The University of Texas at Dallas

Department of Electrical Engineering

EEDG 6304 COMPUTER ARCHITECTURE

GPU-CPU HYBRIDS FOR PERFORMANCE ACCELERATION OF MATHEMATICAL FUNCTIONS

Shanthi Rajsekaran (sxr157230)

Krunal Padhar (krp150630)

Prateek Sharma (pxs157230)

Devang Sankhala (dgs150030)

Contents

ABSTRACT

The GPU-CPU hybrids are designed as special hardware for real-time and high-definition 3D graphics. GPUs have evolved into many-core processors to accelerate highly parallel computations. The GPU is designed such that more transistors are devoted to processing cores rather than the sophisticated control hardware, and therefore able to address problems that can be represented as data-parallel computations. Many applications with data-parallelism can map data elements to processing threads. Such hybrid architectures can be utilized to provide immense computing power to non-graphical problems as well.

This work is a fundamental study of how can elementary mathematical functions be applied to large data sets to obtain acceleration with respect to the parameter of execution time. The results are compared with a CPU-only execution time and presented to provide an insight into how one can utilize parallel computing for engineering applications. In addition to this, design of parallel code is also discussed.

MOTIVATION TO USE HETEROGENEOUS ARCHITECTURE

Amdahl's law (Amdahl, 1967) (Hennessy & Patterson, 2011) is the mathematical basis for speedup of latency in computational theory. The primary focus of the former is to theoretically estimate the reduction in computational time for a given computational workload. The parallelizability of a computation depends on the algorithm chosen to perform the computation with minimum dependencies possible. Thus, if the parallelizability of the computation is known, one can establish the following result.

$$S_{\text{latency}}(s) = \frac{1}{1 - p + \frac{p}{s}}$$

Here, $\mathbf{S_{latency}}$ is the theoretical speedup in latency of the execution of the whole task. **s** is the speedup in latency of the execution of the part of the task that benefits from the improvement of the resources of the system. **p** is the percentage of the execution time of the whole task concerning the part that benefits from the improvement of the resources of the system before the improvement. The above relationship can also be established with regard to computational resources.

$$S_{latency} = \frac{N}{BN + (1 - B)}$$

Here, N = number of processors and B = percentage of code that is serial. Thus, for a lesser or no serial code, we get maximum speedup which is proportional to the number of processors a system has. A CPU may have 4-8 cores on a single die with a large cache memory. But, for a large throughput, a large number of processors are required. Thus, the GPU adds as a computational aid to the CPU to provide large throughput. Thus, to ensure maximum utilization of the GPU, parallel structure of code becomes a vital step to performance acceleration. Such a multiprocessor architecture is can be referred to as Single Instruction Multiple Data (SIMD) as per Flynn's taxonomy of computer architectures.

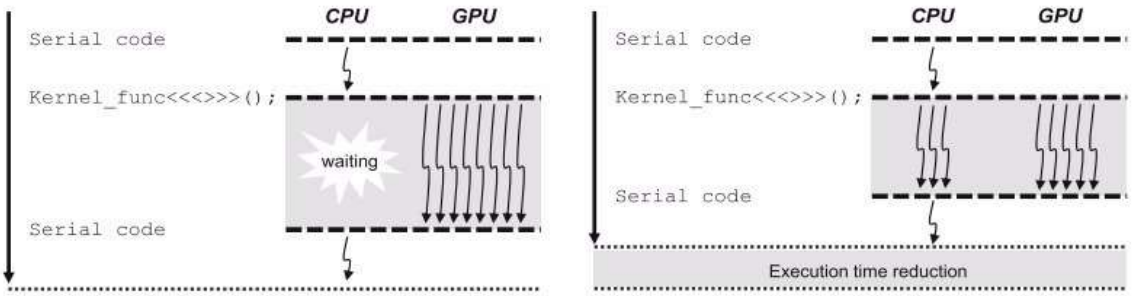


Figure 1 Comparison of serial and parallel code execution

THE CUDA API

The CUDA API is a parallel computing toolkit by NVidia. For this work, it has been used to exploit the parallel computing capabilities of an NVidia GPU for performance acceleration. The architecture of a two shared multiprocessor GPU is shown in figure 2.

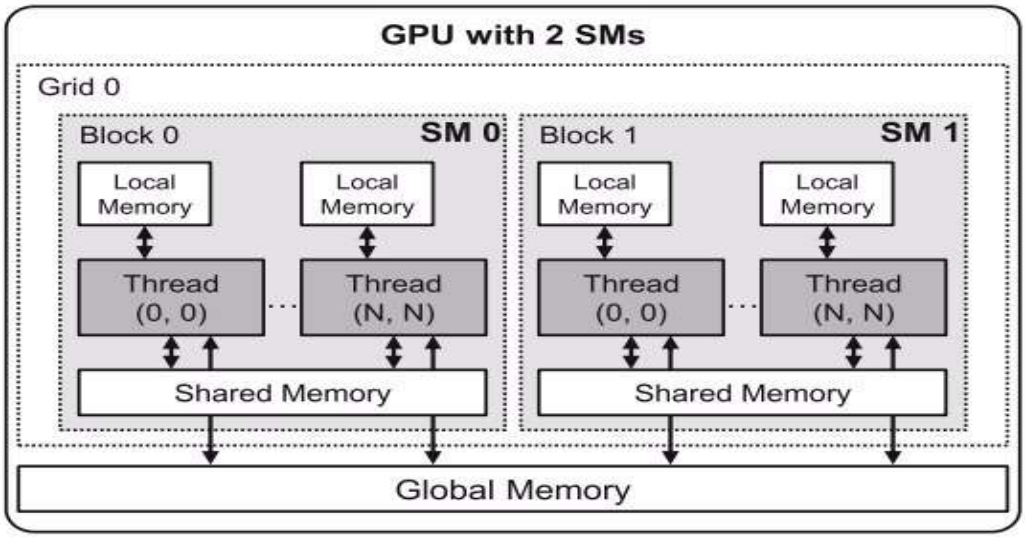


Figure 2 Two shared multiprocessor (SM) GPU architecture

It can be observed that each shared multiprocessor consists of local memory, shared memory and multiple processing cores. Thus, each shared multiprocessor is a miniaturized form of a CPU with multiple cores. Each core inside a shared multiprocessor is capable of running one "thread", which is termed as one instance of the parallel code. The more the number of cores in a multiprocessor, the more the number of threads can be processed. This in turn gives more throughput.

In ideal sense, CPU cores and GPU cores cooperate with each other (e.g., co-processing). They assume that a parallel portion of a code (i.e., CUDA kernels) runs on a GPU and a serial portion of a code (i.e., the rest of the C program) runs on a CPU. This heterogeneity across different processing units is a similar concept to that of the heterogeneous chip multiprocessor which exploits both large cores and small cores to address a much wider spectrum of system workloads.

Figure 3 NVidia GF100 Gaming GPU Architecture. It can be seen that this GPU has 16 shared multiprocessors with 32 cores per shared multiprocessor.

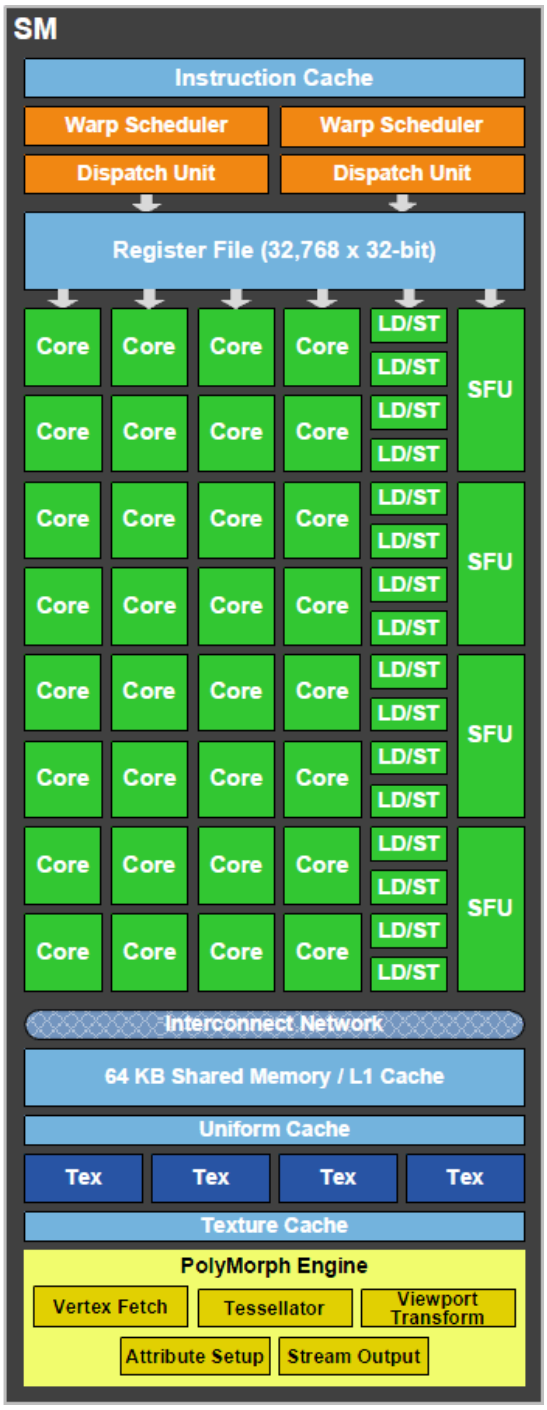


Figure 4 Architecture of a shared multiprocessor

In CUDA applications, the aim is to structure a certain portion of codes to expose data parallelism. Exploiting the massive parallelism with host threads is not a concern for the program design, so that CPU cores are held in the idle state even at runtime. Thus, the benchmarks assume the form such that CPU cores are not used for computation and for this study.

