Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures

Byunghyun Jang, *Student Member*, *IEEE*, Dana Schaa, *Student Member*, *IEEE*, Perhaad Mistry, *Student Member*, *IEEE*, and David Kaeli, *Fellow*, *IEEE*

Abstract—The introduction of General-Purpose computation on GPUs (GPGPUs) has changed the landscape for the future of parallel computing. At the core of this phenomenon are massively multithreaded, data-parallel architectures possessing impressive acceleration ratings, offering low-cost supercomputing together with attractive power budgets. Even given the numerous benefits provided by GPGPUs, there remain a number of barriers that delay wider adoption of these architectures. One major issue is the heterogeneous and distributed nature of the memory subsystem commonly found on data-parallel architectures. Application acceleration is highly dependent on being able to utilize the memory subsystem effectively so that all execution units remain busy. In this paper, we present techniques for enhancing the memory efficiency of applications on data-parallel architectures, based on the analysis and characterization of memory access patterns in loop bodies; we target vectorization via data transformation to benefit vector-based architectures (e.g., AMD GPUs) and algorithmic memory selection for scalar-based architectures (e.g., NVIDIA GPUs). We demonstrate the effectiveness of our proposed methods with kernels from a wide range of benchmark suites. For the benchmark kernels studied, we achieve consistent and significant performance improvements (up to $11.4\times$ and $13.5\times$ over baseline GPU implementations on each platform, respectively) by applying our proposed methodology.

Index Terms—General-purpose computation on GPUs (GPGPUs), GPU computing, memory optimization, memory access pattern, vectorization, memory selection, memory coalescing, data parallelism, data-parallel architectures.

1 Introduction

Driven by the demands for 3D graphics rendering, GPUs have evolved as highly parallel, many-core data-parallel architectures (see [1] for a detailed history). The increased programmability of GPU hardware, along with the support for general-purpose programming languages, has led to the adoption of GPUs as the platform of choice for data-parallel, computationally intensive workloads—ushering in a new era of desktop supercomputing. The benefits of GPGPUs over other high-performance parallel platforms are numerous: GPUs provide impressive GFLOPs per dollar and GFLOPs per watt ratios. GPUs are also available on every desktop computer, which also enables us to accelerate commonly used desktop applications [1], [2], [3], [4].

The underlying hardware architectures of CPUs and GPUs are fundamentally different, mainly due to the differences present in the target class of applications. CPUs are optimally designed to achieve high performance on sequential code, resulting in more transistors dedicated to control logic and caches. In contrast, GPUs are optimally designed for parallel, high-throughput computing, resulting in more transistors dedicated to computational logic rather than control logic and caches. Moreover, massively data-parallel architectures (e.g., GPUs) available today are implemented in a scalable array of Single Instruction Multiple Data (SIMD) blocks, and this trend should continue as these architectures have been shown to be highly scalable cost-effective solutions for large-scale high-performance computing [5], [6], [7]. However, given that these architectures were originally designed for graphics rendering, the resulting distributed and heterogeneous nature of their memory architectures imposes their underlying programming model, and has slowed the pace of adoption of GPUs for some key general-purpose applications.

The most significant difference between CPUs and GPUs resides in the memory subsystem. GPUs typically feature a wide memory bus to feed a large number of parallel threads; their memory subsystem can be characterized as high bandwidth, high latency, and nonadaptive. These characteristics make the performance of an application running on data-parallel architectures extremely sensitive to irregularity in memory access patterns. At any time, hundreds or thousands of threads may try to issue reads or writes, and these accesses are serialized if the threads generate access patterns that are not optimized for the memory organization. Research has shown that memory performance can be the key limiting factor in terms of overall GPGPU performance [8], [9], [10], [11], [12], [13], [14]. *Vectorization* and *memory coalescing* are two commonly used software techniques to avoid thread stalls due to serialized memory access on those platforms.

From a software perspective, loop and array parallelization is a natural consequence of the emergence of massively multithreaded data-parallel computing platforms, where the same loop body is executed on different data points in

• *The authors are with Northeastern University, 360 Huntington Ave., Boston, MA 02115. E-mail: {bjang, dschaa, pmistry, kaeli}@ece.neu.edu.*

Manuscript received 1 Oct. 2009; revised 21 Feb. 2010; accepted 26 Mar. 2010; published online 19 May 2010.
Recommended for acceptance by D.A. Bader.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDSSI-2009-10-0483.
Digital Object Identifier no. 10.1109/TPDS.2010.107.

an array. Today's GPU computing platforms are designed with heterogeneous memory architectures comprising multiple *memory spaces*, each space possessing specific characteristics (e.g., read-only, single-cycle access). Mapping data arrays to the most appropriate memory space, based on their associated memory access patterns, can have a huge impact on overall performance.

In this paper, we present a methodology that optimizes memory performance on data-parallel architectures. We present two different techniques that utilize mathematical models to systematically characterize array-based memory access patterns present in loop nests. The techniques are: 1) vectorization via data transformation for vector-based architectures (e.g., AMD GPUs) and 2) algorithmic memory selection for scalar-based architectures (e.g., NVIDIA GPUs). We demonstrate the effectiveness of our proposed methodology on both platforms using a set of benchmark kernels that covers wide range of memory access patterns.

The contributions of our work are as follows:

- We identify and analyze the characteristics of the memory subsystems on data-parallel architectures, and discuss issues specific to the architectures developed by the two major GPU vendors, AMD and NVIDIA.
- We present a mathematical model that captures and characterizes memory access patterns inside nested loops. We then use this model to improve the performance of the GPU memory subsystem.
- We present data transformation techniques for a number of common memory access patterns that enable loop vectorization to be used on AMD GPUs.
- We present an algorithmic memory selection methodology that can significantly improve memory utilization and bandwidth on NVIDIA GPUs, based on our model that incorporates memory access patterns and thread mapping.
- We provide experimental results that demonstrate the effectiveness of our models and algorithms on a diverse set of benchmarks.

2 GPU Memory Subsystem

GPU computing platforms adopt a coprocessing model, where the data-parallel computationally intensive code is offloaded onto the GPU; these kernels are executed asynchronously with respect to the execution on the CPU. For asynchronous and independent execution, all input data need to be explicitly copied to the GPU prior to kernel execution, and results need to be copied back to the CPU after execution.

GPU memory subsystems are designed to deliver high bandwidth versus low-latency access. High bandwidth is obtained through the use of a wide memory bus and specialized DDR memories that operate most efficiently when the memory access granularity is large. The highest possible throughput is achieved when a large number of small memory accesses are buffered, reordered, and coalesced into a small number of large requests.

