Algorithmic GPGPU Memory Optimization

Byunghyun Jang[1], Minsu Choi[2], and Kyung Ki Kim[3]

Abstract—**The performance of General-Purpose computation on Graphics Processing Units (GPGPU) is heavily dependent on the memory access behavior. This sensitivity is due to a combination of the underlying Massively Parallel Processing (MPP) execution model present on GPUs and the lack of architectural support to handle irregular memory access patterns. Application performance can be significantly improved by applying memory-access-pattern-aware optimizations that can exploit knowledge of the characteristics of each access pattern. In this paper, we present an algorithmic methodology to semi-automatically find the best mapping of memory accesses present in serial loop nest to underlying data-parallel architectures based on a comprehensive static memory access pattern analysis. To that end we present a simple, yet powerful, mathematical model that captures all memory access pattern information present in serial data-parallel loop nests. We then show how this model is used in practice to select the most appropriate memory space for data and to search for an appropriate thread mapping and work group size from a large design space. To evaluate the effectiveness of our methodology, we report on execution speedup using selected benchmark kernels that cover a wide range of memory access patterns commonly found in GPGPU workloads. Our experimental results are reported using the industry standard heterogeneous programming language, OpenCL, targeting the NVIDIA GT200 architecture.**

Index Terms—**GPU memory optimization, memory access pattern, thread mapping, GPGPU, GPU computing**

I. INTRODUCTION

GPUs are increasingly becoming an important component of high performance, heterogeneous, computing platforms by accelerating the ever demanding data-parallel portion of applications [1]. GPUs achieve high performance by executing thousands of kernel instances (i.e., threads) in parallel on different data points using a Single Instruction Multiple Threads (SIMT) model while hiding memory latency exploiting zero-overhead hardware thread switching. Peak performance of a single GPU has reached the multi-TFLOPS level; the performance provided by GPU-computing continues to increase at a rate well surpassing that of multi-core CPU performance growth [2].

Effectively reaping the benefits of GPUs, however, is a very challenging task. It requires a deep understanding of the intricacies of the underlying hardware architecture and associated close-to-the-metal programming model. The GPU hardware architecture is designed for high throughput computing by maximizing memory bandwidth to feed thousands of parallel threads running on many cores, while managing a certain amount of memory latency. The programming model requires programmers to explicitly map two levels of threads to data points while considering hardware resource

Manuscript received Feb. 1, 2014; accepted Apr. 29, 2014
A part of this work was presented in International SoC Design Conference (ISOCC), Busan in Korea, November 2013.
[1] Department of Computer and Information Science, The University of Mississippi, University, MS, 38677 USA
[2] Department Electrical & Computer Engineering, Missouri University of Science & Technology, Rolla, MO, USA
[3] Department Electrical & Computer Engineering, Daegu University, Gyeongsan, South Korea
E-mail : kkkim@daegu.ac.kr

constraints. Given the execution model where groups of threads access memory simultaneously, inattentive thread mapping generates memory access patterns that conflict in the underlying memory organization. These undesirable memory accesses are serialized into a large number of small, performance-starving memory transactions and cause cores to sit idle. Programmers usually apply *trial-and-error* to tune their code, which is not only inefficient, but often leaves the resulting code far from optimized.

In In this paper we apply static memory access pattern analysis to address several programming and optimization challenges that arise when mapping serial data-parallel loops onto massively multithreaded data-parallel GPU hardware. These challenges include memory space selection, thread mapping, and work group sizing. First we present enhancements to our basic memory access pattern model that was introduced in [3]; our new extensions fully consider the impact of two-level thread mapping during memory access analysis. Using this enhanced analysis model, we present a methodology that finds the best two-level thread mapping while considering tradeoffs imposed by thread mapping and work group sizing. These additional constraints are essential if we want to fully exploit GPU acceleration. The contributions of this work can be summarized as follows:

- We show how thread mapping and work group size impact memory access performance, key issues receiving little attention in prior work (Section II-B and Section II-C).
- We extend the memory access pattern analysis model proposed in [3] to include the impact of two-level thread mapping on memory access patterns (Section III).
- We extend the algorithmic memory space selection proposed in [3] and detail the use of local memory, which is a very challenging yet important optimization task [4-9] (Section IV).
- We propose a methodology that algorithmically searches for optimized thread mappings and work group sizes by introducing a new set of constraint metrics (Section V).
- We provide experimental results that demonstrate the effectiveness of our approach on a diverse set of benchmarks using the industry standard heterogeneous programming language, OpenCL [10] (Section VI).

The rest of the paper is organized as follows. The next section briefly summarizes the background of this work, including the available memory spaces found on current GPU architectures, and the impact of thread mapping and work group size on performance. Section III reviews the mathematical model used to characterize memory access patterns (first introduced in [3]) and describes how two-level thread mapping is incorporated into this model. In sections IV and V, we show two uses of our analysis model, demonstrating how memory space selection, thread mapping and work group sizing are accomplished. We report experimental results in section VI and discuss the limitations of our approach and future work in section VII. We conclude the paper in section VIII.

II. BACKGROUND

1. GPU Memory Spaces

Driven by the demand of real-time graphics rendering, GPUs are comprised of multiple memory spaces that have very different characteristics aimed at improving the performance of the device. For applications to obtain high performance on GPUs, the characteristics and requirements of these memory spaces must be well understood. Major factors to consider are whether the memory is physically located on-chip or off-chip (relative to the compute units), whether the memory is automatically cached, and its scope and access requirements.

Global memory is the default memory space for input and output data. It is an off-chip memory and not cached, though it is the most flexible in terms of accessibility and size. Since accesses to global memory are long latency, it is imperative to completely utilize the full memory bus width to deliver as much data as possible in the smallest number of transactions to the compute units. As such, performance is very sensitive to the resulting data access pattern when working with global memory. Ideal access to global memory occurs when a scheduled group of threads requests data in the same address range, allowing requests to be combined into a few accesses that fully utilize the memory bus—thread groups that are

scheduled together are called *warps*[1] and combining memory requests is known as *coalescing*. When memory accesses patterns do not fully exploit the properties of global memory, another memory space is often more desirable to use [9, 11, 12].

Constant memory is an off-chip memory that is cached on-chip. As the name suggests, this memory space is read-only and can hold only a small amount of constant data. The single-banked cache of constant memory has broadcast capability, and thus the bandwidth of constant memory is maximized when all threads in a warp read the same memory address. Once cached on-chip, data access latency is as fast as a register-based access, though the throughput of the cache is decreased by a factor equal to the number of different requests within a warp. Note that the inherent properties of constant memory are not appropriate for every pattern associated with read-only data; constant memory is best utilized when all threads of a warp access the same data element simultaneously.

