CuMAPz: Analyzing the Efficiency of Memory Access Pattern in CUDA

Yooseong Kim and Aviral Shrivastava

Compiler and Microarchitecture Laboratory
Arizona State University, Tempe 85281, USA
{yooseong.kim, aviral.shrivastava}@asu.edu

Abstract

Even though the entry barrier of writing a GPGPU program is lowered with the help of many high-level programming models, such as NVIDIA CUDA, it is still very difficult to optimize a program so as to fully utilize the given architecture's performance. The burden of GPGPGU programmers is increasingly growing as they have to consider many parameters, especially on memory access pattern, and even a small change of those parameters can lead to a drastic performance change, which is not obvious, or often counter-intuitive, before careful analysis. In this paper, we focus on optimizing a CUDA program using shared memory. We present a tool that analyzes the efficiency of given parameters on memory access pattern. Given a set of parameters, the tool analyzes data reuse, global memory access coalescing, shared memory bank conflict, partition camping, and branch path divergence. The output of the tool is profitability, a comprehensive performance metric introduced in this paper. Profitability can be used to compare the efficiencies of different sets of parameters, without even writing a program. Experimental results show that profitability can accurately predict the change of the performance of a program as we change the memory access pattern related parameters.

Categories and Subject Descriptors D.1.3 [*Programming Technique*]: Concurrent Programming; C.1.4 [*Processor Architecture*]: Modeling techniques

General Terms Performance, Design, Experimentation

Keywords GPGPU, CUDA, Memory access pattern, Performance estimation, Analytical Model

1. Introduction

The computational power of modern Graphics Processing Units (GPUs) has been rapidly increasing, and now reached teraFLOPs scale. Given the tremendous computing power from the cheap, easily-accessible device, researchers began to be interested in utilizing GPUs for applications other than graphics. This trend has evolved into the new computing paradigm, GPGPU (General Purpose computation on GPUs), which is to accelerate programs by co-processing on the CPU and GPU.

As high-performance parallel architectures, GPGPUs are giving us a promising view of affordable supercomputing on desktop systems. Traditionally GPGPU programming required a high level of expertise and proficiency. Being a modestly extended version of C, NVIDA CUDA programming model [7] has successfully lowered the entry barrier for programmers to write GPGPU codes. However, to develop high-performance applications that fully utilize the given hardware's potential, or to optimize an existing program, is still very complicated and tricky. It is mainly due to a large number of optimization principles [9, 17] that come from architectural and programming models' details, and complex memory hierarchy of architecture. Programmers need to consider all of performance-critical factors carefully since a slight change in source code can result in much slower, or faster, runtime. Furthermore, programmers have to decide numbers of parameters regarding memory access pattern, such as loop structure, array index requested by each thread, data to be fetched in shared memory, the shared memory array size, etc. To try all of the options is inefficient and often infeasible because writing a different version of a kernel is still complicated work and space of optimization to be explored is too large [16]. We cannot impose programmers this heavy burden of considering all of detail factors at the same time while writing a massively parallel program.

In this paper, we focus on the difficulties in optimizing an existing CUDA program using shared memory. Shared memory is an explicitly-managed memory of a small size that each multiprocessor has on-chip. Using shared memory is especially important in optimizing CUDA programs, because the global memory in CUDA architecture is not cached and also bandwidth-constrained, while shared memory is as fast as registers when there is no bank conflict. Also, shared memory is the only *fast* memory on CUDA where both read and write are enabled. Though cached and fast, other memories in memory hierarchy, texture memory and constant memory, are read-only, making their area of use more limited than shared memory. Even in this small subset of the problem, there are many factors, especially on memory access patterns, that need to be considered very carefully. For example, traditional data reuse analysis for explicitly managed scratchpads is not enough for shared memory in GPUs. Slow global memory can often become a bottleneck when there is uncoalesced accesses or partition camping [9], leading to counter-intuitive results. Shared memory access pattern can cause severe branch divergence which also has a huge impact on performance. Shared memory bank conflicts can't be ignored either. A program sometimes becomes even slower after introducing shared memory buffers due to all the above mentioned factors. One cannot accurately analyze the performance of programs without considering all the factors at the same time.

To address this problem, we present a tool, named CUDA Memory Access Pattern analyZer (CuMAPz), which helps programmers choose the memory access related parameters without even writing the kernels. Given a set of parameters regarding memory access pattern as input, CuMAPz comprehensively analyzes the efficiency of the memory access pattern of the program. Then, it outputs profitability, a metric introduced in this paper, which enables programmers to compare the relative efficiency of one set of parameters over others. Profitability is obtained by combing the result of these analyses: data reuse; global memory access coalescing; shared memory bank conflict; partition camping; and branch divergence. By checking those well-known architectural and programming model's features that have significant impact on performance, all at once, our approach finds any possible performance-limiting factor in memory access pattern that will very-possibly slow down the execution of a program and helps developers make a decision on designing a program.

The rest of this paper is organized as follows. In Section 2, we briefly introduce NVIDIA CUDA programming model and memory hierarchy as background. Then, we present motivational examples to show the importance of memory access parameters. We discuss related work in Section 4. Overview of our approach and detailed explanation are presented in Section 5. In Section 6, experimental results are presented to validate the effectiveness of our approach. Then, we discuss the limitation of our approach in Section 7 and then conclude in Section 8.

2. Backgroud

NVIDIA Compute Unified Device Architecture (CUDA) and its programming mode have evolved over generations. In this paper, we focus on the NVIDIA GT200 architecture, which has compute capability version [7] of 1.3. This does not harm the generality of our work, since the same or similar approach as ours can be applied to different generations.

2.1 CUDA Programming Model

CUDA programming model is basically an extended version of C designed to help programmers write kernels relatively easily with an abstraction of hardware. In CUDA programming model, serial code executes on CPU while parallel code executes on GPU. Code running on GPU is written as functions, called kernels. CPU code launches one kernel at a time for its execution on GPU and transfers data for input and output to and from GPU.

CUDA architecture is massively parallel in that a kernel is executed by thousands of threads. In NVIDIA GT200 architecture, there is a grid of streaming multiprocessors (SMs), and each SM has eight scalar processors (SPs). The number of SMs in NVIDIA Tesla C1060 [8] is 30, which makes the number of cores on device be 240 in total. Each SM can have up to 1024 threads in-flight, unless other conditions are set. Threads are grouped into thread blocks. Each thread has its own thread id to represent the relative location in a block, and thread blocks also have ids for themselves. Combining the block id and the thread id, each thread is assigned a unique id.

