Architectural Analysis and Performance Characterization of NVIDIA GPUs using Microbenchmarking

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Saktheesh Subramoniapillai Ajeetha

Graduate Program in Computer Science and Engineering

The Ohio State University

2012

Master's Examination Committee:

Professor P.Sadayappan

Professor Rajiv Ramnath
Abstract

Emergence of new Graphical Processors for general purpose computing presents new challenges for application developers. Graphical Processors vary in terms of number of processor cores per chip, processor speed and memory subsystems. NVIDIA’s CUDA provides a C-like abstraction layer for software developers to implement their applications on GPUs often with little knowledge of the underlying hardware and they are forced to work with high-level descriptions documented by the manufacturer. Substantial knowledge of the hardware architecture will be useful for harvesting the full potential of GPU architectures while trying to solve complex parallel programming problems.

This work reports the measurements and characterization of performance of several NVIDIA GPU’s using micro benchmark analysis. Our thesis uses and adapts the CUDA Micro-benchmarks [8] and SHOC benchmarks [9] to characterize the important aspects of NVIDIA’s GTX200 series GPU- architecture machine (GTX280) and Fermi series - architecture machines (GTX580, Tesla C2050). The investigation is conducted by performing a micro architectural analysis of these machines and comparing their basic performance parameters. This thesis presents an experiment based methodology for characterizing the properties of the arithmetic pipelines. We also measure the global and shared memory latency and bandwidth of these machines and validate the hardware characteristics presented in CUDA programming guide. We hope that the insights from
this work will be useful for improving the analysis and performance optimization of CUDA programs.
Dedication

I dedicate this work to my parents K. Subramoniapillai and I. Ajeetha.
Acknowledgments

Firstly, I would like to thank my thesis advisor, Professor P. Sadayappan, for his thorough guidance and encouragement. I also thank my advisor Professor Rajiv Ramnath for advising me throughout my studies in The Ohio State University. I would like to thank my friends in HPCRL for their technical inputs, especially Justin Holewinski for his continuous guidance and support.

I thank all the faculty and staff in The Department of Computer Science and Engineering for supporting and motivating me during my Masters program.

Finally, I would like to thank my family for their continuous support and providing me with this excellent opportunity.


Vita

January, 1989 ................................. Born – Nagercoil, India

May, 2010 ................................. B.E. Electrical and Electronics Engineering,

                     SSEC, Anna University, Chennai, TN, India

September, 2010 to present ..................... Masters Student, Department of Computer

                     Science and Engineering, The Ohio State University

Fields of Study

Major Field:  Computer Science and Engineering
Table of Contents

Abstract ................................................................................................................................. ii

Dedication .............................................................................................................................. iv

Acknowledgments ................................................................................................................ v

Vita .......................................................................................................................................... vi

Fields of Study ..................................................................................................................... vi

Table of Contents ................................................................................................................. vii

List of Tables ......................................................................................................................... x

List of Figures ....................................................................................................................... xi

Chapter 1: Introduction ....................................................................................................... 1

1.1 Graphical Processing Units .......................................................................................... 1

1.2 Motivation ...................................................................................................................... 2

1.3 Related Work ................................................................................................................ 2

1.4 Road Map ...................................................................................................................... 4

Chapter 2: Background ..................................................................................................... 5
2.1 CUDA Programming Model ................................................................. 5
2.2 CUDA Memory Model ........................................................................ 6
2.3 GPU Micro-benchmark Suite .............................................................. 7
2.4 SHOC Benchmark Suite ..................................................................... 8
2.5 Device Memory Benchmark .................................................................. 9
2.6 Memory Access Patterns ..................................................................... 9

Chapter 3: Overview of GPU Architectures ................................................. 11
3.1 NVIDIA GT200 Architecture .............................................................. 11
3.2 Fermi Architecture ............................................................................ 11
3.3 Comparison of NVIDIA’s GT200 and Fermi Architecture ..................... 14

Chapter 4: Micro-benchmarking Results and Inferences .............................. 15
4.1 Micro-benchmarks Timing.................................................................... 15
4.2 Micro-benchmark Results .................................................................... 16
  4.2.1 Arithmetic Pipelines .................................................................... 16
  4.2.2 Latency Tests .............................................................................. 16
  4.2.3 Throughput Tests ......................................................................... 20
  4.2.4 Latency hiding using occupancy Tests ........................................... 22
4.3 Device Memory Results ....................................................................... 24
  4.3.1 Global Memory Coalesced Read/Write ............................................ 26
4.3.2 Global Memory Un-coalesced Read/Write ...................................................... 29

4.3.3 Effect of L1 cache in Fermi Architecture for Un-coalesced Global Memory Read......................................................................................................................... 31

4.3.4 Shared Memory Read/Write.................................................................................. 33

4.3.5 Increased Shared Memory Write........................................................................... 35

4.4 Shared and Global Memory Latency ........................................................................ 36

Chapter 5: Conclusion and Future Work ......................................................................... 37

5.1 Conclusion................................................................................................................ 37

5.2 Future Work ............................................................................................................. 38

References.................................................................................................................... 39

Appendix A.................................................................................................................... 42
List of Tables

Table 1. Features of Memory in a GPU ................................................................. 7
Table 2. Detailed Architectural features between GT200 and GF100(Fermi) .......... 14
Table 3. Latency of Arithmetic and Logic Operations (int,uint) ......................... 18
Table 4. Latency of Arithmetic Operations (float) ................................................. 18
Table 5. Latency of Arithmetic Operations (double) ............................................ 19
Table 6. Latency of Mathematical Intrinsics ....................................................... 19
Table 7. Throughput of Arithmetic and Logic Operations (int,uint) .................... 21
Table 8. Throughput of Arithmetic Operations (float) ......................................... 21
Table 9. Throughput of Arithmetic Operations double) ...................................... 21
List of Figures

Figure 1. CUDA Physical Memory Layout [from(11)] ......................................................... 6
Figure 2. Memory Requests from a Single Warp (a) Coalesced Memory Access (b) Un-coalesced Memory Access (from [6]) .................................................................................. 10
Figure 3. Single Streaming Processor of GT200 architecture ........................................... 11
Figure 4. Single Streaming Multiprocessor of Fermi Architecture (from [12]) ............... 13
Figure 5. Timing Code for Micro-benchmarking ................................................................. 15
Figure 6. Throughput and Latency for GTX 280 for dependent ADD instructions ....... 22
Figure 7. Throughput and Latency for GTX 580 for dependent ADD instructions ....... 23
Figure 8. Throughput and Latency for Tesla C2050 for dependent ADD instructions ... 24
Figure 9. Global Memory Bandwidth of GTX 280 ............................................................. 28
Figure 10. Global Memory Bandwidth of GTX 580 for Coalesced Read/Write ............. 28
Figure 11. Global Memory Bandwidth of Tesla C2050 for Coalesced Read/Write ....... 29
Figure 12. Global Memory Bandwidth of GTX 280 for Un-coalesced Read/Write ...... 30
Figure 13. Global Memory Bandwidth of GTX 580 for Un-coalesced Read/Write ....... 30
Figure 14. Global Memory bandwidth of Tesla C 2050 for Un-coalesced Read/Write ... 31
Figure 15. Effect of L1 cache disabled for Global Memory Un-coalesced Read in GTX 580 ............................................................................................................................ 32
Figure 16. Effect of L1 cache disabled for Global Memory Un-coalesced Read in Tesla C2050 ................................................................. 33

Figure 17. Shared Memory Bandwidth of GTX 280 for Read/Write ......................... 34

Figure 18. Shared Memory bandwidth of GTX 580 for Read/Write ......................... 34

Figure 19. Shared Memory bandwidth of Tesla C2050 for Read/Write ................... 35

Figure 20. Improved Shared Memory Write bandwidth for GTX 580 ....................... 36
Chapter 1: Introduction

1.1 Graphical Processing Units

With the development of more massively data parallel algorithms for scientific applications, Graphical Processing Units (GPUs) are now being used for general-purpose computing applications and hence these GPUs are widely used to build supercomputers. The numerous amounts of parallel processing cores, with high floating-point operation rates and high performance memory bandwidth has led the roadmap of these devices to accelerate various high performance computing applications. These GPUs are predominantly designed and developed by NVIDIA and ATI that are currently used in several supercomputers. For example, in the Top500 supercomputer list released in June 2012, the world’s fifth fastest supercomputer Tianhe-1A uses NVIDIA Tesla 2050 [1].

