>8450</td>
<td>115.982</td>
</tr>
<tr>
<td>85000</td>
<td>8500</td>
<td>116.942</td>
</tr>
<tr>
<td>85500</td>
<td>8550</td>
<td>118.346</td>
</tr>
<tr>
<td>86000</td>
<td>8600</td>
<td>120.276</td>
</tr>
<tr>
<td>86500</td>
<td>8650</td>
<td>121.594</td>
</tr>
<tr>
<td>87000</td>
<td>8700</td>
<td>127.834</td>
</tr>
<tr>
<td>87500</td>
<td>8750</td>
<td>123.028</td>
</tr>
<tr>
<td>88000</td>
<td>8800</td>
<td>125.894</td>
</tr>
<tr>
<td>88500</td>
<td>8850</td>
<td>127.322</td>
</tr>
<tr>
<td>89000</td>
<td>8900</td>
<td>127.984</td>
</tr>
<tr>
<td>89500</td>
<td>8950</td>
<td>129.774</td>
</tr>
<tr>
<td>90000</td>
<td>9000</td>
<td>131.514</td>
</tr>
<tr>
<td>90500</td>
<td>9050</td>
<td>133.174</td>
</tr>
<tr>
<td>91000</td>
<td>9100</td>
<td>133.812</td>
</tr>
<tr>
<td>91500</td>
<td>9150</td>
<td>134.628</td>
</tr>
<tr>
<td>92000</td>
<td>9200</td>
<td>137.434</td>
</tr>
<tr>
<td>92500</td>
<td>9250</td>
<td>139.090</td>
</tr>
<tr>
<td>93000</td>
<td>9300</td>
<td>139.726</td>
</tr>
<tr>
<td>93500</td>
<td>9350</td>
<td>142.694</td>
</tr>
<tr>
<td>94000</td>
<td>9400</td>
<td>143.490</td>
</tr>
<tr>
<td>94500</td>
<td>9450</td>
<td>145.248</td>
</tr>
<tr>
<td>95000</td>
<td>9500</td>
<td>144.984</td>
</tr>
<tr>
<td>95500</td>
<td>9550</td>
<td>146.654</td>
</tr>
<tr>
<td>96000</td>
<td>9600</td>
<td>149.706</td>
</tr>
<tr>
<td>96500</td>
<td>9650</td>
<td>151.382</td>
</tr>
<tr>
<td>97000</td>
<td>9700</td>
<td>152.816</td>
</tr>
<tr>
<td>97500</td>
<td>9750</td>
<td>154.710</td>
</tr>
<tr>
<td>98000</td>
<td>9800</td>
<td>156.812</td>
</tr>
<tr>
<td>98500</td>
<td>9850</td>
<td>161.908</td>
</tr>
<tr>
<td>99000</td>
<td>9900</td>
<td>157.444</td>
</tr>
<tr>
<td>99500</td>
<td>9950</td>
<td>159.182</td>
</tr>
<tr>
<td>100000</td>
<td>10000</td>
<td>162.366</td>
</tr>
<tr>
<td>100500</td>
<td>10050</td>
<td>164.120</td>
</tr>
<tr>
<td>101000</td>
<td>10100</td>
<td>164.780</td>
</tr>
<tr>
<td>101500</td>
<td>10150</td>
<td>166.662</td>
</tr>
<tr>
<td>Value1</td>
<td>Value2</td>
<td>Result</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>102000</td>
<td>10200</td>
<td>168.892</td>
</tr>
<tr>
<td>102500</td>
<td>10250</td>
<td>171.596</td>
</tr>
<tr>
<td>103000</td>
<td>10300</td>
<td>178.344</td>
</tr>
<tr>
<td>103500</td>
<td>10350</td>
<td>172.338</td>
</tr>
<tr>
<td>104000</td>
<td>10400</td>
<td>175.582</td>
</tr>
<tr>
<td>104500</td>
<td>10450</td>
<td>177.600</td>
</tr>
<tr>
<td>105000</td>
<td>10500</td>
<td>178.086</td>
</tr>
<tr>
<td>105500</td>
<td>10550</td>
<td>180.044</td>
</tr>
<tr>
<td>106000</td>
<td>10600</td>
<td>182.396</td>
</tr>
<tr>
<td>106500</td>
<td>10650</td>
<td>184.538</td>
</tr>
<tr>
<td>107000</td>
<td>10700</td>
<td>183.678</td>
</tr>
<tr>
<td>107500</td>
<td>10750</td>
<td>195.840</td>
</tr>
<tr>
<td>108000</td>
<td>10800</td>
<td>189.676</td>
</tr>
<tr>
<td>108500</td>
<td>10850</td>
<td>191.708</td>
</tr>
<tr>
<td>109000</td>
<td>10900</td>
<td>199.974</td>
</tr>
<tr>
<td>109500</td>
<td>10950</td>
<td>195.012</td>
</tr>
<tr>
<td>110000</td>
<td>11000</td>
<td>198.082</td>
</tr>
<tr>
<td>110500</td>
<td>11050</td>
<td>198.266</td>
</tr>
<tr>
<td>111000</td>
<td>11100</td>
<td>198.038</td>
</tr>
<tr>
<td>111500</td>
<td>11150</td>
<td>199.718</td>
</tr>
<tr>
<td>112000</td>
<td>11200</td>
<td>203.992</td>
</tr>
<tr>
<td>112500</td>
<td>11250</td>
<td>205.482</td>
</tr>
<tr>
<td>113000</td>
<td>11300</td>
<td>206.350</td>
</tr>
<tr>
<td>113500</td>
<td>11350</td>
<td>208.054</td>
</tr>
<tr>
<td>114000</td>
<td>11400</td>
<td>212.476</td>
</tr>
<tr>
<td>114500</td>
<td>11450</td>
<td>212.842</td>
</tr>
<tr>
<td>115000</td>
<td>11500</td>
<td>212.696</td>
</tr>
<tr>
<td>115500</td>
<td>11550</td>
<td>214.234</td>
</tr>
<tr>
<td>116000</td>
<td>11600</td>
<td>219.900</td>
</tr>
<tr>
<td>116500</td>
<td>11650</td>
<td>220.384</td>
</tr>
<tr>
<td>117000</td>
<td>11700</td>
<td>223.886</td>
</tr>
<tr>
<td>117500</td>
<td>11750</td>
<td>223.024</td>
</tr>
<tr>
<td>118000</td>
<td>11800</td>
<td>226.294</td>
</tr>
<tr>
<td>118500</td>
<td>11850</td>
<td>227.994</td>
</tr>
<tr>
<td>119000</td>
<td>11900</td>
<td>227.988</td>
</tr>
<tr>
<td>119500</td>
<td>11950</td>
<td>229.336</td>
</tr>
<tr>
<td>120000</td>
<td>12000</td>
<td>233.904</td>
</tr>
<tr>
<td>120500</td>
<td>12050</td>
<td>235.764</td>
</tr>
<tr>
<td>121000</td>
<td>12100</td>
<td>236.782</td>
</tr>
<tr>
<td>121500</td>
<td>12150</td>
<td>238.426</td>
</tr>
<tr>
<td>122000</td>
<td>12200</td>
<td>242.558</td>
</tr>
<tr>
<td>122500</td>
<td>12250</td>
<td>246.706</td>
</tr>
<tr>
<td>123000</td>
<td>12300</td>
<td>250.202</td>
</tr>
<tr>
<td>123500</td>
<td>12350</td>
<td>245.168</td>
</tr>
<tr>
<td>124000</td>
<td>12400</td>
<td>249.754</td>
</tr>
<tr>
<td>124500</td>
<td>12450</td>
<td>251.684</td>
</tr>
<tr>
<td>125000</td>
<td>12500</td>
<td>252.590</td>
</tr>
<tr>
<td>125500</td>
<td>12550</td>
<td>254.350</td>
</tr>
<tr>
<td>126000</td>
<td>12600</td>
<td>258.036</td>
</tr>
<tr>
<td>126500</td>
<td>12650</td>
<td>259.806</td>
</tr>
<tr>
<td>127000</td>
<td>12700</td>
<td>259.344</td>
</tr>
<tr>
<td>127500</td>
<td>12750</td>
<td>261.322</td>
</tr>
<tr>
<td>128000</td>
<td>12800</td>
<td>266.096</td>
</tr>
<tr>
<td>128500</td>
<td>12850</td>
<td>268.084</td>
</tr>
<tr>
<td>129000</td>
<td>12900</td>
<td>268.726</td>
</tr>
<tr>
<td>129500</td>
<td>12950</td>
<td>271.340</td>
</tr>
<tr>
<td>130000</td>
<td>13000</td>
<td>274.318</td>
</tr>
<tr>
<td>130500</td>
<td>13050</td>
<td>278.464</td>
</tr>
<tr>
<td>131000</td>
<td>13100</td>
<td>281.576</td>
</tr>
<tr>
<td>131500</td>
<td>13150</td>
<td>286.876</td>
</tr>
<tr>
<td>132000</td>
<td>13200</td>
<td>282.910</td>
</tr>
<tr>
<td>132500</td>
<td>13250</td>
<td>285.704</td>
</tr>
<tr>
<td>133000</td>
<td>13300</td>
<td>285.636</td>
</tr>
<tr>
<td>133500</td>
<td>13350</td>
<td>288.294</td>
</tr>
<tr>
<td>134000</td>
<td>13400</td>
<td>291.780</td>
</tr>
<tr>
<td>134500</td>
<td>13450</td>
<td>294.102</td>
</tr>
<tr>
<td>135000</td>
<td>13500</td>
<td>297.394</td>
</tr>
<tr>
<td>135500</td>
<td>13550</td>
<td>295.874</td>
</tr>
<tr>
<td>136000</td>
<td>13600</td>
<td>301.010</td>
</tr>
<tr>
<td>136500</td>
<td>13650</td>
<td>312.062</td>
</tr>
<tr>
<td>137000</td>
<td>13700</td>
<td>303.602</td>
</tr>
<tr>
<td>137500</td>
<td>13750</td>
<td>385.450</td>
</tr>
<tr>
<td>Value</td>
<td>Round</td>
<td>Value</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>174000</td>
<td>17400</td>
<td>491.720</td>
</tr>
<tr>
<td>174500</td>
<td>17450</td>
<td>494.430</td>
</tr>
<tr>
<td>175000</td>
<td>17500</td>
<td>491.520</td>
</tr>
<tr>
<td>175500</td>
<td>17550</td>
<td>494.898</td>
</tr>
<tr>
<td>176000</td>
<td>17600</td>
<td>502.672</td>
</tr>
<tr>
<td>176500</td>
<td>17650</td>
<td>506.054</td>
</tr>
<tr>
<td>177000</td>
<td>17700</td>
<td>505.434</td>
</tr>
<tr>
<td>177500</td>
<td>17750</td>
<td>509.218</td>
</tr>
<tr>
<td>178000</td>
<td>17800</td>
<td>514.008</td>
</tr>
<tr>
<td>178500</td>
<td>17850</td>
<td>518.218</td>
</tr>
<tr>
<td>179000</td>
<td>17900</td>
<td>513.912</td>
</tr>
<tr>
<td>179500</td>
<td>17950</td>
<td>517.674</td>
</tr>
<tr>
<td>180000</td>
<td>18000</td>
<td>525.712</td>
</tr>
<tr>
<td>180500</td>
<td>18050</td>
<td>529.658</td>
</tr>
<tr>
<td>181000</td>
<td>18100</td>
<td>528.932</td>
</tr>
<tr>
<td>181500</td>
<td>18150</td>
<td>532.478</td>
</tr>
<tr>
<td>182000</td>
<td>18200</td>
<td>537.114</td>
</tr>
<tr>
<td>182500</td>
<td>18250</td>
<td>541.244</td>
</tr>
<tr>
<td>183000</td>
<td>18300</td>
<td>537.552</td>
</tr>
<tr>
<td>183500</td>
<td>18350</td>
<td>541.254</td>
</tr>
<tr>
<td>184000</td>
<td>18400</td>
<td>549.354</td>
</tr>
<tr>
<td>184500</td>
<td>18450</td>
<td>554.762</td>
</tr>
<tr>
<td>185000</td>
<td>18500</td>
<td>551.946</td>
</tr>
<tr>
<td>185500</td>
<td>18550</td>
<td>556.850</td>
</tr>
<tr>
<td>186000</td>
<td>18600</td>
<td>561.302</td>
</tr>
<tr>
<td>186500</td>
<td>18650</td>
<td>563.814</td>
</tr>
<tr>
<td>187000</td>
<td>18700</td>
<td>561.720</td>
</tr>
<tr>
<td>187500</td>
<td>18750</td>
<td>566.628</td>
</tr>
<tr>
<td>188000</td>
<td>18800</td>
<td>573.068</td>
</tr>
<tr>
<td>188500</td>
<td>18850</td>
<td>577.330</td>
</tr>
<tr>
<td>189000</td>
<td>18900</td>
<td>576.410</td>
</tr>
<tr>
<td>189500</td>
<td>18950</td>
<td>579.934</td>
</tr>
<tr>
<td>190000</td>
<td>19000</td>
<td>585.562</td>
</tr>
<tr>
<td>190500</td>
<td>19050</td>
<td>596.152</td>
</tr>
<tr>
<td>191000</td>
<td>19100</td>
<td>587.418</td>
</tr>
<tr>
<td>191500</td>
<td>19150</td>
<td>591.448</td>
</tr>
<tr>
<td>192000</td>
<td>19200</td>
<td>598.000</td>
</tr>
<tr>
<td>192500</td>
<td>19250</td>
<td>601.172</td>
</tr>
<tr>
<td>193000</td>
<td>19300</td>
<td>602.444</td>
</tr>
<tr>
<td>193500</td>
<td>19350</td>
<td>604.500</td>
</tr>
<tr>
<td>194000</td>
<td>19400</td>
<td>611.662</td>
</tr>
<tr>
<td>194500</td>
<td>19450</td>
<td>614.308</td>
</tr>
<tr>
<td>195000</td>
<td>19500</td>
<td>618.934</td>
</tr>
<tr>
<td>195500</td>
<td>19550</td>
<td>613.404</td>
</tr>
<tr>
<td>196000</td>
<td>19600</td>
<td>626.359</td>
</tr>
<tr>
<td>196500</td>
<td>19650</td>
<td>634.986</td>
</tr>
<tr>
<td>197000</td>
<td>19700</td>
<td>632.284</td>
</tr>
<tr>
<td>197500</td>
<td>19750</td>
<td>638.006</td>
</tr>
<tr>
<td>198000</td>
<td>19800</td>
<td>637.248</td>
</tr>
<tr>
<td>198500</td>
<td>19850</td>
<td>639.208</td>
</tr>
<tr>
<td>199000</td>
<td>19900</td>
<td>635.962</td>
</tr>
<tr>
<td>199500</td>
<td>19950</td>
<td>639.034</td>
</tr>
<tr>
<td>200000</td>
<td>20000</td>
<td>658.832</td>
</tr>
</tbody>
</table>
D. PARALLEL VERSION PROGRAMS AND RAW DATA

Results for the parallel versions of the conflict detection algorithm were analyzed in Chapter 6. The columns in the raw data results are represented by the titles listed in Table D.1. For the synchronous parallel version, the execution time was recorded in seconds, whereas the asynchronous versions of the parallel conflict detection algorithm were recording in milliseconds.

Table D.1
Column representation for parallel version raw data results

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Tracks</td>
<td># of Geometries</td>
<td>Execution Time</td>
</tr>
</tbody>
</table>

D.1 Synchronous Parallel Version Program

```c
#include <thrust/device_ptr.h>
#include <thrust/sequence.h>
#include <thrust/remove.h>
#include <thrust/count.h>
#include <cublas.h>
#include <npps.h>
#include <nppi.h>
#include <cstdio>
#include <cstdlib>
#include <ctime>
```

//Defined variables used for TRACK data.
#define MaxTracks 500 //Sets number of total Tracks.
#define StepT 500 //Sets the number of tracks to add on after every loop;
```
```
The following defined constants are used to limit the size of a Geometry.
Because the coordinates of the Geometries are randomly generated, these constants
will limit the size of the Geometries in order to prevent the scenario in which
a Geometry fills the entire coordinate set space. Limiting the size of the Geometries
also helps to reduce the overlapping of Geometries.
#define MaxGeometries 50 //Max number of Geometries to create.
#define StepG 50 //Sets the number of the geometries to add after every loop.
#define Length 0.05*PointBound //Length tolerance (along the z-axis).
#define Width 0.05*PointBound //Width tolerance (along the y-axis).
#define High 0.05*PointBound //High tolerance (along the x-axis).
#define Low 0.05*PointBound //Low tolerance (along the x-axis).
//Random number parameters.
#define PointBound 500 //Sets set space limit.
#define MaxDist 5 //Sets limit of the initial distance to next point.
//Arbitrary time used to describe the difference in time from when
//the current point was collected and when the previous point was collected.
#define TimeDiff 2
//Arbitrary time used to describe the time limit of a potential violation
//of a geometry by a track. If the time until intersection is less than this
//value, then an immediate violation has occurred. Assuming the intersection
//point is within the bounds of the geometry
#define Buffer 180

Text files used to store results.
FILE *ViolationsPar = fopen("Parallel_NoLoop_Cuda_R1_Violations.txt", "w");
FILE *ViolationsParTimes = fopen("Parallel_NoLoop_Cuda_R1_Execution_Times.txt", "w");

Structure predicate used to find values equal to zero.
struct equal_to_zero
{
  __host__ __device__
  bool operator()(const float x)
  {
    return x == 0;
  }
};

Structure predicate used to find values not equal to zero.
struct not_equal_to_zero
{
  __host__ __device__
  bool operator()(const float x)
  {
    return x != 0;
  }
};

Use to exit in case an infinite loop occurs. Press "CTRL C" to exit loop.
void signalHandler( int signum )
{
  printf("Interrupt signal (%i) received.\n", signum);
  fclose(ViolationsParTimes);
  fclose(ViolationsPar);
  exit(signum);
}

Main
int main(int argc, char** argv)
{
  //cudaSetDevice(1);
  //Initializing of Array Lengths.
  int colsT = MaxTracks;
  int colsG = MaxGeometries;
  while(1)
  {
    //Register signal and handler.
    signal(SIGINT, signalHandler);
    //Index variables used in the FOR LOOPS throughout program.
    int i, j, g, k;
    srand(4);
    printf("\nTracks: %i\n", colsT);
    printf("Geometries: %i\n", colsG);
//Development of TRACK (3 x T) matrix.
float* h_TP;
float* h_TC;
float Dist;
//Allocate HOST memory (CPU) for Matrix TP (Previous Track Coordinates).
h_TP = (float*)malloc(3*colsT*sizeof(float));
if (h_TP == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_TP)\n");
    return EXIT_FAILURE;
}
//Allocate HOST memory (CPU) for Matrix TC (Current Track Coordinates).
h_TC = (float*)malloc(3*colsT*sizeof(float));
if (h_TC == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_TC)\n");
    return EXIT_FAILURE;
}
//For loop used to store created previous and current points.
for (i = 0; i < (3*colsT); i++)
{
    //Creates the initial location of a Track.
    h_TP[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    //Calculates distance to next point for each coordinate.
    Dist = 2*MaxDist*((float)rand()/RAND_MAX)-MaxDist;
    //Creates the current location of a Track.
    h_TC[i] = h_TP[i]+Dist;
}

//Development of TCX, TCY, TCZ, BX, BY, and BZ (1 x 6GT)
float* h_TCX;
float* h_TCY;
float* h_TCZ;
float* h_BX;
float* h_BY;
float* h_BZ;
//Allocate HOST memory (CPU) for Array TCX (1 x 6GT).
h_TCX = (float*)malloc(6*colsG*colsT*sizeof(float));
if (h_TCX == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_TCX)\n");
    return EXIT_FAILURE;
}
//Allocate HOST memory (CPU) for Array TCY (1 x 6GT).
h_TCY = (float*)malloc(6*colsG*colsT*sizeof(float));
if (h_TCY == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_TCY)\n");
    return EXIT_FAILURE;
}
//Allocate HOST memory (CPU) for Array TCZ (1 x 6GT).
h_TCZ = (float*)malloc(6*colsG*colsT*sizeof(float));
if (h_TCZ == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_TCZ)\n");
    return EXIT_FAILURE;
}
//Allocate HOST memory (CPU) for Array BX (1 x 6GT).
h_BX = (float*)malloc(6*colsG*colsT*sizeof(float));
if (h_BX == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_BX)\n");
    return EXIT_FAILURE;
}
//Allocate HOST memory (CPU) for Array BY (1 x 6GT).
h_BY = (float*)malloc(6*colsG*colsT*sizeof(float));
if (h_BY == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_BY)\n");
    return EXIT_FAILURE;
}
//Allocate HOST memory (CPU) for Array BZ (1 x 6GT).
h_BZ = (float*)malloc(6*colsG*colsT*sizeof(float));
if (h_BZ == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_BZ)\n");
    return EXIT_FAILURE;
}

//For loop used to store repeated current and repeated previous points.
k = 0;
for (i = 0; i < (6*colsG); i++)
{
    for (j = 0; j < (3*colsT); j=j+3)
    {
        h_TCX[k] = h_TC[j];
        h_TCY[k] = h_TC[j+1];
        h_TCZ[k] = h_TC[j+2];
        h_BX[k] = h_TP[j];
        h_BY[k] = h_TP[j+1];
        h_BZ[k] = h_TP[j+2];
        k = k + 1;
    }
}

//-----------------------------------------------------------------------------
//Development of GEOMETRY (14 x G) matrix and NX, NY, NZ (1 x 6GT) array.
//The normals of each plane of each geometry is need to find the
//intersection point between the plane and a track.
float* h_G;
float* h_NX;
float* h_NY;
float* h_NZ;

//Allocate HOST memory (CPU) for Matrix G (Geometry Coordinates).
h_G = (float*)malloc(14*colsG*sizeof(float));
if (h_G == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_G)\n");
    return EXIT_FAILURE;
}

//Allocate HOST memory (CPU) for Array NX (Geometry Normals X).
h_NX = (float*)malloc(6*colsG*colsT*sizeof(float));
if (h_NX == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_NX)\n");
    return EXIT_FAILURE;
}

//Allocate HOST memory (CPU) for Array NY (Geometry Normals Y).
h_NY = (float*)malloc(6*colsG*colsT*sizeof(float));
if (h_NY == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_NY)\n");
    return EXIT_FAILURE;
}

//Allocate HOST memory (CPU) for Array NZ (Geometry Normals Z).
h_NZ = (float*)malloc(6*colsG*colsT*sizeof(float));
if (h_NZ == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_NZ)\n");
    return EXIT_FAILURE;
}

//Nested for loop used to create and store geometry parameters and corresponding
//normals. Each normal of a plane needs to be repeated MaxTrack amount of times
//to account for each track that is to be compared to each plane of each geometry.
k = 0;
g = 0;
for (i = 0; i < (14*colsG); (i=i+14))
{
    //Data storage for Geometry coordinates.
    //h_G[i] to h_G[i+2] are the NE coordinates of Geometry.
    h_G[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_G[i+1] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_G[i+2] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
\[ y_2 = h_G[i+1] - ((\text{Width} \cdot (\text{float} \cdot \text{rand}()) / \text{RAND\_MAX}) + 1); \]
\[ z_2 = h_G[i+2] - ((\text{Length} \cdot (\text{float} \cdot \text{rand}()) / \text{RAND\_MAX}) + 1); \]

//h_G[i+3] to h_G[i+5] are the NW coordinates of Geometry.
\[ h_G[i+3] = h_G[i]; \]
\[ h_G[i+4] = y_2; \]
\[ h_G[i+5] = h_G[i+2]; \]