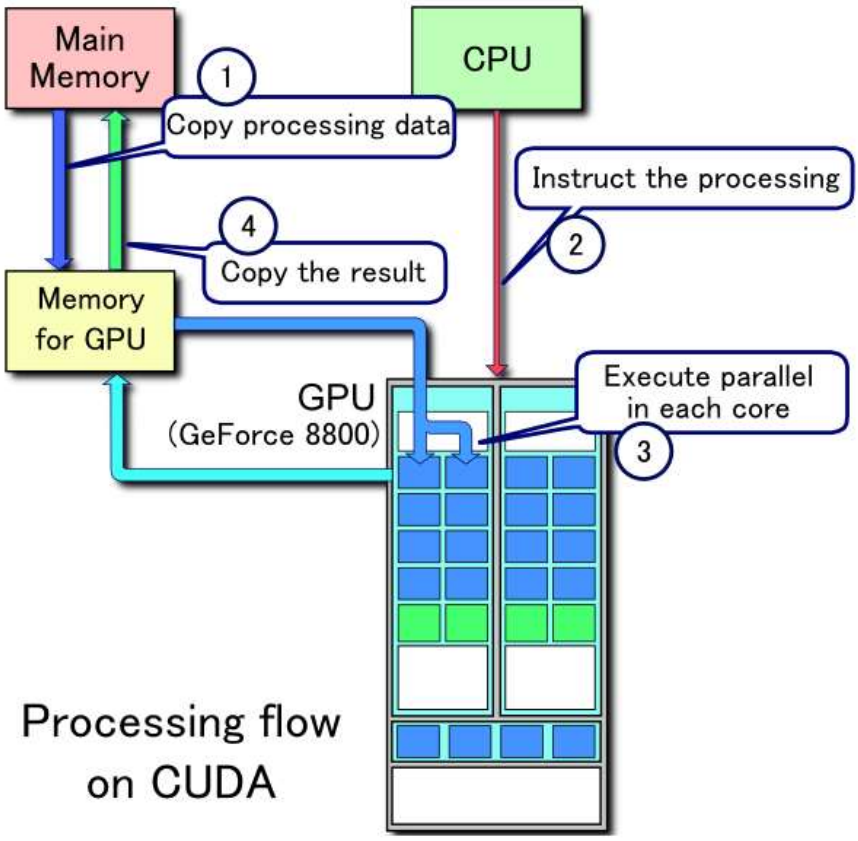


Figure 5 Process flow of the CUDA API

The parallel programs designed using CUDA work as show in figure 5. The data is copied to the GPU memory from the main memory. Then, a single instruction or a set of instructions called "kernel" is sent to the GPU to operate on the data set. This instruction is executed in parallel on the multiple cores of the GPU. Finally, the results are copied back to the main memory.

INDEXING USING GRID, BLOCK AND THREAD CONCEPT

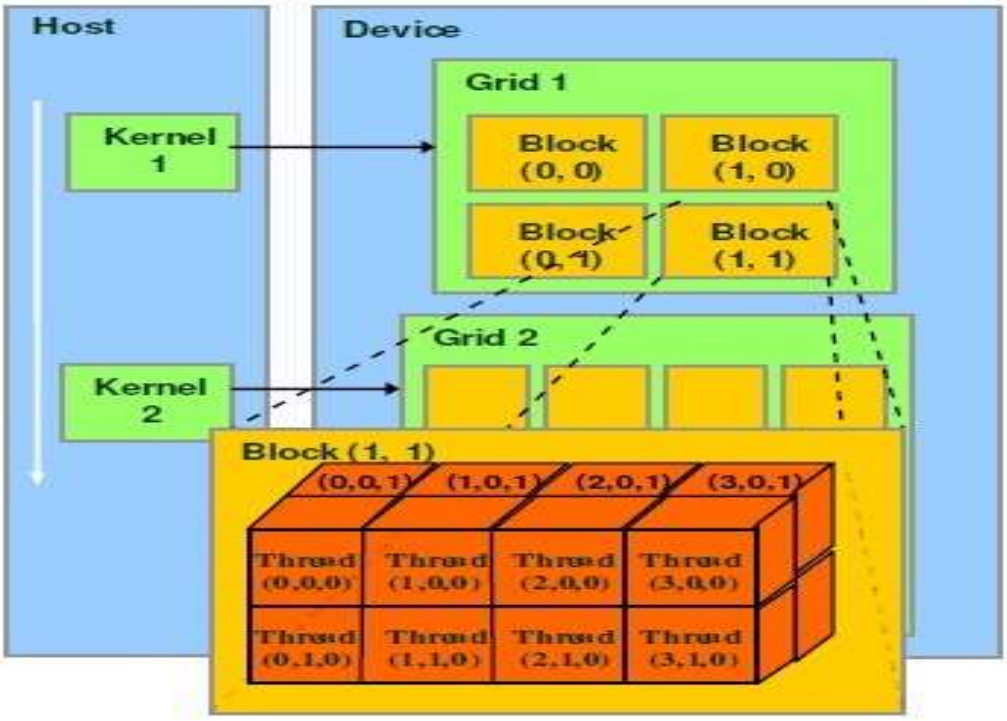


Figure 6 Grid, block and thread with respect to CUDA

It is clear from the architecture of a GPU that it contains a few shared multiprocessors, and these shared multiprocessors contain a large number of stream processors or cores. Each of the cores runs a parallel code on a given input element of the data set. The data passed to a GPU can be referenced as a grid, block or thread which are units of data with respect to the GPU architecture. A "grid" of data is the amount of data a GPU can hold in total. Thus, a GPU deal with data as multiple grids. Each shared multiprocessor can handle a certain amount of data to be processed. This amount of data is called a "block", so a shared multiprocessor can handle data as multiple blocks. Similarly, a "thread" of data is actually a replica of the parallel code running on a stream processor. Thus, a stream multiprocessor handles one or more threads. Thus, for an ideal condition, a thread can be considered as the unit element of data on the GPU. A thread may be operating on

one data element (e.g. square of number) or more data elements (e.g. vector addition). The unit element of data depends on the computation under consideration (NVIDIA Corporation, 2015).

One can also note that GPUs are capable for three-dimensional graphics rendering, thus threads can be three-dimensional. The same dimensions can apply to grids and blocks as well. For a non-graphic application, the dimensions of the grid, block and thread depends on the dimensions of the computation. Although, the programmer is free to modify the dimensional model to suit the needs of the parallel algorithm.

KEYWORDS AND TYPE QUALIFIERS IN CUDA

Type qualifiers define the data type. The type qualifiers of prime importance in CUDA are `__global__`, `__device__`, `__shared__` (NVIDIA Corporation, 2011). The `__global__` keyword is used for kernel definitions and signify kernels that can be called from the CPU. The `__device__` keyword signifies functions that can be called from inside a kernel on the device. They are not accessible from the CPU's serial code. `__shared__` keyword is used to access the shared memory between stream processors in a block for correlated computations.

Keywords define specific functions in a programming language for the compiler. Similarly for a hybrid architecture, the understanding of keywords helps define variables and functions specific to the device or the host processor. Keywords for indexing of threads are `threadIdx` and `blockIdx`. As threads can be three-dimensional, the three dimensions are accessed as `threadIdx.x`, `threadIdx.y` and `threadIdx.z`. The index of a thread is determined based on the number of the thread on a particular block in a particular grid.

DETERMINING DEVICE CAPABILITIES

To determine the maximum dimensions of the grid, block and thread, the device memory and number of processors are to be determined. This is done using the data structure `cudaDeviceProp`. The C code for the test device and the test device specifications are shown in Appendix A (NVIDIA Corporation, 2015).

BUILT-IN FUNCTIONS IN CUDA

A multitude of functions are required to implement the programming model shown in figure 5 which are explained below. For transfer of data from main memory to GPU memory, memory must be allocated on the GPU memory for the given data set to be transferred. This is done by the CUDA's memory management function `cudaMalloc`. Now, to exchange data, `cudaMemcpy` function is used. Based on the argument provided, it transfers data in either direction. Thus, this function creates a device copy for a data set already present on the host memory. The GPU operates on the device copy of the dataset. The same function returns the results back to the main memory.

For parallel execution of a function, a kernel is written. This kernel takes one data element from the large dataset and allocates it to a stream processor. But, to make sure that the data element is recognized correctly, a thread ID needs to be allocated to each data element. This thread ID generation depends on the dimensions of the thread, block and grid defined by the programmer. Device functions defined in Appendix B can be used to generate thread IDs for different programming models. Thus, the thread ID is generated in the kernel for each data element and sent to the stream processors dynamically allocated by CUDA.

WRITING A KERNEL

A kernel is a programming entity similar to a function in the parallel programming domain. Thus, a kernel call start parallel execution of code defined within it. An example of kernel call is shown below (NVIDIA Corporation, 2011).

```
1.    add << < NUMBER_OF_BLOCKS, THREADS_PER_BLOCK >> >(d_a, d_b, d_c);
```

Based on the programming model defined in figure 5, a block contains a certain number of threads. Thus, the data set can be defined into a regular number of blocks with a fixed number of threads. The relationship between the number of blocks, threads and total number of elements can be devised as below.

$$Total\ number\ of\ elements = Number\ of\ threads\ per\ block * Number\ of\ blocks\ per\ grid\ * Number\ of\ grids$$

Considering that the full grid is not utilized, the relationship can be simplified as below.

$$Total\ number\ of\ elements = Number\ of\ threads\ per\ block * Number\ of\ blocks$$

The number of threads per block can be set to maximum based on device capabilities. It should also be noted that all the threads give an output at once for a given block for a given instance of time. Thus, larger the number of threads per block, more is the throughput. Thus, these numbers are calculated based on the dataset size and provided as an argument to the kernel call.

An example of a global kernel is shown below.

```
1.  __global__ void add(float *d_a, float *d_b, float *d_c){
2.      int threadId = blockIdx.x *blockDim.x + threadIdx.x;
3.      d_c[threadId] = d_a[threadId] + d_b[threadId];
4.  }
```

Global kernels do not return values. Thus, to operate on a data set and get back results, one passes pointers of both, the input and output dataset. This kernel uses one dimensional thread indexing, thus the thread ID depends only on the X direction of the thread and block. An equivalent representation of the same can be seen below.

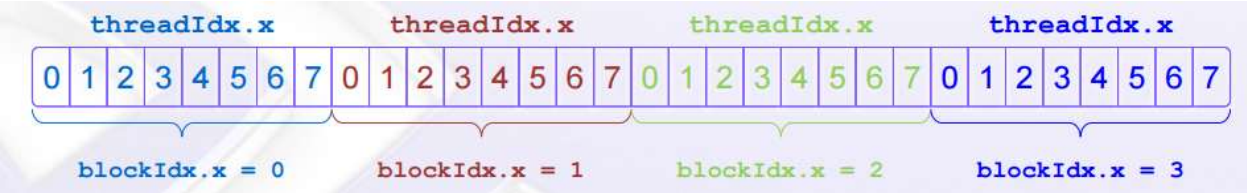


Figure 7 Interpretation of index using thread and block IDs

Consider an array of 32 elements. If it is declared that one block shall operate on 8 threads, we get 4 blocks. Thus, each block is designated a block ID. In addition to that, each element in a block is assigned a thread ID. Thus, for CUDA to assign a particular element to a thread, the index is assigned using the following relationship.

$$Index = threadIDx.x + blockIDx.x * blockDim.x$$

The above relationship can be verified using the following example.

