



WICHITA STATE
UNIVERSITY

UNIVERSITY LIBRARIES

Improving GPU performance by regrouping CPU-memory data

Item Type	Thesis
Authors	Gummadi, Deepthi
Publisher	Wichita State University
Rights	Copyright 2014 Deepthi Gummadi
Download date	2026-05-19 07:29:20
Link to Item	http://hdl.handle.net/10057/10959

IMPROVING GPU PERFORMANCE BY REGROUPING CPU-MEMORY DATA

A Thesis by

Deepthi Gummadi

Bachelor of Engineering and Technology, Jawaharlal Nehru Technological University, 2009

Submitted to the Department of Electrical Engineering and Computer Science
and the faculty of the Graduate School of
Wichita State University
in partial fulfillment of
the requirements for the degree of
Master of Science

May 2014

© Copyright 2014 by Deepthi Gummadi

All Rights Reserved

IMPROVING GPU PERFORMANCE BY REGROUPING CPU-MEMORY DATA

The following faculty members have examined the final copy of this thesis for form and content, and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science, with a major in Computer Networking.

Abu Asaduzzaman, Committee Chair

Ramazan Asmatulu, Committee Member

Zheng Chen, Committee Member

DEDICATION

To the Almighty, my loving husband, parents, in-laws, brother, and cousins for their ultimate encouragement throughout my education and for incomparable advice throughout my life

ACKNOWLEDGMENTS

I am very thankful to my thesis advisor Dr. Abu Asaduzzaman for his continuous encouragement and support throughout my research work. His timely supervision of my work and guidance allowed this research work to be completed on time. He always had time and patience to guide me in accomplishing this work in spite of his busy schedule and offered me assistance from time to time. It has been an honor to work for him as a graduate research assistant.

I express my sincere gratitude and thanks towards Dr. Ramazan Asmatulu (WSU ME Department) and Dr. Zheng Chen for taking time from their busy schedules and serve in my thesis committee.

I take pleasure in recognizing, the efforts of all those who encouraged and assisted me both directly and indirectly with my experimental research. I specially want to thank Chok M. Yip for his help in obtaining the required data. Finally, I acknowledge the WSU CAPPLab research group and facilities for providing me with all the necessary resources to prepare my research work.

ABSTRACT

In order to fast effective analysis of large complex systems, high-performance computing is essential. NVIDIA Compute Unified Device Architecture (CUDA)-assisted central processing unit (CPU) / graphics processing unit (GPU) computing platform has proven its potential to be used in high-performance computing. In CPU/GPU computing, original data and instructions are copied from CPU main memory to GPU global memory. Inside GPU, it would be beneficial to keep the data into shared memory (shared only by the threads of that block) than in the global memory (shared by all threads). However, shared memory is much smaller than global memory (for Fermi Tesla C2075, total shared memory per block is 48 KB and total global memory is 6 GB). In this paper, we introduce a CPU-memory to GPU-global-memory mapping technique to improve GPU and overall system performance by increasing the effectiveness of GPU-shared memory. We use NVIDIA 448-core Fermi and 2496-core Kepler GPU cards in this study. Experimental results, from solving Laplace's equation for 512x512 matrixes using a Fermi GPU card, show that proposed CPU-to-GPU memory mapping technique help decrease the overall execution time by more than 75%.

TABLE OF CONTENTS

Chapter	Page
1. INTRODUCTION.....	1
1.1 CPU Computing.....	1
1.1.1 CPU Components.....	3
1.1.2 Working Procedure in CPU.....	4
1.1.3 Cache Memory Organization.....	6
1.2 GPU Computing.....	9
1.2.1 GPU Components.....	9
1.2.2 Working Procedure in GPU.....	11
1.2.3 GPU Memory Organization.....	13
1.3 CPU-GPU Computing Workflow.....	15
1.4 Problem Statement.....	18
1.5 Contributions.....	18
1.6 Thesis Organization.....	19
2. LITERATURE SURVEY.....	20
3. PROPOSED CPU-MEMORY REGROUPING TECHNIQUE.....	28
3.1 Traditional CPU to GPU Memory Mapping.....	28
3.2 Proposed CPU to GPU Memory Mapping.....	33
4. EXPERIMENTAL DETAILS.....	38
4.1 Assumptions.....	38
4.2 System Parameters.....	38
4.3 Experimental Setup.....	42
4.4 Applications.....	45
4.5 CUDA/C Implementation.....	47
4.6 Input/output Variables.....	48
5. RESULTS AND DISCUSSION.....	49
5.1 Validation of CUDA/C Implementation.....	49
5.2 Impact of having GPU Shared Memory.....	52
5.3 Impact of Number of Threads.....	53
5.4 Impact of the amount of Shared Memory.....	54
5.5 Impact of Proposed CPU-Memory Data Regrouping Technique.....	55

TABLE OF CONTENTS (continued)

Chapter	Page
6. CONCLUSION AND FUTURE EXTENSIONS.....	57
6.1 Conclusion.....	57
6.2 Future Extensions.....	58
REFERENCES.....	59

LIST OF TABLES

Table		Page
1.	GPU Memory general properties.....	15
2.	Specifications of CPU and GPU processors.....	39
3.	Core and Memory specifications of CPU and GPU cards.....	42

LIST OF FIGURES

Figures	Page
1.1 A Sample Architecture of a CPU.....	2
1.2 Working Procedure in a Typical CPU.....	5
1.3 Cache Memory Organization of a Shared CL2 System.....	6
1.4 GPU Memory Organization.....	10
1.5 Working Procedure in GPU.....	11
1.6 CPU Memory Allocation in DRAM.....	15
1.7 Instructions received by GPU from CPU.....	16
1.8 Instructions Execute at GPU.....	17
1.9 Retriving the Executed Data from GPU to CPU.....	17
2.1 Two Parallelization Strategies.....	22
2.2 CUDA 5.5 Version on ARM.....	24
2.3 CUDA Memory Model (a) Unified (b) Actual.....	25
3.1 Traditional CPU to GPU Global Memory Mapping.....	29
3.2 Traditional Memory Mapping based on our Example.....	31
3.3 Proposed CPU to GPU Memory Mapping.....	33
3.4 Proposed Memory Mapping based on our Example.....	34
4.1 Experimental setup on CPU-GPU cards.....	44
4.2 Sample CUDA/C Implementation code.....	47
5.1 An 8X8 Matrix with Boundary Condition at Iteration-0.....	49

LIST OF FIGURES (continued)

Figures	Page
5.2 An 8X8 Matrix with Boundary Condition at Iteration-1.....	50
5.3 An 8X8 Matrix with Boundary Condition at Iteration-10.....	50
5.4 An 8X8 Matrix with Boundary Condition at Iteration-50.....	51
5.5 An 8X8 Matrix with Boundary Condition at Iteration-100.....	51
5.6 Validation of the developed CUDA/C Implementation Code.....	52
5.7 Impact of GPU Performance on with and without Shared Memory.....	53
5.8 GPU Time Vs Number of Threads.....	54
5.9 GPU Time Vs Used Shared Memory.....	55
5.10 Impact of Data Regrouping.....	56

LIST OF ABBREVIATIONS

ALU	Arithmetic/Logic Unit
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random Access Memory
FFT	Fast Fourier Transformations
GPU	Graphic Processing Unit
SDP	Spatial Data Partitioning
SIP	Spatial Instruction Partitioning
SM	Streaming Multiprocessor
TDP	Temporal Data Partitioning
TIP	Temporal Instruction Partitioning

CHAPTER 1

INTRODUCTION

Architecturally, a modern Central Processing Unit (CPU) is composed of only few cores with cache memories which can handle a few threads at a time; whereas a Graphic Processing Unit (GPU) is composed of hundreds of cores which can handle thousands of threads simultaneously [1, 2, 3, 4]. The ability of a GPU with more than hundred cores to process thousands of threads can be accelerated by hundred times over a CPU alone. The combination of a CPU with a GPU can deliver the best value of system performance, price, and power. In this chapter, we will be discussing some of the basic components and mechanisms of CPU and GPU, and we will also discuss how a cache memory is organized in a CPU and how the operations are performed between CPU and GPU.

1.1 CPU Computing

CPU is an integrated circuit, which is housed on a single chip called a microprocessor. CPU converts input data into an information output. The modern usage of computer applications gives many challenges for design engineers; to avoid that challenges, design engineers had to change the design of a chip from single large complex core to many simple small cores on the computer chip, to provide the better performance,. This multiple CPU cores on a single chip are known as multi-core processors [5, 6].

CPU has an extensive set of electronic circuitry that is responsible for interprets and executes program instructions, which are fetched from memory storage or taking the instructions from the input.

As shown in Figure 1, the single-core CPU has mainly consists of two parts:

- Control unit
- Arithmetic/logic unit

Before discussing the each functionality in detail, we need to see the relationship between the CPU and its data storage. Each and every computer has two types of storages – one is primary storage and the second one is secondary storage. Primary storage, which is also called as main memory or RAM, locates within the CPU, technically, it is not within the CPU. Main memory refers to both data and instructions. At the time of executing the instructions main memory holds the data temporarily, whereas the secondary storage holds data permanently or semi permanently. Examples of secondary storage media are flash drives, CD-ROM disks, and etc. in desktop computers and hard disks in personal computers.

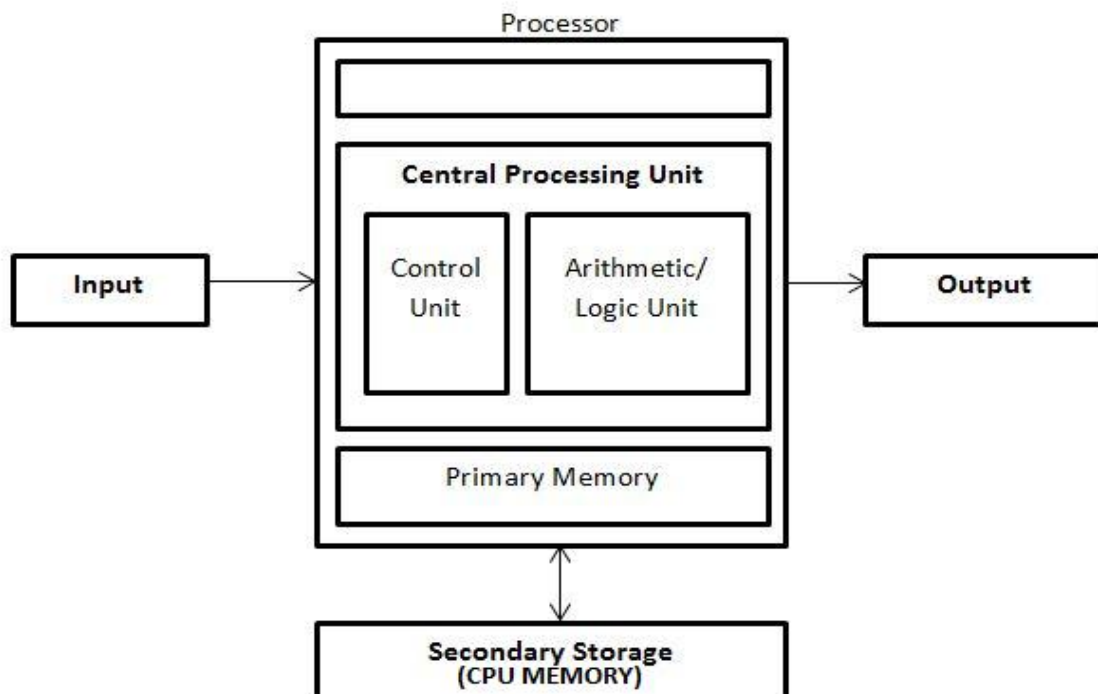


Figure 1. A Sample Architecture of a CPU.

1.1.1 CPU Components

In this section, we will see how the components of CPU behave.

Control Unit:

The Control Unit does not execute any program instructions, it just makes decisions and sends the appropriate signals to direct the computer system with the instructions to execute, carry out, or to store program or data. These signals are electrical signals which are used by the control unit circuitry to direct the instructions from memory to arithmetic/logic unit. It controls the timing of operations and controls the instruction set to the processor and peripheral devices.

Arithmetic/Logic Unit:

All the logical and arithmetic operations are performed in arithmetic/logic unit (ALU). As the name implies, ALU can perform arithmetic operations like addition, subtraction, multiplication, and division. And, Logical operation only performs comparisons based on the given instructions (like greater than, less than, greater than or equal to, etc.).

Registers:

The name itself indicates, registers stores instructions/data. It is a temporary storage area. Registers are the additional storage locations to speed up the processing. Registers are operated under the direction of the control unit to give the high performance to single core CPU. The control unit gives instructions to registers regarding the instructions/data to accept, hold, and transfer to arithmetic or logical operations.

There are some special registers for better performance. Those are:

An accumulator: This gathers the result of different computations.

An address register: In the memory, each and every storage location is assigned by an address. By

using that address computer can identify the location easily. This address register keeps track of all the information about where to store the data/instruction or where to fetch the data/instruction in memory.

Storage register: These types of registers are used for temporary hold of data, which is retrieved from or about to be sent to main memory.

A general purpose register: It is used for several special functions.

1.1.2 Working Procedure in CPU

Let's see how the control unit, ALU, registers, secondary memory, primary memory all works together and how the CPU executes the single instruction in Figure 2. Initially, the control unit takes the input data from input device or from secondary memory and stores the input data and instructions in primary memory (The primary memory stores the data temporarily at registers and the primary memory is faster to access than the secondary memory. But, the primary memory cannot hold the data for a long time whereas the secondary memory stores the data for long periods of time). Whenever the primary memory receives the instructions, the control unit retrieves the data and instruction from the memory and then based on the input commands or instructions the control unit decodes the data and then it transferred it to the ALU, where an arithmetic operation or logical operation is performed. After the instructions being processed, the output information is given to the memory to store, where it is hold until it is ready to be released as an output on the screen [7].

A single instruction may have much number of sub-instructions, for processing each and every sub-instruction it must take at least one clock cycle. For synchronizing all operations at a fixed rate there is an internal clock cycle in each CPU, which produces pulses at a fixed rate. A

particular type of CPU is designed to understand a particular set of instructions called the instruction set. Therefore, one type of CPU instruction set is different from another type of instruction set. For example, Compaq instruction set is different from Macintosh instruction set.

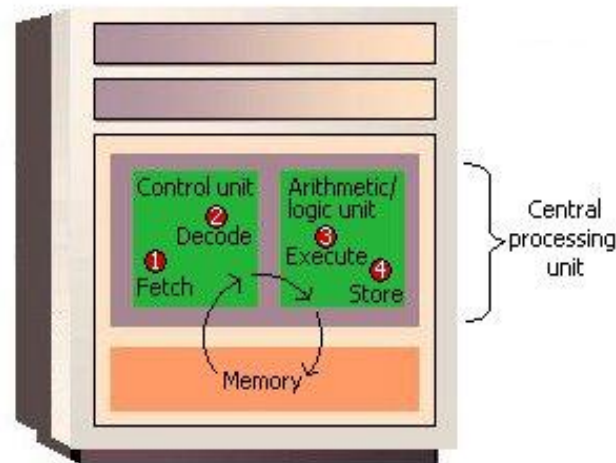


Figure 2. Working Procedure in a Typical CPU [7].

