CuMAPz: A Tool to Analyze Memory Access Patterns in CUDA

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

Abstract
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 several architectural details, and small changes in source code, especially on memory access pattern, affect performance significantly. This paper presents CuMAPz, a tool to compare the memory performance of a CUDA program. CuMAPz can help programmers explore different ways of using shared and global memories, and optimize their program for memory behavior. CuMAPz models several memory effects, e.g., data reuse, global memory access coalescing, shared memory bank conflict, channel skew, and branch divergence. By using CuMAPz to explore memory access design space, we could improve the performance of our benchmarks by 62% over the naive cases, and 32% over previous approach[8].

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 has now reached teraFLOP scale. Traditionally, GPGPU (General Purpose computation on GPUs) programming required a high level of expertise and proficiency, but NVIDIA CUDA [9] and OpenCL [1] 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 [2, 3]. Recently, GPGPU has gained much attention in mobile systems due to its high power efficiency. Low power GPUs such as ARM Mali [5] and NVIDIA ION [4], 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 have large data sizes. Massive parallel architecture and complex memory hierarchy of GPUs have introduced a large set of performance optimization considerations [10, 13, 21], and many of them 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.

In this paper, we focus on the problem of improving application performance by using shared memory. 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 reads and writes are enabled. Other memories, such as texture memory and constant memory, are read-only. Even in this small subset of the problem, several memory performance factors, e.g., data reuse, global memory access coalescing, shared memory bank conflict, channel skew, etc., need to be considered to optimize performance. 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. Programmers can use it to explore different ways of using shared and global memories, and optimize their program for memory performance without even writing the kernels.

To the best of our knowledge, this is the most comprehensive approach to analyze the memory performance of programs, covering all the well-known factors such as the degree of data reuse, global memory access coalescing, shared memory bank conflict, channel skew, and branch divergence. Our experiments on benchmarks from [6] and kernels from CUDA SDK indicate that CuMAPz can accurately predict the relative performance of various ways to use shared and global memories, with average correlation coefficient of 0.93. By using CuMAPaz to explore shared and global memory use design space, we could improve the performance of our benchmarks by 62% over the naive case, and 32% over previous approach[8].

2. Background
CUDA architecture is massively parallel in that a kernel is executed by thousands of threads. There is a grid of streaming multiprocessors (SMs), and each SM has multiple scalar processors (SPs). Threads are grouped into thread blocks. Each thread has its own thread id to represent its relative location in a block, and thread blocks also have ids for themselves. Combining the thread id and the block id, each thread is assigned a unique id. Each thread block is assigned to a 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 ids in a warp are consecutive. SPs execute one warp at a time, and the execution of a warp happens in a SIMD manner, so that the threads in the same warp are executed in lock-step.

Global memory in CUDA is not cached, and thus the latency of accessing global memory is hundreds of cycles. CPU code can only transfer data to this off-chip memory, so basically all data resides in global memory. The long latency can often be hidden with the help of having a large number of threads in-flight and pipelining. 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 [11]) the massively parallel
execution can easily saturate the bandwidth – and this often becomes the performance bottleneck in CUDA programs.

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. Since shared memory is on-chip, the bandwidth is not a performance limiting factor, however, bank conflicts can often slow down a program.

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 1 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 1 shows that among the three options for fetching 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 1 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 coalescing. 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.

3.2 Question 2: How to access shared memory?

In this section, 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 out[4\times4]\times\text{col}\times\text{row}. This would have resulted in disastrous performance as shown in Table 3. 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.

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 out[4\times4]\times\text{col}\times\text{row}. This would have resulted in disastrous performance as shown in Table 3. 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, too.

### Table 1.

<table>
<thead>
<tr>
<th>Data reuse</th>
<th>Execution time (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>not using shared mem</td>
<td>78.15</td>
</tr>
<tr>
<td>row*MAX+col</td>
<td>754925504</td>
</tr>
<tr>
<td>row*MAX+col+1</td>
<td>771670016</td>
</tr>
<tr>
<td>row*MAX+col+2</td>
<td>754786416</td>
</tr>
</tbody>
</table>

Table 1. 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.
in shared memory buffer, c) bank conflicts in shared memory incurred by shared memory usage, b) data reuse of data fetched from 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. Shared memory bank conflicts and channel skew is not considered either. Comparing to all the above works, our approach is a more comprehensive in analyzing memory performance.

As a more aggressive solution to relieve the burden on programmers, there have been efforts to automate optimization of GPGPU applications. Ueng et al. [16] 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. [15] 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 [18]. Later, another work [17] 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. Recently, Yang et al. [14] 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 except for branch path divergence. Our approach is not an optimizing compiler but enables a performance comparison or relative performance estimation of different program designs. None of the above work comes up with a comprehensive performance metric to estimate the efficiency of memory access pattern.

5. Our Approach
As shown in Figure 3, our analysis comprehensively covers performance-critical effects in the architecture: a) branch divergence in SIMD cores incurred by shared memory usage, b) data reuse of data fetched in shared memory buffer, c) bank conflicts in shared memory access patterns, d) coalescing in global memory accesses, and e) channel skew in global memory accesses.