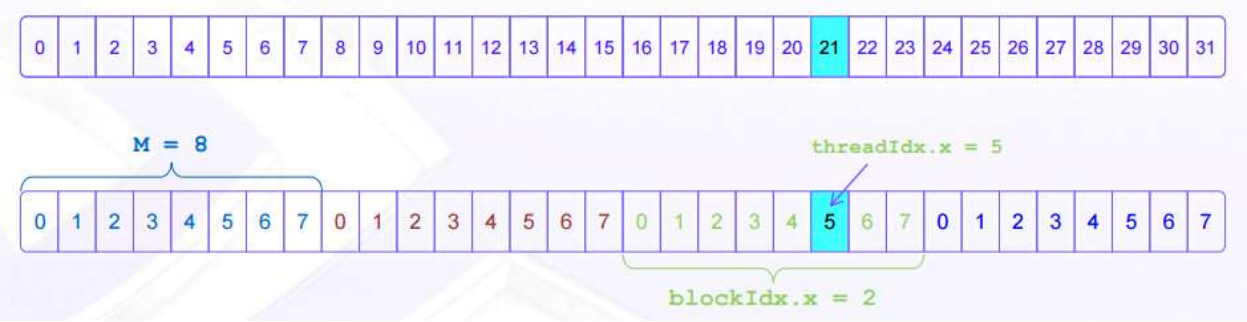


Figure 8 Locating the 22nd element on the dataset using thread and block ID

Consider that the 22nd element needs to be accessed from the given dataset. Thus, if thread ID = 5 and block ID = 2, we get the index = 21 from above equation, which points to element 22. However, the assumption is the program executes in a parallel manner, thus it is not known as to

which element will be processed first. The GPU may perform out of order execution for a faster throughput and process data based on the principle of locality. To enforce that all data is processed and available at a given point of time, some arrangement is necessary for synchronization of the combined throughput.

The `__syncthreads()` call is used to perform barrier synchronization of threads. Thus, all threads will wait at this call until all threads have reached this point. This is useful when data dependencies exist in a program, but is still massively parallelizable. The graphical representation of barrier synchronization is shown in figure 9.

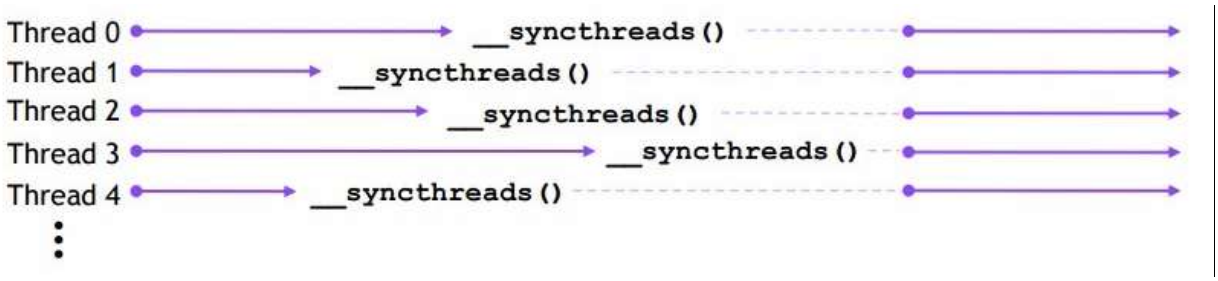


Figure 9 Barrier synchronization in kernel using __syncthreads() method

DESIGNING A PARALLEL PROGRAM

Designing a parallel equivalent of an existing code depends on the extent of parallelizability of the existing code. The following aspects are to be considered for the same:

1. Inherently parallelizability
2. Time and space complexity
3. Nature of data dependency in the algorithm
4. Size of the dataset
5. True minimum hardware and software requirement
6. Desired throughput of the program

Consider the following examples to understand the above aspects. Simple operations like vector addition, subtraction or multiplication require inputs from a given dataset. The computation for a given element does not depend on another element in this case. Thus, such an operation is inherently parallelizable. But, if one requires to perform sum of all elements of a dataset, then this becomes a partially parallelizable problem. This is because even if one thread can add two elements, each of the sums generated have to be reused for addition to generate cumulative results, which will converge towards the total sum. Thus, it can be seen that data dependency des exist in this case.

For the sake of time complexity, consider that a division operation is to be performed using floating point hardware. It is quite clear that division takes larger amount of time than multiplication on floating point hardware. Thus, to ensure better parallel program performance, division operations should be avoided. This can be done by either storing the inverse of a dataset beforehand if it is under frequent reuse. If there is no data reuse observed, an altogether different algorithm may be considered to minimize or totally eliminate such time complexity.

The space complexity of the program depends on the size of the dataset. The larger the dataset and more the dependency, the more is the space complexity. Vector operations require only two operands to be used for computation, thus the space complexity is twice the size of the dataset. But, considering Cramer's rule to solve a one-variable equation, one requires three elements, viz. the coefficient of the variable, and two constants on either side of the equality. Thus, the space complexity is thrice the dataset size. This also affect the true minimum system requirement for running the program, as it requires larger memory.

Size of the dataset and true minimum hardware and software requirements are interrelated. Larger the dataset, larger the memory required for large throughput. Also, larger the element data type, larger the allocation of memory is required, which should also be addressed correctly by the software.

Moreover, it is clear from the hybrid architecture that one consider the overhead time required to transfer the data between the CPU and the GPU before and after computation. Thus, if the dataset is small, using a hybrid architecture may not yield expected results due to the overhead involved in the exchange of data. The threshold of the minimum dataset can be determined by experimenting with the hardware by running benchmarks. Design of benchmarks to reveal data parallelism are discussed below.

ACCELERATION IN ONE DIMENSION

A GPU is a hardware that is designed to provide three dimensional throughput in modern computing scenario. Thus, threads can process three dimensional data elements at a time in a GPU. However, considering a single dimensional large dataset, a benchmark for vector squaring as in Appendix C is written and tested for one dimensional computation in CUDA C for 64E+6 elements. The benchmark works in the following manner:

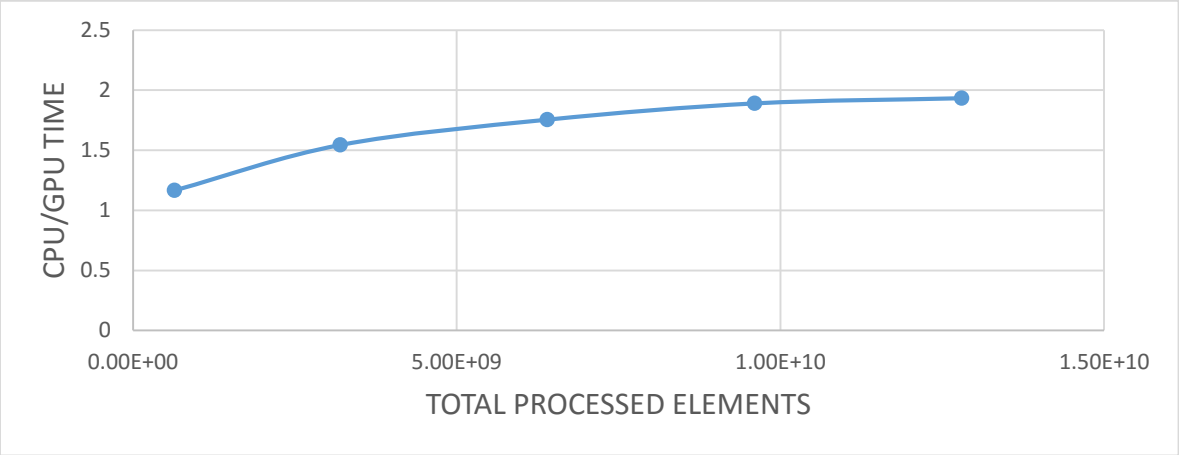
1. Define a dataset using random number generation
2. Operate on the dataset using parallel programming and note execution time
3. Operate on the dataset using serial code and note execution time

The results for this benchmark are tabulated in table 1 and a plot is generated for as in figure 10.

Table 1 Results for benchmark in Appendix C

NUMBER OF LOOPS	GPU TIME	CPU TIME	CPU/GPU TIME	TOTAL PROCESSED ELEMENTS
10	3.75	4.377	1.1672	6.40E+08
50	7.417	11.459	1.544964271	3.20E+09
100	12.318	21.631	1.75604806	6.40E+09
150	17.043	32.236	1.891451036	9.60E+09
200	21.72	42.016	1.934438306	1.28E+10

Table 2 Plot of CPU/GPU time and total number of elements processed per run from Table 1



It is evident that the acceleration results are not satisfactory as observed in Table 1 due to inefficient use of hardware. Thus, it would be required to maximize hardware usage by mapping three uncorrelated elements to a thread to get thrice more throughput theoretically.

ACCELERATION IN THREE DIMENSIONS

Now, to map one dimensional data to three dimensional threads, three dimensional indexing scheme is used. The indexing syntax is as below.

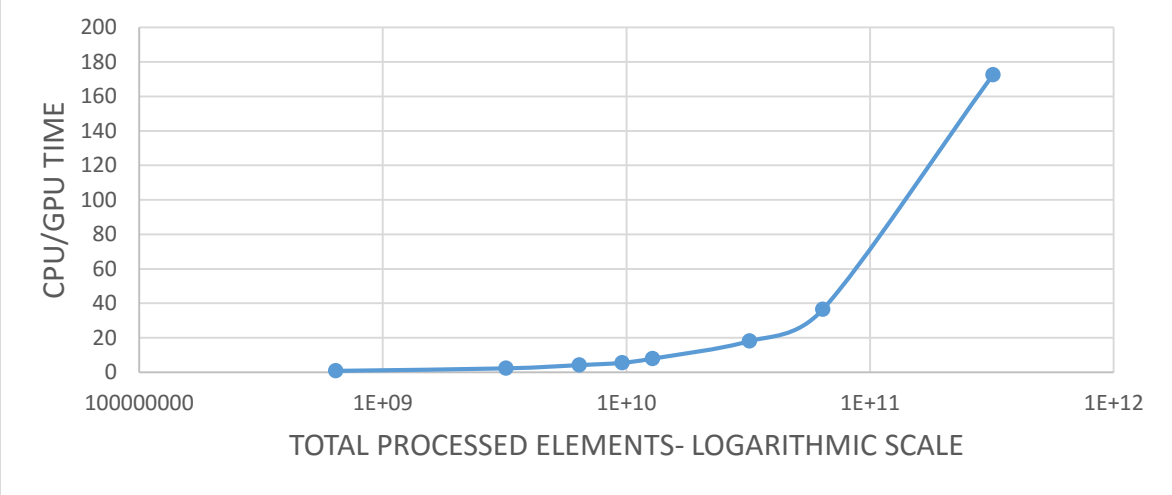
```
1.  int blockId = blockIdx.x
2.                + blockIdx.y * gridDim.x
3.                + gridDim.x * gridDim.y * blockIdx.z;
4.      int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
5.                  + (threadIdx.z * (blockDim.x * blockDim.y))
6.                  + (threadIdx.y * blockDim.x)
7.                  + threadIdx.x;
```

The first declaration is used to identify the three dimensional block on which the kernel will operate. The second declaration gives the location of the element in the three dimensional block. Thus, the benchmark in Appendix D uses the above indexing scheme. The results are shown in tables 3 and 4.

