Memory Performance Estimation of CUDA Programs

YOOSEONG KIM and AVIRAL SHRIVASTAVA, Arizona State University

CUDA has successfully popularized GPU computing, and GPGPU applications are now used in various embedded systems. CUDA programming model provides a simple interface to program on GPUs, but tuning GPGPU applications for high performance is still quite challenging. Programmers need to consider numerous architectural details, and small changes in source code, especially on memory access pattern, can affect performance significantly. This makes it very difficult to optimize CUDA programs. This paper presents CuMAPz, which is a tool to analyze and compare the memory performance of CUDA programs. CuMAPz can help programmers explore different ways of using shared and global memories, and optimize their program for efficient memory behavior. CuMAPz models several memory-performance related factors: data reuse, global memory access coalescing, global memory latency hiding, shared memory bank conflict, channel skew, and branch divergence. Experimental results show that CuMAPz can accurately estimate performance with correlation coefficient of 0.96. By using CuMAPz to explore memory access design space, we could improve the performance of our benchmarks by 30% more than the previous approach[Hong and Kim 2010].

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

General Terms: Performance, Design, Experimentation

Additional Key Words and Phrases: GPGPU, CUDA, Memory performance, Program optimization, Performance estimation

ACM Reference Format:
Kim, Y., Shrivastava, A. 2011. Memory Performance Estimation of CUDA Programs. ACM Trans. Embedd. Comput. Syst. 9, 4, Article 39 (March 2010), 23 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The computational power of modern Graphics Processing Units (GPUs) has been rapidly increasing, and has now reached teraFLOP scale. Traditionally, GPGPU (General Purpose computation on GPUs) programming required a high level of expertise and proficiency, but NVIDA CUDA [NVIDIA 2010b] and OpenCL [OpenCL] have successfully lowered the entry barrier of writing GPGPU codes. GPGPU applications are now used in various embedded systems, including military, aerospace and medical applications [GE Intelligent Platforms ; TechniScan]. Recently, GPGPU has gained much attention in mobile systems due to its high power efficiency. Low power GPUs such as ARM Mali [ARM] and NVIDIA ION [NVIDIA b], support OpenCL or CUDA, invigorating GPGPU applications in embedded systems.

In spite of high compute power of GPUs, fully utilizing the potential of the hardware is challenging. Even if the application is compute-intensive, the performance is significantly affected by memory performance, because GPGPU applications typically

Authors are with the Compiler Microarchitecture Lab at Arizona State university, Tempe, AZ 85281. Author's email addresses: yooseong.kim@asu.edu

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

have large data sizes [1]. Massive parallel architecture and complex memory hierarchy of GPUs have introduced several performance optimization considerations [NVIDIA 2010a; Ryoo et al. 2008; Ruetsch and Micikevicius 2009]. Many of these factors are related to the way of using shared memory and global memory, such as how to access global memory, what to be fetched in shared memory, shared memory buffer size. Programmers need to consider all of these performance-critical factors carefully since a slight change in source code might result in much slower, or faster, execution time. 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 [Ryoo et al. 2008]. 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 problem of improving application performance by using shared memory. Shared memory is an explicitly-managed scratchpad memory of a small size (e.g. 16KB in Tesla[NVIDIA a]). Using shared memory is vitally important in optimizing CUDA programs, because the global memory is not cached and is bandwidth-constrained. On the other hand, shared memory is as fast as registers, and is the only *fast* memory where both read and write accesses are enabled. Other memories, such as texture memory and constant memory, are read-only [2]. Even in this small subset of the problem, optimizing program performance is not trivial. For example, traditional data reuse analysis for explicitly managed scratchpads is not enough for shared memory in GPUs. This is because slow global memory can often become a bottleneck when there is uncoalesced accesses or partition camping [Ruetsch and Micikevicius 2009], leading to counter-intuitive results. Shared memory access pattern is also an important factor to be considered since it can cause bank conflicts or branch divergence. Even a slight change in source code can create a bottleneck, and result in drastic changes in the performance. To address this problem, we present a tool: CuMAPz (CUDA Memory Access Pattern analyZer), which estimates the memory performance of a CUDA program. Since our approach can quantify the performance impact of each of performance critical factors, programmer can find a possible bottleneck and try to improve the corresponding part of the code. Programmers can also use our approach to explore different ways of using shared and global memories, and optimize their programs for memory performance.

This is a comprehensive approach to analyze the memory performance of programs, covering all the well-known factors such as the degree of data reuse, global memory latency hiding, global memory access coalescing, shared memory bank conflict, channel skew, and branch divergence. Our experiments on benchmarks from [Kolson et al. 1996] and Rodinia benchmark suite [Che et al. 2009], and kernels from CUDA SDK show that CuMAPz can accurately predict the relative performance of various ways to use shared and global memories, with average correlation coefficient of 0.96. By using CuMAPz to explore shared and global memory use design space, we could improve the

[1]Computational performance still affects the overall performance, and there have already been researches on it [Hong and Kim 2010; Ryoo et al. 2008]. This work, on the other hand, focuses on memory performance which has not been studied as much.

[2]There is also local memory, which is the same as global memory and only used for register spilling and local arrays. Local memory data is private to each thread, so local memory accesses are always coalesced [NVIDIA 2010b] and thus efficient. A programmer or a compiler cannot control the local memory access behavior except the size of local memory data usage per thread. Therefore, we do not consider local memory in this paper.

performance of our benchmarks by 30% more than the previous approach[Hong and Kim 2010].

2. BACKGROUND

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 [NVIDIA 2010b] 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 Program Execution 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). Each SM can have thousands of threads in-flight. 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, so the basic unit of scheduling in SMs is a thread block. 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 id's 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. This lock-step execution introduces a lot of overhead in the execution of branches because every path has to be serialized, which means every cores must execute both branch paths.

The number of in-flight threads in one SM is affected by many factors, such as the number of thread blocks and threads assigned to the SM, the number of used registers, and the size of shared memory buffers. Since threads are scheduled in a granularity of a thread block, the number of in-flight threads can be much less than the maximum possible. *Occupancy* [NVIDIA 2010b] denotes the ratio of the number of current active threads to the maximum number of active threads, which basically means how busy the program makes the cores.

2.2. Global Memory and Shared Memory

CPU code can only transfer data to this off-chip memory, called global memory, so basically all data resides in global memory. Global memory is not cached in devices of compute capability 1.x, and the latency of accessing global memory is hundreds of cycles. However, this latency can often be hidden with the help of having a large number of threads in-flight (thread level parallelism) and subsequent independent instructions (instruction level parallelism). Thus, the independent instructions of this warp or other warps 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 the bus between global memory and the SMs is quite wide (e.g., 512 bit wide for Tesla C1060 [NVIDIA a]) the massively parallel execution can easily saturate

```
1  int row = blockIdx.y*blockDim.y+threadIdx.y;
2  int col = blockIdx.x*blockDim.x+threadIdx.x;
3
4  if (col >= MAX-2)        // array boundary check
5    return;
6
7  out[row*MAX+col]  = in[row*MAX+col] *
8                      in[row*MAX+col+1] *
9                      in[row*MAX+col+2];
```

Fig. 1. *A simple CUDA program which does not use shared memory. We will optimize this program to use shared memory. bIdx, bDim, and tIdx are respectively abbreviations for blockIdx, blockDim, and threadIdx. $in[]$ and $out[]$ are two dimensional arrays with size MAX*MAX. Array element type is float for both arrays. MAX is defined as 16384*16384.*

the bandwidth, and this often becomes the performance bottleneck in CUDA programs. Global memory access coalescing and channel skew, so called partition camping in [Ruetsch and Micikevicius 2009], are dominant factors in bandwidth utilization. The details of memory coalescing and channel skew will be discussed in Section 5.