//h_G[i+6] to h_G[i+8] are the SW coordinates of Geometry.
\[ h_G[i+6] = h_G[i]; \]
\[ h_G[i+7] = y_2; \]
\[ h_G[i+8] = z_2; \]

//h_G[i+9] to h_G[i+11] are the SE coordinates of Geometry.
\[ h_G[i+9] = h_G[i]; \]
\[ h_G[i+10] = h_G[i+1]; \]
\[ h_G[i+11] = z_2; \]

\[ H = (\text{High} \cdot (\text{float} \cdot \text{rand}()) / \text{RAND\_MAX}) + 1; \]
\[ h_G[i+12] = h_G[i] + H; \]

\[ h_G[i+13] = h_G[i] - ((\text{Low} \cdot (\text{float} \cdot \text{rand}()) / \text{RAND\_MAX}) + 1); \]

\[ \text{for (j = 0; j < \text{colsT}; j++)} \{
\]
  //Data storage for Geometry normals.
  //Normal vector for NORTH SIDE of Geometry.
  \[ h_NX[k] = 0; \]
  \[ h_NY[k] = 0; \]
  \[ h_NZ[k] = -H \cdot (y_2 - h_G[i+1]); \]
  //Normal vector for EAST SIDE of Geometry.
  \[ h_NX[k+\text{colsT}] = 0; \]
  \[ h_NY[k+\text{colsT}] = \text{H} \cdot (h_G[i+2] - z_2); \]
  \[ h_NZ[k+\text{colsT}] = 0; \]
  //Normal vector for TOP SIDE of Geometry.
  \[ h_NX[k+(2 \cdot \text{colsT})] = (y_2 - h_G[i+1]) \cdot (z_2 - h_G[i+2]); \]
  \[ h_NY[k+(2 \cdot \text{colsT})] = 0; \]
  \[ h_NZ[k+(2 \cdot \text{colsT})] = 0; \]
  //Normal vector for SOUTH SIDE of Geometry.
  \[ h_NX[k+(3 \cdot \text{colsT})] = 0; \]
  \[ h_NY[k+(3 \cdot \text{colsT})] = 0; \]
  \[ h_NZ[k+(3 \cdot \text{colsT})] = -H \cdot (h_G[i+1] - y_2); \]
  //Normal vector for WEST SIDE of Geometry.
  \[ h_NX[k+(4 \cdot \text{colsT})] = 0; \]
  \[ h_NY[k+(4 \cdot \text{colsT})] = \text{H} \cdot (z_2 - h_G[i+2]); \]
  \[ h_NZ[k+(4 \cdot \text{colsT})] = 0; \]
  //Normal vector for BOTTOM SIDE of Geometry.
  \[ h_NX[k+(5 \cdot \text{colsT})] = -(h_G[i+2] - z_2) \cdot (h_G[i+1] - y_2); \]
  \[ h_NY[k+(5 \cdot \text{colsT})] = 0; \]
  \[ h_NZ[k+(5 \cdot \text{colsT})] = 0; \]
  \[ k = k++; \]
\}

\[ g++; \]
\[ k = g \cdot 6 \cdot \text{colsT}; \]

//Development of GHX, GHY, GHZ (1 x 6GT) and GLX, GLY, GLZ (1 x 6GT) Array
//Arrays GH is used to store the high end of each plane of each geometry.
//The high ends of the plane will be used to compare with the intersection point.
//Arrays GL is used to store the low end of each plane of each geometry.
//The lows ends of the plane will be used to compare with the intersection point.
//If the intersection point is BELOW GH AND ABOVE GL, then the intersection point
//is WITHIN THE BORDERS of the plane of the geometry.
float* h_GHX;
float* h_GHY;
float* h_GHZ;
float* h_GLX;
float* h_GLY;
float* h_GLZ;

//Allocate HOST memory (CPU) for Array GHX (1 x 6GT).
\[ \text{h\_GHX} = (\text{float} \cdot \text{malloc}(6 \cdot \text{colsG} \cdot \text{colsT} \cdot \text{sizeof(float)})); \]
\[ \text{if (h\_GHX == 0)} \{
\]
  \[ \text{fprintf (stderr, "!!!! host memory allocation error (h\_GHX)\n");} \]
  \[ \text{return EXIT\_FAILURE; \}
\]
// Allocate HOST memory (CPU) for Array GHY (1 x 6GT).
h_GHY = (float*)malloc(6*colsG*colsT*sizeof(float));
if (h_GHY == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_GHY)\n");
    return EXIT_FAILURE;
}

// Allocate HOST memory (CPU) for Array GHZ (1 x 6GT).
h_GHZ = (float*)malloc(6*colsG*colsT*sizeof(float));
if (h_GHZ == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_GHZ)\n");
    return EXIT_FAILURE;
}

// Allocate HOST memory (CPU) for Array GLX (1 x 6GT).
h_GLX = (float*)malloc(6*colsG*colsT*sizeof(float));
if (h_GLX == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_GLX)\n");
    return EXIT_FAILURE;
}

// Allocate HOST memory (CPU) for Array GLY (1 x 6GT).
h_GLY = (float*)malloc(6*colsG*colsT*sizeof(float));
if (h_GLY == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_GLY)\n");
    return EXIT_FAILURE;
}

// Allocate HOST memory (CPU) for Array GLZ (1 x 6GT).
h_GLZ = (float*)malloc(6*colsG*colsT*sizeof(float));
if (h_GLZ == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_GLZ)\n");
    return EXIT_FAILURE;
}

// Nested for loop to store the high and low ends of each plane. Each high and low
// ends of each plane needs to be repeated MaxTrack amount of times to account
// for each track that is to be compared to each plane of each geometry.

k = 0;
g = 0;
for (i = 0; i < colsG; i++)
{
    for (j = 0; j < colsT; j++)
    {
        // High point for North Side
        h_GHX[k] = h_G[12+(i*14)];
        h_GHY[k] = h_G[1+(i*14)];
        h_GHZ[k] = h_G[2+(i*14)];
        // High point for East Side
        h_GHX[k+colsT] = h_G[12+(i*14)];
        h_GHY[k+colsT] = h_G[1+(i*14)];
        h_GHZ[k+colsT] = h_G[2+(i*14)];
        // High point for Top Side
        h_GHX[k+(2*colsT)] = h_G[12+(i*14)];
        h_GHY[k+(2*colsT)] = h_G[1+(i*14)];
        h_GHZ[k+(2*colsT)] = h_G[2+(i*14)];
        // High point for South Side
        h_GHX[k+(3*colsT)] = h_G[12+(i*14)];
        h_GHY[k+(3*colsT)] = h_G[10+(i*14)];
        h_GHZ[k+(3*colsT)] = h_G[11+(i*14)];
        // High point for West Side
        h_GHX[k+(4*colsT)] = h_G[12+(i*14)];
        h_GHY[k+(4*colsT)] = h_G[4+(i*14)];
        h_GHZ[k+(4*colsT)] = h_G[5+(i*14)];
        // High point for Bottom Side
        h_GHX[k+(5*colsT)] = h_G[13+(i*14)];
        h_GHY[k+(5*colsT)] = h_G[1+(i*14)];
        h_GHZ[k+(5*colsT)] = h_G[2+(i*14)];
        // Low point for North Side
        h_GLX[k] = h_G[13+(i*14)];
    }
h_GLY[k] = h_G[4+(i*14)];
h_GLZ[k] = h_G[2+(i*14)];
//Low point for East Side
h_GLX[k+colsT] = h_G[13+(i*14)];
h_GLY[k+colsT] = h_G[1+(i*14)];
//Low point for Top Side
h_GLX[k+(2*colsT)] = h_G[12+(i*14)];
h_GLY[k+(2*colsT)] = h_G[4+(i*14)];
//Low point for South Side
h_GLX[k+(3*colsT)] = h_G[13+(i*14)];
h_GLY[k+(3*colsT)] = h_G[7+(i*14)];
//Low point for West Side
h_GLX[k+(4*colsT)] = h_G[13+(i*14)];
h_GLY[k+(4*colsT)] = h_G[4+(i*14)];
//Low point for Bottom Side
h_GLX[k+(5*colsT)] = h_G[13+(i*14)];
h_GLY[k+(5*colsT)] = h_G[4+(i*14)];
h_GLZ[k+(5*colsT)] = h_G[11+(i*14)];
}
k++;
g++;

//Development of AX, AY, AZ (1 x 6GT) Arrays.
//These arrays are used to store a point on each plane of each geometry.
//Used in the equation to find Thit.
float* h_AX;
float* h_AY;
float* h_AZ;

//Allocate HOST memory (CPU) for Array AX (1 x 6GT).
const int colsG = 6;
const int colsT = 7;
h_AX = (float*)malloc(colsG*colsT*sizeof(float));
if (h_AX == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_AX)\n");
    return EXIT_FAILURE;
}

//Allocate HOST memory (CPU) for Array AY (1 x 6GT).
const int colsG = 6;
const int colsT = 7;
h_AY = (float*)malloc(colsG*colsT*sizeof(float));
if (h_AY == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_AY)\n");
    return EXIT_FAILURE;
}

//Allocate HOST memory (CPU) for Array AZ (1 x 6GT).
const int colsG = 6;
const int colsT = 7;
h_AZ = (float*)malloc(colsG*colsT*sizeof(float));
if (h_AZ == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_AZ)\n");
    return EXIT_FAILURE;
}

//For loop used to store each a point on each plane of each geometry.
//Each point on a plane needs to be repeated MaxTrack amount of times to account
//for each track that is to be compared to each plane of each geometry.
k = 0;
g = 0;
for (i = 0; i < colsG; i++)
{
    for (j = 0; j < colsT; j++)
    {
        if (j == 0 && j == colsT)
        {
            //Point on plane: North Side.
            h_AX[k] = h_G[12+(i*14)];
            h_AY[k] = h_G[1+(i*14)];
            h_AZ[k] = h_G[2+(i*14)];
        }
        else if (j == 0)
        {
            //Point on plane: East Side.
            h_AX[k+colsT] = h_G[12+(i*14)];
            h_AY[k+colsT] = h_G[1+(i*14)];
            h_AZ[k+colsT] = h_G[2+(i*14)];
        }
    }
}
//Point on plane: Top Side.
h_AX[k+(2*colsT)] = h_G[12+(i*14)];
h_AY[k+(2*colsT)] = h_G[1+(i*14)];
h_AZ[k+(2*colsT)] = h_G[2+(i*14)];

//Point on plane: South Side.
h_AX[k+(3*colsT)] = h_G[13+(i*14)];
h_AY[k+(3*colsT)] = h_G[7+(i*14)];
h_AZ[k+(3*colsT)] = h_G[8+(i*14)];

//Point on plane: West Side.
h_AX[k+(4*colsT)] = h_G[13+(i*14)];
h_AY[k+(4*colsT)] = h_G[7+(i*14)];
h_AZ[k+(4*colsT)] = h_G[8+(i*14)];

//Point on plane: Bottom Side.
h_AX[k+(5*colsT)] = h_G[13+(i*14)];
h_AY[k+(5*colsT)] = h_G[7+(i*14)];
h_AZ[k+(5*colsT)] = h_G[8+(i*14)];

k++;
}
g++;
k = g*6*colsT;

//!!!!!!!!!!!!!!!!!!!STARTING VIOLATION CHECK PROCESS!!!!!!!!!!!!!!!!!!!!
clock_t nStart, nEnd;
nStart = clock();

//Initialize CUBLAS.
cublasStatus status;
status = cublasInit();
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! CUBLAS initialization error\n");
    return EXIT_FAILURE;
}

//STEP 1: Calculation Ray Direction Vector for Each Track (B = C - P)
// C = Current Point
// P = Previous Point
float* d_TCX;
float* d_TCY;
float* d_TCZ;
float* d_BX;
float* d_BY;
float* d_BZ;

//Allocate DEVICE memory (GPU) for Array TCX (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_TCX);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (TCX)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array TCY (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_TCY);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (TCY)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array TCZ (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_TCZ);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (TCZ)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array BX (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_BX);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (BX)\n");
    return EXIT_FAILURE;
}
//Allocate DEVICE memory (GPU) for Array BY (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_BY);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (BY)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array BZ (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_BZ);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (BZ)\n");
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array TCX with the HOST Array TCX.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_TCX, 1, d_TCX, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write TCX)\n");
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array TCY with the HOST Array TCY.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_TCY, 1, d_TCY, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write TCY)\n");
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array TCZ with the HOST Array TCZ.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_TCZ, 1, d_TCZ, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write TCZ)\n");
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array BX with the HOST Array BX.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_BX, 1, d_BX, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write BX)\n");
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array BY with the HOST Array BY.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_BY, 1, d_BY, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write BY)\n");
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array BZ with the HOST Array BZ.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_BZ, 1, d_BZ, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write BZ)\n");
    return EXIT_FAILURE;
}

// Part 1: Calculate P - C
//  d_B = d_B - d_TC
//  (element by element)
//Npp Library used to solve equation B = B - C (Dst = Dst - Src).
//d_TC needs to be preserved for other equations later on in the program.
//Thus, d_TC is recognized as the source. This classification results in
//the elements in the resulting arrays to be negative from what is needed.
nppsSub_32f_I(d_TCX, d_BX, 6*colsG*colsT);
nppsSub_32f_I(d_TCY, d_BY, 6*colsG*colsT);
nppsSub_32f_I(d_TCZ, d_BZ, 6*colsG*colsT);

// Part 2: Calculate -B
//  d_B = -1 * d_B
//  (element by element)
//Npp Library used to switch signs for each element from previous result.
//This is so d_B will now contain C-P instead of P-C because d_TC was
//considered the source and needed to be conserved for later used.
nppsMulC_32f_I(-1, d_BX, 6*colsG*colsT);
nppsMulC_32f_I(-1, d_BY, 6*colsG*colsT);
nppsMulC_32f_I(-1, d_BZ, 6*colsG*colsT);

//STEP 2: Calculation T_hit (T_hit = (N . (A - C)) / (N . B))
// N = Normal of Plane of Geometry
// A = Point on Plane
// C = Current Point
// B = Result from Step 1
float* d_AX;
float* d_AY;
float* d_AZ;
float* d_NX;
float* d_NY;
float* d_NZ;
float* d_Thit;

//Allocate DEVICE memory (GPU) for Array AX (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_AX);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! device memory allocation error (AX)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array AY (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_AY);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! device memory allocation error (AY)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array AZ (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_AZ);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! device memory allocation error (AZ)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array NX (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_NX);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! device memory allocation error (NX)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array NY (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_NY);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! device memory allocation error (NY)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array NZ (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_NZ);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! device memory allocation error (NZ)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array Thit (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_Thit);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! device memory allocation error (Thit)\n");
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array AX with the HOST Array AX.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_AX, 1, d_AX, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! device access error (write AX)\n");
    return EXIT_FAILURE;
}
//Initialize the DEVICE Array AY with the HOST Array AY.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_AY, 1, d_AY, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write AY)\n"); 
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array AZ with the HOST Array AZ.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_AZ, 1, d_AZ, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write AZ)\n"); 
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array NX with the HOST Array NX.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_NX, 1, d_NX, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write NX)\n"); 
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array NY with the HOST Array NY.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_NY, 1, d_NY, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write NY)\n"); 
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array NZ with the HOST Array NZ.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_NZ, 1, d_NZ, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write NZ)\n"); 
    return EXIT_FAILURE;
}

// Part 1: Calculating A - C.
//     d_A = d_A - d_TC  
//     (element by element)
nppsSub_32f_I(d_TCX, d_AX, 6*colsG*colsT);
nppsSub_32f_I(d_TCY, d_AY, 6*colsG*colsT);
nppsSub_32f_I(d_TCZ, d_AZ, 6*colsG*colsT);

//     d_A = d_A * d_N  
//     (element by element)
//There is no NPP and CUBLAS dot product function in this software's current  
//library, so finding the dot product requires a two step process. First, the  
//element by element multiplication of the two matrices. Then the summation  
//of each column of the previous step's result.
nppsMul_32f_I(d_NX, d_AX, 6*colsG*colsT);  
nppsMul_32f_I(d_NY, d_AY, 6*colsG*colsT);  
nppsMul_32f_I(d_NZ, d_AZ, 6*colsG*colsT);
//     d_A = d_A + d_A  
//     (element by element)
//Addition of each result is equivalent to summation of the columns. The final  
//result (d_AZ) is now the numerator of the Thit equation (AZ = N . (A - C)).
nppsAdd_32f_I(d_AX, d_AY, 6*colsG*colsT);

// Part 3: Calculate Dot Product of N and B.
//     d_N = d_N * d_B  
//     (element by element)
//Same logic as STEP 2, Part 2
nppsMul_32f_I(d_BX, d_NX, 6*colsG*colsT);  
nppsMul_32f_I(d_BY, d_NY, 6*colsG*colsT);  
nppsMul_32f_I(d_BZ, d_NZ, 6*colsG*colsT);
//     d_N = d_N + d_N  
//     (element by element)
//The final result (d_NZ) is now the denominator of Thit equation (NZ = N . B).
nppsAdd_32f_I(d_NX, d_NY, 6*colsG*colsT);

// Part 4: Divide previous results.
//     d_Thit = d_A / d_N  
//     (element by element)
nppsDiv_32f(d_NZ, d_AZ, d_Thit, 6*colsG*colsT);
//Memory clean up for Array AX, AY, AZ.
free(h_AX);
status = cublasFree(d_AX);
if (status != CUBLAS_STATUS_SUCCESS)
{   fprintf (stderr, "!!!! memory free error (AX)\n");
    return EXIT_FAILURE;
}
free(h_AY);
status = cublasFree(d_AY);
if (status != CUBLAS_STATUS_SUCCESS)
{   fprintf (stderr, "!!!! memory free error (AY)\n");
    return EXIT_FAILURE;
}
free(h_AZ);
status = cublasFree(d_AZ);
if (status != CUBLAS_STATUS_SUCCESS)
{   fprintf (stderr, "!!!! memory free error (AZ)\n");
    return EXIT_FAILURE;
}
//Memory clean up for Array NX, NY, NZ.
free(h_NX);
status = cublasFree(d_NX);
if (status != CUBLAS_STATUS_SUCCESS)
{   fprintf (stderr, "!!!! memory free error (NX)\n");
    return EXIT_FAILURE;
}
free(h_NY);
status = cublasFree(d_NY);
if (status != CUBLAS_STATUS_SUCCESS)
{   fprintf (stderr, "!!!! memory free error (NY)\n");
    return EXIT_FAILURE;
}
free(h_NZ);
status = cublasFree(d_NZ);
if (status != CUBLAS_STATUS_SUCCESS)
{   fprintf (stderr, "!!!! memory free error (NZ)\n");
    return EXIT_FAILURE;
}
//-----------------------------------------------------------------------------//
//STEP 3: Calculation of Intersection Point (I = C + B*Thit)
// C = Current Point
// B = Result from Step 1
// Thit = Result from Step 2
float* d_IntPX;
float* d_IntPY;
float* d_IntPZ;
//Allocate DEVICE memory (GPU) for Array IntPX (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_IntPX);
if (status != CUBLAS_STATUS_SUCCESS)
{   fprintf (stderr, "!!!! device memory allocation error (IntPX)\n");
    return EXIT_FAILURE;
}
//Allocate DEVICE memory (GPU) for Array IntPY (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_IntPY);
if (status != CUBLAS_STATUS_SUCCESS)
{   fprintf (stderr, "!!!! device memory allocation error (IntPY)\n");
    return EXIT_FAILURE;
}
//Allocate DEVICE memory (GPU) for Array IntPZ (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_IntPZ);
if (status != CUBLAS_STATUS_SUCCESS)
{   fprintf (stderr, "!!!! device memory allocation error (IntPZ)\n");
    return EXIT_FAILURE;
return EXIT_FAILURE;
} // Part 1: Copy Array B
    d_IntP = d_B
    // (element by element)
    //This is done to preserve Array B for later use in program.
    nppsCopy_32f(d_BX, d_IntPX, 6*colsG*colsT);
    nppsCopy_32f(d_BY, d_IntPY, 6*colsG*colsT);
    nppsCopy_32f(d_BZ, d_IntPZ, 6*colsG*colsT);
    // Part 2: Calculate B*Thit
    d_IntP = d_IntP * d_Thit
    // (element by element)
    nppsMul_32f_I(d_Thit, d_IntPX, 6*colsG*colsT);
    nppsMul_32f_I(d_Thit, d_IntPY, 6*colsG*colsT);
    nppsMul_32f_I(d_Thit, d_IntPZ, 6*colsG*colsT);
    // Part 3: Calculate I = C + B*Thit
    d_IntP = d_IntP + d_TC
    // (element by element)
    nppsAdd_32f_I(d_TCX, d_IntPX, 6*colsG*colsT);
    nppsAdd_32f_I(d_TCY, d_IntPY, 6*colsG*colsT);
    nppsAdd_32f_I(d_TCZ, d_IntPZ, 6*colsG*colsT);
    //STEP 4: Determine if intersection point is within bounds of geometry.
    NppiSize sz;
    sz.width = 6*colsG*colsT;
    sz.height = 1;
    float* d_GHX;
    float* d_GHY;
    float* d_GHZ;
    unsigned char* d_ComHX;
    unsigned char* d_ComHY;
    unsigned char* d_ComHZ;
    float* d_GLX;
    float* d_GLY;
    float* d_GLZ;
    unsigned char* d_ComLX;
    unsigned char* d_ComLY;
    unsigned char* d_ComLZ;
    //Allocate DEVICE memory (GPU) for Array GHX (1 x 6GT).
    status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_GHX);
    if (status != CUBLAS_STATUS_SUCCESS)
    {
        fprintf (stderr, "!!!! device memory allocation error (GHX)\n");
        return EXIT_FAILURE;
    }
    //Allocate DEVICE memory (GPU) for Array GHY (1 x 6GT).
    status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_GHY);
    if (status != CUBLAS_STATUS_SUCCESS)
    {
        fprintf (stderr, "!!!! device memory allocation error (GHY)\n");
        return EXIT_FAILURE;
    }
    //Allocate DEVICE memory (GPU) for Array GHZ (1 x 6GT).
    status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_GHZ);
    if (status != CUBLAS_STATUS_SUCCESS)
    {
        fprintf (stderr, "!!!! device memory allocation error (GHZ)\n");
        return EXIT_FAILURE;
    }
    //Allocate DEVICE memory (GPU) for Array ComHX (1 x 6GT).
    status = cublasAlloc(6*colsG*colsT, sizeof(unsigned char), (void**)&d_ComHX);
    if (status != CUBLAS_STATUS_SUCCESS)
    {
        fprintf (stderr, "!!!! device memory allocation error (ComHX)\n");
        return EXIT_FAILURE;
    }
    //Allocate DEVICE memory (GPU) for Array ComHY (1 x 6GT).
    status = cublasAlloc(6*colsG*colsT, sizeof(unsigned char), (void**)&d_ComHY);
    if (status != CUBLAS_STATUS_SUCCESS)
    {
        fprintf (stderr, "!!!! device memory allocation error (ComHY)\n");
        return EXIT_FAILURE;
    }
//Allocate DEVICE memory (GPU) for Array ComHZ (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(unsigned char), (void**)&d_ComHZ);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (ComHZ)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array GLX (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_GLX);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (GLX)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array GLY (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_GLY);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (GLY)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array GLZ (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_GLZ);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (GLZ)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array ComLX (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(unsigned char), (void**)&d_ComLX);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (ComLX)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array ComLY (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(unsigned char), (void**)&d_ComLY);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (ComLY)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array ComLZ (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(unsigned char), (void**)&d_ComLZ);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (ComLZ)\n");
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array GHX with the HOST Array GHX.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_GHX, 1, d_GHX, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write GHX)\n");
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array GHY with the HOST Array GHY.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_GHY, 1, d_GHY, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write GHY)\n");
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array GHZ with the HOST Array GHZ.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_GHZ, 1, d_GHZ, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device access error (write GHZ)\n");
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array GLX with the HOST Array GLX.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_GLX, 1, d_GLX, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! device access error (write GLX)\n" );
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array GLY with the HOST Array GLY.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_GLY, 1, d_GLY, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! device access error (write GLY)\n" );
    return EXIT_FAILURE;
}