Table 3 Results from benchmark in Appendix D

LOOPS	GPU TIME	CPU TIME	CPU/GPU TIME	TOTAL PROCESSED ELEMENTS	FLOPS/S	GFLOPS/S
10	4.9	4.33	0.883673469	6.40E+08	1.31E+08	1.31E-01
50	6	14	2.333333333	3.20E+09	5.33E+08	5.33E-01
100	6.57	27.7	4.216133942	6.40E+09	9.74E+08	9.74E-01
150	7.23	40	5.532503458	9.60E+09	1.33E+09	1.33E+00
200	6.74	53.68	7.964391691	1.28E+10	1.90E+09	1.90E+00
500	7	127	18.14285714	3.20E+10	4.57E+09	4.57E+00
1000	6.98	255	36.53295129	6.40E+10	9.17E+09	9.17E+00
5000	7.193	1242	172.6678715	3.20E+11	4.45E+10	4.45E+01

Table 4 Plot of CPU/GPU time and total number of elements processed per run from Table 3



Thus, from table 4, we can observe that three-dimensional indexing makes more efficient use of hardware and provides considerable speedup in computation time. Similar indexing scheme was utilized to perform vector addition as in Appendix E. But, the number of threads per block was

changed to observe the change in acceleration. From the device properties as in Appendix A, it can be noted that the test device has 3 shared multiprocessors with a maximum thread capacity of 1024 per shared multiprocessor. Thus, the maximum number of threads per block is thrice of 1024 which is 3072. However, the threads per block considered here were 1024, 2048 and 4096.

Table 5 Results from benchmark in Appendix E for 1024 threads per block

LOOPS	GPU TIME	CPU TIME	CPU/GPU TIME	TOTAL PROCESSED ELEMENTS	FLOPS/S	GFLOPS/S
10	6.5	5.11	0.786153846	6.40E+08	9.85E+07	9.85E-02
50	8.52	13	1.525821596	3.20E+09	3.76E+08	3.76E-01
100	8.7	27.6	3.172413793	6.40E+09	7.36E+08	7.36E-01
150	8.4	38.76	4.614285714	9.60E+09	1.14E+09	1.14E+00
200	8.5	43.8	5.152941176	1.28E+10	1.51E+09	1.51E+00
500	6.5	106	16.30769231	3.20E+10	4.92E+09	4.92E+00
1000	8.6	206	23.95348837	6.40E+10	7.44E+09	7.44E+00
5000	8.45	1070	126.6272189	3.20E+11	3.79E+10	3.79E+01

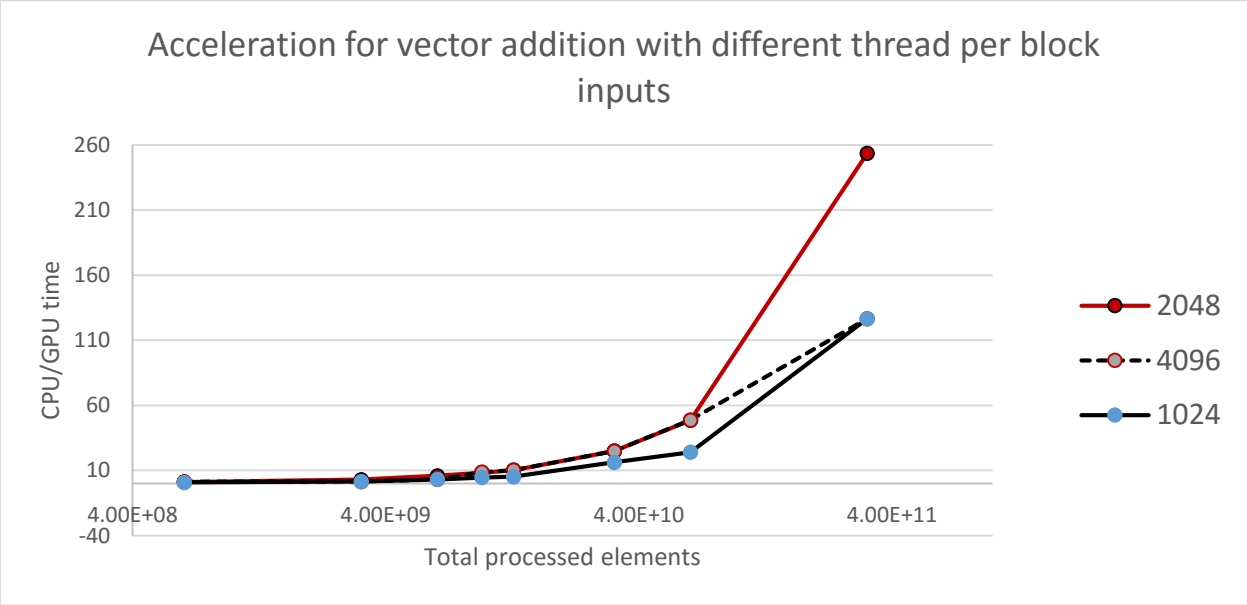
Table 6 Results from benchmark in Appendix E for 2048 threads per block

LOOPS	GPU TIME	CPU TIME	CPU/GPU TIME	TOTAL PROCESSED ELEMENTS	FLOPS/S	GFLOPS/S
10	4.5	5.11	1.13555555	6.40E+08	1.42E+08	1.42E-01
50	4.3	13	3.02325581	3.20E+09	7.44E+08	7.44E-01
100	4.6	27.6	6	6.40E+09	1.39E+09	1.39E+00
150	4.67	38.76	8.29978586	9.60E+09	2.06E+09	2.06E+00
200	4.3	43.8	10.18604651	1.28E+10	2.98E+09	2.98E+00
500	4.24	106	25	3.20E+10	7.55E+09	7.55E+00
1000	4.24	206	48.58490566	6.40E+10	1.51E+10	1.51E+01
5000	4.22	1070	253.5545024	3.20E+11	7.58E+10	7.58E+01

Table 7 Results from benchmark in Appendix E for 4096 threads per block

LOOPS	GPU TIME	CPU TIME	CPU/GPU TIME	TOTAL PROCESSED ELEMENTS	FLOPS/S	GFLOPS/S
10	4.1	5.11	1.246341463	6.40E+08	1.56E+08	1.56E-01
50	8.52	13	1.525821596	3.20E+09	3.76E+08	3.76E-01
100	8.7	27.6	3.172413793	6.40E+09	7.36E+08	7.36E-01
150	4.4	38.76	8.809090909	9.60E+09	2.18E+09	2.18E+00
200	4.3	43.8	10.18604651	1.28E+10	2.98E+09	2.98E+00
500	4.27	106	24.82435597	3.20E+10	7.49E+09	7.49E+00
1000	4.23	206	48.69976359	6.40E+10	1.51E+10	1.51E+01
5000	8.45	1070	126.6272189	3.20E+11	3.79E+10	3.79E+01

Table 8 Cumulative comparison of results from tables 5, 6 and 7



It is evident from table 8 that acceleration increases as we increase the number of threads per block and maximum acceleration is obtained when closing towards the maximum device thread per block capacity. However, declaring more number of threads per block leads to decrease in performance acceleration.

In addition to above results, random numbers were generated using CUDA's random number generation library as in Appendix F. The results are obtained show the highest among all benchmark runs in this study.

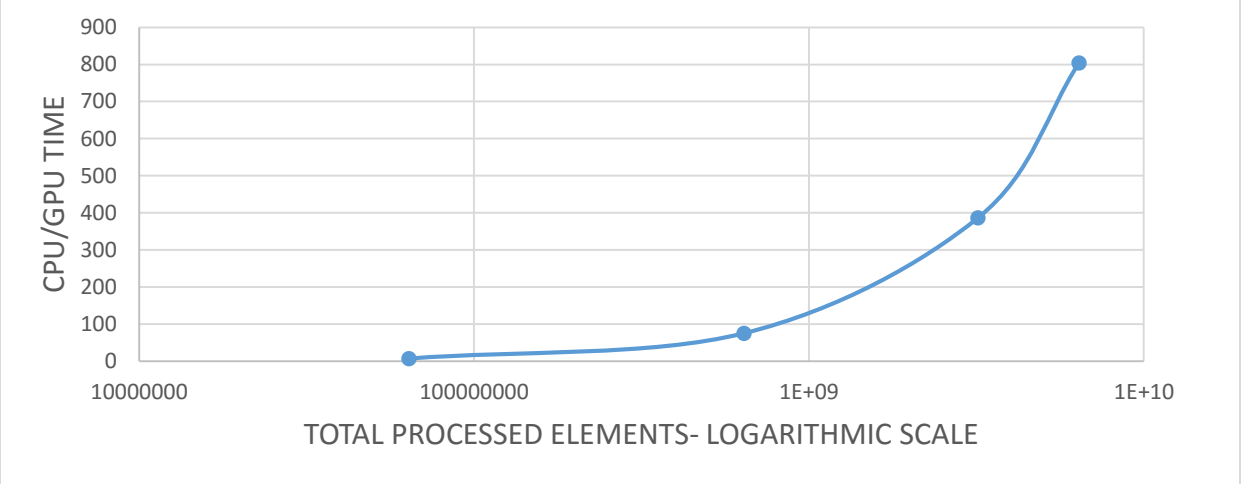
Table 9

Table 10 Results from benchmark in Appendix G

Results from benchmark in Appendix F

LOOPS	GPU TIME	CPU TIME	CPU/GPU TIME	TOTAL PROCESSED ELEMENTS	FLOPS/S	GFLOPS/S
1	0.26	1.79	6.884615385	6.40E+07	2.46E+08	2.46E-01
10	0.28	21	75	6.40E+08	2.29E+09	2.29E+00
50	0.27	104.4	386.6666667	3.20E+09	1.19E+10	1.19E+01
100	0.26	209	803.8461538	6.40E+09	2.46E+10	2.46E+01

Table 11 Plot of CPU/GPU time and total number of elements processed per run from Table 10



ONE-VARIABLE EQUATION SOLVER

The benchmarks discussed in the previous sections are used to study the performance of elementary FLP functions. But, real world problems are a combination of all the above discussed effects of program design. The most dominant and fundamental engineering problem solved using computers is equations. The number of variables increase with the complexity of design and so

does the execution time. Thus, to reduce the execution time of such heavy workloads, GPUs can act as an aid to the CPU.