Most modern GPU memory subsystems are composed of several memory spaces (related to their legacy as rendering devices). The memory spaces include both off-chip and on-chip memories. Off-chip memory is consisted of constant and generic memory. Generic memory is called texture memory when it is bound to the texture fetch unit (a special hardware component). On-chip memory includes some cache, and an optional scratchpad memory that can be used as a software-managed cache for local data sharing or reuse. On-chip memory is local in the sense that it belongs to a single multiprocessor block (i.e., to a Streaming Multiprocessor on an NVIDIA GPU or an SIMD Engine on an AMD GPU) and is inaccessible by other multiprocessor blocks on the device. Fig. 1 shows the various memory spaces present on modern GPUs.

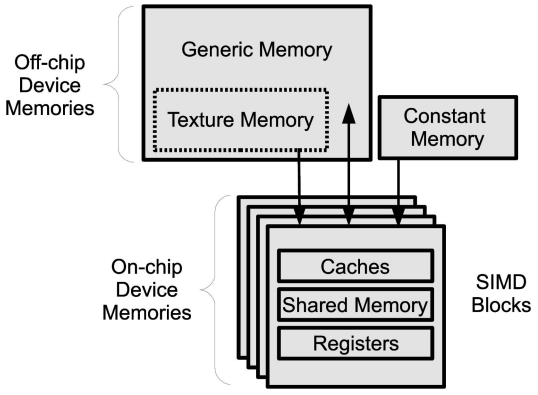


Fig. 1. A look at the GPU memory subsystem. The detailed hardware implementation varies by vendor. The visibility and accessibility of each memory space varies across different GPU programming models.

In the following sections, we summarize the hardware architecture and memory subsystems specific to AMD and NVIDIA GPUs, and discuss the programmability of memory spaces that are exposed to developers in each GPGPU programming model.

2.1 AMD GPUs

AMD GPUs employ a true SIMD vector architecture, where both computation and memory operations are performed on up to four data elements at a time [15], [16]. Brook+, AMD's high-level programming language, provides built-in short vector types that allow code to be explicitly tuned for the architecture. Short vector types include `float`, `double`, or `int` types of 2-4 elements long, with the size appended as a suffix (e.g., `float4`, `int2`, etc.).

Vectorization is a key memory optimization available on AMD platforms [9], [15], [16] that can be exploited to maximize the utilization of the available memory bandwidth. *Memory lanes* are based on 4-element vectors, so bandwidth may be wasted if this vector type is not used. Even though the compiler will attempt to map multiple scalar memory operations into a vector format, using larger vector types explicitly in high-level code will more directly aid the compiler to maximize the utilization of the available execution units and memory bandwidth. Furthermore, since each iteration of a loop is mapped to a thread, vectorization also decreases the total number of threads to be processed, resulting in a smaller number of threads to carry out the same amount of work. As such, vectorization has a profound effect on the performance of AMD GPUs.

AMD's Brook+ programming model abstracts away much of the underlying memory subsystem, making the GPU memory spaces transparent to the programmer.

TABLE 1
GPU Memory Spaces on an NVIDIA Platform

Memory	Location	Cached	Access	Scope
Global	Off Chip	No	R/W	Thread Grid
Constant	Off Chip	Yes	R	Thread Grid
Texture	Off Chip	Yes	R	Thread Grid
Local	Off Chip	No	R/W	Thread
Shared	On Chip	N/A	R/W	Thread Block
Register	On Chip	N/A	R/W	Thread

Memory space mapping is primarily determined by the compiler, and thus, any optimization of memory accesses on an AMD GPU platform should focus on the effective use of vectorization.

2.2 NVIDIA GPUs

NVIDIA GPUs employ a scalar architecture in the sense that computation and memory operations are not necessarily executed in a vector fashion [17], [18]. This threaded execution model is called *Single Instruction Multiple Threads (SIMTs)*, and is particularly useful for parallelizing programs that are not vectorizable. Despite differences in NVIDIA's underlying hardware architecture and execution model, the same memory utilization challenges exist on this architecture. For NVIDIA GPUs, memory coalescing (i.e., simultaneous memory accesses by threads in a single memory transaction) is a key optimization to apply in order to achieve high memory bandwidth [17], [18].

Unlike the AMD programming model, NVIDIA's programming model, CUDA, provides the programmer with full access to each of the different memory spaces. This flexibility comes at a price—the programmer needs to properly place data in these memories based on knowledge of the distinct characteristics of each memory space and memory access patterns of data. Table 1 identifies the different types of GPU-based memory spaces available on current NVIDIA GPUs.

Global memory is the CUDA term for generic GPU memory. It is the default space where input data to the GPU are copied, and where the result of any GPU kernel must be saved to be copied back to the CPU.

Constant memory is a limited off-chip memory space, where a small number of constants (or nonmodifiable input data) can be placed and used by all threads. *Texture memory* is a subset of global memory that is accessed via the texture fetch unit. This region of global memory must be bound through runtime functions before it can be used by a kernel. The salient feature of texture memory comes from its dimensionality that specifies how data are stored and accessed—in one, two, or three dimensions. Both constant and texture memory are cached.

Shared memory is a on-chip scratchpad memory that resides in each multiprocessor. Placing data in shared memory can improve the performance significantly, especially when the data exhibit locality. This is one of the hardware resources that determines the number of active threads on the GPU.

Registers are an on-chip resource that is equally divided across active threads in each multiprocessor. Each multiprocessor has a fixed number of registers and excessive use of registers (i.e., high register pressure) in a kernel limits the number of threads that can run simultaneously, which, in turn, exposes memory latency. The compiler plays a critical role in register usage [19]. *Local memory* is a region of generic memory that is used to store data that do not fit in the registers, and is used to hold *spilled* registers to reduce the impact of register pressure. Both local memory and registers are visible to only individual threads and cannot be programmed directly by the programmer. Therefore, these spaces will not be considered in our discussion of memory selection in the following sections.

```
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];
```

Fig. 2. Serial code example of matrix multiplication.

3 Memory Access Pattern Modeling

In this section, we present a mathematical model that captures the memory access pattern present in a loop nest. This model is an extension of prior work that targets locality improvements on a single core [20]. This model is, in turn, used for our memory optimizations presented in the next section.

Consider a loop nest of depth D that accesses an M-dimensional array. The memory access pattern of the array in the loop is represented as a *memory access vector*, $\vec{m}$, which is a column vector of size M starting from the index of the first dimension. The memory access vector is decomposed into 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 point in an array. Note that we only consider loops whose accesses to arrays are affine functions of loop indices and symbolic variables. We have found that this restriction does not limit us since most scientific applications involve loops possessing affine access patterns [21].

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.

We use the matrix multiplication code example provided in Fig. 2 to describe our model. In the example code, there are three arrays A, B, and C, and their memory access patterns are captured in Fig. 3.

Assuming that arrays are stored in row-major order, the physically contiguous row elements of the array A are accessed linearly as inner loop $i3$ iterates, and the column elements of array A (physically separated by a nonunit stride) are accessed by the outer loop $i1$. Similar representations are provided for arrays B and C.