Each thread block is assigned to SM to be executed. Thus, the basic unit of scheduling in SMs is a thread block. When a kernel is launched, the order in which threads blocks are assigned to SMs is sequential so that adjacent blocks are executed on adjacent SMs. However, it becomes unpredictable after the first round of schedule since the order in which thread blocks finish the execution cannot be determined [9]. On the other hand, the actual execution of threads on SPs is done in groups of 32 threads, called warps. All threads in thread blocks assigned to one SM are grouped into warps, and thread ids in a warp are consecutive. SPs execute one warp at a time, and the execution of a warp is in SIMD manner, so threads in the same warp are executed in lock-step, which means all SPs in one SM execute the same instruction at a time.

```
1  int row = bIdx.y*bDim.y+tIdx.y;
2  int col = bIdx.x*bDim.x+tIdx.x;
3
4  if (col >= MAX-2)
5    return;
6
7  out[row*MAX+col]  =  in[row*MAX+col] *
8                       in[row*MAX+col+1] *
9                       in[row*MAX+col+2];
```

Figure 1. A simple CUDA program

2.2 Global Memory and Shared Memory

In CUDA-enabled GPU devices, there is an off-chip DRAM, called global memory[1]. CPU code can only transfer data to this off-chip memory, so all data reside in global memory unless specified otherwise. Since global memory is not cached, the latency of accessing global memory is hundreds of cycles, making it the slowest memory on device. However, this latency can often be hidden with the help of having a large number of threads in-flight and pipelining. Thus, other warps with independent operations can be executed instead of just waiting for the IO operation to be done in a warp. What is more crucial for performance is the bandwidth utilization. Even though GPUs have broad bus width (512-bit in NVIDIA Tesla C1060), the massive parallelism easily saturates the given bandwidth. This often becomes the performance bottleneck of CUDA programs. We discuss this in detail in Section 5.

Shared memory is on-chip memory whose latency is as fast as registers'. It is an explicitly managed memory which is often used as a local buffer for fast retrieval of data. Since shared memory is on-chip and only accessed by threads running on the chip, the bandwidth hardly becomes a performance limiting factor. However, shared memory bank conflict can slow down a program. We discuss this in detail later in Section 5. Even though shared memory is fast, since it is only shared within one block, and its size is quite limited (16KB in compute capabilities of 1.x), the structure of a program and memory access pattern are very important to fully utilize the shared memory.

3. Motivating Example

In this section, we start from a very simple CUDA program and try to optimize its performance using shared memory. We show that the optimization of CUDA programs cannot be done *intuitively* or *by hand*.

Figure 1 shows a simple CUDA kernel which is not using shared memory. The CUDA built-in variables blockIdx, blockDim, and threadIdx are abbreviated and shown as bIdx, bDim, and tIdx, respectively. There are three references for array $in[]$ which is two dimensional array with size MAX*MAX. The if condition at line 4 is given to avoid accessing over the array boundary by the reference row$*$MAX+col+2. Array element type is $float$ for both $in[]$ and $out[]$, whose size is 4-byte. The thread block is two dimensional, and the size is 256 (16 threads for each dimension). We are now trying to improve the performance of this program by using shared memory. We do not consider the thread block size as a parameter here, so the thread block dimension remains the same.

[1] Although there are other types of memory sharing the off-chip DRAM, we do not mention them here for simplicity because we focus on only global memory and shared memory in this paper.

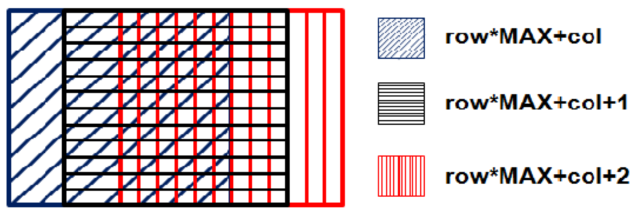


Figure 2. Data reuse within one thread block

3.1 Example 1: What to fetch into shared memory

Since shared memory is often used as a fast local buffer for frequently accessed data, it is intuitively obvious that we get to think about data reuse. There is an overlap among the memory regions accessed by existing three references row∗MAX+col, row∗MAX+col+1, and row∗MAX+col+2 to array *in*[] within one thread block. This means that we can have data reuse by fetching some data from array *in*[]. Figure 3 shows the code converted to use shared memory where the data accessed by the reference row∗MAX+col is fetched into. This code will serve as a running example throughout this section. Here, we only consider the existing three references as our candidates to use as a fetch function, for simplicity. As shown in Figure 2, row∗MAX+col+1 has the largest overlap among three candidates. Thus, fetching data using that reference should incur the most data reuse and result in the fastest runtime. However, the actual runtime turns out differently.

```
1   int row = bIdx.y*bDim.y+tIdx.y;
2   int col = bIdx.x*bDim.x+tIdx.x;
3   float t1, t2, t3;
4
5   __shared__ float s_in[BLKDIM][BLKDIM];
6
7   s_in[tIdx.x][tIdx.y] = in[row*MAX+col];
8
9   if (col >= MAX-2)
10    return;
11
12  t1 = s_in[tIdx.x][tIdx.y];
13
14  if (tIdx.x == bDim.x-1)
15    t2 = in[row*MAX+col+1];
16  else
17    t2 = s_in[tIdx.x+1][tIdx.y];
18
19  if (tIdx.x >= bDim.x-2)
20    t3 = in[row*MAX+col+2];
21  else
22    t3 = s_in[tIdx.x+2][tIdx.y];
23
24  out[row*MAX+col] = t1*t2*t3;
```

Figure 3. A simple CUDA program

Table 1 shows the runtimes of kernels on NVIDIA Tesla C1060. Each row represents the kernel fetching data into shared memory using each of three references. MAX, the input size, is defined as 16384*16384, making 1024 blocks for each dimension. BLKDIM, which is the same as bDim.x and bDim.y, is set to 16, so the thread block size is 256. The second column 'data reuse' represents the number of times the data in shared memory is accessed. Counter-intuitively, the case using the first reference row∗MAX+col is fastest, and even the last case with smallest data reuse is faster than the second case. This is mainly caused by global memory access coalescing and shared memory bank conflict, and we discuss this in Section 5.2.2 and Section 5.2.4.

	Data reuse	Runtime (*in ms*)
not using shared mem	-	76.15
row*MAX+col	3019702272	61.11
row*MAX+col+1	3086680064	64.86
row*MAX+col+2	3019505664	63.77

Table 1. Runtime results of Example1

3.2 Example 2: How to store data into shared memory

In the above section, we only considered what data to be fetched into shared memory. However, how the data is stored is also an important factor that determines the performance of CUDA programs using shared memory.

In Figure 3, data is written to and read back from shared memory buffer s_in in a column-wise manner, as shown at Line 7, 12, 17, and 22. The way data is stored determines the way data is read back. If we change the accesses to row-wise manner as in s_in[tIdx.y][tIdx.x], the runtime becomes much faster. This is not only caused by row-wise and column-wise access. Instead of changing all shared memory references, we can see the same effect with changing the shared memory buffer declaration at Line 5 as below. The largest difference in runtime among three cases is around 20%.

```
__shared__ float s_in[BLKDIM][BLKDIM+1];
```

Table 2 shows the runtime of each case. This improvement of runtime is achieved by removing shared memory bank conflict, and we discuss it in Section 5.2.4.