Texture memory is an abstraction where global memory is accessed through an on-chip hardware texture unit. It is designed and optimized for graphics texture mapping (naturally exploiting 2D locality). The texture memory offers a number of benefits over global memory: 1) it is cached on-chip, 2) it provides better performance for uncoalesced accesses, and 3) it has hardware support for address calculation, automatic boundary checking, and data interpolation. Random memory access patterns are better served by texture memory than with global memory [9].

Local memory is the small, on-chip scratchpad memory on each compute unit. Utilizing local memory is key to harnessing the full computing power of the GPU [4, 9].

Given the large number of processing elements within a GPU, the long-latency accesses to off-chip memory are commonly the bottleneck of a compute kernel. To achieve high performance, it is necessary to load data into low-latency local memory (we use a term, *prefetch*, throughout this paper) as much as possible. Since the memory is very limited in size and partitioned by active work groups simultaneously running on a compute unit, the amount of local memory used is one of the factors that determine the number of work groups that can reside on a compute unit simultaneously. Nevertheless, the use of local memory significantly increases software development complexity. It requires programmers to recognize the potential benefit from data prefetching and then manually partition the data. Note that local memory is only beneficial when there is spatial locality among threads within a work group to compensate for the extra cost of explicitly loading data from off-chip memories.

Registers and *Private Memory* are used to store automatic variables within a kernel. Compute units have a large number of registers that must be statically allocated to a thread for its entire lifetime. Local arrays and any data that does not fit into registers are *spilled* into private memory, which is located off-chip as a part of global memory. Just as local memory size is a constraint on performance, register allocation is a major limiting factor on the number of threads (or work groups) that can be active on a compute unit. Since neither of these memory spaces is explicitly programmable by the programmer, they are not considered during our memory space selection in this work. However, determining an appropriate work group size is highly dependent on register constraints, and so we still need to carefully consider register usage when designing our work group sizing algorithm.

2. Impact of Thread Mapping on Performance

In current GPGPU programming models such as OpenCL and CUDA, each iteration of a serial, data-parallel loop nest[2] is mapped to a thread. Loop nests of more than one level are typically mapped to multi-dimensional thread configurations whose dimensions are less than or equal to the depth of the loops. The organization of multi-dimensional thread configurations is akin to multi-dimensional arrays in that threads in the lowest (finest-grained) dimension are adjacent to each other, while threads at higher dimensions are a fixed stride apart (the stride size depends on the extent of the lower dimensions).

When a kernel is executed, threads are grouped into hardware scheduling units that we refer to as *thread*

[1] In the NVDIA GT200 architecture, warps are the groups of 32 threads, and memory accesses are issued in units of half warps. Neither CUDA nor OpenCL have explicit representations of wraps.

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

batches (called *warp* on NVIDIA platforms). The scheduled threads are ordered with consecutively increasing thread IDs (performed after linearization in the case of multi-dimensional thread configurations). All threads in a warp execute the same instruction (i.e., SIMT execution). The first and second halves of the warp (16 threads each) interleave execution and issue memory instructions, thus a *half warp* is the key unit to consider in terms of memory access.

Consider the serial data-parallel loop nest shown in Listing 1 that computes a matrix multiplication. The loops iterate over three two-dimensional arrays (i.e., **A**, **B**, and **C**). Intuitively, we would like each thread in the GPU kernel to compute a single value in the output array. To do this we map the two outer loop iterators (*i1* and *i2*) to a two-dimensional thread configuration. We have two choices to consider here:

- Mapping α: map *i1* to the lower dimension of the thread configuration (labeled *tx* throughout this paper) and *i2* to the higher dimension of the thread configuration (labeled *ty* throughout this paper), or
- Mapping β: map *i1* to *ty* and *i2* to *tx*.

These two mapped kernels are shown in Listing 2 and Listing 3 respectively.

```
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 ] += A[tx][i3]*B[i3 ][ ty ];
```

Listing 2. Thread mapping α (*i1* maps to *tx* and *i2* maps to *ty*).

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

Listing 3. Thread mapping β(*i1* maps to *ty* and *i2* maps to *tx*).

As Listing 2 and 3 show, the thread mapping changes the memory access pattern (e.g., *C[tx][ty]* in mapping α, versus *C[ty][tx]* in mapping β) and this can have a huge impact on overall performance. Fig. 1 compares the performance of the two mappings on an NVIDIA GT200 architecture (GeForce GTX 285) for a range of input sizes; we see that performance differs by an order of magnitude. We will show how this performance sensitive thread mapping is represented in our model in Section III-A and III-B and how the model is used to derive optimized thread mappings in Section V.

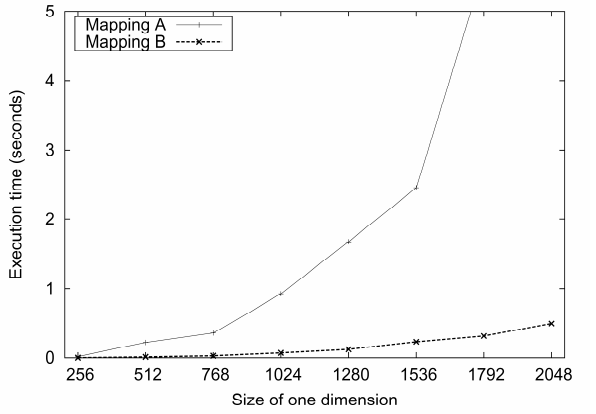


Fig. 1. Impact of thread mapping on performance on a NVIDIA GT200 GPU.

3. Impact of Work Group Size on Performance

To enable scaling of the number of compute units while maintaining same level of hardware utilization, GPUs impose the requirement that compute units are autonomous execution units (i.e., no direct communication is possible between compute units). Each compute unit has the same amount of local hardware resources on which only threads (or work groups) assigned to it can operate. Synonymous with compute units, the programming model divides threads into independent groups (called *work groups* in OpenCL), each of which is identical in the number of threads and resource usage, so as a problem scales in size, additional work groups are created, but no other changes to the algorithm are required. These two concepts, work groups and compute units, are the enablers of scalability in GPU computing on the software side and hardware side, respectively.

Optimization challenges arise due to the fact that per-compute-unit hardware resources determine the number

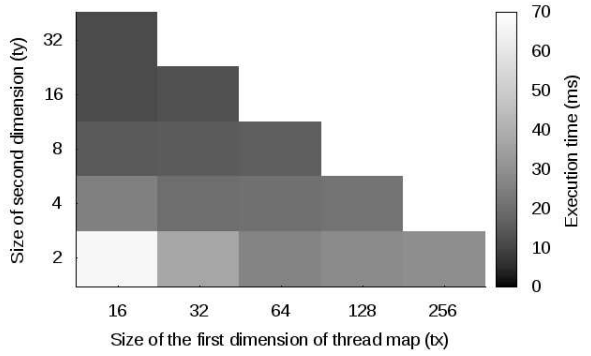


Fig. 2. A map view of the execution times of matrix multiplication comparing different work group size configurations (the shade of gray indicates the execution time, darker is shorter).

of work groups and threads that can run simultaneously. Programmers need to carefully consider the work group size while factoring in these hardware resources to maximize hardware utilization.