These GPU devices are commonly programmed in low-level by either Open Computing Language (OpenCL) or Compute Unified Device Architecture (CUDA) allowing these GPUs accessible to programmers, to exploit the highly parallel nature of the accelerator to be utilized for performing parallel computations. OpenCL is a new royalty-free cross-platform API created by Khronos Group focusing precisely on the issue of portability across GPUs manufactured by different vendors [3] whereas CUDA
enables a programmer to access only NVIDIA GPUs that are required for GPU programming with graphics APIs such as OpenGL and DirectX [2].

1.2 Motivation

Optimizing programs for these GPUs requires in depth knowledge about the architectural features of the computing platform. This information is mostly unavailable in the technical specifications documentation of these GPUs. Benchmarking can quantify the performance of these accelerators thus enabling us to closely look at the various architectural features and thus will aid application developers to optimize them to specific devices. Micro-benchmarking attempts to measure the performance of a small bit of code. There has been relatively little analysis in exploring the architectural features of GT200 and Fermi NVIDIA GPUs.

1.3 Related Work

Ryan et al. [4] presents a micro-benchmark suite for AMD GPUs that supports the AMD StreamSDK by identifying a series of architectural features and basic program characteristics that include vectorization, burst write latency, texture fetch latency, global read and write latency, ALU/Fetch operation ratio, domain size and register usage for AMD GPUs.
Ying Zhang et al. [5] conduct a comprehensive study to characterize the architectural differences between Nvidia’s Fermi and ATI’s Cypress architectures and demonstrate their impact on performance compare the energy efficiencies of the above two platforms.

Sunpyo Hong et al. [6] introduce an analytical model with memory-level and thread-level parallelism awareness to investigate the GPU performance by estimating the execution time of massively parallel programs on NVIDIA’s 8800GTX, Quadro FX5600, 8800GT, GTX280.

Yao Zhang et al. [7] uses a micro-benchmark approach to analyze the GPU performance to develop a throughput model for execution time, instruction pipeline, shared memory access, and global memory access for NVIDIA GeForce 200 series.

Henry Wong et al. [8] presents a micro-benchmark suite for NVIDIA GT200 to reveal the undisclosed internal characteristics of processing elements and the memory hierarchies by exposing some of the undocumented features that impact program performance and correctness.

This thesis work focuses on micro-benchmarking of NVIDIA GT280, NVIDIA GTX580 and NVIDIA Tesla C2050 by further extending the work of Henry Wong et al [8] to reveal the architectural features of these GPUs by focusing on latency and throughput of various arithmetic operations, shared memory access and global memory access. Also, low level SHOC benchmarks [9] are used to determine the various memory bandwidths of these GPUs.
1.4 Road Map

In Chapter one, we introduce the importance of Graphical processing units in today’s world and also outline the motivation behind our work and the concerned related works to briefly outline the research questions we try to address.

In Chapter two, we provide a detailed background on CUDA programming model, memory hierarchy and memory access patterns. We also discuss about the benchmarks used in this study for micro architectural analysis.

In Chapter three, we compare and contrast the NVIDIA GTX200 and NVIDIA Fermi Hardware architectures in detail using the NVIDIA documentation [10]

In Chapter four, we discuss our experimental methodology and setup, results and analysis of our work.

In Chapter five, we focus on the contribution of this thesis to the GPGPU community and the scope for future work.
Chapter 2: Background

2.1 CUDA Programming Model

Compute Unified Device Architecture (CUDA) is a parallel development framework model by NVIDIA to be used on NVIDIA GPUs. The application is executed in a device by launching a kernel function that runs on a device scaling across parallel multiprocessors. Many threads execute each kernel. Array of threads execute a kernel using the same code. Each thread is identified by its ID. A group of threads form a block to utilize the on chip shared memory in a SM. These blocks can be 1, 2 or 3 dimensional groups of threads. These blocks are grouped into grids of blocks [10].

A kernel is defined using the __global__ declaration specifier.

```
__global__ kernel <<< gridDim, blockDim >> (parameters);
```

The block dimension is the total number of threads within each block and grid dimension is the total number of blocks within each grid. Threads within a block can synchronize whereas threads in different blocks cannot synchronize.

In present NVIDIA GPUs, a thread block may contain up to 1536 threads. A single kernel can be executed by multiple equally-shaped thread blocks. Hence the total
number of threads is equal to the number of threads per block times the number of blocks [10].

On a kernel call, blocks of threads gets scheduled to execute on any SM. Threads within a block are subdivided into individual warps containing 32 threads that are processed in parallel in a Single Instruction Multiple Thread (SIMT) fashion.

### 2.2 CUDA Memory Model

![CUDA Physical Memory Layout](image)

Figure 1. CUDA Physical Memory Layout [from(11)]

CUDA enabled devices consists of six different types of memory as shown in [Table 1]; Register, Local, Shared, Global, Constant and Texture memory. Global Memory (device memory) is accessible across all Streaming Multiprocessors (SM) with Read/Write privilege. Constant and Texture memory is accessible across all SMs but only
with read privilege. Local memory is used for register spilling and is accessible by only one thread. Each SM contains a Shared Memory and it is accessible by all threads within a same block. Register memory in a SM is accessible by each thread and it is local to each thread.

Table 1. Features of Memory in a GPU (from[11])

<table>
<thead>
<tr>
<th>S.No</th>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Register</td>
<td>On-chip</td>
<td>N/A</td>
<td>R/W</td>
<td>One thread</td>
<td>Thread</td>
</tr>
<tr>
<td>2.</td>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>R/W</td>
<td>One thread</td>
<td>Thread</td>
</tr>
<tr>
<td>3.</td>
<td>Shared</td>
<td>On-chip</td>
<td>N/A</td>
<td>R/W</td>
<td>All threads in a block</td>
<td>Block</td>
</tr>
<tr>
<td>4.</td>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>R/W</td>
<td>All threads + host</td>
<td>Application</td>
</tr>
<tr>
<td>5.</td>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Application</td>
</tr>
<tr>
<td>6.</td>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Application</td>
</tr>
</tbody>
</table>

2.3 GPU Micro-benchmark Suite

A micro-benchmark is usually a GPU kernel code which is instrumented with timing code around the kernel coding section that exercises the hardware being used. To explore the architectures of NVIDIA GPUs, we use and in some cases extend the micro-benchmarks described in the work of Henry Wong et al. [8] to expose the interesting characteristics we intend to measure across several NVIDIA GPUs. The purpose of micro-benchmarks is to measure the properties of some fundamental operations that are the basis of higher level system behavior. The measured performance in these benchmarks will aid in better understanding of the GPU performance in higher level
benchmarks such as application programs. We analyze and verify the execution times of these benchmarks and draw our conclusions about the architectural characteristics of the systems.