The list of input that CuMAPz takes, as shown in Figure 4, is 1) hardware configuration, 2) thread blocks and grids sizes, 3) global memory references, 4) the structure of loops inside a kernel if any memory references are in the loops, 4) shared memory buffer size, and 4) mapping between buffer and global memory array. Using the information, we construct a loop structure that enumerates all memory addresses that are requested for all warps in a program. Using the loop structure, all memory accesses during the execution of the program are generated and the above memory behaviors are simulated. Note that, instead of taking shared memory reference, we only keep track of the mapping between global memory array elements and shared memory array elements. For example, Line 4 in Figure 2 is a pair of a global memory read reference and a shared memory write reference, which represents the mapping. For each element in shared memory buffer in a thread block, the associated global memory address is obtained by the mapping. If a global memory address generated during the simulation is already mapped to one of buffers within the same block, then corresponding shared memory access is inferred.

5.1 Data Reuse Profit Estimation
CuMAPz maintains a counter to count the number of times shared memory buffers are accessed. The degree of data reuse is represented in a term, data reuse, as follows:

\[
data_{\text{reuse}} = \frac{\text{bytes}_{\text{shmem}}}{\text{bytes}_{\text{buffered}}} \]

\[
\text{bytes}_{\text{buffered}} = \sum_{m \in M} \sum_{w \in W} \text{bytes}_{fr}^{w,m}
\]

\[
\text{bytes}_{\text{shmem}} = \sum_{r \in R} \sum_{b \in B} \sum_{w \in W} \text{bytes}_{\text{shmem}}^{w}
\]

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. \(\text{bytes}_{fr}^{w,m}\) represents the bytes transferred while fetching data from global memory (or updating global memory with data written in buffer), in warp \(w\). \(\text{bytes}_{\text{shmem}}^{w}\) 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 \(\text{bytes}_{fr}\) is detailed in next section.

5.2 Coalesced Access Profit Estimation
When an address accessed by a global memory reference is not mapped to any buffer, global memory access occurs. The transaction
size, of 32-byte, 64-byte, or 128-byte, is determined for the memory access pattern in each half warp (i.e., 16 threads) according to the architecture specification [9]. Due to coalescing, the actual transfer size that will consume bus width can be different from the size of data requested from threads. CuMAPz calculates the bandwidth utilization as the following:

\[
\text{bw}_{\text{util}} = \frac{\text{bytes access}}{\text{bytes fr}} \quad (2)
\]

\[
\text{bytes access} = \sum_{r \in R} \sum_{w \in W} \text{bytes access}_r^w
\]

\[
\text{bytes fr} = \sum_{r \in R} \sum_{w \in W} \text{bytes fr}_r^w
\]

where \(\text{bytes access}_r^w\) and \(\text{bytes fr}_r^w\) 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.

5.3 Channel Skew Cost Estimation

The memory subsystem in CUDA consists of multiple channels, and all of the channels can transmit data at the same time in parallel. Channel skew refers to the case where the concurrent memory accesses are not evenly distributed to all the channels but focused on only a few channels. This phenomenon is sometimes called partition camping [13]. This essentially makes the bus width much narrower on top of the slow latency of global memory.

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 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 [13]. Our channel skew analysis is 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.

If we consider the minimum number of blocks to fill all channels in their execution of first warps, we have:

\[
n_{\text{channels}} \times \text{MIN}(\text{max blks}, \frac{\text{channel width}}{\text{bDimX} \times \text{elem size}}) \quad (3)
\]

where \(\text{max blks}\) is the maximum number of blocks that can run concurrently in one SM for the given kernel, which is similar to occupancy [9] except that occupancy is a ratio. \(\text{elem size}\) is the minimum element size of global memory arrays. When the number of thread blocks is smaller than the number stated in the above, this analysis is skipped.

The impact of channel skew can be stated in figures as the skewness of mapping to channels which can be calculated as follows:

\[
\text{ch skew} = \frac{\text{max n block per ch}}{\text{MAX}(1, \text{min n block per ch})} \quad (4)
\]

where \(\text{max n block per ch}\) and \(\text{min n block per ch}\) denote, respectively, the maximum and minimum number of blocks assigned to a channel for the initial \(I\) number of blocks, where \(I\) is the number acquired from Equation (3).

5.4 Bank Conflict Cost Estimation

Similarly to global memory channels, shared memory space is divided into multiple banks. Successive four bytes data are assigned to successive banks. 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. 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:

\[
n_{\text{bk conflict}} = \sum_{r \in R} \sum_{b \in B} \sum_{w \in W} n_{\text{bk conflict}}^r_w
\]

where \(n_{\text{bk conflict}}^w\) 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:

\[
\text{shm eff} = \frac{n_{\text{half warp}} \times n_{\text{buffer}}}{n_{\text{bk conflict}}} \quad (5)
\]

where \(n_{\text{half warp}}\) and \(n_{\text{buffer}}\) denote number of half warps and number of shared memory buffers in a program.