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

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

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

Fig. 3. The memory access patterns for arrays A, B, and C shown in Fig. 2 as captured by our model.

3.1 Classification of Memory Access Pattern

Having presented a proper mathematical model that captures memory access patterns, we show how to classify memory access patterns to guide the optimization process in this section. This classification, in turn, will be used to select our proposed optimization techniques in later sections. Fig. 4 presents a diagram of various memory access matrices, along with illustrations of the associated memory access patterns.

Linear and *reverse linear* memory access patterns refer to accesses in which a dimension of an array is contiguously accessed with respect to the iteration space (Figs. 4a and 4b, respectively). In the memory access matrix, accesses in the same direction as the loop iteration space are represented by "1," and accesses in the opposite direction are represented by "-1." The offset vector is a zero vector in both cases. Most level 1 and level 2 Basic Linear Algebra Subprograms (BLASs) [22] possess one of these two memory access patterns.

A *shifted* memory access pattern refers to an access in which a dimension of an array is contiguously accessed with respect to the iteration space (as in the linear pattern), but this contiguous access is shifted by some constant. The memory access matrix for this pattern is the same as the linear pattern, but now the offset vector is a nonzero vector. This is shown in Fig. 4c. Shifted patterns are commonly found in multimedia applications, where multiple data streams need to be merged or transformed.

An *overlapping* memory access pattern (Fig. 4d) refers to an access in which a dimension of the array is accessed by more than one iteration space. This pattern is represented as more than one nonzero value in a row in the memory access pattern. This pattern is common in linear equation solver applications that employ LU decomposition where triangular matrices need to be traversed.

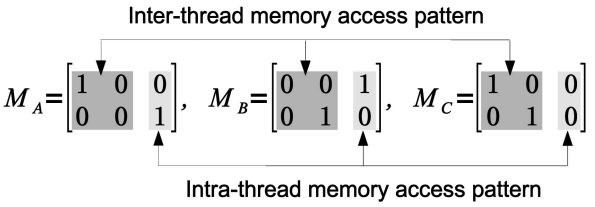


Fig. 5. Memory accesses after thread mapping.

A *nonunit stride* memory access pattern refers to a pattern in which a dimension of an array is accessed with a nonunit stride (shown in Fig. 4e). In the memory access matrix, this pattern is represented by a nonunit constant element. Matrix multiplication typically includes some nonunit stride accesses.

A *random* memory access pattern refers to a pattern in which a dimension of an array is accessed in a random fashion. Memory accesses that are indexed by another array usually fall into this category.

Note that these patterns can be combined, resulting in compound patterns. For example, the pattern that combines reverse linear and shifted accesses is represented as $M = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$, $\vec{o} = \begin{bmatrix} 0 \\ C \end{bmatrix}$ in our model.

3.2 Thread Mapping

In GPGPU programming models (or any data-parallel programming model), one or more of the loops in the loop nest of the serial code are mapped to threads when parallelized.

Suppose that we decide to map the two outer loops ($i1$ and $i2$) in Fig. 2 to a 2D thread configuration (i.e., each iteration of the outer two loops maps to a thread). This means that the first two columns in the memory access matrix represent the memory access pattern among the threads (referred to as *interthread memory access pattern*) and the third column represents the memory access pattern within each thread (referred to as *intrathread memory access pattern*). Fig. 5 illustrates this memory access pattern partitioning and it is of particular importance to memory selection when working on an NVIDIA platform.

Fig. 6 shows a graphical representation of the memory access patterns after mapping threads to the arrays in the matrix multiplication example. The first two columns in the memory access matrix M_A, $\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$, indicate which threads

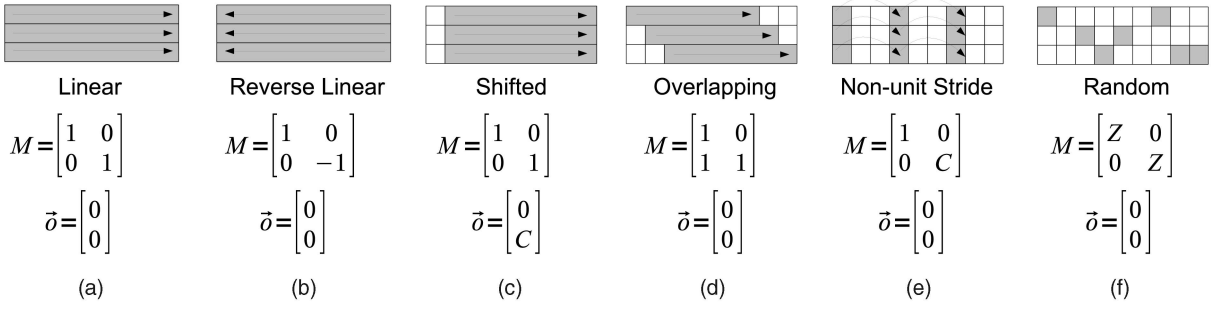


Fig. 4. Classification of memory access patterns and our mathematical representation. Note that the element being accessed is shown in gray, C is a constant, and Z is a random number.

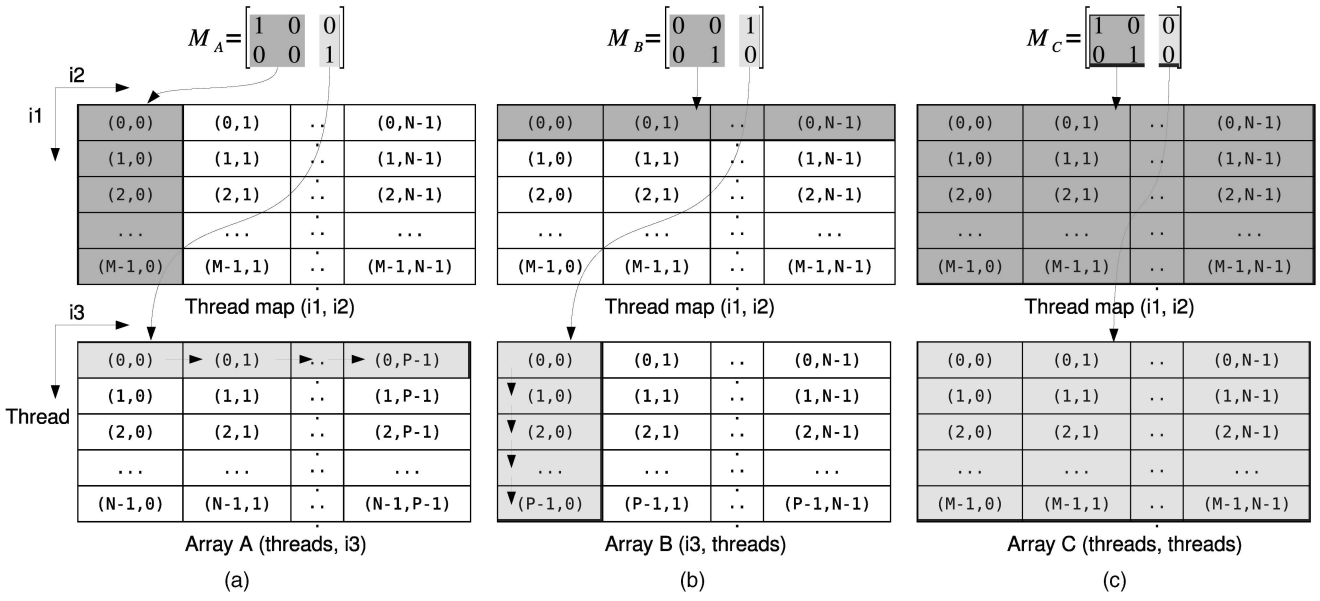


Fig. 6. Graphical interpretation of memory access patterns after thread mapping for arrays in matrix multiplication.