2.4 SHOC Benchmark Suite

Scalable Heterogeneous Operating Computing Suite (SHOC) is a huge spectrum of programs that test the performance and stability of scalable heterogeneous computing systems that uses the low level parallel programming models like CUDA and OpenCL. The lower level SHOC suite uses micro-benchmarks to assess architectural features of a system whereas at higher levels, it uses application kernels to determine system performance including intra-node and inter-node communication among devices [9].

The SHOC benchmark is organized as Level 0, 1 and 2. Level 1 benchmarks measure basic parallel algorithms, such as FFT, Reduction, Scan, SGEMM, Radix Sort, Stencil 2D and Triad. Level 2 benchmarks measures real application kernels.

Level Zero tests are designed to measure low level hardware characteristics is being used in this thesis work. Bus Speed Download and Read back benchmark measure the bandwidth of the interconnection bus between the host processor and the GPU device by repeatedly transferring data of various sizes to and from the device. The Peak FLOPS benchmark measures peak floating point (single or double precision) operations per second using a workload designed to fully exercise device functional units. Device Memory benchmark measures bandwidth for all device memory address spaces,
including global, local, constant, and texture memories. Moreover each benchmark in SHOC suite contains three versions namely Serial, Embarrassingly Parallel (EP) and True Parallel (TP). Serial version is used on a single node. EP benchmarks are used on multiple nodes of a cluster when there is no communication between devices or nodes whereas TP benchmarks are used with one or more devices per node including communication between devices.

In this thesis we use the Level 0, CUDA Serial version of the Device Memory benchmark to measure the bandwidth of read and write for Global Memory and Shared Memory.

### 2.5 Device Memory Benchmark

This benchmarks measures bandwidth for all device memory address spaces, including global, shared, constant, and image memories [9]. The global address space is benchmarked using both coalesced and un-coalesced memory accesses. In all cases, each thread generates 16 independent memory requests.

### 2.6 Memory Access Patterns

Global memory bandwidth can be maximized by minimizing the number of bus transactions. This can be achieved by coalesced memory access. When memory transactions are per half-warp (16 threads) reading a contiguous segment of data per
access, it is called coalesced memory access. Generally a thread loads/stores 1, 2, 4, 8, 16
bytes per access. In devices with compute capability 1.2 and higher, individual threads
can access independent addresses [10]. Coalesced memory access can be achieved either
by using nearby addresses for threads in a warp or by using unit stride accesses. On the
other hand non coalesced memory access gives low global memory bandwidth as the
number of bus transactions is higher when stride size is increased i.e. the warp cannot be
executed until all memory transactions from the same warp are serviced [6].

Figure 2. Memory Requests from a Single Warp (a) Coalesced Memory Access (b) Un-
coalesced Memory Access (from [6])
Chapter 3: Overview of GPU Architectures

3.1 NVIDIA GT200 Architecture

GT200 series has 30 Streaming Multiprocessors (SMs) organized as 3 SMs per Thread Processing Cluster (TPC). Each SM contains 8 Scalar Processors (SPs), 2 Special Function Units (SFUs), 1 Double Precision Units (DPU) and 16,384 32 bit registers along with one warp scheduler. The integer ALU is limited to 24-bit precision for multiply operations; as a result, multi-instruction emulation sequences are required for integer arithmetic [13].

Figure 3. Single Streaming Processor of GT200 architecture
3.2 Fermi Architecture

Fermi Architecture is a name given to a set of advanced GPUs manufactured by NVIDIA. The two GPUs, GTX 580 and Tesla C2050 included in this study uses the Fermi Architecture and they are generally named as GF100. Fermi architecture is derived from previous generation G80 and GT200. GF100 series has 16 Streaming Multiprocessors (SMs). Each SM contains 32 CUDA cores/Scalar Processors, 4 Special Function Units, 16 Double Precision Units and 32,768 32 bit registers. The 16 DPUs provides peak performance for double precision operations. There are 2 warp schedulers per SM alternatively when compared to 1 warp scheduler in GT200 series. Each Scalar Processor contains a fully pipelined integer ALU that supports full 32-bit precision for all instructions and a floating point unit (FPU). Also the 4 SFU units are capable of executing transcendental mathematical operations. The 16 load/store units in each SM, allows source and destination addresses to be calculated for sixteen threads per clock [12]. Moreover, each SM in the Fermi architecture has 48KB, 16KB configurable shared memory and 48KB, 16KB configurable L1 cache. The configurable shared memory allows speeding up applications with carefully planned resource analysis. The memory subsystem comes with a configurable L1 and Unified L2 cache. It also has six 64-bit memory partitions, for a 384-bit memory interface, supporting up to a total of 6 GB of GDDR5 DRAM memory [12] [14] [15].
Figure 4. Single Streaming Multiprocessor of Fermi Architecture (from [12])
### 3.3 Comparison of NVIDIA’s GT200 and Fermi Architecture

This section provides with a thorough comparison of architecture parameters of NVIDIA’s Fermi vs GT 200 as documented by NVIDIA [12] [14] [15]. The detailed comparison can be found in Table 2.

#### Table 2. Detailed Architectural features between GT200 and GF100(Fermi)

<table>
<thead>
<tr>
<th>General GPU Organization</th>
<th>GT 200</th>
<th>GF 100(Fermi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCs(Processing Clusters)</td>
<td>10 TPC</td>
<td>4 GPC</td>
</tr>
<tr>
<td>SMs(Streaming Multiprocessors)</td>
<td>30</td>
<td>16</td>
</tr>
<tr>
<td>Number of CUDA cores</td>
<td>240</td>
<td>512</td>
</tr>
<tr>
<td>Memory</td>
<td>1GB DDR3, 64-bit</td>
<td>up to 6 GB DDR5, 384-bit</td>
</tr>
<tr>
<td>Resources in each SM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPs(Scalar Processor)</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>SFUs(Special Function Unit)</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>DPUs(Double Precision Unit)</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>32 bit Registers</td>
<td>16,384</td>
<td>32,768</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>16 KB</td>
<td>48 KB/16 KB configurable</td>
</tr>
<tr>
<td>L1 cache</td>
<td>None</td>
<td>16 KB/48 KB configurable</td>
</tr>
<tr>
<td>L2 cache</td>
<td>None</td>
<td>768 KB</td>
</tr>
<tr>
<td>Warp Schedulers</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>General Information</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute Capability</td>
<td>1.3</td>
<td>2.0</td>
</tr>
<tr>
<td>ECC Support</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Concurrent kernels</td>
<td>No</td>
<td>Yes, up to 16</td>
</tr>
<tr>
<td>Warps</td>
<td>32 threads</td>
<td>32 threads</td>
</tr>
<tr>
<td>Blocks</td>
<td>1024 threads</td>
<td>1536 threads</td>
</tr>
<tr>
<td>Constant Memory</td>
<td>64KB total</td>
<td>64KB total</td>
</tr>
<tr>
<td>Kernel Size</td>
<td>2M PTX insns max</td>
<td>512M PTX insns max</td>
</tr>
<tr>
<td>Operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double Precision Floating Point</td>
<td>30 FMA ops/clock</td>
<td>256 FMA ops/clock</td>
</tr>
<tr>
<td>Single Precision Floating Point</td>
<td>240 MAD ops/clock</td>
<td>512 FMA ops/clock</td>
</tr>
</tbody>
</table>
Chapter 4: Micro-benchmarking Results and Inferences

