Static Memory Access Pattern Analysis on a Massively Parallel GPU

Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli
Computer Architecture Labratory
Northeastern University
Boston, MA 02188 USA
{bjang, dschaa, pmistry, kaeli}@ece.neu.edu

1 Introduction

The performance of data-parallel processing can be highly sensitive to any contention in memory. In contrast to multi-core CPUs which employ a number of memory latency minimization techniques such as multi-level caching and prefetching, Graphics Processing Units (GPUs) require that the data-parallel computations reference memory in a deterministic pattern in order to reap the benefits of these many-core platforms. Memory access sensitivity is primarily due to the Massively Parallel Processing (MPP) execution model and underlying memory hardware architecture of GPUs which are specifically tuned for graphics rendering [2, 4].

In this paper we present a static memory access pattern analysis model that provides guidance on how best to apply a wide range of memory optimizations on GPUs. Our analysis carefully takes into account the mapping of threads to data, a critical factor when attempting to exploit the full capabilities of current GPU architectures. We formulate a methodology that allows us to build tools to guide programmers on how best to apply algorithmic memory optimizations and can easily be integrated into a pass of a compiler.

We demonstrate the power of our analysis model by showing a case study of a matrix multiplication implementation using the OpenCL programming language on NVIDIA G80 and G200 series GPUs which have slightly different memory architectures.

2 Thread Mapping

In current GPU programming models such as CUDA and OpenCL, each iteration of a data-parallel loop nest[1] is mapped to a thread. This approach often generates thousands or millions of threads, which are called *work-items* in OpenCL. Loop nests of more than one level are typically mapped to multi-dimensional thread maps that correspond to the depth of the loops.

The organization of multi-dimensional thread maps is akin to multi-dimensional arrays in that threads in the lowest (finest-grained) dimension are adjacent to each other, while the nesting level at higher dimensions are a fixed stride apart (the stride size depends on the extent of the lower dimensions). When executed, these threads are grouped into hardware scheduling units that we refer to as *thread-batches*[2] that are ordered in consecutively increasing thread IDs (after linearization in case of multi-dimensional thread configuration). All threads in a thread-batch execute the same instruction (i.e., SIMT execution), interleaving between first and second half of these threads, resulting in half thread-batch (16 threads) as an important memory access unit.

Consider the data-parallel loop nest shown in Listing 1 that computes a matrix multiplication. The outer two loop iterations (i.e., those iterated by *i1* and *i2*) access three two-dimensional arrays (**A**, **B**, and **C**). Intuitively, we map these two loop iterations to a two-dimensional thread map and have two choices, mapping A: *i1* to the lower dimension of the thread map (labeled *tx* in this paper) and *i2* to the higher dimension of the thread map (*ty*), mapping B: *i1* to *ty* and *i2* to *tx*. These mappings are shown in Listing 2. Note that the mapping B is represented by parentheses in the array reference index to save page space.

```
for(i1=0; i1< M; i1++)
    for(i2=0; i2< N; i2++)
        for(i3=0; i3< P; i3++)
            C[i1][i2] += A[i1][i3]*B[i3][i2];
```

Listing 1. Serial matrix multiplication.

```
int tx = get_global_id (0);
int ty = get_global_id (1);
for(i3=0; i3<P; i3++)
    C[tx(ty)][ty(tx)] += A[tx(ty)][i3]*B[i3][ty(tx)];
```

Listing 2. Two thread mappings.

As Listing 2 shows, the thread mapping changes the memory access patterns (e.g., *C[tx][ty]* in mapping A, versus *C[ty][tx]* in mapping B) and can have a huge impact on overall performance. Figure 1 compares performance and the number of uncoalesced global memory accesses as

[1] A data-parallel loop nest is a set of for loops wherein a set of arrays are referenced using the associated loop iteration variables

[2] A thread batch is called a *warp* (32 threads) on NVIDIA platforms

measured on an NVIDIA G80 (GeForce 8800 Ultra) and an NVIDIA G200 (GeForce GTX 285) for a range of input sizes. Performance differs by an order of magnitude for the two different thread mappings, independent of which GPU architecture is considered.

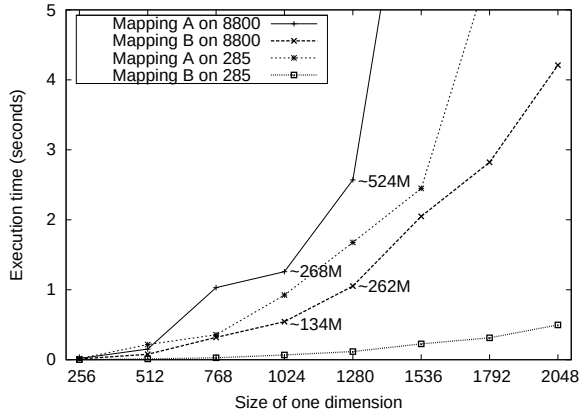


Figure 1. Impact of thread mapping on performance on NVIDIA GPUs. Note that uncoalesced access counts are shown for GeForce 8800 Ultra. A GTX 285 does not have a proper hardware counter to count global uncoalesced accesses.