To understand the impact of work group size on performance, we modified the NVIDIA SDK version of the matrix multiplication kernel that utilizes local memory to be able to test a range of work group size dimensions [13]. Fig. 2 illustrates the performance of each work group configuration. The performance of the kernel when the work group size is 512 (32 (*ty*) × 16 (*tx*), upper-left corner in the graph) is approximately 5.9× better than when it is 32 (2 (*ty*) × 16 (*tx*), bottom-left corner). Even given the same work group size, 512 for example, the performance differs by 2.6× between two different configuration (32 (*ty*) × 16 (*tx*), upper-left corner and (2 (*ty*) × 256 (*tx*), bottom-right corner).

Finding a good work group size is not intuitive, and is very complex when programming with local memory. Static constraints such as the maximum number of active threads and work groups allowed on a single compute unit, and the maximum number of threads per work group must be considered along with the register usage per thread and the local memory usage per work group. Changing the size or dimensions of a work group may end up resulting in very different performance. In this work, we attempt to assist the programmer by suggesting a thread mapping and work group size that best utilizes the underlying hardware by means of static memory access pattern analysis. In Section V, we present an algorithm that recommends sizes and dimensions that are likely to perform well for a given work group.

III. Static Memory Access Pattern Analysis

In Section II-B, we showed how critical it is to choose a proper mapping of data to thread structure and to choose proper work group sizes. Before we present our extension to the previous model which fully incorporates thread mapping, we briefly recap the basic mathematical model presented in [3] that captures memory access patterns in a serial data-parallel loop nest. We then present our extensions in following subsections.

Consider a data-parallel loop nest of depth D that accesses an *M*-dimensional array. The memory access pattern of the array can be represented as *a memory access vector*, $\vec{m}$, which is a column vector of size M that describes how each dimension of the array is accessed. For example, in Listing 1, array **A** is a two-dimensional array where the first dimension is traversed by *i1* and the second dimension is traversed by *i3*; the corresponding memory access vector is:

$$\vec{m}_A = \begin{bmatrix} i1 \\ i3 \end{bmatrix}$$

Although the concept is simple for a trivial example, significant insight can be gained in the general case by decomposing the memory access vector to its affine form:

$$\vec{m}_A = M\vec{i} + \vec{o}$$

where 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 a column vector of size M that denotes the starting offset in each dimension of the array. Note that we only consider loops with array accesses that can be represented as affine functions of loop indices and symbolic variables (e.g., height, width, etc.). We have found that this restriction does not limit us since most scientific applications involve loops possessing affine access patterns [14]. The memory access patterns of the serial matrix multiplication loop nest shown in Listing 1 are captured as follows. A more detailed explanation of how these memory access patterns are captured is available in [3].

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

$$\vec{m}_B = \begin{bmatrix} i3 \\ i2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i1 \\ i2 \\ i3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\vec{m}_C = \begin{bmatrix} i1 \\ i2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i1 \\ i2 \\ i3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Although our model is based on a simple affine form commonly found in other well-known loop optimization models, our proposed analysis model is simpler yet powerful enough to explain all important memory access behaviors present on GPUs which other models are not capable of. Existing models such as the polyhedral [7, 8], distance/direction vector and unimodular [15, 16] are developed to improve locality and minimize synchronization cost targeting single or multicore processors. In contrast, our analysis model is developed under consideration of massively multithreaded data-parallel GPUs where global data communication is not of primary interest and a batch of threads access memory simultaneously. Our model also incorporates two-level thread hierarchy, SIMT execution model, and multiple memory spaces with distinct access preference.

1. Incorporating Thread Mapping

In our representation, thread mapping simply replaces iteration indices with thread indices. We use *tx*, *ty*, and *tz* to denote the thread dimensions, starting from the lowest dimension. For example, the memory access patterns for arrays **A**, **B**, and **C** after thread mapping α (Listing 2) and β (Listing 3) are shown below (parentheses are used to denote thread mapping β).

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

$$\vec{m}_B = \begin{bmatrix} i3 \\ ty\,(tx) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} tx\,(ty) \\ ty\,(tx) \\ i3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\vec{m}_C = \begin{bmatrix} tx\,(ty) \\ ty\,(tx) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} tx\,(ty) \\ ty\,(tx) \\ i3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Our thread mapping classifies memory access patterns into inter-thread and *intra-thread* patterns. The inter-thread memory access patterns tell us how thread dimensions are mapped to array dimensions. We collect this information by extracting the columns of the memory access matrix that are accessed by thread indices. For example, the inter-thread memory access patterns associated with arrays **A**, **B**, **C** in the matrix multiplication example shown above are composed of the first two columns of their respective memory access matrices, respectively:

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} tx\,(ty) \\ ty\,(tx) \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} tx\,(ty) \\ ty\,(tx) \end{bmatrix},$$
$$and \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} tx\,(ty) \\ ty\,(tx) \end{bmatrix}$$

In thread mapping α the "1" in the upper left corner of the inter-thread memory access matrix for array **A** indicates that the highest dimension of the array (the first row) is accessed by threads in the lowest dimension of the thread map. In other words, the first column of the memory access matrix corresponds to *tx* when performing the matrix multiplication. In thread mapping β, the same "1" indicates that the highest dimension of the array is accessed by threads in the highest dimension of the thread map, *ty*.

The intra-thread memory access patterns are the columns of the memory access matrix that correspond to iteration indices that are not mapped to threads. In the matrix multiplication example, the intra-thread access patterns are represented by the third column of each memory access matrix:

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} [i3], \begin{bmatrix} 1 \\ 0 \end{bmatrix} [i3], and \begin{bmatrix} 0 \\ 0 \end{bmatrix} [i3]$$

for matrices **A**, **B**, and **C**, respectively. These patterns indicate how the threads that are mapped by the inter-thread memory access pattern actually access the data. For example, the lowest dimension of array **A**

(represented in the second row of its memory access matrix), is accessed by *i3*, meaning that consecutive data elements will be access by each thread. Matrix **B**, on the other hand, is accessed by *i3* in it's highest dimension, so individual threads will access elements that are a stride apart. Finally, array **C** does not use *i3* as an iterator, so the data that Matrix C is accessing will not change within the third loop. Later we will show how the inter-thread and intra-thread memory access patterns can be used to optimize memory accesses on the GPU.