4.1 Micro-benchmarks Timing

The timing for these micro-benchmarks [8] is measured by using the clock () function. The benchmark code is run through the code block twice and the first iteration of the loop is omitted to eliminate the effects of cold instruction cache misses. Initially we store the clock in the register. At the end of the kernel code, the clock value stored in the register is finally transferred to the global memory to prevent the incorrect timing measurements due to the global memory accesses. The figure below shows the timing code inserted around the code and the value returned by the clock () function is stored in the register.

```
__global__ void global_latency (unsigned int *my_array, int array_length, int iterations)
{
    unsigned int start_time, end_time;
    unsigned int *j = (unsigned int*)my_array;
    volatile unsigned long long sum_time;
    sum_time = 0;
    duration[0] = 0;

    for (int k = -ignore_iterations; k < iterations; k++) {
        if (k%6) {
            sum_time = 0; // ignore some iterations: cold i-cache misses
        }
        start_time = clock();
        repeat256(j=(unsigned int**)&j);
        end_time = clock();
        sum_time += (end_time - start_time);
    }
}
```

Figure 5. Timing Code for Micro-benchmarking
4.2 Micro-benchmark Results

4.2.1 Arithmetic Pipelines

The SMs in each of the GPU chips GTX 280, GTX 580 and Tesla C2050 contains three different types of execution units as shown in Table 2. The Tables 3,4 and 5 shows the latency of the arithmetic operations including the execution unit for the different GPU architectures. The Table 6 shows the latency of mathematical intrinsic for different GPU architectures which primarily use SFUs as the execution unit. The program stores the operands used in these operations in the registers of each of the SMs.

4.2.2 Latency Tests

For the latency test, we launch a kernel with a block of one thread for the entire GPU. The latency test consists of a timing using our clock function, a chain of dependent operations in an unrolled loop. The Tables 3,4,5 and 6 show the average time per instruction in cycles. For example the ADD instruction takes 24 cycles to produce the result in GTX280. This latency can be hidden by launching a kernel with 192 threads which is a total of 6 warps. Single precision 32 bit MUL in GTX 580 maps to a single instruction requiring 18 cycles while the same instruction in GTX 280 takes 96 cycles since it is executed as 4 dependent instruction requiring $24 \times 4 = 96$ cycles. The 32 bit integer mad is executed as 5 dependent instructions requiring 120 cycles in GTX 280.
while in just takes a single instruction in GTX 580 requiring 20 cycles. The latency of operations is less in Fermi architecture when compared to GT200 series GPUs.

We see that the 32 bit integer and double precision integer division is calculated by executing a subroutine call and consequently we see that the latency is too high and the throughput is too low. The table also shows the execution unit(s) which get exercised while performing an operation.

The Mathematics intrinsic functions like __umulhi() and mulhi() instructions have a very high latency in GTX280(Table 6) because they translate to 6 dependent instructions while the same instructions take less than 20 cycles in Fermi series processors (GTX 580, Tesla C2050) suggesting that they have been implemented as native instructions in these processors. Sqrt() is translated to two instructions : 1. Reciprocal-sqrt and 2. Reciprocal and thus takes two times the latency of these individual instructions.
Table 3. Latency of Arithmetic and Logic Operations (int,uint)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
<th>Execution Unit</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GTX 280, GTX 580, Tesla C 2050</td>
<td></td>
</tr>
<tr>
<td>add, sub</td>
<td>uint : int</td>
<td>SP, SP</td>
<td>24, 18, 20</td>
</tr>
<tr>
<td>max, min</td>
<td>uint, int</td>
<td>SP, SP</td>
<td>48, 20, 20</td>
</tr>
<tr>
<td>Mad</td>
<td>uint</td>
<td>SP</td>
<td>122, 20, 20</td>
</tr>
<tr>
<td>Mul</td>
<td>int</td>
<td>SP</td>
<td>120, 20, 20</td>
</tr>
<tr>
<td>Div</td>
<td>uint, int</td>
<td>SP, SP</td>
<td>96, 18, 18</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>SP, SFU</td>
<td>608, 264, 264</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td></td>
<td>684, 300, 300</td>
</tr>
<tr>
<td>Rem</td>
<td>uint</td>
<td>SP</td>
<td>728, 264, 264</td>
</tr>
<tr>
<td>Rem</td>
<td>int</td>
<td>SP</td>
<td>784, 297, 297</td>
</tr>
<tr>
<td>and, or, xor, shl, shr</td>
<td>uint</td>
<td>SP, SP</td>
<td>24, 18, 18</td>
</tr>
<tr>
<td>Abs</td>
<td>int</td>
<td>SP</td>
<td>42, 36, 36</td>
</tr>
</tbody>
</table>

Table 4. Latency of Arithmetic Operations (float)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
<th>Execution Unit</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GTX 280, GTX 580, Tesla C 2050</td>
<td></td>
</tr>
<tr>
<td>add, sub</td>
<td>float</td>
<td>SP</td>
<td>24, 18, 18</td>
</tr>
<tr>
<td>max, min</td>
<td>float</td>
<td>SP</td>
<td>48, 36, 36</td>
</tr>
<tr>
<td>Mad</td>
<td>float</td>
<td>SP, SP</td>
<td>24, 20, 20</td>
</tr>
<tr>
<td>Mul</td>
<td>float</td>
<td>SP, SFU</td>
<td>24, 18, 18</td>
</tr>
<tr>
<td>Div</td>
<td>float</td>
<td></td>
<td>129, 101, 88</td>
</tr>
</tbody>
</table>
Table 5. Latency of Arithmetic Operations (double)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
<th>Execution Unit GTX 280, Tesla C 2050</th>
<th>Latency GTX 280</th>
<th>GTX 580</th>
<th>Tesla C 2050</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, sub</td>
<td>double</td>
<td>DPU</td>
<td>24</td>
<td>24</td>
<td>22</td>
</tr>
<tr>
<td>max, min</td>
<td>double</td>
<td>DPU</td>
<td>66</td>
<td>88</td>
<td>84</td>
</tr>
<tr>
<td>Mad</td>
<td>double</td>
<td>DPU</td>
<td>24</td>
<td>24</td>
<td>22</td>
</tr>
<tr>
<td>Mul</td>
<td>double</td>
<td>DPU</td>
<td>24</td>
<td>24</td>
<td>22</td>
</tr>
<tr>
<td>Div</td>
<td>double</td>
<td>DPU</td>
<td>131</td>
<td>106</td>
<td>92</td>
</tr>
</tbody>
</table>