Shared memory is on-chip memory, with latency equal to that of registers. It is an software-controlled memory which is often used as a local buffer for fast retrieval of data. Similarly to scrachpad memory [Issenin et al. 2004], it is important to keep frequently accessed data in the shared memory. Thus, the degree of data reuse is an important performance consideration but not the only. All data are first present in global memory, so data must be prefetched from global memory to shared memory. Since prefetching data to shared memory must incur global memory accesses which can be very slow as fore-mentioned, the global memory load instructions for prefetching data often dominate memory performance. Shared memory is on-chip, the bandwidth is not a performance limiting factor, but bank conflicts can often slow down a program. We will discuss more details about this in Section 5.

3. MOTIVATING EXAMPLE

In this section, we start from a very simple CUDA program shown in Figure 1 and try to optimize its performance by using shared memory. We show that performance optimization of CUDA programs is not trivial, and can be counter-intuitive – mainly because there are many factors that affect memory performance, and the effect of all of them need to be considered simultaneously.

3.1. Question 1: What to fetch into shared memory?

Shared memory is a fast local buffer, and therefore intuition suggests that keeping the most frequently used data in the shared memory should improve performance. In fact, higher data reuse should imply better performance. In the CUDA example in Figure 1, there are three memory references, `row*MAX+col`, `row*MAX+col+1`, and `row*MAX+col+2`. Table I shows that if we fetch `row*MAX+col+1` to the shared memory (code shown in Figure 2), then out of a total 805208064 accesses to the array $in[]$, 771670016 accesses are found in the shared memory. Table I shows that among the three options for fetch functions, the reference `row*MAX+col+1` has the largest reuse. This should imply that fetching `row*MAX+col+1` should achieve the best performance. However, this is not the case. The third column in Table I shows the execution time (in ms) of prefetching the references on NVIDIA Tesla C1060. It shows that prefetching the reference `row*MAX+col` is best. Even the last case with smallest data reuse is slightly faster than the second case. This counter-intuitive result is mainly caused by global memory access coalesc-

```
1  int row = blockIdx.y*blockDim.y+threadIdx.y;
2  int col = blockIdx.x*blockDim.x+threadIdx.x;
3
4  float t1, t2, t3;
5
6  __shared__ float s_in[BLKDIM][BLKDIM];
7  s_in[tIdx.x][tIdx.y] = in[row*MAX+col+1];
8  __syncthreads();
9
10 if (tIdx.x == 0)          // this part not prefetched
11   t1 = in[row*MAX+col];
12 else
13   t1 = s_in[tIdx.x-1][tIdx.y];
14
15 t2 = s_in[tIdx.x][tIdx.y];
16
17 if (tIdx.x == bDim.x-1)   // this part not prefetched
18   t3 = in[row*MAX+col+2];
19 else
20   t3 = s_in[tIdx.x+1][tIdx.y];
21
22 out[row*MAX+col] = t1*t2*t3;
```

Fig. 2. *Some data is fetched into a shared memory buffer. Now, data that can be read from the buffer is read from shared memory to exploit data reuse. BLKDIM denotes the block dimension which is 16 here.*

ing. Global memory access coalescing is the phenomenon in which hardware automatically combines accesses to contiguous memory to a fewer number of large memory access. We model the effects of global memory coalescing in our performance estimation.

Table I. Prefetching Different Data into Shared Memory

	Data reuse	**Execution time (*in ms*)**
not using shared mem	-	78.15
`row*MAX+col`	754925568	61.11
`row*MAX+col+1`	771670016	64.86
`row*MAX+col+2`	754876416	63.77

We can improve the performance around 20% by using shared memory, but counter-intuitively, the case with highest degree of data reuse, which is shown in the second column as the number of times fetched elements in buffer are read, exhibits the worst execution time.

3.2. Question 2: How to access shared memory?

How data is stored to and read back from shared memory also significantly affects performance. In Figure 2, data is accessed in a column-wise manner, as shown at Line 4, 9, 11, and 16. Another option would be to store and accesses in a row-wise manner (i.e., `s_in[tIdx.y][tIdx.x]`). The second and third columns in Table II compares the execution time of the two options, and shows that row-wise access performs better. The fourth column shows that similar affect is achieved by skewing the access pattern in the form: `__shared__ float s_in[BLKDIM][BLKDIM+1]`. These variations in execution

Table II. Changing the Way to Access Shared memory

	Execution time (*in ms*)		
	Column-wise	**Row-wise**	**BLKDIM+1**
`row*MAX+col`	61.11	45.06	44.98
`row*MAX+col+1`	64.86	54.75	53.69
`row*MAX+col+2`	63.77	55.25	54.39

Changing the way to access shared memory turns up more improvement about 26% in the best case shown at the top row. Note that the performance improvement is limited in other cases where the observation from the Section 3.1 works as a bottleneck.

time are because of shared memory bank conflicts. Shared memory bank conflicts occur if there are multiple requests to different addresses in the same bank. In this case, the requests are serialized. CuMAPz models this effect.

3.3. Question 3: How to access global memory?

Here we present another example of a small change in code causing a significant difference. A programmer might have designed the global memory write reference at Line 18 in Figure 2 to be in a column-wise manner as in `out[col*MAX+row]`. This would have resulted in disastrous performance as shown in Table III. This unexpected slowdown is caused by channel skew. Channel skew is the ratio of the number of concurrent accesses to the most used channel to the least used channel. CuMAPz models this effect also.

Table III. Changing the Way to Access Global Memory

	Execution time (*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

A slight change in a way to access global memory unexpectedly brings drastic performance degradation. Note that there is hardly any benefit of using shared memory. Shared memory accesses kept the same in a row-wise manner as we changed in the Section 3.3.

4. RELATED WORK

Traditional data reuse analysis, such as [Issenin et al. 2004] is no longer the answer for finding better ways to use shared memory in GPUs since there are many other factors that affect performance significantly. To understand the details of architecture better and help programmer optimize applications, there have been many researches on developing analytical performance models. Ryoo et al. [Ryoo et al. 2008] modeled the amount of parallelism employed in a program and the efficiency of a single kernel execution in a thread. However, they did not consider memory performance and their analysis is only for compute intensive benchmarks. Hong and Kim [Hong and Kim 2010] proposed an analytical performance model that includes the effect of parallelism to hide global memory access latency. Their model, however, does not take into account branch divergence. This means that data reuse cost caused by prefetching only part of accessed data and global memory access coalescing effect caused by taking serialized execution paths cannot be modeled. Latency hiding, shared memory bank

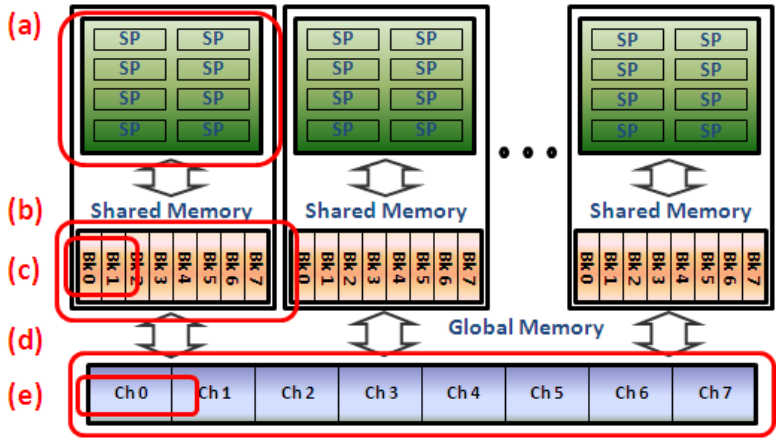


Fig. 3. *CuMAPz comprehensively analyzes the performance critical factors in the architecture: (a) Branch divergence in SIMD cores and latency hiding, (b) Data reuse, (c) Shared memory bank conflict, (d) Global memory coalescing and latency hiding, and (e) Channel skew.*

conflicts and channel skew are not considered either. More recently, Zhang and Owens [Zhang and Owens 2011] presented an quantitative performance model which models the throughput of instruction execution, shared memory access, and global memory access by considering occupancy, memory access pattern, and types of instructions. This work however cannot model more detailed performance issues caused by memory access pattern such as latency hiding, channel skew and branch divergence.