2. Extended Pattern Characterization

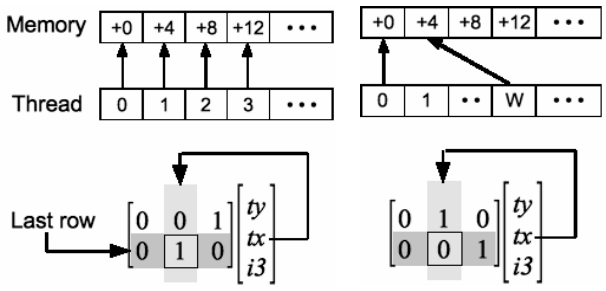
In our earlier work [3], we classified memory access patterns into a number of categories: *linear*, *reverse linear*, *shifted*, *overlapping*, *non-unit stride*, and *random* patterns using our mathematical representation. In this section we extend our analysis to capture more exact memory access patterns and show that they are in fact necessary for proper memory space selection, thread mapping, and work group sizing.

In order to explain how thread mapping impacts the performance of memory accesses, we extend each of our memory access pattern classifications to include true and false sub-patterns. For example, the linear pattern is divided into *true linear* and *false linear* patterns, depending on which thread dimension accesses each dimension of the array.

The true linear pattern refers to accesses where threads with consecutive thread IDs access contiguous and increasing memory addresses. These patterns are represented by a "1" in the last row of the column corresponding to *tx* in the inter-thread memory access matrix (or a "-1" in the reverse linear case). Intra-thread memory access patterns do not play a role in this classification.

The false linear pattern is similar to the true linear pattern except that consecutive memory addresses are accessed by threads with non-consecutive threads IDs. These patterns are represented by "1" or "-1" in any row aside from the last row of the column that corresponds to *tx* in the inter-thread memory access matrix.

All other patterns are similarly subdivided into true and false patterns. Fig. 3 shows a graphical view of the true and false linear patterns, along with their representations in our model.



(a) True linear (b) False Linear

Fig. 3. Linear patterns and their representation in our model (*W* denotes the width of a work group).

3. Coalesced Memory Accesses

When threads of a thread batch execute a memory instruction, the GPU hardware attempts to convert these separate accesses into one or a few *coalesced* accesses whenever possible—the exact number of accesses required depends on the size of data type and the access pattern. Note that the extent to which GPUs can coalesce memory accesses, and hide the penalty for non-linear accesses, is highly dependent on the underlying hardware.

In our memory access pattern analysis model, a true linear pattern and a true reverse linear pattern are considered as fully coalesced memory accesses. Note that all other patterns could result in various types of coalesced access patterns, but we consider them as uncoalesced patterns for simplicity in this work.

4. Accesses to Same Memory Address

Identifying when different threads read from the same memory address is crucial for the utilization of constant memory. Using our model, we can recognize when threads will read from the same address based on two requirements: 1) there are no inter-thread memory access patterns (there are all zeros in the inter-thread entries of the memory access matrix), and 2) there exists an intra-thread memory access pattern (at least one non-zero element), where the non-zero element does not correspond to a loop variable of the input problem size. If the loop variable value is a function of the problem size, then the array could potentially be a candidate for data prefetching.

5. Data Prefetch

Data prefetch is a key mechanism for removing memory

bottlenecks and needs to be used effectively if we want to achieve peak performance on GPUs. Prefetching can be implemented by using local memory after loop strip mining is performed[3]. Efficient use of local memory can significantly boost performance [4, 9, 12].

In our representation, the potential benefits of data prefetch can be explored by characterizing any intra-thread memory access patterns present. If we detect an intra-thread pattern, this is typically due to loops that are not explicitly mapped to threads (e.g., loop bodies that remain inside a kernel body, even after thread mapping is performed). There are potential benefits to prefetching this data to local memory and then access those elements locally. In this work we only consider linear intra-thread patterns as candidates for prefetch, since other patterns tend to be less deterministic and make it difficult to predict the potential benefits of prefetching. Note that there is no benefit from prefetching data to local memory in a kernel that does not have an intra-thread access pattern (e.g., vector addition kernel).

IV. ALGORITHMIC MEMORY SPACE SELECTION

Having described the characteristics of each memory space in section II-A and our representation of static memory access patterns and analysis in section III, we present a detailed algorithm to identify the best memory space to place an array.

Our proposed algorithm (see the algorithm 1) takes as input the memory access pattern, the size of the array, the read-write information of each array instance, and a thread index vector and outputs the memory space selected for the array. While scanning input information, the algorithm first considers the read-write characteristics for the array (line 4). If there are any writes to the array, constant memory will not be selected. If the array has only read accesses, then we first check whether the memory access pattern is suitable for constant memory: the size of the array must be small enough to fit in the available constant memory and all threads must read the same address (line 4). Next the algorithm checks if the memory access pattern meets the requirements for using local memory (line 7): and potential for using data prefetch. Finally the algorithm checks whether the memory access pattern is coalesced (line 24), and if so, then global memory is selected. Otherwise texture memory is selected. A similar procedure is applied to read-write and write-only data, skipping any requirements checking for constant memory.

Note that after running the algorithm, multiple memory spaces may be selected for a single array due to different memory access patterns for each array instance. If this is the case, then we make a final selection using the following priorities: for read-only data 1) texture, 2) global, 3) local, 4) constant memory, for read-write data 1) global, 2) local, and for write-only data 1) texture, 2) global, 3) local memory.

Algorithm 1:Memory selection

input : M and $\vec{o}$ pairs of all instances of an array (SMA), the size, read-write, and a thread index vector ($\vec{t}$)
output: Memory space (MS)

```
1  MS ←0 ;
2    if read-only then
3      for each M ∈ SMA do
4        if Small && Same address read then
5          MS ← MS + Constant;
6        end
7        else if Data preftech then
8          MS ← MS + Local;
9        end
10       else if Coalesced then
11         MS ← MS + Global;
12       end
13       else MS ← MS + Texture;
14     end
15   end
16   else
17     for each M ∈ SMA do
18       if Data prefetch then
19         MS ← MS + Local;
20       end
21       else if Uncoalesced && Write-only then
22         MS ← MS + Texture;
23       end
24       else
25         MS ← MS + Global;
26       end
27     end
28   end
```

[3] Loop strip mining is a technique to split a single loop into a nested loop. The resulting inner loop iterates over a strip of the original loop - the number of iterations of the inner loop is known as the strip length.

V. ALGORITHMICALLY SELECTING THREAD MAPPINGS AND WORK GROUP SIZES

As shown in Section II, thread mappings and work group sizes can have an enormous impact on the performance of GPU programs. On NVIDIA GPUs, uncoalesced memory accesses are the biggest detriment to memory bandwidth, and choosing thread mappings that are inappropriate for the memory access patterns of a kernel will significantly degrade performance. Alternatively, selecting proper thread mappings and appropriately sized work groups can allow for very efficient use of local memory to prefetch and cache data, alleviating a large amount of memory pressure. Our methodology uses the estimated number of uncoalesced memory accesses, and the estimated amount of data that can be stored in local memory as our cost and gain, respectively, to evaluate the efficiency of combinations of thread mappings and work group sizes.

