CUDA PERFORMANCE ANALYZER

A Thesis
Presented to
The Academic Faculty

by

Aniruddha Dasgupta

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2011

CUDA PERFORMANCE ANALYZER

Approved by:

Dr. Hyesoon Kim, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Date Approved: March 30, 2011

To my wife *Pallavi*

ACKNOWLEDGEMENTS

I wish to thank my family back in India and my very supportive wife. I would like to extend my sincere gratitude to my advisor, Dr. Hyesoon Kim for being a constant source of inspiration and guidance. I would also like to thank Dr. Richard Vuduc and Dr. Sudhakar Yalamanchili for agreeing to be a part of my thesis reading committee. Lastly, I would like to thank my colleagues Sunpyo Hong and Jaewoong Sim for their help and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

GPGPU Computing using CUDA is rapidly gaining ground today. GPGPU has been brought to the masses through the ease of use of CUDA and ubiquity of graphics cards supporting the same. Although CUDA has a low learning curve for programmers familiar with standard programming languages like C, extracting optimum performance from it, through optimizations and hand tuning is not a trivial task. This is because, in case of GPGPU, an optimization strategy rarely affects the functioning in an isolated manner. Many optimizations affect different aspects for better or worse, establishing a tradeoff situation between them, which needs to be carefully handled to achieve good performance. Thus optimizing an application for CUDA is tough and the performance gain might not be commensurate to the coding effort put in.

I propose to simplify the process of optimizing CUDA programs using a CUDA Performance Analyzer. The analyzer is based on analytical modeling of CUDA compatible GPUs. The model characterizes the different aspects of GPU compute unified architecture and can make prediction about expected performance of a CUDA program. It would also give an insight into the performance bottlenecks of the CUDA implementation. This would hint towards, what optimizations need to be applied to improve performance. Based on the model, one would also be able to make a prediction about the performance of the application if the optimizations are applied to the CUDA implementation. This enables a CUDA programmer to test out different optimization strategies without putting in a lot of coding effort.

CHAPTER 1

INTRODUCTION

General-Purpose Computation on Graphics Processing Units (GPGPU) [1] is on the fast track in the High Performance Computing (HPC) world today. The proliferation of cost-effective and state-of-the-art graphics cards coupled with new technologies like NVIDIA's Compute Unified Device Architecture (CUDA) [2], Khronos Group's Open Computing Language (OpenCL) [3], Microsoft's DirectCompute etc. have led to rapid strides in GPGPU technology and widespread acceptability of GPGPU in Computing. CUDA has been a forerunner in modern GPGPU technology. The entire range of contemporary NVIDIA Graphics cards is CUDA capable which gives one access to GPGPU in daily computing. A lot of general software is GPGPU optimized like web-browsers, video decoders, physics calculations in games etc. Also the specialized Tesla range of NVIDIA graphics cards are tuned for the HPC market with additional features like high performance double precision support, error correction codes etc. Thus CUDA is used in a very broad spectrum of applications today.

1.1 The Problem

The speedup offered by using NVIDIA CUDA is very significant for problems that offer a high degree of data level parallelism. Also many algorithms, which were traditionally considered to be a forte of fast serial execution, have been parallelized successfully using new parallel approaches. But the coding effort required for such cases might be significant considering the speedup achieved. As shown in Figure 1, in a survey performed by Vuduc et al [4], it is observed that for parallel sorting algorithms CUDA provides a significant benefit but at a price of increased coding effort.

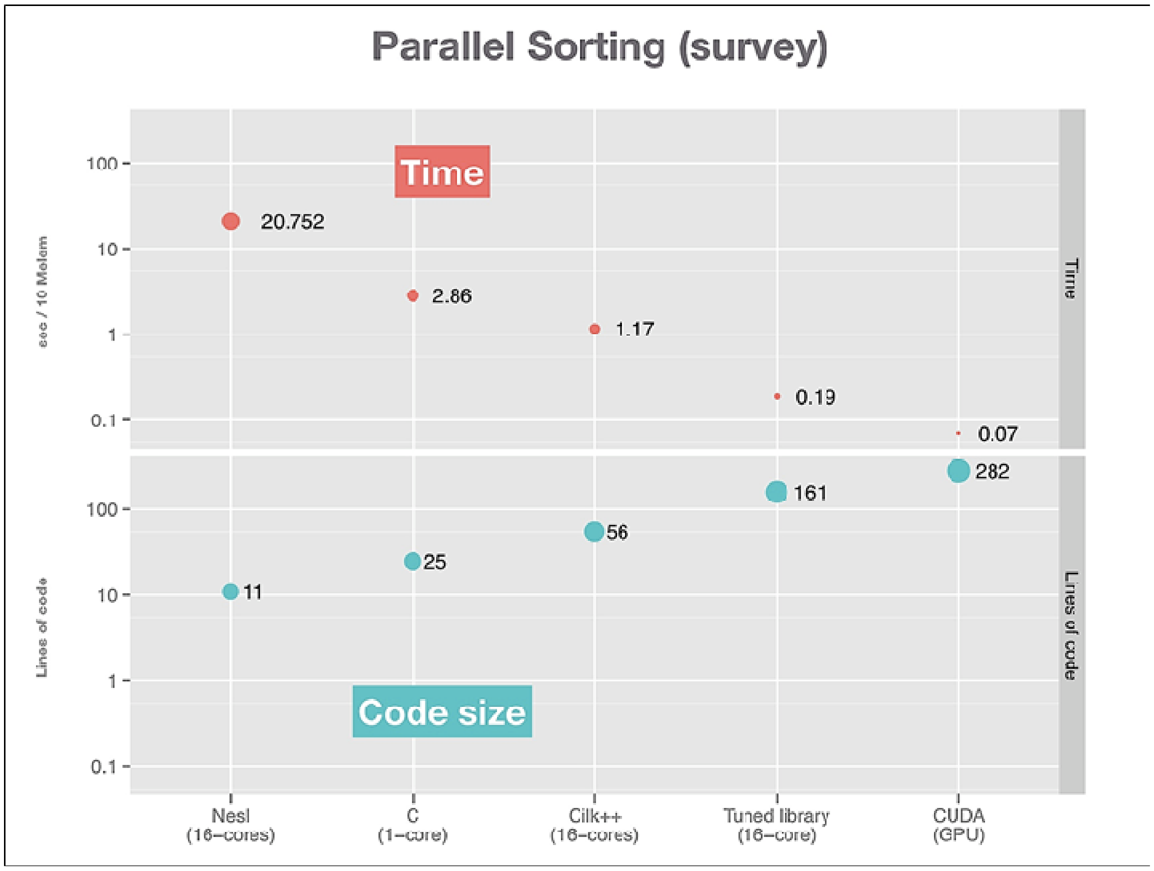

Figure 1: Performance vs. Effort [4]

Optimizing an application for maximum performance using CUDA is a non-trivial task. There are multiple factors that have an implication on the performance and multiple optimization strategies that affect these factors. Additionally these optimization strategies rarely affect the performance in an isolated manner. For example, a loop unrolling optimization implemented in an algorithm might help extract more ILP within a thread but the possible resultant increase in register usage might reduce the number of active threads solving the problem, thus possibly, hurting the performance. Because of such interactions between the optimization strategies, tuning for maximum performance is a difficult problem. Such problems have been discussed in length in works like [5].

Another hindrance to simplifying optimization process in CUDA is the lack of standard performance metrics to aid programmers in tuning. One of the few metrics available to programmer is Occupancy [6], which is defined as the ratio of the number of active warps resident per SM to the maximum possible number of active warps on an SM. However Occupancy is not a direct indicator of performance as illustrated in Figure 2 below.

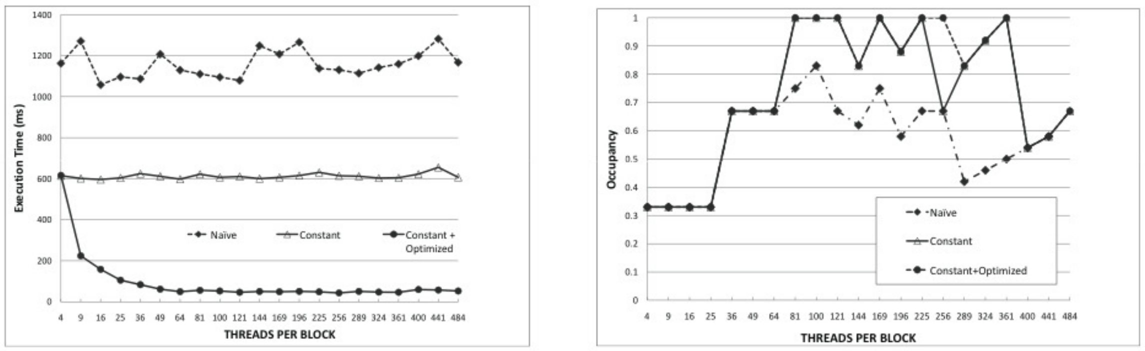

Figure 2: Occupancy and Performance [7]

Figure 2 depicts three versions of SVM algorithm; Naïve, Constant and Constant plus Optimized which have increasing levels of optimization giving higher performance. Doing a variation in the number of threads per block, it can be seen that though the occupancy varies a lot for the optimized cases, the performance remains fairly constant.

1.2 The Solution

A tool that can analyze the performance of CUDA kernels would be useful to the developers. Also there is a need for performance metrics that provide the programmers with an intuitive understanding of the bottlenecks of current implementation and hints towards improving the kernel performance. This would make it easier to create optimized CUDA kernels and bring down the effort of development for the same. A good solution to achieve all this is to create an Analytical model of the GPU which can predict performance of a CUDA kernel based on certain performance metrics. The GPU Analytical model proposed by Hong and Kim [7] is a step in this direction. In this study we enhance Hong's model further to make it more robust, reliable and flexible.

1.3 Contributions

This study aims to build upon Hong's GPU Analytical model to create a better GPU Analytical Model and tool, that we call the CUDA Performance Analyzer. CUDA Performance Analyzer has increased robustness since it brings in support for NVIDIA CUDA GT200 architecture and .introduces some new metrics pertaining to phenomenon like Memory Transaction Merging. It is more flexible since the GPU Analytical Model can now also work using the CUDA Visual Profiler in addition to GPUOcelot. We also put the CUDA Performance Analyzer through its paces with the help of a suite of 44 different kernel optimizations for a Fast Multipole Method (FMM) algorithm which helps improve the reliability of the tool.

1.4 Thesis Organization

We will briefly discuss the outline of the thesis. Chapter 2 deals with background information and gives an overview of CUDA, Hong's GPU Analytical Model and few existing tools. Chapter 3 goes into details of CUDA Performance Analyzer. Chapter 4 discusses the Fast Multipole Method Suite of CUDA based kernels and how it relates to the CUDA Performance Analyzer. Chapter 5 specifies the results and provides analysis of the same. We conclude with Chapter 6 which lays down prospects for future potential of the CUDA Performance Analyzer.

CHAPTER 2

BACKGROUND

The CUDA Performance Analyzer proposed in this study has its roots in the CUDA Analytical Model by Hong et al. In this chapter we describe the base GPU analytical model. We cover some important basics of the CUDA programming model and GT200 GPU architecture [8]. We also discuss some existing tools that help in CUDA performance analysis.

2.1 CUDA Programming Model

The CUDA programming model is based on three key abstractions; a hierarchy of thread groups, shared memories and barrier synchronization.

The thread hierarchy in CUDA consists of threads, blocks and grids. A number of threads come together to form a block of threads. Multiple blocks come together to form a grid. A grid can be defined as a group of thread blocks that executes a kernel function. The threads within a thread block get executed concurrently on a Streaming Multiprocessor (SM). On the other hand, blocks within a grid get allocated to different SMs within a GPU for another level of parallel execution. Figure 3 shows the thread hierarchy in CUDA programming model.

There is a well-defined memory hierarchy in CUDA. Each thread has a private on-chip register space and off-chip local memory. Threads within a thread block can share data using an on-chip shared memory. Thread blocks within a grid can share data through a global memory space called device memory which lies off-chip. This off-chip device memory can also be used as a read-only texture memory or constant memory, both of which have an on-chip cache to help reduce the latency of access. Figure 4 gives a clear picture of the memory hierarchy in the CUDA programming model.

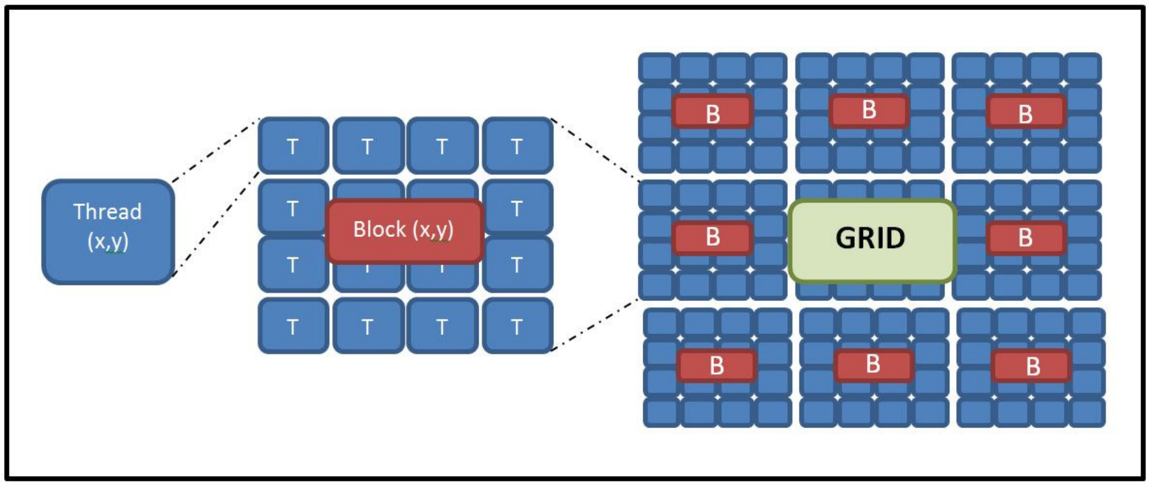

Figure 3: CUDA Thread Hierarchy

Barrier synchronization constructs exist for synchronizing threads within a thread block using the shared memory. There are no CUDA constructs for global level synchronization and this is mostly achieved by spawning a new kernel.

2.2 Base GPU Analytical Model

The CUDA Performance Analyzer has its roots in Hong's GPU Analytical Model [7]. In this section we briefly describe this base model.

2.2.1 Memory Warp Parallelism (MWP)

MWP represents the maximum number of warps per SM that can access the memory simultaneously during the time period from right after the SM processor executes a memory instruction from one warp until all the memory requests from the same warp are serviced.

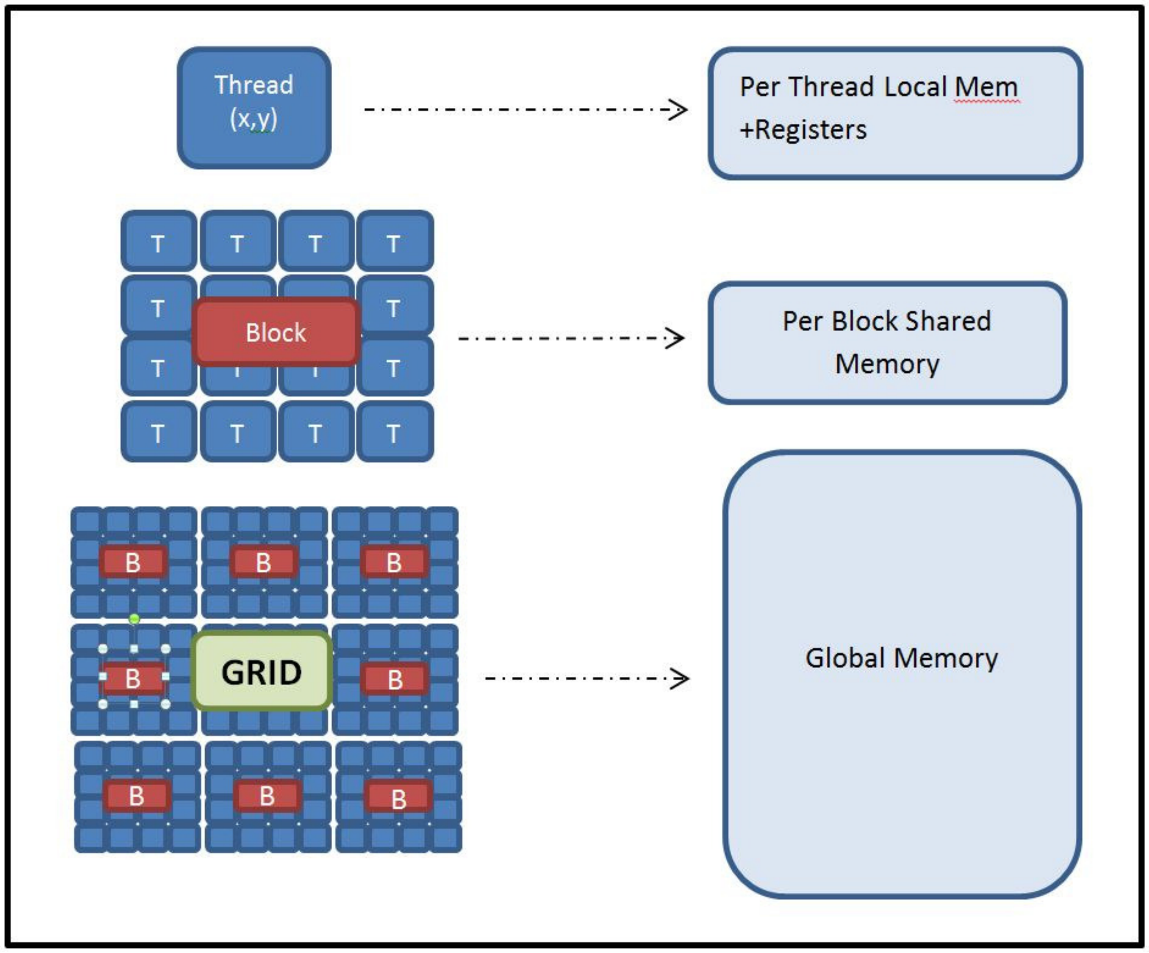

Figure 4: CUDA Memory Hierarchy

MWP is an indicator of Memory level parallelism that can be exploited. MWP is dependent upon the memory bandwidth, certain parameters of memory operations like latency, and the number of active warps in an SM.

2.2.2 Computation Warp Parallelism (CWP)

CWP is defined as the number of warps that the SM can execute while waiting for a memory request to get serviced; plus one. CWP is a measure of computation per memory access. It also helps to know whether a particular kernel is compute bound or memory bound. If CWP is greater than MWP then the kernel is mostly memory bandwidth limited. On the other hand, MWP being greater than CWP implies that the kernel is compute bound. Optimization strategies for both of these cases would vary greatly.

Figure 5 shows the concepts of MWP and CWP. The numbered elements are active warps on a SM; the blue part being a computation operation and the green part being a memory request.