While the previous researches have done extensive work on analyzing instruction execution performance, only little attention has been paid to memory performance. We focus on comprehensive memory performance analysis. Since we do not consider instruction execution performance, it is not possible to use our model to compare performance of different kernels. However, as a detailed analysis of memory performance, our approach can analyze all performance critical factors and identify the bottleneck of a kernel. Then, programmer can try different ways to optimize the given program, such as different buffer size, different memory references, different block ordering, etc.. Thus, our approach can identify the best design choice for performance among many optimization choices.

As a more aggressive solution to relieve the burden on programmers, there have been efforts to automate optimization of GPGPU applications. Ueng et al. [zee Ueng et al. 2008] 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. [Baskaran et al. 2008] 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 [Baskaran et al. 2010]. Later, another work [Leung et al. 2010] 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. However, in both approaches, 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 channel skew is not considered as well. Yang et al. [Yang et al. 2010] presented another optimizing compiler. Their approach takes into account most of the factors that we consider in this paper except for the branch divergence.

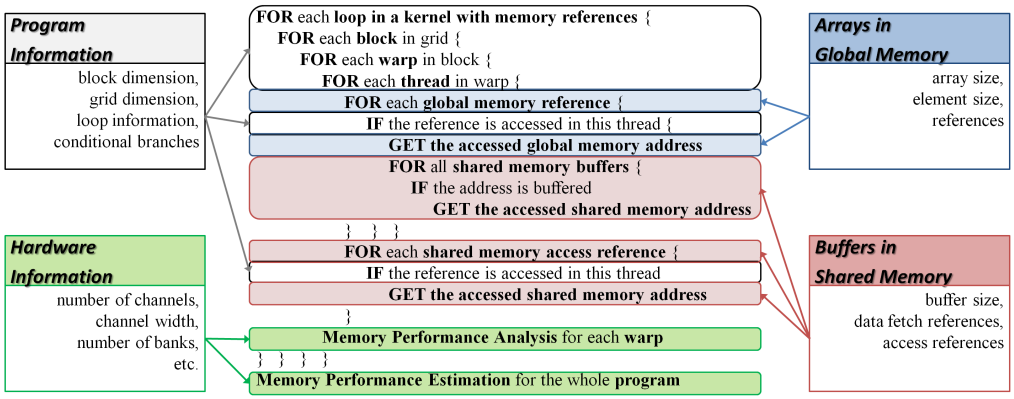


Fig. 4. *Overview of CuMAPz analysis. Using information about the input program, CuMAPz generates a loop structure that emulates memory accesses of the program. Examining the addresses and the order of accesses of each warp, CuMAPz analyzes the memory performance of the given program on the target hardware.*

In all of the above work, an optimizing compiler can only find the worst case scenario for each performance factor they consider and just avoid it. On the other hand, we can quantify the performance impact of each of factors, which enables to even detect the case when it deteriorates performance *slightly*. Moreover, our analysis can analyze the combined effect of all factors. For example, there are cases where the best performance can be achieved when some factors are affecting performance favorably while others are not. We can estimate the performance impact considering all factors together.

5. OUR APPROACH

Figure 3 summarizes the analyses of CuMAPz. It covers all performance-critical effects in the architecture: a) branch divergence in SIMD cores incurred by shared memory usage and latency hiding by simultaneous multithreading, b) data reuse of data fetched in shared memory buffer, c) bank conflicts in shared memory accesses, d) coalescing in global memory accesses, and e) channel skew in global memory accesses.

Figure 4 shows an overview of how CuMAPz works. Given a program, programmer gives information about the program: thread and grid dimension; memory references in the program; loops and branches that have memory references; and shared memory references. The detailed information about the target hardware, such as number of channels, channel width, and number of banks, is also needed, which can be obtained from the device specification sheet.

Using the given thread block size and grid size information, CuMAPz constructs a loop structure that exactly follows the execution of each warp in a block. If there are loops or branches that contain memory references in, they are also included to emulate the execution of threads on hardware. Then, memory references, which are functions of block index, thread index, and loop indices, are plugged in the loop. We essentially generate a trace of all memory addresses accessed in each warp.

For shared memory usage, we take two kinds of references as input: data fetch references and access references. Data fetch references are the global memory references to fetch data from global memory to shared memory buffers, and access references are shared memory references. For example, the right hand side of Line 7 in Figure 2 shows a data fetch reference, and the left hand side an access reference. This pair creates a mapping between global memory addresses and shared memory addresses. Each buffer will have the mapping information. As CuMAPz generates memory addresses that are accessed using the loop and global memory references, it checks whether the

addresses are buffered in shared memory. If so, it generates shared memory addresses using the mapping. Data fetch reference is always paired with an access reference. Some shared memory arrays are not used as buffers for global memory but as temporary storage. These arrays will not have data fetch references and only have access references. Using these access references that are not paired with data fetch references, CuMAPz can directly generate shared memory addresses without using global memory references first.

Then, examining the addresses accessed in each warp and the order of accesses, CuMAPz analyzes 1) how many times data in shared memory is accessed (data reuse), 2) if global memory latency can be hidden, 3) how global memory accesses are coalesced, 4) if global memory accesses are not skewed to some of memory channels, 5) if shared memory accesses generate any bank conflict, and 6) if the use of shared memory introduces a branch. All these factors are explained in detail in next section. It is possible that the same analysis can be done by some static analysis, but we leave this as future work.

For a given program, CuMAPz will output a performance estimation with all quantified performance impact of each of the above factors. From the quantified performance impact, the user can find which part of program needs to be improved. For example, a program shows a very poor bandwidth utilization and shared memory access efficiency, then the programmer will try to change global memory references and shared memory access references to change the corresponding behavior. Also, when a programmer already has a clear picture about how the program will be with several design choices about memory access, CuMAPz can be used to instantly compare performance of all choices without actually writing or modifying the program. This is because CuMAPz does not take a whole program or a PTX code as input but the parametrized input on memory access shown in Figure 4.

5.1. Data Reuse Profit Estimation

For any global memory access, we check whether the corresponding access is buffered in shared memory or not. This can be done by using the mapping created by shared memory data fetch reference and access reference given as input. The corresponding shared memory address can be obtained by the mapping. CuMAPz maintains a counter to count the number of times shared memory buffers are accessed. During the iteration of the loop for each warp shown in Figure 4, if a global memory address accessed by a global memory access references is already mapped to one of buffers within the same block, then the counter is increased. The degree of data reuse is represented in a term, *data_reuse*, which can be modeled as follows:

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

where *bytes_buffered* and *bytes_shmem* are defined as the following.

$$bytes\_buffered = \sum_{b \in B} \sum_{w \in W} bytes\_tr_b^w$$

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

where R, B and W denote 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 fetching data from global memory 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 concept of the bytes transferred in the term *bytes_tr* is detailed in Section 5.3.