Every instruction or a data is stored in a memory location, which are identified by an address. The size of address block is fixed and is constant for all the locations within the memory locations. It can hold only fixed amount of bytes - often two bytes in a modern computer. Now-a-days computers are designed based on these storage hierarchies. Due to slow performance, floppy discs disappeared these days.

Initially, in the 1980's the processor and memory speeds are evolved with the speed range of approximately 1 MHz to 16 MHz. Later in millennium, the processors increased the speed to 1000 MHz (1GHz) and were reached unparalleled. Now, most contemporary CPUs have multicore architecture, which means multiple cores on a single chip, with total speed of nearly 4000 MHz.

1.1.3 Cache Memory Organization

Now a day's memory requirements for embedded applications grow, especially in multimedia devices, multi core processors become crucial for providing high performance and reducing power consumption [8, 9, 10]. Multi core processor in a single package becomes most popular within personal computer market, primarily from Intel, AMD and IBM. As the name itself indicate that multi core processor is the processor having two or more independent cores in a single package. Cache may be on-chip or off-CPU chip.

Cache memory organization of Intel like 4 core processor is shown in Figure 3. This memory organization includes private level - 1 cache (CL1), private or shared level - 2 caches (CL2) and the main memory [11]. The lower level caches (CL1) are faster, smaller capacity and mainly used for performing computations. Whereas, higher level caches (CL2) are slower, higher capacity and mainly used for storage purposes.

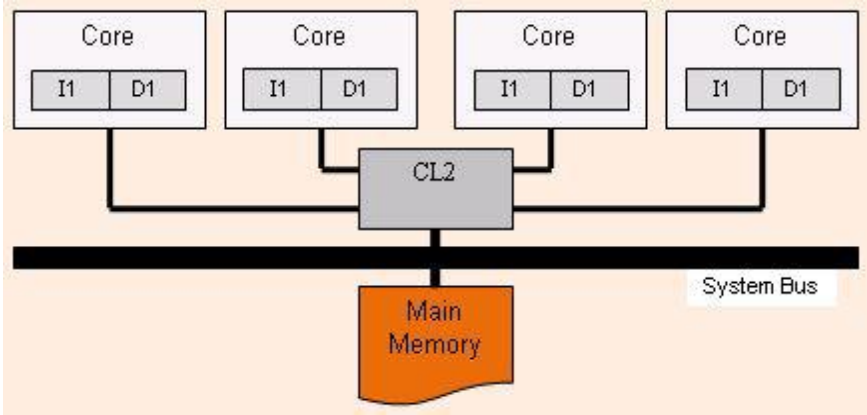


Figure 3. Cache Memory Organization of a Shared CL2 System.

The level 1 cache split into two types of caches, one is instruction cache and the other one is data cache. The instruction cache in the CL1 is fully pipelined and a pre-fetcher is employed in it by using branch prediction to avoid cache misses. This pipeline is dual ported, one port is used

for fetching the instructions and the other port is used for sharing the pre-fetches, and column invalidates. The data cache in CL1 is quad ported; it supports two concurrent loads and two concurrent stores. Each way set is associated with a size of 16KB and a total of line size 64KB. This cache memory process all Integer loads, Integer stores, Floating point loads and floating point stores.

In some multi core processors level 2 caches will be a private or shared cache. If the level-2 cache is unified shared cache then it will single ported and fully pipelined from level 1 cache to level 2 shared caches. It is accessed at full clock speed. All floating point accesses and all semaphore instructions are handled by the level 2 cache memory. The main advantage of unified level cache is that it can avoid cache missing and the data transferring between the cores does not replicate at this level [11].

Main memory sends a wider range of data to the cache than the CPU requires, this is because to fulfill the cache more rapidly. The amount of information required by the cache to load at one time is called the *line size* for the cache. The line size data is the width of the data bus between the main memory and the cache memory and this line size for the cache is used for pre-fetching the instructions or data.

In the cache memory organization, the memory controller determines whether the value is currently being addressed in cache memory or not. The memory controller can be accomplished in many ways. One possibility is to store both the address and the value from main memory into the cache; this type of memory is called *associative memory* or *content addressable memory*.

An associative memory returns the address of the value if it is already stored in the memory; otherwise it indicates the value is not associated in the memory. The comparisons are done simultaneously, so the results are quick and prompt. This type of memory requires a comparator and a storage element individually for value and address which leads the type of memory as expensive. There is a method to implement the cache memory by using "direct mapping", which is cheaper than associative memory. Here, part of the memory address is divided into index and tag. Usually, for addressing the data in cache, index is used for lower order bits of address and the tag is used for remaining higher-order bits, which are stored in the memory along with the data.

In the cache memory organization, whenever the instruction needs to be executed initially it checks at CL1, if the CL1 doesn't contain the required data, then it request for CL2, the data is transferred from CL2 if it contains, otherwise it checks at main memory. Once it identifies the required information for executing the instructions the data is copied in all levels and as well as main memory. Multi core processor increases performance and execution time as less activity over an external bus is needed. The amount of performance that is gained by the use of multi-core processors is highly dependent not only on the adopted parallel structures, but also on several restrictions imposed by the actual hardware architecture (e.g.: cache coherency, system interconnect, etc.)

Even though an on-chip cache exists, it is typically desirable to have an off-chip cache as well. This means that if a miss occurs on the level 1 cache (on-chip), instead of retrieving the data from the slower main memory, information may be retrieved from the level 2 caches, which, although slower than level 1 cache, is still appreciably faster than main memory. Some level 2

caches are stored on-chip and a level 3 cache has been implemented off-chip. Multiple cache levels are implemented to reconcile between the size and latency goals;

1.2 GPU Computing

Now-a-days, GPU gains popularity in supercomputing areas. GPU is primarily used for graphic applications or rendering 3-D images. Modern GPU's are very fast and efficient for manipulating the graphics than the general purpose CPU; this is because of GPU's parallel processing architecture [12, 13, 14]. Parallel processing means processing the blocks of data at the same time. Because, of its high performance GPU's are very popular. Mostly, the GPU's are used in workstations, game engines, embedded systems, play stations, etc. Some of the GPU features are: 2-D or 3-D graphics, Texture mapping, and Application support for high-intensity graphics software such as AutoCAD, etc.

Figure 4, explains the GPU computation with multicore CPU. In a multi core CPU and many core GPU platforms that support CUDA/C applications, the user starts the application on the multicore CPU. The initialization and serial parts are executed in the CPU. The data and code for the parallel parts are sent to the GPU card. For each parallel part, multiple threads are generated. Threads are executed in the GPU cores/execution pipelines concurrently in parallel.

1.2.1 GPU Components

GPU consist of two main components:

- Streaming Multiprocessors (SM)
- Memory Organization

Streaming Multiprocessors (SM):

The streaming multiprocessors are the part of the GPU that runs on kernels, which performs the actual computations. Each SM has a control units, thousands of registers, execution cores(or pipelines) and several caches [12].

Warp schedulers: Warp Schedulers is used for switching the contexts among the threads and the scheduler gives the instructions to warps that are ready to execute.

Execution pipeline: Execution pipeline or execution cores are the multiple pipelines in each SM, used for executing the integer and floating point operations or instructions at the same time. A single execution pipeline can run faster, whereas in each SM there are many execution pipelines and a GPU has many SM's, so the performance of GPU is much faster.

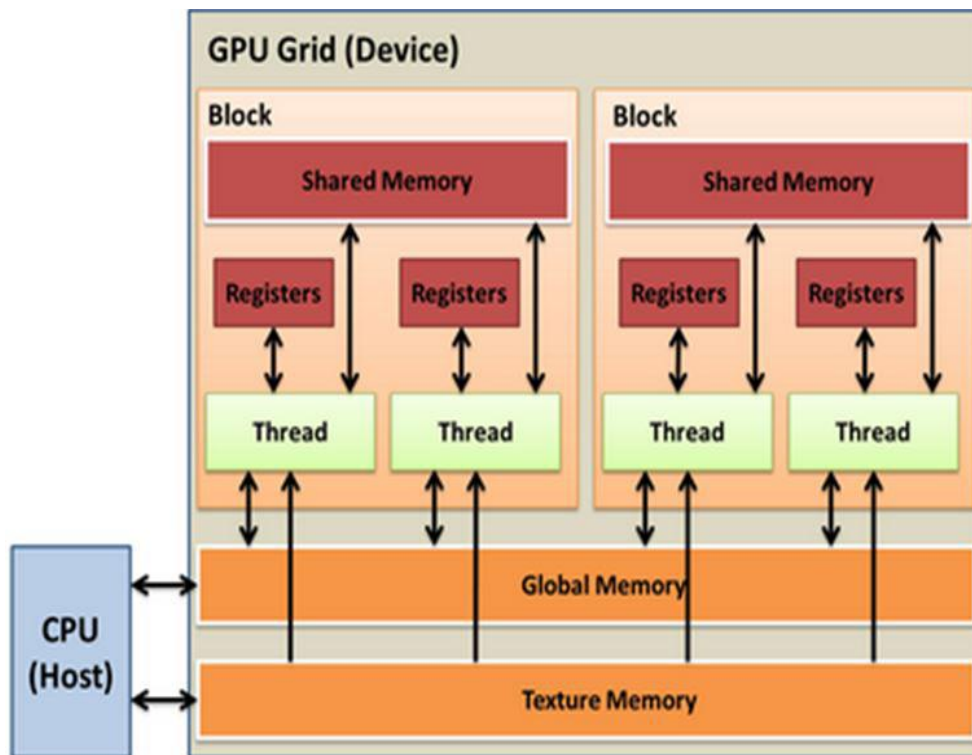


Figure 4. GPU Memory Organization [15].

Registers: Each SM contains thousands of 32-bit registers. Registers are allocated to the threads as specified when the kernel is launched. The obtained data can be partitioned among the threads for process. Registers having a plenty of memory to use and it is very fast in processing.

1.2.2 Working Procedure of GPU

GPU processes the data/instructions in a step by step process for a 3D application. Figure 5 illustrates the procedure [16, 17].

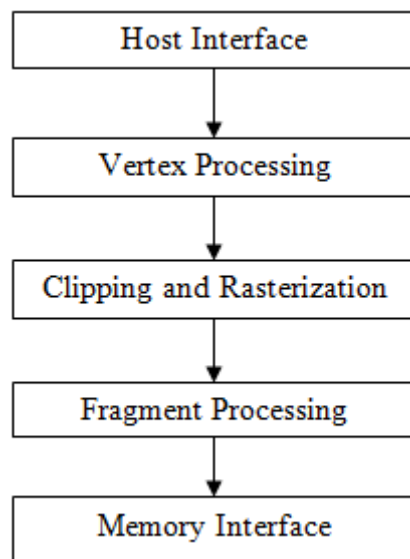


Figure 5. Working Procedure in GPU.

- Initially, GPU receives the input geometric information from the CPU by using the host interface. The host interface is working as a communication channel between the CPU and GPU. Once, host interface receives the instructions from the CPU then it gathers the geometric information from the system memory and sends the data to vertex processing unit as a stream of vertices with all the associated information.

- The vertex processing unit receives the set of vertices attributes (coordinates, texture, etc.) in object space and sends the output information or attributes (texture coordinates, clip space position, etc.) in screen space. No new vertices are created/discarded in this unit. The operation procedure within the vertex processing unit may be a simple linear transformation or complex transformations involving morphing effects. The performance in this stage will be based on the function of work done per vertex, along with the number of vertices being processed. The output from this unit is suitable for rasterization and clipping is transferred to triangle setup.

- Clipping and Rasterization is the stage where screen space geometric information is taken as input and it is transformed to raster information (pixels) as output. Clipping is the procedure that identifies the portion of the picture which is either inside or outside a region. It turns the set of vertices into primitives and fills them. If the center is inside the viewing frustum then fragmentation will be done. In this stage some hidden surface removal can be done when the geometric primitives are completely outside the viewing frustum. The generated fragments with interpolated data are sent to fragment shadier stage.

- In this stage, each fragment is fed into fragment processing as set of attributes (position, tex coordinates, etc.), and generates the output as final fragment associated with color and depth values. These colors are assigned based on the values interpolated from the vertices during rasterization and if the output doesn't need any special effects then depth values are untouched. The computation includes texture mapping and some math operations. Modern fragment processing stages are bottlenecked.

- The obtained fragment colors are stored in this memory interface (or frame buffer). Before final write occurs some of the tests will be performed on this fragment colors and then it

decides which surface is to be visible and which is not, in the elimination process it eliminates some of the hidden surfaces. The tests used in this process will be z-buffer, alpha and stencil. Modern GPUs, Z and color are compressed to reduce the frame buffer bandwidth.

The synchronization between the threads is done by using barrier synchronization. A barrier forces the threads to wait at synchronization point until all the threads reaches that point. Once all the threads reach at synchronization point, the barriers synchronize and release all the threads at a time to proceed. Another popular type of synchronization at operating system is the atomic transaction model. This model is mainly based on mutual exclusion; processes may request an access to shared resources simultaneously.

1.2.2 Memory Organization in GPU

GPU is organized with various memories [18, 19], which are as follows:

- *Shared memory* – Shared memory is restricted to a block. Each SM has a shared memory and it is used for interchanging the data between the threads within the block. Among all the on-chip memories shared memory is the fastest one in the SM. GPU shared memory helps improve performance mainly because it is dedicated to a GPU-block and it is closer to the processing cores.
- *Global memory* - It is an off-chip device memory, which will be the slowest GPU memory but the largest storage memory of all the memories in GPU. The contents of global memory are visible to all the threads of grid. Any thread can read and write to any location of the global memory. The instructions used for reading and writing to global memory are GLD/GST instructions. The total memory of GPU is the usage of total amount of DRAM. Compared to off-chip memory, on-chip memories are hundred times faster but with limited storage capacity. This is also known as L2 cache. L1 cache is the mediator between L2 global memory and local memory.

- Texture memory – Texture memory is cached on a chip and it is mainly optimized for exhibiting a great deal of spatial locality. Global memory is different from texture memory; this is because of its bi-directional data from global memory to threads whereas texture memory data flow is unidirectional. Threads of same wrap that reads the texture addresses that are close together will achieve the best performance and higher effective bandwidth by reducing memory request to off-chip DRAM. Texture memory is also designed for reducing the DRAM bandwidth demand with a constant latency.

All types of memory in GPU and its properties are tabulated in Table 1.

TABLE 1
GPU Memory General Properties.