//Initialize the DEVICE Array GLZ with the HOST Array GLZ.
status = cublasSetMatrix(1, 6*colsG*colsT, sizeof(float), h_GLZ, 1, d_GLZ, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! device access error (write GLZ)\n" );
    return EXIT_FAILURE;
}

// d_IntP = d_IntP - 0.00001
// (element by element)
//Since the variable IntP was calculated and not predetermined, and that
//each element of Intp is limited to 32 bits, there will be a loss of precision.
//In order to pass the geometry border checks these precision errors have to be
//accounted for. This is done by creating a tolerance for the IntP. This first
//equations sets up the lower tolerance of IntP.
nppsSubC_32f_I(0.0001, d_IntPX, 6*colsG*colsT);
nppsSubC_32f_I(0.0001, d_IntPY, 6*colsG*colsT);
nppsSubC_32f_I(0.0001, d_IntPZ, 6*colsG*colsT);

//Part 2: Compares intersection point (d_IntP) with the larger border points of
//each plane (d_GH).
// If d_IntP > d_GH, then d_ComH = 255 (TRUE).
// If d_IntP <= d_GH, then d_ComH = 0 (FALSE).
// (element by element)
//For purposes of this program, the desired result of this comparison is to have
//each corresponding element of the arrays be equal to zero. This would mean
//every coordinate of the intersection point is less than or equal to the higher
//ends of the plane of the geometry.
nppiCompare_32f_C1R(d_IntPX, 6*colsG*colsT*sizeof(float), d_GHX, 6*colsG*colsT*sizeof(uchar), sz, NPP_CMP_GREATER);
nppiCompare_32f_C1R(d_IntPY, 6*colsG*colsT*sizeof(float), d_GHY, 6*colsG*colsT*sizeof(uchar), sz, NPP_CMP_GREATER);
nppiCompare_32f_C1R(d_IntPZ, 6*colsG*colsT*sizeof(float), d_GHZ, 6*colsG*colsT*sizeof(uchar), sz, NPP_CMP_GREATER);

// d_IntP = d_IntP + 0.00002
// (element by element)
//The same logic as STEP 4, Part 1. But this time dealing with the lower ends
//of the plane of the geometry. the value 0.00002 is added to account for
//the 0.00001 that was subtracted in a previous equation.
nppsAddC_32f_I(0.0002, d_IntPX, 6*colsG*colsT);
nppsAddC_32f_I(0.0002, d_IntPY, 6*colsG*colsT);
nppsAddC_32f_I(0.0002, d_IntPZ, 6*colsG*colsT);

//Part 4: Compares intersection point (d_IntP) with the smaller border points of
//each plane (d_GL).
// If d_IntP < d_GL, then d_ComH = 255 (TRUE).
// If d_IntP >= d_GL, then d_ComH = 0 (FALSE).
// (element by element)
//For purposes of this program, the desired result of this comparison is to have
//every corresponding element of the arrays be equal to zero. This would mean
//every coordinate of the intersection point is greater than or equal to the
//lower ends of the plane of the geometry.
nppiCompare_32f_C1R(d_IntPX, 6*colsG*colsT*sizeof(float), d_GLX, 6*colsG*colsT*sizeof(uchar), sz, NPP_CMP_LESS);


nppiCompare_32f_C1R(d_IntPY, 6*colsG*colsT*sizeof(float), d_GLY,
6*colsG*colsT*sizeof(float), d_ComLY, 6*colsG*colsT*sizeof(unsigned char), sz,
NPP_CMP_LESS);

nppiCompare_32f_C1R(d_IntPZ, 6*colsG*colsT*sizeof(float), d_GLZ,
6*colsG*colsT*sizeof(float), d_ComLZ, 6*colsG*colsT*sizeof(unsigned char), sz,
NPP_CMP_LESS);

//  d_IntP = d_IntP - 0.00001
//  (element by element)
//Return d_IntP back to its original value
nppsSubC_32f_I(0.0001, d_IntPX, 6*colsG*colsT);
nppsSubC_32f_I(0.0001, d_IntPY, 6*colsG*colsT);
nppsSubC_32f_I(0.0001, d_IntPZ, 6*colsG*colsT);

//Memory clean up for Array GHX, GHY, GHZ.
free(h_GHX);
status = cublasFree(d_GHX);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (GHX)\n");
    return EXIT_FAILURE;
}

free(h_GHY);
status = cublasFree(d_GHY);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (GHY)\n");
    return EXIT_FAILURE;
}

free(h_GHZ);
status = cublasFree(d_GHZ);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (GHZ)\n");
    return EXIT_FAILURE;
}

//Memory clean up for Array GLX, GLY, GLZ.
free(h_GLX);
status = cublasFree(d_GLX);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (GLX)\n");
    return EXIT_FAILURE;
}

free(h_GLY);
status = cublasFree(d_GLY);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (GLY)\n");
    return EXIT_FAILURE;
}

free(h_GLZ);
status = cublasFree(d_GLZ);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (GLZ)\n");
    return EXIT_FAILURE;
}

//-----------------------------------------------------------------------------//
//STEP 5: Calculate distance from current point to intersecting point.
//DistInt = sqrt(CoorDistX^2 * CoorDistY^2 * CoorDistZ^2)
float* d_CoorDistX;
float* d_CoorDistY;
float* d_CoorDistZ;
float* d_DistInt;

//Allocate DEVICE memory (GPU) for Array CoorDistX (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_CoorDistX);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (CoorDistX)\n");
    return EXIT_FAILURE;
}

//Allocate DEVICE memory (GPU) for Array CoorDistY (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_CoorDistY);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (CoorDistY)\n");
    return EXIT_FAILURE;
}
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (CoorDistY)\n");
    return EXIT_FAILURE;
}

// Allocate DEVICE memory (GPU) for Array CoorDistZ (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_CoorDistZ);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (CoorDistZ)\n");
    return EXIT_FAILURE;
}

// Allocate DEVICE memory (GPU) for Array DistInt (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_DistInt);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (DistInt)\n");
    return EXIT_FAILURE;
}

// Part 1: Calculation of difference for each coordinate of each point.
//  d_CoorDist = d_IntP - d_TC2
// (element by element)
nppsSub_32f(d_TCX, d_IntPX, d_CoorDistX, 6*colsG*colsT);
nppsSub_32f(d_TCY, d_IntPY, d_CoorDistY, 6*colsG*colsT);
nppsSub_32f(d_TCZ, d_IntPZ, d_CoorDistZ, 6*colsG*colsT);

// Part 2: Square each coordinate difference
//  d_CoorDist = CoorDist^2
// (element by element)
nppsSqr_32f_I(d_CoorDistX, 6*colsG*colsT);
nppsSqr_32f_I(d_CoorDistY, 6*colsG*colsT);
nppsSqr_32f_I(d_CoorDistZ, 6*colsG*colsT);

// Part 3: Summation of each squared coordinate.
//  d_DistInt = d_CoorDistX + d_CoorDistY + d_CoorDistZ
// (element by element)
// In other words the summation of each column.
nppsAdd_32f_I(d_CoorDistX, d_CoorDistY, d_CoorDistZ, 6*colsG*colsT);
nppsCopy_32f(d_CoorDistZ, d_DistInt, 6*colsG*colsT);

// Part 4: Calculate square root of each element.
//  d_DistInt = sqrt(d_DistInt)
// (element by element)
// The final result will be the distance from the current point to
// the intersection point.
nppsSqrt_32f_I(d_DistInt, 6*colsG*colsT);

// Memory clean up for Array CoorDistX, CoorDistY, CoorDistZ.
status = cublasFree(d_CoorDistX);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (CoorDistX)\n");
    return EXIT_FAILURE;
}
status = cublasFree(d_CoorDistY);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (CoorDistY)\n");
    return EXIT_FAILURE;
}
status = cublasFree(d_CoorDistZ);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (CoorDistZ)\n");
    return EXIT_FAILURE;
}

//-----------------------------------------------------------------------------
// STEP 6: Calculate velocity (Vel = B / TimeDiff)
float* d_Vel;

// Allocate DEVICE memory (GPU) for Array Vel (1 x 6GT).
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_Vel);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! device memory allocation error (Vel)\n");
    return EXIT_FAILURE;
}
284

}   // Part 1: Square the distance between current point and previous point.  
   //  d_B = d_B^2  
   // (element by element)  
nppsSqr_32f_I(d_BX, 6*colsG*colsT);  
nppsSqr_32f_I(d_BY, 6*colsG*colsT);  
nppsSqr_32f_I(d_BZ, 6*colsG*colsT);  

   // Part 2: Summation of each column.  
   //  d_Vel = d_BX + d_BY + d_BZ  
   // (normal Array multiplication)  
nppsAdd_32f_I(d_BX, d_BY, 6*colsG*colsT);  
nppsAdd_32f_I(d_BY, d_BZ, 6*colsG*colsT);  
nppsCopy_32f(d_BZ, d_Vel, 6*colsG*colsT);  

   // Part 3: Calculate square root of each element.  
   //  d_Vel = sqrt(d_Vel)  
   // (element by element)  
The result is the distance travel from the previous point to current  
//point and will be divided by a predetermined (defined) arbitrary time.  
nppsSqrt_32f_I(d_Vel, 6*colsG*colsT);  

   // Part 4: Calculate velocity.  
   //  d_Vel = d_Vel / TimeDiff  
   // (element by element)  
nppsDivC_32f_I(TimeDiff, d_Vel, 6*colsG*colsT);  

   //STEP 7: Calculate time until intersection (TimeInt = DistInt / Vel).  
float* d_TimeInt;  
//Allocate DEVICE memory (GPU) for Array TimeInt (1 x 6GT).  
status = cublasAlloc(6*colsG*colsT, sizeof(float), (void**)&d_TimeInt);  
if (status != CUBLAS_STATUS_SUCCESS)  
{
  fprintf (stderr, "!!!! device memory allocation error (TimeInt)\n");  
  return EXIT_FAILURE;
}

   // Part 1: Divide distance from intersection point by velocity of track  
   //  d_TimeInt = d_DistInt / d_Vel  
   // (element by element)  
nppsDiv_32f(d_Vel, d_DistInt, d_TimeInt, 6*colsG*colsT);  

   //STEP 8: Finding violation with ComH, ComL, ComBuffer, and Thit comparisons.  
unsigned char* d_ComTotal;  
unsigned char* d_ComThit;  
unsigned char* d_ComBuffer;  
int* d_Index;  
int result;  
//Allocate DEVICE memory (GPU) for Array ComTotal (1 x 6GT).  
status = cublasAlloc(6*colsG*colsT, sizeof(unsigned char), (void**)&d_ComTotal);  
if (status != CUBLAS_STATUS_SUCCESS)  
{
  fprintf (stderr, "!!!! device memory allocation error (ComTotal)\n");  
  return EXIT_FAILURE;
}

   //Allocate DEVICE memory (GPU) for Array ComThit (1 x 6GT).  
status = cublasAlloc(6*colsG*colsT, sizeof(unsigned char), (void**)&d_ComThit);  
if (status != CUBLAS_STATUS_SUCCESS)  
{
  fprintf (stderr, "!!!! device memory allocation error (ComThit)\n");  
  return EXIT_FAILURE;
}

   //Allocate DEVICE memory (GPU) for Array ComBuffer (1 x 6GT).  
status = cublasAlloc(6*colsG*colsT, sizeof(unsigned char), (void**)&d_ComBuffer);  
if (status != CUBLAS_STATUS_SUCCESS)  
{
  fprintf (stderr, "!!!! device memory allocation error (ComBuffer)\n");  
  return EXIT_FAILURE;
}

   //Allocate DEVICE memory (GPU) for Array Index (1 x 6GT).  
status = cublasAlloc(6*colsG*colsT, sizeof(int), (void**)&d_Index);  
if (status != CUBLAS_STATUS_SUCCESS)  
{
  fprintf (stderr, "!!!! device memory allocation error (Index)\n");  
  return EXIT_FAILURE;
}
Part 1: Adds \( d_{\text{ComH}} \) and \( d_{\text{ComL}} \) element by element.
\[
d_{\text{ComL}} = d_{\text{ComH}} + d_{\text{ComL}}
\]
(element by element)

Since \( \text{ComH} \) and \( \text{ComL} \) are unsigned char, the max value for the element by element sum is '255'. This is okay since the only value that matters is '0'. The value '0' means the element in the Array is within the borders of a geometry.

When the corresponding elements in \( \text{ComH} \) and \( \text{ComL} \) add up to '0',
nppsAdd_8u_ISfs(d_ComHX, d_ComLX, 6*colsG*colsT, 0);
nppsAdd_8u_ISfs(d_ComHY, d_ComLY, 6*colsG*colsT, 0);
nppsAdd_8u_ISfs(d_ComHZ, d_ComLZ, 6*colsG*colsT, 0);

Part 3: Summation of each column in \( \text{ComL} \).
\[
d_{\text{ComLZ}} = d_{\text{ComLX}} + d_{\text{ComLY}} + d_{\text{ComLZ}}
\]
(element by element)

This results in one array representing all coordinate comparisons with the geometry highs and lows.

nppsAdd_8u_ISfs(d_ComLX, d_ComLY, 6*colsG*colsT, 0);
nppsAdd_8u_ISfs(d_ComLY, d_ComLZ, 6*colsG*colsT, 0);

Part 4: Checking to see if \( \text{Thit} \) is positive.
\[
\text{If } d_{\text{Thit}} < 0, \text{ then } d_{\text{ComThit}} = 255
\]
\[
\text{If } d_{\text{Thit}} \geq 0, \text{ then } d_{\text{ComThit}} = 0
\]
(element by element)

From this comparison we can tell if \( \text{Thit} \) is positive or negative.

It an element in \( \text{Thit} \) is negative then the intersection point occurred behind the track which means nothing. But if the element in \( \text{Thit} \) is positive then the intersection point is occurring in front of the track which means the track is heading towards the geometry.

nppiCompareC_32f_C1R(d_Thit, 6*colsG*colsT*sizeof(float), 0, d_ComThit, 6*colsG*colsT*sizeof(unsigned char), sz, NPP_CMP_LESS);

Part 6: Adds \( d_{\text{ComThit}} \) and \( d_{\text{ComTotal}} \) element by element.
\[
d_{\text{ComThit}} = d_{\text{ComThit}} + d_{\text{ComLZ}}
\]
(element by element)

If an element in the resulting \( \text{ComThit} \) equals '0',
then that means that the Intersection point is within the boundaries of the plane of the geometry and it is occurring ahead of the track. Otherwise, it is not a violation.

nppsAdd_8u_ISfs(d_ComLZ, d_ComThit, 6*colsG*colsT, 0);

Part 7: Compare each time until intersection with the buffer time.
\[
\text{If } d_{\text{TimeInt}} > \text{Buffer}, \text{ then } d_{\text{ComBuffer}} = 255
\]
\[
\text{If } d_{\text{TimeInt}} \leq \text{Buffer}, \text{ then } d_{\text{ComBuffer}} = 0
\]
(element by element)

The desired result is 0. This means the time until intersection is within the buffer zone, thus an immediate violation has occurred.

Assuming the intersection point is within the bounds of the plane of the geometry.

nppiCompareC_32f_C1R(d_TimeInt, 6*colsG*colsT*sizeof(float), Buffer, d_ComBuffer, 6*colsG*colsT*sizeof(unsigned char), sz, NPP_CMP_GREATER);

Part 9: Adds \( d_{\text{ComBufferF}} \) and \( d_{\text{ComTotal}} \) element by element.
\[
d_{\text{ComTotal}} = d_{\text{ComBuffer}} + d_{\text{ComThit}}
\]
(element by element)

If an element in the resulting \( \text{ComTotal} \) equals '0',
then that means that the Intersection point is within the boundaries of the plane of the geometry and it is occurring ahead of the track and it is within the buffer zone. Otherwise, it is not a violation.

nppsAdd_8u_Sfs(d_ComThit, d_ComBuffer, d_ComTotal, 6*colsG*colsT, 0);

Part 10: Creates an array used to track indices.

The purpose of the Index variable is to represent the Indices of the final array of comparison results. If we know which elements in \( \text{ComTotal} \) equal '0', then we can find out which tracks and geometries are in violation of each other. Once the array \( d_{\text{Index}} \) is created, the program counts how many elements in \( \text{ComTotal} \) equal '0'. This results tells us the size in which to make the final index array, \( \text{Index 2} \), which will be used to store the indices of the elements that are equal to '0' in \( \text{ComTotal} \).

thrust::device_ptr<int> dev_ptr_Index(d_Index);
thrust::sequence(dev_ptr_Index, dev_ptr_Index + (6*colsG*colsT));
thrust::device_ptr<unsigned char> dev_ptr_ComTotal(d_ComTotal);
result = thrust::count_if(dev_ptr_ComTotal, dev_ptr_ComTotal + (6*colsG*colsT), equal_to_zero());
if (result != 0)
{
    int* d_Index2;
    //Allocate DEVICE memory (GPU) for Array Index2 (1 x 6GT).
    status = cublasAlloc(result, sizeof(int), (void**)&d_Index2);
    if (status != CUBLAS_STATUS_SUCCESS)
    {
        fprintf (stderr, "!!!! device memory allocation error
(Index2)\n\n");
        return EXIT_FAILURE;
    }
    thrust::device_ptr<int> dev_ptr_Index2(d_Index2);
    //This is the function that will remove all elements in Index that
    //correspond to values in ComTotal that are not equal to '0' and place
    //what is left into Index2.
    thrust::remove_copy_if(dev_ptr_Index, dev_ptr_Index + 6*colsG*colsT,
              dev_ptr_ComTotal, dev_ptr_Index2, not_equal_to_zero());
    //---------------------------------------------------------------------//
    //STEP 9: Output Violations Results
    int TrackID;
    int GeometryID;
    int* h_Index2;
    float* h_DistInt;
    float* h_Vel;
    float* h_TimeInt;
    float* h_IntPX;
    float* h_IntPY;
    float* h_IntPZ;
    //Allocate HOST memory (CPU) for Array Index2 (1 x result).
    h_Index2 = (int*)malloc(result*sizeof(int));
    if (h_Index2 == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error
(h_Index2)\n\n");
        return EXIT_FAILURE;
    }
    //Allocate HOST memory (CPU) for Array DistInt (1 x 6GT empty Array).
    h_DistInt = (float*)malloc(6*colsG*colsT*sizeof(float));
    if (h_DistInt == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error
(h_DistInt)\n\n");
        return EXIT_FAILURE;
    }
    //Allocate HOST memory (CPU) for Array Vel (1 x 6GT empty Array).
    h_Vel = (float*)malloc(6*colsG*colsT*sizeof(float));
    if (h_Vel == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error
(h_Vel)\n\n");
        return EXIT_FAILURE;
    }
    //Allocate HOST memory (CPU) for Array TimeInt (1 x 6GT empty Array).
    h_TimeInt = (float*)malloc(6*colsG*colsT*sizeof(float));
    if (h_TimeInt == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error
(h_TimeInt)\n\n");
        return EXIT_FAILURE;
    }
    //Allocate HOST memory (CPU) for Array IntPX (1 x 6GT empty Array).
    h_IntPX = (float*)malloc(6*colsG*colsT*sizeof(float));
    if (h_IntPX == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error
(h_IntPX)\n\n");
        return EXIT_FAILURE;
    }
    //Allocate HOST memory (CPU) for Array IntPY (1 x 6GT empty Array).
    h_IntPY = (float*)malloc(6*colsG*colsT*sizeof(float));
    if (h_IntPY == 0)
    {
        fprintf (stderr, "!!!! host memory allocation error
(h_IntPY)\n\n");
        return EXIT_FAILURE;
    }
}
//Allocate HOST memory (CPU) for Array IntPZ (1 x 6GT empty Array).
intPZ = (float*)malloc(6*colsG*colsT*sizeof(float));
if (intPZ == 0)
{
    fprintf(stderr, "!!! host memory allocation error (intPZ)\n");
    return EXIT_FAILURE;
}

//Storing data from the DEVICE (GPU) into the HOST (CPU) for each Array.
//Retrieve DEVICE Array Index2 and place it back to HOST Array Index2.
status = cublasGetMatrix(1, result, sizeof(int), d_Index2, 1, h_Index2, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!! device access error (Index2)\n");
    return EXIT_FAILURE;
}

//Retrieve DEVICE Array DistInt and place it back to HOST Array DistInt.
status = cublasGetMatrix(1, 6*colsG*colsT, sizeof(float), d_DistInt, 1, h_DistInt, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!! device access error (DistInt)\n");
    return EXIT_FAILURE;
}

//Retrieve DEVICE Array Vel and place it back to HOST Array Vel.
status = cublasGetMatrix(1, 6*colsG*colsT, sizeof(float), d_Vel, 1, h_Vel, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!! device access error (Vel)\n");
    return EXIT_FAILURE;
}

//Retrieve DEVICE Array TimeInt and place it back to HOST Array TimeInt.
status = cublasGetMatrix(1, 6*colsG*colsT, sizeof(float), d_TimeInt, 1, h_TimeInt, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!! device access error (TimeInt)\n");
    return EXIT_FAILURE;
}

//Retrieve DEVICE Array IntPX and place it back to HOST Array IntPX.
status = cublasGetMatrix(1, 6*colsG*colsT, sizeof(float), d_IntPX, 1, h_IntPX, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!! device access error (IntPX)\n");
    return EXIT_FAILURE;
}

//Retrieve DEVICE Array IntPY and place it back to HOST Array IntPY.
status = cublasGetMatrix(1, 6*colsG*colsT, sizeof(float), d_IntPY, 1, h_IntPY, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!! device access error (IntPY)\n");
    return EXIT_FAILURE;
}

//Retrieve DEVICE Array IntPZ and place it back to HOST Array IntPZ.
status = cublasGetMatrix(1, 6*colsG*colsT, sizeof(float), d_IntPZ, 1, h_IntPZ, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!! device access error (IntPZ)\n");
    return EXIT_FAILURE;
}

//Prints out Violations based on the values in d_Index2.
for (i=0; i<result; i++)
{
    if ((d_Index2[i]+1)%colsT==0)
    {
        TrackID = colsT;
    }
}  
else  
{  
  TrackID = (h_Index2[i]+1)%colsT;  
}  
GeometryID = (h_Index2[i]/(6*colsT))+1;  
//printf("Track ID: %i\n", TrackID);  
//printf("Geometry ID: %i\n", GeometryID);  
fprintf(ViolationsPar, "Track ID: %i\n", TrackID);  
fprintf(ViolationsPar, "Geometry ID: %i\n", GeometryID);  
fprintf(ViolationsPar, "Distance: %.6f\n",  
    h_DistInt[h_Index2[i]]);  
fprintf(ViolationsPar, "Velocity: %.6f\n", h_Vel[h_Index2[i]]);  
fprintf(ViolationsPar, "Intersect Time: %.6f\n",  
    h_TimeInt[h_Index2[i]]);  
fprintf(ViolationsPar, "Intersect Point: (%.6f, %.6f, %.6f)\n",  
    h_IntPX[h_Index2[i]], h_IntPY[h_Index2[i]], h_IntPZ[h_Index2[i]]);  
fprintf(ViolationsPar, "\n");  
}  
//Memory clean up for Array Index2.  
free(h_Index2);  
status = cublasFree(d_Index2);  
if (status != CUBLAS_STATUS_SUCCESS)  
{  
  fprintf(stderr, "!!!! memory free error (Index2)\n");  
  return EXIT_FAILURE;  
}  
//Memory clean up for Array DistInt.  
free(h_DistInt);  
//Memory clean up for Array Vel.  
free(h_Vel);  
//Memory clean up for Array TimeInt.  
free(h_TimeInt);  
//Memory clean up for Array IntPX.  
free(h_IntPX);  
//Memory clean up for Array IntPY.  
free(h_IntPY);  
//Memory clean up for Array IntPZ.  
free(h_IntPZ);  
}  
พฤศจ hindsight!!ENDING VIOLATION CHECK PROCESS!!