5.2. Latency Hiding Profit Estimation

Global memory latency is hundreds of cycles, but both instruction level parallelism and thread level parallelism can help hiding the latency. Instead of stalling until data arrives, the cores can switch to other warps or the subsequent independent instructions. Thread level parallelism can be simply modeled by occupancy [NVIDIA 2010b]. For instruction level parallelism, we consider the number of load instructions for shared memory data prefetch, i.e. the number of data fetch references for shared memory buffers. Obviously, load instructions for shared memory data prefetch are all independent, and executing them in a row can hide some of the latency of global memory accesses. [3] All dependent instructions will start after data is loaded into shared memory. For example, the code snippet shown below, excerpt from matrix multiplication kernel, has two prefetch functions. While the data for the first shared memory buffer sA is being delivered, the load instruction for the buffer sB can be executed for all active warps.

```
__shared__ float sA[BLK_SIZE][BLK_SIZE];
__shared__ float sB[BLK_SIZE][BLK_SIZE];

sA[threadIdx.y][threadIdx.x] = A[a + wA*threadIdx.y + threadIdx.x];
sB[threadIdx.y][threadIdx.x] = B[b + wB*threadIdx.y + threadIdx.x];
__syncthreads();
```

It is generally known that once occupancy reaches 50%, there is no more performance benefit [NVIDIA 2010a]. Therefore, we only model the negative impact when occupancy is less than 50%. The degree of latency hiding is modeled as follows:

$$\boldsymbol{lat\_hiding} = \frac{MIN(\boldsymbol{Occupancy}, 50)}{50} \cdot \boldsymbol{n\_shmem\_load}^{1/2}, \tag{2}$$

where *Occupancy* is the occupancy in percentile, and *n_shmem_load* denotes the number of load instructions for shared memory data fetch. Both the number of shared memory buffers and the number of data fetch references for each buffer should be taken into account. The square-root of *n_shmem_load* is taken empirically by curve-fitting. However, this can be changed for different generations of architecture where the importance of instruction level parallelism is much more important. [Volkov 2010]

5.3. Coalesced Access Profit Estimation

When address accessed by a global memory reference is not mapped to any buffer, global memory access occurs. When memory accesses from threads in each *half warp*, which is a group of 16 consecutive threads, are aligned and not strided, the actual transactions in memory can be coalesced to reduce the number of transactions. The transactions are of 32-byte, 64-byte, or 128-byte. Since coalescing specifies the granularity of data transfer, the size of data actually transferred from memory can be different from the size of data requested. Thus, some unused data can be transferred, which results in inefficient bandwidth utilization. Figure 5(a) shows an example where coalescing behavior incurs low bandwidth utilization (The figure is based on compute capability 1.3[NVIDIA 2010b]). The coalescing behavior may be different for different

[3]Independent ALU instructions can also hide the latency, but we only consider memory operations in this paper. Since CuMAPz does not take the whole program as input, it is basically not possible to model latency hiding from executing independent ALU instructions. We leave this as future work.

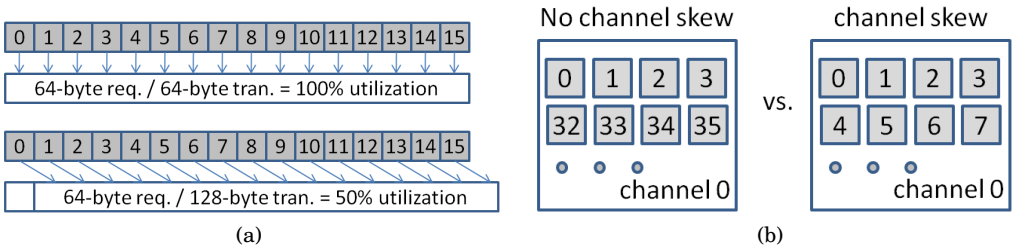


Fig. 5. *(a) Shaded boxes depict threads, and clear box represents memory region. In the upper example, the accesses are perfectly consecutive and aligned to the 64-byte segment, while the accesses are misaligned in the lower example. Bandwidth utilization differs significantly. (b) Shaded boxes denote addresses requested by thread blocks whose id's are written in the boxes. Each outer frame represents a memory channel. Thread blocks with consecutive ids are likely to be executed concurrently. On the right hand side, accesses of consecutive thread blocks are all focused on only one channel, introducing significant channel skew.*

generation of architecture [NVIDIA 2010b], but similar analysis can be applied without loss of generality. [4]

CuMAPz calculates the bandwidth utilization as the following:

$$\boldsymbol{bw\_util} = \frac{bytes\_access}{bytes\_tr} \tag{3}$$

$$bytes\_access = \sum_{r \in R} \sum_{w \in W} bytes\_access_r^w$$

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

where $bytes\_access_r^w$ and $bytes\_tr_r^w$ are the size of accessed (requested) data and actual transferred data, respectively, for reference r in warp w. It is represented in terms of warps for simplicity, but the actual analysis is done at the half warp granularity.

5.4. Channel Skew Cost Estimation

The memory subsystem in CUDA consists of multiple channels, and all of channels can transmit data at the same time in parallel. Channel skew refers to the case where the concurrent accesses are skewed to only a few of channels as shown in Figure 5(b). In other words, channel skew occurs when global memory addresses that are requested by all cores, at one point of time, are not evenly distributed to all the channels but focused on only one channel of two channels. This essentially makes the bus width much narrower on top of the slow latency of global memory. This phenomenon is also called partition camping [Ruetsch and Micikevicius 2009].

To analyze channel skew, we need to know on which channel the concurrent accesses are mapped. This makes the analysis tricky because to determine which blocks are being executed concurrently is impossible. Initially, when a kernel 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 [Ruetsch

[4]Similar analysis can be applied even in the presence of caches on devices with compute capability 2.x. Cache line is of 128 bytes, so even if some part of memory can be cached, it is very likely that the accesses from other warps will still incur cache misses. (The number of warps in an SM is usually at least 8 and 32 at maximum, which makes the size of data required from an SM to be 1024 to 4096 bytes in case of 4-byte word size.) Moreover, since cache line maps to 128-byte aligned segment in global memory, misaligned and strided access will fetch unused data, which will harmfully affect performance.

and Micikevicius 2009]. Our channel skew analysis is basically to estimate the span of initial round of concurrent memory accesses, and to see if those accesses are not skewed to only part of channels.

When the channel width is X-byte, successive X-byte regions of global memory are mapped to successive channels. For example, channel width is 256-byte in Tesla C1060, and adjacent 256-byte regions are mapped to successive channels in that device. Let us consider the initial set of blocks that are assigned to SMs when kernel is launched. A global memory access instruction is likely to be executed by all threads in those blocks at the same time due to simultaneous multithreading (SMT) of the hardware. The addresses accessed by warps from successive blocks will be $bDim.X \times elem\_size$ bytes apart, where $bDim.X$ denotes the X block dimension and $elem\_size$ is the maximum element size of global memory arrays. Then, we can heuristically calculate the number of blocks to initially use all the channels as the following.

$$n\_blk\_to\_check = n\_channels \times MIN(max\_blks, \frac{channel\_width}{bDim.X \times elem\_size}) \tag{4}$$

, where max_blks is the maximum number of blocks that can run concurrently in one SM for the given kernel, which is similar to occupancy except that occupancy is a ratio. When the number of all thread blocks in the kernel is smaller than the number calculated by the above, this analysis is skipped.

The impact of channel skew can be modeled as the skewness of mapping between channels and global memory addresses requested during the execution of the above number of thread blocks, which can be calculated as follows:

$$\boldsymbol{ch\_skew} = \frac{max\_n\_block\_per\_ch}{MAX(1, min\_n\_block\_per\_ch)} \tag{5}$$

, where $max\_n\_block\_per\_ch$ and $min\_n\_block\_per\_ch$ denote, respectively, the maximum and minimum nonzero number of blocks assigned to a channel for the initial $n\_blk\_to\_check$ number of blocks, which is acquired from Equation 4. When $max\_n\_block\_per\_ch$ and $min\_n\_block\_per\_ch$ are the same, which means all the accesses are focused on one channel, ch_skew is just set to $n\_channels$ to represent severe channel skew.

5.5. Bank Conflict Cost Estimation

Similarly to global memory channels, shared memory is divided into multiple 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.

For example, 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 conflict. The execution time 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 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, but analyzed per half warp. Finally, the efficiency of shared memory access is modeled as follows:

$$\boldsymbol{shm\_eff} = \frac{n\_half\_warp \times n\_buffer}{n\_bk\_conflict} \tag{6}$$

where *n_half_warp* and *n_buffer* denote number of half warps and number of shared memory buffers in a program. When *n_bk_conflict* is zero, *shm_eff* is set to one.