Memory	Location	Cache	Access	Memory Utilization
Local	Off-chip	No	Read/Write	One Thread
Shared	On-chip	N/A	Read/Write	All Threads in a block
Global	Off-chip	No	Read/Write	All Threads + CPU
Constant	Off-chip	Yes	Read	All Threads + CPU
Texture	Off-chip	Yes	Read	All Threads + CPU

There are two levels of hardware-managed caches to improve the large difference in access times of on-chip and off-chip memories. These two levels are L1 and L2 caches in GPU. L1-cache is located within each SM at the speed of the shared memory and L2-cache is at the bottom of all SM's, which is for global memory. L2 cache has roughly half the access time because of its off-chip nature. On-chip memory access is very fast compared to off-chip. L1-cache main feature is

to reduce the latency between the local and global memory. Local memory is used for storing large data structures, such as arrays, which cannot be mapped into registers by the compiler. We can discuss the global memory in coming part.

1.3 CPU-GPU Computing Workflow

Figure 6-9 explains the four main steps that refer to data transfer and process between CPU and GPU [20].

Step 1: Initially, the CPU allocates the memory and copies the data from CPU memory to the DRAM of the GPU.

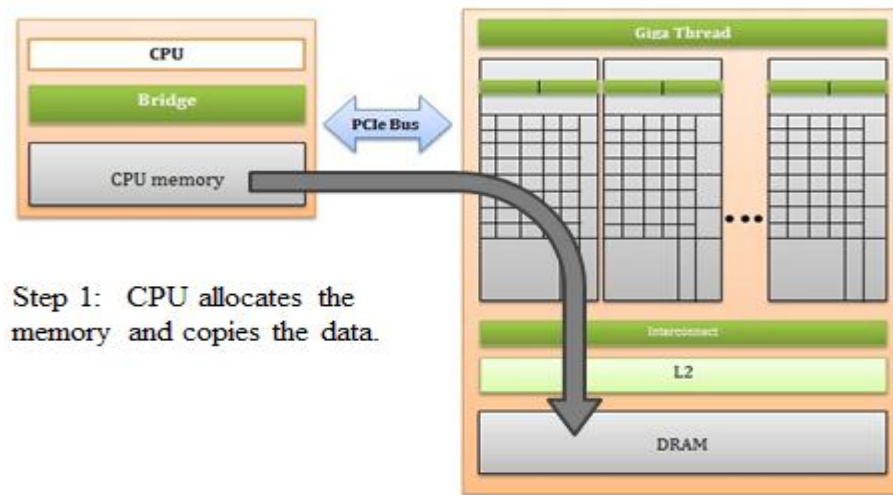


Figure 6. CPU Memory Allocation in DRAM.

By default, these operations are synchronous but an asynchronous mode is also possible. If the data copy process is synchronous then the CPU does not receive any call from GPU until the data has been fully copied. On CUDA API: `cudaMalloc()` and `cudaMemcpy()` are the commands used for memory allocation and memory copy.

Step 2: After copying the data, CPU sends the function parameters and instructions to the GPU, then GPU will execute. This is an asynchronous operation, means control returns as soon as the commands are delivered to the GPU.

CUDA API: `myFunc<<<Blocks, Threads>>>(parameters)`

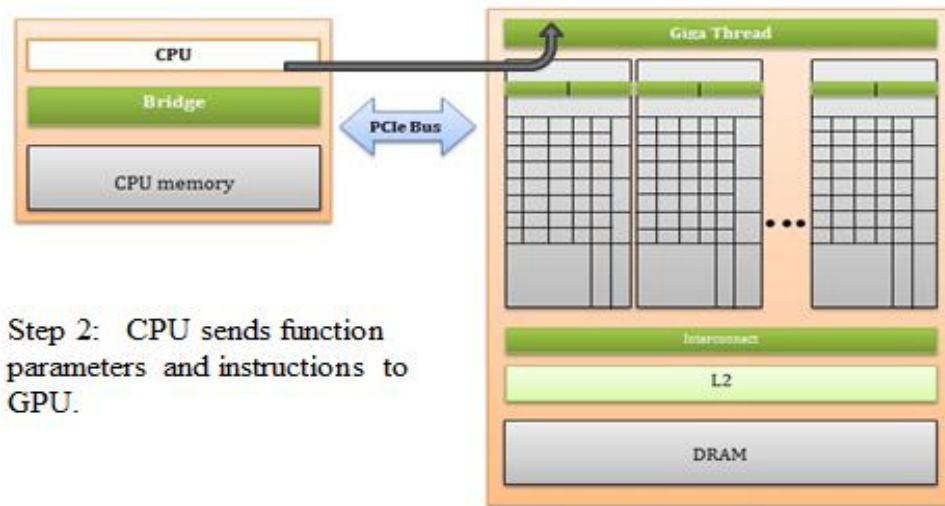


Figure 7. Instructions Received by GPU from CPU.

Step 3: GPU executes instruction based on the commands it received; it schedules the execution in warps.

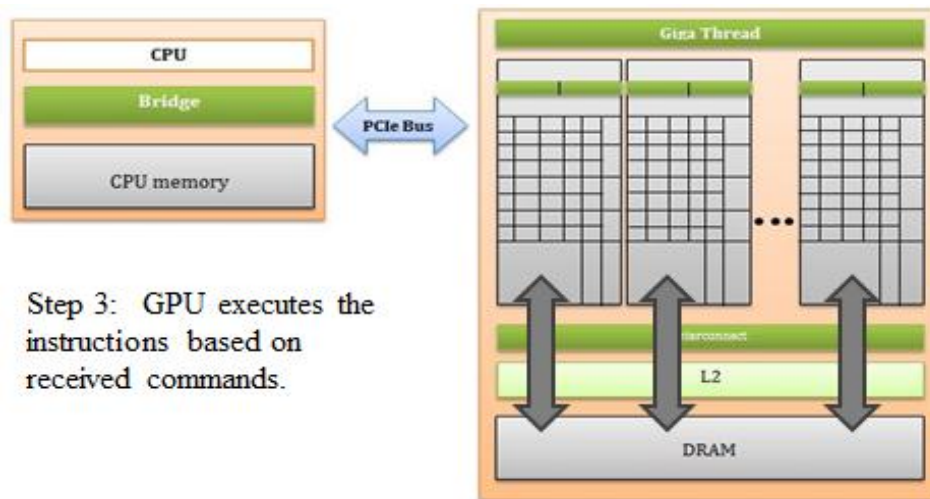


Figure 8. Instructions Execute at GPU.

Step 4: Once the data is computed by GPU, the results will be retrieved from GPU DRAM memory to CPU memory. Before getting the results from GPU, CPU needs to check the commands have been completed. This results transfer from GPU to CPU is synchronous by default. CUDA API: CPU use `cudaMemcpy()` command to retrieve the results from GPU.

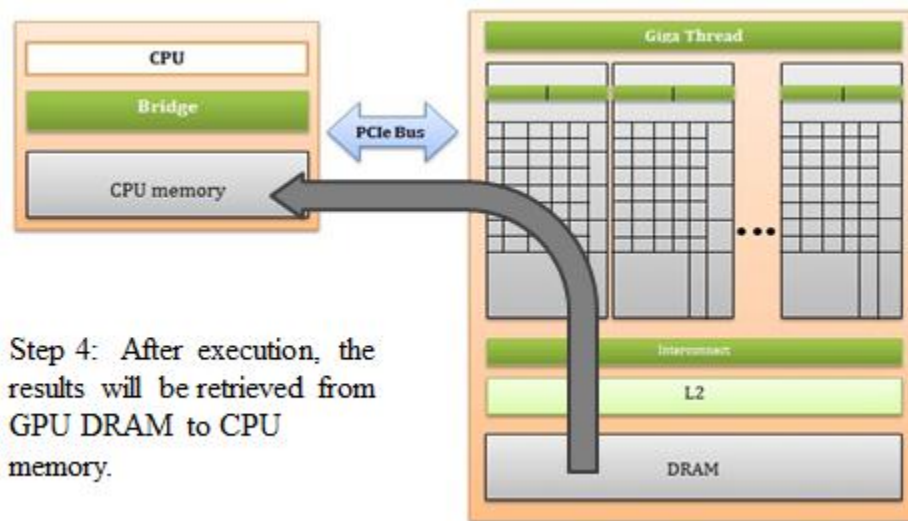


Figure 9. Retriving the Executed Data from GPU to CPU.

Let us see the difference between CPU and GPU – Modern CPU consists of a few cores optimized for sequential serial processing while a GPU consists of hundreds of “smaller”, more “efficient” cores designed for handling multiple tasks simultaneously. CPU has low latency, and GPU has high throughput and moderate latency, cache memory improves CPU performance whereas shared memory improves GPU performance, CPU computing technique has optimized MIMD (multiple instruction, multiple data) to achieve parallelism whereas GPU has optimized SIMD (single instruction, multiple data).

1.4 Problem Statement

In CUDA-assisted multithreaded programming, a thread usually processes data that are not in consecutive CPU memory locations but our traditional memory mapping is in the form of copying the data from consecutive memory locations of CPU, this CPU data normally does not qualify to fit in GPU shared memory. As the shared memory plays an important role in the GPU computing, the overall GPU performance decreases as the threads need to access the data every time from GPU global memory. Currently, there is no effective technique to copy the data from CPU to GPU memory.

1.5 Contributions

In this work, we propose a new memory mapping technique between CPU to GPU global memory. The major contributions are:

- A novel CPU- to- GPU memory mapping technique.
- A study on GPU performance with and without changing the size of the shared memory.
- A study on high electric charge distribution by using three different implementations.

Those are CPU only, GPU without shared memory and GPU with shared memory.

1.6 Thesis Organization

In chapter 2, we are going to discuss some of the related architectures and protocols that are already exist and study of some well approved related concepts from different conference and journal papers.

In chapter 3, we are going to present the clear idea about the proposed mapping technique and the methodologies for better understanding.

In chapter4, we discussed some of the system parameters, experimental setup and input parameters that we used for this experiment.

In chapter 5, we will discuss the experimental results from the proposed technique by using 2D charge distribution.

In chapter 6, we concluded with an improvement in GPU performance and the future scope for this new mapping technique is discussed.

CHAPTER 2

LITERATURE SURVEY

As the multicore processors in all computing systems are becoming popular, the relevance of parallel programming for these systems is enormous. As a result there are large efforts to improve the productivity in the development of parallel codes as well as the resulting performance. In this chapter, I would like to discuss data parallelism and task parallelism from Massachusetts Institute of Technology (MIT).

Since the introduction of dual-core netbook machines to 16-core workstation computers, parallel processing is a reality. To take advantage of the multicore systems, software engineers are developing parallel applications that will also meet the requirements of the growing high performance computation. These parallelized applications are based on data parallelism, which is an important parallel processing technique as it can take advantage of the locality principle.

The data parallelism and task parallelism are the most commonly used design patterns for parallel computing. Usage of task parallelism includes decomposing a program into concurrent tasks which are simultaneously executed as separate instructions. Task parallelism can be implemented using libraries such as POSIX threads, Intel Thread Building Blocks, and OpenMP. Some consider POSIX thread usage as an assembly language of parallelism because it provides minimal functionality. Thread Building Blocks libraries are introduced recently by Intel, by using higher level of abstraction this allows express parallelism. OpenMP [21] is mainly based on simple compiler directives used to guide mostly the parallelization of regular loops.

Let us see what the benefits are provided by task parallelism - Each task represents an asynchronous operation. Tasks are queued to thread pool. And, each thread pool consists of set algorithms which determine and adjust the number of threads that maximizes the throughput, and load balancing. This makes threads as relatively lightweight. Task parallelism is more scalable and efficient in use of system resources. Tasks built around the thread will provide a rich set of APIs that support waiting, cancellation, continuations, robust exception handling, detailed status, custom scheduling, etc. Improved version of task parallelism is out-of-order execution, which allows the instructions for high performance microprocessors to begin execution as soon as the operands are ready.

Data parallelism is used for decomposing a program into the concurrent units which execute the same instructions on distinct data. Among the parallel processing techniques, data parallelism is the important technique as it can take an advantage of locality principle [22, 23]. Where locality principle is the computational processes that pass through the sequence of instructions (or locality set); means next instruction comes from the immediate memory location.

Data parallelism extends both the task and data patterns by noting that applications execute in space and time. Researchers of MIT introduced four partitioning strategies for concurrent execution [24].

- Spatial Data Partitioning (SDP)
- Temporal Data Partitioning (TDP)
- Spatial Instruction Partitioning (SIP)
- Temporal Instruction Partitioning (TIP)

According to our thesis work, only SDP and TDP were considered. Figure 10, represents the two parallelization strategies. In SDP, based on the spatial indexes the data is divided among the processes. Computations are performed on the spatial data by keeping the temporal index as constant. Data/instruction with nearby addresses tends to be referenced close together in time. Each process performs all the instructions assigned to them. Additional instructions are required by SDP for communication and synchronization purpose.

When compared the serial implementation with the applications that are parallelized with SDP pattern we can observe some effects. Those are:

- Throughput of an application will be improved.
- It decreases the latency of a parallelized application.
- As the same instructions are often executed in each process, the load balancing of the parallelized application tends to be easy.
- Communication between the parallel processing is always application dependent.

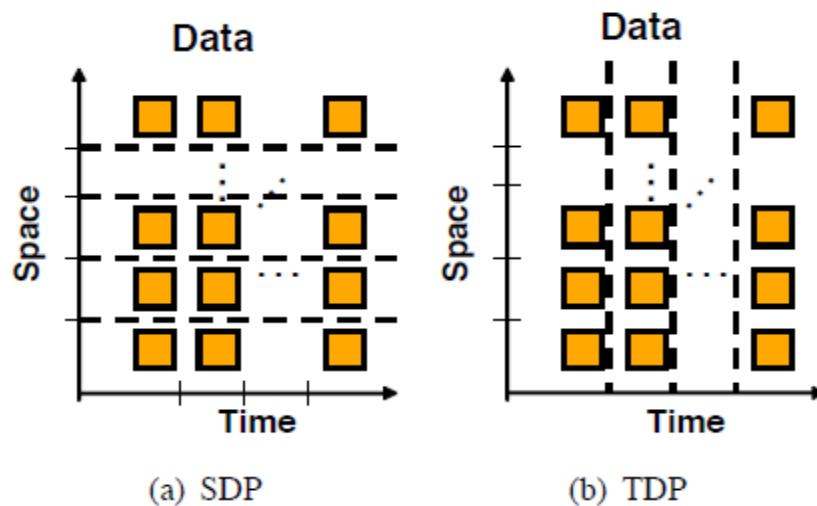


Figure 10. Two Parallelization Strategies [19].

The temporal index data is divided amongst the process (in TDP strategy) and the computations can be performed by them on all spatial indices associated with its assigned temporal index. Recently referenced data/instruction is likely to be referenced in the near future. Figure 3(b) illustrates the TDP. Typical TDP implements each process executions of all the instructions on the data from its assigned temporal index. The communication and synchronization instructions are to be added to the processes to handle the temporal dependencies. When the applications are parallelized with TDP pattern, there have been some effects to observe. Those are:

- This increases the throughput of application.
- Latency remains constant.
- Even the computation between the inputs vary tremendously, the load balance of the applications tends to be easy.
- Same as with SDP, communication between the parallel processing is always application dependent.