1. Cost and Gain Calculation

Our cost and gain calculation is performed on a per-work-group basis, since all work groups have the same memory access behavior and hardware resource usage. Our proposed cost estimation is computed as follows. Fully coalesced memory patterns (i.e., true linear or true reverse linear) have no cost. False linear and false reverse linear patterns, on the other hand, have a cost of either "H", "W", "D", or the product of a combination of these, where "H", "W", "D" are the height, width, and depth of the work group. The values that are chosen for the cost depend on the mapping of thread dimensions to array access dimensions. For example, in the case of the mapping α of the matrix multiplication kernel, array **A** exhibits a false linear pattern because adjacent threads are accessing different rows of the array. Since the "1" in the inter-thread pattern corresponds to *tx*, and *tx* is the width of the work group, the cost becomes "W" in this case. The cost of array **B** is zero because *tx* is not involved and we assume that higher dimensions (*ty* in this case) of the thread map are always a multiple of a half warp size. The cost for the array **C** is "H*W" due to the false linear pattern present in both *tx* and *ty*. Therefore, when the work group size is 8 (*ty*) × 32 (*tx*) the cost associated with arrays **A**, **B**, and **C** are 32 (*tx*), 0, and 8 (*ty)* × 32 (*tx*), respectively. Fig. 4 shows a graphical view of our cost calculation for the two mappings of the matrix multiplication kernel.

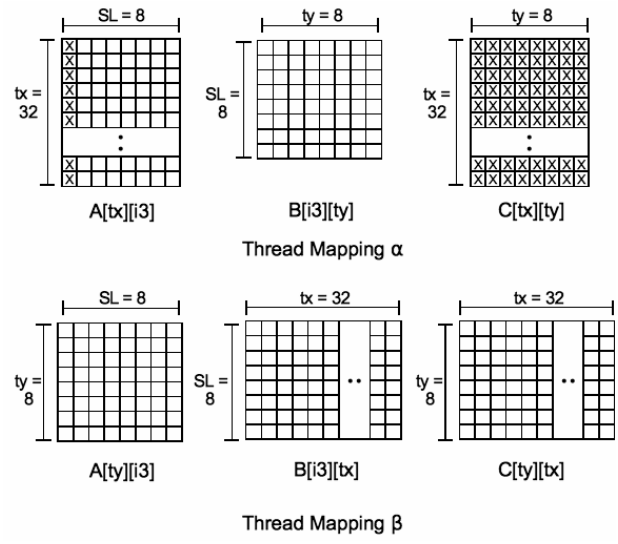


Fig. 4. Cost in the two thread mapping schemes for matrix multiplication kernel when the size of work group is 32(*tx*) ×8(*ty*): X indicates uncoalesced memory accesses and SL denotes the loop strip length.

The estimated gain is based on the number of elements in local memory, while taking loop strip mining into account. Note that data prefetching always entails loop strip mining. The strip length cannot exceed the size of the smallest dimension of thread map; otherwise we could not specify the entire size of the prefetched data simultaneously using local thread IDs. The gain is calculated by multiplying the size of smaller dimensions and size of the strip length. For example, for the *α* mapping matrix multiplication kernel, when the work group size is 8 (*ty*) × 32 (*tx*) the local memory accesses for array **A** and **B** are 32 (*tx*) × 8 (*strip length*) and 8 (*ty*) × 8 (*strip length*), respectively and the gains become 8 (*smaller of 32 and 8*) × 8 (*strip length*) and 8 (*smaller of 8 and 8*) × 8 (*strip length*). Choosing the smaller dimension takes into account the fact that selecting a smaller strip dictates a higher inner loop trip count. The final cost and gain are the total of the individual costs and gains for each array.

2. Resource Constraints and Search Space

There are two per-compute-unit hardware resources to consider when searching for a good work group size:

registers and local memory. Registers are a hardware resource that is shared among all active threads on a single compute unit (directly related to the number of active threads). Local memory is shared by active work groups (directly related to the number of active work groups). The following two equations show how the two hardware resources are related to each other in terms of work group size.

$$\frac{\#Registers}{Thread} \times \frac{\#Threads}{WG} \times \frac{\#WG}{CU} < \frac{\#Registers}{CU} \quad (1)$$

$$\frac{\#Local\ Memory}{Work\ Group} \times \frac{\#WG}{CU} < \frac{\#Local\ Memory}{CU} \quad (2)$$

Eq. (1) implies that given a particular register pressure ratio (denoted as *#Registers / Thread* in the equation) computed on a per-kernel-body basis, the number of active work groups (denoted as *#WG / CU*) must decrease as the work group size (denoted as *#Threads / WG*) increases. Eq. (2) indicates that if the number of active work groups decreases, the amount of local memory that each work group can use (denoted as *#Local Memory / WG*) increases. When a kernel does not use local memory, Eq. (2) can be ignored. When multi-dimensional thread mapping is involved, there are a number of choices for different combinations for selecting the dimension sizes of a work group. In this case, we search for all possible choices where the lowest dimension size (*tx*) is multiple of a half warp size (the key unit for memory accesses). For example, the work group size of 256 in a two-dimensional thread mapping has 2 (*ty*) × 128 (*tx*), 4 (*ty*) × 64 (*tx*), 8 (*ty*) × 32 (*tx*), and 16 (*ty*) × 16 (*tx*) as possible combinations.

3. Our Proposed Methodology

Hardware vendors usually provide ideal work group sizes for their GPUs [9, 12, 17, 18][4]. Since hardware resources almost always constrain the work group size, our proposed search space starts from the largest ideal size and progresses down toward the minimum acceptable size until we find enough good candidates (specified by the programmer). For NVIDIA GPUs, the ideal work group size is not the maximum possible number of active threads per compute unit, as creating multiple work groups will allow for better hardware utilization.

Our methodology to search for a thread mapping and work group size is summarized in Algorithm 2. Given the memory space selections provided by Algorithm 1, we start with the ideal work group size that allows for at least two work groups to be active per compute unit after accounting for register pressure (Eq. (1) is used). From there, we decrease the work group size using predefined ordered sets of work group configurations, calculating the cost and gain of each configuration. Finally, the selection of candidates that produce optimized thread mappings and work group sizes is based on the following priority. When local memory is used, (i.e., there exists positive gain), we use 1) high gain, 2) low cost, 3) large work group size, otherwise we use, 1) low cost, 2) high occupancy, 3) high number of active work groups. Note that when data is prefetched into local memory, the number of device memory accesses tends to decrease significantly, implying that the associated cost is reduced. Note also that if local memory is not used, the work group size does not play any role.