Table 6. Latency of Mathematical Intrinsics

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
<th>Execution Unit GTX 280, Tesla C 2050</th>
<th>Latency GTX 280</th>
<th>GTX 580</th>
<th>Tesla C 2050</th>
</tr>
</thead>
<tbody>
<tr>
<td>__umul24()</td>
<td>uint</td>
<td>SP</td>
<td>24</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>__mul24()</td>
<td>int</td>
<td>SP</td>
<td>24</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>__usad()</td>
<td>uint</td>
<td>SP</td>
<td>24</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>__sad()</td>
<td>int</td>
<td>SP</td>
<td>24</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>__umulhi()</td>
<td>uint</td>
<td>SP</td>
<td>144</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>__mulhi()</td>
<td>int</td>
<td>SP</td>
<td>173</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>__fadd_rn(), __fadd_rz()</td>
<td>float</td>
<td>SP</td>
<td>24</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>__fmul_rn(), __fmulrz()</td>
<td>float</td>
<td>SP,SFU</td>
<td>26</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>__fdividef()</td>
<td>float</td>
<td>SP</td>
<td>52</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>__sinf(), __cosf()</td>
<td>float</td>
<td>SFU</td>
<td>48</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>__tanf()</td>
<td>float</td>
<td>SP</td>
<td>98</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>__exp2f()</td>
<td>float</td>
<td>SFU</td>
<td>48</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>__expf(), __exp10f()</td>
<td>float</td>
<td>SP</td>
<td>72</td>
<td>58</td>
<td>58</td>
</tr>
<tr>
<td>__log2f()</td>
<td>float</td>
<td>SFU</td>
<td>28</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>__logf(), __log10f()</td>
<td>float</td>
<td>SFU</td>
<td>52</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>__powf()</td>
<td>float</td>
<td>SP</td>
<td>75</td>
<td>58</td>
<td>58</td>
</tr>
<tr>
<td>rsqrt()</td>
<td>float</td>
<td>SFU</td>
<td>28</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>sqrt()</td>
<td>float</td>
<td>SFU</td>
<td>56</td>
<td>44</td>
<td>44</td>
</tr>
</tbody>
</table>
4.2.3 Throughput Tests

To measure the throughput, a kernel is launched with a block of 512 threads for GTX 280 and a block of 1536 threads for GTX 580 and Tesla C2050 so that we reach the maximum occupancy limit of each of the SMs in the GPUs we investigate. Single precision floating point MUL takes 11.2 ops/clock while the scalar processor throughput is 8 cycles. This shows that the MUL is executed by both the SP and SFU. This result verifies the results produced by Wong paper on micro benchmarking the GTX 280 series processors. We don’t observe this increase in GTX 580 and Tesla C2050 series processors which implies that the MUL instruction is just issued to the SP and the SFU’s are not used. The single precision MAD operation has a throughput of 7.9 ops/clock and it cannot be executed by the SFU and verifies the result of Henry Wong et al [8]. We also find the same behavior in Fermi architectures suggesting that the SFU unit in this architecture is also not capable of executing the single precision floating point MAD operation which has the throughput of 31.5 ops/clock. The results for throughput of arithmetic and logic and mathematical intrinsic functions for the three GPUs used in our study are put in Table 7, 8 and 9.
### Table 7. Throughput of Arithmetic and Logic Operations (int,uint)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
<th>Throughput (ops/clock)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GTX 280</td>
</tr>
<tr>
<td>add,sub</td>
<td>uint,int</td>
<td>7.9</td>
</tr>
<tr>
<td>max,min</td>
<td>uint,int</td>
<td>4.0</td>
</tr>
<tr>
<td>mad</td>
<td>uint</td>
<td>1.4</td>
</tr>
<tr>
<td>mad</td>
<td>int</td>
<td>1.4</td>
</tr>
<tr>
<td>mul</td>
<td>uint,int</td>
<td>1.7</td>
</tr>
<tr>
<td>div</td>
<td>uint</td>
<td>0.3</td>
</tr>
<tr>
<td>div</td>
<td>int</td>
<td>0.2</td>
</tr>
<tr>
<td>rem</td>
<td>uint</td>
<td>0.2</td>
</tr>
<tr>
<td>rem</td>
<td>int</td>
<td>0.2</td>
</tr>
<tr>
<td>and,or,xor,shr</td>
<td>uint</td>
<td>7.9</td>
</tr>
<tr>
<td>abs</td>
<td>int</td>
<td>3.9</td>
</tr>
</tbody>
</table>

### Table 8. Throughput of Arithmetic Operations (float)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
<th>Throughput (ops/clock)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GTX 280</td>
</tr>
<tr>
<td>add,sub</td>
<td>float</td>
<td>7.9</td>
</tr>
<tr>
<td>max,min</td>
<td>float</td>
<td>3.9</td>
</tr>
<tr>
<td>mad</td>
<td>float</td>
<td>7.9</td>
</tr>
<tr>
<td>mul</td>
<td>float</td>
<td>11.2</td>
</tr>
<tr>
<td>div</td>
<td>float</td>
<td>1.6</td>
</tr>
</tbody>
</table>

### Table 9. Throughput of Arithmetic Operations double)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
<th>Throughput (ops/clock)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GTX 280</td>
</tr>
<tr>
<td>add,sub</td>
<td>Double</td>
<td>7.9</td>
</tr>
<tr>
<td>max,min</td>
<td>Double</td>
<td>2.7</td>
</tr>
<tr>
<td>mad</td>
<td>double</td>
<td>7.9</td>
</tr>
<tr>
<td>mul</td>
<td>double</td>
<td>11.2</td>
</tr>
<tr>
<td>div</td>
<td>double</td>
<td>1.7</td>
</tr>
</tbody>
</table>
4.2.4 Latency hiding using occupancy Tests

A series of kernels is launched sequentially one at a time with increasing number of warps in each of the kernels. The kernels were launched ranging from a single block of 32 threads (1 warp) to a block of 512 threads (16 warps) for GTX 280 GPU and a block of 1024 threads (32 warps) for GTX 580 and Tesla C2050 GPUs so that a block of these threads exercise the SM which processes these kernels.

The latency and throughput of a series of dependent ADD instructions is plotted in the following figure for GTX 280. The latency is 24 cycles below 6 concurrent warps. It is also observed that the throughput increases linearly while the pipeline is still is not full and then saturates at eight warps which shows that the entire eight SPs in GTX280 are saturated and our results verify the results produced by Henry Wong et al. [8].

![Figure 6. Throughput and Latency for GTX 280 for dependent ADD instructions](image-url)
The latency and throughput of a series of dependent ADD instructions is plotted in the following figure for GTX 580. The latency is 18 cycles below 16 concurrent wraps. We also observe that the throughput increases linearly while the pipeline is still not full and then saturates at 16 warps which shows that all the 32 cores in GTX580 are saturated.

![Figure 7. Throughput and Latency for GTX 580 for dependent ADD instructions](image)

The latency and throughput of a series of dependent ADD instructions is plotted in the following figure for Tesla C2050. The latency is 18 cycles below 6 concurrent wraps. We also observe that the throughput increases linearly while the pipeline is still not full and then saturates at 6 warps which shows that the SM is saturated.
4.3 Device Memory Results

SHOC benchmark suite as described in Chapter 2 is used to explore the bandwidth characteristics of the GPU’s we try to explore. We use Level Zero tests in SHOC suite to assess the architectural features of the system. Data access latency and bandwidth performance in modern GPUs can significantly affect program performance when off-chip resources are frequently accessed. The applications which are sensitive to data access performance requires some tuning to avoid remote accesses as far as possible. The bandwidth of a data-access operation is the number of bytes of data that are transferred between a device’s memory and the requesting processor in a given amount of time.
In our thesis we are also exploring the bandwidth of the global memory and on-chip shared memory for GTX280, GTX580 and Tesla C2050.

Launch Configuration:

The following are the artificial kernel design considerations for performing memory bandwidth tests.

1. We need enough threads to keep the GPU busy. Typically we would like 512+ threads per SM and have enough concurrent accesses to saturate the bus and launch enough threads to maximize throughput and also process several elements per thread so that multiple loads get pipelined.