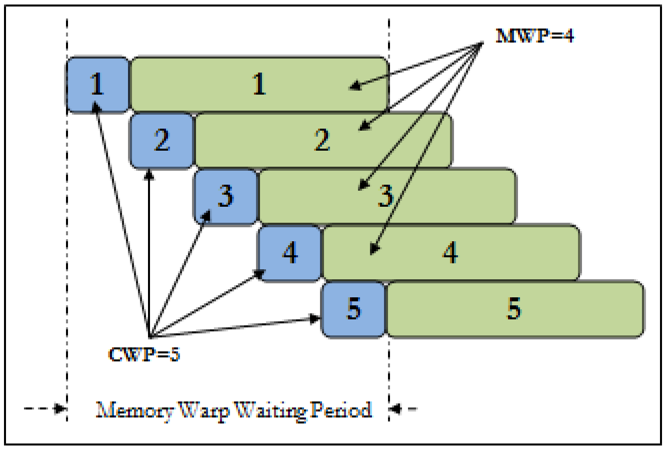

Figure 5: MWP and CWP

2.2.3 Predicting CUDA Kernel Performance

Apart from MWP and CWP, the analytical model defines many other terminologies. *Mem_LD* is the minimum number of latency cycles for a memory transaction. Uncoalesced memory requests spawn *#Uncoal_per_mw* number of memory transactions. *Departure_delay_uncoal* is defined as the minimum time interval between two consecutive memory transactions of an uncoalesced memory request. Similarly *Departure_delay_coal* is the time interval between the two transactions of a coalesced memory request. The term *departure_delay* gives the overall weighed time interval between two memory warps. *Mem_L* is the overall round trip latency to the DRAM for a memory warp request and is a function of the number of coalesced and uncoalesced memory requests (given by *#Coal_Mem_insts* and *#Uncoal_Mem_insts* respectively) performed by a warp. Additionally *Mem_cycles* is the number of memory cycles per warp while *Comp_cycles* is the total number of Compute cycles. Based on these parameters and some others, a prediction can be made about the performance of the CUDA kernel. A summary of the how to calculate various parameters and derive the prediction is given below:

$$Mem\_L\_Uncoal = Mem\_LD + (\#Uncoal\_per\_mw-1) * Departure\_del\_uncoal \quad (1)$$

$$Mem\_L\_Coal = Mem\_LD \quad (2)$$

$$Mem\_L = Mem\_L\_Uncoal * Weight\_uncoal + Mem\_L\_Coal * Weight\_coal \quad (3)$$

$$Weight\_uncoal = \#Uncoal\_Mem\_insts / (\#Uncoal\_Mem\_insts + \#Coal\_Mem\_insts) \quad (4)$$

$$Weight\_coal = \#Coal\_Mem\_insts / (\#Uncoal\_Mem\_insts + \#Coal\_Mem\_insts) \quad (5)$$

$$Departure\_delay = (Departure\_del\_uncoal * \#Uncoal\_per\_mw) * Weight\_uncoal + Departure\_del\_coal * Weight\_coal \quad (6)$$

$$MWP\_Without\_BW\_full = Mem\_L \ / \ Departure\_delay \quad (7)$$

$$MWP\_Without\_BW = MIN(MWP\_Without\_BW\_full, \#Active\_warps\_per\_SM) \quad (8)$$

$$Mem\_cycles = Mem\_L\_Uncoal * \#Uncoal\_Mem\_insts + Mem\_L\_Coal * \#Coal\_Mem\_insts \quad (9)$$

$$Comp\_cycles = \#Issue\_cycles * \#total\_insts \quad (10)$$

$$N = \#Active\_warps\_per\_SM \quad (11)$$

$$\#Rep = \#Blocks / (\#Active\_blocks\_per\_SM * \#Active\_SMs) \quad (12)$$

If (MWP is N warps per SM) and (CWP is N warps per SM)

$$Exec\_cycles\_app = (Mem\_cycles + Comp\_cycles + (Comp\_cycles/\#Mem\_insts) * (MWP - 1)) * \#Rep \quad (13)$$

If (CWP >= MWP) or (Comp_cycles > Mem_cycles)

$$Exec\_cycles\_app = (Mem\_cycles * N / MWP + (Comp\_cycles/\#Mem\_insts) * (MWP - 1)) * \#Rep \quad (14)$$

If (MWP > CWP)

$$Exec\_cycles\_app = (Mem\_L + Comp\_cycles * N) * \#Rep \quad (15)$$

2.3 NVIDIA GT200 Architecture

In this section we discuss some important aspects of NVIDIA GT200 architecture [8] which have changed over the G80 architecture.

2.3.1 GT200 Memory Coalescing Model

The G80 architecture GPUs have a very strict coalescing model in which a non-coalesced request would always spawn 32 memory transactions. Also for a memory request to be coalesced, threads of a warp would have to access consecutive memory locations; and would then result into two memory transactions. In GT200, the coalescing model has been revamped; the restrictions on coalescing have been loosened but the protocol has become more complicated. The concepts of coalesced/non-coalesced accesses have been diluted to give more importance to variable memory transactions of 32 bytes, 64 bytes and 128 bytes. A memory request spawns a variable number of different sizes of memory transactions depending on the size of data being fetched per thread, the alignment of the memory being fetched and ability to reduce a transaction to minimum size possible. This results in a much lesser number of memory transactions on average over the G80 architecture and thus improves utilization of bandwidth. The exact protocol can be referred to in CUDA programming guide [6].

2.3.2 Shared Memory Model

The degree of banking in shared memory has doubled over the previous architecture and now stands at 32 banks with 4 bytes being interleaved in each of the banks. This results in reduced number of shared memory conflicts for certain cases. There are other enhancements like support for broadcast of data to multiple threads within a warp if they are accessing the same location in the same bank. This would have resulted in access serialization in G80 architecture.

2.3.3 Other enhancements

In GT200 architecture each SM has some Special Function Units (SFUs) which operate on transcendental functions like sine, log etc and can also perform floating point operations. So there can be simultaneous execution of instructions between the stream processors (SPs) and the SFUs resulting in higher instruction throughput. There is also a double precision floating point unit. The memory controllers are smarter and can perform memory transaction merging which is discussed later on. Apart from these there are a lot of improvements. We have just briefly discussed the ones which are important from the point of view of analytical modeling.

2.4 CUDA Performance Analysis Tools

In this section we discuss a few important tools that help in analysis of CUDA performance. We made use of all these tools in this study.

2.4.1 CUDA Visual Profiler

NVIDIA CUDA GPUs have built in hardware counters and many predefined performance events which can use these. The CUDA Visual Profiler [9] tool uses these performance counters and events to collect a lot of useful statistics of a running CUDA kernel. CUDA Visual Profiler gives a lot of important information like the number of instructions, memory requests and transactions, occupancy, number of branches etc which can help in performance analysis. It has both a command line as well as a GUI interface. It can additionally analyze the previously mentioned data into performance metrics such as global memory throughput, instruction throughput, ipc etc. It is a NVIDIA tool and comes packed with the CUDA toolkit [10].

2.4.2 GPUOcelot

GPUOcelot [11] is a dynamic compilation framework for heterogeneous systems. It realizes numerous backend targets for CUDA programs. So a program written in CUDA can be executed without recompilation on x86 CPUs using PTX emulation or LLVM translation as well as run natively on AMD and NVIDIA GPUs. PTX is a virtual Instruction Set Architecture for NVIDIA CUDA Architecture. PTX provides a stable programming model and instruction set for general purpose parallel programming [12].

Using Ocelot one can also instrument the CUDA kernel PTX code while performing emulation by writing instruction analyzers using the API provided with Ocelot. These instruction analyzers can be used to collect a lot of statistics and metrics relevant to the kernel performance. In this study we have used GPUOcelot to create an instruction analyzer which collects performance relevant data from the CUDA kernel, to be used by the Analytical model for further analysis. The scope of GPUOcelot is much bigger than what we have used in this study and more details about it can be referenced in [11].

2.4.3 Decuda

The PTX assembly code is not a true representation of the binary code that lies in the NVIDIA CUDA Binary (.cubin) file. There is some optimization that takes place between the PTX and the .cubin file. DECUDA is a disassembler for the NVIDIA CUDA binary format. It provides insight into the internal instructions generated and can better explain the performance aspects from these instructions. DECUDA is a third party tool and can be obtained from [13].

2.4.4 Cuobjdump

Cuobjdump is a tool for manipulating CUDA object files. Supported inputs are pre-CUDA 3.0 text-based cubins or a CUDA 3.0 ELF-based cubins. Cuobjdump can display the assembly instructions for a particular kernel, making it useful for optimization and debugging. Cuobjdump is an NVIDIA provided tool available to CUDA registered developers [14]. The difference between DECUDA and Cuobjdump is that the latter gives the actual instructions from the GT200 Instruction set instead of the PTX abstraction.

CHAPTER 3

CUDA PERFORMANCE ANALYZER

The CUDA Performance Analyzer can be used to predict the performance of a CUDA kernel. It also gives insight into the bottlenecks of the particular kernel implementation. It helps the programmer to understand the parameters to which the particular kernel is sensitive in terms of performance and provides hints towards how to improve the performance. The CUDA performance analyzer consists of two main parts: Frontend Data Collector and GPU Analytical Model Based Predictor. Figure 6 shows the block diagram of the CUDA Performance Analyzer.

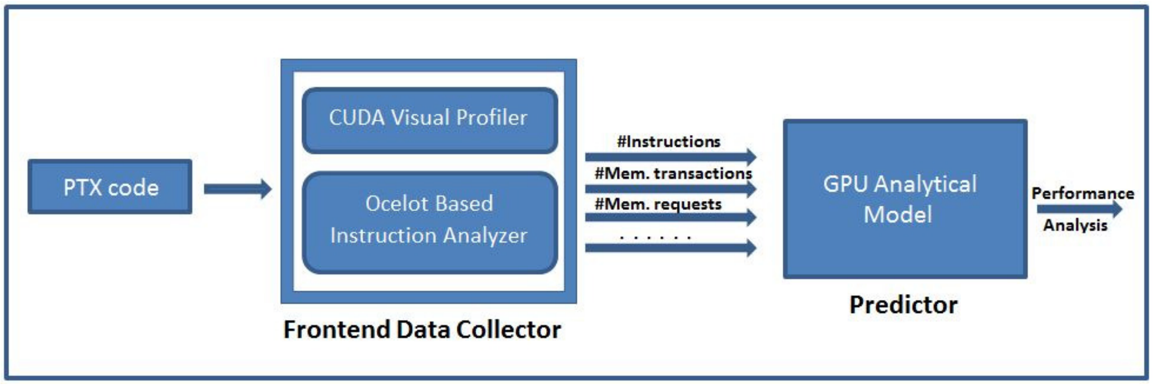

Figure 6: CUDA Performance Analyzer Block Diagram

3.1 Frontend Data Collector

The Frontend Data Collector (FDC) works to gather a number of statistics pertinent to the particular kernel under analysis. How this information is used would be discussed later in the chapter. The most important ones amongst these are:

a. Number of threads/block

b. Number of blocks/grid

c. Register Usage for the kernel

d. Shared Memory used per Block

e. Occupancy

f. Total number of Instructions

g. Total number of Global Memory Requests

h. Number of 32 Byte memory transactions

i. Number of 64 Byte memory transactions

j. Number of 128 Byte memory transactions

k. Total number of shared memory requests

l. Number of shared memory requests that have bank conflicts

m. Avg. Latency for conflicted shred memory requests

n. Number of synchronization instructions

o. Number of high latency compute instructions

For the CUDA Performance Analyzer, from the above list, the items from a. to j. can be called as compulsory parameters for making a performance prediction for the kernel. The factors k. to o. are not compulsory but help to increase the accuracy of the prediction. There are two tools that can fit into the role of the Frontend Data Collector. These are:

3.1.1 CUDA Visual Profiler

As discussed earlier, the CUDA Visual Profiler can reliably provide a lot of useful information about the CUDA kernel. From the parameter list given above, parameters a. to j. can be obtained using the profiler which are sufficient for the analytical model to make a performance prediction. Also since the profiler has a command line interface it can be easily used to profiler multiple kernels through simple scripting methods. It is convenient since it gets installed as part of the CUDA toolkit. The downside is that it gives no information on parameters k. to o., which could have been used to predict performance more accurately. Another point to note is that the data we need for the Analytical Model is in terms of per warp per SM, while the one provided by CUDA Visual Profiler is the total count over all warps per SM or per Texture Processing Cluster (TPC); depending on the parameter under consideration. Table 1 gives an overview of the parameters provided by the CUDA Visual Profiler and the conversion to be applied to the same for use with the Analytical Model.

Table 1: CUDA Visual Profiler - Relevant Performance Counters

Performance Counter	**Description**	**Conversion to be used in Analytical Model**
instructions	Number of dynamic instructions executed on an SM	Instructions/(total warps per SM)
gld_request	Number of global memory load requests per SM	gld_request/(total warps per SM)
gst_request	Number of global memory store requests per SM	gst_request/(total warps per SM)
gld_32/64/128 byte	Number of 32/64/128 byte global memory load transactions per TPC	(gld_32/64/128 byte) /(total warps per TPC)
gst_32/64/128 byte	Number of 32/64/128 byte global memory store transactions per TPC	(gst_32/64/128 byte) /(total warps per TPC)

Apart from these performance counters the CUDA Visual profiler also provides useful information like number of registers per thread, static shared memory, dynamic shared memory, occupancy etc.

3.1.2 GPUOcelot

GPUOcelot provides API to create instruction analyzers which can be used to instrument an emulating PTX kernel. The Instruction Analyzer used by Hong was based on the NVIDIA G80 architecture of GPUs. As discussed in section 2.3, GT200 architecture brings in many changes over the G80 architecture. Based on these architectural changes as well as some deficiencies in the previous instruction analyzer, the instruction analyzer was enhanced as a part of this study. The major changes made to the instruction analyzer were as follows:

3.1.2.1 New Coalescing Model

The previous model used to measure the number of coalesced and uncoalesced memory requests in the kernel. Also the number of memory transactions was fixed; two for coalesced and thirty two for uncoalesced requests. The new model implements the coalescing model of the GT200 architecture which calculates the number of dynamic memory transactions, depending on size of data fetched, memory alignment and potential for transaction size reduction.

3.1.2.2 New Shared Memory model

The new instruction analyzer calculates the number of shared memory bank conflicts for the kernel. It also calculates the average shared memory access latency for the conflicted shared memory operations. Factors like data broadcast are also considered while calculating this.

3.1.2.3 Improved reliability

The instruction analyzer considers control divergence while taking count of number of instructions, and different operations. It uses a warp level bit-mask to figure out control divergence and takes into account instruction serialization due to the same.
Thus using GPUOcelot we are able to gather all the information that is required to be fetched by the Frontend Data Collector. Even those parameters that cannot be tracked by

the CUDA Visual profiler like the shared memory related statistics, high latency compute instructions etc. can be obtained using this.

However there is a slight disadvantage in using this approach. GPUOcelot makes use of the PTX kernel for emulation while running the instruction analyzer. In CUDA computing, PTX gets optimized before it actually runs on the GPU, so we are essentially making a prediction about the kernel from an un-optimized version of the code. We will discuss this in greater detail in section 4.2.5.2. However practically this has not been a significant deterrent and the predictions are fairly accurate.

Next we take a look at the other piece of the puzzle, which is the GPU Analytical Model based predictor.

3.2 GPU Analytical Model Based Predictor

We have already discussed Hong's GPU analytical model in section 2.2. The original model was more suited towards the G80 architecture. In this study, the GPU analytical model has been enhanced to make use of memory transactions instead of coalesced and non-coalesced memory requests. The original model was able to use only Ocelot as the Frontend Data Collector since CUDA Visual Profiler does not give number of coalesced and non-coalesced memory instructions. But with a switch to memory transaction based model, both Ocelot as well as CUDA Visual Profiler can act as the front-end as explained earlier. The following is a list of enhancements to the GPU analytical model as an outcome of this study:

- Ability to use model with the output of CUDA Visual Profiler
- Memory transactions based memory modeling
- Merging of memory requests is considered and accounted for
- Independent memory requests are accounted for while determining performance
- Shared Memory Banked conflicts are considered
- High latency compute instructions are considered

As shown in the block diagram in figure 6, the GPU Analytical model based predictor takes an input from the Frontend Data Collector. This consists of various parameters listed in section 3.1. Now based on these parameters, the model calculates performance related metrics like MWP, CWP etc. and based on these, makes a prediction about the performance of the kernel. Let us now examine this process in greater detail.

3.2.1 Calculating MWP

Overall MWP is determined by three factors:

3.2.1.1 MWP with available bandwidth consideration

Since practically the SM consumes a part of the total bandwidth of the system, the total number of warps being able to access the memory simultaneously might depend on the bandwidth usage per warp per SM. This bandwidth usage is governed by average number of transaction and transaction sizes per warp per memory request. Since we have both the number of memory requests and the number of memory transactions of different sizes available to us, it is possible to determine MWP with peak bandwidth.

*Data_per_request = ((warp_memtran_32b*32) + (warp_memtran_64b*64) + (warp_memtran_128b*128))/warp_mem* *(16)*

Where;

Data_per_request is Avg. size of data fetched per memory request

warp_memtran_32b/64b/128b is the number of memory transactions of size 32 bytes/64 bytes/128 bytes respectively, per warp for the entire kernel

warp_mem is the total number of memory requests per warp for the kernel

*bandwidth_requirements_per_warp = (Data_per_request * Freq)/Mem_Latency* *(17)*

Where;

bandwidth_requirements_per_warp is the amount of bandwidth required per warp

Mem_latency is the avg. memory latency to service a memory request

*warps_per_sm_to_reach_peak_bw = Total_Mem_bandwidth / (bandwidth_requirements_per_warp *#Active SMs)* *(18)*

The term *warps_per_sm_to_reach_peak_bw* is the *MWP_peak_bandwidth*.

<u>3.2.1.2 MWP without bandwidth consideration</u>

For calculating *MWP_without_BW*, we consider the number of transactions per memory request as obtained through:

num_trans_per_request = (warp_memtran_32b + warp_memtran_64b + warp_memtran_128b) / (warp_mem) *(19)*

where;

num_trans_per_request is avg. number of memory transactions per memory request

32 Byte, 64 Byte and 128 Byte memory transactions have different departure delays but same base memory latency. So depending on the number of 32 Byte, 64 Byte and 128 Byte transactions per request and their respective departure delays, we can calculate the effective departure delay as follows:

weight_32b = warp_memtran_32b/ (warp_memtran_32b + warp_memtran_64b + warp_memtran_128b) *(20)*

weight_64b = warp_memtran_64b/ (warp_memtran_32b + warp_memtran_64b + warp_memtran_128b) *(21)*

weight_128b = warp_memtran_128b/ (warp_memtran_32b + warp_memtran_64b + warp_memtran_128b) *(22)*

*avg_dep_delay_per_tran = (dep_delay_32b * weight_32b) + (dep_delay_64b * weight_64b) + (dep_delay_128b * weight_128b)* *(23)*

where;

avg_dep_delay_per_tran is the avg. departure delay per memory transaction *dep_delay_32b/64b/128b* is departure delay for 32 bytes/64 bytes/128 bytes transactions respectively (obtained through tuning from micro-benchmark; refer to section 3.2.6)

The final memory latency is calculated as follows:

*mem_latency_final = base_mem_latency + (num_trans_per_request-1) * avg_dep_delay_per_tran* *(24)*

where;

base_mem_latency is obtained through tuning process as documented in section 3.2.6

Also the final value of departure delay is calculated as:

*dep_delay_per_request = $num_trans_per_request * avg_dep_delay_per_tran* *(25)*

Now with the effective memory latency (*mem_latency_final*) and effective departure delay per memory request (*dep_delay_per_request*) calculated, the MWP without bandwidth full can be calculated as:

MWP_without_BW_full = (mem_latency_final / dep_delay_per_request) *(26)*

3.2.1.3 Active warps per SM

The third factor influencing MWP is the number of active warps per SM. In the new GPU Analytical Model we consider two new parameters, which play along with number of Active warps per SM to give *Active_warps_for_MWP*.