nEnd = clock();  
double time = (nEnd-nStart)*1.0/CLOCKS_PER_SEC;  
fprintf(ViolationsParTimes, "%i %i %.3f\n", colsT, colsG, fabs(time));  
colsT = colsT + StepT;  
colsG = colsG + StepG;  
//Freeing up all allocated memory for both HOST (CPU) and DEVICE (GPU)  
//Memory clean up for Matrix TP.  
free(h_TP);  
//Memory clean up for Matrix TC.  
free(h_TC);  
//Memory clean up for Array TCX, TCY, TCZ.  
free(h_TCX);  
status = cublasFree(d_TCX);  
if (status != CUBLAS_STATUS_SUCCESS)  
{  
  fprintf(stderr, "!!!! memory free error (TCX)\n");  
  return EXIT_FAILURE;  
}  
free(h_TCY);  
status = cublasFree(d_TCY);  
if (status != CUBLAS_STATUS_SUCCESS)  
{  
  fprintf(stderr, "!!!! memory free error (TCY)\n");  
  return EXIT_FAILURE;  
}  
free(h_TCZ);  
status = cublasFree(d_TCZ);  
if (status != CUBLAS_STATUS_SUCCESS)
{    fprintf(stderr, "!!!! memory free error (TCZ)\n");    return EXIT_FAILURE;
}

//Memory clean up for Array BX, BY, BZ.
free(h_BX);
status = cublasFree(d_BX);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!!! memory free error (BX)\n");    return EXIT_FAILURE;
}

free(h_BY);
status = cublasFree(d_BY);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!!! memory free error (BY)\n");    return EXIT_FAILURE;
}

free(h_BZ);
status = cublasFree(d_BZ);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!!! memory free error (BZ)\n");    return EXIT_FAILURE;
}

//Memory clean up for Matrix G.
free(h_G);
//Memory clean up for Array Thit
status = cublasFree(d_Thit);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!!! memory free error (Thit)\n");    return EXIT_FAILURE;
}

//Memory clean up for Array IntPX, IntPY, IntPZ.
status = cublasFree(d_IntPX);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!!! memory free error (IntPX)\n");    return EXIT_FAILURE;
}

status = cublasFree(d_IntPY);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!!! memory free error (IntPY)\n");    return EXIT_FAILURE;
}

status = cublasFree(d_IntPZ);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!!! memory free error (IntPZ)\n");    return EXIT_FAILURE;
}

//Memory clean up for Array ComHX, ComHY, ComHZ.
status = cublasFree(d_ComHX);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!!! memory free error (ComHX)\n");    return EXIT_FAILURE;
}

status = cublasFree(d_ComHY);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!!! memory free error (ComHY)\n");    return EXIT_FAILURE;
}

status = cublasFree(d_ComHZ);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!!! memory free error (ComHZ)\n");    return EXIT_FAILURE;
}
Memory clean up for Array ComLX, ComLY, ComLZ.
status = cublasFree(d_ComLX);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (ComLX)\n");
    return EXIT_FAILURE;
}
status = cublasFree(d_ComLY);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (ComLY)\n");
    return EXIT_FAILURE;
}
status = cublasFree(d_ComLZ);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (ComLZ)\n");
    return EXIT_FAILURE;
}
Memory clean up for Array DistInt.
status = cublasFree(d_DistInt);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (DistInt)\n");
    return EXIT_FAILURE;
}
Memory clean up for Array Vel.
status = cublasFree(d_Vel);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (Vel)\n");
    return EXIT_FAILURE;
}
Memory clean up for Array TimeInt.
status = cublasFree(d_TimeInt);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (TimeInt)\n");
    return EXIT_FAILURE;
}
Memory clean up for Array ComTotal, ComThit, ComBuffer.
status = cublasFree(d_ComTotal);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (ComTotal)\n");
    return EXIT_FAILURE;
}
status = cublasFree(d_ComThit);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (ComThit)\n");
    return EXIT_FAILURE;
}
status = cublasFree(d_ComBuffer);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (ComBuffer)\n");
    return EXIT_FAILURE;
}
Memory clean up for Array Index.
status = cublasFree(d_Index);
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf (stderr, "!!!! memory free error (Index)\n");
    return EXIT_FAILURE;
}
//The closing of files used.
fclose(ViolationsPar);
ViolationsPar = fopen("Parallel_NoLoop_Cuda_R1_Violations.txt", "w");
//Shutdown CUBLAS
status = cublasShutdown();
if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! shutdown error\n");
    return EXIT_FAILURE;
}

//END PROGRAM
return EXIT_SUCCESS;

D.2 Synchronous Parallel Version Raw Data

500 50 2.220
1000 100 0.030
1500 150 0.070
2000 200 0.100
2500 250 0.140
3000 300 0.200
3500 350 0.270

D.3 Asynchronous Single GPU Parallel Version Program

#include <thrust/device_ptr.h>
#include <thrust/sequence.h>
#include <thrust/remove.h>
#include <thrust/count.h>
#include <npps.h>
#include <nppi.h>
#include <nppcore.h>
#include <csdio>
#include <cstdlib>
#include <ctime>
#include <signal>

//Defined variables used for TRACK data.
#define MaxTracks 500 //Sets number of total Tracks.
#define StepT 500 //Sets the number of tracks to add on after every loop;
//Defined variables used for GEOMETRY data.
//The following defined constants are used to limit the size of a Geometry.
//Because the coordinates of the Geometries are randomly generated, these constants
//will limit the size of the Geometries in order to prevent the scenario in which
//a Geometry fills the entire coordinate set space. Limiting the size of the Geometries
//also helps to reduce the overlapping of Geometries.
#define MaxGeometries 50 //Max number of Geometries to create.
#define StepG 50 //Sets the number of the geometries to add on after every loop.
#define Length 0.05*PointBound //Length tolerance (along the z-axis).
#define Width 0.05*PointBound //Width tolerance (along the y-axis).
#define High 0.05*PointBound //High tolerance (along the x-axis).
#define Low 0.05*PointBound //Low tolerance (along the x-axis).
//Random number parameters.
#define PointBound 500 //Sets set space limit.
#define MaxDist 5 //Sets limit of the initial distance to next point.
//Arbitrary time used to describe the difference in time from when
//the current point was collected and when the previous point was collected.
#define TimeDiff 2
//Arbitrary time used to describe the time limit of a potential violation
//of a geometry by a track. If the time until intersection is less than this
//value, then an imminent violation has occurred. Assuming the intersection
//point is within the bounds of the geometry
#define Buffer 180

//-----------------------------------------------------------------------------------------//
//Text files used to store results.
FILE *ParallelViolations = fopen("Parallel_Cuda_SingleGPU_Breadth_Violations.txt", "w");
FILE *ParallelExecutionTimes = fopen("Parallel_Cuda_SingleGPU_Breadth_Execution_Times_Ave.txt", "w");
//-----------------------------------------------------------------------------------------//
//Structure predicate used in Thrust functions to find values equal to zero.
struct equal_to_zero
{
    __host__ __device__
    bool operator()(const float x)
    {
        return x == 0;
    }
};
//Structure predicate used in Thrust functions to find values not equal to zero.
struct not_equal_to_zero
{
    __host__ __device__
    bool operator()(const float x)
    {
        return x != 0;
    }
};

//-----------------------------------------------------------------------------------------//
//Use to stop program in case of infinite loop
//Enter: "CTRL + C"
void signalHandler( int signum )
{
    printf("Interrupt signal (%i) received.\n", signum);
    fclose(ParallelExecutionTimes);
    fclose(ParallelViolations);
    exit(signum);
}

//-----------------------------------------------------------------------------------------//
//Cuda Error Check
#define CudaErrCheck(ans) {gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
    if (code != cudaSuccess)
    {
        fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}

//-----------------------------------------------------------------------------------------//
//Main
int main(int argc, char** argv)
{
    cudaSetDevice(1);
    int colsT = MaxTracks;
    int colsG = MaxGeometries;
    signal(SIGINT, signalHandler);
    int i, j, g, k, n;
    float AveTime = 0;
    int result;
    int nElements;
    int nStreams;
    int StreamSize;
    int nBytesI;
    int nBytesF;
    int nStreamBytesF;
    int nBytesUC;
    int offset;
    while(1)
    {
        srand(1);
printf(\"nTracks: %i\n\", colsT);
printf(\"Geometries: %i\n\", colsG);
float msec;
cudaEvent_t StartEvent, StopEvent;
CudaErrCheck(cudaEventCreate(&StartEvent));
CudaErrCheck(cudaEventCreate(&StopEvent));
//Initializing Array Sizes and Number of Streams.
nElements = 6*colsG*colsT;
nBytesI = nElements*sizeof(int);
nBytesF = nElements*sizeof(float);
nBytesUC = nElements*sizeof(unsigned char);
nStreams = 4;
StreamSize = nElements/nStreams;
nStreamBytesF = StreamSize*sizeof(float);
NppiSize sz;
sz.height = 6*colsG/nStreams;
sz.width = colsT;
// Development of TRACK (3 x T) matrix.
float* h_TP;
float* h_TC;
float Dist;
//Allocate HOST memory (CPU) for Matrix TP (Previous Track Coordinates).
h_TP = (float*)malloc(3*colsT*sizeof(float));
if (h_TP == 0)
{
    fprintf(stderr, \"!!!! host memory allocation error (h_TP)\n\");
    return EXIT_FAILURE;
}
//Allocate HOST memory (CPU) for Matrix TC (Current Track Coordinates).
h_TC = (float*)malloc(3*colsT*sizeof(float));
if (h_TC == 0)
{
    fprintf(stderr, \"!!!! host memory allocation error (h_TC)\n\");
    return EXIT_FAILURE;
}
//For loop used to store created previous and current points.
for (i = 0; i < (3*colsT); i++)
{
    //Creates the initial location of a Track.
h_TP[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    //Calculates distance to next point for each coordinate.
    Dist = 2*MaxDist*((float)rand()/RAND_MAX)-MaxDist;
    //Creates the current location of a Track.
h_TC[i] = h_TP[i]+Dist;
}
// Development of TCX, TCY, TCZ, BX, BY, and BZ (1 x 6GT)
float* h_TCX;
float* h_TCY;
float* h_TCZ;
float* h_BX;
float* h_BY;
float* h_BZ;
//Allocate HOST PINNED memory (CPU) for Arrays TCX, TCY, and TCZ (1 x 6GT).
CudaErrCheck(cudaMallocHost((void**)&h_TCX, nBytesF));
CudaErrCheck(cudaMallocHost((void**)&h_TCY, nBytesF));
CudaErrCheck(cudaMallocHost((void**)&h_TCZ, nBytesF));
//Allocate HOST PINNED memory (CPU) for Arrays BX, BY, and BZ (1 x 6GT).
CudaErrCheck(cudaMallocHost((void**)&h_BX, nBytesF));
CudaErrCheck(cudaMallocHost((void**)&h_BY, nBytesF));
CudaErrCheck(cudaMallocHost((void**)&h_BZ, nBytesF));
//For loop used to store repeated current and repeated previous points.
k = 0;
for (i = 0; i < (6*colsG); i++)
{
    for (j = 0; j < (3*colsT); j=j+3)
    {
        h_TCX[k] = h_TC[j];
        h_TCY[k] = h_TC[j+1];
        h_TCZ[k] = h_TC[j+2];
        h_BX[k] = h_TP[j];
    }
h_BY[k] = h_TP[j+1];
h_BZ[k] = h_TP[j+2];
k = k + 1;
}

/**************************************************************************/
//Development of GEOMETRY (14 x G) matrix and NX, NY, NZ (1 x 6GT) array.
//The normals of each plane of each geometry is need to find the
//intersection point between the plane and a track.
float* h_G;
float* h_NX;
float* h_NY;
float* h_NZ;
float y2, z2, H;
//Allocate HOST memory (CPU) for Matrix G (Geometry Coordinates).
h_G = (float*)malloc(14*colsG*sizeof(float));
if (h_G == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_G)\n");
    return EXIT_FAILURE;
}
//Allocate HOST PINNED memory (CPU) for Arrays NX, NY, and NZ (1 x 6GT).
CudaErrCheck( cudaMallocHost((void**)&h_NX, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_NY, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_NZ, nBytesF) );
//Nested for loop used to create and store geometry parameters and corresponding
//normal. Each normal of a plane needs to be repeated MaxTrack amount of times to
//account for each track that is to be compared to each plane of each geometry.
k = 0;
g = 0;
for (i = 0; i < (14*colsG); (i=i+14))
{
    //Data storage for Geometry coordinates.
    //h_G[i] to h_G[i+2] are the NE coordinates of Geometry.
    h_G[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
h_G[i+1] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
h_G[i+2] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
y2 = h_G[i+1]-((Width*((float)rand()/RAND_MAX))+1);
z2 = h_G[i+2]-((Length*((float)rand()/RAND_MAX))+1);
    //h_G[i+3] to h_G[i+5] are the NW coordinates of Geometry.
    h_G[i+3] = h_G[i];
h_G[i+4] = y2;
h_G[i+5] = h_G[i+2];
    //h_G[i+6] to h_G[i+8] are the SW coordinates of Geometry.
    h_G[i+6] = h_G[i];
h_G[i+7] = y2;
h_G[i+8] = z2;
    //h_G[i+9] to h_G[i+11] are the SE coordinates of Geometry.
    h_G[i+9] = h_G[i];
h_G[i+10] = h_G[i+1];
h_G[i+11] = z2;
    H = (High*((float)rand()/RAND_MAX))+1;
    h_G[i+12] = h_G[i]+H;
    h_G[i+13] = h_G[i]-((Low*((float)rand()/RAND_MAX))+1);
    for (j = 0; j < colsT; j++)
    {
        //Data storage for Geometry normals.
        //Normal vector for NORTH SIDE of Geometry.
        h_NX[k] = 0;
h_NY[k] = 0;
h_NZ[k] = -H*(y2-h_G[i+1]);
        //Normal vector for EAST SIDE of Geometry.
        h_NX[k+colsT] = 0;
h_NY[k+colsT] = H*(h_G[i+2]-z2);
h_NZ[k+colsT] = 0;
        //Normal vector for TOP SIDE of Geometry.
        h_NX[k+(2*colsT)] = (y2-h_G[i+1])*(z2-h_G[i+2]);
h_NY[k+(2*colsT)] = 0;
h_NZ[k+(2*colsT)] = 0;
        //Normal vector for SOUTH SIDE of Geometry.
\[ h_{NX}[k+(3*colsT)] = 0; \]
\[ h_{NY}[k+(3*colsT)] = 0; \]
\[ h_{NZ}[k+(3*colsT)] = -H*(h_G[i+1]-y2); \]
\[ // Normal vector for WEST SIDE of Geometry. \]
\[ h_{NX}[k+(4*colsT)] = 0; \]
\[ h_{NY}[k+(4*colsT)] = H*(z2-h_G[i+2]); \]
\[ h_{NZ}[k+(4*colsT)] = 0; \]
\[ // Normal vector for BOTTOM SIDE of Geometry. \]
\[ h_{NX}[k+(5*colsT)] = -(h_G[i+2]-z2)*(h_G[i+1]-y2); \]
\[ h_{NY}[k+(5*colsT)] = 0; \]
\[ h_{NZ}[k+(5*colsT)] = 0; \]
\[ k = k++; \]
\]
\[ g++; \]
\[ k = g*6*colsT; \]
\]
//Development of GHX, GHY, GHZ (1 x 6GT) and GLX, GLY, GLZ (1 x 6GT) Array
//Arrays GH is used to store the high end of each plane of each geometry.
//The high ends of the plane will be used to compare with the intersection point.
//Arrays GL is used to store the low end of each plane of each geometry.
//The lows ends of the plane will be used to compare with the intersection point.
//IF the intersection point is BELOW GH AND ABOVE GL, then the intersection point
//is WITHIN THE BORDERS of the plane of the geometry.
float* h_GHX;
float* h_GHY;
float* h_GHZ;
float* h_GLX;
float* h_GLY;
float* h_GLZ;

//Allocate HOST PINNED memory (CPU) for Arrays GHX, GHY, and GHZ (1 x 6GT).
CudaErrCheck( cudaMallocHost((void**)&h_GHX, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_GHY, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_GHZ, nBytesF) );

//Allocate HOST PINNED memory (CPU) for Arrays GLX, GLY, and GLZ (1 x 6GT).
CudaErrCheck( cudaMallocHost((void**)&h_GLX, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_GLY, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_GLZ, nBytesF) );

//Nested for loop to store the high and low ends of each plane. Each high and low
//ends of each plane needs to be repeated MaxTrack amount of times to account for
//each track that is to be compared to each plane of each geometry.
k = 0;
g = 0;
for (i = 0; i < colsG; i++)
{
    for (j = 0; j < colsT; j++)
    {
        //High point for North Side
        h_GHX[k] = h_G[12+(i*14)];
        h_GHY[k] = h_G[1+(i*14)];
        h_GHZ[k] = h_G[2+(i*14)];
        //High point for East Side
        h_GHX[k+colsT] = h_G[12+(i*14)];
        h_GHY[k+colsT] = h_G[1+(i*14)];
        h_GHZ[k+colsT] = h_G[2+(i*14)];
        //High point for Top Side
        h_GHX[k+(2*colsT)] = h_G[12+(i*14)];
        h_GHY[k+(2*colsT)] = h_G[1+(i*14)];
        h_GHZ[k+(2*colsT)] = h_G[2+(i*14)];
        //High point for South Side
        h_GHX[k+(3*colsT)] = h_G[12+(i*14)];
        h_GHY[k+(3*colsT)] = h_G[10+(i*14)];
        h_GHZ[k+(3*colsT)] = h_G[11+(i*14)];
        //High point for West Side
        h_GHX[k+(4*colsT)] = h_G[12+(i*14)];
        h_GHY[k+(4*colsT)] = h_G[4+(i*14)];
        h_GHZ[k+(4*colsT)] = h_G[5+(i*14)];
        //High point for Bottom Side
        h_GHX[k+(5*colsT)] = h_G[13+(i*14)];
        h_GHY[k+(5*colsT)] = h_G[1+(i*14)];
        h_GHZ[k+(5*colsT)] = h_G[2+(i*14)];
        //Low point for North Side

h_GLX[k] = h_G[13+(i*14)];
h_GLY[k] = h_G[4+(i*14)];
h_GLZ[k] = h_G[2+(i*14)];
//Low point for East Side
h_GLX[k+colsT] = h_G[13+(i*14)];
h_GLY[k+colsT] = h_G[1+(i*14)];
h_GLZ[k+colsT] = h_G[11+(i*14)];
//Low point for Top Side
h_GLX[k+(2*colsT)] = h_G[12+(i*14)];
h_GLY[k+(2*colsT)] = h_G[4+(i*14)];
h_GLZ[k+(2*colsT)] = h_G[11+(i*14)];
//Low point for South Side
h_GLX[k+(3*colsT)] = h_G[13+(i*14)];
h_GLY[k+(3*colsT)] = h_G[7+(i*14)];
h_GLZ[k+(3*colsT)] = h_G[11+(i*14)];
//Low point for West Side
h_GLX[k+(4*colsT)] = h_G[13+(i*14)];
h_GLY[k+(4*colsT)] = h_G[4+(i*14)];
h_GLZ[k+(4*colsT)] = h_G[8+(i*14)];
//Low point for Bottom Side
h_GLX[k+(5*colsT)] = h_G[13+(i*14)];
h_GLY[k+(5*colsT)] = h_G[4+(i*14)];
h_GLZ[k+(5*colsT)] = h_G[11+(i*14)];
k++;
 g++;
k = g*6*colsT;

//Development of AX, AY, AZ (1 x 6GT) Arrays.
//These arrays are used to store a point on each plane of each geometry.
//Used in the equation to find Thit.
float* h_AX;
float* h_AY;
float* h_AZ;
//Allocate HOST PINNED memory (CPU) for Arrays AX, AY, and AZ (1 x 6GT).
CudaErrCheck( cudaMallocHost((void**)&h_AX, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_AY, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_AZ, nBytesF) );
//For loop used to store each a point on each plane of each geometry.
//Each point on a plane needs to be repeated MaxTrack amount of times to account
//for each track that is to be compared to each plane of each geometry.
k = 0;
g = 0;
for (i = 0; i < colsG; i++)
{
    for (j = 0; j < colsT; j++)
    {
        //Point on plane: North Side.
        h_AX[k] = h_G[12+(i*14)];
        h_AY[k] = h_G[1+(i*14)];
        h_AZ[k] = h_G[2+(i*14)];
        //Point on plane: East Side.
        h_AX[k+colsT] = h_G[12+(i*14)];
        h_AY[k+colsT] = h_G[1+(i*14)];
        h_AZ[k+colsT] = h_G[2+(i*14)];
        //Point on plane: Top Side.
        h_AX[k+(2*colsT)] = h_G[12+(i*14)];
        h_AY[k+(2*colsT)] = h_G[1+(i*14)];
        h_AZ[k+(2*colsT)] = h_G[2+(i*14)];
        //Point on plane: South Side.
        h_AX[k+(3*colsT)] = h_G[13+(i*14)];
        h_AY[k+(3*colsT)] = h_G[7+(i*14)];
        h_AZ[k+(3*colsT)] = h_G[8+(i*14)];
        //Point on plane: West Side.
        h_AX[k+(4*colsT)] = h_G[13+(i*14)];
        h_AY[k+(4*colsT)] = h_G[7+(i*14)];
        h_AZ[k+(4*colsT)] = h_G[8+(i*14)];
        //Point on plane: Bottom Side.
        h_AX[k+(5*colsT)] = h_G[13+(i*14)];
        h_AY[k+(5*colsT)] = h_G[7+(i*14)];
        h_AZ[k+(5*colsT)] = h_G[8+(i*14)];
    }
}
k++;
}
g++;
k = g*6*colsT;

//Host variables used to retrieve date from the four Streams
int TrackID;
int GeometryID;
float* h_IntPX;
float* h_IntPY;
float* h_IntPZ;
float* h_DistInt;
float* h_Vel;
float* h_TimeInt;
//Allocate HOST PINNED memory (CPU) for Array IntPX (1 x 6GT).
CudaErrCheck( cudaMallocHost((void**)&h_IntPX, nBytesF) );
//Allocate HOST PINNED memory (CPU) for Array IntPY (1 x 6GT).
CudaErrCheck( cudaMallocHost((void**)&h_IntPY, nBytesF) );
//Allocate HOST PINNED memory (CPU) for Array IntPZ (1 x 6GT).
CudaErrCheck( cudaMallocHost((void**)&h_IntPZ, nBytesF) );
//Allocate HOST PINNED memory (CPU) for Array DistInt (1 x 6GT).
CudaErrCheck( cudaMallocHost((void**)&h_DistInt, nBytesF) );
//Allocate HOST PINNED memory (CPU) for Array Vel (1 x 6GT).
CudaErrCheck( cudaMallocHost((void**)&h_Vel, nBytesF) );
//Allocate HOST PINNED memory (CPU) for Array TimeInt (1 x 6GT).
CudaErrCheck( cudaMallocHost((void**)&h_TimeInt, nBytesF) );
//For loop to repeat 5 times to in order to get an average execution time.
//This for loop is for testing purposes only.
for (n = 0; n < 5; n++)
{
    printf("Loop: %i\n", n+1);
    //----------------------------------------------------------------------------
    //!!!!!!!!!!!!!!!!!!!STARTING VIOLATION CHECK PROCESS!!!!!!!!!!!!!!!!!!!!!!!
    //Recording of execution time begins now.
    CudaErrCheck( cudaEventRecord(StartEvent,0) );
    //----------------------------------------------------------------------------
    //Creation of Streams.
    cudaStream_t s[nStreams];
    for(i=0; i<nStreams; i++)
        {cudaStreamCreate(&s[i]);