2. The thread block size should be a multiple of warp size (32). Fermi architecture SM can concurrently execute up to 8 thread blocks. Really small thread blocks prevent achieving good occupancy. Really large thread blocks are less flexible.

3. We should strive for perfect coalescing per warp which means a warp should access within a contiguous region.

SHOC Level Zero Kernel Launch Configuration:

The artificial kernels in Level Zero device memory test launch a total of 32768 threads in the following configurations:

1. 64 blocks of 512 threads
2. 128 blocks of 256 threads
3. 256 blocks of 128 threads
4. 512 blocks of 64 threads
5. 1024 blocks of 32 threads

For each of these configurations a series of tests exercising the global and shared memory is designed as follows:

1. Global memory read coalesced
2. Global memory write coalesced
3. Global memory read un-coalesced
4. Global memory write un-coalesced
5. Shared memory read
6. Shared memory write

**Bandwidth Tests**

Global memory resides in the device memory and the device memory is accessed via 32-, 64 or 128 byte transactions. These memory transactions must be naturally aligned for coalesced memory accesses.

**4.3.1 Global Memory Coalesced Read/Write**

When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads.

The advertised theoretical bandwidth in the CUDA programming guide for the GPU’s are as follows:
Theoretical bandwidth for GTX 280  = memory clock rate * memory interface
= 1107 x 10^6 x (512/8) x 2 / 10^9
= 141.6 GB/sec

Theoretical bandwidth for GTX 580  = 2004 x 10^6 x (384/8) x 2 / 10^9
= 192.38 GB/sec

Theoretical bandwidth for TeslaC2050 = 1500 x 10^6 x (384/8) x 2 / 10^9
= 144 GB/sec

In practice the observed memory bandwidth will be less than (and is guaranteed not to exceed) the theoretical bandwidth. The Figures 9,10 and 11 show that the measured bandwidth for both coalesced read and writes operation for a series of artificial configurations we launched which transfers 128 MB of data from the global memory to the registers during the read operation and writes back the 128 MB of data to the global memory during the write operation. We see that we are able to achieve 80 – 90 GB/s for different threads block sizes for GTX 280 and 140 GB/s for GTX 580 and 80 - 100 GB/s for C2050. These measurements are below the theoretical maximum as expected. .
Figure 9. Global Memory Bandwidth of GTX 280

Figure 10. Global Memory Bandwidth of GTX 580 for Coalesced Read/Write
4.3.2 Global Memory Un-coalesced Read/Write

The Figures 12,13 and 14 shows the plot of measured bandwidth for a series of kernel launches with different block sizes copying 128 MB of data from device memory to registers in SM in an un-coalesced access pattern for GTX280, GTX 580 and Tesla C2050 respectively. As expected the measurements show a significant decrease in the bandwidth measured. We see a decrease of the order of 10x times while accessing in an un-coalesced fashion. Even on Tesla2 or Fermi class hardware, failing to coalesce global memory transactions can result in a 10x performance hit.
Figure 12. Global Memory Bandwidth of GTX 280 for Un-coalesced Read/Write

Figure 13. Global Memory Bandwidth of GTX 580 for Un-coalesced Read/Write
4.3.3 Effect of L1 cache in Fermi Architecture for Un-coalesced Global Memory

Read

It is important to study about the effect of L1 cache in Fermi Architecture for un-coalesced read memory accesses. By default when L1 cache is enabled, hardware issues 128 bytes of segment transactions, whereas if L1 cache is disabled, hardware issues 32 bytes of segment transactions [16]. Smaller contiguous region (32B) of segment improves the speedup of transfer for un-coalesced and sparse access thus increasing the global memory read bandwidth.

The nvcc compilation flag -Xptxas -dlcm=eg was used to disable L1 cache and the device memory test was rerun to examine the results for global memory un-coalesced...
read accesses. For GTX 580, there was a marginal increase in bandwidth from 12 GB/s to 18GB/s in bandwidth for the initial kernel configuration (1024 blocks, 32 block size) whereas for other kernel configurations there was no significant difference.

Figure 15. Effect of L1 cache disabled for Global Memory Un-coalesced Read in GTX 580
Figure 16. Effect of L1 cache disabled for Global Memory Un-coalesced Read in Tesla C2050

4.3.4 Shared Memory Read/Write

The on-chip shared memory is hundreds of times faster than the global memory. The threads of a block can cooperate via the on-chip shared memory per SM. The kernel in our program allocates 8KB of shared memory per block of thread. The data transfer actually involves transferring data from the global memory to the shared memory and then from shared memory to registers. The Figure.15 shows a peak bandwidth of 250 GB/s for GTX 280 and a peak of ~700 GB/s is measured for GTX 580 since the global memory to shared memory access latency is also involved in the transfer. The Figure.16, Figure.17 shows the measured shared memory bandwidth for GTX 580 and Tesla C2050 respectively. The shared memory bandwidth is not as high as expected because the benchmarks we use take into account the global memory to shared memory access
latency too while transferring data to the registers. The improved shared memory write bandwidth for GTX 580 is explained in Section 4.3.5.

Figure 17. Shared Memory Bandwidth of GTX 280 for Read/Write

Figure 18. Shared Memory bandwidth of GTX 580 for Read/Write
4.3.5 Increased Shared Memory Write Bandwidth

As described in Section 4.3.3, the default shared memory write in SHOC benchmark suite [9], writes back the data to global memory from shared memory which is not necessary. The device memory source code was modified for Shared Memory Write. There was a 50% increase of shared memory write bandwidth when this part of the code was removed for GTX 580. However there was no significant increase in Shared Memory write bandwidth for GTX 280 and Tesla C2050.
Figure 20. Improved Shared Memory Write bandwidth for GTX 580

4.4 Shared and Global Memory Latency

All running threads from different blocks can access the global memory. The NVIDIA documentation states that the global memory latency of GTX 280 is in the range of 400 – 600 cycles and Fermi series GPUs have an access latency of 400 – 800 cycles. The benchmark executes a sequence of pointer chasing dependent reads to the global memory and we measure the read latency of 440 cycles for GTX 280 and 425 cycles for Fermi series machines. Shared memory read latency is also measured and we get 40 - 45 cycles for GTX 280 and 30 -35 cycles for Fermi architectures. The results for GTX 280 coincide with the results by Henry Wong et al [8].
Chapter 5: Conclusion and Future Work

5.1 Conclusion

This thesis was motivated by a similar micro-benchmark analysis work carried out by Henry Wong et al.[8]. We have aimed at analyzing the performance characteristics of between GTX280 and Fermi architecture GPUs (GTX 580, Tesla C2050) and compare the performance of each of these architectures based on the chosen performance metrics. We were able to expose the architectural details of processors and memory characteristics in detail.

At the outset of this thesis two research questions were posed. First, is it possible to reverse engineer often undisclosed details of a complex device like the NVIDIA GPU. Second, is it possible to help a CUDA programmer understand the differences in GPU architecture in this detail and determine the performance and verify accuracy in comparison to what is stated in the NVIDIA documentation, which would enable him to make informed decisions when choosing the right hardware for the problems the programmer sets out to solve. Therefore, this thesis contributes by examining and evaluating the behavior of GT-200 and Fermi architecture GPUs using micro-benchmarking techniques.
The thesis work demonstrates that the GPU’s does not introduce speedup unless the following parameters are carefully handled.