5.6. Branch Divergence Cost Estimation

Besides memory latency or bottleneck, one of the factors that affect performance most significantly is within-warp branch divergence. The execution of threads is in SIMD manner. When threads in a warp take different execution paths, then all paths are serialized. Branches are introduced when there is uncovered region that is not fetched into shared memory, as shown at Line 6 and 13 in Figure 2. 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. [5] The existing branches given as input are considered in this analysis since we intend to focus on the performance impact of memory usage, i.e. how shared memory usage can introduce branch divergence.

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 2 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 6. 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 6, has the maximum number of paths taken in a warp of three while it is four in the code in Figure 2. Though having less number of divergent paths, the code in Figure 6 runs much more slowly. In this paper, we assume that programmers would not write a code in this way, and all the benchmarks in our experiments are not written in this way either.

We simply model the impact of branch divergence as follows:

$$\boldsymbol{branch\_eff} = \frac{n\_warp}{n\_path} \tag{7}$$

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

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

[5]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.

```
1  ...
2
3  if (tIdx.x == bDim.x-2)
4  {
5    out[row*MAX+col] = s_in[tIdx.y][tIdx.x] *
6                       s_in[tIdx.y][tIdx.x+1] *
7                       in[row*MAX+col+2];
8  }
9  else if (tIdx.x > bDim.x-2)
10 {
11   out[row*MAX+col] = s_in[tIdx.y][tIdx.x] *
12                      in[row*MAX+col+1] *
13                      in[row*MAX+col+2];
14 }
15 else
16 {
17   out[row*MAX+col] = s_in[tIdx.y][tIdx.x] *
18                      s_in[tIdx.y][tIdx.x+1] *
19                      s_in[tIdx.y][tIdx.x+2];
20 }
21
22 ...
```

Fig. 6. *A worst case coding style for branch divergence. Every branch paths are serialized in CUDA. Every memory access is duplicated on every path, which makes the number of memory requests three times more in this example.*

where n_warp denotes the number of total warps in a program.

5.7. Overall Memory Performance Estimation

Now all the factors that we explained in the above are combined together to estimate overall memory performance. Memory performance estimate is calculated by the following formula.

$$\boldsymbol{MPE} = data\_reuse \times lat\_hiding \times \frac{bw\_util}{ch\_skew} \times branch\_eff \times shm\_eff^{1/2} \tag{8}$$

shm_eff is square-rooted to reflect the relatively smaller impact of shared memory bank conflict on performance over other terms, which is empirically determined by curve fitting. The larger MPE value is, the better the memory performance is.

6. EMPIRICAL EVIDENCE

We have implemented CuMAPz using C language. We ran CUDA driver version 3.2 on NVIDIA Tesla C1060 [NVIDIA a] for all experiments. Table IV shows the list of benchmarks used in this paper. The Laplace edge enhancement and Wavelet transformation applications are from benchmark suites in [Kolson et al. 1996], and matrix multiplication and matrix transpose are from CUDA SDK. Gaussian, LUD, and Hotspot are from Rodinia benchmark suite [Che et al. 2009]. The input size used for our benchmarks are large enough to saturate all the processing cores for thousands of iterations. To avoid the device startup time affecting the results, we included a dummy kernel launch as follows before the actual kernel launch and measured only the latter kernel execution time.

```
Device code:
  __global__ void dummy() {} // dummy kernel declaration
Host code:
  dummy<<<1,1>>>(); // dummy kernel launch
```

We divide our experiments section into two parts i) Validation: in which we study the correlation between our memory performance estimation and the performance of the benchmarks for different ways of accessing shared and global memory, and ii) Performance Optimization: in which we try to find the best way to accesses shared and global memory using CuMAPz and the previous technique [Hong and Kim 2010].

Table IV. Kernel Characteristics

	Input size	**Thread block size**	**Description**
Laplace	8192x8192	16x16	Laplace transform
Wavelet	8388608	128	Wavelet filter
MatrixMul	1024x1024	varied	Matrix multiplication
Transpose	2048x2048	32x32	Matrix transpose
Gaussian	160x160	varied	Gaussian elimination
LUD	2048x2048	varied	LU decomposition
Hotspot	1024x1024	16x16	Thermal simulation of a circuit
BackProp	65536	16x16	Machine learning algorithm on a nerual network

6.1. Validation

Here we compare estimated memory performance *MPE* by CuMAPz and the execution time of the real code using different ways to use global and shared memories. The purpose of these experiments is to see how well CuMAPz can predict the *relative* performance change as we try various ways of using memories. We refer to the reciprocal of execution time as performance. Both values are normalized in order to compare two sets of values in different scales. In all experiments we tried, performance change was caused by factors that previous approach [Hong and Kim 2010] does not take into account. As a result, the performance prediction by their approach often stays almost the same for all cases while the real performance varies a lot.

Laplace loop has two arrays, and one of them has nine references having different strides. We only change what data is fetched into shared memory by using each reference one at a time. `[row-1][col]`, `[row][col]`, and `[row+1][col]` are coalesced accesses. Using one of these as a fetch function, a large part of uncoalesced accesses caused by other references are substituted with corresponding shared memory accesses, and much more data reuse can be exploited. Figure 7(a) shows the comparison result. References are denoted only using stride of accesses to save space. CuMAPz can accurately predict the relative performance of all cases with correlation coefficient of 0.99. To see how CuMAPz estimated the memory performance, we show the normalized values of each term in Equation 8 in Figure 7(c). The values are normalized because of different scales of terms. The terms that have the same value for all cases are not 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. 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 references are removed and substituted with corresponding shared memory accesses.

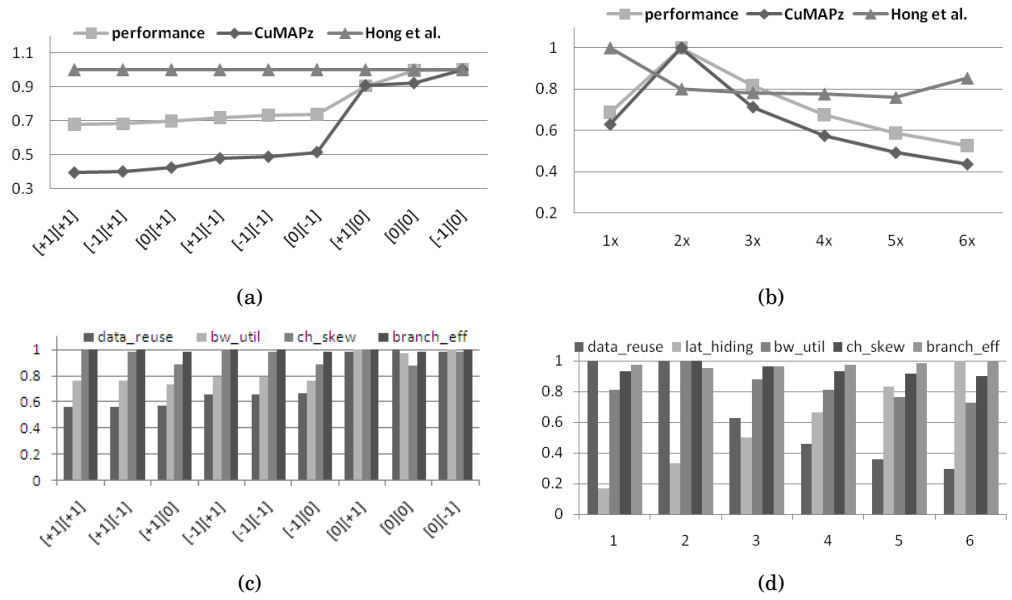


Fig. 7. *Results for Laplace and Wavelet (a),(b) CuMAPz can estimates the relative performance more accurately than the previous approach. (c) Mainly data reuse and bandwidth utilization are affecting performance. (d) Data reuse, latency hiding, and branch divergence are most significant factors affecting performance.*

Wavelet loop has three arrays in it, and one array 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. 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. Thus, we have at most six independent load instructions for shared memory data prefetching. Figure 7(b) shows the comparison result, and the correlation coefficient to the performance is 0.92. As shown in Figure 7(d) branch divergence reduces as buffer size increases, because a branch disappears as one buffer is dedicated to one reference. However, the best performance is achieved when the buffer size is the twice the thread block size. This is because the buffer size of twice of the thread block size can employ data reuse the most. Because of the overlap between the regions covered by each reference, larger buffer size than this results in unnecessarily fetching duplicated elements. Also, it is shown that the degree of latency hiding increases as the number of buffers goes up because the number of independent data fetch references grows.