*Active_warps_for_MWP = Active_warps_per_SM * (Independent Loads) / (Duplicate Loads)* *(27)*

Independent loads is the average number of independent memory loads per thread.

If a kernel has independent loads then a single warp can issue multiple loads thus effectively increasing the MWP over the active number of warps. This is illustrated in figure 7.

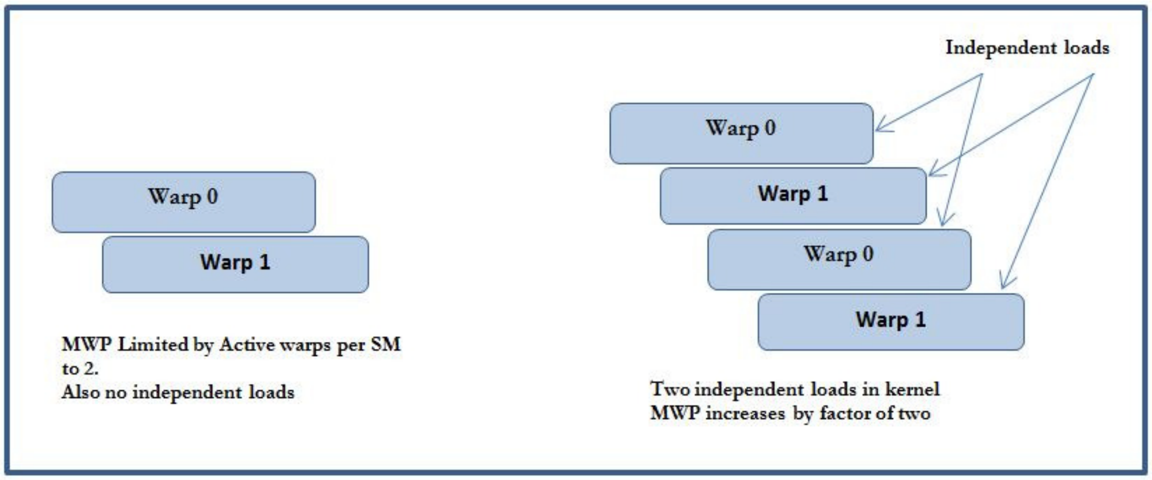
Independent loads
Warp 0
Warp 1
Warp 0
Warp 1
Warp 0
Warp 1
MWP Limited by Active warps per SM to 2.
Also no independent loads
Two independent loads in kernel
MWP increases by factor of two

Figure 7: MWP increase due to Independent Loads

Duplicate loads is the average number of load requests that fetch the same data and are being handled by the memory subsystem within the same timeframe. For e.g. in the following code snippet, the load would be accessing the same data for all warps within the thread block.

```
-----------------------------------------------------
int Index = blockIdx.x + (threadIdx.x % 32);
float data = FloatArray[Index];
-----------------------------------------------------
```

Thus if there are duplicate loads, it will lead to memory merging effect. *Memory Merging Effect* means that when there are memory loads being serviced by the memory subsystem, that are fetching the same data, then data for one load will piggyback automatically when data for previous duplicate load is fetched. Thus the latter load should not be accounted for as a part of MWP, and MWP from point of view of active warps reduces. This scenario is depicted in figure 8.

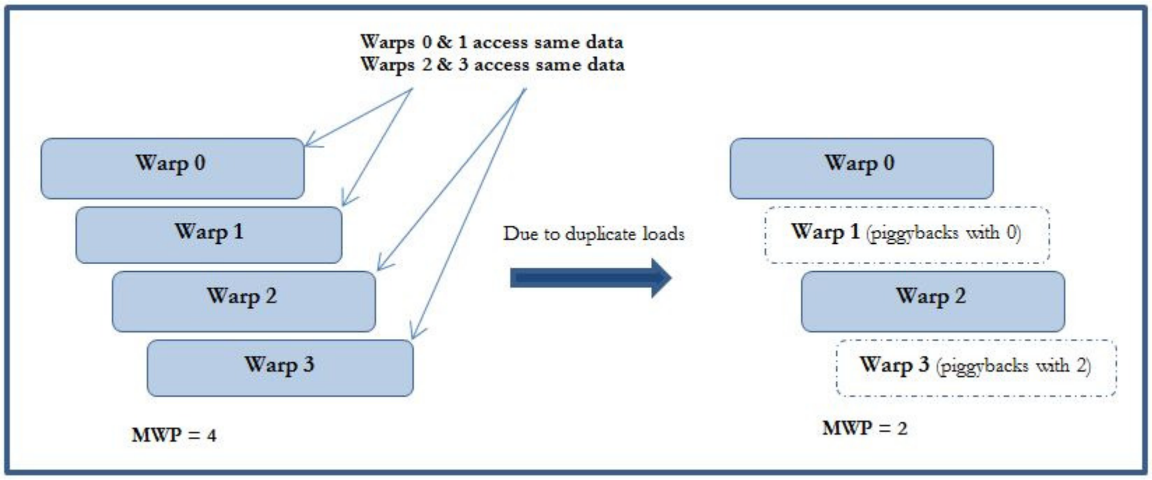
Warps 0 & 1 access same data
Warps 2 & 3 access same data
Warp 0
Warp 1
Warp 2
Warp 3
MWP = 4
Due to duplicate loads
Warp 0
Warp 1 (piggybacks with 0)
Warp 2
Warp 3 (piggybacks with 2)
MWP = 2

Figure 8: MWP decrease due to Duplicate Loads

Please note that the actual Active number of warps does not change due to these two parameters. They just contribute towards change in MWP.

Now from the above three factors, MWP can be obtained as:

MWP_final = min(MWP with bandwidth, MWP without Bandwidth, Active_warps_for_MWP) *(28)*

3.2.2 Calculating CWP

CWP calculation remains the same and is given by:

CWP_full = (Compute_cycles + Memory_cycles) / (Compute_cycles) *(29)*

CWP = min(CWP_full, Active warps per SM) *(30)*

3.2.3 Calculating Memory and Compute cycles

Once MWP and CWP have been calculated, the other important parameters we need for computing performance are compute cycles and memory cycles. All the dynamic instruction in the kernel thread are assumed to take four cycles; except for high latency instructions like sqrt, rsqrt, floating point division etc. and shared memory operations which have bank conflicts. Our Front-end data collector keeps a count of all these special latency instructions and factors them into the calculation of total compute cycles.

On the other hand, the memory cycles are calculated as the product of the final memory latency per memory request and the total number of memory requests in the kernel thread. Then depending on the relationship between MWP and CWP, the number of predicted execution cycles for the kernel is calculated.

3.2.4 Implications of Memory Transaction Merging

Consideration of memory merging effect is also a new addition to the analytical model. There has not been any extensive study in academia in documenting the memory merging effect; so we show it through an actual example in this study.

3.2.4.1 Observing merging effect in GT200 architecture – Tesla C1060 GPU:

Merging of memory requests at the memory controller might take place for in-flight requests that target the same memory locations. In such cases data requested by one warp might be fetched while the memory request for another warp is being catered, if the relevant data lies within the same memory transaction segment. In order to verify if such merging takes place, we designed a micro-benchmark, which would clearly demonstrate such a phenomenon. For the same number of memory requests (i.e. same amount of work done), if one case has memory addressed in a manner such that merging should take place; while the other case is designed to avoid merging; it is expected that the former case should show faster execution time due to lesser dynamic number of memory transactions. Our micro-benchmark is designed to create memory-addressing patterns that should translate to two cases; one without any merging and the other one with potential memory transaction merging. We would then compare the execution time of the benchmark to observe if any merging is taking place or not.

3.2.4.2 The Micro-benchmark :

The host code allocates a chunk of linearly addressed single dimensional float array on the device memory and preloads the elements in the array with a constant value. The kernel consists of a computation loop that performs some fused multiply-add operations and some global memory reads. The memory-addressing pattern for these global-memory reads can be manipulated using different indexing schemes. The parameters to the kernel are the number of iterations of the loop, the number of threads per block and a pointer to the float array which consists of constant value which can change depending on the scenario. The index is incremented by the value read from the global memory before the next global memory read. To ensure no merging in the first case, we need to take care that each warp of each block on all SMs access different memory locations throughout different iterations of the loop in the micro-benchmark. Table 2 shows the indexing scheme while Figure 9 shows the memory access pattern by the warps so as to avoid merging. It can be seen in Figure 9 that warp 0 and warp 1 of block 0 do not overlap in their memory access for the entirety of 100 iterations of their loop. The memory access regions of subsequent warps and blocks follow those shown in the figure to create a linear and consecutive memory access space for all the warps on all the blocks, thus avoiding any memory merging effect. On the other hand in Figure 10, it can be clearly seen that warp 0 and warp 1 are accessing similar memory locations in different iterations which can lead to memory transaction merging by the memory controller.

To take care that performance differences between merging and non-merging cases are accounted by only the memory request merging phenomenon, we found it better to have our micro-benchmarks designed such that the memory access is regular and would spawn only a single type of memory transaction. The micro-benchmark code is in Appendix A.

Table 2 shows the results.

Table 2: Memory Transaction Merging on 32 Byte Transactions

	No-merging	**With merging**
index	(blockIdx.x*(((int)blocksize/(int) WARPSIZE)*(CONFIG * ITERS * WARPSIZE))) + (((int) threadIdx.x / (int)WARPSIZE)*(CONFIG *ITERS * WARPSIZE)) + ((((int)(threadIdx.x % WARPSIZE)/(int)16)*8) + ((threadIdx.x % WARPSIZE)) % 8)	(blockIdx.x * blocksize) + (((int)threadIdx.x/(int)16)*8) + (threadIdx.x % 8);
Threads/ block	256	256
Blocks	120	120
Iterations	100	100
Memory requests/ warp	400	400
32B memory transactions/ warp	800	800
Execution Time(us)	703.2	630

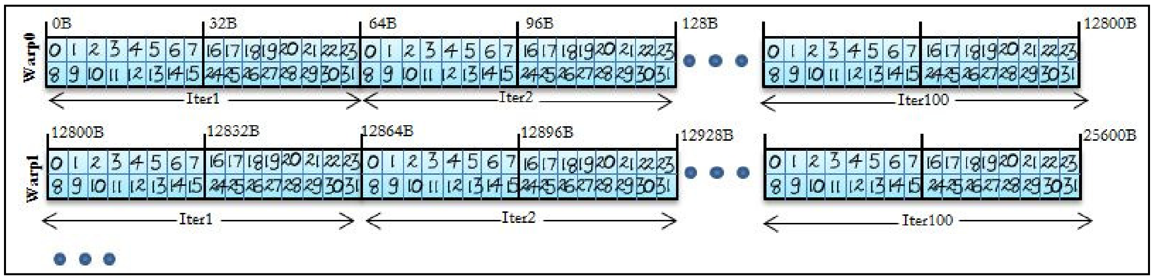

Figure 9: Memory Access Pattern for 32B transaction non-merging

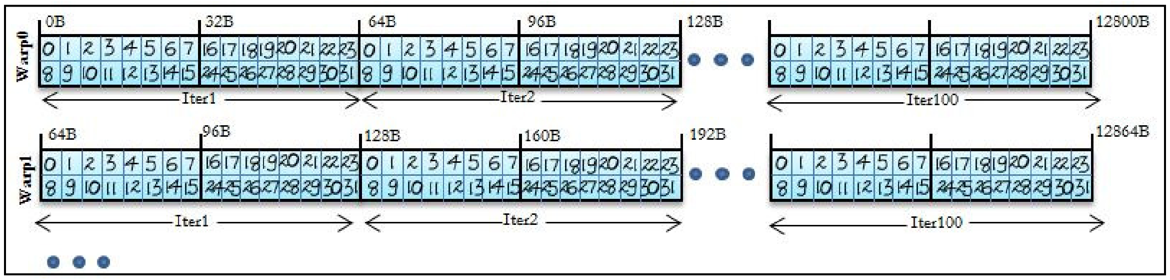

Figure 10: Memory Access Pattern for 32B transaction with merging

It can be seen from the table 2 that due to the sharing of memory requests, the execution time reduces by 10.38% for the same number and type of memory transactions.

Table 3 shows the results when the access pattern is such that 64 byte memory transactions are spawned with the first case showing no merging and the second case having merging of memory transactions. The memory access patterns for the non-merging and merging cases are shown in figures 11 and 12 respectively.

Table 3: Memory Transaction Merging on 64 Byte Transactions

	No-merging	**With merging**
index	(blockIdx.x*(((int)blocksize/(int)WARPSIZE)*(CONFIG* ITERS * WARPSIZE))) + (((int) threadIdx.x / (int) WARPSIZE) * (CONFIG *ITERS * WARPSIZE)) + (threadIdx.x % WARPSIZE)	blockIdx.x * blocksize + threadIdx.x
Threads/ block	256	256
Blocks	120	120
Iterations	100	100
Memory requests/ warp	400	400
64B memory transactions/ warp	800	800
Execution Time(us)	730	627

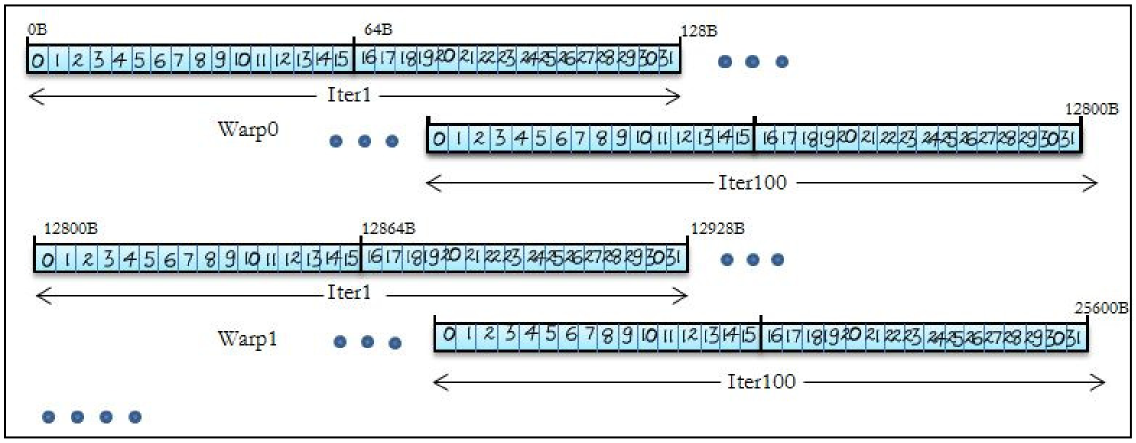

Figure 11: Memory Access Pattern for 64B transaction non-merging

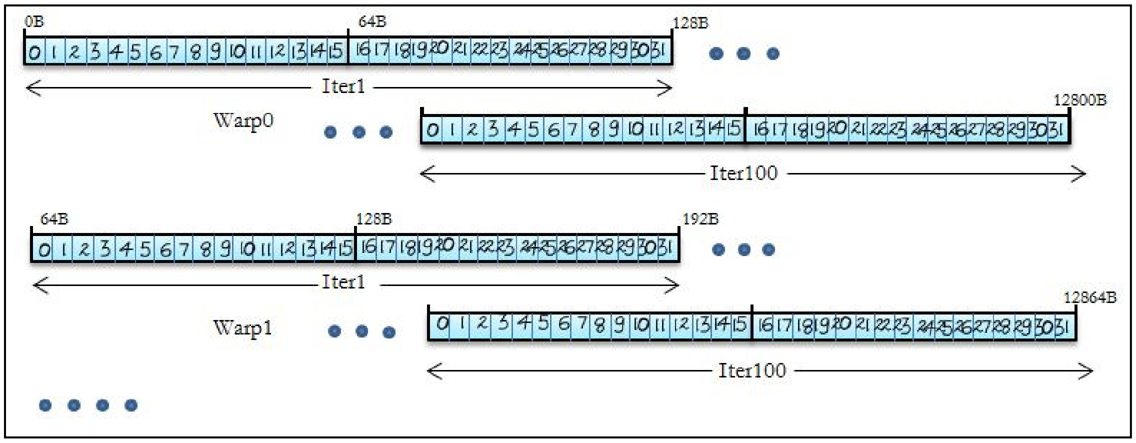

Figure 12: Memory Access Pattern for 64B transaction with merging

From Table 3, again due to sharing of memory transactions, the execution time reduces by 14.10%.

3.2.4.3 How is merging effect accounted for in Analytical Model?

There are two ways in which we account for memory merging effect in our model.
As seen previously, the *Active_warps_for_MWP* has a factor of *duplicate_Loads*. This *duplicate_loads* accounts for the memory merging effect since duplicate loads target the same memory locations are likely to be merged by the memory controller. This in turn affects the MWP.
We also reduce the number of memory transactions by a factor of '*duplicate_loads*' to account for merging of memory requests. This makes sense since duplicate memory transactions would just piggyback on transactions fetching the same data and consequently should not be considered.

3.2.5 Implications of Vector Memory Operations

GT200 architecture supports some vector memory operations. Examples from the PTX ISA about these operations are ld.global.v4.f32, ld.global.v2.s32 etc. It needs to be investigated if vector memory operations have different memory transaction latencies and other peculiarities as compared to the scalar memory operations. In order to put light on this issue we tweak our micro-benchmark code such that we get same number of memory requests and memory transactions for scalar as well as vector operations cases and then compare their performance. A significant digression in their performance would suggest towards different effective latencies for these two types of operations. Micro-benchmark codes for the scalar memory operations and vector memory operations case can be seen in Appendix A. It has been ensured that:

- Number of total instructions is nearly equal so that any divergence in performance can be attributed to memory characteristics.
- Number of memory requests is the same
- Number of memory transactions is the same so that any divergence in performance can be attributed to difference in memory latencies in scalar and vector case.

- There is no merging of memory transactions by having each warp of each block on all SMs access non-overlapping memory locations

Table 4 shows various parameters and result as obtained from the CUDA Visual Profiler for running the above benchmarks for 256 threads, 120 blocks and 100 iterations of the loop. From Table 4, it is clear that despite the performance sensitive parameters being the same, there is a 27.62% rise in execution time due to the vector memory operations. This points to conclude that the global memory latency for vector operations is greater than that for the scalar operations.