Algorithm 2: Searching for beneficial thread mappings and work group sizes.

input : Memory access patterns and memory space selections for arrays, a thread index vector, the number of registers used, the desired number of candidates for good work group sizes (C)

output : Candidates for a good thread mapping and work group size

```
1  Compute the largest starting work group size from the
   predefined work group configuration table (WGT) that
   allows at least 2 active work groups;
2  while WGT is not empty do
3      Compute the number of active work groups (AWG)
       based on the register count;
4      Compute the amount of local memory required
       (ALM);
5      if ALM ≤ local memory size then
6          Compute the cost and gain;
7          Push it into the priority queue (PQ) based on our
           priority;
8      end
9      Get the next smaller work group size configuration
       from WGT;
10 end
11 Output the first C number of elements from PQ;
```

[4] The ideal work group size can be different across GPU architectures. NVIDIA recommends that GPUs with compute capability 1.3 (e.g., GTX 285) should have between 256 and 512 threads per work group.

Table 1. An example of algorithmic thread mapping and work group size search: matrix multiplication, †*ty*×*tx* (total number of threads), * measured execution time in milliseconds when input matrix size is 1024×1024. Costs are all zeros. WGS=Work Group Size, AWG=the number of Active Work Groups, OR=Our Rank

#	WGS†	AWG	GAIN	OR	MET*
1	2×256 (512)	2	2×2 + 2×2 (8)	9	29.31
2	4×128 (512)	2	4×4 + 4×4 (32)	6	21.49
3	8×64 (512)	2	8×8 + 8×8 (128)	3	15.58
4	16×32 (512)	2	16×16 + 16×16 (512)	1	12.73
5	32×16 (512)	2	16×16 + 16×16 (512)	1	11.30
6	2×128 (256)	4	2×2 + 2×2 (8)	10	28.16
7	4×64 (256)	4	4×4 + 4×4 (32)	7	20.77
8	8×32 (256)	4	8×8 + 8×8 (128)	4	15.35
9	16×16 (256)	4	16×16 + 16×16 (512)	2	11.19
10	2×64 (128)	8	2×2 + 2×2 (8)	11	26.15
11	4×32 (128)	8	4×4 + 4×4 (32)	8	19.70
12	8×16 (128)	8	8×8 + 8×8 (128)	5	14.77

Consider the mapping β in the matrix multiplication kernel. According to Algorithm 1, arrays **A** and **B** possess access patterns that can benefit from data prefetching using local memory, and the output array **C** does not, so it is placed in global memory. The search begins with the largest ideal work group size, 512. Since the register pressure allows for 2 active work groups on a single compute unit, it is a valid starting size. Because the algorithm uses a two-dimensional thread mapping, we search all predefined configurations of a two-dimensional work group (attempt #1 through #12 in Table 1). Note that the lowest dimension (*tx*) is a multiple of the warp size. For each work group configuration containing the same total number of threads, we compute the cost and gain and store it in a *priority queue* data structure. The algorithm continues this process until it visits all entries of the predefined work group size configuration table. Finally, the algorithm outputs the specified number of elements (denoted as C in the algorithm) in the priority queue. Table 1 summarizes the work group search process along with metrics, our rank, and actual measured performance in the order of search.

VI. EXPERIMENTAL RESULTS

We evaluate our proposed methodology using four different algorithms: two common numerical kernels and two complex real world kernels. These four benchmarks contain a wide range of memory access patterns, thread configurations, and resource requirements. For comparison purposes, two different versions of the algorithm were implemented for each benchmark: one using only the default (global) memory for each array, and the other using the memory spaces selected by Algorithm 1. For the latter implementation, we use our framework to generate suggested thread mappings and work group sizes, and we compare the performance of our suggested configurations against actual execution times.

The experiments were conducted on an NVIDIA Geforce GTX 285 GPU using the OpenCL programming language and the CUDA 4.2 Toolkit and SDK (graphic device driver 295.41). The host system is configured with a 2.66 GHz Intel Core 2 Duo running 64-bit Linux, with 2 GB of main memory.

The first kernel evaluated is *vector addition*. This is a very simple kernel, and is usually the first kernel that a programmer learns for GPU programming. In vector addition, there is a single loop that iterates over three one-dimensional arrays, and each array exhibits the same access pattern: a true linear inter-thread pattern and no intra-thread pattern. Since the kernel has only one loop, the only possible thread mapping is to a one-dimensional thread configuration. The number of registers required per thread is 3.

According to Algorithm 1, global memory is selected for all arrays. Since the number of registers used does not restrict the recommended work group size, we start with the largest work group size that hardware vendor recommends (512). For any configuration, the cost and the gain are always zero since all memory access patterns are coalesced and no local memory is utilized. Algorithm 2 repeats the cost and gain calculations on the ordered, predefined set of work group sizes and pushes each result into the priority queue. The work group configurations that remain at the frontend of the priority queue become the candidates for selection. Table 2 summarizes the experimental results. We see that our ranking of work group sizes closely aligns with the actual measured performance.

The second benchmark kernel is *matrix-vector multiplication*. This kernel has a two-level nested loop that iterates over arrays that differ in their number of

Table 2. Experimental results of vector addition when the input size is 2^{25} and the register count is 3. Acronyms and units are the same as in Table 1

#	WGS	AWG	Cost	Gain	OC	OR	MET
1	512	2	0	0	100%	3	3.39
2	256	4	0	0	100%	2	3.38
3	128	8	0	0	100%	1	3.39
4	64	8	0	0	50%	4	3.60
5	32	8	0	0	25%	5	5.89
6	16	8	0	0	25%	5	11.61

dimensions. There are two one-dimensional arrays and one two-dimensional array. Again, there is only one choice for thread mapping: the memory access pattern of the one-dimensional output (write-only) array is $\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}tx\\ i2\end{bmatrix}$ and it exhibits a true linear inter-thread pattern with no intra-thread access. The memory access pattern of the one-dimensional input (read-only) array is $\begin{bmatrix}0 & 1\end{bmatrix}\begin{bmatrix}tx\\ i2\end{bmatrix}$ and possesses no inter-thread accesses and is a true linear intra-thread pattern. Finally, the memory access pattern of the two-dimensional input (read-only) array is captured as $\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}tx\\ i2\end{bmatrix}$ and exhibits a false linear inter-thread pattern and a true linear intra-thread pattern. Given the memory access patterns, algorithm 1 selects global memory for the first array. The second and third arrays exhibit the potential of benefiting from data prefetching, so for the second array texture memory is selected with prefetching to local memory, and for the third array global memory is selected, with prefetching to local memory.

In this kernel, the amount of local memory used is a major limiting factor that influences the work group size. Due to the local memory usage, the first valid work group size we can consider is 32. Table 3 summarizes the search process, and includes the cost and gain, measured execution time, and resulting speedup when compared to using only global memory.