    //----------------------------------------------------------------------------
    //Declaration of device variables used in program.
    float* d_TCX;
    float* d_TCY;
    float* d_DistInt;
    float* d_BX;
    float* d_PY;
    float* d_Vel;
    float* d_AX;
    float* d_AY;
    float* d_NX;
    float* d_NV;
    float* d_NZ;
    float* d_Thit;
    float* d_IntPX;
    float* d_IntPY;
    float* d_IntPZ;
    float* d_TimeInt;
    float* d_GHX;
    float* d_GHY;
    float* d_GHZ;
    float* d_GLX;
    float* d_GLY;
    float* d_GLZ;
    unsigned char* d_ComTemp;
    unsigned char* d_ComTotal;
    int* d_Index;
    //DEVICE MEMORY ALLOCATION
for (i = 0; i < nStreams; i++)
{
    offset = i * StreamSize;
    CudaErrCheck( cudaMemcpyAsync(&d_TCX[offset], &h_TCX[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_TCY[offset], &h_TCY[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_DistInt[offset], &h_TCZ[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_BX[offset], &h_BX[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_BY[offset], &h_BY[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_Vel[offset], &h_BZ[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_AX[offset], &h_AX[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_AY[offset], &h_AY[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_Thit[offset], &h_AZ[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_NX[offset], &h_NX[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_NY[offset], &h_NY[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_NZ[offset], &h_NZ[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_GHX[offset], &h_GHX[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_GHY[offset], &h_GHY[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_GHZ[offset], &h_GHZ[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_GLX[offset], &h_GLX[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_GLY[offset], &h_GLY[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
    CudaErrCheck( cudaMemcpyAsync(&d_GLZ[offset], &h_GLZ[offset], nStreamBytesF, cudaMemcpyHostToDevice, s[i]);
}

for (i = 0; i < nStreams; i++)
{
    offset = i * StreamSize;
    nppSetStream(s[i]);
    //STEP 1: Calculation Ray Direction Vector for Each Track (B=C-P)
// C = Current Point
// P = Previous Point
// Part 1: Calculate C - P
// d_B = d_TC - d_B
nppsSub_32f(&d_BX[offset], &d_TCX[offset], &d_BX[offset], StreamSize);
nppsSub_32f(&d_BY[offset], &d_TCY[offset], &d_BY[offset], StreamSize);
nppsSub_32f(&d_Vel[offset], &d_DistInt[offset], &d_Vel[offset], StreamSize);

//STEP 2: Calculation T_hit (T_hit = (N . (A - C)) / (N . B))
// N = Normal of Plane of Geometry
// A = Point on Plane
// C = Current Point
// B = Result from Step 1
// Part 1: Calculating A - C.
// d_A = d_A - d_TC
nppsSub_32f_I(&d_TCX[offset], &d_AX[offset], StreamSize);
nppsSub_32f_I(&d_TCY[offset], &d_AY[offset], StreamSize);
nppsSub_32f_I(&d_DistInt[offset], &d_Thit[offset], StreamSize);

// d_A = d_A * d_N
// There is no NPP dot product function in this software's current library, so finding the dot product requires a two step process.
// First, the element by element multiplication of the two matrices. Then the summation of each column.
nppsMul_32f_I(&d_NX[offset], &d_AX[offset], StreamSize);
nppsMul_32f_I(&d_NY[offset], &d_AY[offset], StreamSize);
nppsMul_32f_I(&d_NZ[offset], &d_Thit[offset], StreamSize);
// d_A = d_A + d_A
// Addition of each result is equivalent to summation of the columns. The final result (d_Thit) is now the numerator of the T_hit equation (T_hit = N . (A - C)).
nppsAdd_32f_I(&d_AX[offset], &d_AY[offset], StreamSize);
nppsAdd_32f_I(&d_AY[offset], &d_Thit[offset], StreamSize);
// Part 3: Calculate Dot Product of N and B.
// d_N = d_N * d_B
// Same logic as STEP 2, Part 2
nppsMul_32f_I(&d_BX[offset], &d_NX[offset], StreamSize);
nppsMul_32f_I(&d_BY[offset], &d_NY[offset], StreamSize);
nppsMul_32f_I(&d_Vel[offset], &d_NZ[offset], StreamSize);
// d_N = d_N + d_N
// The final result (d_NZ) is now the denominator of the T_hit equation (d_NZ) is now the denominator of the T_hit equation (T_hit = N . (A - C)).
nppsAdd_32f_I(&d_NX[offset], &d_NY[offset], StreamSize);
nppsAdd_32f_I(&d_NY[offset], &d_NZ[offset], StreamSize);
// Part 4: Divide previous results.
// d_Thit = d_Thit / d_N
nppsDiv_32f_I(&d_NZ[offset], &d_Thit[offset], StreamSize);

//STEP 3: Calculation of Intersection Point (I = C + B*Thit)
// C = Current Point
// B = Result from Step 1
// Thit = Result from Step 2
// Part 1: Copy Array B
// d_IntP = d_B
// This is done to preserve Array B for later use in program.
nppsCopy_32f(&d_BX[offset], &d_IntPX[offset], StreamSize);
nppsCopy_32f(&d_BY[offset], &d_IntPY[offset], StreamSize);
nppsCopy_32f(&d_Vel[offset], &d_IntPZ[offset], StreamSize);
// Part 2: Calculate B*Thit
// d_IntP = d_IntP * d_Thit
nppsMul_32f_I(&d_Thit[offset], &d_IntPX[offset], StreamSize);
nppsMul_32f_I(&d_Thit[offset], &d_IntPY[offset], StreamSize);
nppsMul_32f_I(&d_Thit[offset], &d_IntPZ[offset], StreamSize);
// Part 3: Calculate I = C + B*Thit
// d_IntP = d_IntP + d_TC
nppsAdd_32f_I(&d_TCX[offset], &d_IntPX[offset], StreamSize);
nppsAdd_32f_I(&d_TCY[offset], &d_IntPY[offset], StreamSize);
nppsAdd_32f_I(&d_DistInt[offset], &d_IntPZ[offset], StreamSize);

//STEP 4: Calculate velocity (Vel = B / TimeDiff)
// Part 1: Square the distance between
// d_B = d_B^2
nppsSqr_32f_I(&d_BX[offset], StreamSize);
nppsSqr_32f_I(&d_BY[offset], StreamSize);
nppsSqr_32f_I(&d_Vel[offset], StreamSize);
// Part 2: Summation of each column.
//  d_Vel = d_BX + d_BY + d_Vel
nppsAdd_32f_I(&d_BX[offset], &d_BY[offset], StreamSize);
nppsAdd_32f_I(&d_BY[offset], &d_Vel[offset], StreamSize);
// Part 3: Calculate square root of each element.
//  d_Vel = sqrt(d_Vel)
//The result is the distance travel from the previous point to
//current point and will be divided by a predetermined (defined)
//arbitrary time.
nppsSqrt_32f_I(&d_Vel[offset], StreamSize);
// Part 4: Calculate velocity.
//  d_Vel = d_Vel / TimeDiff
nppsDivC_32f_I(TimeDiff, &d_Vel[offset], StreamSize);

//STEP 5: distance from current point to intersecting point.
//  DistInt = sqrt(CoorDist^2 * CoorDistY^2 * CoorDistZ^2)
// Part 1: Calculation of difference for each coordinate
//  d_TC = d_IntP - d_TC
nppsSub_32f(&d_TCX[offset], &d_IntPX[offset], &d_TCX[offset], StreamSize);
nppsSub_32f(&d_TCY[offset], &d_IntPY[offset], &d_TCY[offset], StreamSize);
nppsSub_32f(&d_DistInt[offset], &d_IntPZ[offset], &d_DistInt[offset], StreamSize);
// Part 2: Square each coordinate difference
//  d_TC = d_TC^2
nppsSqr_32f_I(&d_TCX[offset], StreamSize);
nppsSqr_32f_I(&d_TCY[offset], StreamSize);
nppsSqr_32f_I(&d_DistInt[offset], StreamSize);
// Part 3: Summation of each squared coordinate.
//  d_DistInt = d_TCX + d_TCY + d_DistInt
nppsAdd_32f_I(&d_TCX[offset], &d_TCY[offset], StreamSize);
nppsAdd_32f_I(&d_TCY[offset], &d_DistInt[offset], StreamSize);
// Part 4: Calculate square root of each element.
//  d_DistInt = sqrt(d_DistInt)
//The final result will be the distance from the current point to
//the intersection point.
nppsSqrt_32f_I(&d_DistInt[offset], StreamSize);
//STEP 6: Calculate time until intersection
// Part 1: Divide distance from intersection point
//  d_TimeInt = d_DistInt / d_Vel
nppsDiv_32f(&d_Vel[offset], &d_DistInt[offset], &d_TimeInt[offset], StreamSize);
//STEP 7: Finding violation
//  d_IntP = d_IntP - 0.00001
//Since the variable IntP was calculated and not predetermined,
//and that each element of IntP is limited to 32 bits, there may
//be a loss of precision. In order to pass the geometry border
//checks these precision errors have to be accounted for. This is
//done by creating a tolerance for the IntP. This first equations
//sets up the lower tolerance of IntP.
nppsSubc_32f_I(0.0001, &d_IntPX[offset], StreamSize);
nppsSubc_32f_I(0.0001, &d_IntPY[offset], StreamSize);
nppsSubc_32f_I(0.0001, &d_IntPZ[offset], StreamSize);
// Part 2: Compares (d_IntP) with the UPPER border points
//  If d_IntP > d_GH, then d_ComTemp = 255 (TRUE).
//  If d_IntP <= d_GH, then d_ComTemp = 0 (FALSE).
//For this program, an element by element comparison result of 0
//means the element passed the violation check. These results are
//added together to form a single array. If a zero is within the
//final array, this means the corresponding element is in
//violation with the geometry. So in this part, a zero means the
//intersection point coordinates are within the upper bound of
//the side (plane) of a geometry.
nppicompare_32f_c1r(&d_IntPX[offset], colsT.sizeof(float),
&d_GHX[offset], colsT.sizeof(float), &d_ComTemp[offset],
colsT.sizeof(unsigned char), sz, NPP_CMP_GREATER);
nppicompare_32f_c1r(&d_IntPY[offset], colsT.sizeof(float),
&d_GHY[offset], colsT.sizeof(float), &d_ComTotal[offset],
colsT.sizeof(unsigned char), sz, NPP_CMP_GREATER);
nppsAdd_8u_ISfs(&d_ComTemp[offset], &d_ComTotal[offset], StreamSize, 0);
nppiCompare_32f_C1R(&d_IntPZ[offset], colsT*sizeof(float),&d_GLZ[offset], colsT*sizeof(float), &d_ComTemp[offset], colsT*sizeof(unsigned char), sz, NPP_CMP_GREATER);
nppsAdd_8u_ISfs(&d_ComTemp[offset], &d_ComTotal[offset], StreamSize, 0);

// d_IntP = d_IntP + 0.00002
// The same logic as STEP 7, Part 1. But this time dealing with
// lower ends of the plane of the geometry. The value 0.00002 is
// added to account for the 0.00001 that was subtracted in a
// previous equation.
nppsAddC_32f_I(0.00002, &d_IntPX[offset], StreamSize);
nppsAddC_32f_I(0.00002, &d_IntPY[offset], StreamSize);
nppsAddC_32f_I(0.00002, &d_IntPZ[offset], StreamSize);

// Part 4: Compares (d_IntP) with the LOWER border points
// If d_IntP < d_GL, then d_ComTemp = 255 (TRUE).
// If d_IntP >= d_GH, then d_ComTemp = 0 (FALSE).
// After this part, if there is still a zero in the array
// d_ComTotal, then the intersection point is within the bounds of
// the plane representing the side of a geometry. More
// comparisons are needed to determine if the intersection point
// is in front of the plane or not and if the track is within the
// buffer zone.
nppiCompare_32f_C1R(&d_IntPX[offset], colsT*sizeof(float),&d_GLX[offset], colsT*sizeof(float), &d_ComTemp[offset], colsT*sizeof(unsigned char), sz, NPP_CMP_LESS);
nppsAdd_8u_ISfs(&d_ComTemp[offset], &d_ComTotal[offset], StreamSize, 0);
nppiCompare_32f_C1R(&d_IntPY[offset], colsT*sizeof(float),&d_GLY[offset], colsT*sizeof(float), &d_ComTemp[offset], colsT*sizeof(unsigned char), sz, NPP_CMP_LESS);
nppsAdd_8u_ISfs(&d_ComTemp[offset], &d_ComTotal[offset], StreamSize, 0);
nppiCompare_32f_C1R(&d_IntPZ[offset], colsT*sizeof(float),&d_GLZ[offset], colsT*sizeof(float), &d_ComTemp[offset], colsT*sizeof(unsigned char), sz, NPP_CMP_LESS);
nppsAdd_8u_ISfs(&d_ComTemp[offset], &d_ComTotal[offset], StreamSize, 0);

// Part 5: Checking to see if Thit is positive
// If d_Thit2 < 0, then d_ComTemp = 255
// If d_Thit2 >= 0, then d_ComTemp = 0
// From this comparison we can tell if Thit is positive or
// negative. It an element in Thit is negative (d_ComTemp = 255)
// then the intersection point occurred behind the track which
// means nothing. But if the element in Thit is positive
// (d_ComTemp = 0) then the intersection point is occurring in
// front of the track which means the track is heading towards the
// geometry.
nppiCompareC_32f_C1R(&d_Thit[offset], colsT*sizeof(float), 0,&d_ComTemp[offset], colsT*sizeof(unsigned char), sz, NPP_CMP_LESS);
nppsAdd_8u_ISfs(&d_ComTemp[offset], &d_ComTotal[offset], StreamSize, 0);

// Part 6: Compare buffer time.
// If d_TimeInt > Buffer, then d_ComTemp = 255
// If d_TimeInt <= Buffer, then d_ComTemp = 0
// If the result is zero this means the time until intersection
// is within the buffer zone, thus an immediate violation has
// occurred. Assuming the intersection point is within the bounds
// of the plane of the geometry. If the array d_ComTotal contains
// a 0 then a violation has occurred and its corresponding
// information will be recorded.
nppiCompareC_32f_C1R(&d_TimeInt[offset], colsT*sizeof(float), Buffer, &d_ComTemp[offset], colsT*sizeof(unsigned char), sz, NPP_CMP_GREATER);
nppsAdd_8u_ISfs(&d_ComTemp[offset], &d_ComTotal[offset],
StreamSize, 0);
}

// DEVICE TO HOST DATA TRANSFER
for (i = 0; i < nStreams; i++) {
    offset = i * StreamSize;
    CudaErrCheck( cudaMemcpyAsync(&h_IntPX[offset], &d_IntPX[offset],
nStreamBytesF, cudaMemcpyDeviceToHost, s[i] ) );
    CudaErrCheck( cudaMemcpyAsync(&h_IntPY[offset], &d_IntPY[offset],
nStreamBytesF, cudaMemcpyDeviceToHost, s[i] ) );
    CudaErrCheck( cudaMemcpyAsync(&h_IntPZ[offset], &d_IntPZ[offset],
nStreamBytesF, cudaMemcpyDeviceToHost, s[i] ) );
    CudaErrCheck( cudaMemcpyAsync(&h_DistInt[offset],
&d_DistInt[offset], nStreamBytesF, cudaMemcpyDeviceToHost,
s[i] ) );
    CudaErrCheck( cudaMemcpyAsync(&h_Vel[offset], &d_Vel[offset],
nStreamBytesF, cudaMemcpyDeviceToHost, s[i] ) );
    CudaErrCheck( cudaMemcpyAsync(&h_TimeInt[offset],
&d_TimeInt[offset], nStreamBytesF, cudaMemcpyDeviceToHost,
s[i] ) );
}

// Creates an array used to track indices.
// The purpose of the Index variable is to represent the Indices
// of the final array of comparison results. If we know which
// elements in ComTotal equal '0', then we can find out which
// tracks and geometries are in violation of each other. Once
// the array d_Index is created, the program counts how many
// elements in ComTotal equal '0'. This results tells us the size
// in which to make the final index array, IndexFinal, which will
// be used to store the indices of the elements that are equal
// to '0' in ComTotal.
thrust::device_ptr<int> dev_ptr_Index(d_Index);
thrust::device_ptr<unsigned char> dev_ptr_ComTotal(d_ComTotal);
thrust::sequence(dev_ptr_Index, dev_ptr_Index + nElements);
result = thrust::count_if(dev_ptr_ComTotal, dev_ptr_ComTotal + nElements,
equal_to_zero());
//---------------------------------------------------------------------//
// STEP 8: Output Violations Results
if (result != 0) {
    int* h_IndexFinal;
    int* d_IndexFinal;
    // HOST MEMORY ALLOCATION
    h_IndexFinal = (int*)malloc(result*sizeof(int));
    if (h_IndexFinal == 0) {
        fprintf (stderr, "!!!! host memory allocation error
(h_IndexFinal)\n");
        return EXIT_FAILURE;
    }
    // DEVICE MEMORY ALLOCATION
    CudaErrCheck( cudaMalloc((void**)&d_IndexFinal,
result*sizeof(int)) );
    // CALCULATIONS
    thrust::device_ptr<int> dev_ptr_IndexFinal(d_IndexFinal);
    // This is the function that will remove all elements in Index
    // that correspond to values in ComTotal that are not equal to '0'
    // and place what is left into IndexFinal.
    thrust::remove_copy_if(dev_ptr_Index, dev_ptr_Index + nElements,
dev_ptr_DisIndexFinal, dev_ptr_IndexFinal, not_equal_to_zero());
    // DEVICE TO HOST DATA TRANSFER
    CudaErrCheck( cudaMemcpy(h_IndexFinal, d_IndexFinal,
result*sizeof(int), cudaMemcpyDeviceToHost) );
    // Prints out Violations based on the values in d_IndexFinal.
    for (i=0; i<result; i++) {
        if ((h_IndexFinal[i]+1)%colsT==0)
            TrackID = colsT;
    } else
{  
    TrackID = (h_IndexFinal[i]+1)%colsT;  
}

GeometryID = (h_IndexFinal[i]/(6*colsT))+1;
//printf("Track ID: %i\n", TrackID);
//printf("Geometry ID: %i\n", GeometryID);
fprintf(ParallelViolations, "Track ID: %i\n", TrackID);
fprintf(ParallelViolations, "Geometry ID: %i\n", GeometryID);
fprintf(ParallelViolations, "Distance: %.6f\n", 
    h_DistInt[h_IndexFinal[i]]);
fprintf(ParallelViolations, "Velocity: %.6f\n", 
    h_Vel[h_IndexFinal[i]]);
fprintf(ParallelViolations, "Intersect Time: %.6f\n", 
    h_TimeInt[h_IndexFinal[i]]);
fprintf(ParallelViolations, "Intersect Point: 
    (%.6f, %.6f, %.6f)\n", h_IntPX[h_IndexFinal[i]], 
    h_IntPY[h_IndexFinal[i]], h_IntPZ[h_IndexFinal[i]]);
}

//MEMORY CLEAN UP IndexFinal.
free(h_IndexFinal);
CudaErrCheck( cudaFree(d_IndexFinal) );

//Destruction of Streams.
for (i = 0; i < nStreams; i++)
{
    CudaErrCheck( cudaStreamDestroy(s[i]) );
}

/installation process
CudaErrCheck( cudaEventRecord(StopEvent, 0) );
CudaErrCheck( cudaEventSynchronize(StopEvent) );
CudaErrCheck( cudaEventElapsedTime(&msec, StartEvent, StopEvent) );
AveTime = AveTime + msec;
if (n == 4)
{
    AveTime = AveTime/5;
    fprintf(ParallelExecutionTimes, "%i %i %.3f\n", colsT, colsG, 
        msec);
    colsT = colsT + StepT;
    colsG = colsG + StepG;
    AveTime = 0;
}

//Freeing up leftover allocated memory for DEVICE (GPU)
//MEMORY CLEAN UP IntPX, IntPY, IntPZ.
CudaErrCheck( cudaFree(d_IntPX) );
CudaErrCheck( cudaFree(d_IntPY) );
CudaErrCheck( cudaFree(d_IntPZ) );
//MEMORY CLEAN UP TimeInt.
CudaErrCheck( cudaFree(d_TimeInt) );
//MEMORY CLEAN UP Index.
CudaErrCheck( cudaFree(d_Index) );
//MEMORY CLEAN UP ComTotal.
CudaErrCheck( cudaFree(d_ComTotal) );
//MEMORY CLEAN UP AX, AY, AZ.
CudaErrCheck( cudaFree(d_AX) );
CudaErrCheck( cudaFree(d_AY) );
CudaErrCheck( cudaFree(d_AZ) );
//MEMORY CLEAN UP NX, NY, NZ.
CudaErrCheck( cudaFree(d_NX) );
CudaErrCheck( cudaFree(d_NY) );
CudaErrCheck( cudaFree(d_NZ) );
//MEMORY CLEAN UP BX, BY, Vel.
CudaErrCheck( cudaFree(d_BX) );
CudaErrCheck( cudaFree(d_BY) );
CudaErrCheck( cudaFree(d_Vel) );
//MEMORY CLEAN UP TCX, TCY, TCZ.
CudaErrCheck( cudaFree(d_TCX) );
CudaErrCheck( cudaFree(d_TCY) );
CudaErrCheck( cudaFree(d_DistInt) );
// MEMORY CLEAN UP GHX, GHY, GHZ.
CudaErrCheck(cudaFree(d_GHX));
CudaErrCheck(cudaFree(d_GHY));
CudaErrCheck(cudaFree(d_GHZ));
// MEMORY CLEAN UP GLX, GLY, GLZ.
CudaErrCheck(cudaFree(d_GLX));
CudaErrCheck(cudaFree(d_GLY));
CudaErrCheck(cudaFree(d_GLZ));
// MEMORY CLEAN UP Thit.
CudaErrCheck(cudaFree(d_Thit));
// MEMORY CLEAN UP ComTemp.
CudaErrCheck(cudaFree(d_ComTemp));