3. Static Memory Access Pattern Analysis

The memory access patterns associated with data-parallel applications are a result of a combination of the inherent structure of each loop nest and the selected thread mapping. The per loop pattern is a function of the underlying computation, while the thread mapping is generally selected by the programmer. Given a particular thread mapping, the memory access pattern for each array present in a serial data-parallel loop nest can be captured into an affine form, information used to drive many loop optimization algorithms [3].

Consider a data-parallel loop nest of depth *D* that accesses an *M* dimensional array. The memory access pattern associated with accessing the array can be represented as a *memory access vector*, $\vec{m}$, which is a column vector of size *M*, starting from the index of the highest dimension. The memory access vector is then decomposed to the affine form: $\vec{m} = \mathbf{M}\vec{i} + \vec{o}$ where $\mathbf{M}$ is a memory access matrix whose size is $M \times D$, $\vec{i}$ is an iteration vector of size D iterating from the outermost to innermost loop, and $\vec{o}$ is an offset vector that is a column vector of size M that determines the starting index in an array. Note that we only consider loops whose accesses to arrays are affine functions of the loop indices and symbolic variables or can be changed to those forms. We have found that most scientific or engineering applications targeting GPUs possess affine access patterns [1].

Each column in the memory access matrix, $\mathbf{M}$, represents the memory access pattern of the corresponding loop level, and the number of columns is dictated by the depth of the loop nest. Each row in $\mathbf{M}$ and $\vec{o}$ represents the memory access pattern of a dimension of the array, and the number of rows is dictated by the number of dimensions in the array.

Thread mapping further divides memory access patterns into *inter-thread* and *intra-thread memory access patterns*. In our representation, the thread mapping simply replaces iteration indices with thread indices which we refer to as *tx*, *ty*, and *tz* in this paper to denote each thread dimension starting from the lowest to highest dimension. For example, the memory access patterns for array A before and after thread mapping A of Listing 2 are shown below (parentheses are used to denote the version after applying thread mapping). The arrays B and C are not shown due to space, but possess similar representations.

$$\vec{m}_A = \begin{bmatrix} i1(tx) \\ i3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i1(tx) \\ i2(ty) \\ i3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Inter-thread memory access patterns provide us with information on which threads access which data (e.g., threads with consecutive or non-consecutive thread IDs). We collect this information by assembling only the columns that are accessed by thread indices. For example, for the inter-thread memory access pattern associated with array A in the matrix multiplication example shown above is composed of the first two columns of its memory access matrix, $\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} tx \\ ty \end{bmatrix}$. The "1" in left upper corner of the inter-thread memory access pattern indicates that the higher dimension (the first row) of the array is accessed by threads in the lower dimension of the thread map (denoting that the first column corresponds to *tx* when performing the matrix multiply).

Intra-thread memory access patterns are represented by columns that corresponds to iteration indices which are mapped to threads (e.g., $\begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} i3 \end{bmatrix}$ for array A). This pattern indicates how the threads that are selected by their inter-thread memory access pattern actually access the data. The lower dimension (represented by the second row) of array A, as an example, is accessed by $i3$ in a linear fashion.

Using our mathematical memory characterization, all memory access patterns are classified into one of a number of categories.

True linear patterns refer to accesses where threads with consecutive (and increasing) thread IDs access contiguous (and increasing) memory addresses (decreasing valued thread IDs are called *true reverse linear*). These patterns are represented by "1" and "-1", respectively, in the last row of the column corresponding to *tx* in the inter-thread memory access matrix. Intra-thread memory access pattern do not play a role in this classification and the offset vector is zero. *False linear* patterns are similar to true linear patterns. The only difference is that they are accessed by threads with

Table 1. Summary of memory access patterns and their characteristics. H: work-group height, W: work-group width. †We consider local memory when there is any intra-thread pattern. ‡We assume that last dimension of a work-group is a multiple of a half-thread batch (i.e., 16 threads). §An uncoalesced memory access is considered only in global memory in this paper.