Let us see the applicability of both data partitioning. The SDP strategy is used to parallelize the application whenever a single processor does not meet the demands of application's latency. If the processor meets the demands of application's latency, but not the requirements of the throughput then use the TDP strategy to parallelize. If the dimensions of spatial data and temporal data are large with few dependencies use the SDP and TDP strategies respectively [25]. When the application performs same amount of work on each of the spatial indices, it uses the SDP strategy. If the application performs different computations on inputs it may benefit from load balancing patterns that synergize with TDP.

CUDA:

CUDA is a parallel computing platform and programming model, which increases computing performance dramatically by harnessing the GPU power. Following are the recent CUDA versions.

CUDA Version 5.5: Among the CUDA versions CUDA 5.5 is the first that supports ARM. There are two main reasons for NVIDIA to implement CUDA on ARM – NVIDIA is hoping to ultimately leverage CUDA on consumer side to compute on SoC based devices. The second reason is mainly based on NVIDIA's HPC ambition which is opposed by today's common scenario of pairing Tesla cards with x86 AMD and Intel processors, for this ARM based HPC environment is implemented that can be powered exclusively by NVIDIA processors Figure 11.

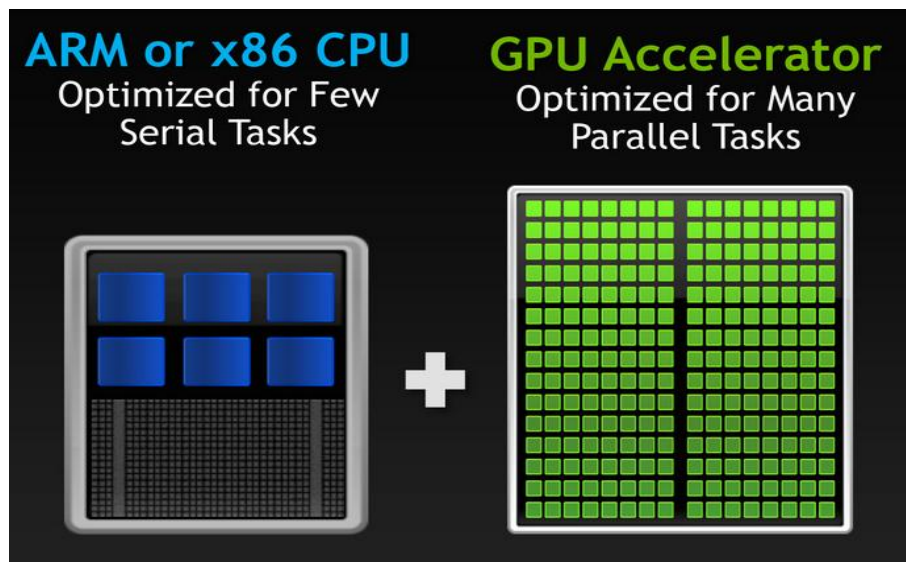


Figure 11. CUDA 5.5 version on ARM.

Along with supporting ARM, CUDA 5.5 introduces cross compilation support between ARM or x86, it allows the ARM binaries to be built natively on ARM platform or much more quickly on

faster x86 system. CUDA 5.5 improves MPI workload prioritization and HyperQ gaining the ability to receive jobs from multiple MPI processes on Linux system.

CUDA Version 6: Intel, AMD and ARM vendors are trying to unify the CPU and GPU memories without deadlocking [26]. But, it is very difficult to design a model, where two processors have to access the same memory without deadlock. It took many years to design that type of model. While all the other are busy in trying a new memory model between CPU and GPU, NVIDIA announced CUDA 6 as the latest version in parallel computing platform and programming model, it has unified memory as a key feature, shown in Figure 12. It cuts the development time by up to 50 percent.

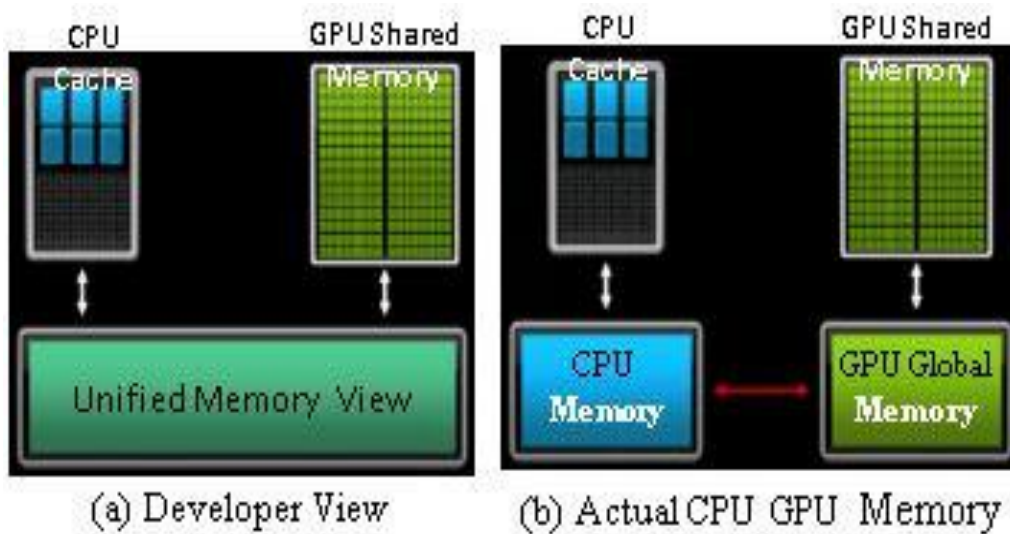


Figure 12. CUDA Memory Model (a) Unified (b) Actual.

In figure 8(b) the CPU and GPU memory connections are represented. The memories of CPU and GPU are distinct and separate physically by using PCI-Express bus. In the case of data sharing between the CPU and GPU, they must allocate the memories, and then it explicitly copied by the program (between them). This makes CUDA program complex. Where, CUDA 6 has unified

memory access from developers point of view (figure 8(a)), which is the shared memory architecture used in parallel computers. Unified memory creates a pool of “managed memory” and is shared among the CPU and GPU, and by using a ‘single pointer’ this managed memory is accessible to both CPU and GPU. The key thing here is the system automatically “migrates” data allocated in unified memory between the host and device. In CUDA 6, We do not need `cudaMemcpy()` as a requirement but we are still free to use `cudaMemcpy()` and `cudaMemcpyAsync()` for performance, all we need is `cudaMallocManaged()` which is worked as a single pointer to data. This makes much easier to write CUDA programs in CUDA 6 because we can go straight to directly writing kernels, rather than writing a lot of data management code and maintaining duplicate copies of all data in host and device.

CUDA 6 has different key features when compared to CUDA 5.5. By replacing existing CPU based libraries, CUDA 6 offers new performance enhancement which enables the developers to accelerate applications instantly by up to 8 times faster. The key features included in CUDA 6 are as follows:

- *Unified Memory* – Unified memory makes the programs simpler because as the name indicates, the applications access memory in a unified way. Applications need not to wait until the data is transferred manually from CPU to GPU and vice versa. It enables us to quickly prototype the kernels running on the GPU because of its automatically handling data management. It makes easier to add support for GPU acceleration in a wide range of programming languages.
- *Drop-in libraries* – Drop-in Libraries make applications more accelerate. A calculation of Basic Linear Algebra Subprograms (BLAS) and Fast Fourier Transformations (FFT) takes very

less time. The time required to execute an instruction is 8 times faster than the time required for other libraries.

- *Multi-GPU scaling* – BLAS and FFT GPU libraries are re-designed. By delivering over nine teraflops of double precision performance per node, the scale performance can increase up to 8 GPUs automatically in a single node. And, compared to the previous workloads, it supports more workloads which are up to 512 GB. Multi-GPU scaling can also be used with the new BLAS drop-in library.

CHAPTER 3

PROPOSED TECHNIQUE

In the proposed technique, we change the way of memory mapping between CPU and GPU memories. The main goal is to increase the overall performance of the GPU shared memory. In the traditional memory mapping technique every single thread from CPU memory is directly copied to global memory in GPU. As we know in multithreaded programming, threads are usually processes data that are not in the consecutive memory locations, so the data from CPU does not fit in the GPU shared memory. Whenever the threads need to execute, the data from the global memory is to access. As the shared memory is an on-chip, and global is off-chip, it is difficult to access the off-chip every time. Considering the performance of GPU, the proposed memory mapping technique is introduced by regrouping the CPU data such a way that the data associated to a same thread can store in a consecutive memory location of GPU's global memory. If the data stored from CPU is not in a consecutive way, then the threads can process the data by taking the memory from shared memory. In proposed mapping, threads use shared memory rather than global memory, the GPU performance will definitely improve the performance compared to traditional memory mapping because of shared memory on-chip nature. Before discussing proposed memory mapping let us see in detailed about traditional memory mapping.

3.1 Traditional CPU-to-GPU Memory Mapping

In traditional, the memory mapping between CPU memory and GPU global memory is the single thread/block data is directly copied from CPU memory to GPU global memory in GPU. We all know that the cache size of a directly mapping technique is of 32 byte block with 128 set of caches. As shown in Figure 13. $X_1, X_2 \dots$ are the data in CPU, and $t_0, t_1 \dots$ are the single thread

processing in CPU. Whereas, if a block need to copy from CPU to GPU global memory, CPU copies 32 bytes of data (i.e. from 0 to 31). After copying the data from CPU to GPU global memory in a consecutive manner, the data/threads from GPU global memory to GPU shared memory are not copied in a consecutive way, which leads the threads to not fit in the shared memory.

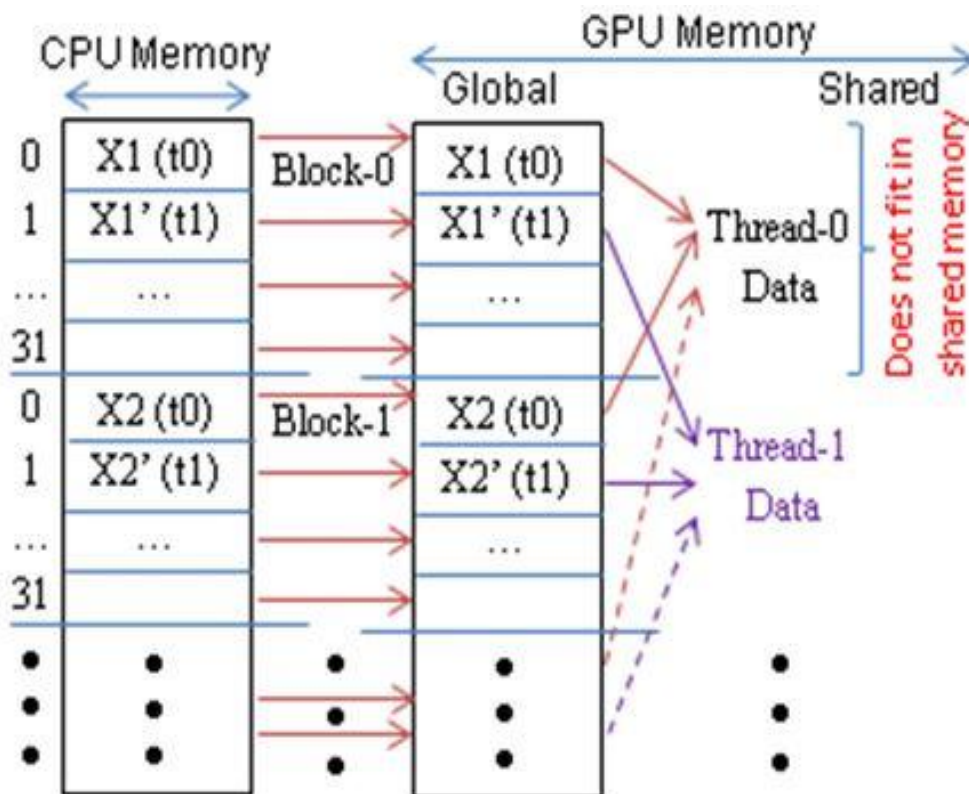


Figure 13. Traditional CPU to GPU Global Memory Mapping.

Let us see the bit of notation regarding matrices. An, ' $n \times m$ ' matrix has ' n ' rows and ' m ' columns and (i,j) entry is the entry in the i th row, j th column. Whereas, ' $n \times n$ ' matrix having same number of rows and same number of columns i.e. ' n '. For computing the product of two $n \times n$ matrices, we multiply the i th row of matrix A by the j th column of matrix B for n products and summing them. Let us see how the data is copied from CPU to GPU with an example of multiplication of

two 4 x 4 matrices. Let, M1 and M2 are the names of the two matrices having the data from input devices and M3 be the resultant matrix, which stores the output of the multiplication of M1 and M2. M1(1,1) represents the position of the matrix/data element i.e. it is in first row and first column of matrix M1.

$$\begin{array}{c}
 \text{M1 Matrix} = \left(\begin{array}{cccc}
 \text{M1(1,1)} & \text{M1(1,2)} & \text{M1(1,3)} & \dots \\
 \text{M1(2,1)} & \text{M1(2,2)} & \text{M1(2,3)} & \dots \\
 \text{M1(3,1)} & \text{M1(3,2)} & \text{M1(3,3)} & \dots \\
 \vdots & \vdots & \vdots & \vdots
 \end{array} \right)
 \end{array}$$

$$\text{M2 Matrix} = \left(\begin{array}{cccc}
 \text{M2(1,1)} & \text{M2(1,2)} & \text{M2(1,3)} & \dots \\
 \text{M2(2,1)} & \text{M2(2,2)} & \text{M2(2,3)} & \dots \\
 \text{M2(3,1)} & \text{M2(3,2)} & \text{M2(3,3)} & \dots \\
 \vdots & \vdots & \vdots & \vdots
 \end{array} \right)$$

$$\text{M3} = \text{M1} * \text{M2}$$

$$= \left(\begin{array}{cccc}
 \text{M1(1,1)} & \text{M1(1,2)} & \text{M1(1,3)} & \dots \\
 \text{M1(2,1)} & \text{M1(2,2)} & \text{M1(2,3)} & \dots \\
 \text{M1(3,1)} & \text{M1(3,2)} & \text{M1(3,3)} & \dots \\
 \vdots & \vdots & \vdots & \vdots
 \end{array} \right) \cdot \left(\begin{array}{ccc|ccc}
 \text{M2(1,1)} & \text{M2(1,2)} & \text{M2(1,3)} & \dots & & & \\
 \text{M2(2,1)} & \text{M2(2,2)} & \text{M2(2,3)} & \dots & & & \\
 \text{M2(3,1)} & \text{M2(3,2)} & \text{M2(3,3)} & \dots & & & \\
 \vdots & \vdots & \vdots & \vdots & & &
 \end{array} \right)$$