// The closing of files used.
fclose(ParallelViolations);
// fclose(ParallelExecutionTimes);
ParallelViolations = fopen("Parallel_Cuda_SingleGPU_Breadth_Violations.txt", "w");
unuE-------------------------//
// Freeing up leftover allocated memory for HOST (CPU)
unuE-------------------------//
// MEMORY CLEAN UP TP.
free(h_TP);
// MEMORY CLEAN UP TC.
free(h_TC);
// MEMORY CLEAN UP TCX, TCY, TCZ.
cudaFreeHost(h_TCX);
cudaFreeHost(h_TCY);
cudaFreeHost(h_TCZ);
// MEMORY CLEAN UP BX, BY, BZ.
cudaFreeHost(h_BX);
cudaFreeHost(h_BY);
cudaFreeHost(h_BZ);
// MEMORY CLEAN UP G.
free(h_G);
// MEMORY CLEAN UP NX, NY, NZ.
cudaFreeHost(h_NX);
cudaFreeHost(h_NY);
cudaFreeHost(h_NZ);
// MEMORY CLEAN UP GHX, GHY, GHZ.
cudaFreeHost(h_GHX);
cudaFreeHost(h_GHY);
cudaFreeHost(h_GHZ);
// MEMORY CLEAN UP GLX, GLY, GLZ.
cudaFreeHost(h_GLX);
cudaFreeHost(h_GLY);
cudaFreeHost(h_GLZ);
// MEMORY CLEAN UP AX, AY, AZ.
cudaFreeHost(h_AX);
cudaFreeHost(h_AY);
cudaFreeHost(h_AZ);
// MEMORY CLEAN UP IntPX, IntPY, IntPZ.
cudaFreeHost(h_IntPX);
cudaFreeHost(h_IntPY);
cudaFreeHost(h_IntPZ);
// MEMORY CLEAN UP DistInt.
cudaFreeHost(h_DistInt);
// MEMORY CLEAN UP Vel.
cudaFreeHost(h_Vel);
// MEMORY CLEAN UP UptimeInt.
cudaFreeHost(h_TimeInt);
unuE-------------------------//
// END PROGRAM
return EXIT_SUCCESS;
unuE-------------------------//
D.4  Asynchronous Single GPU Parallel Version Raw Data

500  50  23.543
1000 100  24.850
1500 150  34.269
2000 200  54.168
2500 250  79.225
3000 300  110.556
3500 350  147.531
4000 400  186.018
4500 450  237.356
5000 500  293.066
5500 550  351.483
6000 600  417.242
6500 650  489.211
7000 700  560.575
7500 750  643.964
8000 800  751.731
8500 850  828.884

D.5  Asynchronous Multiple GPU Parallel Version Program

---
//--- Parallel_Cuda_MultiGPU_Breadth.cu
// Nathan Clem
// Compile and Execute (Make sure in correct directory):
// nvcc -I/usr/local/cuda-5.5/include -L/usr/local/cuda-5.5/lib64
// Parallel_Cuda_MultiGPU_Breadth.cu -lnppc -lnppi -lnpps -lm
// ./a.out
//---
#include <thrust/device_ptr.h>
#include <thrust/sequence.h>
#include <thrust/remove.h>
#include <thrust/count.h>
#include <npps.h>
#include <nppi.h>
#include <nppcore.h>
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <csignal>

//--- Defined variables used for TRACK data.
#define MaxTracks 500 // Sets number of total Tracks.
#define StepT 500 // Sets the number of tracks to add on after every loop;
//--- Defined variables used for GEOMETRY data.
#define MaxGeometries 50 // Max number of Geometries to create.
#define StepG 50 // Sets the number of the geometries to add on after every loop.
#define Length 0.05*PointBound // Length tolerance (along the z-axis).
#define Width 0.05*PointBound // Width tolerance (along the y-axis).
#define High 0.05*PointBound // High tolerance (along the x-axis).
#define Low 0.05*PointBound // Low tolerance (along the x-axis).
//--- Random number parameters.
#define PointBound 500 // Sets set space limit.
#define MaxDist 5 // Sets limit of the initial distance to next point.
#define TimeDiff 2 // Arbitrary time used to describe the difference in time from when
// the current point was collected and when the previous point was collected.
#define Buffer 180 // Arbitrary time used to describe the time limit of a potential violation
// of a geometry by a track. If the time until intersection is less than this
// value, then an imminent violation has occurred. Assuming the intersection
// point is within the bounds of the geometry
#define Buffer 180
//---
// Text files used to store results.
FILE *ParallelViolations = fopen("Parallel_Cuda_MultiGPU_Breadth_Violations.txt", "w"); // Stores violation results
FILE *ParallelExecutionTimes = fopen("Parallel_Cuda_MultiGPU_Breadth_Execution_Times_Ave.txt", "w"); // Stores program runtimes

// Structure predicate used in Thrust functions to find values equal to zero.
struct equal_to_zero
{
   __host__ __device__
   bool operator()(const float x)
   {
      return x == 0;
   }
};

// Structure predicate used in Thrust functions to find values not equal to zero.
struct not_equal_to_zero
{
   __host__ __device__
   bool operator()(const float x)
   {
      return x != 0;
   }
};

// Use to stop program in case of infinite loop
// Enter: "CTRL + C"
void signalHandler( int signum )
{
   printf("Interrupt signal (%i) received.\n", signum);
   fclose(ParallelExecutionTimes);
   fclose(ParallelViolations);
   exit(signum);
}

// Cuda Error Check
#define CudaErrCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort=true)
{
   if (code != cudaSuccess)
   {
      fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
      if (abort) exit(code);
   }
}

// Main
int main(int argc, char** argv)
{
   int colsT = MaxTracks;
   int colsG = MaxGeometries;
   signal(SIGINT, signalHandler);
   int i, j, g, k, n;
   float AveTime = 0;
   int result0, result1;
   int nElements;
   int nElementsDevice;
   int nStreams;
   int StreamSize;
   int nBytesI;
   int nBytesF;
   int nStreamBytesF;
   int nBytesUC;
   int offset;
   int LineStepF;
   int LineStepUC;
   while(1)
   {
      srand(1);
      printf("\nTracks: %i\n", colsT);
      printf("Geometries: %i\n", colsG);
      float msec;
cudaEvent_t StartEvent, StopEvent;
cudaSetDevice(0);
CudaErrCheck( cudaEventCreate(&StartEvent) );
CudaErrCheck( cudaEventCreate(&StopEvent) );
// Initializing Array Sizes and Number of Streams.
nElements = 6*colsG*colsT;
nElementsDevice = nElements/2;
nBytesI = nElementsDevice*sizeof(int);
nBytesF = nElementsDevice*sizeof(float);
nBytesUC = nElementsDevice*sizeof(unsigned char);
nStreams = 4;
StreamSize = nElementsDevice/nStreams;
nStreamBytesF = StreamSize*sizeof(float);
NppiSize sz;
sz.height = 6*colsG/nStreams;
sz.width = colsT/2;
LineStepF = (colsT/2)*sizeof(float);
LineStepUC = (colsT/2)*sizeof(unsigned char);
//-----------------------------------------------------------------------------//
// Development of TRACK (3 x T) matrix.
float* h_TP;
float* h_TC;
float Dist;
// Allocate HOST memory (CPU) for Matrix TP (Previous Track Coordinates).
h_TP = (float*)malloc(3*colsT*sizeof(float));
if (h_TP == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_TP)\n");
    return EXIT_FAILURE;
}
// Allocate HOST memory (CPU) for Matrix TC (Current Track Coordinates).
h_TC = (float*)malloc(3*colsT*sizeof(float));
if (h_TC == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_TC)\n");
    return EXIT_FAILURE;
}
// For loop used to store created previous and current points.
for (i = 0; i < (3*colsT); i++)
{
    // Creates the initial location of a Track.
    h_TP[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    // Calculates distance to next point for each coordinate.
    Dist = 2*MaxDist*((float)rand()/RAND_MAX)-MaxDist;
    // Creates the current location of a Track.
    h_TC[i] = h_TP[i]+Dist;
}
//-----------------------------------------------------------------------------//
// Development of TCX, TCY, TCZ, BX, BY, and BZ (1 x 6GT)
float* h_TCX;
float* h_TCY;
float* h_TCZ;
float* h_BX;
float* h_BY;
float* h_BZ;
// Allocate HOST memory (CPU) for Array TCX (1 x 6GT).
h_TCX = (float*)malloc(nElements*sizeof(float));
if (h_TCX == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_TCX)\n");
    return EXIT_FAILURE;
}
// Allocate HOST memory (CPU) for Array TCY (1 x 6GT).
h_TCY = (float*)malloc(nElements*sizeof(float));
if (h_TCY == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_TCY)\n");
    return EXIT_FAILURE;
}
// Allocate HOST memory (CPU) for Array TCZ (1 x 6GT).
h_TCZ = (float*)malloc(nElements*sizeof(float));
if (h_TCZ == 0)
{    fprintf (stderr, "!!!! host memory allocation error (h_TCZ)\n");
    return EXIT_FAILURE;
}

//Allocate HOST memory (CPU) for Array BX (1 x 6GT).
    h_BX = (float*)malloc(nElements*sizeof(float));
if (h_BX == 0)    {    fprintf (stderr, "!!!! host memory allocation error (h_BX)\n");
    return EXIT_FAILURE;
    }

//Allocate HOST memory (CPU) for Array BY (1 x 6GT).
    h_BY = (float*)malloc(nElements*sizeof(float));
if (h_BY == 0)    {    fprintf (stderr, "!!!! host memory allocation error (h_BY)\n");
    return EXIT_FAILURE;
    }

//Allocate HOST memory (CPU) for Array BZ (1 x 6GT).
    h_BZ = (float*)malloc(nElements*sizeof(float));
if (h_BZ == 0)    {    fprintf (stderr, "!!!! host memory allocation error (h_BZ)\n");
    return EXIT_FAILURE;
    }

//For loop used to store repeated current and repeated previous points.
k = 0;
for (i = 0; i < (6*colsG); i++)
    {    for (j = 0; j < (3*colsT); j=j+3)
    {
        h_TCX[k] = h_TC[j];
        h_TCY[k] = h_TC[j+1];
        h_TCZ[k] = h_TC[j+2];
        h_BX[k] = h_TP[j];
        h_BY[k] = h_TP[j+1];
        h_BZ[k] = h_TP[j+2];
        k = k + 1;
    }
    }

//MEMORY CLEAN UP TP.
free(h_TP);
//MEMORY CLEAN UP TC.
free(h_TC);

//-----------------------------

//Splitting of Data to Send to Separate GPUs
float* h_TCX0;
float* h_TCY0;
float* h_TCZ0;
float* h_TCX1;
float* h_TCY1;
float* h_TCZ1;

//Allocate HOST PINNED memory (CPU) for Arrays TCX0, TCY0, and TCZ0 (1 x 3GT).
cudaSetDevice(0);
CudaErrCheck( cudaMallocHost((void**)&h_TCX0, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_TCY0, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_TCZ0, nBytesF) );

//Allocate HOST PINNED memory (CPU) for Arrays TCX1, TCY1, and TCZ1 (1 x 3GT).
cudaSetDevice(1);
CudaErrCheck( cudaMallocHost((void**)&h_TCX1, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_TCY1, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_TCZ1, nBytesF) );
for (i = 0; i < nElementsDevice; i++)
    {    h_TCX0[i] = h_TCX[i];
    h_TCY0[i] = h_TCY[i];
    h_TCZ0[i] = h_TCZ[i];
    h_TCX1[i] = h_TCX[i+nElementsDevice];
    h_TCY1[i] = h_TCY[i+nElementsDevice];
    h_TCZ1[i] = h_TCZ[i+nElementsDevice];
    }
//MEMORY CLEAN UP TCX, TCY, TCZ.
free(h_TCX);
free(h_TCY);
free(h_TCZ);
float* h_BX0;
float* h_BY0;
float* h_BZ0;
float* h_BX1;
float* h_BY1;
float* h_BZ1;
//Allocate HOST PINNED memory (CPU) for Arrays BX0, BY0, and BZ0 (1 x 3GT).
cudaSetDevice(0);
CudaErrCheck( cudaMallocHost((void**)&h_BX0, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_BY0, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_BZ0, nBytesF) );
//Allocate HOST PINNED memory (CPU) for Arrays BX1, BY1, and BZ1 (1 x 3GT).
cudaSetDevice(1);
CudaErrCheck( cudaMallocHost((void**)&h_BX1, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_BY1, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_BZ1, nBytesF) );
for (i = 0; i < nElementsDevice; i++)
{
    h_BX0[i] = h_BX[i];
    h_BY0[i] = h_BY[i];
    h_BZ0[i] = h_BZ[i];
    //
    h_BX1[i] = h_BX[i+nElementsDevice];
    h_BY1[i] = h_BY[i+nElementsDevice];
    h_BZ1[i] = h_BZ[i+nElementsDevice];
}
//MEMORY CLEAN UP BX, BY, BZ.
free(h_BX);
free(h_BY);
free(h_BZ);
//-----------------------------------------------------------------------------//
//Development of GEOMETRY (14 x G) matrix and NX, NY, NZ (1 x 6GT) array.
The normals of each plane of each geometry is need to find the
.intersection point between the plane and a track.
float* h_G;
float* h_NX;
float* h_NY;
float* h_NZ;
float y2, z2, H;
//Allocate HOST memory (CPU) for Matrix G (Geometry Coordinates).
h_G = (float*)malloc(14*colsG*sizeof(float));
if (h_G == 0)
{
    fprintf (stderr, "!!!!! host memory allocation error (h_G)\n");
    return EXIT_FAILURE;
}
//Allocate HOST memory (CPU) for Array NX (Geometry Normals X).
h_NX = (float*)malloc(nElements*sizeof(float));
if (h_NX == 0)
{
    fprintf (stderr, "!!!!! host memory allocation error (h_NX)\n");
    return EXIT_FAILURE;
}
//Allocate HOST memory (CPU) for Array NY (Geometry Normals Y).
h_NY = (float*)malloc(nElements*sizeof(float));
if (h_NY == 0)
{
    fprintf (stderr, "!!!!! host memory allocation error (h_NY)\n");
    return EXIT_FAILURE;
}
//Allocate HOST memory (CPU) for Array NZ (Geometry Normals Z).
h_NZ = (float*)malloc(nElements*sizeof(float));
if (h_NZ == 0)
{
    fprintf (stderr, "!!!!! host memory allocation error (h_NZ)\n");
    return EXIT_FAILURE;
}
// Nested for loop used to create and store geometry parameters and corresponding
// normals. Each normal of a plane needs to be repeated MaxTrack amount of times
// to account for each track that is to be compared to each plane of each geometry.

k = 0;
g = 0;
for (i = 0; i < (14*colsG); i = i+14)
{
    // Data storage for Geometry coordinates.
    // h_G[i] to h_G[i+2] are the NE coordinates of Geometry.
    h_G[i] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_G[i+1] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    h_G[i+2] = 2*PointBound*((float)rand()/RAND_MAX)-PointBound;
    y2 = h_G[i+1]-((Width*((float)rand()/RAND_MAX)))+1; // (Length*((float)rand()/RAND_MAX))+1);
    // h_G[i+3] to h_G[i+5] are the NW coordinates of Geometry.
    h_G[i+3] = h_G[i];
    h_G[i+4] = y2;
    h_G[i+5] = h_G[i+2];
    // h_G[i+6] to h_G[i+8] are the SW coordinates of Geometry.
    h_G[i+6] = h_G[i];
    h_G[i+7] = y2;
    h_G[i+8] = z2;
    // h_G[i+9] to h_G[i+11] are the SE coordinates of Geometry.
    h_G[i+9] = h_G[i];
    h_G[i+10] = h_G[i+1];
    h_G[i+11] = z2;
    H = (High*((float)rand()/RAND_MAX))+1;
    h_G[i+12] = h_G[i]+H;
    h_G[i+13] = h_G[i]-((Low*((float)rand()/RAND_MAX))+1);
    for (j = 0; j < colsT; j++)
    {
        // Data storage for Geometry normals.
        // Normal vector for NORTH SIDE of Geometry.
        h_NX[k] = 0;
        h_NY[k] = 0;
        h_NZ[k] = -H*(y2-h_G[i+1]);
        // Normal vector for EAST SIDE of Geometry.
        h_NX[k+colsT] = 0;
        h_NY[k+colsT] = H*(h_G[i+2]-z2);
        h_NZ[k+colsT] = 0;
        // Normal vector for TOP SIDE of Geometry.
        h_NX[k+(2*colsT)] = (y2-h_G[i+1])*(z2-h_G[i+2]);
        h_NY[k+(2*colsT)] = 0;
        h_NZ[k+(2*colsT)] = 0;
        // Normal vector for SOUTH SIDE of Geometry.
        h_NX[k+(3*colsT)] = 0;
        h_NY[k+(3*colsT)] = 0;
        h_NZ[k+(3*colsT)] = -H*(h_G[i+1]-y2);
        // Normal vector for WEST SIDE of Geometry.
        h_NX[k+(4*colsT)] = 0;
        h_NY[k+(4*colsT)] = H*(z2-h_G[i+2]);
        h_NZ[k+(4*colsT)] = 0;
        // Normal vector for BOTTOM SIDE of Geometry.
        h_NX[k+(5*colsT)] = -(h_G[i+2]-z2)*(h_G[i+1]-y2);
        h_NY[k+(5*colsT)] = 0;
        h_NZ[k+(5*colsT)] = 0;
        k = k++;
    }
    g++;
    k = g*6*colsT;
}

//-----------------------------------------------------------------------------//
// Splitting of Data to Send to Separate GPUs

float* h_NX0;
float* h_NY0;
float* h_NZ0;
float* h_NX1;
float* h_NY1;
float* h_NZ1;
// Allocate HOST PINNED memory (CPU) for Arrays NX0, NY0, and NZ0 (1 x 3GT).
cudaSetDevice(0);
CudaErrCheck( cudaMemcpy((void**)&h_NX0, nBytesF );
CudaErrCheck( cudaMemcpy((void**)&h_NY0, nBytesF );
CudaErrCheck( cudaMemcpy((void**)&h_NZ0, nBytesF );
//Allocate HOST PINNED memory (CPU) for Arrays NX1, NY1, and NZ1 (1 x 3GT).
cudaSetDevice(1);
CudaErrCheck( cudaMemcpy((void**)&h_NX1, nBytesF );
CudaErrCheck( cudaMemcpy((void**)&h_NY1, nBytesF );
CudaErrCheck( cudaMemcpy((void**)&h_NZ1, nBytesF );
for (i = 0; i < nElementsDevice; i++)
    {
        h_NX0[i] = h_NX[i];
        h_NY0[i] = h_NY[i];
        h_NZ0[i] = h_NZ[i];
        //
        h_NX1[i] = h_NX[i+nElementsDevice];
        h_NY1[i] = h_NY[i+nElementsDevice];
        h_NZ1[i] = h_NZ[i+nElementsDevice];
    }
    //MEMORY CLEAN UP NX, NY, NZ.
    free(h_NX);
    free(h_NY);
    free(h_NZ);
    //-------------------------------------------------------------------------------------
    //Development of GHX, GHY, GHZ (1 x 6GT) and GLX, GLY, GLZ (1 x 6GT) Array
    //Arrays GH is used to store the high end of each plane of each geometry.
    //The high ends of the plane will be used to compare with the intersection point.
    //Arrays GL is used to store the low end of each plane of each geometry.
    //The lows ends of the plane will be used to compare with the intersection point.
    //IF the intersection point is BELOW GH AND ABOVE GL, then the intersection point
    //is WITHIN THE BORDERS of the plane of the geometry.
    float* h_GHX;
    float* h_GHY;
    float* h_GHZ;
    float* h_GLX;
    float* h_GLY;
    float* h_GLZ;
    //Allocate HOST memory (CPU) for Array GHX (1 x 6GT).
    h_GHX = (float*)malloc(nElements*sizeof(float));
    if (h_GHX == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GHX)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GHY (1 x 6GT).
    h_GHY = (float*)malloc(nElements*sizeof(float));
    if (h_GHY == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GHY)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GHZ (1 x 6GT).
    h_GHZ = (float*)malloc(nElements*sizeof(float));
    if (h_GHZ == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GHZ)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GLX (1 x 6GT).
    h_GLX = (float*)malloc(nElements*sizeof(float));
    if (h_GLX == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GLX)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GLY (1 x 6GT).
    h_GLY = (float*)malloc(nElements*sizeof(float));
    if (h_GLY == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GLY)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GLZ (1 x 6GT).
    h_GLZ = (float*)malloc(nElements*sizeof(float));
    if (h_GLZ == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GLZ)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GHX (1 x 6GT).
    h_GHX = (float*)malloc(nElements*sizeof(float));
    if (h_GHX == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GHX)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GHY (1 x 6GT).
    h_GHY = (float*)malloc(nElements*sizeof(float));
    if (h_GHY == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GHY)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GHZ (1 x 6GT).
    h_GHZ = (float*)malloc(nElements*sizeof(float));
    if (h_GHZ == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GHZ)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GLX (1 x 6GT).
    h_GLX = (float*)malloc(nElements*sizeof(float));
    if (h_GLX == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GLX)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GLY (1 x 6GT).
    h_GLY = (float*)malloc(nElements*sizeof(float));
    if (h_GLY == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GLY)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GLZ (1 x 6GT).
    h_GLZ = (float*)malloc(nElements*sizeof(float));
    if (h_GLZ == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GLZ)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GHX (1 x 6GT).
    h_GHX = (float*)malloc(nElements*sizeof(float));
    if (h_GHX == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GHX)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GHY (1 x 6GT).
    h_GHY = (float*)malloc(nElements*sizeof(float));
    if (h_GHY == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GHY)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GHZ (1 x 6GT).
    h_GHZ = (float*)malloc(nElements*sizeof(float));
    if (h_GHZ == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GHZ)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GLX (1 x 6GT).
    h_GLX = (float*)malloc(nElements*sizeof(float));
    if (h_GLX == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GLX)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GLY (1 x 6GT).
    h_GLY = (float*)malloc(nElements*sizeof(float));
    if (h_GLY == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GLY)\n");
            return EXIT_FAILURE;
        }
    //Allocate HOST memory (CPU) for Array GLZ (1 x 6GT).
    h_GLZ = (float*)malloc(nElements*sizeof(float));
    if (h_GLZ == 0)
        {
            fprintf (stderr, "!!!! host memory allocation error (h_GLZ)\n");
            return EXIT_FAILURE;
//Allocate HOST memory (CPU) for Array GLZ (1 x 6GT).
float* h_GLZ = (float*)malloc(nElements*sizeof(float));
if (h_GLZ == 0)
{
    fprintf(stderr, "!!!! host memory allocation error (h_GLZ)\n");
    return EXIT_FAILURE;
}

for (i = 0; i < colsG; i++)
{
    for (j = 0; j < colsT; j++)
    {
        //High point for North Side
        h_GHX[k] = h_G[12+(i*14)];
        h_GHY[k] = h_G[1+(i*14)];
        h_GHZ[k] = h_G[2+(i*14)];
        //High point for East Side
        h_GHX[k+colsT] = h_G[12+(i*14)];
        h_GHY[k+colsT] = h_G[1+(i*14)];
        h_GHZ[k+colsT] = h_G[2+(i*14)];
        //High point for Top Side
        h_GHX[k+(2*colsT)] = h_G[12+(i*14)];
        h_GHY[k+(2*colsT)] = h_G[1+(i*14)];
        h_GHZ[k+(2*colsT)] = h_G[2+(i*14)];
        //High point for South Side
        h_GHX[k+(3*colsT)] = h_G[12+(i*14)];
        h_GHY[k+(3*colsT)] = h_G[1+(i*14)];
        h_GHZ[k+(3*colsT)] = h_G[2+(i*14)];
        //High point for West Side
        h_GHX[k+(4*colsT)] = h_G[12+(i*14)];
        h_GHY[k+(4*colsT)] = h_G[1+(i*14)];
        h_GHZ[k+(4*colsT)] = h_G[2+(i*14)];
        //High point for Bottom Side
        h_GHX[k+(5*colsT)] = h_G[12+(i*14)];
        h_GHY[k+(5*colsT)] = h_G[1+(i*14)];
        h_GHZ[k+(5*colsT)] = h_G[2+(i*14)];
        //Low point for North Side
        h_GLX[k] = h_G[13+(i*14)];
        h_GLY[k] = h_G[4+(i*14)];
        h_GLZ[k] = h_G[2+(i*14)];
        //Low point for East Side
        h_GLX[k+colsT] = h_G[13+(i*14)];
        h_GLY[k+colsT] = h_G[4+(i*14)];
        h_GLZ[k+colsT] = h_G[2+(i*14)];
        //Low point for Top Side
        h_GLX[k+(2*colsT)] = h_G[13+(i*14)];
        h_GLY[k+(2*colsT)] = h_G[4+(i*14)];
        h_GLZ[k+(2*colsT)] = h_G[2+(i*14)];
        //Low point for South Side
        h_GLX[k+(3*colsT)] = h_G[13+(i*14)];
        h_GLY[k+(3*colsT)] = h_G[7+(i*14)];
        h_GLZ[k+(3*colsT)] = h_G[2+(i*14)];
        //Low point for West Side
        h_GLX[k+(4*colsT)] = h_G[13+(i*14)];
        h_GLY[k+(4*colsT)] = h_G[4+(i*14)];
        h_GLZ[k+(4*colsT)] = h_G[2+(i*14)];
        //Low point for Bottom Side
        h_GLX[k+(5*colsT)] = h_G[13+(i*14)];
        h_GLY[k+(5*colsT)] = h_G[4+(i*14)];
        h_GLZ[k+(5*colsT)] = h_G[2+(i*14)];
        k++;
    }
    g++;
}

k = g*6*colsT;