Table 4: Comparing Vector and Scalar Memory Operations

	Vector memory Ops	**Scalar memory Ops**
Threads/block	256	256
Blocks	120	120
Occupancy	1	1
Instructions/warp	7934	7942
Mem.requests/warp	400	400
Mem Transactions /warp (all 32 B transactions)	800	800
Execution Time(us)	1058.8	829.6

3.2.5.1 Effect of Memory Transaction merging on Vector Memory Operations

Since the vector load operations show a significant performance deviation from the scalar ones, we decided to test if they exhibit a memory merging effect like the scalar memory operations do. We change the indexing scheme in the micro-benchmarks to allow for overlapping in different warps within the same block. Different blocks though, still access non-overlapping memory regions. Thus this is a case of intra-block merging if it does exist. Table 5 documents the indexing scheme used in the micro-benchmark as well as the results of the test. Figures 13 and 14 show the memory access patterns.

Table 5: No Merging Effect in Vector Memory Operations

	Vector Mem. Ops – No merge	**Vector Mem. Ops – With merge**
Indexing	((blockIdx.x)*((int)blocksize/(int) WARPSIZE) * ITERS*CONFIG*2) + (((int)threadIdx.x/(int)WARPSIZE) *ITERS*CONFIG*2)	((blockIdx.x)*((int)bloc ksize/ (int)WARPSIZE) *ITERS*CONFIG*2)
Threads/ block	256	256
Blocks	120	120
Num_Iterations	100	100
Occupancy	1	1
Instructions/warp	7934	7929
Mem.requests/warp	400	400
Mem Transactions /warp (32 Bytes in this case)	800	800
Execution Time(us)	1058.8	1047.7

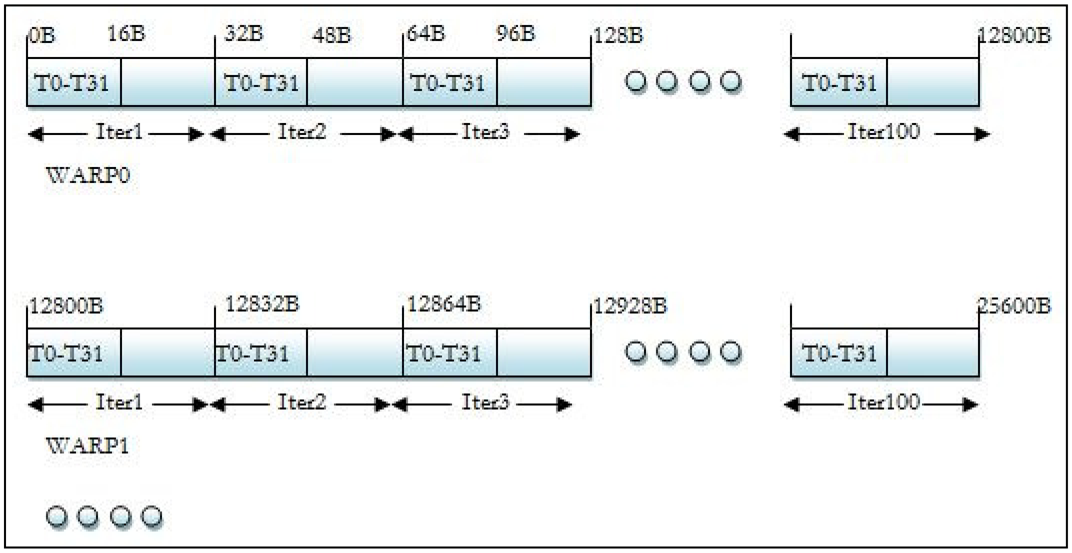

Figure 13: Memory Access Pattern for vector memory operations (v4.f32) with no merging

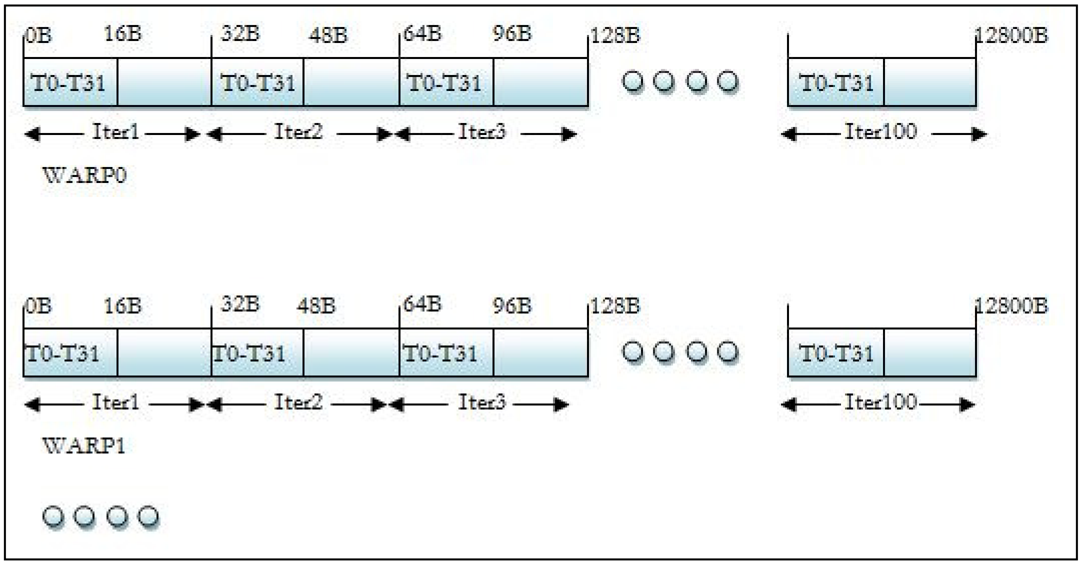

Figure 14: Memory Access Pattern for vector memory operations (v4.f32) with merging

Thus in case of vector memory operations, because of merging, the execution time reduces by 1.04%. In comparison, for scalar memory operations a similar comparison yields a benefit of 10% – 14% as seen in Table 2 and Table 3. Thus merging effect is quite insignificant in case of vector memory operations and has been not been considered as an influencing factor in this study.

3.2.6 Tuning of the GPU Analytical Model

There are five parameters to be tuned in the model for achieving accurate results. These five parameters are:

1. Frequency
2. Base Memory Latency
3. Departure delay for 32 Byte memory transactions with scalar operations
4. Departure delay for 64 Byte memory transactions with scalar operations
5. Departure delay for 128 Byte memory transactions with scalar operations
6. Departure delay for transactions of vector memory operations

For finding out each of these parameters we make variations to the micro-benchmark such that for each case, only one of the parameters is responsible for determining the performance of the benchmark. Then we can compare the actual execution time of the benchmark with the model predicted time and tweak the value of the parameter so that the actual and predicted execution times converge. The reference GPU used for the tuning is NVIDIA Tesla C1060. Though our model is consistent across all GPUs across this architecture, for each GPU this tuning would be required to be done before applying the model for prediction. Now we will discuss in detail how tuning was done for each individual parameter. One thing to keep in mind is that tuning should be performed in the order specified below. This is because subsequent parameters take previously tuned parameters as input.

3.2.6.1 Frequency

For tuning the frequency, we reduce the number of memory operations in the benchmark to zero. Reducing the number of memory operations to zero makes the MWP and CWP independent of the memory related parameters since MWP becomes equal to the number of active warps and CWP becomes undefined i.e. takes a value of zero. Consequently frequency becomes the only parameter, which defines the convergence of actual and predicted value. Tuning the frequency gave us a value of 1312 MHz, which is pretty close to the value 1296 MHz which is the defined shader frequency for Tesla C1060. Table 6 shows the result when the execution parameters are 512 threads per block, 120 blocks and 100 iterations of the kernel loop.

Table 6: Tuning the shader frequency

Tuned Value of Freq Parameter(MHz)	Actual Execution Time(ms)	Model Predicted Time(ms)	% Difference
1312	0.846	0.844	0.23

3.2.6.2 Base Memory Latency

Regarding the memory related parameters; it is very difficult to create a scenario, which isolates the effect of base memory latency from the memory transaction processing time while predicting performance. It was observed that the performance predicted by the model was not very sensitive to changes in the base memory latency as compared to the other memory related model parameters. So it was deemed proper to assume a realistic value for the base memory latency and have the other memory related parameters tuned around this assumed value.

NVIDIA specifies the global memory latency for GT200 to be between 400 to 600 cycles. This memory latency value accounts for the base memory latency and departure delay. So we made a reasonable assumption of 450 cycles for the base memory latency.

Table 7: Tuning the base memory latency

Value of Base Memory Latency	450 cycles

3.2.6.3 Departure delay for 32 Byte memory transactions with scalar operations

A memory request can manifest into multiple memory transactions. The GT200 architecture specifies memory transaction sizes of 32 Bytes, 64 Bytes and 128 Bytes. The final memory latency for a multi-transaction request depends not only on the base memory latency but also the departure delay, which varies with the size of the memory transaction. In order to tune our model with respect to processing time for 32 Byte transactions we use a micro-benchmark in which the memory loads only spawn 32 Byte transactions. Appropriate indexing by the threads into the global memory space can ensure this. Also we ensure that no memory transaction merging takes place in the micro-benchmark so that the merging phenomenon does not affect performance, thus aiding proper tuning. The micro-benchmark is given in Appendix A. Table 8 shows the tuned value.

Table 8: Tuning the Departure Delay for 32B Scalar Memory Transactions

Tuned value of departure delay for 32 byte transaction	Actual Execution Time(ms)	Model Predicted Time(ms)	% Difference
37 cycles	0.7243	0.7222	0.28

<u>3.2.6.4 Departure delay for 64 Byte memory transactions with scalar operations</u>

Similar to the previous case, we altered our micro-benchmark for it to spawn 64 byte transactions only for each memory request. Also it was ensured that there would be no memory transaction merging. The micro-benchmark is given in Appendix A.

Table 9: Tuning the Departure Delay for 64B Scalar Memory Transactions

Value of Processing time for 64 byte transaction	Actual Execution Time(ms)	Model Predicted Time(ms)	% Difference
37 cycles	0.7240	0.7222	0.2486

<u>3.2.6.5 Departure delay for 128 Byte memory transactions with scalar operations</u>

Similar to the previous case, we altered our micro-benchmark for it to spawn 128 byte transactions only for each memory request. Also it was ensured that there would be no memory transaction merging. The micro-benchmark is given in Appendix A.

Table 10: Tuning the Departure Delay for 128B Scalar Memory Transactions

Value of Processing time for 128 byte transaction	Actual Execution Time(ms)	Model Predicted Time(ms)	% Difference
58 cycles	1.137	1.131	0.52

3.2.6.6 Departure delay for transactions of vector memory operations

Using the GT200 analytical model we tried to find out the latency of vector memory operations. Since overall latency is a consequence of base memory latency as well as transaction processing time, we considered both these factors. We found running the model that the performance was not very sensitive to base memory latency but it was quite responsive to changes in transaction processing time. Tuning it for equivalence in actual and predicted performance we got the result as shown in Table 11.

Table 11: Tuning the Departure Delay for Vector Memory Transactions

	ld.global.f32 (scalar)	ld.global.v4.f32 (vector)
Departure delay for 32 Byte transactions (cycles)	37	57

CHAPTER 4

FAST MULTIPOLE METHOD

To put our CUDA Performance Analyzer through its paces, we use a suite of CUDA kernels based on the Fast Multipole Method (FMM) algorithm [15]. The suite has around forty different variations of FMM CUDA kernel, each with different kinds of optimizations for improving performance. If our analytical model can track the performance variation and make correct prediction for different optimizations, we are ensured of a robust prediction model.

4.1 FMM – The Baseline Algorithm

The Fast Multipole Method (FMM) is a mathematical technique that greatly speeds up the calculation of long-ranged forces in the n-body problem. A detailed explanation of Fast Multipole Method can be found in [16]. In this appendix we will briefly describe Fast Multipole Method mathematically and specify the baseline CUDA algorithm for the same, which we have used in this study.

Mathematically speaking, Fast Multipole Method is about solving the following problem: Given a system of N source particles, with positions given by $\{y_1, . . . , y_N\}$, and N targets with positions $\{x_1, . . . , x_N\}$, we wish to compute the N sums,

$$f(x_i) = \sum_{j-1}^{n} K(x_i, y_i).\, s(y_j), \qquad i = 1, \ldots\ldots, N$$

Where; f(x) is the desired potential at target point x; s(y) is the density at source point y; and K(x,y) is the an interaction kernel that specifies the “physics” of the problem. In our particular case we model gravitational interactions.

The important components of FMM in our case are target boxes, source boxes and ulist. Depending on the dataset being processed we have a certain number of target boxes and a particular number of source boxes. Each target box and source box is a collection of a certain number of points. Each point is a quad-component single precision floating point structure representing x, y, z co-ordinates and another component called density potential. The aim is to calculate interaction between the points inside a target box with nearest neighbor points in certain source boxes. This mapping between target box and relevant source boxes is specified by the ulist. Thus each target box is processed by calculating the interaction between the points inside that box with the points in the source boxes as indicated by the ulist. Figure 15 shows the diagrammatic representation of FMM.

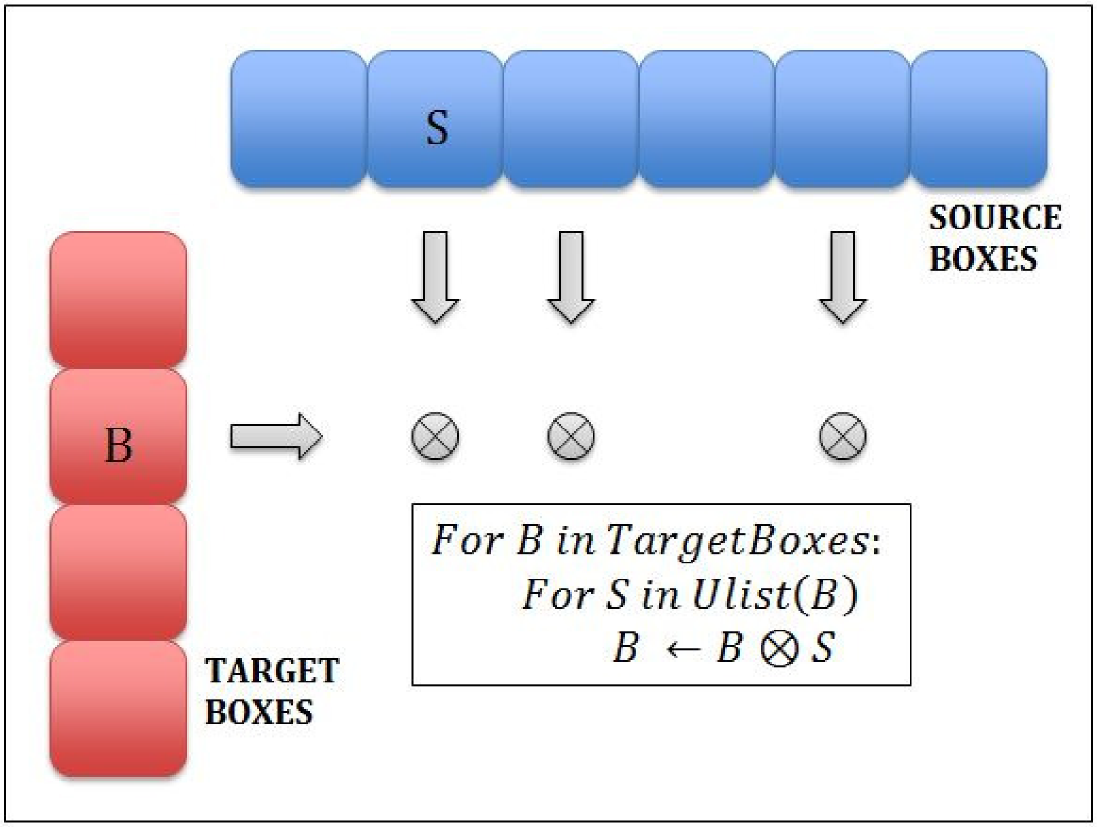

Figure 15: Fast Multipole Method representation

The computation for each target box can be performed independently. So from a CUDA programming model point of view target boxes can be mapped to blocks and the points within each target box can be mapped to threads within a block. Thus, naturally each target box i.e. block gets processed independently on the GPU.

The points within the target as well as source boxes are stored in a Structure of Array (SOA) form i.e. all x-coordinates of points are stored contiguously and same are the case for the other components. Index arrays are used to partition the points in each individual box. The following code snippet represents the CUDA kernel code for the baseline FMM algorithm on the GPU:

```
---------------------------------------------
int NB = blockDim.x;
int tboxid = blockIdx.x;
int di = threadIdx.x; // Local thread ID
int tid_min = TBptr[tboxid];  // Target box start
int tid_max = tid_min + TBn[tboxid];  // Target box end
int kbox_min = Uptr[tboxid];
int kbox_max = Uptr[tboxid+1];
for (int tid = tid_min + di; tid < tid_max; tid += NB) { // Targets
    float tx = Tx[tid];
    float ty = Ty[tid];
    float tz = Tz[tid];
    float td = 0.0;
    for (int kbox = kbox_min; kbox < kbox_max; ++kbox) { // Source
boxes
      int sboxid = Ulist[kbox];    // Source box ID
      int sid_min = SBptr[sboxid];
      int sid_max = sid_min + SBn[sboxid];
      for (int sid = sid_min; sid < sid_max; ++sid) {  // Sources
        float dx = tx - Sx[sid];
        float dy = ty - Sy[sid];
        float dz = tz - Sz[sid];
        float ds = Sd[sid];
        float rsq = dx*dx + dy*dy + dz*dz;
        float r = sqrtf (rsq);
        td += ds / r;
      } /* sid */
     } /* kbox */
    Td[tid] = OOFP_R* td;
  }
---------------------------------------------
```

Tx, Ty, Tz and Tw represent the SOA for points in the target boxes. TBptr and TBn are the indexing arrays that specifies the index for start of each target box and number of points in each target box respectively. Similarly Sx, Sy, Sz and Sw represent the SOA for points in the source boxes. SBptr and SBn are the indexing arrays that specifies the index for start of each source box and number of points in each source box respectively. The Ulist array contains a flat list of source boxes for each target box in succession. Uptr is the index array which provides demarcation for the start of each target box in the Ulist.

The particular dataset used in this study contains 585 target and source boxes. So the number of blocks is always 585. The number of points within a box varies from 0 to around 230. We run the FMM algorithms with various thread configurations like 64, 128, 192 and 256 threads. We have 47 different versions of the FMM kernel code that differ in the particular CUDA optimizations employed for improving performance over baseline.

4.2 FMM Optimization Strategies

The major optimizations that we considered for FMM are as follows:

4.2.1 Shared Memory

The baseline FMM algorithm is memory bandwidth limited. All target points within a block fetch the same source points and so there is a lot of data re-use. Thus shared memory can be used to overcome the memory bottleneck and achieve greater performance.