Considering a one variable equation as below and its solution. Considering all operations are floating point type, one can estimate that an addition or subtraction would take the same amount of time, but division would take a fairly large amount of time than addition, subtraction or even multiplication. Thus, solving equations on a parallel program using this approach is highly inefficient.

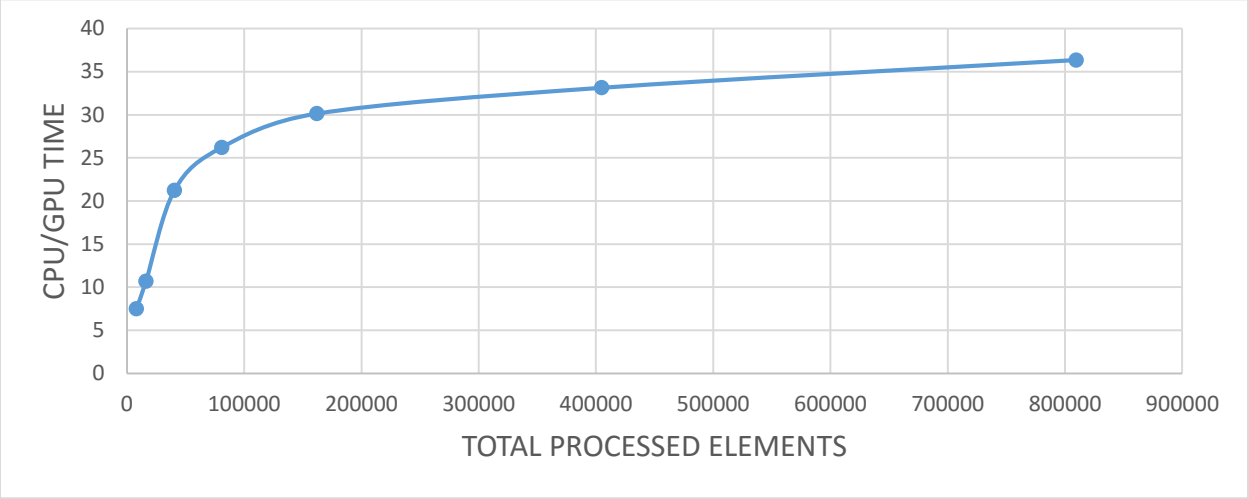
$$ax + b = c \rightarrow x = \frac{c - b}{a}$$

The benchmark in Appendix G is a one variable equation solver of the above form. Thus, it performs one subtraction and one division to obtain the solution. The results are tabulated below and there is scope of improvement for the same. Thus, one can utilize a different approach or algorithm which eliminates the need for division for equation solving (Jan Verschelde) (Jonathan M. Cohen, 2012) (Taek-Jun Kwon, 2008). Such an approach would yield performance results comparable to the benchmarks for elementary floating point operations.

Table 12 Results from benchmark in Appendix G

LOOPS	GPU TIME	CPU TIME	CPU/GPU TIME	TOTAL PROCESSED ELEMENTS	FLOPS/S	KFLOPS/S
1	0.7	5.25	7.5	8.10E+03	1.16E+04	1.16E+01
2	0.9	9.6	10.66666667	1.62E+04	1.80E+04	1.80E+01
5	1.3	27.6	21.23076923	4.05E+04	3.11E+04	3.11E+01
10	2	52.43	26.215	8.10E+04	4.05E+04	4.05E+01
20	3.53	106.35	30.12747875	1.62E+05	4.59E+04	4.59E+01
50	8	265	33.125	4.05E+05	5.06E+04	5.06E+01
100	15.6	567	36.34615385	8.10E+05	5.19E+04	5.19E+01

Table 13 Plot of CPU/GPU time and total number of elements processed per run from Table 12



TWO-VARIABLE EQUATION SOLVER

An approach similar to the above benchmark is used to solve two variable equation systems using Cramer's approach. The Cramer's approach of two variable equation solver is described using below equations.

$$a_1 x + b_1 y = c_1$$

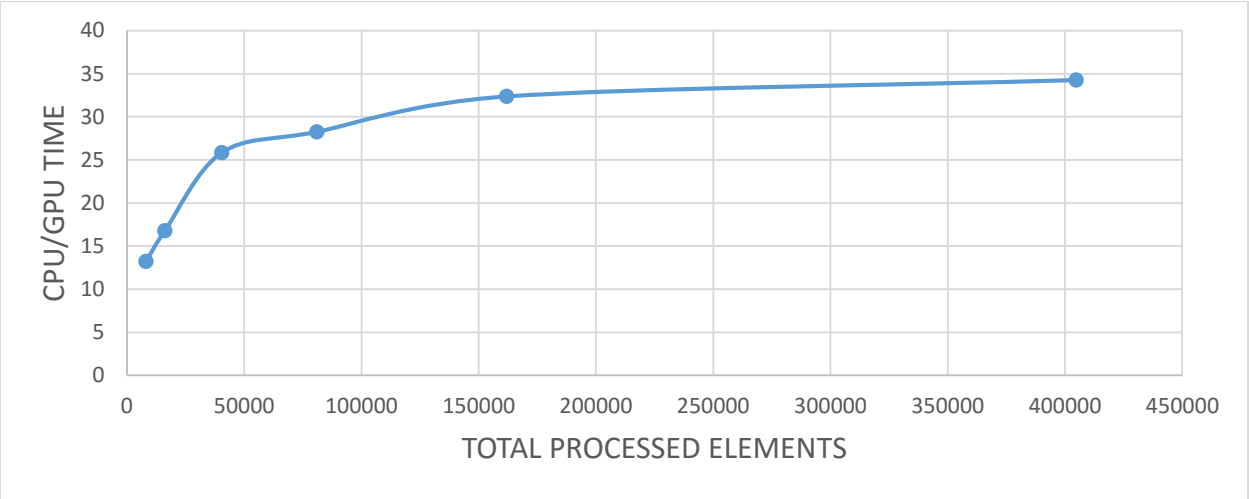
$$a_2 x + b_2 y = c_2$$

$$x = \frac{c_1 b_2 - c_2 b_1}{a_1 b_2 - a_2 b_1} \quad \text{and} \quad y = \frac{a_1 c_2 - a_2 c_1}{a_1 b_2 - a_2 b_1}$$

Table 14 Results from benchmark in Appendix H

LOOPS	GPU TIME	CPU TIME	CPU/GPU TIME	TOTAL PROCESSED ELEMENTS	FLOPS/S	GFLOPS/S
1	0.82	10.86	13.24390244	8.10E+03	9.87E+03	9.87E-06
2	1.13	18.99	16.80530973	1.62E+04	1.43E+04	1.43E-05
5	2	51.7	25.85	4.05E+04	2.02E+04	2.02E-05
10	3.56	100.6	28.25842697	8.10E+04	2.27E+04	2.27E-05
20	6.55	212	32.36641221	1.62E+05	2.47E+04	2.47E-05

Table 15 Plot of CPU/GPU time and total number of elements processed per run from Table 14



APPLICATIONS OF HETEROGENOUS COMPUTING ARCHITECHTURE

The applications of heterogeneous computing for artwork, entertainment and gaming is widely known. Although, the applications of heterogeneous parallel computing in science, medicine, engineering design and military are quite vast. The most noted example of parallel computing in space physics is the n-body simulation. This simulation is used to study the effect Newtonian forces of n number of bodies in space and how this force makes the bodies move towards a given point. Here, the number of combinations of forces between the bodies depends on the number of bodies. Thus, as the number of bodies increases, so does the time and space complexity. Thus, one can use a heterogeneous architecture to accelerate this simulation as well as display it on the screen (NVIDIA Corporation, 2007).

Parallel computing can be used to accelerate MRI and CAT scans for patients with a sever disease who regularly have to undergo diagnosis. Since scanning technologies use high power penetrating

waves, this can be harmful for the patient's body. Using such an accelerated imaging system can reduce the exposure time significantly.

Pertaining to engineering design, parallel computing helps solve multi-variable equations for multi-physical systems. Electrical, mechanical and chemical properties for a system can be simulated at once for faster design. In addition to that, attempts have been made to accelerate design automation of semiconductor chips to reduce execution time for synthesis of digital design, placement and route and circuit partitioning.

Military applications of heterogeneous computing encompass engineering, communication and mathematical problems with time-critical importance. Increasing the complexity of cipher using a longer passkey makes it difficult to decrypt using brute force methods. Thus, important information can be transmitted lest the decryption by the enemy. Accurate engineering simulations for aircraft and nuclear and non-nuclear missile design can be performed. In addition, casualty analysis for a nuclear or biological weapon outbreak and preventive measure analysis can be accelerated using the same.

SUMMARY

This work proposed an insight into parallel programming using CUDA for acceleration of mathematical functions. The work also gives an insight about creating benchmarks specifically to test the capability of a device for a given algorithm. Testing the capabilities of the device and the algorithm can help make decisions as to what approach would provide maximum acceleration for a given application. In addition, results have been presented for a test device for elementary floating point operations and Cramer's rule equations solvers. By observation, a precise design of

parallel code in the bounds of the test device, a fairly large dataset and avoiding division operation can immensely accelerate the application.

REFERENCES

1. Amdahl, G. (1967). Validity of the single processor approach to achieving large scale. *American Federation of Information Processing Societies*.