	Runtime (*in ms*)		
	Column-wise	Row-wise	BLKDIM+1
row*MAX+col	61.11	46.06	45.98
row*MAX+col+1	64.86	54.75	54.69
row*MAX+col+2	63.77	55.25	55.39

Table 2. Runtime results of Example2

3.3 Example 3: How to access global memory

Besides shared memory, the way global memory is accessed also affects performance significantly. There is a global memory write instruction to array *out*[] at the end of the code in Figure 3, Line 24. The access is in a row-wise manner. Let us, however, suppose it is in a column-wise manner as in out[col∗MAX+row] and see what the results will be different. Shared memory accesses kept the same in a row-wise manner as we changed in Example 2.

	Runtime (*in ms*)	
	Row-wise	Column-wise
not using shared memory	-	3938.08
row*MAX+col	46.06	3933.88
row*MAX+col+1	54.75	3936.23
row*MAX+col+2	55.25	3937.56

Table 3. Runtime results of Example3

The consequence of this change is so tremendous, as shown in Table 3, that the runtime of the program becomes two orders of magnitude slower. We even show the case without using shared memory, and we can hardly see benefit of using shared memory in three cases. This emphasizes the importance of memory access pattern analysis in CUDA. The slowdown of this example is caused by partition camping, and we discuss this in Section 6.2.3.

4. Related Work

Data reuse analysis and hierarchical buffer organizations for scratchpad memories have been widely studied in many papers such as [1, 3, 4]. However, there are fundamental differences between the baseline architecture in these papers and GPUs. Firstly, global memory in GPUs is not cached. The loss of performance when accessing data not fetched into scratchpads is relatively much more dramatic. Also, the access pattern when fetching data into buffer is also very important. Secondly, shared memory is only accessible within threads in the same block, and thread block configuration is not fixed but can change, so the data reuse analysis is more difficult in GPUs. Moreover, data reuse is not the only dominant factor in performance as described in Section 3.

Since traditional data reuse analysis does not apply very well to GPUs, many researchers started to work on analytical performance models, to help developers optimize GPGPU applications more easily. Ryoo et al. [16] modeled the amount of parallelism employed in a program and the efficiency of a single kernel execution in a thread, but they did not consider memory access latency assuming all programs as non-memory intensive applications. Later work [5, 6] considered the cost of both computation and memory access. Hong et al. [6] proposed an analytical model that includes the effect of parallelism to hide global memory access latency. Their model, however, does not take into account shared memory and other detailed architecture-specific features such as global memory access coalescing. The model proposed in [5] includes shared memory bank conflict, but neither of these two approaches consider data reuse, branch path divergence, partition camping, etc. Comparing to all the above works, our approach is a more complete model for analyzing memory performance in that every aspect of global memory and shared memory is considered.

As a more aggressive solution to relieve the burden on programmers, many researchers have been interested in automated optimization of GPGPU applications. Ueng et al. [12] first presented a tool which optimizes a program automatically. However, it required programmers' annotations on source code in a specific format to find out any possibility of optimization. Baskaran et al. [11] proposed a compiler framework which automatically explores better program design and transforms a program to the more desirable structure. They presented a source to source compiler in more recent work [14]. Later, another work [13] also presented a source-to-source compiler with a few improvements such as weighted cost model that balances parallelism and locality, exploring more beneficial tile size and thread block size, local buffer size, etc. However, in both approach, data reuse in shared memory can only be employed when the corresponding global memory accesses cannot be coalesced. The serialized execution by branch path divergence or partition camping is not considered as well. Recently, Yang et al. [10] presented another optimizing compiler that generates an optimized program from a very naive kernel function. Their approach takes into account most of the factors that we consider in this paper, but the model of coalescing is incomplete and branch path divergence is not considered either. Overall, none of the above work comes up with a comprehensive performance metric to estimate the efficiency of memory access pattern. Any undesirable pattern is simply avoided shortsightedly, even if it is possible that the overall performance can be improved by changes in other factors.

5. Our Approach

5.1 Overview

The examples in the previous section show that a small change in memory access pattern can result in a huge performance difference. Those performance considerations are often affected unexpectedly by a small change. To address this problem, we aim to help programmers decide design parameters related to memory access pattern of a program. Given a set of parameters listed below as input, CuMAPz analyzes the program's memory access pattern and checks all of the well-known performance considerations.

- Thread block / grid dimension
- Information for each *in global memory* array
  - Array dimension
  - Array element size
  - Reference(s) for each array
- Information for each *in shared memory* buffer
  - Buffer dimension
  - Mapping to global memory array
    - Global memory read reference & Shared memory write reference (when used as a *read* buffer)
    - Global memory write reference & Shared memory read reference (when used as a *write* buffer)
- Loop information (only if it is with array references)
  - Initialization, Terminating condition, and Step
- Hardware Information
  - Number of channels in global memory
  - Width of channel
  - Number of banks in shared memory

Using the given thread block size and the loop information, CuMAPz constructs a loop structure that exactly follows the execution of each warp in a block. It essentially traces memory addresses which are accessed in a program. Then, it checks 1) how many times data in shared memory is accessed (data reuse), 2) how global memory accesses are coalesced, 3) if global memory accesses are not skewed to some of memory channels, 4) if shared memory accesses generate any bank conflict, and 5) if the use of shared memory introduces a branch. All these factors are explained in detail in next section. Combining all of these factors, the output, *profitability*, is calculated according to the formula which will also be described in next section.

Note that we do not explicitly take shared memory references as input. We only keep track of the mapping between global memory array elements and shared memory array elements. For example, Line 7 in Figure 3 is a pair of a global memory read reference and a shared memory write reference, which represents the mapping. Shared memory accesses are inferred by global memory references read from (or write to) the position where shared memory buffer is connected. We assume that the mapping is one-to-one, so one element in global memory array is only mapped to an elment in one buffer.

Profitability is a relative metric, so there is no use in having only one profitability value. Once a set of profitability values are calculated for different input parameters, each profitability value can represent the relative memory performance of the corresponding parameters. Having higher profitability means more efficient memory access pattern which can lead to faster runtime.

5.2 Memory Access Pattern Analysis

5.3 Data Reuse

As discussed in Section 3, the use of shared memory can improve program performance dramatically. Since shared memory is used as a local buffer, it is obvious that we should maximize data reuse as long as other factors are not negatively affected, which will be discussed later in this paper.