access array A. The third column, $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$, indicates how array A is accessed within the kernel by the threads specified by the first two columns. Therefore, in this example, threads ranging from (0, K) to (M-1, K), where $0 \leq \mathrm{K} < \mathrm{N}$, access elements of array A ranging from (L, 0) to (L, P-1), where $0 \leq \mathrm{L} < \mathrm{N}$, in the same linear pattern. Similarly, threads ranging from (Q, 0) to (Q, N-1), where $0 \leq \mathrm{Q} < \mathrm{M}$, access elements in array B ranging from (0, R) to (P-1, R), where $0 \leq \mathrm{R} < \mathrm{N}$, in the same nonunit stride pattern.

Array C exhibits a slightly different access pattern. The first two columns in its memory access matrix, $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, indicate all threads, and the third column, $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$, indicates that there is only a single access in each thread. Taken together, the access matrix captures that each thread accesses a single element in a corresponding location.

4 GPU Memory Optimizations

Having presented the mathematical modeling and characterization of memory access patterns in serial loop nests in the previous sections, we next present two different architecture-specific optimizations. First, we present a vectorization technique that applies data transformations to benefit the vector-based AMD platform. Second, we present an algorithmic memory selection algorithm to optimize scalar-based architectures such as the NVIDIA platform that feature different types of memory spaces visible to the programmer. These two optimization techniques are very effective at enhancing memory efficiency on their respective architectures [15], [17].

4.1 Vectorization via Data Transformation for AMD GPUs

AMD GPUs employ a vector architecture, where memory operations and computations are performed on a vector granularity. The full computational power of these parallel chipsets can only be achieved when software utilizes the entire vector length via loop vectorization. However, loop vectorization is significantly hindered by irregular memory access patterns present in arrays. In this section, we present data transformations that allow us to vectorize loops that are not vectorizable by the compiler alone. The data transformation rules required to guide loop vectorization are generated using an algorithm, which takes as input a classification of the target data access patterns. We begin by describing our vectorization test and a new metric for quantifying the degree of vectorization possible.

4.1.1 Vectorization Test

The loop vectorization task can vary significantly depending on the details of the underlying hardware and programming language, which will have implications on the analysis used and the transformations possible. Unlike typical short SIMD machines where vector mapping occurs when loading/storing between memory and vector registers, vector mapping on the AMD platform takes place when copying data from the CPU to the GPU. In order to vectorize a loop on an AMD platform, the data being accessed must be located contiguously in GPU memory. Therefore, our primary focus is on the last dimension of the array access (i.e., the row dimension in a row-major memory layout).

In our mathematical model, vectorizing multiple arrays in a loop nest simultaneously requires that each array possess the same memory access pattern with respect to its last dimension. This condition can be checked for by inspecting the last row of the memory access matrices for the selected arrays.

Consider the memory access matrices $\mathbf{M_A}$ and $\mathbf{M_B}$ for the two arrays A and B taken from the matrix multiplication shown in Fig. 3. The last rows of A and B, $[0, 0, 1]$ and $[0, 1, 0]$, respectively, differ, indicating that the row dimension of array A is accessed by $i3$ (index of innermost loop) and the row dimension of array B is accessed by $i2$ (index of middle loop). These two access patterns differ, and thus, cannot be vectorized in a single vectorized computation. We define multiple arrays as *vectorizable* if the last rows in their memory access matrices and their offset vectors are the same.

4.1.2 Quantifying Vectorization

Since it frequently occurs that not all statements in a loop nest can be vectorized, traditional vectorizing compilers categorize vectorization into *completely* and *partially* vectorizable loops [23]. Completely vectorizable loops refer to those in which every statement is vectorizable, while partially vectorizable loops refer to those in which some, but not all, statements are vectorizable.

In our case, we need a somewhat different metric for quantifying vectorization as our vectorization methods are different. We define loops as *intrinsically vectorizable* if all arrays can be directly mapped to vector types without any changes to the loop body. We also define *fractionally vectorizable* if some, but not all, arrays can be mapped to vector types. In the fractionally vectorizable case, we use the equation below to further quantify the degree of vectorization:

$$Q_V(\%) = \frac{\#\ of\ Vectorizable\ Array\ Instances}{Total\ \#\ of\ Array\ Instances} \times 100.$$

Note that the Q_V value of an intrinsically vectorizable loop is 100 percent.

4.1.3 Generating Data Transformation Rules

Given our ability to test for and quantify vectorization, we present a method to generate data transformation rules for nonvectorizable loops that pass our vectorization test. The data transformation rule for each array is composed of a transformation matrix $\mathbf{T_m}$ of size $D \times D$ and a shift vector $\vec{t}_v$ of size D, where D is the size of the iteration vector, and is denoted by the pair $(\mathbf{T_m}, \vec{t}_v)$.

The application of $(\mathbf{T_m}, \vec{t}_v)$ entails two steps. First, the actual data (i.e., the array layout) are transformed on the CPU using our data transformation routine that takes as input the original data and transformation rules, and produces restructured data. Second, a new memory access vector, $\vec{m}'$, is generated as follows:

$$\vec{m}' = \mathbf{M}'\vec{i} + \vec{o}' = \mathbf{T_m}\mathbf{M}\vec{i} + (\mathbf{T_m}\vec{o} + \vec{t}_v).$$

It is common that the same array will be referenced multiple times with different memory access patterns within a loop body; we handle these instances individually if their $\vec{m}$ vectors differ. Next, we will present the required data transformations for each class of memory access pattern.