2. Hennessy, J. L., & Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann.

3. Jan Verschelde, X. Y. (n.d.). *GPU acceleration of Newton's method for large systems of polynomial equations in double double and quad double arithmetic.* Retrieved from http://homepages.math.uic.edu/~jan/gpunewton2.pdf

4. Jonathan M. Cohen, M. J. (2012, May 14). *A Fast Double Precision CFD Code using CUDA* . Retrieved from http://www.nvidia.com/content/cudazone/cudabrowser/downloads/papers/doubleprecision-cfd-cohen-parcfd09.pdf

5. Michels, D. (2011). Sparse-Matrix-CG-Solver in CUDA. *The 15th Central European Seminar on Computer Graphics.* Bonn: Central European Seminar on Computer Graphics for students. Retrieved from http://www.cescg.org/CESCG-2011/papers/Bonn-Michels-Dominik.pdf

6. NVIDIA Corporation. (2011). *CUDA C/C++ Basics.* Retrieved from http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf

7. NVIDIA Corporation. (2015, September 1). *CUDA C Programming Guide.* Retrieved from CUDA Toolkit Documentation: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

8. Taek-Jun Kwon, J. S. (2008, February 24). *Floating-Point Division and Square Root using a Taylor-Series Expansion Algorithm* . Retrieved from http://www.isi.edu/~draper/papers/mwscas07_kwon.pdf

APPENDIX A: TEST DEVICE PROPERTIES

The devices properties were procured using the following code.

```
1.  // Device properties report generator
2.  #include <stdio.h>
3.  #include <conio.h>
4.  #include "cuda_runtime.h"
5.  #include "device_launch_parameters.h"
6.
7.  int main() {
8.          //for multiple devices
9.          int nDevices;
10.         //count available devices
11.         cudaGetDeviceCount(&nDevices);
12.         for (int xx = 0; xx < nDevices; xx++) {
13.             cudaDeviceProp prop;
14.             cudaGetDeviceProperties(&prop, xx);
15.             printf("Device Number: %d\n", xx);
16.             printf("    Device name: %s\n", prop.name);
17.             printf("    Memory Clock Rate (KHz): %d\n",
18.                 prop.memoryClockRate);
19.             printf("    Memory Bus Width (bits): %d\n",
20.                 prop.memoryBusWidth);
21.             printf("    Peak Memory Bandwidth (GB/s): %f\n",
22.                 2.0*prop.memoryClockRate*(prop.memoryBusWidth / 8) / 1.0e6);
23.             printf("    Major revision number:         %d\n", prop.major);
24.             printf("    Minor revision number:         %d\n", prop.minor);
25.             printf("    Total global memory:           %u", prop.totalGlobalMem);
26.             printf(" bytes\n");
27.             printf("    Number of multiprocessors:     %d\n", prop.multiProcessorCount)
    ;
28.             printf("    Total amount of shared memory per block: %u\n", prop.sharedMemP
    erBlock);
29.             printf("    Total registers per block:     %d\n", prop.regsPerBlock);
30.             printf("    Warp size:                     %d\n", prop.warpSize);
31.             printf("    Maximum memory pitch:          %u\n", prop.memPitch);
32.             printf("    Maximum grid size:          %u\n", prop.maxGridSize);
33.             printf("    Maximum thread dimension:          %u\n", prop.maxThreadsDim);

34.             printf("    Maximum threads per block:          %u\n", prop.maxThreadsPerBl
    ock);
35.             printf("    Total amount of constant memory:         %u\n", prop.totalConst
    Mem);
36.             int cores = 0;
37.             int mp = prop.multiProcessorCount;
38.             switch (prop.major){
39.             case 2: // Fermi
40.                 if (prop.minor == 1) cores = mp * 48;
41.                 else cores = mp * 32;
42.                 break;
43.             case 3: // Kepler
44.                 cores = mp * 192;
45.                 break;
46.             case 5: // Maxwell
47.                 cores = mp * 128;
48.                 break;
49.             default:
50.                 printf("Unknown device type\n");
```

```
51.                 break;
52.             }
53.             printf("    Number of cores:        %d\n\n",  cores);
54.         }
55.         getch();
56.         return 0;
57. }
```

The output generated was as below.

```
Device Number: 0
Device name: GeForce 940M
Memory Clock Rate (KHz): 900000
Memory Bus Width (bits): 64
Peak Memory Bandwidth (GB/s): 14.400000
Major revision number:          5
Minor revision number:          0
Total global memory:            2147483648 bytes
Number of multiprocessors:      3
Total amount of shared memory per block:        49152
Total registers per block:      65536
Warp size:                      32
Maximum memory pitch:           2147483647
Maximum grid size:          18020092
Maximum thread dimension:           18020080
Maximum threads per block:           1024
Total amount of constant memory:          65536
Number of cores:          384
```

APPENDIX B: INDEXING FUNCTION PROTOTYPES

```
8.  //1D grid of 1D blocks
9.  __device__ int getGlobalIdx_1D_1D()
10. {
11.     return blockIdx.x *blockDim.x + threadIdx.x;
12. }
13.
14. //1D grid of 2D blocks
15. __device__ int getGlobalIdx_1D_2D()
16. {
17.     return blockIdx.x * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.
    x;
18. }
19.
20. //1D grid of 3D blocks
21. __device__ int getGlobalIdx_1D_3D()
22. {
23.     return blockIdx.x * blockDim.x * blockDim.y * blockDim.z
24.      + threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;
```

```
25. }
26.
27. //2D grid of 1D blocks
28. __device__ int getGlobalIdx_2D_1D()
29. {
30.     int blockId   = blockIdx.y * gridDim.x + blockIdx.x;
31.     int threadId = blockId * blockDim.x + threadIdx.x;
32.     return threadId;
33. }
34.
35. //2D grid of 2D blocks
36.  __device__ int getGlobalIdx_2D_2D()
37. {
38.     int blockId = blockIdx.x + blockIdx.y * gridDim.x;
39.     int threadId = blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) + t
    hreadIdx.x;
40.     return threadId;
41. }
42.
43. ///2D grid of 3D blocks
44. __device__ int getGlobalIdx_2D_3D()
45. {
46.     int blockId = blockIdx.x
47.               + blockIdx.y * gridDim.x;
48.     int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
49.                  + (threadIdx.z * (blockDim.x * blockDim.y))
50.                  + (threadIdx.y * blockDim.x)
51.                  + threadIdx.x;
52.     return threadId;
53. }
54.
55. //3D grid of 1D blocks
56. __device__ int getGlobalIdx_3D_1D()
57. {
58.     int blockId = blockIdx.x
59.               + blockIdx.y * gridDim.x
60.               + gridDim.x * gridDim.y * blockIdx.z;
61.     int threadId = blockId * blockDim.x + threadIdx.x;
62.     return threadId;
63. }
64.
65. ///3D grid of 2D blocks
66. __device__ int getGlobalIdx_3D_2D()
67. {
68.     int blockId = blockIdx.x
69.                   + blockIdx.y * gridDim.x
70.               + gridDim.x * gridDim.y * blockIdx.z;
71.     int threadId = blockId * (blockDim.x * blockDim.y)
72.                + (threadIdx.y * blockDim.x)
73.                + threadIdx.x;
74.     return threadId;
75. }
76.
77. //3D grid of 3D blocks
78. __device__ int getGlobalIdx_3D_3D()
79. {
80.     int blockId = blockIdx.x
81.               + blockIdx.y * gridDim.x
82.               + gridDim.x * gridDim.y * blockIdx.z;
```

```
    int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
              + (threadIdx.z * (blockDim.x * blockDim.y))
              + (threadIdx.y * blockDim.x)
              + threadIdx.x;
    return threadId;
}
```

APPENDIX C: BENCHMARK FOR ONE DIMENSIONAL VECTOR SQUARING

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <time.h>
#define N (8096*8096)
// keep threads per block a multiple of 32
#define THREADS_PER_BLOCK 1024
#define LOOPS 100

// a function on the device for the threads to use
__device__ float squared(float x) {
    return x*x;
}

//kernel 1D grid of 1D blocks
__global__ void square(float *d_a, float *d_c) {
    int threadId = blockId * blockDim.x  +threadIdx.x;
    d_c[threadId] = squared(d_a[threadId]);
}


//////////////////////////////////////////////////
void random_floats(float* a, int x)
{
    int  j;
    for (j = 0; j < x; ++j)
        a[j] = (float)rand();
}

int main(void) {
    long i; //counter
    float *a, *c; // host copies of a, b, c
    float *d_a, *d_c; // device copies of a, b, c
    long int size = N * sizeof(float);
    clock_t start, end;

    _beep(2000, 500);
    printf("\n %d \n", N*LOOPS);

    //with CUDA

    start = clock();

        // Alloc space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        //cudaMalloc((void **)&d_b, size);
```

```
50.         cudaMalloc((void **)&d_c, size);
51.         // Alloc space for host copies of a, b, c and setup input values
52.         a = (float *)malloc(size); random_floats(a, N);
53.         //b = (float *)malloc(size); random_floats(b, N);
54.         c = (float *)malloc(size);
55.         // Copy inputs to device
56.         cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
57.         //cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
58.
59.         // <<<B,T>>>, B = no of blocks, T = number of threads
60.
61.
62.         // start code here
63.         for (int A = 0; A < LOOPS; A++){
64.             square << < N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(d_a, d_c);
65.         }
66.         // Copy result back to host
67.         cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
68.
69.         //end clock here
70.         end = clock();
71.
72.     printf("\n %f seconds with CUDA", (double)(end - start) / CLOCKS_PER_SEC);
73.     _beep(2000, 500);
74.     // Cleanup
75.     free(a);
76.     //free(b);
77.     free(c);
78.     cudaFree(d_a);
79.     //cudaFree(d_b);
80.     cudaFree(d_c);
81.
82.     //without CUDA
83.     start = clock();
84.     // start code here
85.
86.     // Alloc space for host copies of a, b, c and setup input values
87.     a = (float *)malloc(size); random_floats(a, N);
88.     //b = (float *)malloc(size); random_floats(b, N);
89.     c = (float *)malloc(size);
90.
91.     for (int k = 0; k < LOOPS; k++)
92.     {
93.         for (i = 0; i < N; i++)
94.         {
95.             c[i] = a[i] * a[i];
96.             //printf("\n %l -> %f * %f = %f", i, a[i], a[i], c[i]);
97.         }
98.     }
99.
100.           // end of code
101.           end = clock();
102.           printf("\n %f seconds without CUDA", (double)(end - start) / CLOCKS_PER_SEC)
    ;
103.           //
104.           // Cleanup
105.           free(a);
106.           //free(b);
107.           free(c);
```

```
108.            _beep(2000, 500);
109.            getch();
110.            return 0;
111.        }
```

APPENDIX D: BENCHMARK FOR THREE DIMENSIONAL VECTOR SQUARING

```
1.  #include "cuda_runtime.h"
2.  #include "device_launch_parameters.h"
3.  #include <stdio.h>
4.  #include <conio.h>
5.  #include <stdlib.h>
6.  #include <malloc.h>
7.  #include <time.h>
8.  #define N (8096*8096)
9.  // keep threads per block a multiple of 32
10. #define THREADS_PER_BLOCK 1024
11. #define LOOPS 100
12.
13. // a function on the device for the threads to use
14. __device__ float squared(float x) {
15.     return x*x;
16. }
17.
18. //kernel 3D grid of 3D blocks
19. __global__ void square(float *d_a, float *d_c) {
20.     int blockId = blockIdx.x
21.         + blockIdx.y * gridDim.x
22.         + gridDim.x * gridDim.y * blockIdx.z;
23.     int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
24.         + (threadIdx.z * (blockDim.x * blockDim.y))
25.         + (threadIdx.y * blockDim.x)
26.         + threadIdx.x;
27.     d_c[threadId] = squared(d_a[threadId]);
28. }
29.
30.
31. ////////////////////////////////////////////////////
32. void random_floats(float* a, int x)
33. {
34.     int  j;
35.     for (j = 0; j < x; ++j)
36.         a[j] = (float)rand();
37. }
38.
39. int main(void) {
40.     long i; //counter
41.     float *a, *c; // host copies of a, b, c
42.     float *d_a, *d_c; // device copies of a, b, c
43.     long int size = N * sizeof(float);
44.     clock_t start, end;
45.
46.     _beep(2000, 500);
47.     printf("\n %d \n", N*LOOPS);
48.
49.     //with CUDA
```