For MatMul, at first we prefetch one of three matrices (A, B, and C) and then both A and B into the shared memory. Then, we try the similar conversion shown in [Volkov 2010] such that we shrink the block dimension of X or Y direction (denoted as AB_x and AB_y respectively in Figure 8(a) and 8(c)) while not changing shared memory buffer size and grid dimensions. This conversion gives each thread twice more work. Half number of threads prefetch the shared memory buffer of the same size, and thus the number of prefetch load instructions is doubled. As shown in 8(a), CuMAPz can accurately estimate the performance change of each case. The correlation coefficient between performance and CuMAPz output was 0.94. The accesses to the first input matrix A are not coalesced at all because all threads in a half warp access exactly the same element at a time, accessing the same row, which makes prefetching matrix A more beneficial than B. If we prefetch both A and B, then the number of load in-

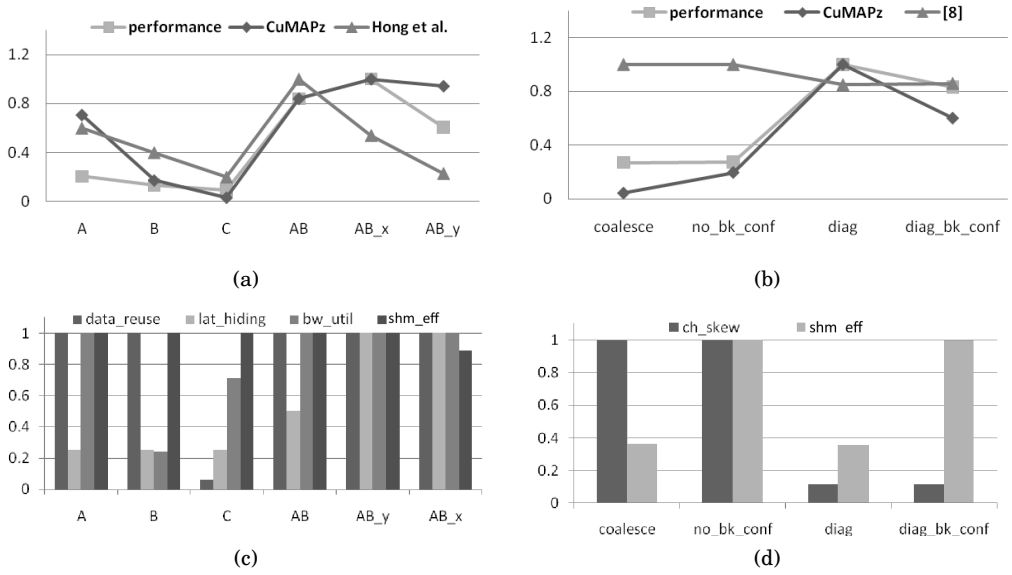


Fig. 8. *Results for MatrixMul and Transpose (a),(b) CuMAPz can estimates the relative performance more accurately than the previous approach. (c) Latency hiding and bandwidth utilization affect performance most significantly. (d) Channel skew and bank conflict are only dominating factors.*

structions becomes two. The above mentioned conversion makes the degree of latency hiding increase, but reducing the block dimension in X axis introduces two-way shared memory bank conflicts. This is because the `blockDim.X` becomes 8, which makes two threads within a half warp (e.g. thread 0 and thread 8) access the same bank. Figure 8(c) describes all of the phenomena.

For Transpose, the same conversions as in [Ruetsch and Micikevicius 2009] are done. First, uncoalesced global memory accesses are substituted with coalesced ones using shared memory. Data are first copied into shared memory in such a way that accesses are coalesced, then data are read from shared memory. Second, global memory accesses are now coalesced, but there are shared memory bank conflicts. The bank conflicts are removed by increasing the X-dimension of the buffer by one as we described in Section 3.2. However, this conversion does not bring much performance improvement because the code is suffering from severe channel skew. Then, we remove channel skew by using diagonal block ordering [Ruetsch and Micikevicius 2009], which is to change the interpretation of `blockIdx.x` and `blockIdx.y` as follows so blocks can work on different data. Note that `blockIdx.x` and `blockIdx.y` in the rest of the code need to be substituted with `blockIdx_x` and `blockIdx_y`, respectively.

```
blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;
blockIdx_y = blockIdx.x;
```

Lastly, we again roll back the modification done in the second step to see the effect of shared memory bank conflict. Figure 8(d) shows the change of corresponding performance critical factors at each step. At first step, shared memory efficiency is low due to shared memory bank conflict, and it is resolved at second step. At third step, channel skew is resolved, and bank conflict appears again at the last step. Figure 8(b) shows that CuMAPz can accurately estimate the performance variation with correlation coefficient of 0.97.

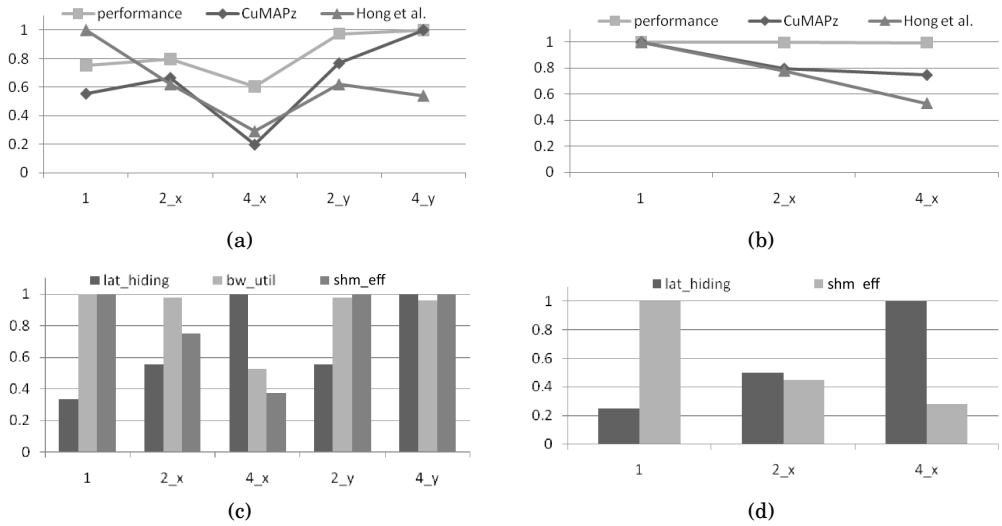


Fig. 9. *Results for Gaussian and LUD (a),(b) CuMAPz can estimates the relative performance more accurately than the previous approach. (c) Latency hiding, global memory coalescing and shared memory bank conflict affect performance. (d) Latency hiding and shared memory bank conflicts determine the performance change*

Similarly to MatrixMul, we do the same conversion as done in [Volkov 2010] for Gaussian and LUD. In Gaussian, the block dimension along X axis is first reduced from 16 to 8 (2_x) and then 4 (4_x). Then the same modification is done along Y axis (2_y and 4_y). The performance estimation accuracy of CuMAPz is shown in Figure 9(a), and the correlation coefficient is 0.95. As we reduce the block dimension, global memory access latency can be hidden more, but shared memory bank conflicts increases as we shrink the block dimension along X axis. Also, the global memory accesses are not coalesced in 4_x case since only four consecutive memory addresses are requested at a time, which is utilizing only half of the minimal transaction size of 32-byte. All of the phenomena mentioned above are depicted in Figure 9(c).