Any instance of global memory reference can be represented as an absolute memory address. For each element in shared memory buffer in a thread block, the associated global memory address is obtained by the mapping. During the iteration of loop for each warp, if a global memory address accessed by the references is already mapped to one of buffers within the same block, then a counter is increased. In order to state the degree of data reuse in figures, CuMAPz maintains the counter to count the number of times shared memory buffers are accessed. Then, the degree of data reuse is represented in a term, $data\_reuse$, as follows:

$$data\_reuse = \frac{bytes\_shmem}{bytes\_buffered} \tag{1}$$

$$bytes\_buffered = \sum_{m \in M} \sum_{w \in W} bytes\_tr_m^w$$

$$bytes\_shmem = \sum_{r \in R} \sum_{b \in B} \sum_{w \in W} bytes\_shmem_r^w$$

, where M, R, B and W denote the set of all global memory references in mappings between global memory array and buffer, the set of all global memory references, the set of all buffers, and the set of all warps, respectively. $bytes\_tr_m^w$ represents the bytes transferred while feching data from global memory (or updating global memory with data written in buffer), in warp w. $bytes\_shmem_r^w$ denotes the bytes read from (or written to) shared memory buffer for data required (or written) by global memory reference r, during the execution of warp w. The buffer b is not in the variable because we assuem one-to-one mapping between global memory array and buffer, which means a unique b is determined by the address accessed by r. The concept of the bytes transferred in the term $bytes\_tr$ is detailed in next section. $bytes\_buffered$ can be regarded as the buffer size, but as in the example from Section 3, the buffer dimension is not always the same as the amount of data filled to the buffer. Therefore, we maintain a seperate variable to keep the amount of data transfer between global memory and shared memory.

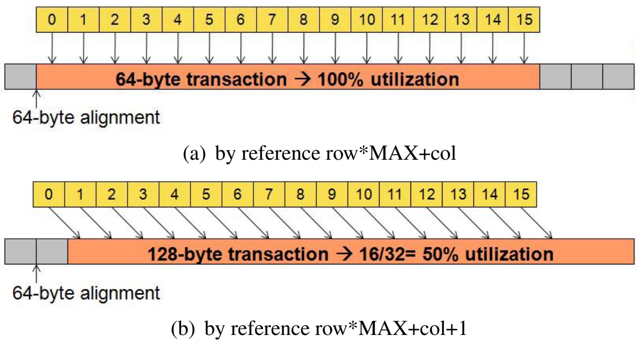
5.4 Coalesced Global Memory Accesses

In CUDA architecture, global memory accesses by a half warp, 16 threads with consecutive ids, are coalesced into fewer number of transactions. Coalescing happens when threads in a half warp access an aligned and continuous memory region[2]. The alignment should be 32-byte, 64-byte, and 128-byte for the element size of one, two, and four byte, respectively. The best bandwidth utilization is achieved when threads in a half warp access aligned 16 consecutive elements in an array. In other words, if the access pattern does not satisfy the above condition, coalescing incurs some waste of bandwidth. It is because even unnecessary data is transferred as a chunk.

Figure 4 shows the global memory accesses of threads of a half warp by references in the program from Section 3. Boxes in the upper row represent threads in a half warp, and the ones in the lower row represent global memory space. As you can see, reference row$*$MAX+col+1 is misaligned, which results in only 50% of bandwidth utilization. Misaligned access is often the main cause of any waste of bandwidth, but not only is.

CuMAPz analyzes the coalescing behavior according to the complete description in CUDA Programming Guide [7]. When addresss accessed by a global memory reference is not mapped to any buffer, global memory access occurs. The transaction size, of 32-byte, 64-byte, and 128-byte, is determined for the memory access

[2] The coalescing behavior in devices of compute capability 1.0 or 1.1 is slightly different from that of compute capability 1.2 or higher, and our analysis is based on the latter.



(a) by reference row*MAX+col

(b) by reference row*MAX+col+1

Figure 4. Memory accesses by threads in a half warp

pattern in each half warp according to the architecture specification [7]. Then, CuMAPz calculates the bandwidth utilization as the following:

$$bw\_util = \frac{bytes\_acc}{bytes\_tr} \tag{2}$$

$$bytes\_acc = \sum_{r \in R} \sum_{w \in W} bytes\_acc_r^w$$

$$bytes\_tr = \sum_{r \in R} \sum_{w \in W} bytes\_tr_r^w$$

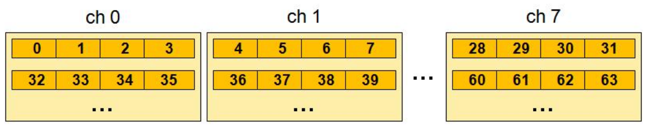
, where $bytes\_acc_r^w$ and $bytes\_tr_r^w$ are the size of accessed (read or write) data and transmitted data, respectively, for reference r in warp w. It is represented in terms of warps for simplicity, but, as described in the above, the actual analysis is done at the half warp granularity.

5.5 Partition Camping

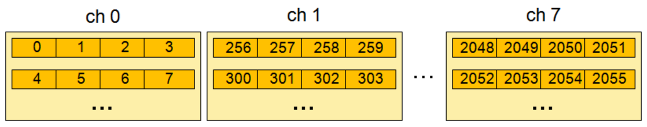
The memory subsystem in CUDA is multi-channel architecture. NVIDIA GT200 architecture has eight memory channels, and consecutive 256-byte regions in global memory are mapped to consecutive channels. Each channel is of 64-bit width, and all of channels can transmit data from memory to cores at the same time in parallel. In order to fully utilize this setup, the accesses that occur concurrently on all SMs, should be evenly distributed among all channels. Partition camping refers to the case where the accesses are skewed to only a few of channels, so called partitions [9]. As shown in Section 3.3, this results in the most significant performance degradation. It is because partition camping essentially makes the bus width much narrower worsening the bottleneck problem on top of the slow latency of global memory.

The serious slowdown of a program, in Section 3.3, after changing the global memory reference row$*$MAX+col to col$*$MAX+row is caused by partition camping. Figure 5 shows how the accesses of the first warps in blocks are mapped to memory channels. The numbers inside boxes in each channel represent thread block ids. Considering the thread block dimension, which is 16x16 and the size of the array element type, which is 4-byte, we can see that a groups of four consecutive blocks is assigned to each channel, when using reference row$*$MAX+col. However, when using reference col$*$MAX+row, a group of 256 successive blocks is assigned to each channel as shown in the figure. This leads to the severe partition camping where all accesses in-flight are focused on one channel.

To analyze partition camping, we need to know on which channel the accesses of all warps running concurrently are mapped. This makes this analysis tricky because to determine which blocks are being executed concurrently is impossible. Initially, when a kernel



(a) by reference row*MAX+col



(b) by reference col*MAX+row

Figure 5. Memory accesses of blocks mapped to channels

is launched, threads blocks are assigned to SMs in a sequential order so that adjacent blocks are executed on adjacent SMs. Then, it becomes unpredictable after the first round of schedule since the order in which thread blocks finish the execution cannot be determined [9]. If we consider the number of blocks to fill all channels in their execution of first warps, it is:

$$n\_channels \times \frac{channel\_width}{bDim.X \times elem\_size} \quad (3)$$

, where $elem\_size$ is the minimum element size of global memory arrays. This is basically to estimate the span of initial memory accesses of warps. To see the skewness of the accesses among those blocks should be enough to check if there is any potential performance-aggravating factor in a given memory access pattern. Thus, the impact of partition camping can be stated in figures as the skewness of mapping to channels which can be calculated as follows:

$$ch\_skew = \frac{max\_n\_block\_per\_ch}{MAX(1, min\_n\_block\_per\_ch)} \quad (4)$$

, where $max\_n\_block\_per\_ch$ and $min\_n\_block\_per\_ch$ denote, respectively, the maximum and minimum number of blocks assigned to a channel. When the grid dimension, which is the number of thread blocks, is smaller than the number stated in the above, this analysis is skipped.