Normal code:

```
for (int kbox = kbox_min; kbox < kbox_max; ++kbox) { // Source boxes
          int sboxid = Ulist[kbox];     // Source box ID
          int sid_min = SBptr[sboxid];
    (cont'd)
```

```
---------------------------------------------------------
     (cont'd)
        int sid_max = sid_min + SBn[sboxid];
        for (int sid = sid_min; sid < sid_max; ++sid) {  // Sources
         float dx = tx - Sx[sid];
         float dy = ty - Sy[sid];
         float dz = tz - Sz[sid];
         float ds = Sd[sid];
         float rsq = dx*dx + dy*dy + dz*dz;
         float r = sqrtf (rsq);
         td += ds / r;
        } /* sid */
} /* kbox */
```

<u>Shared memory code:</u>

```
for (kbox = kbox_min; kbox < kbox_max; ++kbox) {//Source boxes
      const int sboxid = Ulist[kbox]; /* Source box ID */
      /* Loop over source points */
      const int sid_min = SBptr[sboxid];
      const int sid_max = sid_min + SBn[sboxid];

      int sid; /* Source point ID */
      for (sid = sid_min; (sid+NB) <= sid_max; sid += NB) {

        /* All threads help load 1 source point */
        extern __shared__ float SRCS__[];
        float* SX__ = SRCS__;
        float* SY__ = SX__ + NB;
        float* SZ__ = SY__ + NB;
        float* SD__ = SZ__ + NB;
        __syncthreads ();
        SX__[di] = Sx[sid+di];
        SY__[di] = Sy[sid+di];
        SZ__[di] = Sz[sid+di];
       (cont'd)
---------------------------------------------------------
-----------------------------------------------------------
       (cont'd)
        SD__[di] = Sd[sid+di];
        __syncthreads ( );
        int i;
        for (i = 0; i < NB; ++i) {
          float dx = tx - SX__[i];
          float dy = ty - SY__[i];
          float dz = tz - SZ__[i];
          float sd = SD__[i];
          float r2 = dx*dx + dy*dy + dz*dz;
          float r = sqrtf (r2);
          td += sd / r;
        } /* i */
      } /* sid */
      while (sid < sid_max) {//remaining points
        float sd = Sd[sid];
        float dx = tx - Sx[sid];
        float dy = ty - Sy[sid];
        float dz = tz - Sz[sid];
        float r2 = dx*dx + dy*dy + dz*dz;
        float r = sqrtf (r2);
        td += sd / r;
        ++sid;
      }
    } /* kbox */
-----------------------------------------------------------
```

4.2.1.1 How is this optimization captured by the model?

Usage of shared memory drastically reduces the number of global memory operations. Since the baseline code is bandwidth limited, this provides a significant boost to the performance. Table 12 shows a comparison between the baseline code and shared memory optimization for 64 threads per block.

Table 12: Comparison between Baseline and Shared Memory Optimization

	Baseline Kernel	**Shared Memory Optimized**
#Instructions	235880	262520
#Memory operations	47013	8854
MWP	9.722	9.070
CWP	16	4.425
Actual Performance(ms)	66.21315	35.952
Predicted Performance(ms)	66.33431	35.99694

From Table 12, it can be seen that CWP is greater than MWP for baseline, which suggests that baseline is memory bandwidth limited. However for share memory optimized case, MWP becomes greater than CWP which suggests that the memory bottleneck was removed by the optimization. It can also be noted that the number of memory operations has reduced drastically thus pushing MWP to become greater than CWP. The predicted performance is also in synergy with actuality which shows that our model can track this optimization quite accurately.

4.2.2 Unroll and Jam (Ujam)

The points within a target box get mapped to threads within a block. If the number of threads within a block is less than the number of points in the target box, there is another iteration of the outermost loop in which target points are fetched. So loop unrolling can be performed for each thread to process multiple target points to begin with. This also helps in reducing the pressure on memory bandwidth, since source points are fetched only once for multiple target points. The following code snippet shows the optimization:

Unroll and Jam(ujam) code:

```
  while ((tid_block + UNROLL_FACTOR*NB) <= tid_next) {
      const int t0id = tid_block + di + 0*NB;
      const real_t t0x = Tx[t0id];
      const real_t t0y = Ty[t0id];
      const real_t t0z = Tz[t0id];
      real_t t0w = 0.0;

      const int t1id = tid_block + di + 1*NB;
      const real_t t1x = Tx[t1id];
      const real_t t1y = Ty[t1id];
      const real_t t1z = Tz[t1id];
      real_t t1w = 0.0;

      /* Loop over source boxes in the U-list of tboxid */
      int kbox;
      for (kbox = kbox_min; kbox < kbox_max; ++kbox) {
        const int sboxid = Ulist[kbox]; /* Source box ID */
        /* Loop over source points */
        const int sid_min = SBptr[sboxid];
        const int sid_max = sid_min + SBn[sboxid];
        int sid; /* Source point ID */
        for (sid = sid_min; sid < sid_max; ++sid) {
          const real_t sx = Sx[sid];
          const real_t sy = Sy[sid];
          const real_t sz = Sz[sid];
          const real_t sw = Sd[sid];

          const real_t d0x = t0x - sx;
          const real_t d1x = t1x - sx;
          const real_t d0y = t0y - sy;
          const real_t d1y = t1y - sy;
        (cont'd)
```

```
---------------------------------------------------
        (cont'd)
          const real_t d0z = t0z - sz;
          const real_t d1z = t1z - sz;

          const real_t r0sq = d0x*d0x + d0y*d0y + d0z*d0z;
          const real_t r1sq = d1x*d1x + d1y*d1y + d1z*d1z;

          const real_t r0 = sqrtf (r0sq);
          const real_t r1 = sqrtf (r1sq);

          t0w += sw / r0;
          t1w += sw / r1;
        } /* sid */
      } /* kbox */
      if (t0id < tid_max)
        Td[t0id] += OOFP_R * t0w;
      if (t1id < tid_max)
        Td[t1id] += OOFP_R * t1w;
      tid_block += UNROLL_FACTOR*NB;
    } /* tid_0 */
---------------------------------------------------
```

UNROLL_FACTOR determines how many target points would be handled by each thread.

4.2.2.1 How is this optimization captured by the model?

Doing Ujam should ideally result in a decrease in memory operations for our case, since source data is re-used for multiple target points, which should increase the performance. On the downside, it can result in an increase in number of instructions and also an increase in the number of registers per thread thus hurting occupancy.
Table 13 shows the comparison between baseline and ujam optimization for the case of 64 threads per block.

Table 13: Comparison between Baseline and Ujam optimization

	Baseline Kernel	**Ujam Optimized**
#Instructions	235880	215413
#Memory operations	47013	36426.86
Occupancy	0.5	0.375
MWP	9.722	10.08268
CWP	16	12
Actual Performance(ms)	66.21315	49.972
Predicted Performance(ms)	66.33431	49.356

From Table 13 it can be seen that overall the performance increases and model tracks the increases performance quite accurately. MWP increase can be accounted for by the reduction in memory operations. CWP is reducing because of the active number of warps per SM decreases due to drop in occupancy.

4.2.3 Vector Packing (VecPack)

For the points within the boxes, the four components of the points are stored in SOA form in baseline. In case of Vector packing, these points are stored in AOS with the four components of each point stored contiguously. This enables the use of vector memory operations, which in turn brings down the total number of memory operations as compared to the baseline. Thus for the vector packing we have a single float4 vector memory load instead of four independent float scalar memory loads. The following code snippet shows the optimization:

Normal code:

```
float sx = Sx[tid];
float sy = Sy[tid];
float sz = Sz[tid];
float sd = Sd[tid];
//Sx, Sy, Sz and Sd are float arrays that store x, y, z and w components of
//all source points respectively
```

Vector Packing code:

```
float4 src = S[tid];
//S is a float4 array that stores packed x, y, z and w components for each
point and for all source points over all source boxes.
```

4.2.3.1 How is this optimization captured by the model?

One significant difference in VecPack optimization is that it uses vector memory operations as opposed to the scalar ones. Vector memory operations have different departure delays as compared to the scalar ones as has been described in Section 3.2.5. Also vector memory operations do not show memory merging effect. Also the overall number of memory operations is reduced since now we have one vector memory load instead of four scalar memory loads.

Table 14 shows the comparison between baseline configuration and Vector Packing case for 64 threads per block.

Table 14: Comparison between Baseline and Vector Packing Optimization

	Baseline Kernel	**Vecpack Optimized**
#Instructions	235880	165275
#Memory operations	47013	11893
Occupancy	0.5	0.5
MWP	9.722	3.969
CWP	16	8.486
Actual Performance(ms)	66.21315	44.318
Predicted Performance(ms)	66.33431	45.608

From the Table 14, it can be seen that VecPack reduces both the total instruction count as well as memory operations count. Also despite the decrease in the number of memory operations, CWP is still higher than MWP. This is because vector memory operations do not participate in memory transaction merging according to findings in Section 3.2.5. Also it can be seen that our model tracks these optimizations in a very accurate manner.

4.2.4 Fast Reverse Square Root (Rsqrt)

In the computation loop, it can be observed that there is square root operation and a floating point division operation. In GT200, the latency for the above two operations is 32 cycles and 36 cycles respectively. In comparison the latency for a "normal" operation like floating point multiplication is 4 cycles. So instead of using square root and division, we use the "rsqrt" operation (reciprocal square root) which has a latency of 16 cycles. Thus it reduces the number of compute instructions and eliminates some very high latency instructions. The following code snippet shows the optimization:

Normal code:

```
float rsq = dx*dx + dy*dy + dz*dz;
float r = sqrtf (rsq);
td += ds / r;
```

Rsqrt code:

```
float rsq = dx*dx + dy*dy + dz*dz;
td += ds * rsqrt (rsq);
```

4.2.4.1 How is this optimization captured by the model?

Using Reciprocal Square-root instruction has a direct impact on the total number of instructions for the kernel. This is what improves the performance. Table 15 shows a comparison between the baseline kernel and 'rsqrt' optimized one.

Table 15: Comparison between Baseline and Rsqrt optimization

	Baseline Kernel	**Rsqrt Optimized**
#Instructions	235880	224177
#Memory operations	47013	47013
MWP	9.722	9.722
CWP	16	16
Actual Performance(ms)	66.21315	64.98121
Predicted Performance(ms)	66.33431	66.33422

From Table 15 it can be observed that the model predicts a performance improvement but the magnitude of predicted improvement is quite less. This can be attributed to the fact the square root and reciprocal square root operations are performed by the SFUs which work in parallel with the SPs. Since our analytical model does not incorporate this parallel behavior our prediction is a little off in terms of magnitude.

4.2.5 Prefetching

In case of prefetching, a source point is pre-fetched before entering the loop, which iterates through all the source points in the box. Inside the loop we fetch the next source point while computation is performed on the previous source point and thus these operations can be done in parallel i.e. computation helps to cover up the latency of fetch to certain extent. Also the code is compacted by using make_float4 and make_float3. The following code snippet shows the optimization:

```
-------------------------------------------------
    Normal code:
    for (int sid = sid_min; sid < sid_max; ++sid) {  // Sources
      float dx = tx - Sx[sid];
      float dy = ty - Sy[sid];
      float dz = tz - Sz[sid];
      float ds = Sd[sid];
      float rsq = dx*dx + dy*dy + dz*dz;
      float r = sqrtf (rsq);
      td += ds / r;
    } /* sid */

  (cont'd)

-------------------------------------------------
```

(cont'd)

```
Prefetching code:
if (sid < sid_max)
      src_next = make_float4 (Sx[sid], Sy[sid], Sz[sid], Sd[sid]);
   while (sid < sid_max) {
      float4 src = src_next;
      ++sid;
      src_next = make_float4 (Sx[sid], Sy[sid], Sz[sid], Sd[sid]);
      float3 dr = make_float3 (tx - src.x, ty - src.y, tz - src.z);
      float rsq = dr.x*dr.x + dr.y*dr.y + dr.z*dr.z;
      float r = sqrtf (rsq);
      td += src.w / r;
   } /* sid */
```

4.2.5.1 How is this optimization captured by the model?

Prefetching is a difficult optimization for our current analytical model. It manifests in the form of an increase in total instructions as well as memory operations which are counter-intuitive for the model for performance improvement. In reality, the advantage of prefetching comes with the fact that due to this optimization you have a number of compute instructions which are independent of a memory operation (since the relevant load has been pre-fetched) and hence can be executed while the memory operation is being serviced. Our model does not recognize such an opportunity at this time and mostly prefetching optimizations do not give an accurate prediction in the model.

Table 16 shows the comparison between the baseline case and prefetching optimization.

Table 16: Comparison between Baseline and prefetching optimization

	Baseline Kernel	**Prefetching Optimized**
#Instructions	235880	331252.1
#Memory operations	47013	47253.99
MWP	9.722	9.722
CWP	16	15.317
Actual Performance(ms)	66.21315	62.896
Predicted Performance(ms)	66.33431	64.672

From Table 16, it can be seen that prefetching leads to an increase in both the total number of instructions as well as the number of memory operations.

4.2.5.2 Failed attempts at getting prefetching to work:

We had several lines of thought as to why prefetching was being a particularly difficult optimization to predict performance for:

Decuda: We knew that there is a difference in the PTX file and what gets executed on the GPU. We thought that perhaps this difference might be a deciding factor in case of prefetching. So we compared the PTX file for baseline and prefetch with the decuda dissembled PTX files (decuda_ptx) for the same. We saw that the number of 'mov' instructions in baseline was 8 for the original PTX but it was 22 for the baseline decuda_ptx. On the other hand, for original prefetch PTX the number of 'mov' was 27 while it was 28 for decuda_ptx.

This disparity in the increase in 'mov' instructions between baseline and prefetch for original PTX and decuda_ptx led us to suspect that it might be an issue of source PTX being an unoptimized one. So we tried to use the decuda PTX with Ocelot for emulation. However we had multiple difficulties in having a decuda_ptx to work with Ocelot:

a) The register usage was not well defined and segregated according to data type. For e.g. r0 was once used for u32 and then later on for f32 data type. Original PTX had well defined register usage. We had difficulties in establishing a correspondence between original PTX and decuda code.

b) The syntax and semantics of some decuda instructions were not true to the ISA docs and were ambiguous in interpretation. Especially 'setp' (used for predication) and 'cvt'(convert and move) were quite off from original PTX.

So we quit the attempt of using decuda for creating a custom PTX to be run with Ocelot.
Cuobjdump: We then came to know of cuobjdump as another disassemble for cubin files and tried to use it for creating a custom PTX. But the 'assembly' code given out by cuobjdump was very different than the PTX ISA. Also there is no documentation on the instruction format for these 'assembly' instructions, so creating a PTX from the assembly proved to be quite difficult.
We then gave up our attempts of creating a custom PTX and qualitatively reasoned out why prefetching wasn't working for our model. Since our other optimizations are accurately tracked by the model we concluded that unoptimized PTX is not a reason for the model to falter.

4.2.6 Other Optimizations

Following are some minor optimization used in the FMM suite:

4.2.6.1 Tight

In this optimization, the code uses make_float4 or make_float3 and performs computation on float4 or float3 vectors instead of having different float variables for each component. This mostly does not provide any optimization since in the PTX the operations are performed on individual float components instead of as a vector.

4.2.6.2 Trans

In case of shared memory optimization cases, when data is loaded into a shared memory, there are two choices of doing so; AOS or SOA. The default mechanism used in the FMM suite is SOA, but in case of "Trans" optimization AOS is used for shared memory. This does provide benefit in certain cases and has implication of shared memory banked conflicts as compared to "non-trans" versions.

4.2.6.3 Combinations

All the previously mentioned optimizations are combined with each other to create additional optimizations. To give an example, the code version "cuda_vecpack_shmem_pref_ujam_rsqrt" combines all of the base optimizations into a single code. In this manner we have 44 different optimized versions of the FMM CUDA kernel used in this study.

4.2.6.4 Varying number of threads

Having different number of threads per block has major implications on occupancy and resource usage, and ultimately performance. For e.g. even if optimization X is the best performer for 64 threads per block, it might perform badly when number of threads is increased to 256. We have used four different thread combinations of 64, 128, 192 and 256 threads per block in this study. So in totality we have around 170 cases of the FMM kernel in this suite. The number of blocks is kept constant at 585 since it is related to the dataset size used.

In the next chapter we look at the results of the study.

CHAPTER 5

RESULTS AND ANALYSIS

We have applied our GPU Analytical model to the entire suite of FMM kernels and show the results in this section. We also explain how the CUDA Performance Analyzer would be useful to a CUDA programmer through examples.

5.1 Summary of Results

We test the FMM kernel suite with four different threads/block; 64, 128, 192 and 256. We compare the performance prediction of our model with actual performance on a Tesla C1060. We have used CUDA Toolkit 2.3 in this study since it mostly concentrates on the GT200 GPU Architecture. Figure 16 gives an overall look at how well the model is able to track the different optimizations.

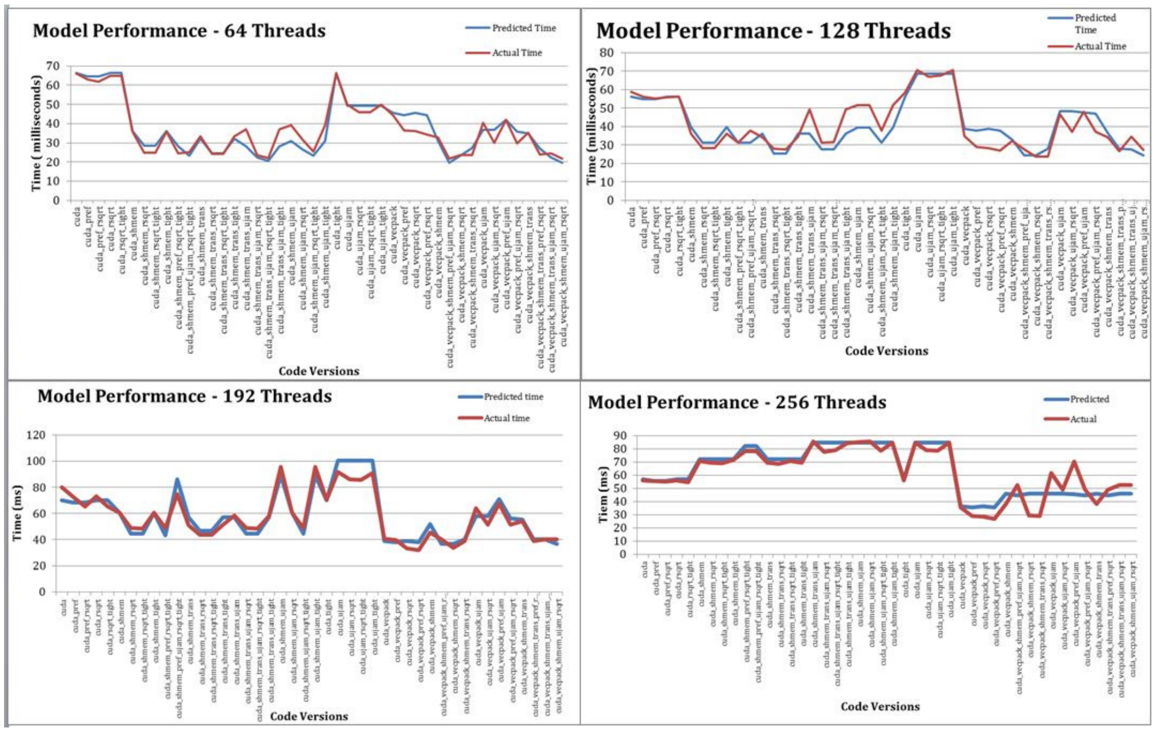

Figure 16: Coherency between predicted and actual performance

In order to look at the base optimizations in more detail, one can refer to Figure 17. It represents a comparison between the predicted and actual performance for the base optimizations for all four thread combinations. Each block in the graph represents a pair of threads/block and an optimization. The color of the block gives us the predicted performance (the color translates to a number through the color bar on the right). And the number inside the block gives the actual execution time in milliseconds. For ideal tracking, these two should be the same.