```
50.
51.     start = clock();
52.
53.         // Alloc space for device copies of a, b, c
54.         cudaMalloc((void **)&d_a, size);
55.         //cudaMalloc((void **)&d_b, size);
56.         cudaMalloc((void **)&d_c, size);
57.         // Alloc space for host copies of a, b, c and setup input values
58.         a = (float *)malloc(size); random_floats(a, N);
59.         //b = (float *)malloc(size); random_floats(b, N);
60.         c = (float *)malloc(size);
61.         // Copy inputs to device
62.         cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
63.         //cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
64.
65.         // <<<B,T>>>, B = no of blocks, T = number of threads
66.
67.
68.         // start code here
69.         for (int A = 0; A < LOOPS; A++){
70.             square << < N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(d_a, d_c);
71.         }
72.         // Copy result back to host
73.         cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
74.
75.         //end clock here
76.         end = clock();
77.
78.     printf("\n %f seconds with CUDA", (double)(end - start) / CLOCKS_PER_SEC);
79.     _beep(2000, 500);
80.     // Cleanup
81.     free(a);
82.     //free(b);
83.     free(c);
84.     cudaFree(d_a);
85.     //cudaFree(d_b);
86.     cudaFree(d_c);
87.
88.     //without CUDA
89.     start = clock();
90.     // start code here
91.
92.     // Alloc space for host copies of a, b, c and setup input values
93.     a = (float *)malloc(size); random_floats(a, N);
94.     //b = (float *)malloc(size); random_floats(b, N);
95.     c = (float *)malloc(size);
96.
97.     for (int k = 0; k < LOOPS; k++)
98.     {
99.         for (i = 0; i < N; i++)
100.                {
101.                    c[i] = a[i] * a[i];
102.                    //printf("\n %l -> %f * %f = %f", i, a[i], a[i], c[i]);
103.                }
104.            }
105.
106.            // end of code
107.            end = clock();
```

```
108.            printf("\n %f seconds without CUDA", (double)(end - start) / CLOCKS_PER_SEC)
    ;
109.            //
110.            // Cleanup
111.            free(a);
112.            //free(b);
113.            free(c);
114.            _beep(2000, 500);
115.            getch();
116.            return 0;
117.        }
```

APPENDIX E: BENCHMARK FOR VECTOR ADDITION

```
1.  #include "cuda_runtime.h"
2.  #include "device_launch_parameters.h"
3.  #include <stdio.h>
4.  #include <conio.h>
5.  #include <stdlib.h>
6.  #include <malloc.h>
7.  #include <time.h>
8.  #define N (8096*8096)
9.  // keep threads per block a multiple of 32
10. #define THREADS_PER_BLOCK 1024
11. #define LOOPS 200
12.
13. __global__ void add(float *d_a, float *d_b, float *d_c){
14.     int blockId = blockIdx.x
15.         + blockIdx.y * gridDim.x
16.         + gridDim.x * gridDim.y * blockIdx.z;
17.     int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
18.         + (threadIdx.z * (blockDim.x * blockDim.y))
19.         + (threadIdx.y * blockDim.x)
20.         + threadIdx.x;
21.     d_c[threadId] = d_a[threadId] + d_b[threadId];
22. }
23.
24. ////////////////////////////////////////////////////
25. void random_floats(float* a, int x)
26. {
27.     int  j;
28.     for (j = 0; j < x; ++j)
29.         a[j] = (float)rand();
30. }
31.
32. int main(void) {
33.     long i; //counter
34.     float *a, *c, *b; // host copies of a, b, c
35.     float *d_a, *d_c, *d_b; // device copies of a, b, c
36.     long int size = N * sizeof(float);
37.     clock_t start, end;
38.
39.     _beep(2000, 500);
40.     printf("\n %ld \n", N*LOOPS);
41.
42.     //with CUDA
43.
44.     start = clock();
```

```
45.
46.         // Alloc space for device copies of a, b, c
47.         cudaMalloc((void **)&d_a, size);
48.         cudaMalloc((void **)&d_b, size);
49.         cudaMalloc((void **)&d_c, size);
50.         // Alloc space for host copies of a, b, c and setup input values
51.         a = (float *)malloc(size); random_floats(a, N);
52.         b = (float *)malloc(size); random_floats(b, N);
53.         c = (float *)malloc(size);
54.         // Copy inputs to device
55.         cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
56.         cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
57.
58.         // <<<B,T>>>, B = no of blocks, T = number of threads
59.
60.
61.         // start code here
62.         for (int A = 0; A < LOOPS; A++){
63.             //stencil_1d <<< N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>(d_a, d_c);
64.             square << < N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(d_a, d_c);
65.         }
66.         // Copy result back to host
67.         cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
68.
69.         //end clock here
70.         end = clock();
71.
72.     printf("\n %f seconds with CUDA", (double)(end - start) / CLOCKS_PER_SEC);
73.     _beep(2000, 500);
74.     // Cleanup
75.     free(a);
76.     free(b);
77.     free(c);
78.     cudaFree(d_a);
79.     cudaFree(d_b);
80.     cudaFree(d_c);
81.
82.
83.     //without CUDA
84.     start = clock();
85.     // start code here
86.
87.     // Alloc space for host copies of a, b, c and setup input values
88.     a = (float *)malloc(size); random_floats(a, N);
89.     b = (float *)malloc(size); random_floats(b, N);
90.     c = (float *)malloc(size);
91.
92.
93.     for (int k = 0; k < LOOPS; k++)
94.     {
95.         for (i = 0; i < N; i++)
96.         {
97.             c[i] = a[i] + b[i];
98.             //printf("\n %l -> %f * %f = %f", i, a[i], a[i], c[i]);
99.         }
100.           }
101.
102.           // end of code
103.           end = clock();
```

```
104.            printf("\n %f seconds without CUDA", (double)(end - start) / CLOCKS_PER_SEC)
    ;
105.            //
106.            // Cleanup
107.            free(a);
108.            free(b);
109.            free(c);
110.            _beep(2000, 500);
111.            getch();
112.            return 0;


118.        }
```

APPENDIX F: BENCHMARK FOR RANDOM NUMBER GENERATION

```
1.  #include <stdio.h>
2.  #include "cuda_runtime.h"
3.  #include "device_launch_parameters.h"
4.  #include <conio.h>
5.  #include <curand.h>
6.  #include <curand_kernel.h>
7.  #include <time.h>
8.  #define N 8096*8096
9.  // keep threads per block a multiple of 32
10. #define THREADS_PER_BLOCK 1024
11. #define LOOPS 1
12.
13. __device__ int x1[N];
14. int x2[N];
15.
16. // RNG kernel
17. __global__ void print_kernel(curandState *state, int MAX) {
18.     int blockId = blockIdx.x
19.         + blockIdx.y * gridDim.x
20.         + gridDim.x * gridDim.y * blockIdx.z;
21.     //the thread ID
22.     int id = blockId * (blockDim.x * blockDim.y * blockDim.z)
23.         + (threadIdx.z * (blockDim.x * blockDim.y))
24.         + (threadIdx.y * blockDim.x)
25.         + threadIdx.x;
26.     /* Each thread gets same seed, a different sequence
27.     number, a different  offset */
28.     curand_init(13173, id, id, &state[id]);
29.     x1[id] = abs( (int) curand(&state[id]) % MAX);
30.     //printf("\n %d <-> T %d", x1[id] ,id);
31. }
32.
33. // RNG host function
34. void random_ints(long int X, int MAX)
35. {
36.     for (long int i = 0; i < X; i++)
37.         x2[i] = abs((int)rand() % MAX);
38. }
39.
40. //curand(state) % 100
41. int main() {
42.     //counter
43.     int i;
```

```
    clock_t start, end;
    // with CUDA
    printf("\n Random Numbers with CUDA \n");
    curandState *devStates;
    /* Allocate space for prng states on device */
    cudaMalloc((void **)&devStates, 32*32);
    start = clock();
    for (i=0; i<LOOPS; i++)
        print_kernel << <N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(devStates, 12000
);
    cudaDeviceSynchronize();
    end = clock();
    printf("Execution time: %f seconds\n",(float)(-start+end)/CLOCKS_PER_SEC);
    cudaFree(devStates);

    // without CUDA
    printf("\n Random Numbers with CPU \n");
    start = clock();
    for (i = 0; i < LOOPS; i++)
        random_ints(N, 12000);
    end = clock();
    printf("Execution time: %f seconds\n",(float) (-start + end) / CLOCKS_PER_SEC);
    getch();