5.6 Shared Memory Bank Conflict

Similarly to global memory channels, shared memory space is divided into 16 banks. Successive four bytes data are assigned to successive banks. All banks can transmit data in parallel, but each bank can serve one address at a time. When threads in a half warp access K different addresses within one bank, the accesses are serialized K times, and this is called K-way shared memory bank conflicts.

Let us consider the example in Section 3.2. Since, originally, the shared memory buffer dimension is 16x16 and its type is float (4-byte), each column in a row is mapped to a bank, and tIdx.x determines the bank number of the access. Using reference [tIdx.y][tIdx.x], each thread in a half warp should access one address in each bank. After changing the reference to [tIdx.x][tIdx.y], all threads in a half warp now access 16 different addresses in one bank. This results in 16-way bank conflicts. Interestingly, changing the shared buffer array dimension to 16x17 can avoid the bank conflicts. It makes the addresses requested by [tIdx.x][tIdx.y] spread over all banks so that there is no bank conflicts. The runtime changes in the example in Section 3.2 can be well explained in this way. CuMAPz analyzes all addresses requested in each half warp and checks if bank conflicts occur. Then, it accumulates all

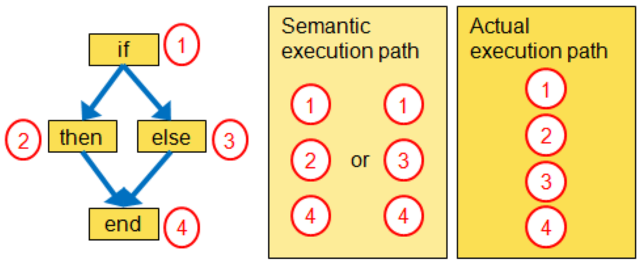


Figure 6. Serialized execution of diverged execution paths in a warp

numbers of bank conflicts in half warps as the following:

$$n\_bk\_conflict = \sum_{r \in R} \sum_{b \in B} \sum_{w \in W} n\_bk\_conflict_r^w$$

, where $n\_bk\_conflict_r^w$ is the number of bank conflicts by shared memory accesses, in warp w, caused by reference r. It is represented in terms of warps for simplicity, similarly to bw_util, but analyzed per half warp. Finally, the efficiency of shared memory access is modeled as follows:

$$shm\_eff = \frac{n\_half\_warp \times n\_buffer}{n\_bk\_conflict} \quad (5)$$

, where n_half_warp and n_buffer denote number of half warps and number of shared memory buffers in a program.

5.7 Branch Divergence

Besides memory latency or bottleneck, one of the factors that affect performance most significantly is within-warp branch divergence. As explained in Section 2, the execution of threads is in SIMD manner. When threads in a warp take different execution paths, then all paths are serialized as shown in Figure 6.

Branches are introduced when there is uncovered region that is not buffered into shared memory. As shown at Line 14 and 19 in Figure 3, two branches are added in order to fetch data that could not be fetched into shared memory buffer. The other case where branches can be introduced is when the shared memory buffer size is not a multiple of the thread block size, but as discussed in the previous section, shared memory buffer size can often be adjusted to reduce or avoid bank conflicts. Therefore, we do not consider this case in this paper.

The penalty of serialized execution can be very different, even when the number of paths remains the same, according to the program structure. It is mainly because if the same memory reference is spread over different execution paths, then the accesses cannot be coalesced because each path is taken one after another. Therefore, the coding style of the kernel in Figure 3 is encouraged to achieve better performance, which is more accurately predictable by our approach. In the figure, new variables t1, t2, and t3 are introduced so that all memory references can happen in a synchronized way. The same kernel can be coded as shown in Figure 7. The first part of the code is omitted. Every memory access is duplicated on every path, which makes the number of memory requests 3 times more due to serialized execution. Also, note that the code in Figure 7, has the maximum number of paths taken in a warp of three while it is four in the code in Figure 3. Though having less number of divergent paths, the code in Figure 7 runs much more slowly. In this paper, we assume that programmers would not write a code in this way. Therefore, we simply model the impact of branch divergence as follows:

$$branch\_eff = \frac{n\_path}{n\_warp} \quad (6)$$

```
...

if (tIdx.x == bDim.x-2)
{
  out[row*MAX+col] = s_in[tIdx.y][tIdx.x] *
                     s_in[tIdx.y][tIdx.x+1] *
                     in[row*MAX+col+2];
}
else if (tIdx.x > bDim.x-2)
{
  out[row*MAX+col] = s_in[tIdx.y][tIdx.x] *
                     in[row*MAX+col+1] *
                     in[row*MAX+col+2];
}
else
{
  out[row*MAX+col] = s_in[tIdx.y][tIdx.x] *
                     s_in[tIdx.y][tIdx.x+1] *
                     s_in[tIdx.y][tIdx.x+2];
}
```

Figure 7. A worst case coding style for branch divergence

$$n\_path = \sum_{r \in R} \sum_{b \in B} \sum_{w \in W} n\_path_r^w$$

$$n\_path_s^w = \begin{cases} 2, & \text{if paths diverged in warp } w \text{ for } r \\ 1, & \text{otherwise} \end{cases}$$

, where n_warp is the number of warps in a program. To be in more detail, to check whether paths are diverged or not is done in this way: if some of addresses accessed by a given reference in a warp are mapped to shared memory buffers, while others not, then this not-perfect-coverage introduces branch divergence. CuMAPz checks this when analyzing data reuse.

5.8 Profitability Calculation

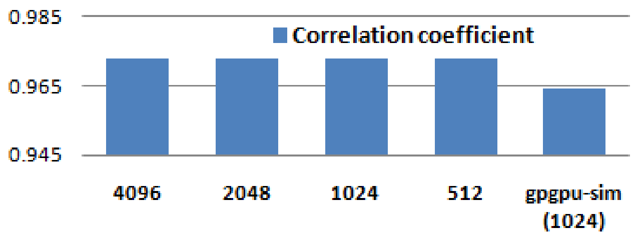
Now all the factors that we explained in the above are combined together to form *profitablity*. Profitability stands for how *relatively* beneficial the given parameters are for the overall memory performance of a program. Profitability is calculated by the following formula.

$$Profitability = data\_reuse \times \frac{bw\_util}{ch\_skew} \times \frac{branch\_eff}{\log(1/shm\_eff)} \quad (7)$$

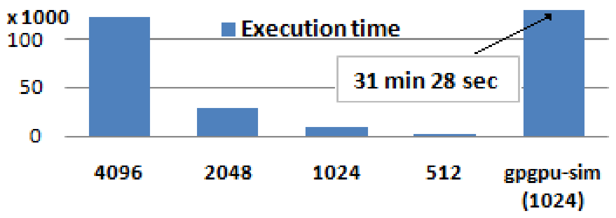
The first term represents the degree of data reuse. The second term represents the efficiency of accessing global memory. The logarithm of shm_eff is taken in order to reflect the relatively smaller impact of shared memory bank conflict on performance over other terms.