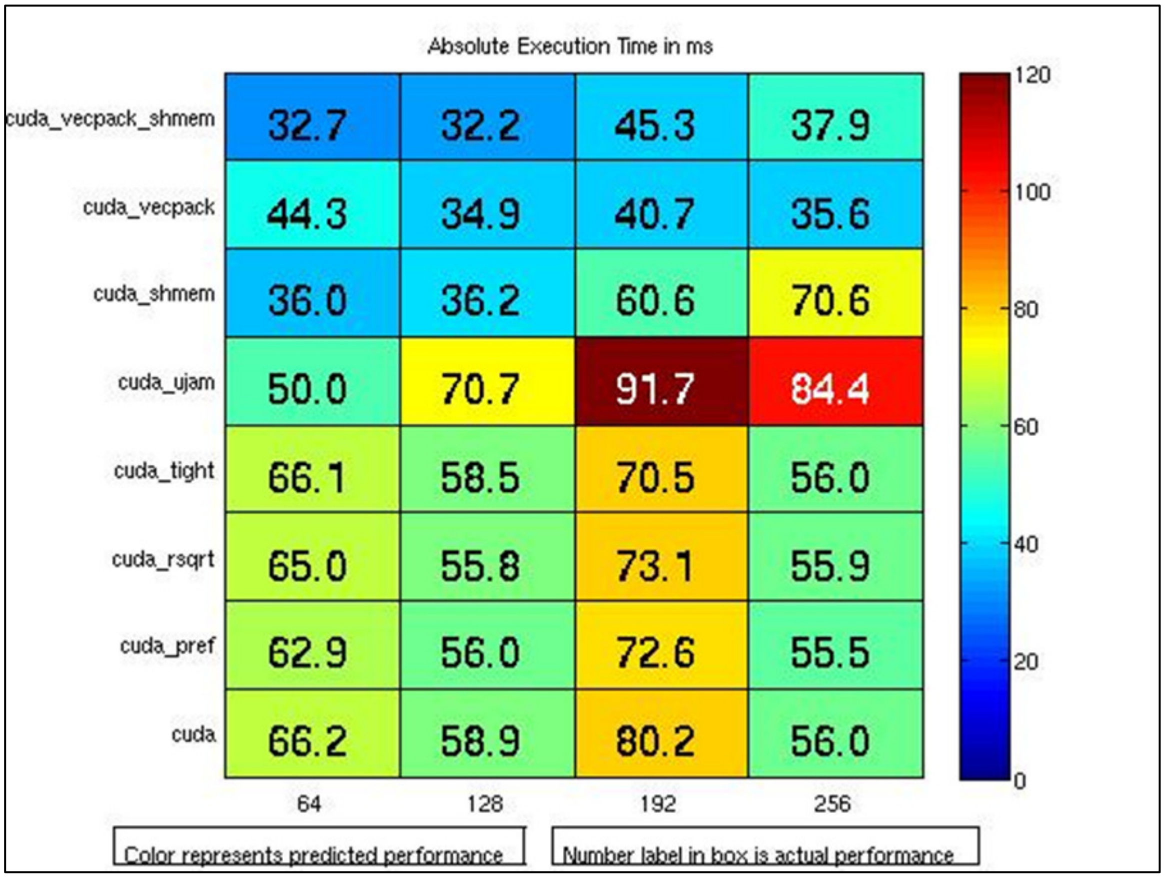

Figure 17: Predicted Vs. Actual Performance 3D plot

The deviation in predicted and actual performance is mostly seen in rsqrt and prefetch cases and their combinations, for reasons discussed in Section 4.2.4 and 4.2.5 respectively.

5.2 Data Analysis

We go deeper into the 64 threads per block case and try to stress on the usefulness of the CUDA Performance Analyzer to a CUDA programmer.

5.2.1 Compute Bound Optimizations

Figure 18 shows base optimization performance on Tesla C1060, in Gflops/sec. Further we select the base optimization which gives highest performance (shared memory for 64 threads case) and add optimizations over it for better performance. This can be seen from Figure 19 where '^' refers to the base optimization of shared memory for all the code versions specified on the x-axis. Similar convention is used for rest of the figures also, wherever relevant.

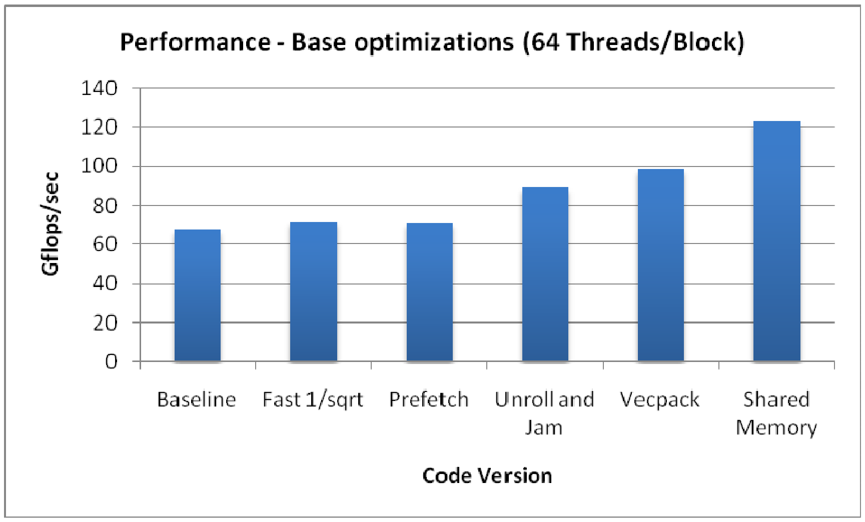

Figure 18: Base Optimizations with 64T

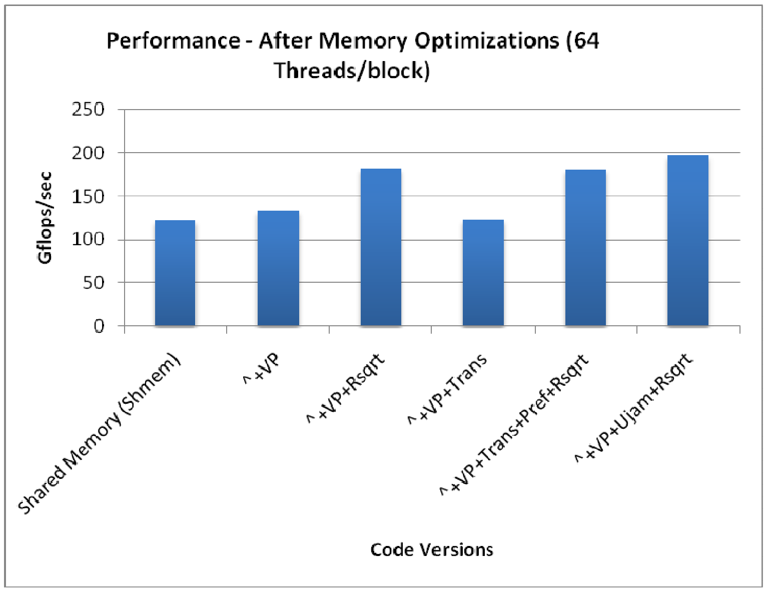

Figure 19: Another layer of optimizations over Shared Memory with 64T

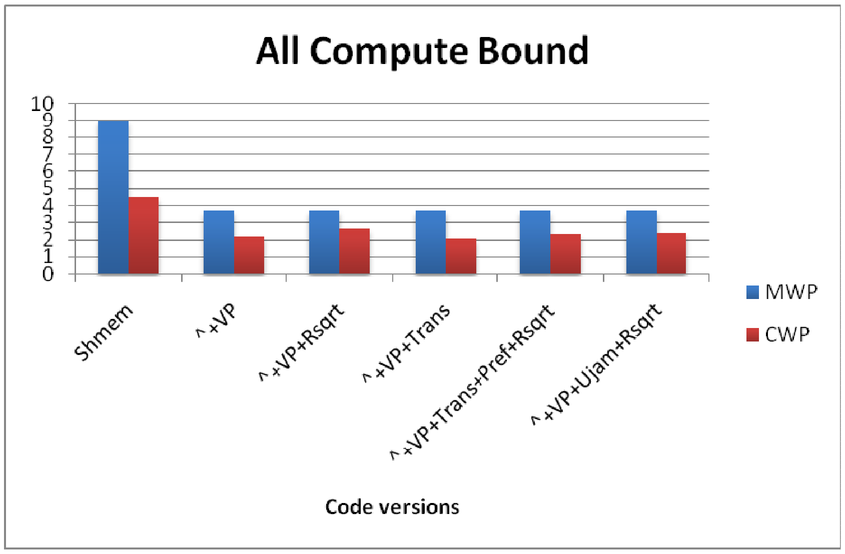

Figure 20: MWP & CWP for optimized versions with 64T

Figure 20 shows the MWP and CWP values obtained by applying the analytical model to the optimized cases in Figure 19. It can be seen that all these optimizations result in MWP greater than CWP which is a rough indicator of these kernel being compute bound at this point. To reinforce this notion, Figure 21 shows the performance of these optimized kernels along with the #instructions metric and shows their correlation.

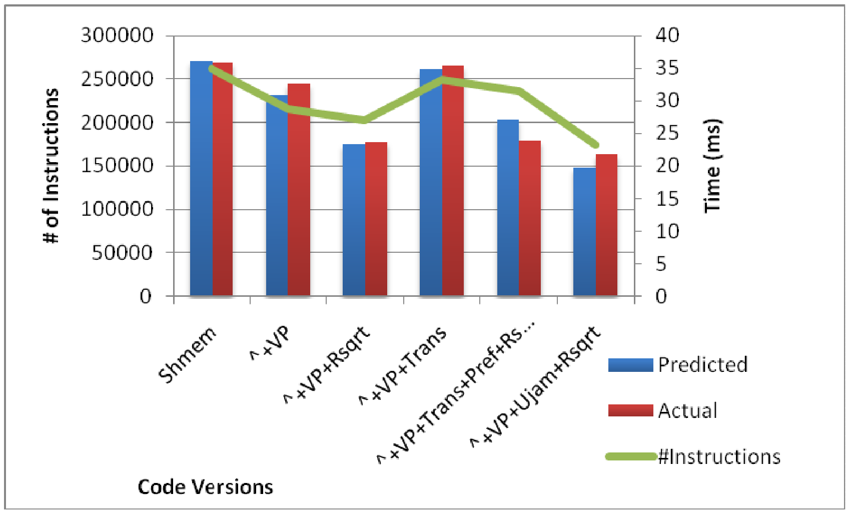

Figure 21: Performance & #Instructions Correlation with 64T

Thus for compute bound optimized kernels the performance is tracked fairly well by the #instructions metric. This is useful information for the programmer if he wishes to do further optimizations which share such characteristics.

Figure 22 shows optimization combinations for 128 threads/block, which are compute bound. It shows a good correlation with the number of instructions in the kernel as was also true for the 64 threads per block case.

5.2.2 Memory Bound Optimizations

Some optimization combinations could result in the resultant kernel being memory bound still. We have such a set represented in Figure 23. Observe that CWP is greater than MWP for all these cases.

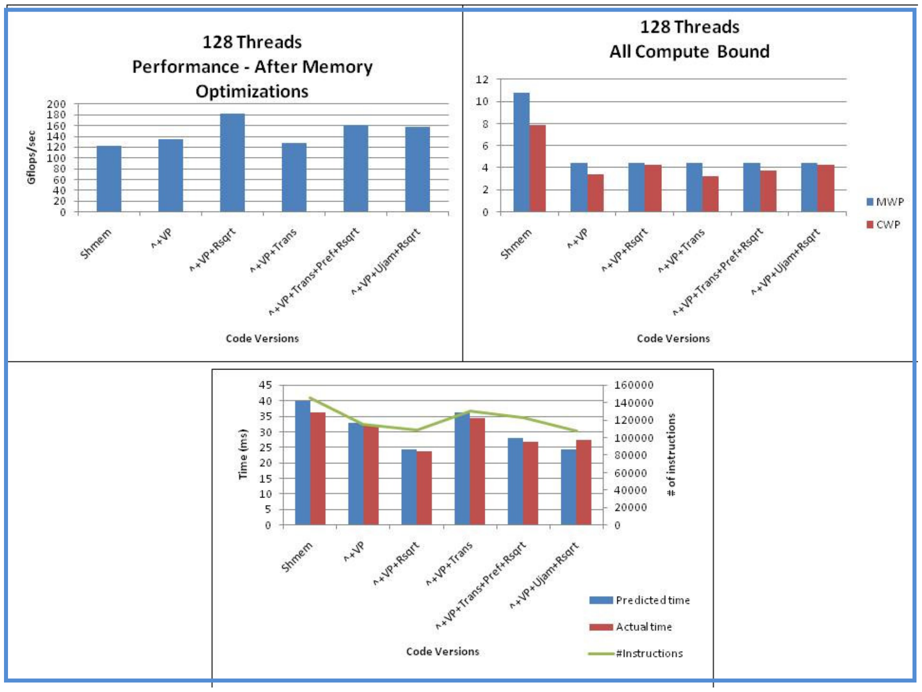

Figure 22: Performance & #Instructions correlation with 128T

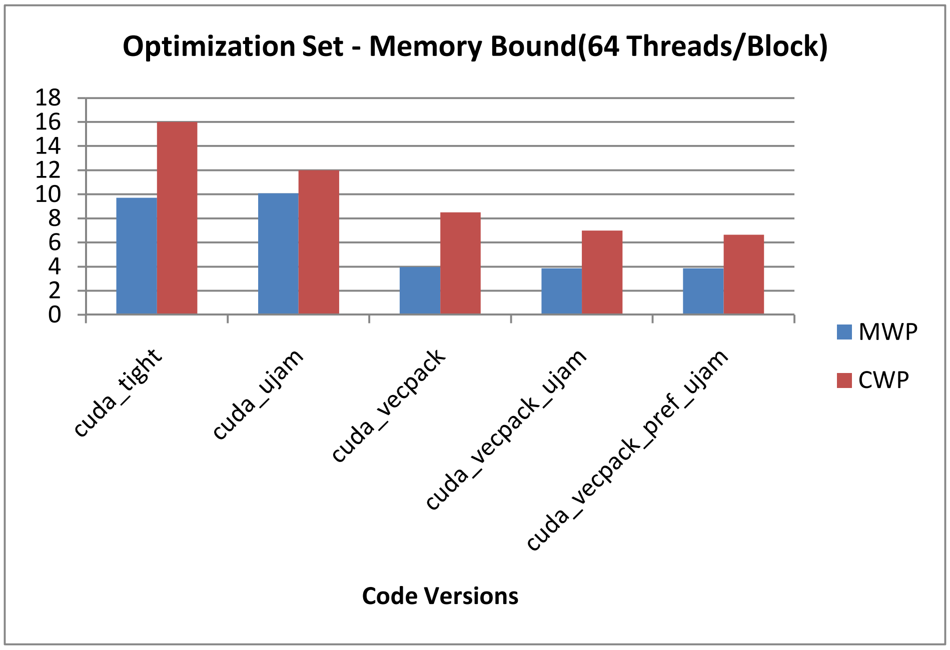

Figure 23: Memory bound optimizations with 64T

We found that a good indicator for performance in optimizations where CWP is greater than MWP, is the #Memory_Operations. Intuitively this makes sense since varying #Memory_Operations is changing the stress on the memory bandwidth for a memory bound kernel, so performance should change accordingly. This is also in sync with the concept in Figure 21 where the kernels were compute bound and were correlated with the number of total instructions. Figure 24 shows the correlation between Performances for memory bound optimized kernels and their respective #Memory_Operations in the kernel.

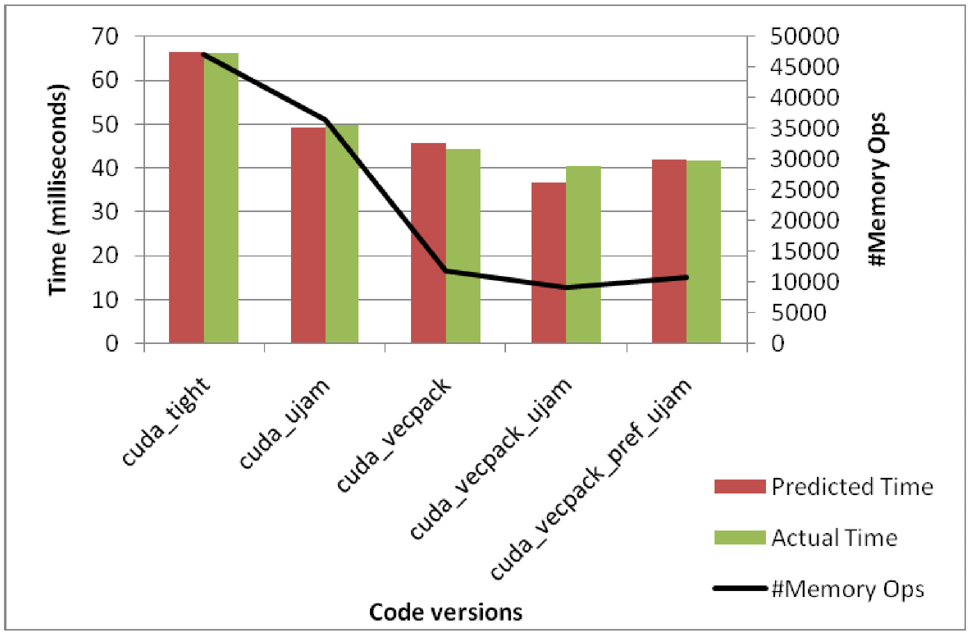

Figure 24: Performance & #Memory_Ops Correlation with 64T

Similar analysis was performed on 128 Threads per block case to ensure that the model was consistent when this factor was changed.

Figure 25 considers optimizations which keep the kernel memory bound with 128 Threads/block. If the performance of these kernels is compared to the number of memory operations per kernel, a direct relationship between them is established to be present.