1. Reducing off-chip memory access
2. Maximizing data parallelism and instruction level parallelism
3. Distribute the work load efficiently on the GPU using the optimal execution configuration to exercise all the hardware resources

5.2 Future Work

Future work will involve measuring the cache parameters of the GPU’s under investigation and deducing the cache characteristics from latency plots. Another interesting future work would be to investigate the micro architectural characteristics of NVIDIA’s new generation of CUDA computing architecture “Kepler” since it shows plenty of optimizations.

Furthermore since this study investigates and verifies the performance of GPUs by adapting the benchmarks which are used in other studies, it would also be useful to validate the results for higher precision and better performance measures using relatively better benchmarks.
References

http://www.top500.org/list/2012/06/100

[2] CUDA Developer Zone

[3] OpenCL, Khronos Group
http://www.khronos.org/


[12] NVIDIA Fermi White Papers


http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-280/specifications

[14] Geforce GTX 580 Specifications

http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580/specifications

[15] NVIDIA Tesla C2050


Appendix A

Code Snippets for Arithmetic Latency & Throughput and Device Memory Tests

```c
#define MEASURE_LATENCY(FUNC)
  do {
    Db.x = 32;
    FUNC <<<Dg, Db>>>(d_ts, d_out, 4, 6, 2);
    cudaMemcpy(d_ts, ts, sizeof(d_ts), cudaMemcpyDeviceToHost);
    printf("%s latency:	", #FUNC);
    if ((error = cudaGetLastError()) != cudaSuccess) {
      printf("failed. %s\n", cudaGetErrorString(error));
      break;
    }
    cudaMemcpy(ts, d_ts, sizeof(ts), cudaMemcpyDeviceToHost);
    printf ("%u clk (%.3f clk/warp)\n", ts[1]-ts[0], ((double)(ts[1]-ts[0])/kernel_ops));
  } while(0)

#define MEASURE_THROUGHPUT(FUNC)
  do {
    Db.x = 512;
    FUNC <<<Dg, Db>>>(d_ts, d_out, 4, 6, 2);
    cudaThreadSynchronize();
    cudaMemcpy(d_ts, ts, sizeof(d_ts), cudaMemcpyDeviceToHost);
    printf("%s throughput:	", #FUNC);
    if ((error = cudaGetLastError()) != cudaSuccess) {
      printf("failed. %s\n", cudaGetErrorString(error));
      break;
    }
    cudaMemcpy(ts, d_ts, sizeof(ts), cudaMemcpyDeviceToHost);
    printf ("%u clk (%.3f clk/warp)\n", ts[1]-ts[0], ({double}(ts[1]-ts[0])/kernel_ops));
  } while(0)
```

---

Demyystifying GPU Microarchitecture through Microbenchmarking
This is only a part of the working code. Entire source code can be downloaded from http://www.stuffedcow.net/research/cudabmk
Measures Latency and Throughput for Arithmetic Pipeline and for multiple warps
---
void measure_pipeline()
{
    const int kernel_ops = 256;  // kernels have this many operations
    unsigned int ts[2 * 1024];   // ts, output from kernel. Two elements used per
                                 // thread.
    unsigned int *d_ts;
    unsigned int *d_out;         // Unused memory for storing output

    dim3 Db = dim3(1,1,1);
    dim3 Dg = dim3(1,1,1);

    // Allocate device array.
    if (cudaSuccess != cudaMalloc((void**)&d_ts, sizeof(ts)))
        printf("cudaMalloc failed %s:%d
", __FILE__, __LINE__);
    return;
    
    if (cudaSuccess != cudaMalloc((void**)&d_out, 4))
        printf("cudaMalloc failed %s:%d
", __FILE__, __LINE__);
    return;
    
    cudaGetLastError();
    fprintf(stderr, "Running pipeline tests...\n");

    /* Pipeline latency/throughput for all three functional units */
    MEASURE_LATENCY(K_ADD_UINT_DEP128);
    MEASURE_THROUGHPUT(K_ADD_UINT_DEP128);
    cudaFree(d_ts);
    cudaFree(d_out);
}

/* Run the test for increasing number of warps, and generate histogram of runtime. */
#define PRINT_HISTOGRAM(FUNC)
    do {
        printf("nPipeline latency/throughput with multiple warps (200 iterations of %d
ops)\n", kernel_ops);
        for (Db.x = 1; Db.x <= 1024; Db.x += (Db.x < 4)? 1 : (Db.x < 8)? 2 : (Db.x < 32)? 8
            : 32)
            {
                unsigned int histogram[1024] = {0};
                unsigned int sum_time = 0;
                unsigned int max_time;
                unsigned int min_time;
                unsigned int sum_max_time = 0;
                bool failed = false;

                for (int i=0;i<200 && !failed ;i++)
                    {
                        cudaGetLastError();  /* Clear previous error code, if any */
                        FUNC <<<Dg, Db>>>(d_ts, d_out, 4, 6, 2);
                        if (cudaGetLastError() != cudaSuccess)
                        {
                            printf("cudaMalloc failed %s:%d
", __FILE__, __LINE__);
                            return;
                        }
                        
                        max_time = MIN(max_time, d_out[i]);
                        min_time = MAX(min_time, d_out[i]);
                        sum_max_time += d_out[i];
                        sum_time += d_out[i];
                        if (d_out[i] > max_time)
                            max_time = d_out[i];
                        if (d_out[i] < min_time)
                            min_time = d_out[i];
                        if (d_out[i] > sum_max_time)
                            sum_max_time = d_out[i];
                        }
failed = true;
break;
}
cudaThreadSynchronize();
cudaMemcpy(ts, d_ts, sizeof(ts), cudaMemcpyDeviceToHost);
max_time = 0;
min_time = (unsigned)-1;
/* Compute histogram. */
for (int j=0; j< Db.x*2; j+= 64)
{
    sum_time += (ts[j+1] - ts[j]);
    max_time = max(max_time, ts[j+1]);
    min_time = min(min_time, ts[j]);
    histogram[(ts[j+1] - ts[j])/kernel_ops]++;
}
sum_max_time += max_time-min_time;
}
if (failed)
{
    printf ("%2d warp%c (%3d thread%c) failed.", (Db.x+31)/32, Db.x>=64 ? 's': ' ', Db.x, Db.x > 1 ? 's': ' ');
} else
{
    /* Compute average latency over the lifetime of each warp (sum_time), and average throughput of the kernel (sum_max_time). */
    printf ("%2d warp%c (%3d thr) %u clk (%.3f clk/warp, %.3f ops/clk)",
            (Db.x+31)/32, Db.x>=64 ? 's': ' ', Db.x, sum_max_time,
            sum_time/200.0/kernel_ops/((Db.x+31)/32),
            kernel_ops*200.0*Db.x/sum_max_time);
    printf (" Histogram { "); /* Print a histogram of each thread's runtime for the last iteration. */
    for (int i=0; i<1024; i++) /* Print the non-zero entries only */
    {
        if (histogram[i] != 0)
            printf ("(%d: %d) ", i, histogram[i]);
    }
    printf ("} ");
}
printf ("\n");
}
printf ("\n");
}