The next kernel studied is a real world application that computes radiological paths—the most computationally expensive step in medical image reconstruction [19]. The particular algorithm is called the *improved Siddon* algorithm [20] and is the most widely used in its domain. The algorithm computes the contribution of the various

Table 3. Experimental results of *matrix-vector multiplicat*ion when each dimension of the input arrays are *4096* and the register count is 17. † Speedup over default global memory space. SU = Speedup, Other acronyms and units are the same as in Table 1

#	WGS	AWG	Cost	Gain	OR	MET	SU†
1	32	4	32	32×32+32 (1056)	1	3.25	6.57
2	16	4	16	16×16+16 (272)	2	3.95	6.87

parts of the body to the radiation received by a number of X-ray detector cells. For each radiological path from the radiation source to a detector cell, the local intensities of object cells (i.e., cells in the body) that the rays hit are integrated along the path. Given the number of detector cells and the radiation emission, radiological paths are calculated more than a million times. The kernel involves four arrays: one three-dimensional read-only array, one two-dimensional write-only array, and two one-dimensional read-only arrays. Given that we decide to map this loop to a two-dimensional thread configuration, there are two potential mappings. Here we show only the better of the mappings, which has the following memory access patterns (the order of the arrays is the same as the order when they were listed above).

$$\begin{bmatrix}0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & Z\end{bmatrix}\begin{bmatrix}tx\\ ty\\ i3\end{bmatrix},\ \begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix}\begin{bmatrix}tx\\ ty\\ i3\end{bmatrix},$$

$$\begin{bmatrix}1 & 0 & 0\end{bmatrix}\begin{bmatrix}tx\\ ty\\ i3\end{bmatrix},\ \begin{bmatrix}0 & 1 & 0\end{bmatrix}\begin{bmatrix}tx\\ ty\\ i3\end{bmatrix}$$

Following the order shown above, memory access patterns are classified as: no inter-thread and random intra-thread pattern, true linear inter-thread and no intra-thread pattern, true linear inter-thread and no intra-thread pattern, and true linear inter-thread and no intra-thread pattern. Given these pattern classifications, our memory selection algorithm selects texture memory for the first array and global memory for the rest of the arrays. The algorithm then searches to find good candidates for thread mapping and work group size, as shown in Table 4.

The final example is also taken from a real world application called *Speeded Up Robust Features (SURF)* [21]. The SURF algorithm is used to detect features in images for applications like stabilization and panorama stitching. The particular kernel we tested is called *non-*

Table 4. Experimental results for the *radiological path calculation* using an input image size of 256 × 32. The register count is 28. † Speedup over default global memory space. Cost and Gain are all zeros. Acronyms and units are the same as in Table 1

#	WGS	AWG	OC	OR	MET	SU†
1	2×128 (256)	2	50%	3	176.90	1.61
2	4×64 (256)	2	50%	3	176.50	1.61
3	8×32 (256)	2	50%	3	174.17	1.60
4	16×16 (256)	2	50%	3	164.92	1.62
5	2×64 (128)	4	50%	2	169.85	1.58
6	4×32 (128)	4	50%	2	170.16	1.58
7	8×16 (128)	4	50%	2	164.75	1.63
8	2×32 (64)	8	50%	1	166.09	1.59
9	4×16 (64)	8	50%	1	163.01	1.63
10	2×16 (32)	8	25%	4	162.35	1.63

max suppression (NMS), which plays a critical role in the SURF algorithm. The kernel looks for the largest value in a stack of images which have been subjected to convolutions with different filters. The kernel involves four arrays: three two-dimensional read-only arrays which possess the same memory access pattern, and one two-dimensional write-only array. We map this loop to a two-dimensional thread configuration and again only show the better of the two possible mapping choices. The memory access patterns are as follows.

$$\begin{bmatrix} 0 & 0 & Z \\ 0 & 0 & Z \end{bmatrix} \begin{bmatrix} tx \\ ty \\ i3 \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} tx \\ ty \\ i3 \end{bmatrix}$$

The patterns are classified as no inter-thread and random intra-thread pattern, and true linear inter-thread and no intra-thread pattern, respectively. Texture memory is selected for the first three arrays and global memory is selected for the last array. The results of the work group size search are summarized in Table 5.

VII. LIMITATIONS OF OUR APPROACH AND FUTURE WORK

Our analysis models and optimization techniques are best employed when considering memory-intensive kernels (i.e., where the ratio of ALU to memory operations is low). In general, when arithmetic intensity is high, the GPU hardware can perform efficient memory

Table 5. Experimental result of *NMS* algorithm when input image size is 256 × 256. The register count is 15. † Speedup over default global memory space. Cost and Gain are all zeros. Acronyms and units are the same as in Table 1

#	WGS	AWG	OC	OR	MET	SU†
1	2×256 (512)	2	100%	3	16.8	5.11
2	4×128 (512)	2	100%	3	16.8	5.12
3	8×64 (512)	2	100%	3	16.8	5.12
4	16×32 (512)	2	100%	3	16.8	5.13
5	32×16 (512)	2	100%	3	16.9	5.12
6	2×128 (256)	4	100%	2	16.8	5.13
7	4×64 (256)	4	100%	2	16.8	5.12
8	8×32 (256)	4	100%	2	16.8	5.12
9	16×16 (256)	4	100%	2	16.9	5.11
10	2×64 (128)	8	100%	1	16.8	5.11
11	4×32 (128)	8	100%	1	16.8	5.12
12	8×16 (128)	8	100%	1	16.9	5.13
13	2×32 (64)	8	50%	4	16.9	5.12
14	4×16 (64)	8	50%	4	16.9	5.13
15	2×16 (32)	8	25%	5	17.1	5.08

latency hiding. If hardware can hide most memory latencies, then memory optimizations will have a limited performance impact. Although this is a limitation, in reality the majority of GPGPU kernels are memory-bound.

Similarly, since our cost metrics do not assign weights to different arrays based on their potential contribution to overall memory contention, our approach may not find the best possible result as the number of memory access patterns and associated arrays in a kernel increase. For most kernels, however, our methodology has been shown to work very well, and our simple, highly automated algorithms accurately represent the complex interplay between GPU hardware and software.

Presently, our framework only handles standard uses of the memory spaces. Programmers frequently utilize memory spaces differently. A good example is local memory. Instead of using local memory for data prefetching, it can be used to avoid uncoalesced accesses. Texture memory can be effectively used to exploit hardware-based interpolation and filtering present in selected application domains.

We made several assumptions and simplifications in this work as well. Our thread mapping assumes that the problem size is a multiple of a half warp. Real-world applications are unlikely to always meet this requirement, so techniques such as data padding are required to apply

our methodology. We also assume that the register size per thread of a kernel is fixed and therefore it always has priority to determine the work group size over local memory. However, the programmer can perform explicit actions to reduce the number of registers if a certain number of active threads are desired. The programmer can also give priority to local memory usage to increase performance. Another assumption made is that the data we are working with is naturally-aligned (e.g., *int* or *float*). Though this assumption is true in most common cases, our methodology would require some modifications to work for unaligned data.