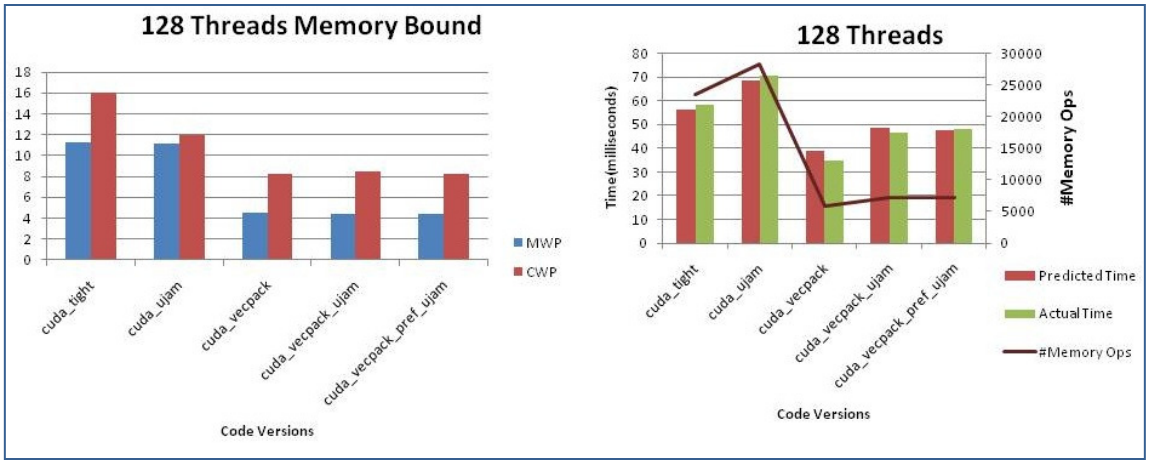

Figure 25: Performance & #Memory_Ops Correlation with 128T

5.2.3 3D Sign Plot

With so many different cases and thread combinations it may become difficult to see patterns in the data for analysis. Another effective way for representing data is shown in Figure 26. It's a 3D Sign plot, where the performance of one version is compared to another for the case of 64 threads/block. '1' represents an improvement in performance over 10% for kernel listed on y-axis over kernels on x-axis; a '0' represents performance parity within plus-minus 10% and '-1' represents degradation in performance over 10%. Another dimension is present because of the color, which represents performance difference in predicted data. The numbers in the boxes on the other hand represent performance difference in actual data. Thus if the magnitude and color is a mismatch we know that the performance prediction is off for that combination of kernels. So based on Figure 26, boxes where you have a '.0' with a blue color, it means that actually the performance of the two cases is equal but the model thinks that one performs worse than the other. Similarly there is discrepancy in prediction when we have a '.0' with maroon color or '1.0' with a green color.

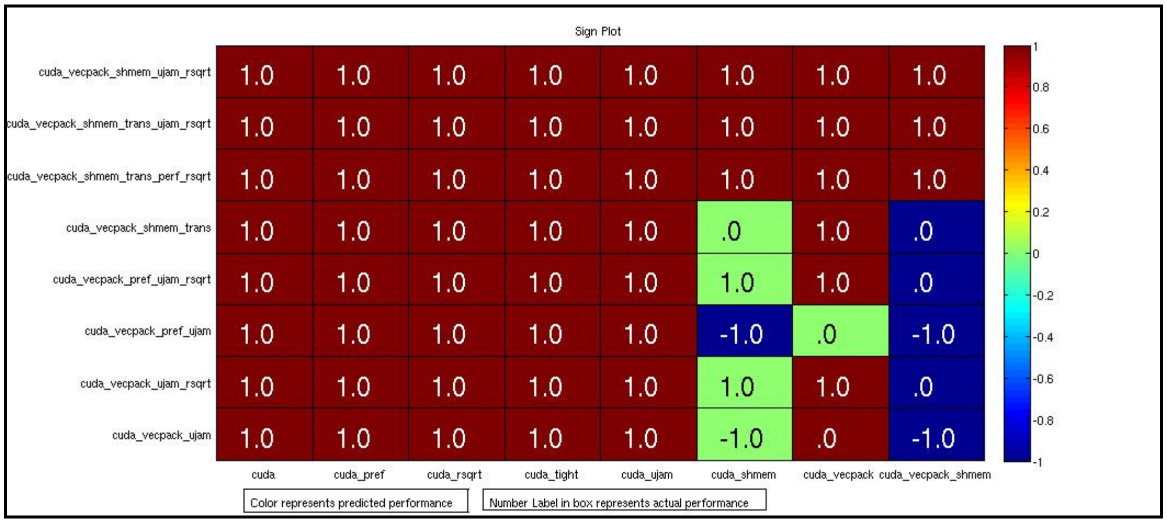

Figure 26: 3D Sign Plot for 64 Threads/block

5.2.4 Determining intensity of Memory Merging Effect

The amount of memory transaction merging does not depend on duplicate loads alone. This is because memory controllers have limited capability of realizing merging of transactions depending on the queue size in controller and other hardware peculiarities. In case of FMM kernel the duplicate loads factor was coming in due to the fact that the warps within a thread block access the same source point data. So a first degree duplicate load factor would be the number of warps per thread block (for FMM).

But other factors that affect the duplicate load are:

5.2.4.1 Occupancy

If the occupancy is high, we have higher number of active warps per SM, which mostly would mean a higher number of thread blocks per SM. Since in FMM, duplicate loads take place amongst warps within a thread block, a higher number of thread blocks per SM means a lower potential for merging since the memory controller queue would be filled with memory requests from diverse warps from different blocks which are trying to access different unique locations. On the other hand, if the occupancy is low, there is a smaller number of active blocks and the memory controller queue would have more requests from warps within the same block which it can merge due to duplicate loads.

5.2.4.2 Number of threads per block

If there are a high number of threads per block, then for a fixed occupancy, there will be lesser active blocks per SM as compared to the case when you have low number of threads per block. Thus if the occupancy is frozen, having more number of threads per block leads to higher potential merging of memory requests. Also considering the merging factor properly is more critical in cases of high number of threads per block since our first degree duplicate load count is the number of warps per block which would be quite high, and thus affect performance in a significant way.

Thus there is a relationship between the occupancy, number of threads per block and the duplicate load factor. In this study we have not defined a formal relationship between them but consider it in a discrete manner; i.e. for higher occupancy kernels we consider a lower effective duplicate load factor, while for lower occupancy kernels we consider the effective duplicate loads to be high. In Figure 27 we show the case of 192 threads per block and show the variation of occupancy across the different kernels. We also show ideal duplicate loads and effective duplicate loads, the latter being duplicate loads which account for occupancy.

Observe the inverse relationship between occupancy and effective duplicate loads.

Also in Figure 27, note that towards the end of kernel list ideal as well as effective duplicate loads are flat at '1' since Vector packing optimized kernels do not show any memory merging effect as discussed in section 3.2.5.

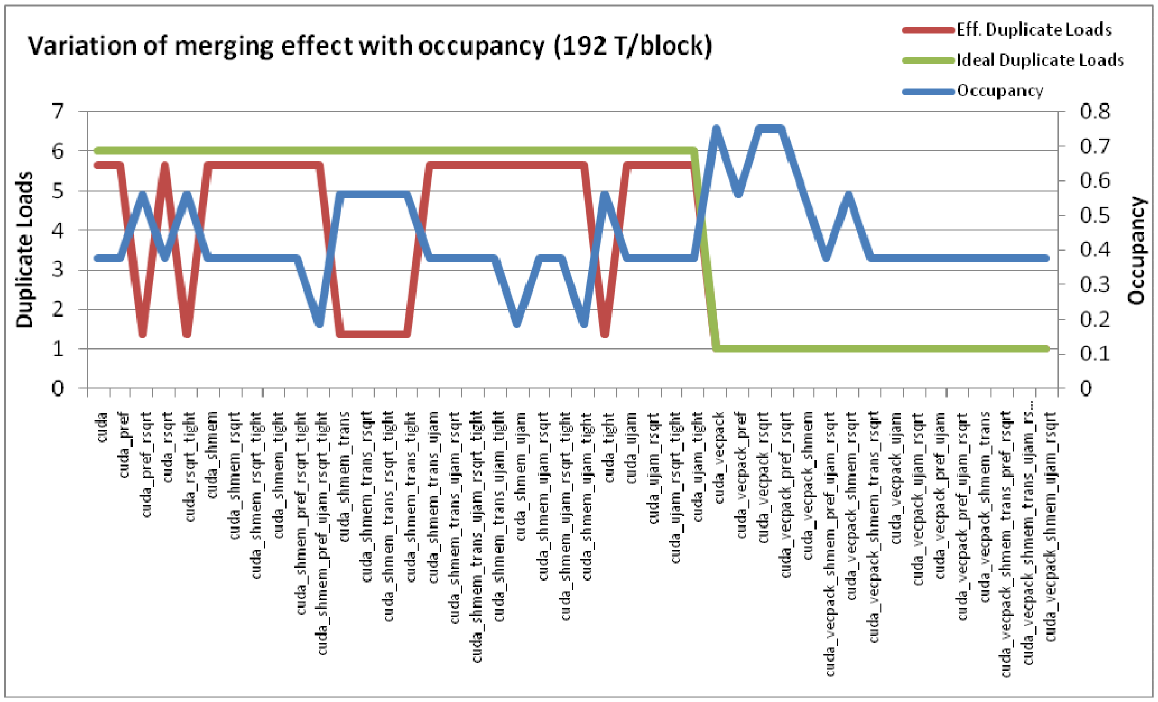

Figure 27: Relationship between Occupancy and Effective duplicate load factor (192T)

In Figure 28, we consider the duplicate load factor to be constant and unchanging with occupancy and show comparison between actual and predicted performance across the range of FMM kernels. For Figure 29, we change the effective duplicate load factor to be adaptive with occupancy.

It can be clearly seen that in Figure 29 we have a more accurate prediction of kernel performance due to consideration of effective duplicate load factor which is adaptive with occupancy.

Figure 30 shows the variation in occupancy across the FMM suite for 256 threads per block. Again we have an inverse relationship between occupancy and effective duplicate loads. Figure 31 and Figure 32 show performance of prediction with and without consideration of effective duplicate load adaptive with occupancy for 256 threads per block. As expected, considering the adaptive nature of effective duplicate loads improves the performance of prediction.

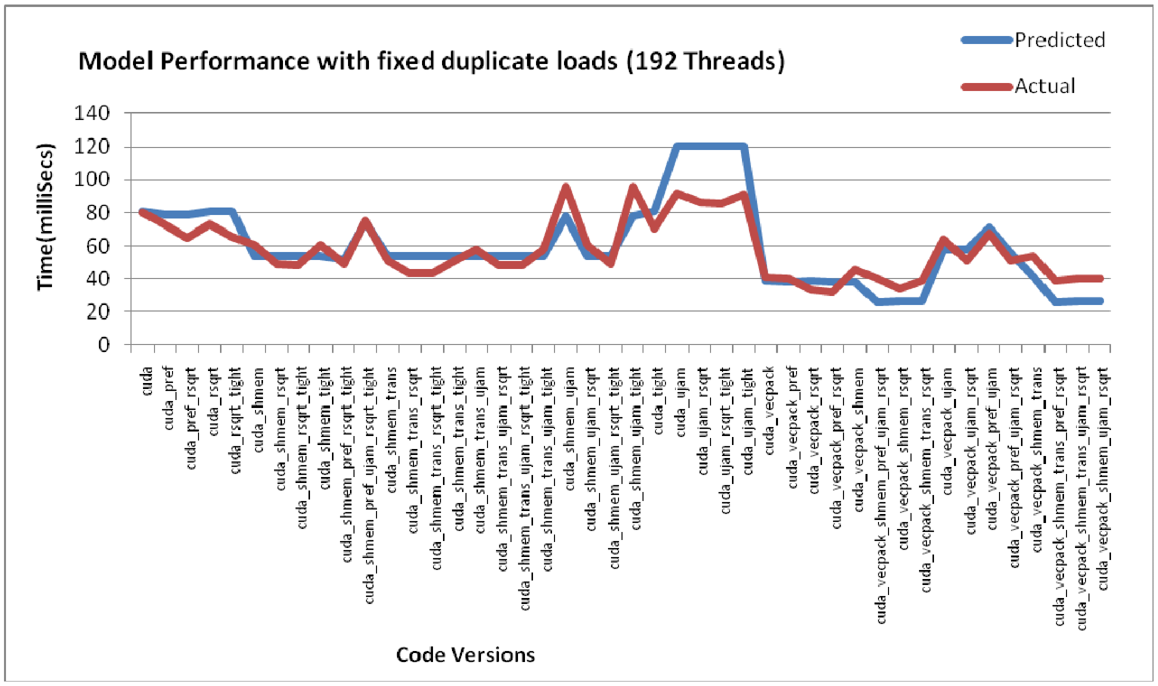

Figure 28: Model performance with fixed duplicate loads (192T)

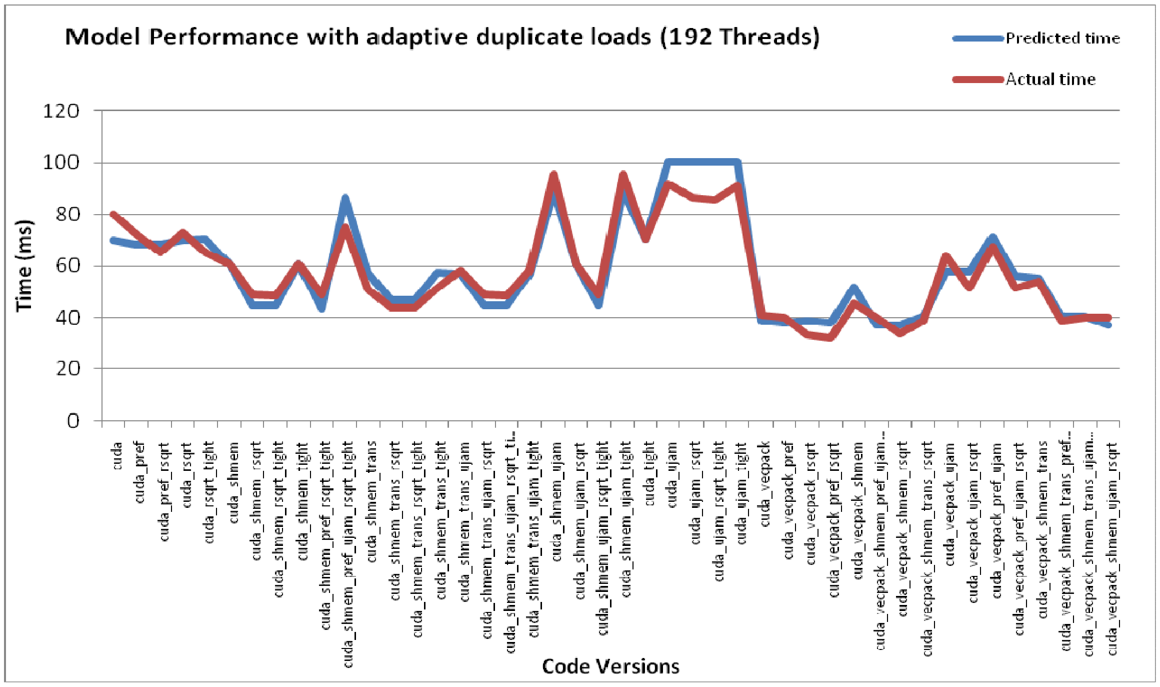

Figure 29: Model performance with adaptive duplicate loads (192T)

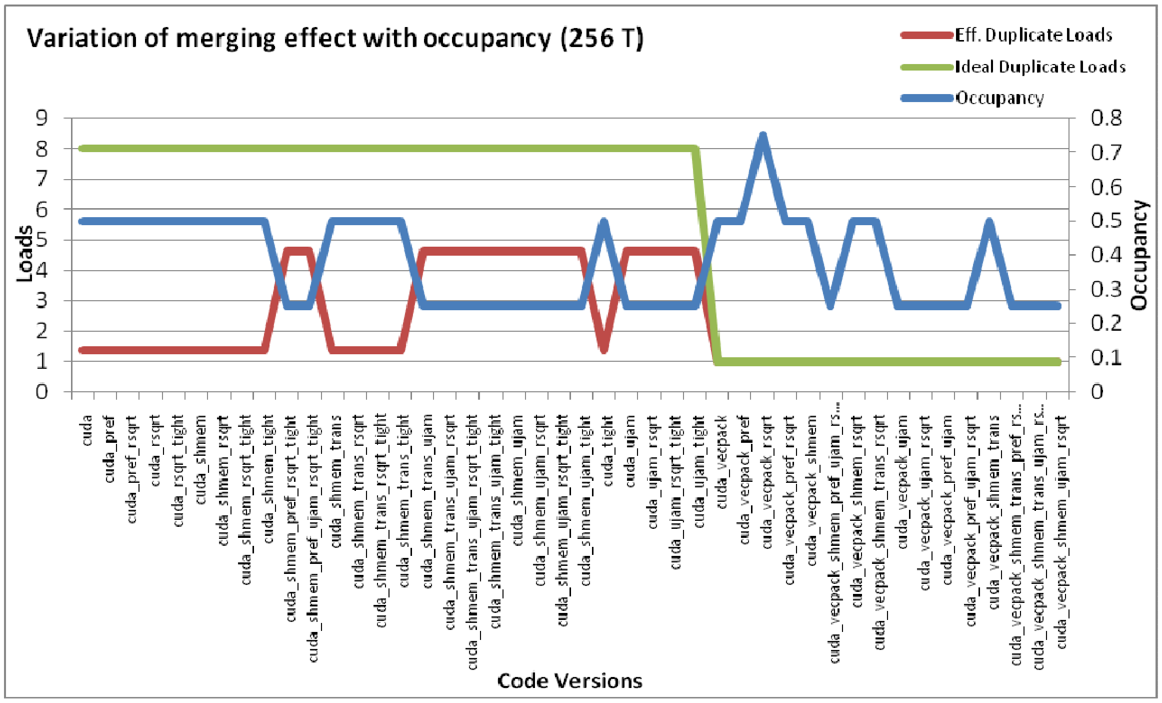

Figure 30: Relationship between Occupancy and Effective duplicate load factor (256T)