4.1.4 Linear and Reverse Linear Access Patterns

The data transformation required for the linear pattern (including the reverse linear pattern) is to reverse the order of the data. Consider the loop body shown below. The memory access matrices for arrays X and Y are $\left[\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right]$ and $\left[\begin{smallmatrix}1 & 0\\ 0 & -1\end{smallmatrix}\right]$. Both dimensions of array X and the first dimension of array Y are linear patterns, while the second dimension of array Y is a reverse linear pattern. If we choose to transform Y to match the pattern in X, the data transformation matrix, $\mathbf{T^Y_m}$, would be $\left[\begin{smallmatrix}1 & 0\\ 0 & -1\end{smallmatrix}\right]$ and $\vec{t}_v^{\,Y}$ would be a zero vector.

```
for(i1=0; i1<=N; i1++)
  for(i2=0; i2<=M; i2++)
    X[i1][i2] = Y[i1][M-i2] + 1;
```

4.1.5 Shifted Access Pattern

A shifted access pattern can cause alignment problems during vectorization. The data transformation required to correct for a shifted access pattern shifts all array elements by a constant offset. This type of transformation can be done by simply defining an additional address offset, and typically introduces very little overhead. As a result, $\vec{t}_v$ will be a nonzero vector, whereas T_m will be the same as in the linear pattern case. A code snippet taken from a Livermore Loops kernel (*Hydro Fragment*) that exhibits this pattern is shown below, along with its data transformation rules. Note that there are two instances of array Z, each of which has a different shift vector. Therefore, we generate individual data transformation rules for each case. This results in generating two (versus a single) new arrays, which comes with added storage overhead and possibly increased off-chip memory access:

```
for(i=0; i<N; i++)
  X[i] = q + Y[i]*(r*Z[i+10]+t*Z[i+11]);
```

$$\mathbf{T_m^{Z_1,Z_2}} = [1],\ \vec{t}_v^{\,Z_1} = [-10],\ \vec{t}_v^{\,Z_2} = [-11].$$

4.1.6 Overlapping Access Pattern

An overlapping access pattern can occur when multiple loop indices are involved in the computation of a single array dimension. Consider the following sample code.

```
for(i=0; i<N; i++)
   for(j=0; j<M; j++)
      X[i][j] = Y[i][i+j] + 1;
```

The memory access matrix of array Y, $\mathbf{M_Y}$, is $\left[\begin{smallmatrix}1 & 0\\ 1 & 1\end{smallmatrix}\right]$. Since multiple elements in the last row are nonzero ($[1, 1]$ in this case), this indicates that multiple loop indices are involved in the access of the last dimension of the array. In other words, the last dimension of array Y is accessed jointly by two loop levels (indexed by i and j). In this example, the elements of array Y accessed by inner loop j are shifted by one as outer loop i iterates. The data transformation matrix, $\mathbf{T^Y_m}$, is $\left[\begin{smallmatrix}1 & 0\\ -1 & 1\end{smallmatrix}\right]$ and shift vector, $\vec{t}^{\,Y}$, is a zero vector.

4.1.7 Nonunit Stride Access Pattern

The data transformation needed for linearizing a nonunit stride pattern is to gather elements being accessed into a big chunk of linear addressable memory. The overhead associated with this data transformation may be relatively large since it requires fine-grained restructuring. A loop possessing this pattern is shown below.

```
for(i=0; i<N; i++)
   for(j=0; j<M; j++)
      X[i][j] = Y[i][2*j] + 1;
```

The memory access matrix of array Y, $\mathbf{M_Y}$, is $\left[\begin{smallmatrix}1 & 0\\ 0 & 2\end{smallmatrix}\right]$ and the data transformation matrix, $\mathbf{T^Y_m}$, becomes $\left[\begin{smallmatrix}1 & 0\\ 0 & \frac{1}{2}\end{smallmatrix}\right]$.

4.1.8 Random and Compound Access Patterns

Linearization of a random memory access pattern remains an open problem for loop vectorization—random patterns make

loop vectorization complicated, hindering wider adoption of vectorization for general-purpose computation. Even though there are custom techniques for enabling vectorization of semirandom patterns which possess some degree of regularity [5], it is generally impossible to vectorize completely random patterns. For these reasons, we do not attempt to transform loops possessing random patterns in this work.

The data transformation required for compound access patterns (arrays that exhibit more than one pattern) is straightforward. We first decompose the pattern into one of the simpler patterns above, and then apply secondary transformations for each simple pattern. The order in which we apply the different data transformations does not matter and the generation of the data transformation matrix is straightforward as long as the memory access matrix is a nonsingular matrix.

4.1.9 Vectorization Algorithms

Algorithm 1 illustrates how vectorizability is tested for and how Q_V is computed. The algorithm takes as input the memory access matrices of all array instances in the targeted loop nest and outputs the largest vectorizable group of array instances and Q_V. The algorithm scans all input memory access matrices and compares each row to find the largest group of array instances that are vectorizable together. Two rows are vectorizable (as in line 7 in Algorithm 1) if they have a relationship described by one of the patterns described in the previous section. Note that the offset vector is not considered in the algorithm because it does not impact vectorizability (any shifted pattern can be easily vectorized by shifting data). Q_V is simply computed at the end of algorithm using the number of elements present in the largest vectorizable group.

Algorithm 1: Vectorization test and Q_V computation.

```
   input : M for all instances of arrays in a loop nest
           (SM)
   output: a vectorizable group (VG) and Q_V
 1 VG ← ∅; G ← ∅;
 2 for each M ∈ SM do
 3 |   for each row r⃗ ∈ M do
 4 |   |   Put M into G;
 5 |   |   for each N ∈ SM do
 6 |   |   |   for each row l⃗ ∈ N do
 7 |   |   |   |   if l⃗ is vectorizable with r⃗ then
 8 |   |   |   |   |   G ← G + N;   break;
 9 |   |   |   |   end
10 |   |   |   end
11 |   |   end
12 |   |   if (size of G)>(size of VG) then
13 |   |   |   VG ← G;
14 |   |   end
15 |   |   G ← ∅;
16 |   end
17 end
18 Q_V ← (size of VG) / (size of SM) × 100;
```

Algorithm 2 presents the process of generating data transformation rules. The input is the group of vectorizable array instances discovered by Algorithm 1, and the output is a set of data transformation rules for each array. The goal of the algorithm is to find the position of a vectorizable row in the memory access matrix and the relationship between the vectorizable row and the given target row that we are trying to vectorize. The algorithm takes each memory access matrix and offset vector in an input vectorizable group one at a time and compares each row of $\mathbf{M}$ and $\vec{o}$ with a given target row. Memory access pattern relationships are identified by the sign (e.g., negative for a reverse linear pattern), the magnitude (e.g., a nonunit stride pattern), the number of nonzero components (e.g., an overlapping pattern) of a given row in $\mathbf{M}$, and the value of $\vec{o}$ (e.g., present in a shifted pattern). The array permutation transformation rule (e.g., switching dimensions in multidimensional arrays) can be used when the vectorizable row is not the last row (line 8). Note that the process for determining the best data transformation for compound memory access patterns is handled by identifying each pattern and adding it to the set.