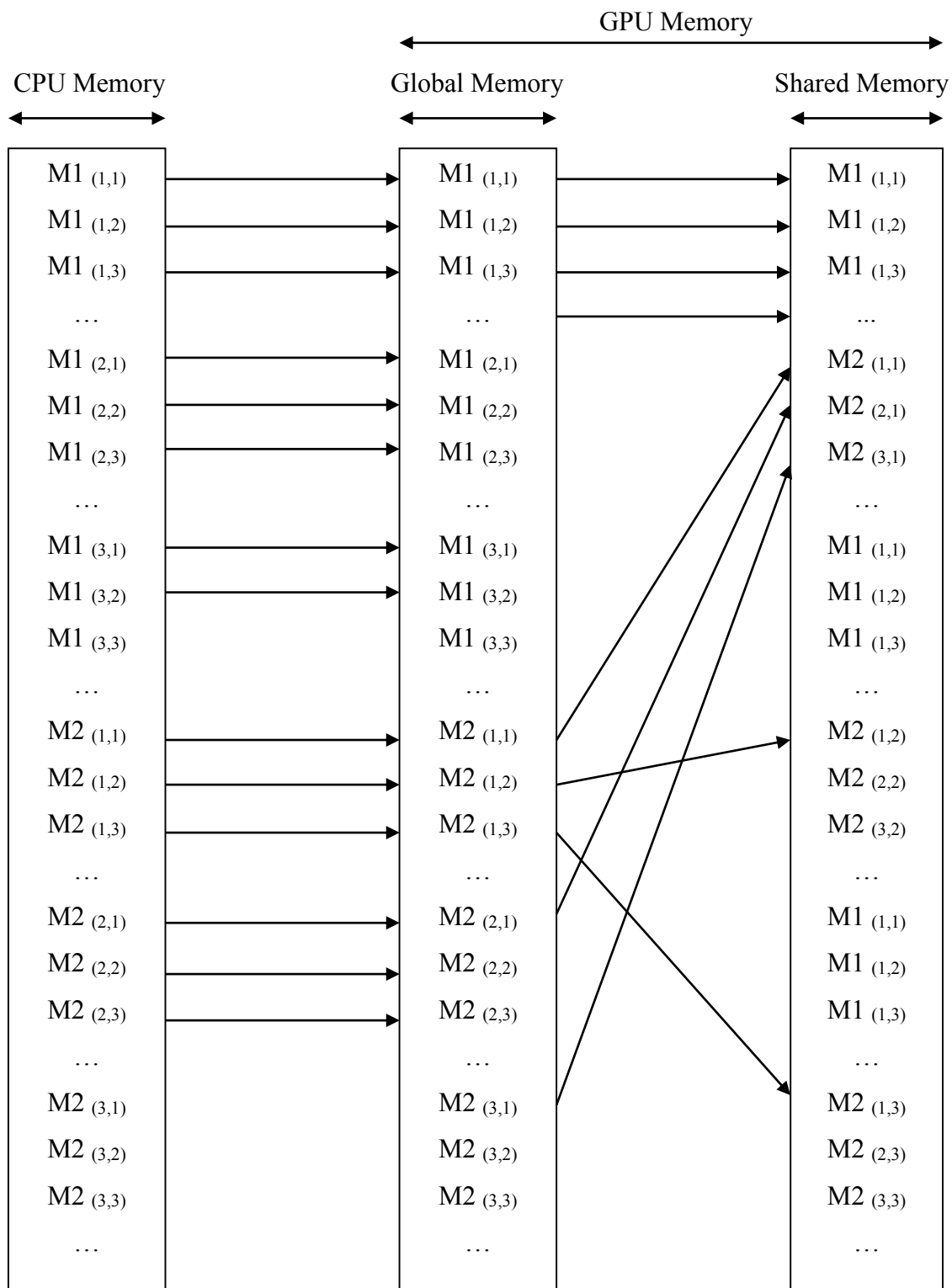


Figure 14. Traditional Memory Mapping for our example.

After assigning the input values to matrices M1 and M2. The data/input values are stored in the memory of CPU in the form of array. From our example, each row of the matrix is stored in a sequential order in the CPU memory. From Figure 14, as the traditional mapping is the copy of single thread or block directly to GPU global memory. From GPU global memory to GPU shared memory, the copying of data from threads are not in the consecutive way. Because, we all know that in a multiplication of two matrices, the top row of the first matrix is multiplied and then added the terms with left most column of the second matrix. That means, in shared memory the data at M1(1,1), M1(2,1), M1(3,1) and M1(4,1), M2(1,1), M2(2,1), M2(3,1) and M2(4,1) to be stored in sequence for computing the first row, first column element of resultant matrix. This calculation is stored in the M3 matrix as an output.

$$M3(1,1) = [M1(1,1) * M2(1,1)] + [M1(1,2) * M2(2,1)] + [M1(1,3) * M2(3,1)] + [M1(1,4) * M2(4,1)] \dots\dots\dots(1)$$

$$M3(1,2) = [M1(1,1) * M2(1,2)] + [M1(1,2) * M2(2,2)] + [M1(1,3) * M2(3,2)] + [M1(1,4) * M2(4,2)] \dots\dots\dots(2)$$

Similarly, for all M3 (i, j) values. For getting clear idea, we choose a small matrix multiplication. In real time scenario, we have to compute big matrices like 500 x 500, etc. This bigger computation performs quickly in GPU because in GPU we have multiple streaming multiprocessor(SM)/grids; each grid has several threads to execute at a time. If the GPU performance is more compared to traditional technique, we can get the results for high computations very quickly. For improving the GPU performance, we came up with a novel proposed technique. We can see the proposed technique in detail in the following section.

3.2 Proposed CPU to GPU Memory Mapping:

We proposed a novel CPU to GPU-memory mapping technique to increase the overall performance. As shown in Figure 15, we already know that $X_1, X_2 \dots$ are the data in CPU, and $t_0, t_1 \dots$ are the single thread processing in CPU. In the proposed mapping, CPU regroups the data such a way that all the data associated to the same thread can be grouped and stored in the consecutive memory locations in GPU global memory.

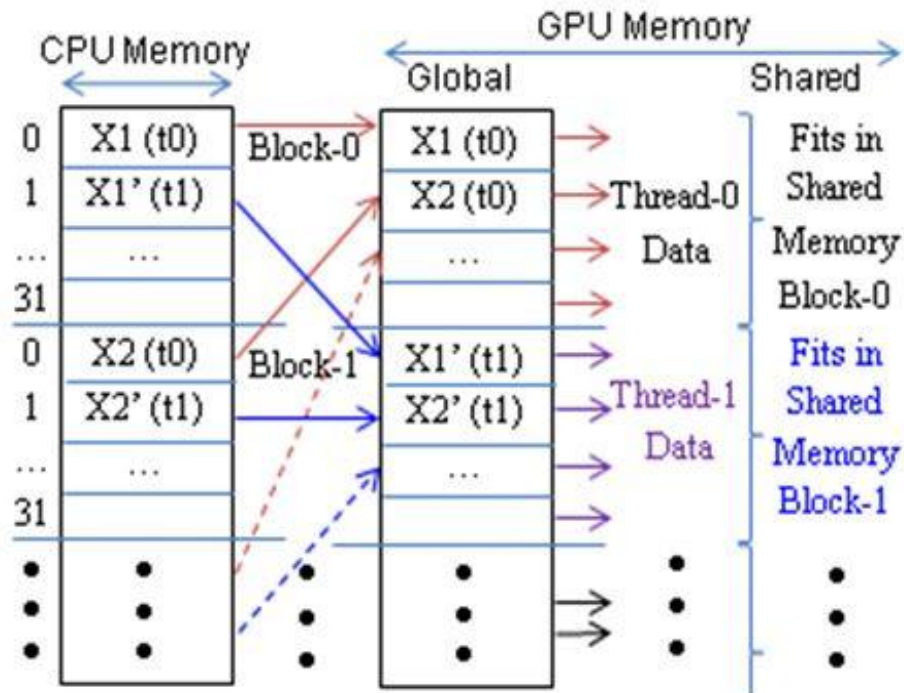


Figure 15. Proposed CPU to GPU Memory Mapping.

Whereas, from GPU global memory to GPU shared memory the threads are directly copied from GPU global memory to GPU shared memory without any regrouping or reordering. The regrouping technique can be applied at copying of data from CPU to GPU global memory only, this is because there is no internal program or technique that regroups the data between GPU global and shared memory.

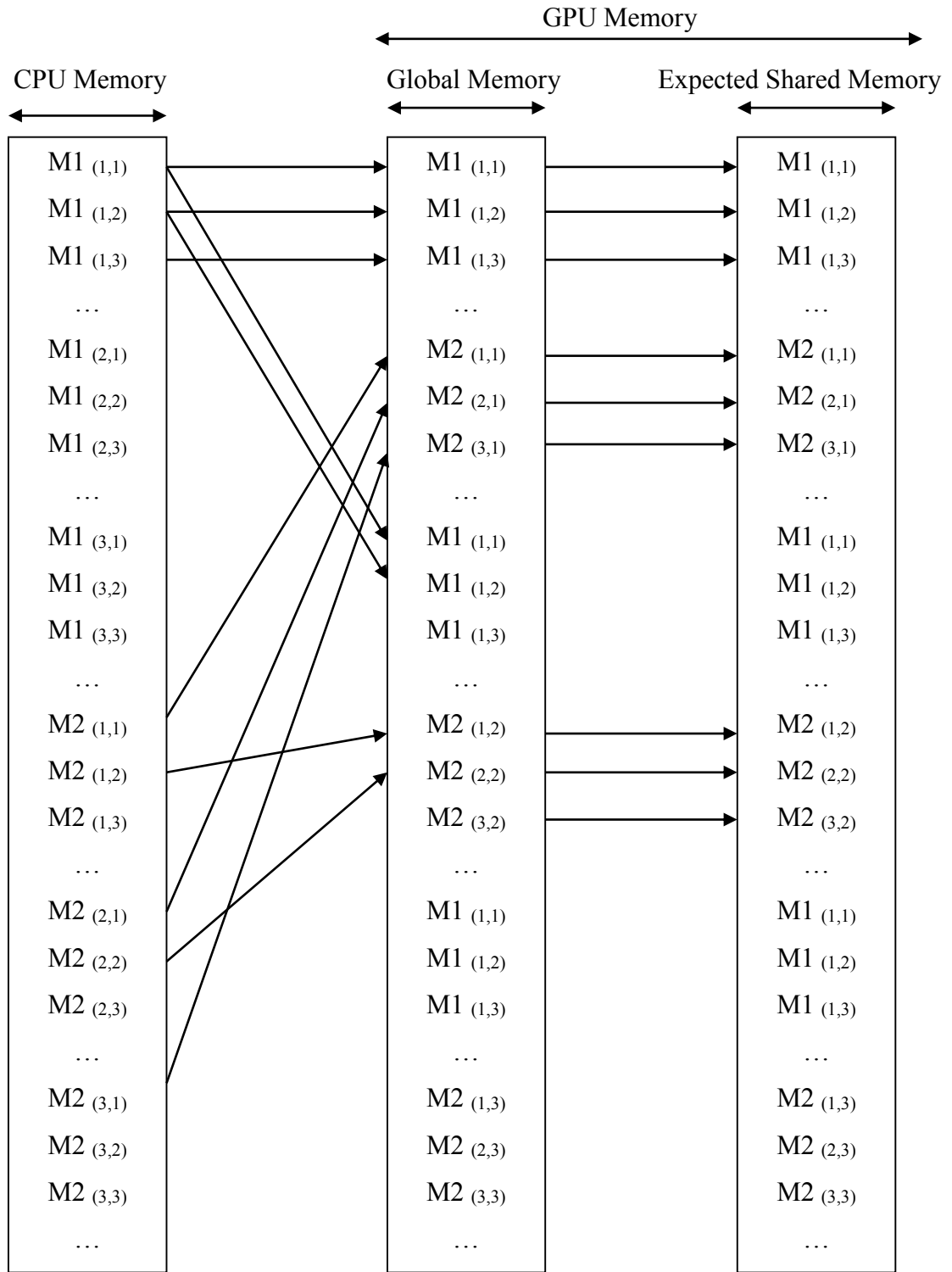


Figure 16. Memory Mapping in the Proposed Technique based on the example.

Figure 16, is the proposed memory mapping for our matrix multiplication. Unlike, the traditional method, this regrouping mapping technique allows the data to fit in the shared memory. As the shared memory is an on-chip memory, all the threads processing within the same grid can access these data very efficiently without any time delay; by this we are expecting the overall processor performance will increase.

The major steps on how a CPU computing transfer's data to GPU computing based on our matrix example.

Start

Step 1: CPU analyzes the input/problem parameters.

Step 2: After analyzing the input parameters, CPU analyzes the GPU card.

Step 3: Then it determines the number of computations and determines the number of threads.

Step 4: Based on the computations and threads CPU identifies/partition the data-blocks for each thread.

Step 5: CPU copies/regroups the data-blocks to GPU Global memory.

Stop

All the above process will be done on CPU side. Let's see step by step briefly with our matrix multiplication example, When the programs need to execute, initially it analyzes the rows and columns of the matrix. Based on the analyzed rows and columns, the CPU needs to analyze the GPU card and then it has to determine the number of threads and number of computations it is required to process; because some blocks of the threads may be one-dimensional, two-dimensional or three-dimensional as the programmer prefers. Since our matrices are two dimensional, we will use two-dimensional blocks. And since we want our code to run on devices of all compute

capabilities, we will use block size of 16×32 so that they contain 512 threads or else in case of bigger matrices we can use higher block size. All the blocks must have the same dimensions as all other blocks, and we must launch enough blocks to cover all entries of the product matrix. Therefore if our matrix's dimensions are not both multiples of 16, then some threads will not be computing elements of the product matrix. Once the block size is decided, the CPU identifies and partition the data blocks for each thread. Finally, it copies the data blocks to the GPU global memory.

Once the data blocks are transferred to GPU, a kernel that works the main program running on the host computer offloads to the GPU for computation on that device. In CUDA, launching a kernel requires three specific things: Dimensions of the Grid, Dimensions of the blocks, and the kernel functions to run on the device. When a kernel is launched on the GPU a grid of thread blocks is created and the blocks are queued to be run on the GPU's multiprocessors. These blocks are given to multiprocessors as they become available, and once started on a multiprocessor, the threads of that block will run to completion on that multiprocessor. The CUDA programming model requires that these blocks be able to compute in any order, that is, the programmer may make no assumptions about the order in which the GPU schedules and runs the blocks of threads. The program must run correctly regardless of the order in which the blocks are scheduled. This restriction may make programming the kernel a bit more difficult, but the benefit is that more parallelism is achieved, and the program will run without modification on cards with more multiprocessors, scaling optimally. From our example, the blocks themselves form a two-dimensional grid of blocks. We are having a grid of 5×7 thread blocks. A 5×7 grid of 16×32 blocks covers 70×200 product matrixes. 5×7 grids of threads create 35 blocks. Although, all

512 threads in every block will run, the first thing our threads will do is to locate themselves to the relative matrix, and the threads that lie outside the matrix will immediately terminate. If a laptop's video card has seven multiprocessors. The 35 blocks of the previous example were to be scheduled on laptop's card; a block would be placed on each multiprocessor, and as the threads of each block terminated, a new block would be placed on that multiprocessor for computation, and so on, until all 35 blocks had been processed. The host code could then be informed that the GPU computation is done, and the host can read the product off of the card.