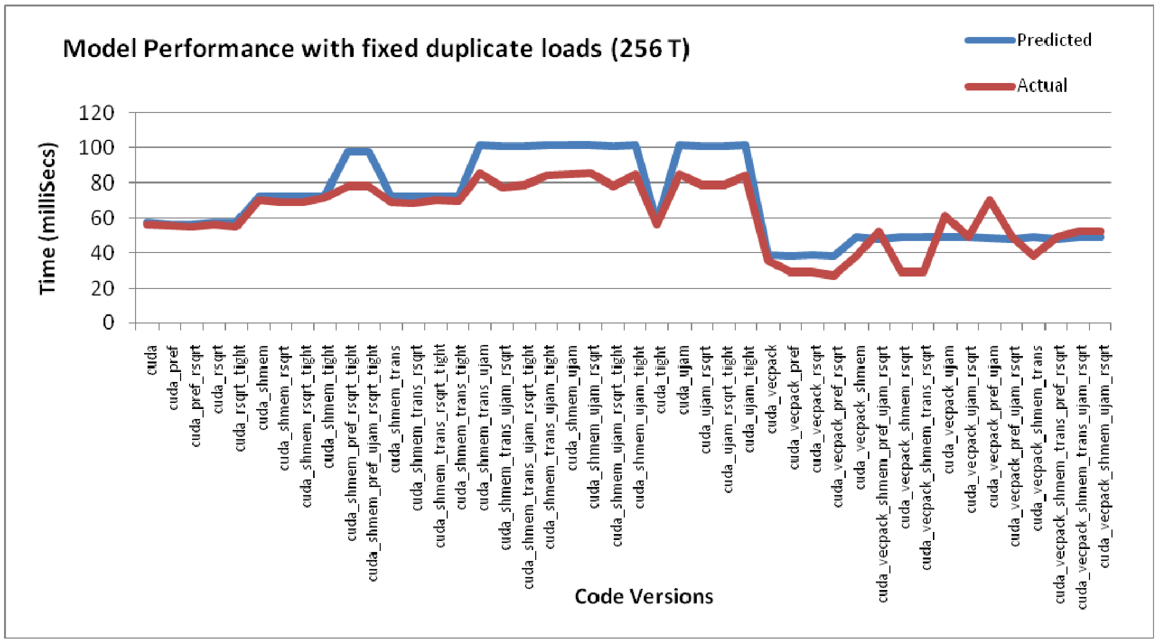

Figure 31: Model performance with fixed duplicate loads (256T)

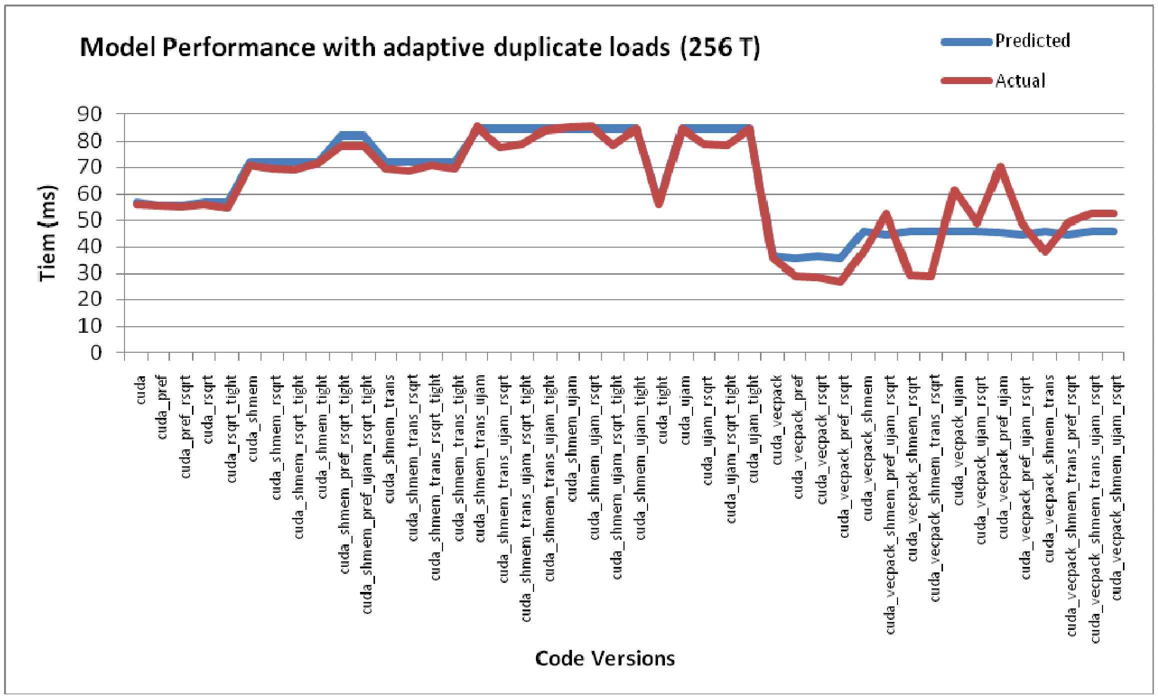

Figure 32: Model performance with adaptive duplicate loads (256T)

Thus it is seen that the duplicate load factor is quite sensitive to occupancy and number of threads especially when the number of threads per block is high.

Based on all the results, it can be said that our GPU analytical model is quite robust since it can track a wide spectrum of optimizations barring a few. It gives the developer a holistic perspective regarding what makes the performance of a kernel tick. In the following section we conclude giving some interesting approaches and ideas of utilizing our GPU analytical model.

CHAPTER 6

CONCLUSION

Thus in this study, we present a robust and accurate GPU analytical model for the GT200 GPU Architecture. A recap of the important features of CUDA Performance Analyzer is listed below:

- Modeling of the memory transaction based Coalescing model
- Accurate shared memory modeling
- In-depth consideration of the memory transaction merging effect
- Vector memory operations modeling
- Modular nature of the analyzer with a Frontend Data Collector and GPU Analytical Model based Predictor
- Ability to use either CUDA Visual Profiler or GPU Analytical Model as Frontend Data Collector
- Deep coverage of studied optimizations through an extensive optimization suite

However the CUDA Performance Analyzer can be taken to the next level through some additional features like the ones suggested below:

- Compiler Tools Support: At this point, some of the metrics used in the model like duplicate loads, independent loads have been calculated through manual code analysis. It would be good to have a compiler tool support the calculation of these factors and automatically feed to the Analyzer.
- Auto-tuning feature: Now that we know about all the gears that tinker with performance, it would be great to have an auto-tuning tool that tunes through factors like MWP, CWP, Duplicate Load factor, #Instructions etc to automatically suggest an optimal configuration for an optimized kernel.

- Additional features in GPU Model: The GPU Analytical model can be made more advanced by handling issue of parallel execution between SPs and SFUs, execution of independent compute instructions in parallel with memory operations etc. Also the model can be taken to the next level with adding support for the Fermi Architecture.

APPENDIX A

MICROBENCHMARK

This appendix illustrates the various micro-benchmarks used in tuning the model, showing the traits of memory merging effect and studying the effect of vector memory operations.

A.1 Microbenchmark for Scalar Memory Operations

The base micro-benchmark kernel code is as given below:

```
__global__ void gflops(int Num_Iterations, int blocksize, float *dm_src)
{
  int ori_index = blockIdx.x*blocksize + threadIdx.x;
  //ori_index can be changed to create different memory access patterns
  int index = ori_index;
  float loadregister;
  __shared__ float result[520];
  for (int i = 1; i <= Num_Iterations; i++)
  {
    FMAD2(a, b)
    FMAD2(a, b)
    FMAD2(a, b)
    FMAD2(a, b)
    loadregister = dm_src[index];
    index+=loadregister; //dependent load on the previous LD
    (cont'd)
    (cont'd)
    FMAD2(a, b)
    FMAD2(a, b)
    FMAD2(a, b)
    FMAD2(a, b)
    loadregister = dm_src[index];
    index+=loadregister; //dependent load on the previous LD
    FMAD2(a, b)
    FMAD2(a, b)
    FMAD2(a, b)
    FMAD2(a, b)
   loadregister = dm_src[index];
   index+=loadregister; //dependent load on the previous LD
    FMAD2(a, b)
    FMAD2(a, b)
    FMAD2(a, b)
    FMAD2(a, b)
    loadregister = dm_src[index];
    index+=loadregister; //dependent load on the previous LD
    FMAD2(a, b)
    FMAD2(a, b)
    FMAD2(a, b)
    FMAD2(a, b)
}
if (threadIdx.x==0 && blockIdx.x==0) result[threadIdx.x] = a + b + index;
```

FMAD2(a,b) represents two floating point multiply-add operations with variables ‘a’ and ‘b’. Customizations through the micro-benchmark:

1. “ori_index” field can be changed to create different access patterns in the data. This also helps in creating memory access patterns that would create only a particular type of memory transaction, for e.g. only 32 byte memory transactions would be spawned.
2. The global memory array “dm_src” can be programmed in the host code to contain any value. Depending on this value memory access patterns for threads within a warp can be changed.
3. The number of load operations in the ‘for’ loop can be changed using the CONFIG macro. The code listing above shows the case of CONFIG=4. For e.g. for tuning the model parameter of “frequency” as given in section 3.2.6, one needs to have CONFIG=0 which implies no memory load operations inside the loop. Also the macro ‘ITERS’ refers to the number of iterations of the ‘for’ loop and is generally kept fixed at 100.

Table 17 lists the different variations of the micro-benchmark for various purposes like tuning of the model, showing the memory merging effect etc. The indexing scheme points to the location from where a thread would make the first memory access. Also each thread goes on to make multiple memory accesses due to the load instruction in the ‘for’ loop, and the step size of subsequent accesses is given by contents of global memory array “dm_src”. Manipulation of these two parameters can result in different kinds of memory access patterns. For e.g. for avoiding any kind of memory merging effect the “index” can be formed such that each block of threads accesses a different chunk of global memory space; while factors like “dm_src” value, CONFIG and ITERS can be used to ensure that within a block, each thread accesses independent memory locations throughout each iteration of the loop.

Table 17: Memory access patterns for Scalar Memory Operations

Operation	Indexing scheme	Value of global memory array "dm_src"	Access Pattern Illustration
Tuning of departure delay for 32 byte transactions	(blockIdx.x*(((int)blocksize/(int)WARPSIZE)*(CONFIG* ITERS * WARPSIZE))) + (((int)threadIdx.x /(int)WARPSIZE) * (CONFIG *ITERS * WARPSIZE)) + ((((int)(threadIdx.x % WARPSIZE)/(int)16)*8) + ((threadIdx.x % WARPSIZE)) % 8)	32	Fig 9
Tuning of departure delay for 64 byte transactions	(blockIdx.x*(((int)blocksize/(int)WARPSIZE)*(CONFIG* ITERS * WARPSIZE))) + (((int)threadIdx.x/(int)WARPSIZE)*(CONFIG *ITERS * WARPSIZE)) + (threadIdx.x % WARPSIZE)	32	Fig 11
Tuning of departure delay for 128 byte transactions	(blockIdx.x*(((int)blocksize/(int)WARPSIZE)*(CONFIG* ITERS * WARPSIZE))*2) + (((int)threadIdx.x/(int)WARPSIZE)*(CONFIG *ITERS * WARPSIZE)*2) + ((threadIdx.x % WARPSIZE)*2)	64	Fig 33

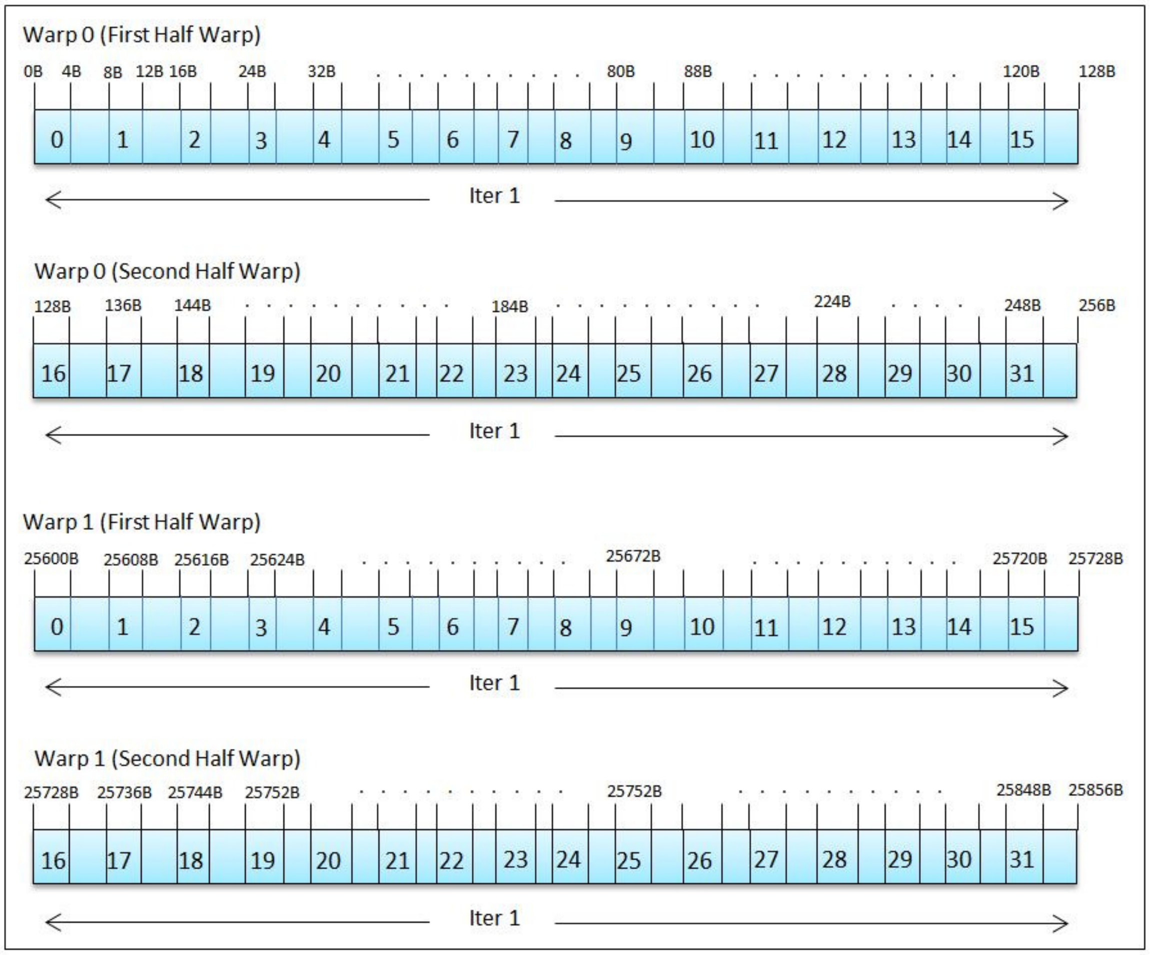
Warp 0 (First Half Warp)
0B 4B 8B 12B 16B 24B 32B . . . . 80B 88B 120B 128B
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Iter 1
Warp 0 (Second Half Warp)
128B 136B 144B . . . . 184B . . . . 224B 248B 256B
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Iter 1
Warp 1 (First Half Warp)
25600B 25608B 25616B 25624B . . . . 25672B 25720B 25728B
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Iter 1
Warp 1 (Second Half Warp)
25728B 25736B 25744B 25752B . . . . 25752B 25848B 25856B
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Iter 1

Figure 33: Memory access pattern for 128 Byte transactions

A.2 Microbenchmark for Vector Memory Operations:

The micro-benchmark kernel code for vector memory operations is given below:

```
__global__ void gflops(int Num_Iterations, int blocksize, float4 *dm_src)
{
    int ori_index = ((blockIdx.x)*((int)blocksize/(int)WARPSIZE)*ITERS*CONFIG*2) +
(((int)threadIdx.x/(int)WARPSIZE)*ITERS*CONFIG*2);
  float index3 = 0;
  float4 loadregister;
  __shared__ float result[520];
  float a ;
  float b ;
  for (int i = 1; i <= Num_Iterations; i++)
  {
    FMAD2(a, b)
    FMAD2(a, b)
    FMAD2(a, b)
    FMAD2(a, b)

    loadregister = dm_src[index];
    index+=loadregister.w;
    a+=loadregister.x;
    b+=loadregister.y;
    index3+=loadregister.z;
    (cont'd)
```

```
(cont'd)
FMAD2(a, b)
FMAD2(a, b)
FMAD2(a, b)
FMAD2(a, b)

loadregister = dm_src[index];
index+=loadregister.w;
a+=loadregister.x;
b+=loadregister.y;
index3+=loadregister.z;

FMAD2(a, b)
FMAD2(a, b)
FMAD2(a, b)
FMAD2(a, b)

loadregister = dm_src[index];
index+=loadregister.w;
a+=loadregister.x;
b+=loadregister.y;
index3+=loadregister.z;
FMAD2(a, b)
FMAD2(a, b)
(cont'd)
```

```
  (cont'd)
  FMAD2(a, b)
  FMAD2(a, b)

  loadregister = dm_src[index];
  index+=loadregister.w;
  a+=loadregister.x;
  b+=loadregister.y;
  index3+=loadregister.z;

  FMAD2(a, b)
  FMAD2(a, b)
  FMAD2(a, b)
  FMAD2(a, b)
 }
 if (threadIdx.x==0 && blockIdx.x==0)
 result[threadIdx.x] = a + b + index + index3;
}
```

Table 18 lists the different variations of the micro-benchmark for various purposes like tuning of the model, showing the memory merging effect etc. The indexing scheme points to the location from where a thread would make the first memory access. Also each thread goes on to make multiple memory accesses due to the load instruction in the 'for' loop, and the step size of subsequent accesses is given by contents of global memory array "dm_src". Manipulation of these two parameters can result in different kinds of memory access patterns.

Table 18: Memory Access Patterns for Vector Memory Operations

operation	indexing scheme	value of float4 global memory array	Access pattern illustration
tuning of departure delay for vector memory operations (32 byte in our case)	((blockidx.x)*((int)blocksize/(int)warpsize)*iters*config*2) + (((int)threadidx.x/(int)warpsize)*iters*config*2)	2	Fig 13
non-overlapping memory accesses	((blockidx.x)*((int)blocksize/(int)warpsize)*iters*config*2) + (((int)threadidx.x/(int)warpsize)*iters*config*2)	2	Fig 13
overlapping memory accesses	((blockidx.x)*((int)blocksize/(int)warpsize)*iters*config*2) + (((int)threadidx.x/(int)warpsize)*2)	2	Fig 14

REFERENCES

[1] http://gpgpu.org/
Date of last access: 02/26/2011

[2] http://www.nvidia.com/object/cuda_home_new.html
` Date of last access: 02/22/2011

[3] http://www.khronos.org/opencl/
Date of last access: 01/31/2011

[4] Richard Vuduc, Aparna Chandramowlishwaran, Jee Whan Choi, Murat Efe Guney, and Aashay Shringarpure. On the limits of GPU acceleration. In Proc. USENIX Wkshp. Hot Topics in Parallelism (HotPar), Berkeley, CA, USA, June 2010

[5] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08)

[6] http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf
Date of last access: 02/29/2011

[7] Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09)

[8] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro 28, 2 (March 2008)

[9] http://developer.nvidia.com/object/visual-profiler.html
Date of last access: 12/28/2010

[10] http://developer.nvidia.com/object/cuda_2_3_downloads.html
Date of last access: 01/18/2011

[11] Gregory Diamos, Andrew Kerr, and Sudhakar Yalamanchili. Gpuocelot: A binary translation framework for ptx., June 2009.
http://code.google.com/p/gpuocelot/

[12] http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/ptx_isa_2.1.pdf
Date of last access: 02/26/2011

[13] DECUDA; https://github.com/laanwj/decuda/wiki
Date of last access: 03/11/2011

[14] CUOBJDUMP; http://developer.nvidia.com/page/home.html
Date of last access: 03/15/2011

[15] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In Proc. IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS), Atlanta, GA, USA, April 2010

[16] H. Cheng, L. Greengard, and V. Rokhlin. 1999. A fast adaptive multipole algorithm in three dimensions. J. Comput. Phys. 155, 2 (November 1999)