Algorithm 2: Generating data transformation rules.

```
   input : all M and o⃗ pairs for vectorizable arrays
           (VG) and a target row (c⃗)
   output: a set of data transformation rules (T and t⃗
           pair) (DT)
 1 DT ← ∅;   T ← null;   t⃗ ← 0⃗;
 2 for each (M, o⃗) pair ∈ VG do
 3 |   for each row (r⃗, o) ∈ M do
 4 |   |   if r⃗ ≡ c⃗ then
 5 |   |   |   if (r⃗ is last row of M) and (o ≡ 0) then
 6 |   |   |   |   T ← (no transformation);
 7 |   |   |   end
 8 |   |   |   else if o ≡ 0 then
 9 |   |   |   |   T ← (permutation, row number);
10 |   |   |   end
11 |   |   |   if o ≠ 0 then
12 |   |   |   |   T ← T + (shifted);  t⃗ ← −o⃗;
13 |   |   |   end
14 |   |   end
15 |   |   else
16 |   |   |   if sign is different then
17 |   |   |   |   T ← (reversing);
18 |   |   |   end
19 |   |   |   if magnitude is different then
20 |   |   |   |   T ← T + (non-unit stride, magnitude);
21 |   |   |   end
22 |   |   |   if (number of non-zero components in r⃗) >
   |   |   |   (number of non-zero components in c⃗) then
23 |   |   |   |   T ← T + (overlap, non-zero
   |   |   |   |   components);
24 |   |   |   end
25 |   |   |   if o ≠ 0 then
26 |   |   |   |   T ← T + (shifted); t⃗ ← −o⃗;
27 |   |   |   end
28 |   |   end
29 |   |   if (T ≡ null) and (t⃗ ≡ 0⃗) then  continue;
30 |   |   else DT ← DT + (T, t⃗);
31 |   end
32 end
```

4.2 Memory Selection and Memory Access Coalescing for NVIDIA GPUs

On the NVIDIA GPU platform, multiple memory spaces are exposed to the programmer. This is not the case in the AMD Brook+ environment. Proper memory selection can have a

large impact on performance, but remains a challenging optimization problem [10], [17], [24]. Because the characteristics and the best usage of each memory space are distinct, we can select the most appropriate space for each array based on its associated memory access pattern.

The following sections summarize the characteristics and the best usage of each memory space on the NVIDIA platform and present our algorithm that selects the best memory space based on the model presented in Section 3. In the following sections, we use NVIDIA's terminology for memory spaces for clarity.

4.2.1 Global Memory and Coalescing

Global memory is an off-chip memory space that is not cached. Even though its usage is the most flexible (it can be used for any part of the program), its performance varies substantially depending on the associated memory access pattern. The use of this memory should be restricted to fully coalesced access cases. Serialized access to global memory is very expensive given the high cost of off-chip memory access latency (the latency to global memory is two orders of magnitude greater than accesses to on-chip memory). We define *memory coalescing* as related to our model as follows[1]:

- a linear, reverse linear interthread memory access pattern (e.g., M_B and M_C in Fig. 6), or shifted interthread memory access pattern with the shift amount that is a multiple of a half warp[2] size [17].

Note that detailed requirements for coalesced access are different across devices of different compute capabilities (see [17] for detailed information) and our definition is conservative (a sufficient condition rather than a necessary one) to satisfy the requirement of all devices. For example, an overlapped pattern can be coalesced when the shift amount in later iterations becomes a multiple of the half warp size.

We also define *temporal reuse* in our model as follows:

- multiple appearances of the array in the source code, or
- a nonzero intrathread memory access pattern, implying multiple accesses occurring inside the kernel (e.g., a loop inside the kernel).

Using the above definitions, we develop the following rules for the best use of global memory:

- for output data, or
- for read-only or read-write input data that are fully coalesced and have no temporal reuse.

4.2.2 Constant Memory

Constant memory is a small off-chip space (64 KB on the current generation of NVIDIA GPUs) that is cached and read-only. Its scope is global, in the sense that it is visible to all threads. The latency of constant memory is similar to reading from a register when all threads of a half warp read the same address from the cached data. Reads to different addresses by threads within a half warp are serialized, resulting in a cost that increases linearly with the number of different addresses read by all threads within a half warp [17]. We define a *same address read* in our model as follows:

- no interthread memory access patterns (all zero elements) and only intrathread memory access pattern (at least one nonzero element), implying that memory accesses to this array are globally identical across all threads.

Note that this is also a sufficient condition in that only threads of a half warp are required to read the same memory address in order for access to remain nonserialized. This detailed investigation depends on the thread configuration (e.g., thread block[3] size) and is beyond the scope of this paper.

The following summarize the cases in which constant memory should be selected:

- read-only data that are small enough to fit in the available constant memory space, and
- accesses that satisfy the "same address read" requirement.

Consider the following simple loop nest and associated memory access patterns for arrays X and Y, which show the case where constant memory is best utilized.

```
for (i=0; i<M; i++) {
  float tmp = 0;
  for (j=0; j<N; j++) {
    tmp += Y[j];
  }
  X[i] = tmp;
}
```

$$\vec{m}_X = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix}$$

$$\vec{m}_Y = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix}$$

Given that the outer loop is mapped to threads (the thread vector is $[i]$), the first column of each memory access matrix represents the interthread memory access pattern for each array and the second column represents the intrathread memory access pattern. Therefore, each element of array X is accessed by a corresponding thread once (denoted by 1 in an interthread and 0 in an intrathread memory access pattern). Alternatively, each element of array Y is accessed by all threads with the same linear pattern (denoted by 0 in the interthread and 1 in the intrathread memory access pattern cases). Therefore, array Y meets the requirements for using constant memory. Fig. 7 illustrates the performance impact of placing array Y in constant memory. As shown,

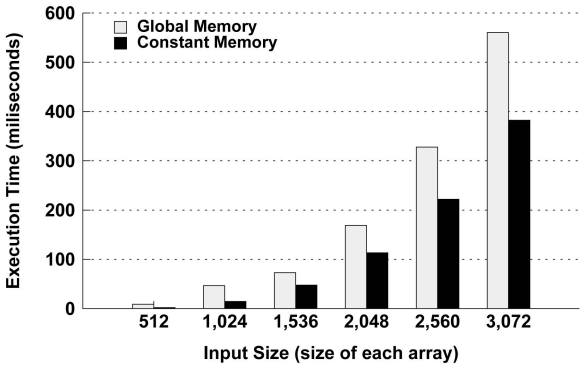


Fig. 7. Performance impact of placing array Y in constant memory in the example shown above.

1. We assume that the element size of data being considered is 32 bits (type of float) throughout the paper.

2. NVIDIA defines a half warp as the group of threads that are scheduled together (16 threads on the current generation GPU).

3. Thread block is an NVIDIA term that refers to a group of threads assigned to a multiprocessor that can be synchronized together.