LUD benchmark has three separate kernels: `lud_diagonal`, `lud_perimeter`, and lud_internal. Only `lud_internal` is modified since it is iterated for the most number of times and dominates the program performance. Figure 9(b) and 9(d) show the similar phenomena for LUD as the X axis block dimension gets shrunk from 16 to 8 and then to 4. Interestingly, for all three cases, performance barely changes. More latency hiding can be achieved, but bank conflicts also increases at the same time. Also, this kernel has a loop in it, which accesses shared memory extensively, so shared memory bank conflicts affect the performance more significantly. The benefit from latency hiding is almost compensated and performance starts to slightly decrease in the last case. Overall, correlation coefficient between CuMAPz and performance is 0.97.

Hotspot benchmark has three global memory arrays and three shared memory buffers, with various conditional branches and loops with memory references. First, we apply diagonal block ordering as we did in Transpose. As shown in the second columns in Figure 10(a) and 10(c), it causes a significant channel skew, and performance becomes worse. As opposed to the case with Transpose, diagonal block ordering deteriorates channel skew, which urges the importance of comprehensive memory analysis. Then, we apply the following thread id translation code to switch `threadIdx.x` and `threadIdx.y`.

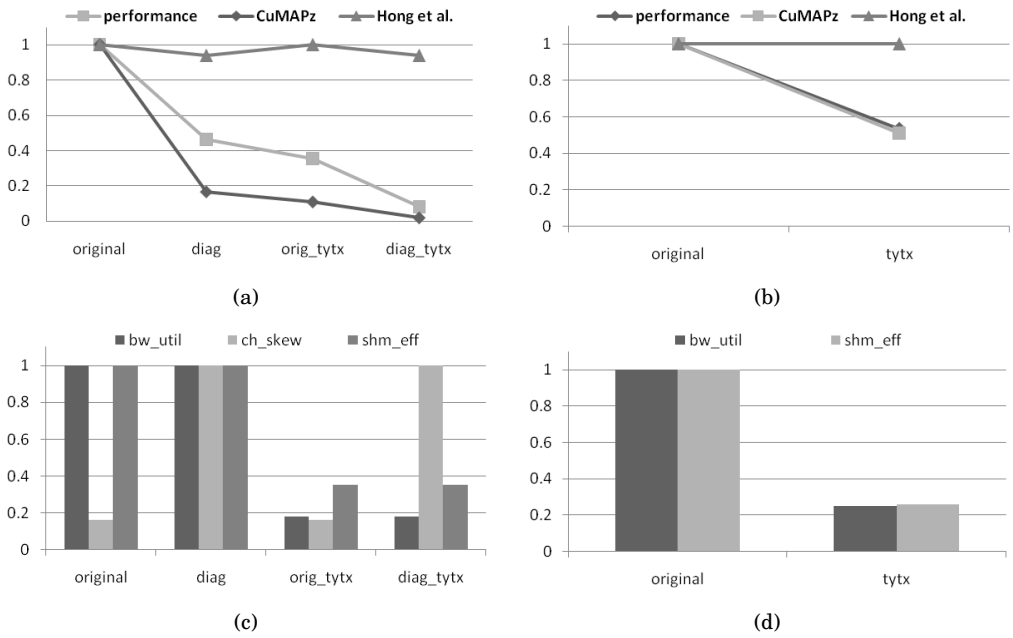


Fig. 10. *Results for Hotspot and BackProp (a),(b) CuMAPz can estimates the relative performance more accurately than the previous approach. (c) Bandwidth utilization, channel skew, and shared memory bank conflict affect performance. (d) Bandwidth utilization and shared memory bank conflict affect performance.*

```
tx = threadIdx.y;
ty = threadIdx.x;
```

The global memory references and shared memory references in this benchmark are complex functions of thread id's, and now it uses `tx` and `ty` instead of `threadIdx.x` and `threadIdx.y`, which completely changes the access pattern of both global memory and shared memory. The third and fourth columns in figure show the effect of this applied to both the original benchmark and the one with diagonal block ordering. Global memory bandwidth utilization and shared memory access efficiency are aggravated greatly, and performance is degraded. CuMAPz can analyze all of these effect and estimate performance accurately, showing the correlation coefficient of 0.96.

Figure 10(b) shows performance degradation of BackProp benchmark, caused by switching thread id's as we did for Hotspot. Using CuMAPz, we can estimate the performance degradation and analyze the cause as shown in Figure 10(d). Global memory bandwidth utilization and shared memory access efficiency are greatly reduced. Since there are only two points, the correlation coefficient is one.

6.2. Performance Optimization

In this section, to see the usefulness of the detailed memory performance analysis, we show the performance improvement that we could achieve using the proposed technique. For each benchmark, we prepared a set of input parameters on memory usage (shown in Figure 4) and tested the performance. To be more specific, we compared the performance of 1) fetching different global memory arrays into shared memory buffers, 2) using different data fetch references for each shared memory buffer, 3) using different shared memory buffer sizes, 4) using different the global memory and shared memory access references, and 5) using different thread block sizes.

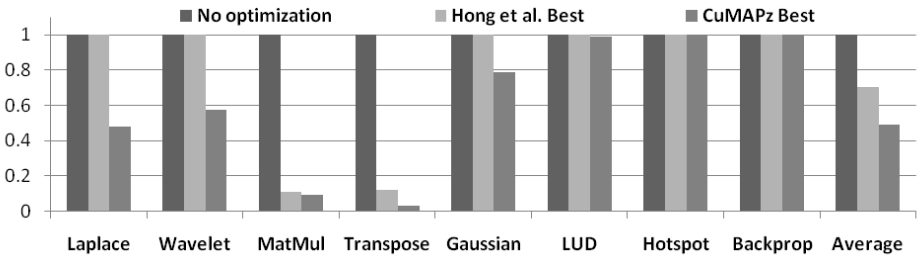


Fig. 11. *Normalized execution time comparison among the original benchmarks with no optimization, and the cases using the best way of using memories selected by [Hong and Kim 2010] and CuMAPz. The memory usages selected by CuMAPz can reduce execution time more than 30% on average compared to the ones selected by [Hong and Kim 2010].*

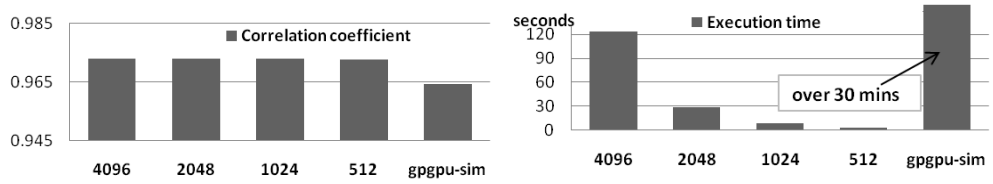
Figure 11 compares the execution time of the best way of using shared and global memories, among the input we tried , selected by [Hong and Kim 2010] and CuMAPz. The baseline is the execution time of the original benchmark, without any modification. Gaussian and LUD already had shared memory buffers, so the amount of execution time reduction is not as much as we have for Laplace, Wavelet, MatMul, and Transpose. Moreover, all the design choices we tried for Hotspot and BackProp resulted in performance degradation, i.e. we could not optimize them. This however does not necessarily mean that the benchmarks are already optimized enough. We can still try other design choices which may result in better performance. For this purpose, automatic design space exploration is part of our future work.

Using [Hong and Kim 2010], we could not identify the best performance design choices due to lack of detailed memory performance analysis. Many performance critical factors are not considered such as channel skew, latency hiding, and branch divergence. Also, shared memory instructions are just regarded as register operations. Unless the number of global memory instructions is reduced as in matrix multiplication and matrix transpose, shared memory buffers only increases instruction count in [Hong and Kim 2010]'s viewpoint, so it chose rather not to use shared memory. In matrix transpose, it could not differentiate the worst case and the best case in the explored design space, because it does not consider channel skew and shared memory bank conflict. Note that when the performance estimation using [Hong and Kim 2010] was the same for multiple design choices, we chose the best performance case among them. This means that performance of the best design choices found by their model can also be worse than that in the figure.