CHAPTER 4

EXPERIMENTAL DETAILS

In this chapter we are going to discuss some of the assumptions that we made in order to evaluate the results for performance improvement. We considered some of the parameters for conducting the experiment by using proposed technique; CPU/GPU system parameters and 2D electrical charge distribution for experimental setup and reconstructing a smooth surface for large number of threads. We developed CUDA/C code for GPU shared memory. All these are discussed in the following subsections.

4.1 Assumptions

The following assumptions were made in this work:

- In all the techniques considered in this thesis, traditional memory mapping from CPU to GPU and proposed memory mapping between CPU to GPU, we assumed that all blocks go to different SMs.
- We assumed that CPU has 8 cores and each core is of 4MB size.
- On GPU side, we used only Fermi and Kepler cards. We assumed, Fermi card has global memory of size 5.6 GB and shared memory of size 48 KB/Block, and Kepler card has 5.0 GB global memory and 48 KB/Block shared memory.

4.2 System Parameters

In this section, the CPU/GPU system parameters for the workstations are tabulated. For observing the performance of the proposed techniques we mainly used two popular GPU cards with a multicore CPU: Fermi card and Kepler card. On the CPU side we use Intel Xeon processor

and on the GPU side we used two different GPU cards for evaluating the time required for GPU to compute. We mainly conducted this experiment on three different versions for two different GPU cards. Those versions are on CPU only (by using CPU/C Program- sequential processing), GPU without shared memory and GPU with shared memory (by using CUDA/C Program - parallel processing).

The key features are tabulated below in Table-2 and Table-3 [27, 28]. Table-2 indicates the type of processor that we used for this experiment and their speeds. Number of cores and the memory sizes of CPU, GPU global and shared memories are tabulated in Table-3.

TABLE 2

Specifications of CPU and GPU processors.

Parameter	Description
CPU	Intel Xeon
Fermi GPU Card	NVIDIA Tesla C2075
Fermi Clock Speed	1.15 GHz
Kepler GPU Card	NVIDIA Tesla K20m
Kepler Clock Speed	0.71 GHz
Operating System	Linux Debian

Let us see some details about CPU Intel Xeon dual processor workstation, GPU Fermi card and GPU Kepler card specifications. The operating systems used here is Linux Debian 6.0. There are many types of Intel Xeon dual core processors available at different clock speeds, but we considered the workstation runs at 2.13 GHz and it supports 2MB advanced smart cache for L2.

The first 'complete' architecture for GPU computing is "Fermi" as it delivers any kind of features required for the most high performance computing applications [27]. It brings supercomputer performance to more users than ever before. A Fermi card has 14 SMs and each SM has 32 CUDA cores, each of which can perform floating-point and integer operations. There are many warps in each blocks/grids, and there are many blocks in a GPU. In the Fermi GPU card, threads executes the different warps from different threads simultaneously to increase the hardware utilization and energy efficiency. These group of thread blocks executes when a unique kernel is launched. The kernel relationship is specified by threads and thread blocks. Each data (input and output) in the GPU memory has corresponding memory IDs which are worked as indexes. Compared to the previous generation GPUs in switching applications, Fermi GPU card is 20 times faster. Fermi GPU card is introduced to maintain high utilization on multiple applications, the logic behind this vendors use both the combination of graphics code and computer code. GPU architecture has GigaThread, which manages hardware scheduler to switch the context among the threads, each GigaThreads can handle thousands of active threads at the same time across 16 kernels on each streaming multiprocessor. The data is transferred twice between system memory and GPU memory, initially at setting up a GPU computation and secondly after executing the results move back from GPU memory to system memory.

Fermi has 16 load per store unit and is used to handle all the memory operations in each SM. We all know that memory location with address are referred to as indexes. Each streaming multiprocessor has some load/store instructions, in which the memories of these instructions are stored in terms of two-dimensional arrays. For this experiment, we considered NVIDIA Tesla C2075 GPU card this can be configured by the end user to enable or disable Error Correcting

Codes (ECC) that can fix single-bit errors and report double-bit errors. Enabling ECC will cause some of the memory to be used for the ECC bits so the user available memory will decrease to 5.25 GB. The key features of Fermi GPU Tesla are tabulated below in Table -2.

NVIDIA Kepler GPU architecture and is the world's most pervasive parallel computing model [28]. SM, Dynamic Parallelism, and Hyper-Q are included in the Kepler compute architecture design. When compared to the SM in Fermi, the SM delivers 3x more performance per watt and also delivers one petaflop of computing within ten server racks. There is property in GPU threads called Dynamic Parallelism, which enables the GPU threads to automatically spawn for new threads. Along with dynamic parallelism, Kepler architecture enables GPU acceleration of popular algorithms, like Adaptive Mesh Refinement (AMR), fast multipole method (FMM), and multi-grid methods. On a single Kepler GPU card, multiple CPU cores are enabled by Hyper-Q to simultaneously utilize the CUDA cores. Hyper-Q dramatically increases the GPU utilization, and it cuts the CPU idle times, and advances programmability— Kepler card is ideal for cluster applications that use MPI. Tesla GPU computing is supported by both Linux and Windows platform. 64-bit operating system supports the server modules whereas 32-bits support workstation and desktop modules.

NVIDIA Tesla K20m Kepler GPU card is used for this experiment. Tesla K20m accelerators mainly designed to be the performance leader on applications like in double precision and the broader supercomputing market; it delivers 10 times faster the performance of a single CPU. Tesla K20 features includes the Dynamic Parallelism and Hyper-Q features. GPU accelerators are mainly ideal for the high-performance computations, some of the applications like

climate and weather modeling, biomedical computations, computational physics, biochemistry simulations, and computational finance.

TABLE 3

Core and Memory specifications of CPU and GPU cards.

Parameter	Description
CPU Cores	8
CPU RAM	6 GB
Fermi GPU Cores	448
Fermi Global Memory	5.6 GB
Fermi Shared Memory	48 KB/Block
Kepler GPU Cores	2496
Kepler Global Memory	5.0 GB
Kepler Shared Memory	48 KB/Block

4.3 Experimental Setup

Let us see experimental setup for Fermi based architecture. Similar to other GPU cards, Fermi architecture has local memory in each block. The advantage about Fermi card is the ability to use some part of local memory as a level-1 cache (CL1). Local memory is of size 64K in Fermi card, and this memory can be split into 16K/48K or 48K/16K among L1 cache and shared memory. Shared memory being the traditional memory, in addition that the use of local memory provides low-latency access to moderate amounts of data. The access latency to the local memory is completely predictable and so the algorithms can be written to interleave load calculations, and

stores with maximum efficiency. Allocation of some of the local memory (i.e. 16K or 48K) as cache is usually depends on two factors - one of the factor is to check how much shared memory is needed, and the second factor is to check, how predictable the kernel's accesses to the global memory (usually the off-chip DRAM). Very less cache memory capacity is needed for a larger shared-memory; but larger regions of DRAM argue for more cache because of its more frequent utilization or unpredictable accesses. Each Fermi GPU is equipped with level-2 (L2) cache, and for 512-core chip processors L2 cache is of size 768KB. Another feature implemented by L2 cache subsystem is read-modify-write operations are atomic and uninterruptible, which is not found on CPUs. And L2 cache covers GPU local DRAM, and thus it is ideal for managing the data which is shared across thread blocks or even kernels.

Compared to the previous generation GPUs, Fermi based atomic operations are 5 to 20 times faster, this is because of inbuilt conventional synchronization method. And, the final processing stage of the local memory hierarchy is DRAM, which is directly connected to GPU and works like a global memory in the GPU. Among the other GPU processing cards, Fermi card is the first processing card to have Error Correcting Coding (ECC) protection. ECC protection is provided to the GPU parts like - DRAM, register files in the chip, shared memories, L1 and L2 caches. Among all the parts ECC protection for DRAM is unique; so is its implementation. Each memory channel in the GPU carries 64-bit location and for ECC protection it need extra 8 bits to store the ECC information. To avoid this extra memory utilization, NVIDIA introduced a proprietary solution. The solution is packing the ECC bits into reserved lines of memory.

Figure 17, illustrates the experimental setup on CPU-GPU computation. Processor and memory specifications are diagrammatically represented in this figure.

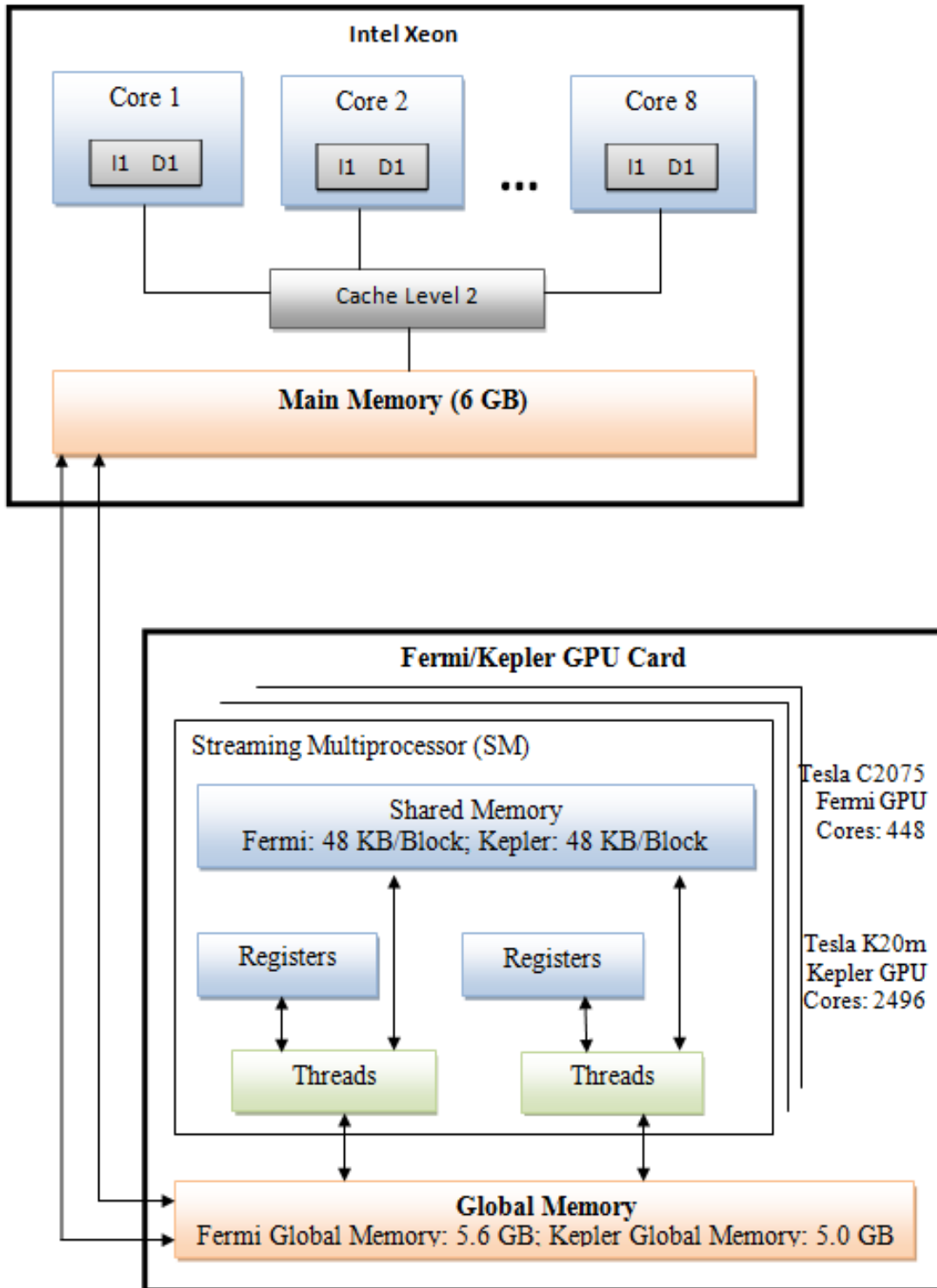


Figure 17. Experimental Setup on CPU-GPU Cards.

4.4 Applications

High performance computations are required in many of the applications like Nanotechnology, Bioinformatics, Data center, etc. Laplace's equation takes advantage on high charge distribution if we use that in high performance computing applications it will give better performance. For example, many composite formations/mixtures, especially those encountered in electromagnetic meta-material applications, often consist of periodic arrangement of unit elements [29]. Let us consider a two dimensional aluminum alloy sheet, internally it has many atoms. Aluminum alloy is mostly used in aircraft applications because they are good conductors for electricity. If the lightning strikes the metal surface, the electric charge is distributed evenly throughout the surface. To avoid this, aluminum alloy needs the protection from lightning strike, so someone has to understand the behavior of heterogeneous components. There are many cells in each component; CPU/C program alone can take lot of time to process this many number of computations. To take advantage of parallel programming, this high performance computation can be executed in fraction of seconds with the help of multithreaded multicore/many core CPUs and General Purpose Graphics Processing Unit (GPGPUs).

Here we considered the Laplace's equation for two dimensional electric charge distributions as the problem. Why this Laplace's equation is because Laplace equation is widely used in Finite Element Analysis, such as lightning strike analysis. Each cell represents electric charge, and this equation simulates the behavior of charge distribution. If there are many cells in a surface, it requires more number of iterations. Each iteration in laplace equation is independent, and spatial locality holds. The non-homogeneous version of Laplace's equation is called Poisson's equation [30]. In many cases, when the charge distribution is not known, the Poisson's equation

can be used to solve electrostatic problems. In our work, we will use electric charge distribution with Laplace's equation.