Finally, our analysis model is device dependent. Should the characteristics of memory spaces change, our analysis needs to be updated accordingly. For example, latest commercial GPUs have an on-chip cache for global memory, so the sensitivity of performance to uncoalesced memory accesses should be reduced substantially. In this case our cost metrics need updating.

VIII. CONCLUSION

Current programming languages such as OpenCL and CUDA have made it easier for programmers to accelerate their code on GPU devices, though to reap the full performance benefits offered by these devices is still heavily dependent on a programmer's ability to fine tune their code to the underlying hardware—this is where the steep learning curve of GPU programming actually exists. Tools or frameworks that guide the programmer or automatically select optimizations can have a huge impact on the wider adoption of GPUs.

Memory continues to be a key bottleneck in many GPU applications. Selecting and applying the right set of memory optimizations is a daunting programming task. We have focused our work on characterizing memory access patterns, and developed tools and algorithms to address this important challenge. In this paper we have shown how thread mapping and work group size impacts memory access patterns and thus performance. We have presented our model for memory access pattern analysis that also considers how to select an appropriate thread mapping and work group size.

Although our methodology is used to assist a programmer in developing highly optimized code, the idea can also be used to develop an automatic source-to-source parallelizing compiler, or can be implemented as an optimization pass during compilation.

REFERENCES

[1] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," in Proceedings of the IEEE, vol. 96, 2008, pp. 879–899.

[2] J. Vetter, "Toward exascale computational science with heterogeneous processing," in GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. New York, NY, USA: ACM, 2010, pp. 1–1.

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

[4] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens, "Efficient computation of sum-products on GPUs through softwaremanaged cache," in ICS '08: Proceedings of the 22nd annual international conference on Supercomputing. New York, NY, USA: ACM, 2008, pp. 309–318.

[5] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," in HWWS '04: Proceedings of the ACM SIGGRAPH/ EUROGRAPHICS conference on Graphics hardware. New York, NY, USA: ACM, 2004, pp. 133–137.

[6] C. Jiang and M. Snir, "Automatic tuning matrix multiplication performance on graphics hardware," in Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on, Sept. 2005, pp. 185–194.

[7] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories," in PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. New York, NY, USA: ACM, 2008, pp. 1–10.

[8] M. M. Baskaran, U. Bondhugula, S.

Krishnamoorthy, et. al., "A compiler framework for optimization of affine loop nests for GPGPUs," in ICS '08: Proceedings of the 22nd annual international conference on Supercomputing. New York, NY, USA: ACM, 2008, pp. 225–234.

[9] NVIDIA, "NVIDIA CUDA C Programming Guide 4.2." [Online]. Available: {http://www.nvidia.com/cuda/}

[10] Khronos Group, "OpenCL 1.0 Specification," Dec. 2008. [Online]. Available: {http://www.khronos.org/opencl/}

[11] NVIDIA, "OpenCL Programming Guide for the CUDA Architecture," May 2010. [Online]. Available:{http://developer.nvidia.com/object/cuda_3_1_ downloads .html}

[12] AMD, "OpenCL Programming Guide," Jun 2013. [Online]. Available: {http://developer.amd.com/}

[13] NVIDIA, "GPU Computing SDK Code Samples 4.2." [Online]. Available: {www.nvidia.com/object/cuda develop.html}

[14] 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. New York, NY, USA: ACM, 1997, pp. 317–324.

[15] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation. New York, NY, USA: ACM, 1991, pp. 30–44.

[16] M. E. Wolf, M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," IEEE Trans. Parallel Distrib. Syst., vol. 2, no. 4, pp. 452– 471, 1991.

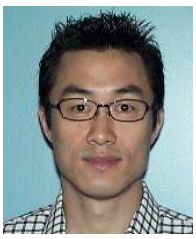
[17] AMD, "Stream Computing Forum." [Online]. Available: {http: //forums.amd.com/devforum/}

[18] NVIDIA, "CUDA Forum." [Online]. Available: {http://forums.nvidia. com/}

[19] B. Jang, D. Kaeli, S. Do, and H. Pien, "Multi GPU implementation of iterative tomographic reconstruction algorithms," in ISBI'09: Proceedings of the Sixth IEEE international conference on Symposium on Biomedical Imaging. Piscataway, NJ, USA: IEEE Press, 2009, pp. 185–188.

[20] M. Christiaens, B. De Sutter, K. De Bosschere, J. Van Campenhout, and I. Lemahieu, "A fast, cache-aware algorithm for the calculation of radiological paths exploiting subword parallelism," Journal of Systems Architecture, vol. 45, no. 10, pp. 781–790, 4 1999.

[21] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded Up Robust Features," in Computer Vision ECCV 2006, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, vol. 3951, pp. 404–417.



Byunghyun Jang received a BS in Bio-Mechatronic Engineering from Sungkyunkwan University, South Korea, a MS degrees in Computer Science from Oklahoma State University, Stillwater OK, and a PhD in Computer Engineering from Northeastern University, Boston MA. He is an Assistant Processor of the Computer and Information Science Department at the University of Mississippi, University, MS where he directs the heterogeneous systems research laboratory (HEROES). Prior to joining Ole Miss in 2012, he spent several years at AMD and Samsung. His research focuses on GPU computing, CPU-GPU heterogeneous computing, hardware architecture and compilers for data parallel architectures, and automatic parallelization.

Minsu Choi received the B.S., M.S., and Ph.D. degrees from Oklahoma State University, Stillwater, in 1995, 1998, and 2002, respectively, all in computer science. He is currently an Associate Professor with the Department of Electrical and Computer Engineering, Missouri University of Science and Technology (formerly U of Missouri-Rolla), Rolla. His research mainly focuses on computer architecture and VLSI, nanoelectronics, embedded systems, fault tolerance, testing, quality assurance, reliability modeling and analysis, configurable computing, parallel and distributed systems, and dependable instrumentation and measurement. Dr. Choi is a member of the Golden Key National Honor Society.

Kyung Ki Kim received the B.S. and M.S. degrees in electronic engineering from Yeungnam University, Kyeongsan, South Korea, in 1995 and 1997, respectively, and the Ph.D. degree in computer engineering from Northeastern University, Boston, MA, in 2008. In 2008, he was a member of the technical staff with Sun Microsystems, Santa Clara, CA, where he was involved in ROCK project. In 2009, he was a senior researcher with Illinois Institute of Technology, Chicago, IL. Currently, he is an assistant professor at Daegu University, South Korea. His current research focuses on nanoscale CMOS design, high speed low power VLSI design, analog VLSI circuit design, electronic CAD and nano-electronics.