//Splitting of Data to Send to Separate GPUs
float* h_GHX0;
float* h_GHY0;
float* h_GHZ0;
float* h_GHX1;
float* h_GHY1;
float* h_GHZ1;

//Allocate HOST PINNED memory (CPU) for Arrays GHX0, GHY0, and GHZ0 (1 x 3GT).
cudaSetDevice(0);
CudaErrCheck( cudaMalloc((void**)&h_GHX0, nBytesF) );
CudaErrCheck( cudaMalloc((void**)&h_GHY0, nBytesF) );
CudaErrCheck( cudaMalloc((void**)&h_GHZ0, nBytesF) );
//Allocate HOST PINNED memory (CPU) for Arrays GHX1, GHY1 and GHZ1 (1 x 3GT).
cudaSetDevice(1);
CudaErrCheck( cudaMalloc((void**)&h_GHX1, nBytesF) );
CudaErrCheck( cudaMalloc((void**)&h_GHY1, nBytesF) );
CudaErrCheck( cudaMalloc((void**)&h_GHZ1, nBytesF) );

for (i = 0; i < nElementsDevice; i++)
{
    h_GHX0[i] = h_GHX[i];
    h_GHY0[i] = h_GHY[i];
    h_GHZ0[i] = h_GHZ[i];
    h_GHX1[i] = h_GHX[i+nElementsDevice];
    h_GHY1[i] = h_GHY[i+nElementsDevice];
    h_GHZ1[i] = h_GHZ[i+nElementsDevice];
}

//MEMORY CLEAN UP GHX, GHY, GHZ.
free(h_GHX);
free(h_GHY);
free(h_GHZ);
float* h_GLX0;
float* h_GLY0;
float* h_GLZ0;
float* h_GLX1;
float* h_GLY1;
float* h_GLZ1;

//Allocate HOST PINNED memory (CPU) for Arrays GLX0, GLY0, and GLZ0 (1 x 3GT).
cudaSetDevice(0);
CudaErrCheck( cudaMalloc((void**)&h_GLX0, nBytesF) );
CudaErrCheck( cudaMalloc((void**)&h_GLY0, nBytesF) );
CudaErrCheck( cudaMalloc((void**)&h_GLZ0, nBytesF) );
//Allocate HOST PINNED memory (CPU) for Arrays GLX1, GLY1, and GLZ1 (1 x 3GT).
cudaSetDevice(1);
CudaErrCheck( cudaMalloc((void**)&h_GLX1, nBytesF) );
CudaErrCheck( cudaMalloc((void**)&h_GLY1, nBytesF) );
CudaErrCheck( cudaMalloc((void**)&h_GLZ1, nBytesF) );

for (i = 0; i < nElementsDevice; i++)
{
    h_GLX0[i] = h_GLX[i];
    h_GLY0[i] = h_GLY[i];
    h_GLZ0[i] = h_GLZ[i];
    h_GLX1[i] = h_GLX[i+nElementsDevice];
    h_GLY1[i] = h_GLY[i+nElementsDevice];
    h_GLZ1[i] = h_GLZ[i+nElementsDevice];
}

//MEMORY CLEAN UP GLX, GLY, GLZ.
free(h_GLX);
free(h_GLY);
free(h_GLZ);

//Development of AX, AY, AZ (1 x 6GT) Arrays.
//These arrays are used to store a point on each plane of each geometry.
//Used in the equation to find Thit.
float* h_AX;
float* h_AY;
float* h_AZ;

//Allocate HOST memory (CPU) for Array AX (1 x 6GT).
h_AX = (float*)malloc(nElements*sizeof(float));
if (h_AX == 0)
{
    fprintf (stderr, "!!!! host memory allocation error (h_AX)\n");
    return EXIT_FAILURE;
//Allocate HOST memory (CPU) for Array AY (1 x 6GT).
h_AY = (float*)malloc(nElements*sizeof(float));
if (h_AY == 0)
    {
        fprintf (stderr, "!!! host memory allocation error (h_AY)\n");
        return EXIT_FAILURE;
    }

//Allocate HOST memory (CPU) for Array AZ (1 x 6GT).
h_AZ = (float*)malloc(nElements*sizeof(float));
if (h_AZ == 0)
    {
        fprintf (stderr, "!!! host memory allocation error (h_AZ)\n");
        return EXIT_FAILURE;
    }

//For loop used to store each a point on each plane of each geometry.
//Each point on a plane needs to be repeated MaxTrack amount of times to account
//for each track that is to be compared to each plane of each geometry.
k = 0;
g = 0;
for (i = 0; i < colsG; i++)
    {
        for (j = 0; j < colsT; j++)
            {
                //Point on plane: North Side.
                h_AX[k] = h_G[12+(i*14)];
                h_AY[k] = h_G[1+(i*14)];
                h_AZ[k] = h_G[2+(i*14)];
                //Point on plane: East Side.
                h_AX[k+colsT] = h_G[12+(i*14)];
                h_AY[k+colsT] = h_G[1+(i*14)];
                h_AZ[k+colsT] = h_G[2+(i*14)];
                //Point on plane: Top Side.
                h_AX[k+(2*colsT)] = h_G[12+(i*14)];
                h_AY[k+(2*colsT)] = h_G[1+(i*14)];
                h_AZ[k+(2*colsT)] = h_G[2+(i*14)];
                //Point on plane: South Side.
                h_AX[k+(3*colsT)] = h_G[13+(i*14)];
                h_AY[k+(3*colsT)] = h_G[7+(i*14)];
                h_AZ[k+(3*colsT)] = h_G[8+(i*14)];
                //Point on plane: West Side.
                h_AX[k+(4*colsT)] = h_G[13+(i*14)];
                h_AY[k+(4*colsT)] = h_G[7+(i*14)];
                h_AZ[k+(4*colsT)] = h_G[8+(i*14)];
                //Point on plane: Bottom Side.
                h_AX[k+(5*colsT)] = h_G[13+(i*14)];
                h_AY[k+(5*colsT)] = h_G[7+(i*14)];
                h_AZ[k+(5*colsT)] = h_G[8+(i*14)];
                k++;
            }
        g++;
        k = g*6*colsT;
    }

//Splitting of Data to Send to Separate GPUs
float* h_AX0;
float* h_AY0;
float* h_AZ0;
float* h_AX1;
float* h_AY1;
float* h_AZ1;
//Allocate HOST PINNED memory (CPU) for Arrays AX0, AY0, and AZ0 (1 x 3GT).
cudaSetDevice(0);
CudaErrCheck( cudaMallocHost((void**)&h_AX0, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_AY0, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_AZ0, nBytesF) );
//Allocate HOST PINNED memory (CPU) for Arrays AX1, AY1, and AZ1 (1 x 3GT).
cudaSetDevice(1);
CudaErrCheck( cudaMallocHost((void**)&h_AX1, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_AY1, nBytesF) );
CudaErrCheck( cudaMallocHost((void**)&h_AZ1, nBytesF) );
for (i = 0; i < nElementsDevice; i++)
{  
  h_AX0[i] = h_AX[i];  
  h_AY0[i] = h_AY[i];  
  h_AZ0[i] = h_AZ[i];  
  //  
  h_AX1[i] = h_AX[i+nElementsDevice];  
  h_AY1[i] = h_AY[i+nElementsDevice];  
  h_AZ1[i] = h_AZ[i+nElementsDevice];  
}  
//MEMORY CLEAN UP G.  
free(h_G);  
//MEMORY CLEAN UP AX, AY, AZ.  
free(h_AX);  
free(h_AY);  
free(h_AZ);  
//-----------------------------------------------------------------------------//  
//Host variables used to retrieve date from the four Streams  
int TrackID;  
int GeometryID;  
float* h_IntPX0;  
float* h_IntPY0;  
float* h_IntPZ0;  
float* h_DistInt0;  
float* h_Vel0;  
float* h_TimeInt0;  
float* h_IntPX1;  
float* h_IntPY1;  
float* h_IntPZ1;  
float* h_DistInt1;  
float* h_Vel1;  
float* h_TimeInt1;  
cudaSetDevice(0);  
//Allocate HOST PINNED memory (CPU) for Array IntPX0 (1 x 3GT).  
CudaErrCheck( cudaMallocHost((void**)&h_IntPX0, nBytesF) );  
//Allocate HOST PINNED memory (CPU) for Array IntPY0 (1 x 3GT).  
CudaErrCheck( cudaMallocHost((void**)&h_IntPY0, nBytesF) );  
//Allocate HOST PINNED memory (CPU) for Array IntPZ0 (1 x 3GT).  
CudaErrCheck( cudaMallocHost((void**)&h_IntPZ0, nBytesF) );  
//Allocate HOST PINNED memory (CPU) for Array DistInt0 (1 x 3GT).  
CudaErrCheck( cudaMallocHost((void**)&h_DistInt0, nBytesF) );  
//Allocate HOST PINNED memory (CPU) for Array Vel0 (1 x 3GT).  
CudaErrCheck( cudaMallocHost((void**)&h_Vel0, nBytesF) );  
//Allocate HOST PINNED memory (CPU) for Array TimeInt0 (1 x 3GT).  
CudaErrCheck( cudaMallocHost((void**)&h_TimeInt0, nBytesF) );  
cudaSetDevice(1);  
//Allocate HOST PINNED memory (CPU) for Array IntPX1 (1 x 3GT).  
CudaErrCheck( cudaMallocHost((void**)&h_IntPX1, nBytesF) );  
//Allocate HOST PINNED memory (CPU) for Array IntPY1 (1 x 3GT).  
CudaErrCheck( cudaMallocHost((void**)&h_IntPY1, nBytesF) );  
//Allocate HOST PINNED memory (CPU) for Array IntPZ1 (1 x 3GT).  
CudaErrCheck( cudaMallocHost((void**)&h_IntPZ1, nBytesF) );  
//Allocate HOST PINNED memory (CPU) for Array DistInt1 (1 x 3GT).  
CudaErrCheck( cudaMallocHost((void**)&h_DistInt1, nBytesF) );  
//Allocate HOST PINNED memory (CPU) for Array Vel1 (1 x 3GT).  
CudaErrCheck( cudaMallocHost((void**)&h_Vel1, nBytesF) );  
//Allocate HOST PINNED memory (CPU) for Array TimeInt1 (1 x 3GT).  
CudaErrCheck( cudaMallocHost((void**)&h_TimeInt1, nBytesF) );  
//-----------------------------------------------------------------------------//  
//For loop to repeat 5 times to in order to get an average execution time.  
//This for loop is for testing purposes only.  
for (n = 0; n < 5; n++)  
{  
  printf("Loop: %i\n", n+1);  
  //--------------------------------------------------------------------------------------------------  
  //!!!!!!!!!!!!!!!!!!!STARTING VIOLATION CHECK PROCESS!!!!!!!!!!!!!!!!!!!!!!!  
  //Recording of execution time begins now.  
  cudaSetDevice(0);  
  CudaErrCheck( cudaEventRecord(StartEvent,0) );  
  //--------------------------------------------------------------------------------------------------  
  //Creation of Streams.  
  cudaStream_t s0[nStreams];
cudaStream_t s1[nStreams];
cudaSetDevice(0);
for(i=0; i < nStreams; i++)
{
    cudaStreamCreate(&s0[i]);
}
cudaSetDevice(1);
for(i=0; i < nStreams; i++)
{
    cudaStreamCreate(&s1[i]);
}

//--Declaration of device variables used in program.
float* d_TCX0;
float* d_TCY0;
float* d_DistInt0;
float* d_BX0;
float* d_BY0;
float* d_Vel0;
float* d_AX0;
float* d_AV0;
float* d_Thit0;
float* d_NX0;
float* d_NV0;
float* d_NZ0;
float* d_IntPX0;
float* d_IntPY0;
float* d_IntPZ0;
float* d_TimeInt0;
float* d_GHX0;
float* d_GHY0;
float* d_GHZ0;
float* d_GLX0;
float* d_GLY0;
float* d_GLZ0;
unsigned char* d_ComTemp0;
unsigned char* d_ComTotal0;
int* d_Index0;
float* d_TCX1;
float* d_TCY1;
float* d_DistInt1;
float* d_BX1;
float* d_BY1;
float* d_Vel1;
float* d_AX1;
float* d_AV1;
float* d_Thit1;
float* d_NX1;
float* d_NV1;
float* d_NZ1;
float* d_IntPX1;
float* d_IntPY1;
float* d_IntPZ1;
float* d_TimeInt1;
float* d_GHX1;
float* d_GHY1;
float* d_GHZ1;
float* d_GLX1;
float* d_GLY1;
float* d_GLZ1;
unsigned char* d_ComTemp1;
unsigned char* d_ComTotal1;
int* d_Index1;

//DEVICE MEMORY ALLOCATION
cudaSetDevice(0);
CudaErrCheck( cudaMalloc((void **)d_TCX0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_TCY0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_DistInt0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_BX0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_AX0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_BY0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_AV0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_GHX0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_GHY0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_GHZ0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_GLX0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_GLY0, nBytesF) );
CudaErrCheck( cudaMalloc((void **)d_GLZ0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_AY0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_Thit0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_NX0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_NY0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_NZ0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_IntPX0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_IntPY0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_IntPZ0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_TimeInt0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_GHX0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_GHY0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_GHZ0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_GLX0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_GLY0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_GLZ0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_ComTemp0, nBytesUC) );
CudaErrCheck( cudaMalloc((void **) &d_ComTotal0, nBytesUC) );
CudaErrCheck( cudaMalloc((void **) &d_Index0, nBytesI) );
cudaSetDevice(1);
CudaErrCheck( cudaMalloc((void **) &d_TCX1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_TCY1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_DistInt1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_BX1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_BY1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_Vel1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_AX1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_AY1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_AZ1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_NX1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_NY1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_NZ1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_IntPX1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_IntPY1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_DistInt1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_TCZ0, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_GH1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_GF1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_GLX1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_GLY1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_GLZ1, nBytesF) );
CudaErrCheck( cudaMalloc((void **) &d_ComTemp1, nBytesUC) );
CudaErrCheck( cudaMalloc((void **) &d_ComTotal1, nBytesUC) );
CudaErrCheck( cudaMalloc((void **) &d_Index1, nBytesI) );

// HOST TO DEVICE DATA TRANSFER
cudaSetDevice(0);
for (i = 0; i < nStreams; i++) {
  offset = i * StreamSize;
  CudaErrCheck( cudaMemcpyAsync(&d_TCX0[offset], &h_TCX0[offset],
      nStreamBytesF, cudaMemcpyHostToDevice, s0[i]) );
  CudaErrCheck( cudaMemcpyAsync(&d_TCY0[offset], &h_TCY0[offset],
      nStreamBytesF, cudaMemcpyHostToDevice, s0[i]) );
  CudaErrCheck( cudaMemcpyAsync(&d_TCZ0[offset], &h_TCZ0[offset],
      nStreamBytesF, cudaMemcpyHostToDevice, s0[i]) );
  CudaErrCheck( cudaMemcpyAsync(&d_BX0[offset], &h_BX0[offset],
      nStreamBytesF, cudaMemcpyHostToDevice, s0[i]) );
  CudaErrCheck( cudaMemcpyAsync(&d_BY0[offset], &h_BY0[offset],
      nStreamBytesF, cudaMemcpyHostToDevice, s0[i]) );
  CudaErrCheck( cudaMemcpyAsync(&d_BZ0[offset], &h_BZ0[offset],
      nStreamBytesF, cudaMemcpyHostToDevice, s0[i]) );
  CudaErrCheck( cudaMemcpyAsync(&d_AX0[offset], &h_AX0[offset],
      nStreamBytesF, cudaMemcpyHostToDevice, s0[i]) );
  CudaErrCheck( cudaMemcpyAsync(&d_AY0[offset], &h_AY0[offset],
      nStreamBytesF, cudaMemcpyHostToDevice, s0[i]) );
  CudaErrCheck( cudaMemcpyAsync(&d_AZ0[offset], &h_AZ0[offset],
      nStreamBytesF, cudaMemcpyHostToDevice, s0[i]) );
  CudaErrCheck( cudaMemcpyAsync(&d_NX0[offset], &h_NX0[offset],
      nStreamBytesF, cudaMemcpyHostToDevice, s0[i]) );
  CudaErrCheck( cudaMemcpyAsync(&d_NY0[offset], &h_NY0[offset],
      nStreamBytesF, cudaMemcpyHostToDevice, s0[i]) );
  CudaErrCheck( cudaMemcpyAsync(&d_NZ0[offset], &h_NZ0[offset],
      nStreamBytesF, cudaMemcpyHostToDevice, s0[i]) );
}
CUDAErrCheck( cudaMemcpyAsync(&d_NZ0[offset], &h_NZ0[offset], nStreamBytesF, cudaMemcpyHostToDevice, s0[i] ));
CUDAErrCheck( cudaMemcpyAsync(&d_GHX0[offset], &h_GHX0[offset], nStreamBytesF, cudaMemcpyHostToDevice, s0[i] ));
CUDAErrCheck( cudaMemcpyAsync(&d_GHY0[offset], &h_GHY0[offset], nStreamBytesF, cudaMemcpyHostToDevice, s0[i] ));
CUDAErrCheck( cudaMemcpyAsync(&d_GHZ0[offset], &h_GHZ0[offset], nStreamBytesF, cudaMemcpyHostToDevice, s0[i] ));
CUDAErrCheck( cudaMemcpyAsync(&d_GLX0[offset], &h_GLX0[offset], nStreamBytesF, cudaMemcpyHostToDevice, s0[i] ));
CUDAErrCheck( cudaMemcpyAsync(&d_GLY0[offset], &h_GLY0[offset], nStreamBytesF, cudaMemcpyHostToDevice, s0[i] ));
CUDAErrCheck( cudaMemcpyAsync(&d_GLZ0[offset], &h_GLZ0[offset], nStreamBytesF, cudaMemcpyHostToDevice, s0[i] ));
}
cudaSetDevice(1);
for (i = 0; i < nStreams; i++)
{
    offset = i * StreamSize;
    CUDAErrCheck( cudaMemcpyAsync(&d_TCX1[offset], &h_TCX1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_TCY1[offset], &h_TCY1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_DistInt1[offset], &h_TCZ1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_BX1[offset], &h_BX1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_BY1[offset], &h_BY1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_Vel1[offset], &h_BZ1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_AX1[offset], &h_AX1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_AY1[offset], &h_AY1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_Thit1[offset], &h_AZ1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_NX1[offset], &h_NX1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_NY1[offset], &h_NY1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_NZ1[offset], &h_NZ1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_GHX1[offset], &h_GHX1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_GHY1[offset], &h_GHY1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_GHZ1[offset], &h_GHZ1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_GLX1[offset], &h_GLX1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_GLY1[offset], &h_GLY1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
    CUDAErrCheck( cudaMemcpyAsync(&d_GLZ1[offset], &h_GLZ1[offset], nStreamBytesF, cudaMemcpyHostToDevice, s1[i] ));
}