6. Experimental Results

To validate the effectiveness of our approach, we conducted experiments on how well profitability conforms to the real performance. We have implemented CuMAPz using C language. For a CUDA program, we have first made some sets of parameters that can change the performance. Then, we compared the profitability from CuMAPz and the runtime of the real code using the parameters of each set. Since runtime is the opposite concept of performance, we take the reciprocal of runtime, 1/runtime, and refer to it as performance. Both profitability and performance values are normalized in order to compare two sets of values in different scales.



(a) Correlation coefficients



(b) Average execution time for a set of parameter

Figure 8. Correlation coefficients and execution time for different input sizes

We used CUDA version 2.3 for all experiments. For CUDA-enabled GPU device, NVIDIA Tesla C1060 is used. It is of compute capability 1.3 and has 30 SMs and 4GB of memory. The host machine is Intel Core2 Duo E4500 with 3GB of memory. Both CUDA programs and CuMAPz ran on the same machine, on 64-bit Ubuntu linux 8.10.

We present results for four benchmarks. Two of them are Laplace edge enhancement and Wavelet transformation from benchmark suites in [2]. The other two of them are matrix multiplication and matrix transpose from CUDA SDK. The input sizes and thread block sizes are given on table 4. The Laplace loop is used to show the problem of what data should be fetched from an array to shared memory buffer. The Wavelet loop is used to describe the problem of how large the buffer size should be. Similarly to Laplace, matrix multiplication is also about which data should be buffered at larger granularity. The problem here is to decide which array to be fetched, not which data within an array. Lastly, matrix transpose is to show the problem of choosing the order of access and shared memory buffer dimension, very similar to the example in Section 3.

6.1 Complexity Considerations

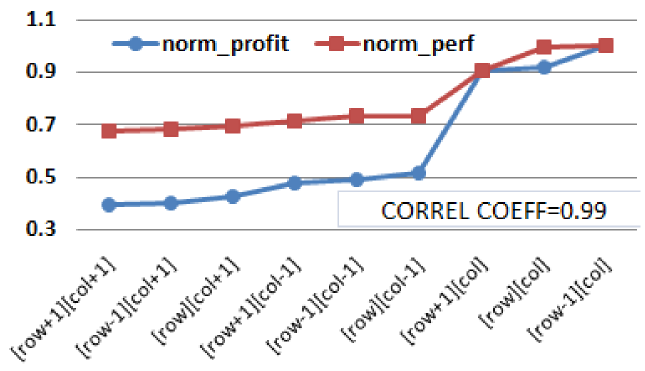
The analysis in CuMAPz is of quite high complexity. To be more specific, the complexity is:

$$O(n\_warp \cdot n\_gmem\_ref \cdot n\_buffer \cdot K)$$

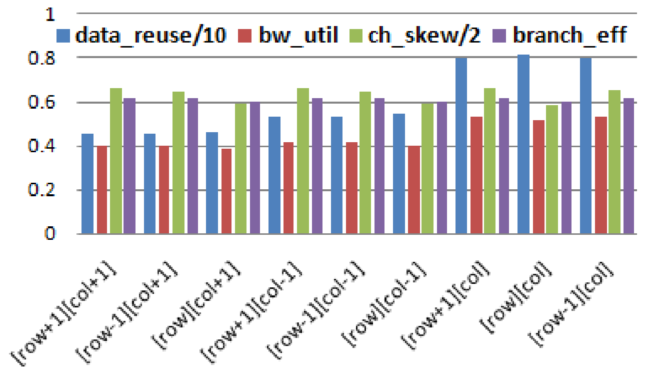
, where K is the complexity of the loop inside kernel, if there is any loop involving array references. n_warp, n_gmem_ref, and n_buffer are the number of warps, the number of global memory references, and the number of shared memory buffers, respectively.

Benchmark	Input array size	Block dimension
Laplace	8192x8192	16x16
Wavelet	8388608x2	128
matrix multiplication	1024x1024	16x16
matrix transpose	2048x2048	32x32

Table 4. The setup for each benchmark

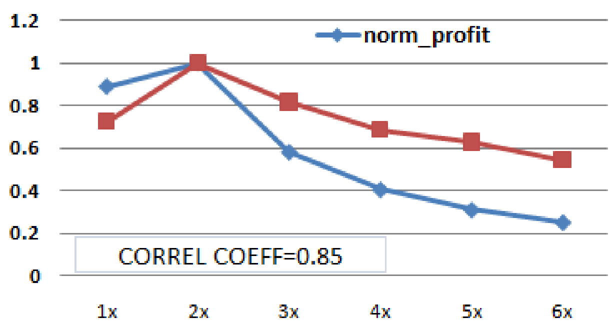


(a) Profitability and performance

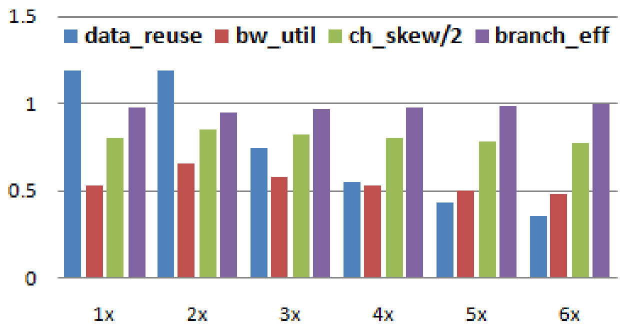


(b) Terms in profitability

Figure 9. Results for Laplace loop



(a) Profitability and performance



(b) Terms in profitability

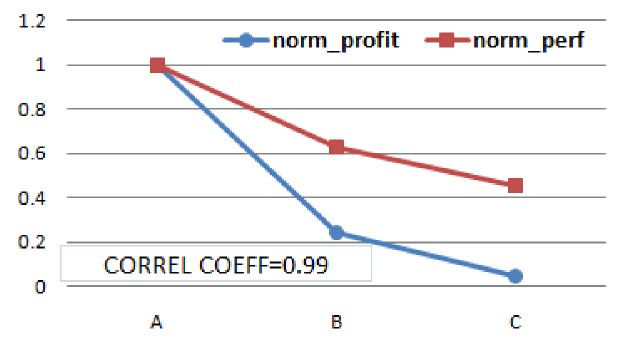
Figure 10. Results for Wavelet loop

The whole process can take quite long since it is basically serializing what was supposed to be done in parallel. However, profitability is a proportional metric. Except for the partition camping, all other terms are independent of input size. The ratio between shared memory access and global memory access remains the same for different input size. Also, as long as full warps are executed, the pattern of global memory coalescing and shared memory bank conflict do not change either. In other words, we can run the CuMAPz with smaller input size and still get the correct result. For partition camping analysis, we should at least maintain the number of blocks more than the number in Equation (3).