5.5 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 buffered 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. We simply model the impact of branch divergence as follows:

\[
\text{branch eff} = \frac{n_{\text{warp}}}{n_{\text{path}}}
\]

\[
n_{\text{path}} = \sum_{r \in R} \sum_{b \in B} \sum_{w \in W} n_{\text{path}}^w
\]

\[
\text{branch eff} = \begin{cases} 2, & \text{if paths diverged in warp } w \text{ for } r \\ 1, & \text{otherwise} \end{cases}
\]

where \(n_{\text{warp}}\) denotes the number of total warps in a program.

5.6 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.

\[
\text{MPE} = \text{data reuse} \times \frac{\text{bw}_{\text{util}}}{\text{ch skew}} \times \text{branch eff} \times \text{shm eff}^{1/2}
\]

The square root of \(\text{shm eff}\) is to reflect the relatively smaller impact of shared memory bank conflict on performance over other terms, and is empirically determined by curve fitting.

6. Empirical Evidence

We have implemented CuMAPz using C language. We ran CUDA driver version 3.2 on NVIDIA Tesla C1060 for all experiments. The Laplace edge enhancement and Wavelet transformation applications are from benchmark suites in [6], and the other two, matrix multiplication and matrix transpose are from CUDA SDK. 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 [8].

6.1 Validation

To see how well CuMAPz can predict the relative performance change as we try various design choices, MPE numbers obtained by CuMAPz and the performance obtained by actually running the real kernels on hardware are compared. 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.

Laplace loop has two arrays, and one of them has nine references having different strides. We only change what data is fetched into

\(^1\) 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.
shared memory by using each reference one at a time. \([\text{row} - 1][\text{col}],\) \([\text{row}][\text{col}],\) and \([\text{row} + 1][\text{col}]\) 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 5.(a) shows the result. References are denoted only using stride of accesses to save space. CuMAPz can accurately predict the performance of all cases with correlation coefficient of 0.99.

Wavelet loop has three arrays in it, and one array has six references. Here we increase the buffer size, introducing one more buffer at a time, whose size is the same as block size. Each buffer is mapped to one of the reference. The 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 twice the thread block size. This is because the buffer size of twice of the thread block size can employ data reuse the most. Figure 5.(b) shows the result, and the correlation coefficient to the performance is 0.85.

For matrix multiplication, we compare three cases of fetching one of three matrices as a whole. 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. The accesses for other two matrices, the second input matrix B and output matrix C, are completely coalesced, which makes fetching matrix A more beneficial. The degree of data reuse is much higher when prefetching A or B than C. Figure 5.(c) shows the result, and the correlation coefficient to the performance is 0.99.

Figure 5.(d) shows the result for matrix transpose. We use the same conversions done in [13]. First, uncoalesced global memory accesses are substituted with coalesced ones using shared memory. Second, the shared memory bank conflicts are removed by the technique we described in Section 3.2. 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 coordinates [13] in next case. Lastly, we again roll back the modification done in the second case to see the effect of shared memory bank conflict. CuMAPz can accurately predict the performance with correlation coefficient of 0.89.

6.2 Performance Optimization

For each benchmark, we explored various ways to use shared memory and global memory. We try: 1) several fetch functions for shared memory buffer, 2) different arrays to be fetched into shared memory buffer, 3) various shared memory buffer size, and 4) several ways to access shared memory and global memory. Figure 6 compares the performance of the best way of using shared and global memories selected by [8] and CuMAPz. In [8], branch divergence is not considered and 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 [8]’s viewpoint, so it chose rather not to use shared memory. Even for matrix transpose, it could not differentiate the worst way and the best way in the explored design space, because it does not consider channel skew and shared memory bank conflict. Overall, CuMAPz-chosen usages could improve performance by average 32.3% more over the best ways chosen by [8]. Also, for [8], we had to write a complete kernel for each case of different way of using memory and compile it to get PTX code, while CuMAPz can generate results only by changing input configurations.

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, the kernel contains loops that have memory references, the complexity increases exponentially with the loop depth. To work around this, we 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 sizes. Also, as long as full warps are executed, the 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 Equation (3).

We compared the correlation coefficients between CuMAPz output and performance for various input sizes. Figure 7(a) shows the...
Computation on Graphics Processing Units

8. Limitation
Our approach is compile-time analysis. 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 assume that adequate occupancy is achieved. Occupancy is a measure of how many thread blocks can be scheduled on one SM so that the hardware is kept busy. Low occupancy can lead to significant performance degradation. However, NVIDIA suggests that 50% of occupancy is often enough to achieve the best performance [9]. Checking occupancy is also relatively easy using the occupancy calculator [12].

9. Acknowledgment
This work was partially supported by funding from National Science Foundation grants CCF-0916652, CCF-1055094 (CAREER), and equipment donation from Nvidia.

10. Conclusion
GPUs provide power-efficient processing power in embedded systems, but optimizing GPGPU programs is not easy due to complex memory hierarchy and many inter-coupled 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 a tool, CuMAPz: A tool to analyze the memory performance. Even though this is a simulation based approach, this is one possible implementation of our proposed memory performance estimation technique, and there is a possibility that the same result can be acquired analytically.

References