Overall, CuMAPz-chosen usages could reduce execution time by average 30% more than the best ways chosen by [Hong and Kim 2010].

7. RUNTIME CONSIDERATIONS

The timing complexity of the CuMAPz analysis is O($|W|\cdot|R|\cdot|B|$), where W, R, and B are the set of all warps, global memory references, and shared memory buffers respectively. In case where the kernel contains loops that have memory references, the complexity increases with the loop depth exponentially. To work around this, we would like to note that CuMAPz analysis scales with input data size. In other words, it is possible to perform the analysis on a smaller data set, and the choice of the best way to access the shared and global memory remains valid for the original program. Except for the channel skew, all other terms are independent of input size. The ratio between shared memory and global memory accesses for data reuse and branch divergence analysis remains the same for different input size. Also, as long as full warps are executed, the



(a) Correlation coefficients change (b) Average execution time change

Fig. 12. *The correlation coefficients stay almost the same as we reduce the input size while the execution times decrease dramatically.*

pattern of global memory coalescing and shared memory bank conflict do not change either. For channel skew analysis to scale, we should at least have number of blocks more than that obtained by *n_blk_to_check*, which is acquired from Equation 4.

We compared the correlation coefficients between CuMAPz output and performance for various input sizes. Figure 12(a) shows the four results for Laplace benchmark, varying the input 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 is less than 0.001. We also show the correlation coefficient and the execution time of GPGPU-sim (v. 2.1.2b) [Bakhoda et al. 2009], a cycle-accurate GPGPU simulator. GPGPU-sim takes more than 30 minutes for 1024x1024 input array, while CuMAPz takes only about two minutes for 4096x4096 arrays and about two seconds for 512x512 arrays. For results in this paper, we used orders of magnitude smaller input arrays. (e.g., 256x256 array instead of 8192x8192 array.) The average correlation coefficient between actual runtime and CuMAPz output for all benchmarks was 0.96, and the maximum CuMAPz runtime among all cases was less than 2 seconds on E4500 Intel Core 2 Duo system with 3GB memory.

8. LIMITATION

Our approach is compile-time analysis, and therefore we cannot handle any information that can only be determined during run-time, such as dynamically allocated shared memory, indirect array accesses, etc. Also, we only focus on memory performance. Thus, our approach may not show the same accuracy for compute-intensive kernels as for memory-intensive kernels. For example, we cannot model latency hiding from executing independent ALU instructions. We only model the latency hiding from having multiple data fetch references as they are obviously independent. For the same reason, the *MPE* values from Equation 8 are only meaningful for different versions of one program. Since different program may have different computation performance, performance estimation of CuMAPz is only relative to other design choices of a given program.

9. CONCLUSION AND FUTURE WORK

GPUs provide power-efficient processing power in embedded systems, but optimizing GPGPU programs is not easy due to complex memory hierarchy and many intercoupled performance factors to be considered. The memory performance affects the program performance very significantly; therefore optimizing the memory behavior of a program is crucial in optimizing GPGPU programs. In this paper, we present CuMAPz, a tool which can help developers to explore different ways to use global and shared memories, estimate their performance, and thereby optimize the program. CuMAPz comprehensively analyzes various performance critical effects on memory behavior, such as data reuse, global memory latency hiding, global memory access coa-

lescing, bank conflicts, channel skew, and branch path divergence. Since our approach can quantify the performance impact of each of performance critical factors, programmer can find a possible bottleneck and try to improve the corresponding part of the code. Experimental results show very high correlation between the actual execution time and CuMAPz memory performance estimation. Two main threads of research that start from this work are: i) automatic design space exploration to find the best way to use memories, and ii) taking texture memory and constant memory into consideration.

REFERENCES

ARM. ARM Mali GPU. http://www.arm.com/products/multimedia/mali-graphics-hardware.

BAKHODA, A., YUAN, G., FUNG, W., WONG, H., AND AAMODT, T. 2009. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. 163 –174.

BASKARAN, M., RAMANUJAM, J., AND SADAYAPPAN, P. 2010. Automatic C-to-CUDA Code Generation for Affine Programs. In *Compiler Construction*, R. Gupta, Ed. Lecture Notes in Computer Science Series, vol. 6011. Springer Berlin / Heidelberg, 244–263.

BASKARAN, M. M., BONDHUGULA, U., KRISHNAMOORTHY, S., RAMANUJAM, J., ROUNTEV, A., AND SADAYAPPAN, P. 2008. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd annual international conference on Supercomputing*. ICS '08. ACM, New York, NY, USA, 225–234.

CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J., LEE, S.-H., AND SKADRON, K. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. 44 –54.

GE INTELLIGENT PLATFORMS. IPN250 single board computer. http://www.ge-ip.com/products/3514.

HONG, S. AND KIM, H. 2010. An integrated gpu power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*. ISCA '10. ACM, New York, NY, USA, 280–289.

ISSENIN, I., BROCKMEYER, E., MIRANDA, M., AND DUTT, N. 2004. Data reuse analysis technique for software-controlled memory hierarchies. In *IN DESIGN AUTOMATION AND TEST IN EUROPE (DATE'04)*. 202–207.

KOLSON, D., NICOLAU, A., AND DUTT, N. 1996. Elimination of redundant memory traffic in high-level synthesis. *IEEE Trans. on Comp-aided Design 15*, 1354–1363.

LEUNG, A., VASILACHE, N., MEISTER, B., BASKARAN, M., WOHLFORD, D., BASTOUL, C., AND LETHIN, R. 2010. 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 '10. ACM, New York, NY, USA, 51–61.

NVIDIA. Board Specification, Tesla C1060 Computing Processor Board. http://www.nvidia.com/docs/IO/56483/Tesla_C1060_boardSpec_v03.pdf.

NVIDIA. NVIDIA ION processors. http://www.nvidia.com/object/sff_ion.html.

NVIDIA. 2010a. NVIDIA CUDA Best Practices Guide, Version 3.1.

NVIDIA. 2010b. NVIDIA CUDA Programming Guide, Version 3.1.

OpenCL. OpenCL. http://www.khronos.org/opencl/.

RUETSCH, G. AND MICIKEVICIUS, P. 2009. Optimizing matrix transpose in cuda.

RYOO, S., RODRIGUES, C. I., BAGHSORKHI, S. S., STONE, S. S., KIRK, D. B., AND HWU, W.-M. W. 2008. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. PPoPP '08. ACM, New York, NY, USA, 73–82.

RYOO, S., RODRIGUES, C. I., STONE, S. S., BAGHSORKHI, S. S., ZEE UENG, S., STRATTON, J. A., AND MEI W. HWU, W. 2008. Hwu. program optimization space pruning for a multithreaded gpu. In *In Intl Symp. on Code Generation and Optimization (CGO)*.

TECHNISCAN. 3D breast cancer detection system using Tesla. http://www.techniscanmedicalsystems.com.

VOLKOV, V. 2010. Better performance at lower occupancy. Presentation at GPU Technology Conference 2010.

YANG, Y., XIANG, P., KONG, J., AND ZHOU, H. 2010. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '10. ACM, New York, NY, USA, 86–97.

ZEE UENG, S., LATHARA, M., BAGHSORKHI, S. S., AND MEI W. HWU, W. 2008. W.m.w.: Cuda-lite: Reducing gpu programming complexity. In *In: LCPC08. Volume 5335 of LNCS*. Springer, 1–15.

ZHANG, Y. AND OWENS, J. D. 2011. A Quantitative Performance Analysis Model for GPU Architectures. In *Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA 17)*.