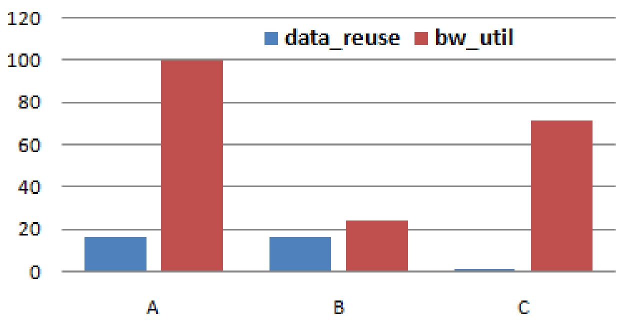
We compared the correlation coefficients between profitability numbers and performance numbers for various input sizes. Figure 8(a) shows the four results for Laplace benchmark, varying the array size from 4096x4096 to 512x512. The correlation coefficients stay almost the same as we reduce the input size while the execution times decrease dramatically. The difference between correlation coefficients are less than 0.001. We also show the correlation coefficient and the execution time of GPGPU-sim [15], a cycle-accurate GPGPU simulator, for the array size 1024x1024. Even though the difference is less than 0.01, the correlation coefficient of the results from GPGPU-sim is lower than the results from CuMAPz. GPGPU-sim even takes more than 30 minutes on average, while CuMAPz takes only about two minutes on average for 4096x4096 arrays and less than 1 minute on average for 512x512 arrays. Note that we are only showing the execution time of one case (a set of parameters), and we tested 9 different cases for Laplace benchmark. In other words, the total execution time is more than four hours for GPGPU-sim, while CuMAPz takes less than 20 minutes even with 16 times larger input size. Also note that GPGPU-sim requires a complete executable program that can run on GPUs as input, while GPGPU-sim can be run only with parameters even before writing a kernel.

6.2 Validation

Laplace loop has two arrays, and one of them, which is the input, has nine references. There is overlap between the regions covered by each reference. While we leave all other parameters untouched, we only change what data is fetched from global memory into shared memory by using each reference one at a time. The buffer dimension is the same as block dimension, and shared memory store function is in a row-major access. The input array size for CuMAPz is set to 256x256, and the execution time of CuMAPz is 0.97 seconds on average for each case. Figure 9(a) shows the comparison between normalized profitability and performance for each case. As shown in the figure, the curve of profitability values well conform the one of performance values. The correlation coefficient between two sets is 0.99. To see how profitability is calculated, we show the values of each term from Equation (6) in Figure 9(b). We do not normalize the values in this case, but in order to show the differences among each case more clearly, we put some weights on $data\_reuse$ and ch_skew as shown in the figure. The degree of data reuse has affected the profitability much as we can see the value becomes almost twice from the first case to the last case. However, the one with the highest degree of data reuse does not show the best performance mainly because of the bandwidth utilization, which varies from around 40% to over 50%. Bandwidth utilization is highest when the accesses by a fetch function are aligned, which applies only to these three cases: $[row-1][col]$, $[row][col]$, and $[row+1][col]$. Using one of these as a fetch function, a large part of uncoalesced accesses caused by other six refer-

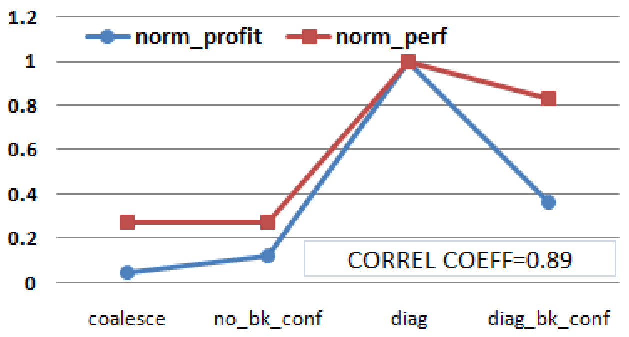


(a) Profitability and performance

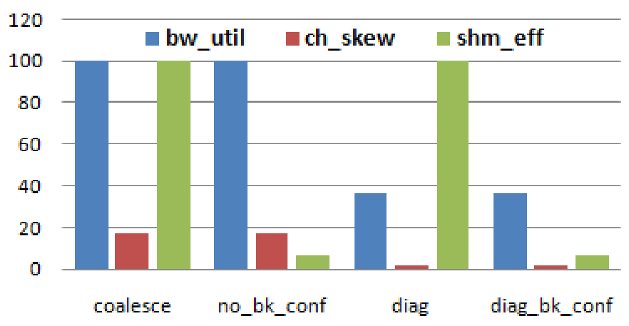


(b) Terms in profitability

Figure 11. Results for matrix multiplication



(a) Profitability and performance



(b) Terms in profitability

Figure 12. Results for matrix transpose

ences are removed and substituted with corresponding shared memory accesses. Shared memory bank conflict is not shown in the figure because there is no bank conflict in any case. The $branch\_eff$ stays almost the same over all cases.

Wavelet loop has three arrays in it, and one array which serves as an input has six references. Here we increase the buffer size, shared memory array size, starting from the same as block size to the sixth multiple of it. We model this situation in this way: each time we introduce a shared memory buffer whose size is the same as the thread block size. Each buffer is assigned one of the six references as a fetch function. Figure 10(a) shows the comparison between normalized profitability and performance for each case. The input array size for CuMAPz is set to 2^{17}, and the execution time of CuMAPz is 1.24 seconds on average for each case. We can see that the best performance can be achieved when the buffer size is the twice of the thread block size. It is because those six references in each thread access six consecutive array elements, and the starting address of memory accesses in each thread is in a stride of two. In other words, the buffer size of twice of the thread block size can employ data reuse the best. Because of the overlap between the regions covered by each reference, larger buffer size than this results in unnecessarily fetching duplicated elements. This can be shown in 10(b) as the degree of data reuse significantly reduces after the second case, even to less than one. The branch divergence reduces as buffer size increases, because a branch disappears as one reference is dedicated to one buffer. All data can be directly read from shared memory without selecting. Profitability can correctly predict the relative performance difference among all cases except for the first case. The correlation coefficient is 0.85.