}
```

APPENDIX G: BENCHMARK FOR ONE-VARIABLE EQUATION SOLVER

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <time.h>
#include <curand.h>
#include <curand_kernel.h>

#define N (8096)
// keep threads per block a multiple of 32
#define THREADS_PER_BLOCK 1024
#define LOOPS 100

//kernel for random number generation
__global__ void random(curandState *state, float *d_rn, int X) {
    int blockId = blockIdx.x
        + blockIdx.y * gridDim.x
        + gridDim.x * gridDim.y * blockIdx.z;
    int n = blockId * (blockDim.x * blockDim.y * blockDim.z)
        + (threadIdx.z * (blockDim.x * blockDim.y))
        + (threadIdx.y * blockDim.x)
        + threadIdx.x;
    if (n < N){
        curand_init(X, n, 0, &state[n]);
        d_rn[n] = (float) curand(&state[n]);
        //printf("\n %d -> %f",n,d_rn[n]);
```

```
29.         __syncthreads();
30.     }
31. }
32. 
33. //kernel to solve x
34. __global__ void solve1(float *d_A, float *d_B, float *d_C, float *d_X )
35. {
36.     int blockId = blockIdx.x
37.         + blockIdx.y * gridDim.x
38.         + gridDim.x * gridDim.y * blockIdx.z;
39.     int n = blockId * (blockDim.x * blockDim.y * blockDim.z)
40.         + (threadIdx.z * (blockDim.x * blockDim.y))
41.         + (threadIdx.y * blockDim.x)
42.         + threadIdx.x;
43.     //printf("\n 1");
44.     if (n < N)
45.     {
46.         //printf("\n %f %f %f", d_A[n], d_B[n], d_C[n]);
47.         d_X[n] = (d_C[n] - d_B[n]) / d_A[n];
48.         //printf("\n %d -> %f", n, d_X[n]);
49.         __syncthreads();
50.     }
51. }
52. 
53. //simple function to solve x
54. float solve1D(float a, float b, float c)
55. {
56.     return (c - b) / a;
57. }
58. 
59. void random_floats(float* a, int x)
60. {
61.     int  j;
62.     for (j = 0; j < x; ++j)
63.         a[j] = (float)rand();
64. }
65. 
66. int main(void) {
67.     long i; //counter
68.     float *a, *c, *b , *x; // host copies
69.     float *d_a, *d_b, *d_c, *d_x; // device copies
70.     long int size = N * sizeof(float);
71.     clock_t start, end;
72. 
73.     _beep(2000, 500);
74.     printf("\n %d elements and %d loops \n", N,LOOPS);
75. 
76.     //with CUDA
77.     start = clock();
78. 
79.         // Alloc space for device copies
80.         cudaMalloc((void **)&d_a, size);
81.         cudaMalloc((void **)&d_b, size);
82.         cudaMalloc((void **)&d_c, size);
83.         cudaMalloc((void **)&d_x, size);
84.         curandState *devStates;
85.         cudaMalloc((void **)&devStates, N);
86. 
87.         // start code here
```

```
88.         for (int k = 0; k < LOOPS; k++){
89.             random << < N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(devStates,d_a, ra
    nd());
90.             cudaDeviceSynchronize();
91.             random << < N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(devStates, d_b, r
    and());
92.             cudaDeviceSynchronize();
93.             random << < N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(devStates, d_c,ra
    nd());
94.             cudaDeviceSynchronize();
95.             solve1 << < N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(d_a, d_b, d_c, d_
    x);
96.             cudaDeviceSynchronize();
97.         }
98.
99.         //end clock here
100.                end = clock();
101.
102.            printf("\n %f seconds with CUDA", (double)(end - start) / CLOCKS_PER_SEC);
103.            _beep(2000, 500);
104.            cudaFree(d_a);
105.            cudaFree(d_b);
106.            cudaFree(d_c);
107.            cudaFree(d_x);
108.
109.
110.            //without CUDA
111.            start = clock();
112.            // start code here
113.
114.            b = (float *)malloc(size); random_floats(b, N);
115.            c = (float *)malloc(size); random_floats(c, N);
116.            a = (float *)malloc(size); random_floats(a, N);
117.            x = (float *)malloc(size); random_floats(x, N);
118.
119.            for (int k = 0; k < LOOPS; k++)
120.            {
121.                for (i = 0; i < N; i++)
122.                {
123.                    random_floats(a, N);
124.                    random_floats(b, N);
125.                    random_floats(c, N);
126.                    x[i] = c[i] - b[i] / a[i];
127.                    //printf("\n %d -> %f", i, x[i]);
128.                }
129.            }
130.
131.            // end of code
132.            end = clock();
133.            printf("\n %f seconds without CUDA", (double)(end - start) / CLOCKS_PER_SEC)
    ;
134.
135.            // Cleanup
136.            free(a);
137.            free(b);
138.            free(c);
139.            free(x);
140.            _beep(2000, 500);
141.            getch();
```

```
            return 0;
        }
```

APPENDIX H: BENCHMARK FOR TWO-VARIABLE EQUATION SOLVER

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <time.h>
#include <curand.h>
#include <curand_kernel.h>

#define N (8096)
// keep threads per block a multiple of 32
#define THREADS_PER_BLOCK 1024
#define LOOPS 50

//kernel for random number generation
__global__ void random(curandState *state, float *d_rn, int X) {
    int blockId = blockIdx.x
        + blockIdx.y * gridDim.x
        + gridDim.x * gridDim.y * blockIdx.z;
    int n = blockId * (blockDim.x * blockDim.y * blockDim.z)
        + (threadIdx.z * (blockDim.x * blockDim.y))
        + (threadIdx.y * blockDim.x)
        + threadIdx.x;
    if (n < N){
        curand_init(X, n, 0, &state[n]);
        d_rn[n] = (float) curand(&state[n]);
        //printf("\n %d -> %f",n,d_rn[n]);
        __syncthreads();
    }
}

//kernel to solve D
__global__ void getD(float *A1, float *B1, float *A2, float *B2, float *D)
{
    int blockId = blockIdx.x
        + blockIdx.y * gridDim.x
        + gridDim.x * gridDim.y * blockIdx.z;
    int n = blockId * (blockDim.x * blockDim.y * blockDim.z)
        + (threadIdx.z * (blockDim.x * blockDim.y))
        + (threadIdx.y * blockDim.x)
        + threadIdx.x;
    if (n < N)
    {
        //printf("\n %f %f %f", d_A[n], d_B[n], d_C[n]);
        D[n] = A1[n]*B2[n] - A2[n]*B1[n];
        //printf("\n %d -> %f", n, d_X[n]);
        __syncthreads();
    }
}

```

```
52. //kernel to solve x
53. __global__ void solve2(float *A1, float *B1, float *C1, float *A2, float *B2, float *C2
    , float *X, float *Y, float *D)
54. {
55.     int blockId = blockIdx.x
56.         + blockIdx.y * gridDim.x
57.         + gridDim.x * gridDim.y * blockIdx.z;
58.     int n = blockId * (blockDim.x * blockDim.y * blockDim.z)
59.         + (threadIdx.z * (blockDim.x * blockDim.y))
60.         + (threadIdx.y * blockDim.x)
61.         + threadIdx.x;
62.     //printf("\n 1");
63.     if (n < N)
64.     {
65.         //printf("\n %f %f %f", d_A[n], d_B[n], d_C[n]);
66.         X[n] = (C1[n]*B2[n] - B1[n]*C2[n]) / D[n];
67.         Y[n] = (C2[n] * A1[n] - A2[n]*C1[n]) / D[n];
68.         __syncthreads();
69.     }
70. }
71.
72. void random_floats(float* a, int x)
73. {
74.     int  j;
75.     for (j = 0; j < x; ++j)
76.         a[j] = (float)rand();
77. }
78.
79. int main(void) {
80.     long i; //counter
81.     float *a1, *a2, *c1, *c2, *b1, *b2, *x, *y,*D; // host copies
82.     float *d_a1, *d_b1, *d_c1, *d_x, *d_y, *d_D; // device copies
83.     float *d_a2, *d_b2, *d_c2; // device copies
84.     long int size = N * sizeof(float);
85.     clock_t start, end;
86.
87.     _beep(2000, 500);
88.     printf("\n %d elements and %d loops \n", N,LOOPS);
89.
90.     //with CUDA
91.     start = clock();
92.
93.         // Alloc space for device copies
94.         cudaMalloc((void **)&d_a1, size);
95.         cudaMalloc((void **)&d_b1, size);
96.         cudaMalloc((void **)&d_c1, size);
97.         cudaMalloc((void **)&d_a2, size);
98.         cudaMalloc((void **)&d_b2, size);
99.         cudaMalloc((void **)&d_c2, size);
100.                cudaMalloc((void **)&d_x, size);
101.                cudaMalloc((void **)&d_y, size);
102.                cudaMalloc((void **)&d_D, size);
103.                curandState *devStates;
104.                cudaMalloc((void **)&devStates, N);
105.
106.                // start code here
107.                for (int k = 0; k < LOOPS; k++){
108.                    random << < N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(devStates,
    d_a1, rand());
```

```
109.                    cudaDeviceSynchronize();
110.                    random << < N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(devStates,
    d_a2, rand());
111.                    cudaDeviceSynchronize();
112.                    random << < N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(devStates,
    d_b1, rand());
113.                    cudaDeviceSynchronize();
114.                    random << < N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(devStates,
    d_b2, rand());
115.                    cudaDeviceSynchronize();
116.                    random << < N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(devStates,
    d_c1,rand());
117.                    cudaDeviceSynchronize();
118.                    random << < N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(devStates,
    d_c1, rand());
119.                    cudaDeviceSynchronize();
120.                    getD << < N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(d_a1, d_b1,
   d_a2, d_b2, d_D);
121.                    cudaDeviceSynchronize();
122.                    solve2 << < N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >> >(d_a1, d_b1
   , d_c1, d_a2, d_b2, d_c2, d_x,d_y, d_D);
123.                    cudaDeviceSynchronize();
124.                }
125.
126.                //end clock here
127.                end = clock();
128.
129.            printf("\n %f seconds with CUDA", (double)(end - start) / CLOCKS_PER_SEC);
130.            _beep(2000, 500);
131.            cudaFree(d_a1);
132.            cudaFree(d_b1);
133.            cudaFree(d_c1);
134.            cudaFree(d_a2);
135.            cudaFree(d_b2);
136.            cudaFree(d_c2);
137.            cudaFree(d_x);
138.            cudaFree(d_y);
139.            cudaFree(d_D);
140.
141.            //without CUDA
142.            start = clock();
143.            // start code here
144.
145.            b1 = (float *)malloc(size);
146.            c1 = (float *)malloc(size);
147.            a1 = (float *)malloc(size);
148.            b2 = (float *)malloc(size);
149.            c2 = (float *)malloc(size);
150.            a2 = (float *)malloc(size);
151.            x = (float *)malloc(size);
152.            y = (float *)malloc(size);
153.            D = (float *)malloc(size);
154.
155.            for (int k = 0; k < LOOPS; k++)
156.            {
157.                for (i = 0; i < N; i++)
158.                {
159.                    random_floats(a1, N);
160.                    random_floats(b1, N);
```

```
161.                    random_floats(c1, N);
162.                    random_floats(a2, N);
163.                    random_floats(b2, N);
164.                    random_floats(c2, N);
165.                    D[i] = a1[i]*b2[i] - b1[i]*a2[i];
166.                    x[i] = (c1[i]*b2[i] - c2[i]*b1[i] ) / D[i];
167.                    y[i] = (c2[i] * a1[i] - c1[i] * a2[i]) / D[i];
168.                    //printf("\n %d x-> %f", i, x[i]);
169.                    //printf("\n %d y-> %f", i, y[i]);
170.                }
171.            }
172.
173.            // end of code
174.            end = clock();
175.            printf("\n %f seconds without CUDA", (double)(end - start) / CLOCKS_PER_SEC)
    ;
176.
177.            // Cleanup
178.            free(a1);
179.            free(b1);
180.            free(c1);
181.            free(a2);
182.            free(b2);
183.            free(c2);
184.            free(x);
185.            free(y);
186.            free(D);
187.
188.            _beep(2000, 500);
189.            getch();
190.            return 0;
191.        }
```

Code snippets were generated using Google's Syntax Highlighter JavaScript code which can be found at http://www.planetb.ca/syntax-highlight-word