Pattern	Mathematical representation			Cost function‡	Memory selection†
	Inter-thread	Intra-thread	Offset		
True (False) linear	1 (0)	N/A†	0	0 (H, W, or H*W)	Global
True (False) reverse linear	−1 (0)	N/A†	0	0 (H, W, or H*W)	Global
True (False) shifted	1 (0)	N/A†	C	N/A§	Texture
True (False) overlapping	multiple 1 (0)	N/A†	0	N/A§	Texture
True (False) non-unit stride	C (0)	N/A†	0	N/A§	Texture
True (or false) random	Z (0)	N/A	0	N/A§	Texture
Intra-thread only	zero matrix	non-zero†	N/A	0	Constant

non-consecutive thread IDs. These patterns are represented by "1" and "-1" in other rows than the last row of the column corresponding to *tx* in the inter-thread memory access matrix. Figure 2 shows a graphical view of these patterns, along with their representations in the model.

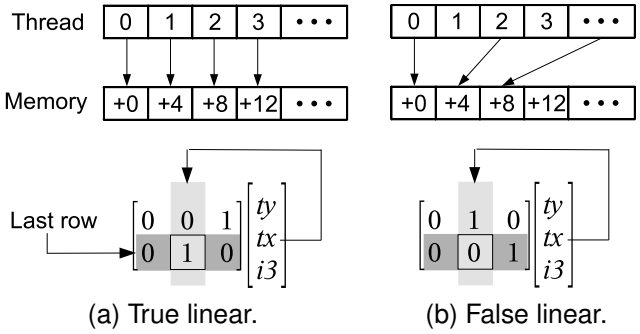


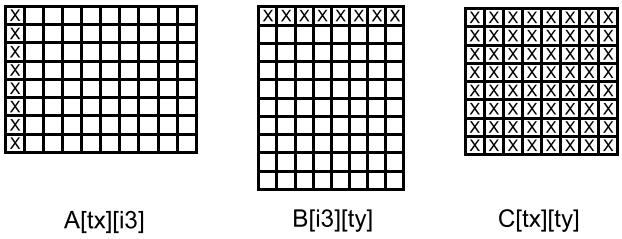
Figure 2. Linear patterns and their representations in the model.

Due to space, all other patterns are only briefly summarized in Table 1. A detailed description of all of the basic patterns considered in this work can be found in [2]. Cost functions are shown in the fifth column, capturing the approximate number of uncoalesced accesses in a single work-group, which are used to produce an improved thread mapping (i.e., shape of work-group).

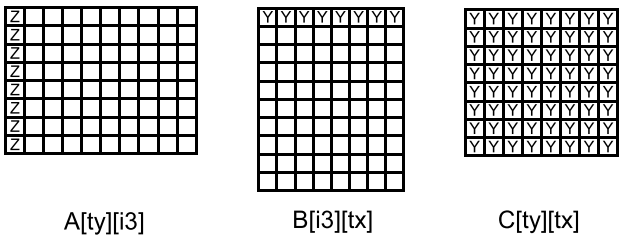
A geometrical view of the relationship between memory access patterns and cost functions (i.e., the number of uncoalesced access) in two thread mappings for our matrix multiplication example is shown in Figure 3. Thread mapping A (Figure 3a) exhibits all three false linear patterns resulting in a large number of uncoalesced accesses (e.g., global uncoalesced load (store): 524,288,000 (409,600) for an input matrix of size 1280×1280 elements), while thread mapping B (Figure 3b) exhibits two true linear patterns that are fully coalesced and one false linear pattern (e.g., global uncoalesced load (store): 262,144,000 (0) for an input matrix size of 1280×1280 elements). This accounts for the performance differences shown in Figure 1.

The last column in Table 1 shows the memory space selection for each pattern. Originally designed for graphics rendering, modern GPUs provide multiple programmable memory spaces each of which has distinct characteristics and usage. An appropriate utilization of these spaces depends on memory access patterns and can significantly increase effective memory bandwidth and performance [2, 4]. Note that the aforementioned patterns can be combined, resulting in compound patterns.

In the presentation we will provide more details of analysis and experimental results from more benchmark kernels.



(a) Thread mapping A: $i1$ to tx, $i2$ to ty.



(b) Thread mapping B: $i1$ to ty, $i2$ to tx

Figure 3. Two thread mapping schemes for the matrix multiplication kernel: only memory accesses of threads in the first work-group are shown. X, Y, and Z indicate uncoalesced, coalesced, and statically nondeterministic memory accesses respectively.

References

[1] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: an analytical representation of cache misses. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 317–324, New York, NY, USA, 1997. ACM.

[2] B. Jang, P. Mistry, D. Schaa, , and D. Kaeli. Exploiting memory access patterns to improve memory performance in data parallel architectures. *Parallel and Distributed Systems, IEEE Transactions on (submitted)*, 2010.

[3] S. T. Leung and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report TR 95-09-01, University of Washington, 1995.

[4] NVIDIA. CUDA Programming Guide 3.0, Feb 2010.