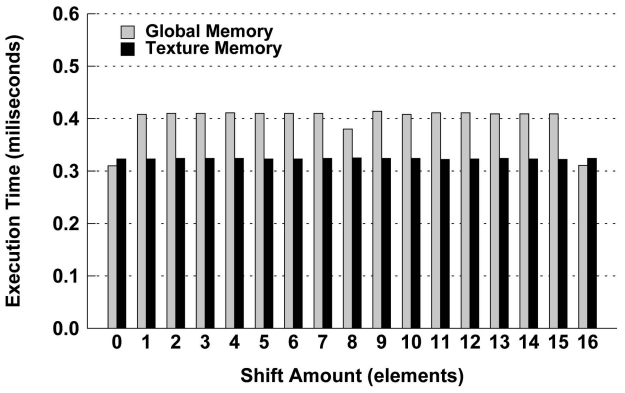


Fig. 8. Performance impact of placing array Y in texture memory in the example shown above at array size 2,048.

the performance gain associated with using constant memory is substantial (approximately $1.5\times$ speedup across all array sizes).

4.2.3 Texture Memory

Similar to constant memory, texture memory is an off-chip space that is cached and read-only. But its size and usage differ from those of the constant memory. First, texture memory can be much larger in size than constant memory. Theoretically, the entire global memory space can be bound to the texture unit. Second, the texture cache is optimized for 2D spatial locality, meaning that the best performance is achieved when all threads of a half warp access the addresses that are close to each other in 2D [17], [18], [24].

These characteristics of texture memory guide us to use it in the following cases:

- read-only data that are uncoalesced (see the definition of coalesced accesses in our model in Section 4.2.1), including random access patterns, or
- read-only data that do not meet the requirements of global, constant, and shared memory.

Consider the following simple loop extracted from [24] and the memory access vector for array Y. The loop simply copies the data from input array Y to array X, offset by a `shift` amount; array Y exhibits a shifted memory access pattern and is uncoalesced when the `shift` is not a multiple of the size of a half warp.

```
for (i=0; i<M; i++) {
  X[i] = Y[i + shift];
}
```

$$\vec{m}_Y = [1][i] + [shift]$$

Fig. 8 shows the performance impact when a memory access is not coalesced. Placing data in texture memory improves memory performance by approximately 30 percent over placing data in global memory. Uncoalesced access to global memory serializes memory access, introducing a significant memory bottleneck. Note that if the memory accesses are fully coalesced, global memory slightly outperforms texture memory (as shown when the shift amount is "0" or "16" in Fig. 8).

4.2.4 Shared Memory

Shared memory is a small on-chip software-managed memory on the NVIDIA platform. Shared memory differs from global, constant, or texture memory in that it is local to the multiprocessor and visible to only active threads running on that multiprocessor. The latency of a shared memory access (assuming no bank conflicts) is similar to a register access (two orders of magnitude faster than global memory [17]). Effective utilization of this low-latency on-chip shared memory can have a large impact on performance.

Data that possess high temporal locality should be placed into shared memory [11], [17], [24]. Shared memory can be used as a local cache for data placed in other data spaces on the device. Given the limited shared memory space per multiprocessor, the amount of shared memory used determines the number of active threads that can run simultaneously (i.e., the hardware occupancy). Since finding the best usage for shared memory is directly related to thread block size (which usually requires programmer's intervention), this trade-off is not considered further in this work. We refer to the task of sizing shared memory requests as *chunking* and consider whether data are *chunkable*, which denotes the case where the data are divisible into multiple chunks, which satisfies the optimal thread configuration and size limitations.

The following summarize the characteristics of the data that are suitable for shared memory:

- read-write data that are temporally reused (see the definition of temporal reuse in our model in Section 4.2.1) and chunkable, or
- read-only data that are temporally reused and satisfy all constant memory requirements except the size, and are chunkable.

4.2.5 Memory Selection Algorithm

Having described the characteristics of each memory space, as well as specific memory access patterns present in data-parallel systems, we present an algorithm to identify the best memory space to place data based on its access characteristics. The algorithm takes as input the memory access matrix and offset vector pairs of all instances of an array, the size, read-write, and temporal reuse information, and the thread index vector. It then scans the input data to select the most appropriate memory space for the array. The following criteria are used for selection:

1. read-only versus read-write,
2. size of data,
3. memory access classification,
4. interthread memory access pattern,
5. intrathread memory access pattern, and
6. temporal reuse information.

Pseudocode for this algorithm is shown in Algorithm 3. Note that the term *Small* in line 4 refers to the case where the data are smaller than the available constant memory.

After running the algorithm, multiple memory spaces may be selected for a single array depending on the number of instances of the array. In this case, we make a final selection using the following ordered secondary criteria. For read-only data:

1. texture,
2. global,
3. shared,
4. constant,

TABLE 2
Loop Characteristics for the Workloads Studied in This Section

Benchmark	Candidate Loops	Trivially Vectorizable	Intrinsically Vectorizable after Data Transformation	Non Vectorizable			
				Total	Loop-carried Dep.	Random	Etc.
Linpack	7	5	0	2	0	2	0
Livermore Loops	24	3	8	13	7	5	1
NAS Kernel	32	13	12	7	2	0	5

and for read-write data, 1) global and 2) shared. For example, if a single array is accessed twice in a loop body, and shared memory is selected for its first access (i.e., the array reference exhibits the chunkable characteristic) and texture memory is selected for its second access (i.e., the array reference exhibits random access), then we map the array to texture memory because it is not chunkable and so should not be placed in shared memory.

Algorithm 3: Memory selection

```
input : M and o⃗ pairs of all instances of an array
        (SMA), the size, read-write, and temporal
        reuse information of the array, and a thread
        index vector (t⃗)
output: memory space (MS)
MS ← 0 ;
if read-only then
    for each M ∈ SMA do
        if Small && Accessing same address then
            MS ← MS + Constant;
        end
        else if Chunkable && Temporal reuse then
            MS ← MS + Shared;
        end
        else if Coalesced && No temporal reuse then
            MS ← MS + Global;
        end
        else MS ← MS + Texture;
    end
end
else
    for each M ∈ SMA do
        if Chunkable && Temporal reuse then
            MS ← MS + Shared;
        end
        else MS ← MS + Global;
    end
end
```

5 Experimental Results

We have evaluated the effectiveness of our transformations on both AMD and NVIDIA GPUs using a number of different benchmark kernels. All benchmarks are executed on an Intel Core 2 Duo at 2.66 GHz running 64-bit Linux, with 2 GB main memory and with a single AMD or NVIDIA GPU. In the following sections, we present experimental results of the architecture-specific memory optimizations applied on each platform.

5.1 Vectorization via Data Transformation on AMD GPUs

We evaluate the effectiveness of our loop vectorization strategy using AMD's Stream Computing SDK 1.4 and the Catalyst 9.2 graphics driver (which incorporates the graphics compiler) on an AMD Radeon HD3870 (R670 family) GPU. For all benchmarks selected, two GPU versions (scalar and vector) are implemented using the Brook+ programming language.