As for matrix multiplication benchmark, we compare three cases of fetching one of three matrices (two for input, one for output) as a whole. Figure 11(a) shows the comparison between normalized profitability and performance for each case. The input array size for CuMAPz is set to 64x64, and the execution time of CuMAPz is 0.87 seconds on average for each case. We can see that CuMAPz can accurately predict the relative performance among three cases, showing the correlation coefficient of 0.99. In matrix multiplication code in CUDA SDK, the accesses to the first input matrix A are not coalesced at all. All threads in a half warp access exactly the same element at a time because they have to access the same row. This access pattern results in broadcasting of the data in devices of compute capability 1.3. The accesses for other two matrices, the second input matrix B and output matrix C, are completely coalesced. This is clearly shown in Figure 11(b) in that the bandwidth utilization for the first case is 100%. Since the thread block size is 16x16, the degree of data reuse is 16 in two cases of fetching A or B, while it is only 1 when C is fetched. We exclude the data for other terms because there is not any bank conflict, partition camping, and branch divergence.

We show the comparison results for the matrix transpose benchmark in Figure 12(a). The kernel has been used as a model to show how various parameters impact the performance in CUDA. We also use the same conversions done in [9]. First, the naive transpose kernel is converted to use shared memory so that uncoalesced global memory accesses become coalesced ones. Second, the shared memory bank conflicts are removed by the same technique we used in Section 3.2. This conversion does not bring much of performance improvement because the code is suffering from severe partition camping. Therefore, we remove partition camping using diagonal coordinates [9] in next case. Then, in the last case we again roll back the modification done in the second case to see the effect of shared memory bank conflict. This entire story is apparently shown in Figure 12(b). ch_skew values in the first two cases are very high, which can be interpreted as partition camping, while they are not for the other two cases. Bank conflicts are basically turned on and off for each case, switching 100% and 6.25% of shared memory efficiency. The input array size for CuMAPz is set to 512x512, and the execution time of CuMAPz is 1.98 seconds on average for each case.

7. Limitation

Our approach is basically in a form of static code analysis. Therefore, we cannot handle any information that can only be determined during run-time, such as dynamic allocated shared memory, indirect array accesses where the indices are loaded from memory, etc.

Also, we assume that adequate occupancy is achieved. Occupancy is a term that represents how many thread blocks can be scheduled on one SM so that the hardware is kept busy. Occupancy is determined by thread block size, shared memory array size, the number of registers used, etc. Low occupancy leads to significant performance degradation. However, NVIDIA suggests that 50% of occupancy, thus 50% of the maximum number of in-flight threads that a SM can have, is often enough to achieve the best performance [7]. We do not consider the effect of occupancy in this paper.

Another limitation is that we are not able to identify the bottleneck of a given program. For example, let us say the bottleneck of one program is in computation time, not in memory load and store time. Then, the change in memory access pattern may not affect the performance of the program. Thus, even if profitability values are significantly different for different input parameters, the actual runtimes may remain unchanged. However, having a higher profitability means better memory efficiency, which would never harm the performance. Thus, it should still be useful to check the efficiency of memory access pattern.

Lastly, profitability cannot be used to compare performance between two different kernels. It is only a relative metric, and this is because no information about the program structure or instructions is considered in the metric. However, the use of profitability lies in helping programmers decide parameters regarding memory access, by providing a method to estimate the performance of using those parameters, before even writing and running a program.

8. Conclusion and Future Work

GPGPUs are affordable yet powerful high performance computing platforms. It is, however, very hard to optimize a program to exploit the best performance, due to complex memory hierarchy and many details in programming model to be considered. Especially, the memory efficiency affects the program performance dominantly, so being able to choose more beneficial design parameters on memory access pattern is crucial in optimizing GPGPU programs. In this paper, we present a tool, CuMAPz, to analyze the efficiency of memory access pattern of a program, which can guide developers to decide design better design parameters without even writing a kernel. Given a set of parameters on memory access pattern, all well-known performance critical factors, such as data reuse, memory access coalescing, bank conflicts, etc. are analyzed thoroughly. The tool outputs profitability, a relative performance metric introduced in this paper. The experimental results show that profitability can accurately and efficiently estimate the performance of applications. As for future work, we would like to work on design space exploration to find the best parameters. As a continuing effort to relieve the burden on GPGPU developers, our final goal should be to automate the optimization of a program. Also, the hybrid memory hierarchy, which has both L1 cache and scratchpads, in new NVIDIA Fermi architecture, is of our great interests.

References

[1] Panda, P. R., Dutt, N. D., and Nicolau, A. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the 1997 European Design and Test Conference (ED&TC)*. March, 1997.

[2] Kolson, D. J., Nicolau, A., and Dutt, N. Elimination of redundant memory traffic in high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. Vol.15, No.11, November, 1996.

[3] Kandemir, M. and Choudhary, A. Compiler-directed scratch pad memory hierarchy design and management. In *Proceedings of the 39th Annual Design Automation Conference (DAC)*. June, 2002.

[4] Issenin, I., Brockmeyer, E., Miranda, M., and Dutt, N. Data reuse analysis technique for software-controlled memory hierarchies. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. February, 2004.

[5] Kothapalli, K., Mukherjee, R., Rehman, M.S., Patidar, S., Narayanan, P.J., Srinathan, K. A performance prediction model for the CUDA GPGPU platform. In *Proceedings of The 16th IEEE International Conference on High Performance Computing (HiPC)*. December, 2009.

[6] Hong, S. and Kim, H. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*. June, 2009.

[7] NVIDIA Corporation. *NVIDIA CUDA Programming Guide, Version 2.3.1*.

[8] NVIDIA Corporation. *Board Specification, Tesla C1060 Computing Processor Board*. `http://www.nvidia.com/docs/IO/56483/Tesla_C1060_boardSpec_v03.pdf`.

[9] Ruetsch G. and Micikevicius P. *Optimizing Matrix Transpose in CUDA*. NVIDIA. 2009.

[10] Yang, Y., Xiang, P., Kong, J., and Zhou, H. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. June, 2010.

[11] Baskaran, M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proceedings of the 22nd annual International Conference on Supercomputing (ICS)*. June, 2008.

[12] Ueng, S., Lathara, M., Baghsorkhi, S. S., and Hwu, W. W. CUDA-Lite: reducing GPU programming complexity. In *Proceedings of the 21th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. July, 2008.

[13] Leung, A., Vasilache, N., Meister, B., Baskaran, M., Wohlford, D., Bastoul, C., and Lethin, R. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*. March, 2010.

[14] Baskaran, M., Ramanujam, J., and Sadayappan, P. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the International Conference on Compiler Construction (CC)*. March, 2010.

[15] Bakhoda, A., Yuan, G. L., Fung, W. W. L., Wong, H., Aamodt, T. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. April, 2009.

[16] Ryoo, S., Rodrigues, C. I., Stone, S. S., Baghsorkhi, S. S., Ueng, S., Stratton, J. A., and Hwu, W. W. Program optimization space pruning for a multithreaded gpu. In *Proceedings of the 6th Annual IEEE/ACM international Symposium on Code Generation and Optimization (CGO)*. April, 2008.

[17] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W. W. 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)*. February, 2008.