//************************************************************************************
// Scalable Heterogeneous Benchmark(SHOC) Computing Suite
// Programmer: Kyle Spafford
// Creation: September 08, 2009
// Filename: Devicememory.cu
// Measures read and write bandwidth for Global Memory Coalesced/Uncoalesced Access
// and Shared Memory bandwidth
// This is only a part of the working code. Entire source code can be downloaded from
// https://github.com/spaffy/shoc/wiki
*******************************************************************************/
__global__
readGlobalMemoryCoalesced(float *data, float *output, int size, int repeat)
{
    int gid = threadIdx.x + (blockDim.x * blockIdx.x), j = 0;
    float sum = 0;
    int s = gid;
    for (j=0; j<repeat; ++j)
    {
        float a0 = data[(s+0)*(size-1)];
        float a1 = data[(s+32768)*(size-1)];
        float a2 = data[(s+65536)*(size-1)];
        float a3 = data[(s+98304)*(size-1)];
        float a4 = data[(s+131072)*(size-1)];
        float a5 = data[(s+163840)*(size-1)];
        float a6 = data[(s+196608)*(size-1)];
        float a7 = data[(s+229376)*(size-1)];
        float a8 = data[(s+262144)*(size-1)];
        float a9 = data[(s+294912)*(size-1)];
        float a10 = data[(s+327680)*(size-1)];
        float a11 = data[(s+360448)*(size-1)];
        float a12 = data[(s+393216)*(size-1)];
        float a13 = data[(s+425984)*(size-1)];
        float a14 = data[(s+458752)*(size-1)];
        float a15 = data[(s+491520)*(size-1)];
        sum += a0+a1+a2+a3+a4+a5+a6+a7+a8+a9+a10+a11+a12+a13+a14+a15;
        s = (s+524288)*(size-1);
    }
    output[gid] = sum;
}

__global__
readGlobalMemoryUnit(float *data, float *output, int size, int repeat)
{
    int gid = threadIdx.x + (blockDim.x * blockIdx.x), j = 0;
    float sum = 0;
    int s = gid+512;
    for (j=0; j<repeat; ++j)
    {
        float a0 = data[(s+0)*(size-1)];
        float a1 = data[(s+32768)*(size-1)];
        float a2 = data[(s+65536)*(size-1)];
        float a3 = data[(s+98304)*(size-1)];
        float a4 = data[(s+131072)*(size-1)];
        float a5 = data[(s+163840)*(size-1)];
        float a6 = data[(s+196608)*(size-1)];
        float a7 = data[(s+229376)*(size-1)];
        float a8 = data[(s+262144)*(size-1)];
        float a9 = data[(s+294912)*(size-1)];
        float a10 = data[(s+327680)*(size-1)];
        float a11 = data[(s+360448)*(size-1)];
        float a12 = data[(s+393216)*(size-1)];
        float a13 = data[(s+425984)*(size-1)];
        float a14 = data[(s+458752)*(size-1)];
        float a15 = data[(s+491520)*(size-1)];
        sum += a0+a1+a2+a3+a4+a5+a6+a7+a8+a9+a10+a11+a12+a13+a14+a15;
        s = (s+16)*(size-1);
    }
    output[gid] = sum;
}

__global__
readLocalMemory(const float *data, float *output, int size, int repeat)
{
    int gid = threadIdx.x + (blockDim.x * blockIdx.x), j = 0;
    float sum = 0;
    int tid = threadIdx.x, localSize = blockDim.x, grpid = blockIdx.x,
void lbuf[524288];

for ( ; j<items & j<(size-goffset) ; ++j)
    lbuf[tid*items+j] = data[goffset+j];
for (int i=0 ; j<items ; ++j,++i)
    lbuf[tid*items+j] = data[i];
__syncthreads();
for (j=0 ; j<repeat ; ++j)
{
    float a0 = lbuf[(s*0)&(2047)];
    float a1 = lbuf[(s*1)&(2047)];
    float a2 = lbuf[(s*2)&(2047)];
    float a3 = lbuf[(s*3)&(2047)];
    float a4 = lbuf[(s*4)&(2047)];
    float a5 = lbuf[(s*5)&(2047)];
    float a6 = lbuf[(s*6)&(2047)];
    float a7 = lbuf[(s*7)&(2047)];
    float a8 = lbuf[(s*8)&(2047)];
    float a9 = lbuf[(s*9)&(2047)];
    float a10 = lbuf[(s+10)&(2047)];
    float a11 = lbuf[(s+11)&(2047)];
    float a12 = lbuf[(s+12)&(2047)];
    float a13 = lbuf[(s+13)&(2047)];
    float a14 = lbuf[(s+14)&(2047)];
    float a15 = lbuf[(s+15)&(2047)];
    sum += a0+a1+a2+a3+a4+a5+a6+a7+a8+a9+a10+a11+a12+a13+a14+a15;
    s = (s+16)&(2047);
}
output[0] = sum;

__global__ void
writeGlobalMemoryCoalesced(float *output, int size, int repeat)
{
    int gid = threadIdx.x + (blockDim.x * blockIdx.x), j = 0;
    int s = gid;
    for (j=0 ; j<repeat ; ++j)
    {
        output[(s*0)&(size-1)] = gid;
        output[(s*2047)&(size-1)] = gid;
        output[(s*4094)&(size-1)] = gid;
        output[(s*8190)&(size-1)] = gid;
        output[(s*16380)&(size-1)] = gid;
        output[(s*32760)&(size-1)] = gid;
        output[(s*65520)&(size-1)] = gid;
        output[(s*131040)&(size-1)] = gid;
        output[(s*262144)&(size-1)] = gid;
        output[(s*524288)&(size-1)] = gid;
        s = (s*524288)&(size-1);
    }
}

__global__ void
writeGlobalMemoryUnit(float *output, int size, int repeat)
{
    int gid = threadIdx.x + (blockDim.x * blockIdx.x), j = 0;
    int s = gid*512;
    for (j=0 ; j<repeat ; ++j)
    {
        output[(s*0)&(size-1)] = gid;
__global__
for (int j = 0; j < litems; ++j)
{
    lbuf[(s+16) & (2047)] = gid;
    lbuf[(s+15) & (2047)] = gid;
    lbuf[(s+14) & (2047)] = gid;
    lbuf[(s+13) & (2047)] = gid;
    lbuf[(s+12) & (2047)] = gid;
    lbuf[(s+11) & (2047)] = gid;
    lbuf[(s+10) & (2047)] = gid;
    lbuf[(s+9) & (2047)] = gid;
    lbuf[(s+8) & (2047)] = gid;
    lbuf[(s+7) & (2047)] = gid;
    lbuf[(s+6) & (2047)] = gid;
    lbuf[(s+5) & (2047)] = gid;
    lbuf[(s+4) & (2047)] = gid;
    lbuf[(s+3) & (2047)] = gid;
    lbuf[(s+2) & (2047)] = gid;
    lbuf[(s+1) & (2047)] = gid;
    lbuf[(s) & (2047)] = gid;
    lbuf[(s-1) & (2047)] = gid;
    lbuf[(s-2) & (2047)] = gid;
    lbuf[(s-3) & (2047)] = gid;
    lbuf[(s-4) & (2047)] = gid;
    lbuf[(s-5) & (2047)] = gid;
    lbuf[(s-6) & (2047)] = gid;
    lbuf[(s-7) & (2047)] = gid;
    lbuf[(s-8) & (2047)] = gid;
    lbuf[(s-9) & (2047)] = gid;
    lbuf[(s-10) & (2047)] = gid;
    lbuf[(s-11) & (2047)] = gid;
    lbuf[(s-12) & (2047)] = gid;
    lbuf[(s-13) & (2047)] = gid;
    lbuf[(s-14) & (2047)] = gid;
    lbuf[(s-15) & (2047)] = gid;
    s = (s+16) & (2047);
}
__syncthreads();
for (j = 0; j < litems; ++j)
    output[gid] = lbuf[tid];