The Laplace's operator on the 2D electric potential function over a region of the space where the charge density is not zero, the Poisson's equation is follows in Equation 3.

$$\nabla^2 \varphi = \partial^2 \varphi / \partial x^2 + \partial^2 \varphi / \partial y^2 = -\rho / \varepsilon \dots\dots\dots (3)$$

If the charge density is zero all over the region, the Poisson's equation becomes Laplace's equation.

$$\nabla^2 \varphi = \partial^2 \varphi / \partial x^2 + \partial^2 \varphi / \partial y^2 = 0 \dots\dots\dots (4)$$

A composite mixture used in electromagnetic meta-material applications often consists of periodic arrangement of unit elements whose sizes are much smaller than the operating wavelength. Therefore, Laplace's equation can be simplified applying the quasi-static assumptions as shown in Equation 5, where ε is the medium permittivity and φ represents the electric potential.

$$\nabla \cdot (\varepsilon \nabla \varphi) = 0 \dots\dots\dots (5)$$

Now based on the finite-difference approximations, Equation 5 can be rewritten as Equation 6.

$$\begin{aligned} &\varepsilon_{x(i,j)} (\varphi_{i+1,j} - \varphi_{i,j})/dx + \varepsilon_{y(i,j)} (\varphi_{i,j+1} - \varphi_{i,j})/dy + \\ &\varepsilon_{x(i-1,j)} (\varphi_{i,j} - \varphi_{i-1,j})/dx + \varepsilon_{x(i,j-1)} (\varphi_{i,j} - \varphi_{i,j-1})/dy = 0 \dots\dots\dots(6) \end{aligned}$$

Where,

dx and dy are the spatial grid size,

$\varphi_{i,j}$ is the electric potential defined at lattice point (i, j) , and

$\epsilon_{x(i,j)}$ and $\epsilon_{y(i,j)}$ are the effective x- and y-direction permittivity defined at edges of the element cell (i, j).

For very uniform material, electric potential can be considered the same in all directions.

Therefore, Equation 6 becomes a 2D problem as shown in Equation 7 and can be solved using the discrete approach.

$$(\phi_{i+1,j} - \phi_{i,j})/dx + (\phi_{i,j+1} - \phi_{i,j})/dy + (\phi_{i,j} - \phi_{i-1,j})/dx + (\phi_{i,j} - \phi_{i,j-1})/dy = 0 \dots(7)$$

The multithreaded CUDA/C shared memory implementation of the 2D Poisson's equation for charge distribution is shown in Figure 19. Here, the right values of i (i.e., current threadIdx.x) and j (i.e., threadIdx.y) for each thread, and the shared variables As[i][j] are used for memory latency hiding optimization. Thread executions are synchronized to ensure correctness.

4.5 CUDA/C Implementation

```

__global__ void LSP2D_CUDA_SM(float *A, float *B, int N) {
    int index;
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    index = i * N + j;
    __shared__ float As[32][32]; /* shared between 32 threads per block */
    if ((threadIdx.x < 32) && (threadIdx.y < 32)) {
        As[threadIdx.x][threadIdx.y] = A[index];
    }
    __syncthreads();

    if ((threadIdx.x < 32) && (threadIdx.y < 32) &&
        (threadIdx.x > 0) && (threadIdx.y > 0)) { /* retain boundary */
        B[index] = (As[threadIdx.x - 1][threadIdx.y] * Xs[threadIdx.x - 1][threadIdx.y] +
                    As[threadIdx.x + 1][threadIdx.y] * Xs[threadIdx.x + 1][threadIdx.y] +
                    As[threadIdx.x][threadIdx.y] * Xs[threadIdx.x][threadIdx.y] +
                    As[threadIdx.x][threadIdx.y - 1] * Xs[threadIdx.x][threadIdx.y - 1] +
                    As[threadIdx.x][threadIdx.y + 1] * Xs[threadIdx.x][threadIdx.y + 1]) / 5.0;
    }
    /* end if */
}
/* end LSP2D_CUDA_SM */

```

Figure 18. Sample CUDA/C Implementation Code.

CUDA/C shared memory implementation of the Laplace's equations for charge distribution of NXN 2D space is shown above.

4.6 Input/Output Variables

Input variables are the input/computation parameters (for a matrix input parameters are size of rows and size of columns (say n,m)) and output parameter is time. The time required to execute the applications with the proposed technique is considered to be less when compared to the time required by the traditional technique. The logical reason for this kind of assumption is the basic behaviour of a computing components. When the data is copied from the CPU memory to GPU, the threads checks for the data in the shared memory. If the data is in shared memory the threads working in that grid/block will access that data at a full speed. If the data is not in the shared memory, our traditional memory mapping, threads need to access the GPU global memory for the required data. Traditional memory mapping certainly consumes more energy and time until the particular data to be executed.

For example, to check the volume of all the atoms in Aluminium sheet, we need to check the size of the atom/element and the radius of the single atom. Based on the size of the metal, the atoms are arranged with the matrix format. Similarly, in the case of electric charge distribution. But, once the charge distributions starts it distributes the computed values to the neighboring elements. We can see in detailed in the validation of CUDA/C implementation code.

CHAPTER 5

EXPERIMENTAL RESULTS

In this chapter, we conduct a study on Laplace equation for high electric charge distribution on 2D thin surface with three different versions to observe the experimental results. The three different versions are (i) CPU only, (ii) GPU with shared memory and (iii) GPU without shared memory. We conducted this experiment based on the proposed memory mapping and we used two different GPU cards (Fermi and Kepler). The results obtained are accounted and analyzed. Accordingly, graphs are plotted to show the measure of time required to execute.

5.1 Validation of CUDA/C Implementation

The developed CUDA/C program is to solve the application using proposed technique. We first validate only CUDA/C program for that we consider an 8x8 matrix. After copying the data from CPU to GPU, Node (4,4), Node(4,5), Node(5,4), and Node(5,5) are initially set with a high value of 10000 and all other nodes are set to a low value 0 (zero). As shown in Figure 19, nodes right outside of the 8x8 matrix are also set to a low value 0 (as a boundary condition).

Iteration 0	1	2	3	4	5	6	7	8		
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	2
0	0	0	0	0	0	0	0	0	0	3
0	0	0	0	10000	10000	0	0	0	0	4
0	0	0	0	10000	10000	0	0	0	0	5
0	0	0	0	0	0	0	0	0	0	6
0	0	0	0	0	0	0	0	0	0	7
0	0	0	0	0	0	0	0	0	0	8
0	0	0	0	0	0	0	0	0	0	0

Figure 19. An 8X8 Matrix with Boundary Condition at Iteration 0.

When the GPU starts executing the instructions for an 8X8 matrix, the loops within the program runs repeatedly till all the threads for 8X8 matrix needs to execute. After the initial stage (i.e. Iteration 0), the GPU divides the data among grids and then divided data in each grid is divided among threads. For the next iteration or loop the values are spread to the neighboring positions.

Following is the Figure 20 at Iteration 1.

Iteration 1	1	2	3	4	5	6	7	8	
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	2000	2000	0	0	0	0
0	0	0	2000	6000	6000	2000	0	0	0
0	0	0	2000	6000	6000	2000	0	0	0
0	0	0	0	2000	2000	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Figure 20. An 8X8 Matrix with Boundary Conditions at Iteration 1.

Similarly, for each iteration the threads share the data and execute the instructions. Following Figures 21-23, indicates the matrix values at Iteration-10, 50 and at 100.

Iteration 10	1	2	3	4	5	6	7	8	
0	0	0	0	0	0	0	0	0	0
0	64	149	246	313	313	246	149	64	0
0	149	339	547	691	691	547	339	149	0
0	246	547	872	1094	1094	872	547	246	0
0	313	691	1094	1367	1367	1094	691	313	0
0	313	691	1094	1367	1367	1094	691	313	0
0	246	547	872	1094	1094	872	547	246	0
0	149	339	547	691	691	547	339	149	0
0	64	149	246	313	313	246	149	64	0
0	0	0	0	0	0	0	0	0	0

Figure 21. An 8X8 Matrix with Boundary Conditions at Iteration 10.

Iteration 50	1	2	3	4	5	6	7	8		
0	0	0	0	0	0	0	0	0	0	0
0	17	32	44	50	50	44	32	17	0	1
0	32	61	83	95	95	83	61	32	0	2
0	44	83	113	129	129	113	83	44	0	3
0	50	95	129	147	147	129	95	50	0	4
0	50	95	129	147	147	129	95	50	0	5
0	44	83	113	129	129	113	83	44	0	6
0	32	61	83	95	95	83	61	32	0	7
0	17	32	44	50	50	44	32	17	0	8
0	0	0	0	0	0	0	0	0	0	0

Figure 22. An 8X8 Matrix with Boundary Conditions at Iteration 50.

Iteration 100	1	2	3	4	5	6	7	8		
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	1	1	0	0	0	0	2
0	0	0	2	2	2	2	0	0	0	3
0	0	1	2	3	3	2	1	0	0	4
0	0	1	2	3	3	2	1	0	0	5
0	0	0	2	2	2	2	0	0	0	6
0	0	0	0	1	1	0	0	0	0	7
0	0	0	0	0	0	0	0	0	0	8
0	0	0	0	0	0	0	0	0	0	0

Figure 23. An 8X8 Matrix with Boundary Conditions at Iteration 100.

Using the CPU/C and CUDA/C codes we calculate the new value of the matrix as stated in Equation 5. Where, $1 \leq n \leq 8$ and $1 \leq m \leq 8$.

$$N_{n,m} = 1/5 (N_{n,m-1} + N_{n,m+1} + N_{n,m} + N_{n-1,m} + N_{n+1,m}) \dots \dots \dots (5)$$

The program stops when each and every node has a value less than 1. Figure 24, shows values for Node (1,1), Node(3,4), Node(5,5) and Node(8,8) after iterations 0, 1, 10, 50 and 100. As expected, it is observed that both CPU/C and CUDA/C versions produce exactly the same value for each

node after any number of iteration. Following graph represents the validation of CPU and GPU computing.

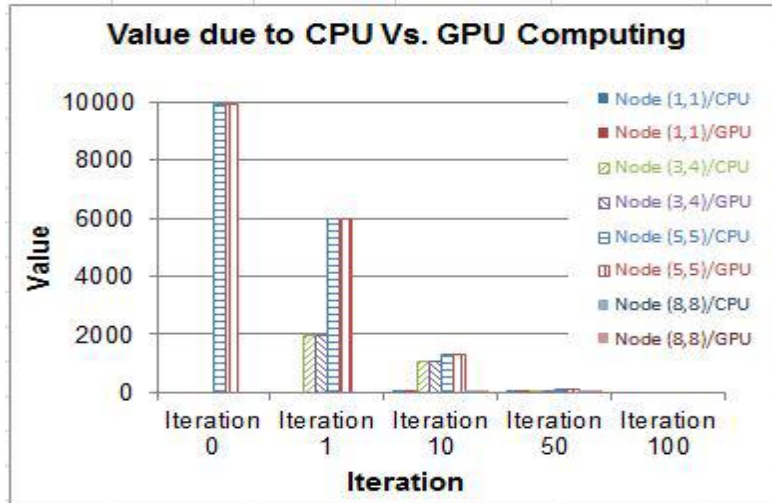


Figure 24. Validation of the Developed CUDA/C Implementation.

5.2 Impact of having GPU Shared Memory

In this section, we will discuss the impact of having GPU with shared memory and without shared memory. The GPU card we used here is Fermi. The graph plotting the filled rhombus (i.e. blue line) represents the GPU card with shared memory usage of size 32 KB. Whereas, the graph plotting the rhombus shapes with a hallow inside (i.e. black line) represents the GPU card without shared memory. Figure 25 illustrates that the execution time decreases as the number of threads increases. Results show that for small number of threads (less than 8), GPU card without shared memory takes less time compared to with-shared memory; but for large number of threads (greater than 8), GPU card with shared memory takes less time compared to without shared memory. The reason for no shared memory takes less time for less number of threads is data can be accessed easily from any where from global memory; but for more number of threads access GPU global memory is time taking, whereas with the shared memory, the more number of threads can access

the data easily from the shared memory compared to less number of threads. The time is efficient for higher number of threads with shared memory. For this reason, the proposal technique takes advantage in higher computations.

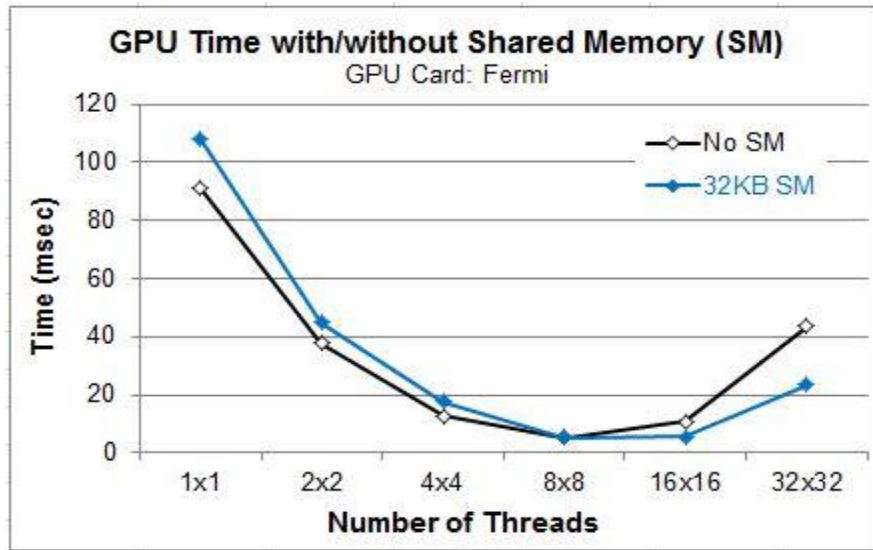


Figure 25. Impact of GPU Performance on with and without Shared Memory.

5.3 Impact of the Number of Threads

In this section of results, by using proposed memory mapping technique we are observing the amount of time required to process a particular number of threads. The shared memory size for a single SM is 32KB, which is kept to be constant. Figure 26, shows the graph with plotted line of rhombus and triangles. The line joining rhombus shapes indicates the Kepler card (i.e. Orange colored line) and line joining triangle shapes indicates the Fermi card graph (i.e. Blue colored line). The impact of number of threads and it is the combination of both GPU cards that is Fermi and Kepler. By observing the Figure, it clearly indicates that as the number of threads increases the amount of time required to process the threads is decreased. Among the Fermi and Kepler cards, for small number of threads (less than 8) Kepler is taking more time compared to Fermi this is

because Fermi's clock speed(1.15Hz) is high compared to Kepler(0.71Hz). And for large number of threads(greater than 16), Fermi is taking more time compared to Kepler because of Fermi's load/store unit(Fermi having 16 units in each SM whereas Kepler has 32 units).