//All calculations are the same as the single GPU version.
//The only difference is the data is divided among the two GPUs.
//CALCULATIONS
cudaSetDevice(0);
for (i = 0; i < nStreams; i++)
{
    offset = i * StreamSize;
nppsSetStream(s0[i]);
nppsSub_32f(&d_BX0[offset], &d_TCX0[offset], &d_BX0[offset], StreamSize);
nppsSub_32f(&d_BY0[offset], &d_TCY0[offset], &d_BY0[offset], StreamSize);
nppsSub_32f(&d_Vel0[offset], &d_DistInt0[offset], &d_Vel0[offset], StreamSize);
nppsSub_32f_I(&d_TCX0[offset], &d_AX0[offset], StreamSize);
nppsSub_32f_I(&d_TCY0[offset], &d_AY0[offset], StreamSize);
}
nppsSub_32f_I(&d_DistInt0[offset], &d_Thit0[offset], StreamSize);
nppsMul_32f_I(&d_NX0[offset], &d_AX0[offset], StreamSize);
nppsMul_32f_I(&d_NY0[offset], &d_AY0[offset], StreamSize);
nppsMul_32f_I(&d_NZ0[offset], &d_AZ0[offset], StreamSize);
nppsAdd_32f_I(&d_AY0[offset], &d_Thit0[offset], StreamSize);
nppsMul_32f_I(&d_BX0[offset], &d_NX0[offset], StreamSize);
nppsMul_32f_I(&d_BY0[offset], &d_NY0[offset], StreamSize);
nppsMul_32f_I(&d_Vel0[offset], &d_NZ0[offset], StreamSize);
nppsAdd_32f_I(&d_NX0[offset], &d_NY0[offset], StreamSize);
nppsAdd_32f_I(&d_NY0[offset], &d_NZ0[offset], StreamSize);
nppsDiv_32f_I(&d_NZ0[offset], &d_Thit0[offset], StreamSize);
nppsCopy_32f(&d_BX0[offset], &d_IntPX0[offset], StreamSize);
nppsCopy_32f(&d_BY0[offset], &d_IntPY0[offset], StreamSize);
nppsCopy_32f(&d_Vel0[offset], &d_IntPZ0[offset], StreamSize);
nppsMul_32f_I(&d_Thit0[offset], &d_IntPX0[offset], StreamSize);
nppsMul_32f_I(&d_Thit0[offset], &d_IntPY0[offset], StreamSize);
nppsMul_32f_I(&d_Thit0[offset], &d_IntPZ0[offset], StreamSize);
nppsAdd_32f_I(&d_TCX0[offset], &d_IntPX0[offset], StreamSize);
nppsAdd_32f_I(&d_TCY0[offset], &d_IntPY0[offset], StreamSize);
nppsAdd_32f_I(&d_DistInt0[offset], &d_IntPZ0[offset], StreamSize);
nppsSqr_32f_I(&d_BX0[offset], StreamSize);
nppsSqr_32f_I(&d_BY0[offset], StreamSize);
nppsSqr_32f_I(&d_Vel0[offset], StreamSize);
nppsAdd_32f_I(&d_BX0[offset], &d_BY0[offset], StreamSize);
nppsAdd_32f_I(&d_BY0[offset], &d_Vel0[offset], StreamSize);

nppsSqrt_32f_I(&d_Vel0[offset], StreamSize);
nppsDivC_32f_I(TimeDiff, &d_Vel0[offset], StreamSize);
nppsSub_32f(&d_TCX0[offset], &d_IntPX0[offset], &d_TCX0[offset], StreamSize);
nppsSub_32f(&d_TCY0[offset], &d_IntPY0[offset], &d_TCY0[offset], StreamSize);
nppsSub_32f(&d_DistInt0[offset], &d_IntPZ0[offset], &d_DistInt0[offset], StreamSize);
nppsSqr_32f_I(&d_TCX0[offset], StreamSize);
nppsSqr_32f_I(&d_TCY0[offset], StreamSize);
nppsSqr_32f_I(&d_DistInt0[offset], StreamSize);
nppsAdd_32f_I(&d_TCX0[offset], &d_TCY0[offset], StreamSize);
nppsAdd_32f_I(&d_TCY0[offset], &d_DistInt0[offset], StreamSize);
nppsSqrt_32f_I(&d_DistInt0[offset], StreamSize);
nppsDiv_32f(&d_Vel0[offset], &d_DistInt0[offset], &d_TimeInt0[offset], StreamSize);
nppsSubC_32f_I(0.0001, &d_IntPX0[offset], StreamSize);
nppsSubC_32f_I(0.0001, &d_IntPY0[offset], StreamSize);
nppsSubC_32f_I(0.0001, &d_IntPZ0[offset], StreamSize);
nppiCompare_32f_C1R(&d_IntPX0[offset], LineStepF, &d_GHX0[offset], LineStepF, &d_ComTemp0[offset], LineStepUC, sz, NPP_CMP_GREATER);
nppiCompare_32f_C1R(&d_IntPY0[offset], LineStepF, &d_GHY0[offset], LineStepF, &d_ComTemp0[offset], LineStepUC, sz, NPP_CMP_GREATER);
nppiCompare_32f_C1R(&d_IntPZ0[offset], LineStepF, &d_GHZ0[offset], LineStepF, &d_ComTemp0[offset], LineStepUC, sz, NPP_CMP_GREATER);
nppsAdd_8u_ISfs(&d_ComTemp0[offset], &d_ComTotal0[offset], StreamSize, 0);
nppiCompare_32f_C1R(&d_IntPZ0[offset], LineStepF, &d_GHZ0[offset], LineStepF, &d_ComTemp0[offset], LineStepUC, sz, NPP_CMP_LESS);
nppiCompare_32f_C1R(&d_IntPY0[offset], LineStepF, &d_GHY0[offset], LineStepF, &d_ComTemp0[offset], LineStepUC, sz, NPP_CMP_LESS);
nppiCompare_32f_C1R(&d_IntPX0[offset], LineStepF, &d_GHX0[offset], LineStepF, &d_ComTemp0[offset], LineStepUC, sz, NPP_CMP_LESS);
nppsAdd_8u_ISfs(&d_ComTemp0[offset], &d_ComTotal0[offset], StreamSize, 0);
nppsSub_32f_I(0.0001, &d_IntPX0[offset], StreamSize);
nppsSub_32f_I(0.0001, &d_IntPY0[offset], StreamSize);
nppsSub_32f_I(0.0001, &d_IntPZ0[0ffset], StreamSize);
nppiCompareC_32f_C1R(&d_Thit0[0ffset], LineStepF, 0,
&d_ComTemp0[0ffset], LineStepUC, sz, NPP_CMP_LESS);
nppsAdd_8u_ISfs(&d_ComTemp0[0ffset], &d_ComTotal0[0ffset], StreamSize, 0);
nppiCompareC_32f_C1R(&d_TimeInt0[0ffset], LineStepF, Buffer,
&d_ComTemp0[0ffset], LineStepUC, sz, NPP_CMP_GREATER);
nppsAdd_8u_ISfs(&d_ComTemp0[0ffset], &d_ComTotal0[0ffset], StreamSize, 0);
}
cudaSetDevice(1);
for (i = 0; i < nStreams; i++) {
    offset = i * StreamSize;
    nppSetStream(s1[i]);
    nppSub_32f(&d_BX1[0ffset], &d_TCX1[0ffset], &d_BX1[0ffset], StreamSize);
    nppSub_32f(&d_BY1[0ffset], &d_TCY1[0ffset], &d_BY1[0ffset], StreamSize);
    nppSub_32f(&d_Vel1[0ffset], &d_DistInt1[0ffset], &d_Vel1[0ffset], StreamSize);
    nppsSubC_32f_I(0.0001, &d_IntPX1[0ffset], StreamSize);
    nppiCompareC_32f_C1R(&d_IntPX1[0ffset], LineStepF, &d_GHX1[0ffset], LineStepUC, sz, NPP_CMP_GREATER);
    nppiCompare_32f_C1R(&d_IntPY1[0ffset], LineStepF, &d_GHY1[0ffset], LineStepUC, sz, NPP_CMP_GREATER);
    nppsDiv_32f_I(&d_Vel1[0ffset], &d_DistInt1[0ffset], &d_TimeInt1[0ffset], StreamSize);
    nppSub_32f_I(&d_TCX1[0ffset], &d_IntPX1[0ffset], &d_TCX1[0ffset], StreamSize);
    nppSub_32f_I(&d_TCY1[0ffset], &d_IntPY1[0ffset], &d_TCY1[0ffset], StreamSize);
    nppSub_32f_I(&d_DistInt1[0ffset], &d_IntPZ1[0ffset], StreamSize);
}

nppsSub_32f(&d_BX1[0ffset], &d_TCX1[0ffset], &d_BX1[0ffset], StreamSize);
}
nppsAdd_8u_ISfs(&d_ComTemp1[offset], &d_ComTotal1[offset], StreamSize, 0);
nppiCompare_32f_C1R(&d_IntPZ1[offset], LineStepF, &d_GHZ1[offset], LineStepF, &d_ComTemp1[offset], LineStepUC, sz, NPP_CMP_GREATER);
nppsAdd_8u_ISfs(&d_ComTemp1[offset], &d_ComTotal1[offset], StreamSize, 0);
nppiCompare_32f_C1R(&d_IntPZ1[offset], LineStepF, &d_GLZ1[offset], LineStepF, &d_ComTemp1[offset], LineStepUC, sz, NPP_CMP_LESS);
nppsAdd_8u_ISfs(&d_ComTemp1[offset], &d_ComTotal1[offset], StreamSize, 0);

nppsAddC_32f_I(0.0002, &d_IntPX1[offset], StreamSize);
nppsAddC_32f_I(0.0002, &d_IntPY1[offset], StreamSize);
nppsAddC_32f_I(0.0002, &d_IntPZ1[offset], StreamSize);

nppiCompare_32f_C1R(&d_IntPX1[offset], LineStepF, &d_GLX1[offset], LineStepF, &d_ComTemp1[offset], LineStepUC, sz, NPP_CMP_LESS);
nppsAdd_8u_ISfs(&d_ComTemp1[offset], &d_ComTotal1[offset], StreamSize, 0);
nppiCompare_32f_C1R(&d_IntPY1[offset], LineStepF, &d_GLY1[offset], LineStepF, &d_ComTemp1[offset], LineStepUC, sz, NPP_CMP_LESS);
nppsAdd_8u_ISfs(&d_ComTemp1[offset], &d_ComTotal1[offset], StreamSize, 0);

//DEVICE TO HOST DATA TRANSFER
cudaSetDevice(0);
for (i = 0; i < nStreams; i++)
{
    offset = i * StreamSize;
    CudaErrCheck( cudaMemcpyAsync(&h_IntPX0[offset], &d_IntPX0[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s0[i]));
    CudaErrCheck( cudaMemcpyAsync(&h_IntPY0[offset], &d_IntPY0[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s0[i]));
    CudaErrCheck( cudaMemcpyAsync(&h_IntPZ0[offset], &d_IntPZ0[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s0[i]));
    CudaErrCheck( cudaMemcpyAsync(&h_DistInt0[offset], &d_DistInt0[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s0[i]));
    CudaErrCheck( cudaMemcpyAsync(&h_Vel0[offset], &d_Vel0[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s0[i]));
    CudaErrCheck( cudaMemcpyAsync(&h_TimeInt0[offset], &d_TimeInt0[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s0[i]));
}
cudaSetDevice(1);
for (i = 0; i < nStreams; i++)
{
    offset = i * StreamSize;
    CudaErrCheck( cudaMemcpyAsync(&h_IntPX1[offset], &d_IntPX1[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s1[i]));
    CudaErrCheck( cudaMemcpyAsync(&h_IntPY1[offset], &d_IntPY1[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s1[i]));
    CudaErrCheck( cudaMemcpyAsync(&h_IntPZ1[offset], &d_IntPZ1[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s1[i]));
    CudaErrCheck( cudaMemcpyAsync(&h_DistInt1[offset], &d_DistInt1[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s1[i]));
    CudaErrCheck( cudaMemcpyAsync(&h_Vel1[offset], &d_Vel1[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s1[i]));
    CudaErrCheck( cudaMemcpyAsync(&h_TimeInt1[offset], &d_TimeInt1[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s1[i]));
}
CudaErrCheck( cudaMemcpyAsync(&h_Vel1[offset], &d_Vel1[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s1[i]) );
CudaErrCheck( cudaMemcpyAsync(&h_TimeInt1[offset], &d_TimeInt1[offset], nStreamBytesF, cudaMemcpyDeviceToHost, s1[i]) );

} cudaSetDevice(0);
thrust::device_ptr<int> dev_ptr_Index0(d_Index0);
thrust::device_ptr<unsigned char> dev_ptr_ComTotal0(d_ComTotal0);
result0 = thrust::count_if(dev_ptr_ComTotal0, dev_ptr_ComTotal0 + nElementsDevice, equal_to_zero());
cudaSetDevice(1);
thrust::device_ptr<int> dev_ptr_Index1(d_Index1);
thrust::device_ptr<unsigned char> dev_ptr_ComTotal1(d_ComTotal1);
result1 = thrust::count_if(dev_ptr_ComTotal1, dev_ptr_ComTotal1 + nElementsDevice, equal_to_zero());

//---------------------------------------------------------------------//
//STEP 8: Output Violations Results
if ((result0 != 0) || (result1 != 0))
{
  int* h_IndexFinal0;
  int* h_IndexFinal1;
  int* d_IndexFinal0;
  int* d_IndexFinal1;
  //HOST MEMORY ALLOCATION
  h_IndexFinal0 = (int*)malloc(result0*sizeof(int));
  if (h_IndexFinal0 == 0) {
    fprintf (stderr, "!!!! host memory allocation error
                  (h_IndexFinal0)\n");
    return EXIT_FAILURE;
  }
  h_IndexFinal1 = (int*)malloc(result1*sizeof(int));
  if (h_IndexFinal1 == 0) {
    fprintf (stderr, "!!!! host memory allocation error
                  (h_IndexFinal1)\n");
    return EXIT_FAILURE;
  }
  //DEVICE MEMORY ALLOCATION
cudaSetDevice(0);
  CudaErrCheck( cudaMalloc((void **)&d_IndexFinal0,
                  result0*sizeof(int)) );
cudaSetDevice(1);
  CudaErrCheck( cudaMalloc((void **)&d_IndexFinal1,
                  result1*sizeof(int)) );
  //CALCULATIONS
  cudaSetDevice(0);
  thrust::device_ptr<int> dev_ptr_IndexFinal0(d_IndexFinal0);
  thrust::remove_copy_if(dev_ptr_Index0, dev_ptr_Index0 + nElementsDevice, dev_ptr_ComTotal0, dev_ptr_IndexFinal0, not_equal_to_zero());
cudaSetDevice(1);
  thrust::device_ptr<int> dev_ptr_IndexFinal1(d_IndexFinal1);
  thrust::remove_copy_if(dev_ptr_Index1, dev_ptr_Index1 + nElementsDevice, dev_ptr_ComTotal1, dev_ptr_IndexFinal1, not_equal_to_zero());
  //DEVICE TO HOST DATA TRANSFER
cudaSetDevice(0);
  CudaErrCheck( cudaMemcpy(h_IndexFinal0, d_IndexFinal0, result0*sizeof(int), cudaMemcpyDeviceToHost) );
cudaSetDevice(1);
  CudaErrCheck( cudaMemcpy(h_IndexFinal1, d_IndexFinal1, result1*sizeof(int), cudaMemcpyDeviceToHost) );
  //Prints out Violations based on the values in d_IndexFinal.
  for (i=0; i<result0; i++)
  {
    if ((h_IndexFinal0[i]+1)%colsT==0)
    {
      TrackID = colsT;
else
{
    TrackID = (h_IndexFinal0[i]+1)%colsT;
}
GeometryID = (h_IndexFinal0[i]/(6*colsT))+1;
//printf("Track ID: %i
", TrackID);
//printf("Geometry ID: %i\n", GeometryID);
fprintf(ParallelViolations, "Track ID: %i\n", TrackID);
fprintf(ParallelViolations, "Geometry ID: %i\n", GeometryID);
fprintf(ParallelViolations, "Distance: %.6f\n", h_DistInt0[h_IndexFinal0[i]]);
fprintf(ParallelViolations, "Velocity: %.6f\n", h_Vel0[h_IndexFinal0[i]]);
fprintf(ParallelViolations, "Intersect Time: %.6f\n", h_TimeInt0[h_IndexFinal0[i]]);
fprintf(ParallelViolations, "Intersect Point: (%.6f, %.6f, %.6f)\n", h_IntPX0[h_IndexFinal0[i]], h_IntPY0[h_IndexFinal0[i]], h_IntPZ0[h_IndexFinal0[i]]);
fprintf(ParallelViolations, "\n");
}
for (i=0; i<result1; i++)
{
    if ((h_IndexFinal1[i]+1+nElementsDevice)%colsT==0)
    {
        TrackID = colsT;
    }
    else
    {
        TrackID = (h_IndexFinal1[i]+1+nElementsDevice)%colsT;
    }
GeometryID = ((h_IndexFinal1[i]+nElementsDevice)/(6*colsT))+1;
//printf("Track ID: %i\n", TrackID);
//printf("Geometry ID: %i\n", GeometryID);
fprintf(ParallelViolations, "Track ID: %i\n", TrackID);
fprintf(ParallelViolations, "Geometry ID: %i\n", GeometryID);
fprintf(ParallelViolations, "Distance: %.6f\n", h_DistInt1[h_IndexFinal1[i]]);
fprintf(ParallelViolations, "Velocity: %.6f\n", h_Vel1[h_IndexFinal1[i]]);
fprintf(ParallelViolations, "Intersect Time: %.6f\n", h_TimeInt1[h_IndexFinal1[i]]);
fprintf(ParallelViolations, "Intersect Point: (%.6f, %.6f, %.6f)\n", h_IntPX1[h_IndexFinal1[i]], h_IntPY1[h_IndexFinal1[i]], h_IntPZ1[h_IndexFinal1[i]]);
fprintf(ParallelViolations, "\n");
}
//MEMORY CLEAN UP IndexFinal.
free(h_IndexFinal0);
free(h_IndexFinal1);
cudaSetDevice(0);
CudaErrCheck( cudaFree(d_IndexFinal0) );
cudaSetDevice(1);
CudaErrCheck( cudaFree(d_IndexFinal1) );

//Destruction of Streams.
cudaSetDevice(0);
for (i = 0; i < nStreams; i++)
{
    CudaErrCheck( cudaStreamDestroy(s0[i]) );
}
cudaSetDevice(1);
for (i = 0; i < nStreams; i++)
{
    CudaErrCheck( cudaStreamDestroy(s1[i]) );
}
//---------------------------------------------------------------------//
//!!!!!!!!!!!!!!!!!!!!ENDING VIOLATION CHECK PROCESS!!!!!!!!!!!!!!!!!!!!!
//End of execution timer
cudaSetDevice(0);
CudaErrCheck( cudaEventRecord(StopEvent, 0) );
CudaErrCheck( cudaEventSynchronize(StopEvent) );
CudaErrCheck( cudaEventElapsedTime(&msec, StartEvent, StopEvent) );
AveTime = AveTime + msec;
if (n == 4)
{
    AveTime = AveTime/5;
    fprintf(ParallelExecutionTimes, "%i %i %.3f\n", colsT, colsG,
        msec);
    colsT = colsT + StepT;
    colsG = colsG + StepG;
    AveTime = 0;
}

//-----------------------------------------------------------------------------//
//Freeing Up DEVICE (GPU) Memory
cudaSetDevice(0);
//MEMORY CLEAN UP TCX0, TCY0, DistInt0.
CudaErrCheck( cudaFree(d_TCX0) );
CudaErrCheck( cudaFree(d_TCY0) );
CudaErrCheck( cudaFree(d_DistInt0) );
//MEMORY CLEAN UP BX0, BY0, Vel0.
CudaErrCheck( cudaFree(d_BX0) );
CudaErrCheck( cudaFree(d_BY0) );
CudaErrCheck( cudaFree(d_Vel0) );
//MEMORY CLEAN UP AX0, AY0, Thit0.
CudaErrCheck( cudaFree(d_AX0) );
CudaErrCheck( cudaFree(d_AY0) );
CudaErrCheck( cudaFree(d_Thit0) );
//MEMORY CLEAN UP NX0, NY0, NZ0.
CudaErrCheck( cudaFree(d_NX0) );
CudaErrCheck( cudaFree(d_NY0) );
CudaErrCheck( cudaFree(d_NZ0) );
//MEMORY CLEAN UP IntPX0, IntPY0, IntPZ0.
CudaErrCheck( cudaFree(d_IntPX0) );
CudaErrCheck( cudaFree(d_IntPY0) );
CudaErrCheck( cudaFree(d_IntPZ0) );
//MEMORY CLEAN UP TimeInt0.
CudaErrCheck( cudaFree(d_TimeInt0) );
//MEMORY CLEAN UP GHX0, GHY0, GHZ0.
CudaErrCheck( cudaFree(d_GHX0) );
CudaErrCheck( cudaFree(d_GHY0) );
CudaErrCheck( cudaFree(d_GHZ0) );
//MEMORY CLEAN UP GLX0, GLY0, GLZ0.
CudaErrCheck( cudaFree(d_GLX0) );
CudaErrCheck( cudaFree(d_GLY0) );
CudaErrCheck( cudaFree(d_GLZ0) );
//MEMORY CLEAN UP ComTemp0.
CudaErrCheck( cudaFree(d_ComTemp0) );
//MEMORY CLEAN UP ComTotal0.
CudaErrCheck( cudaFree(d_ComTotal0) );
//MEMORY CLEAN UP Index0.
CudaErrCheck( cudaFree(d_Index0) );
//cudaSetDevice(1);
//MEMORY CLEAN UP TCX1, TCY1, DistInt1.
CudaErrCheck( cudaFree(d_TCX1) );
CudaErrCheck( cudaFree(d_TCY1) );
CudaErrCheck( cudaFree(d_DistInt1) );
//MEMORY CLEAN UP BX1, BY1, Vel1.
CudaErrCheck( cudaFree(d_BX1) );
CudaErrCheck( cudaFree(d_BY1) );
CudaErrCheck( cudaFree(d_Vel1) );
//MEMORY CLEAN UP AX1, AY1, Thit1.
CudaErrCheck( cudaFree(d_AX1) );
CudaErrCheck( cudaFree(d_AY1) );
CudaErrCheck( cudaFree(d_Thit1) );
//MEMORY CLEAN UP NX1, NY1, NZ1.
CudaErrCheck( cudaFree(d_NX1) );
CudaErrCheck( cudaFree(d_NY1) );
CudaErrCheck( cudaFree(d_NZ1) );
//MEMORY CLEAN UP IntPX1, IntPY1, IntPZ1.
CudaErrCheck( cudaFree(d_IntPX1) );
CudaErrCheck( cudaFree(d_IntPY1) );
CudaErrCheck( cudaFree(d_IntPZ1) );
//MEMORY CLEAN UP TimeInt1.
CudaErrCheck( cudaFree(d_TimeInt1) );
//MEMORY CLEAN UP GHX1, GHY1, GHZ1.
CudaErrCheck( cudaFree(d_GHX1) );
CudaErrCheck( cudaFree(d_GHY1) );
CudaErrCheck( cudaFree(d_GHZ1) );
//MEMORY CLEAN UP GLX1, GLY1, GLZ1.
CudaErrCheck( cudaFree(d_GLX1) );
CudaErrCheck( cudaFree(d_GLY1) );
CudaErrCheck( cudaFree(d_GLZ1) );
//MEMORY CLEAN UP ComTemp1.
CudaErrCheck( cudaFree(d_ComTemp1) );
//MEMORY CLEAN UP ComTotal1.
CudaErrCheck( cudaFree(d_ComTotal1) );
//MEMORY CLEAN UP Index1.
CudaErrCheck( cudaFree(d_Index1) );
//---------------------------------------------------------------------//
//The closing of files used.
fclose(ParallelViolations);
fclose(ParallelExecutionTimes);
ParallelViolations = fopen("Parallel_Cuda_MultiGPU_Breadth_Violations.txt", "w");
//---------------------------------------------------------------//
}extérieur de
//---------------------------------------------------------------------//
//Freeing Up HOST (CPU) Pinned Memory
cudaSetDevice(0);
//MEMORY CLEAN UP TCX0, TCY0, TCZ0.
cudaFreeHost(h_TCX0);
cudaFreeHost(h_TCY0);
cudaFreeHost(h_TCZ0);
//MEMORY CLEAN UP BX0, BY0, BZ0.
cudaFreeHost(h_BX0);
cudaFreeHost(h_BY0);
cudaFreeHost(h_BZ0);
//MEMORY CLEAN UP AX0, AY0, AZ0.
cudaFreeHost(h_AX0);
cudaFreeHost(h_AY0);
cudaFreeHost(h_AZ0);
//MEMORY CLEAN UP NX0, NY0, NZ0.
cudaFreeHost(h_NX0);
cudaFreeHost(h_NY0);
cudaFreeHost(h_NZ0);
//MEMORY CLEAN UP GHX0, GHY0, GHZ0.
cudaFreeHost(h_GHX0);
cudaFreeHost(h_GHY0);
cudaFreeHost(h_GHZ0);
//MEMORY CLEAN UP GLX0, GLY0, GLZ0.
cudaFreeHost(h_GLX0);
cudaFreeHost(h_GLY0);
cudaFreeHost(h_GLZ0);
//MEMORY CLEAN UP IntPX0, IntPY0, IntPZ0.
cudaFreeHost(h_IntPX0);
cudaFreeHost(h_IntPY0);
cudaFreeHost(h_IntPZ0);
//MEMORY CLEAN UP DistInt0.
cudaFreeHost(h_DistInt0);
//MEMORY CLEAN UP Vel0.
cudaFreeHost(h_Vel0);
//MEMORY CLEAN UP UptimeInt0.
cudaFreeHost(h_TimeInt0);
cudaSetDevice(1);
//MEMORY CLEAN UP TCX1, TCY1, TCZ1.
cudaFreeHost(h_TCX1);
cudaFreeHost(h_TCY1);
cudaFreeHost(h_TCZ1);
//MEMORY CLEAN UP BX1, BY1, BZ1.
cudaFreeHost(h_BX1);
D.6 Asynchronous Multiple GPU Parallel Version Raw Data

500 50 41.649
1000 100 52.910
1500 150 52.811
2000 200 60.515
2500 250 74.915
3000 300 84.734
3500 350 107.298
4000 400 138.030
4500 450 166.065
5000 500 211.225
5500 550 244.546
6000 600 294.038
6500 650 244.546
7000 700 294.482
7500 750 355.460
8000 800 549.911
8500 850 555.846
9000 900 651.042
9500 950 707.872
10000 1000 848.150
10500 1050 827.540
11000 1100 1041.912
11500 1150 986.343
12000 1200 1213.081