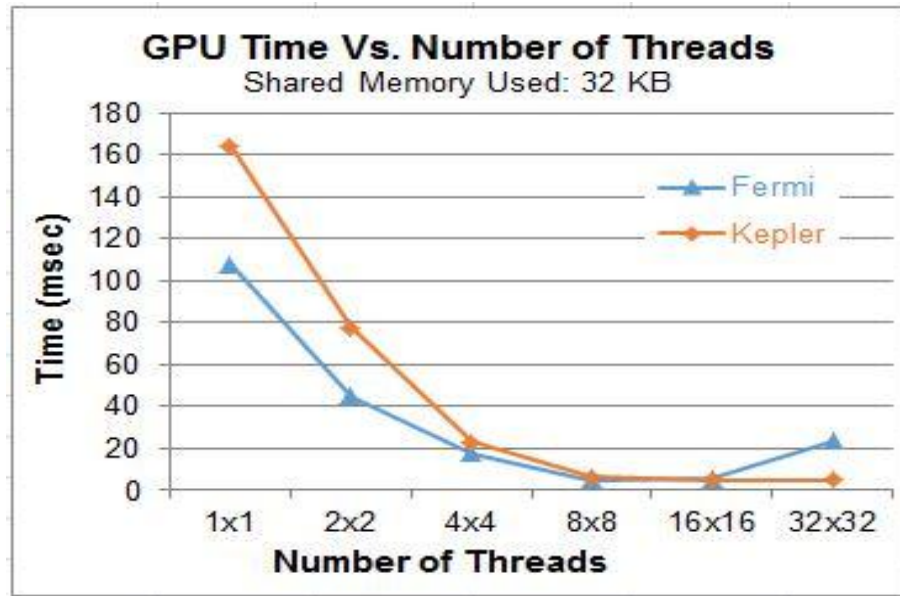


Figure 26. GPU Time Vs Number of Threads.

5.4 Impact of the amount of shared memory

In this section, we want to compare the performance of both Fermi and Kepler cards. For observing the performance time for both the cards, we run our program by keeping the number of threads as constant (i.e. for 16 threads) and the used GPU shared memory of sizes 4, 8, 16 and 32. As shown in Figure 27, the graph representation is same with the section 5.3 that is impact of the number of threads, rhombus plotted graph represents Kepler card and triangle plotted graph represents the Fermi card. Both times decreases as the GPU shared memory size increases. It is noted that the decreases in execution time for Fermi is much faster than Kepler; this is probably because Fermi runs faster than Kepler.

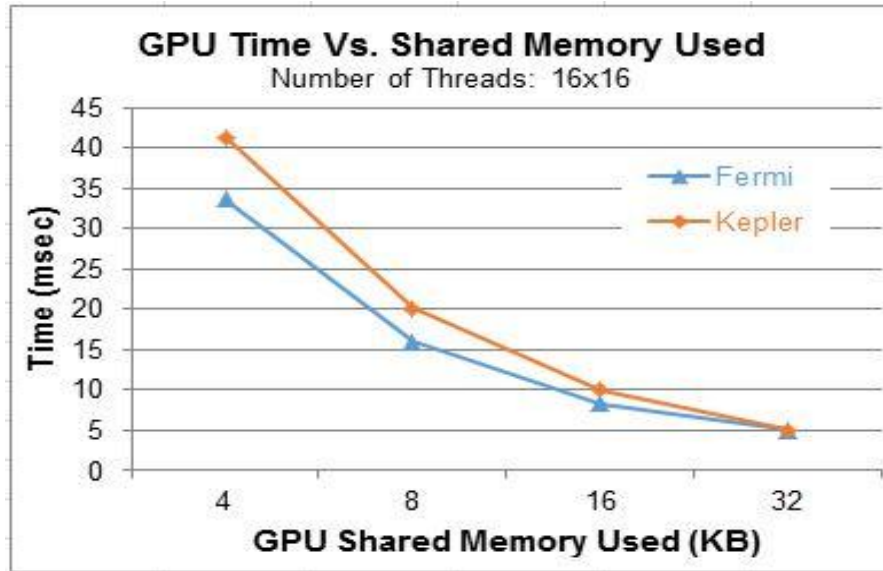


Figure 27. GPU Time Vs Used Shared Memory.

5.5 Impact of Proposed CPU-Memory Data Regrouping Technique

Finally, we evaluate the impact of the proposed CPU-to-GPU memory mapping technique with data regrouping. From the Figure 28, the line joining the square plots represents the global memory without shared memory (i.e. Orange colored line), the line joining the rhombus plots represents the global memory with shared memory (i.e. Blue colored line) and the line joining the plots of all triangles represents the global memory with shared memory and data regrouping (i.e. Green colored line). Graph shows the execution times due to solving the Laplace's equation for electric charge distribution on a 512x512 thin surface. For the number of threads less than 9x9, GPU global memory without shared memory takes less time compared to GPU global memory with shared memory, this we already observed in section 5.2, whereas GPU with shared memory shows improvement for the number of threads between 9X9 and 16X16. For more than 16x16 threads execution time increases, this probably due to the limitation of 16 load/store units. Experimental

results direct that the proposed CPU-to-GPU memory mapping with GPU shared memory and data regrouping provides the best performance compared to with and without shared memory.

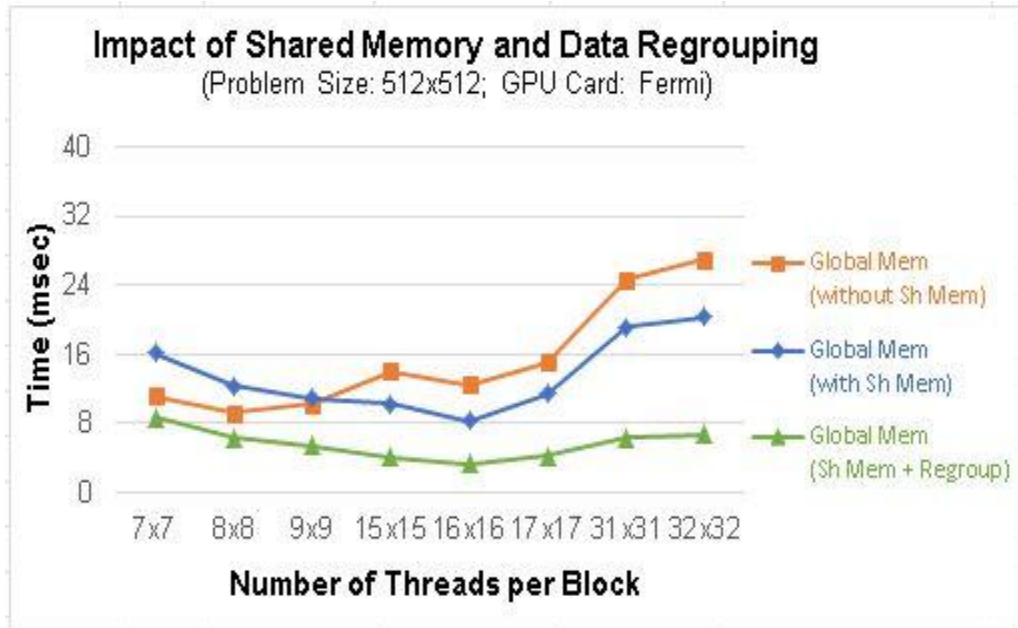


Figure 28. Impact of Data Regrouping.

CHAPTER 6

CONCLUSION AND FUTURE SCOPES

We hope the discussion presented in the thesis motivates the interested scholars into considering research in the challenging but prosperous area of GPU computing. Manycore architecture is the future of all modern computing areas from server to desktop to possible embedded environment. With the appropriate usage, the potential of manycore systems can be enormous. In this work, our contributions lead to improve the performance of manycore NVIDIA GPU by reducing the communication. This chapter concludes our work and we offer some possible future extension of this work.

6.1 Conclusion

It is proven that proposed memory mapping from CPU to NVIDIA CUDA-accelerated GPU provides better performance for big data applications. GPU computing has potential to provide faster and cheaper solutions to address massively large/complex problems. However, in current CPU/GPU computing, CPU-data is first copied into GPU global memory but it is not suitable to improve the GPU performance. As GPU global memory is larger than the GPU shared memory, it would be beneficial to keep the data in the GPU shared memory. In this paper, we proposed a new memory mapping between CPU memory and GPU global memory (to GPU shared memory) to enhance GPU and overall system performance. By using new memory mapping, we solve the Laplace's equation for electric charge distribution on a 2D thin surface by using three different versions; that is by using only CPU, CPU/GPU computing without shared memory, and CPU/GPU computing with shared memory. For this experiment we use NVIDIA Fermi GPU card

with 448 cores and Kepler GPU card with 2496 cores. According to the experimental results, it clearly directs the usefulness of GPU shared memory for both the Fermi and Kepler GPU cards. Using a Fermi GPU card for a 512x512 2D problem space, the proposed CPU-to-GPU memory mapping technique improves the system performance by cutting down the execution time by more than 75%.

6.2 Future Extensions

This work, especially the new CPU memory regrouping technique can be extended to scope many important research areas including:

- *Improved CPU-to-GPU memory mapping:* In this regrouped memory mapping, we might have the duplicate data. Avoiding the duplicate data in transferring the data from CPU to GPU will lead a better GPU performance.
- *Modeling and simulation of Nanocomposites:* Nanocomposites is the primary challenge because it requires large number of computations at high speed. Proposed technique can be extended to assist in nanocompositions will significantly reduce the processing time.
- *Aircraft applications:* To study the composite materials for aircraft applications will need the better CPU/GPU computing for faster computation, if that better computing has this proposed memory mapping will improve give faster computational analysis.

REFERENCES

REFERENCES

- [1] Pawel, G., and Kowalik, M.F., "Multicore Processors: New Way to Achieve High System Performance," *IEEE proceedings of the International Symposium on Parallel Computing*, 2006; doi: 0-7695-2554-7/06
- [2] Creel, M., and Goffe, W. L., "Multi-core CPUs, Clusters, and Grid Computing: a Tutorial," Volume 32, Issue 4, pp. 353-382, 2008; doi: 10.1007/s10614-008-9143-5
- [3] Schauer, B., "Multicore Processors – A Necessity," Proquest discovery guides, [online] URL: <http://www.csa.com/discoveryguides/multicore/review.pdf>. 2008.
- [4] Bryan, C., "An Introduction to CUDA/OpenCL and Manycore Graphics Processors," *NVIDIA Research*, [online] URL: http://www.cs.berkeley.edu/~demmel/cs267_Spr12/Lectures/CatanzaroIntroToGPUs.pdf, 2012.
- [5] Luque, C., Moreto, M., Cazorla, F.J., Gioiosa, R., Buyuktosunoglu, A., and Valero, M., "CPU Accounting for Multi-core Processors," *Computers, IEEE Transactions on*, vol.61, No.2, pp. 251-264, 2012; doi: 10.1109/TC.2011.152.
- [6] Barroso, L.A., Gharachorloo, K., McNamara, R., Nowatzyk, A., and Qadeer, S., "Piranha: A scalable architecture based on single-chip multiprocessing," *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, B.C, 2000, pp. 282-293.
- [7] Wolfe, V. F., "How the Computer Works: The CPU and Memory," [online], URL: <http://homepage.cs.uri.edu/faculty/wolfe/book/Readings/Reading04.htm>. 2005.
- [8] Asaduzzaman, A., "A power-aware multi-level cache organization effective for multi-core embedded systems," *Journal of Computers*, Vol. 8, No. 1, 2011.
- [9] Asaduzzaman, A., "Cache optimization for real-time embedded systems," Ph.D. Dissertation, Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, 2009, ISBN-13: 9781243688828.
- [10] Garrison, P., Jain, R., "Overview of Performance Measurement and Analytical Modeling Techniques for Multi-core Processors," [online], URL: <http://www.cse.wustl.edu/~jain/cse567-11/ftp/multicore.pdf>, 2011.
- [11] Asaduzzaman, A., Sibai, F.N., and Manira, R., "Impact of level-2 cache sharing on the performance and power requirements of homogeneous multicore embedded systems," *Elsevier 2009 on Microprocessors and Microsystems*, pp: 388-397, 2009 doi: 10.1016/j.micpro.2009.06.001

- [12] NVIDIA Developer Zone, [online], URL: <http://www.nvidia.com/object/what-is-gpu-computing.html>, 2007.
- [13] Manuel, U., “Many-core GPUs: Achievements and Perspectives,” *NVIDIA CUDA Fellow*, [online], URL: <http://icpp2013.ens-lyon.fr/GPUs-ICPP.pdf>, 2013.
- [14] Yang, Y., Xiang, P., Mantor, M., Zhou, H., “CPU-assisted GPGPU on fused CPU-GPU architectures,” *IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*, pp.1-12, 2012. doi: 10.1109/HPCA.2012.6168948
- [15] Edison, T., “A GPU Memory System Comparison for an Elliptic Test problem,” [online], URL: <http://www.yuwang-cg.com/project1.html>, 2012.
- [16] Cebenoyan, C., “Graphics Pipeline Performance,” [online]. URL: http://http.developer.nvidia.com/GPUGems/gpugems_ch28.html. 2007.
- [17] Boyer, M., Meng, J., and Kumaran, K., “Improving GPU Performance Prediction with Data Transfer Modeling,” *IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum*. 2013. DOI 10.1109/IPDPSW.2013.236.
- [18] NVIDIA Developer Zone, “CUDA C Programming Guide,” [online], URL: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>, 2013.
- [19] Nicholas, W., “CUDA Handbook: A comprehensive Guide to GPU programming,” *Published by Pearson Press*, pp: 528. 2013.
- [20] Microsoft Developer Network, “NVIDIA GPU Architecture and CUDA Programming Environment,” [online]. URL: <http://code.msdn.microsoft.com/windowsdesktop/NVIDIA-GPU-Architecture-45c11e6d>. 2013.
- [21] Chapman, B., Jost, G., and Pas, R.D., “Using OpenMP: Portable shared memory Parallel Programming,” *The MIT Press (Scientific and Engineering Computation)*. Cambridge, Massachusetts, 2007.
- [22] Denning, P.J., “The Locality Principle,” *Communication Networks and Computer Systems*, Naval Postgraduate School, Monterey, California. [Online Journal], URL: http://cs.gmu.edu/cne/pjd/PUBS/locality_2006.pdf. 2008.
- [23] McKinley, K. S., Carr, S., and Tseng, C., “Improving data locality with loop transformations,” *ACM Transactions on Programming Languages and Systems*, vol. 18, No. 4, pp. 424, 1996.
- [24] Hoffmann, H., Agarwal, A., and Devadas, S., “Partitioning strategies for concurrent programming,” *Massachusetts Institute of Technology (MIT), CSAL Lab*, 2014.

- [25] Andrade, D., Fraguera, B.B., Brodman, J., and Padua, D., "Task-Parallel versus Data-Parallel Library-Based Programming in Multicore Systems," *17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp.101-110. 2009. doi: 10.1109/PDP.2009.54.
- [26] Harris, M., [online]. Available: <http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/>. 2013.
- [27] NVIDIA. [online]. Available: http://www.nvidia.com/docs/IO/43395/BD-05880-001_v02.pdf. 2011.
- [28] NVIDIA. [online]. Available: <http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf>. 2012.
- [29] Wu, W., Chen, J., "Efficient characterizations of composite materials electrical properties based on GPU accelerated finite difference method," *Antennas and Propagation Society International Symposium (APSURSI)*, pp.1-4. 2010. doi: 10.1109/APS.2010.5561731.
- [30] Asaduzzaman, A., Yip, C.M., Kumar, S.S.A., Asmatulu, R., "An effective CUDA based simulation for lightning strike protection on nanocomposite materials," *Proceedings of IEEE Southeastcon, 2013*, pp.1-5, 2013. doi: 10.1109/SECON.2013.6567368.