To illustrate the power of our optimizations, we have selected three high-performance computing benchmark suites: Linpack, Livermore Loops, and NAS kernels. These suites implement scientific kernels that are prime candidates for acceleration on data-parallel architectures. Table 2 provides some important characteristics for these three benchmarks suites. Within these benchmarks are a number of loops that initialize arrays and wrap functions for test purposes (in the NAS kernels and Linpack). These are considered to be nonkernel loops and are excluded from the number of total loops reported. The remainder of the loops are then divided into categories that identify the number of loops that are trivially vectorizable, intrinsically vectorizable (only after applying data transformations), and nonvectorizable (such as those with loop-carried dependencies). The statistics show that a significant percentage of loops in high-performance applications are vectorizable after applying data transformations. The last three columns in Table 2 list the reasons why loops could not be vectorized.

We focus our evaluation on five loops: the first one is a commonly used GPGPU benchmark, matrix multiply, and the remaining four loops are taken from the three benchmark suites mentioned above. Our selections are based on the following criteria:

- The loops are initially nonvectorizable but vectorizable after data transformation.
- The loops possess a range of memory access behaviors and loop structures that are well suited to demonstrate the effectiveness of our proposed methods.

Each benchmark is run with two different input sizes. For each size, the benchmark is run five times and the arithmetic mean of the execution time is plotted in Fig. 9. Table 3 also summarizes the detailed results of the benchmark experiments.

The matrix multiply implementation used in this work is shown in Fig. 2. Our algorithm informs us that there are three possible choices for vectorization ({A,B},{A,C},{B,C}) via a dimension switch (2D matrix transpose), and since Q_V is the same in all three cases, we arbitrarily choose {A,B} for our implementation. Fig. 9a shows a performance comparison

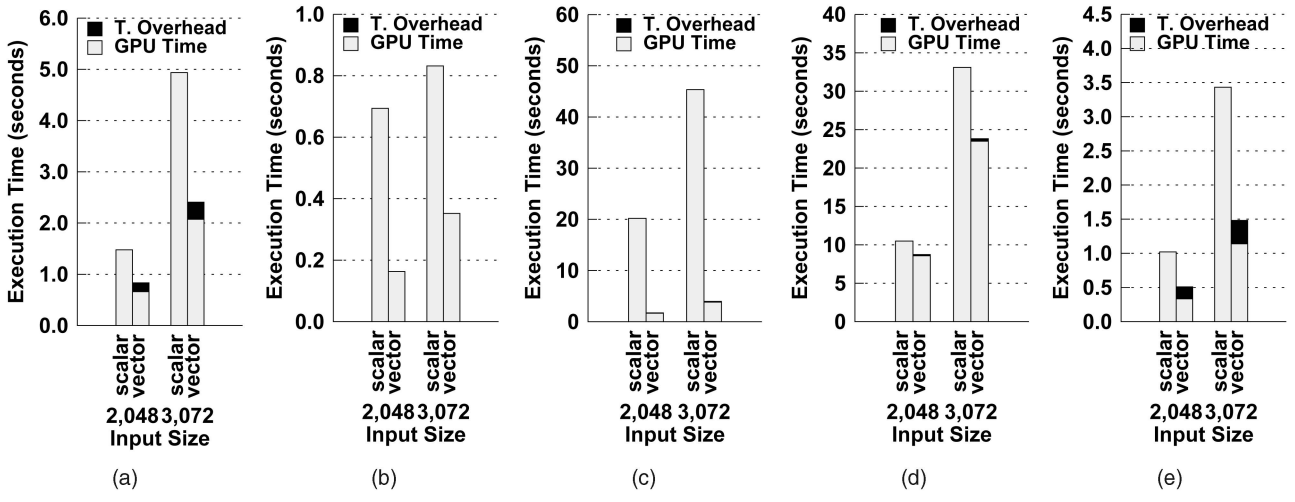


Fig. 9. Performance comparison between scalar and vector implementations on AMD platform. Note that the data transfer time between the CPU and GPU is included in the GPU time. (a) Matrix multiply, (b) Livermore 1, (c) Livermore 19, (d) NAS Cholesky, and (e) NAS Vpenta.

between the scalar and vector implementations. It is clear that the vectorized kernel outperforms its scalar counterpart and that the overhead associated with the data transformation is relatively small. Additional information is summarized in Table 3.

We selected two of the Livermore loops: kernel 1 (*Hydro Fragment*) and kernel 19 *General Linear Recurrence Equations*. Kernel 1 is fairly small in size and there are three arrays involved. The first two arrays are accessed with the same linear pattern and the last array is accessed twice using a shifted address pattern with different offsets. Our algorithm states us to duplicate the third array and shift each element by a constant offset. The resulting transformed arrays enable us to vectorize the loop. The effect on performance is shown in Fig. 9b. Note that the overhead associated with this data transformation (i.e., shifting in this case) is almost zero due to simply defining a new offset using a pointer construct in C/C++, though there is storage overhead associated with data duplication.

Kernel 19 operates on 4 one-dimensional arrays. Two of these arrays possess linear and reverse linear access patterns, and the other two have shifted and reverse shifted patterns. Each array is transformed after duplication by our algorithm. The effect on performance is plotted in Fig. 9c. The vectorization effect on performance is substantial (11.4× speedup over the scalar version at an input size of 3,072).

We also studied two loops from the NAS benchmark suite: one is the *Cholesky decomposition/substitution subroutine* and the other is from *Vpenta*. The Cholesky loop accesses 2 three-dimensional arrays in the loop nest, where one array has a linear memory access pattern and the other has a nonunit stride pattern. Our algorithm recognizes that the two arrays can be vectorized if the dimensions of one of the arrays are switched. The second loop chosen from the NAS benchmark exhibits the same memory access pattern and data transformation rules, though the array dimension is different. The performance effects of these two kernels are plotted in Figs. 9d and 9e, respectively.

5.2 Memory Selection for NVIDIA GPUs

Next, we evaluated the effectiveness of our memory selection methodology using NVIDIA's CUDA SDK 2.3 and graphic driver version 190.18 on an NVIDIA Geforce GTX 285 (G200 family) GPU. For all benchmarks selected, two GPU versions are implemented: one using only the default (global) memory and another using algorithmically selected memories.

The serial loops evaluated here are selected from the *Parboil* benchmark suite [25] and *PhysBAM* [26], [27]. Parboil is a set of benchmarks consisted of real-world applications to measure the performance of general-purpose programs on GPU platforms, and PhysBAM is an object-oriented C++ library capable of solving a variety of problems in computational fluid dynamics, computational mechanics, and computer graphics.

TABLE 3
Summary of Benchmarks for Vectorization on AMD Platform

Benchmark	Memory Access Patterns	Data Transformations	Q_V	Speedup over Scalar*	Data Trans. Overhead
Matrix Multiply	Linear, Non-unit Stride	Dimension Switch (2D)	66.6%	2.05×	16.12%
Livermore 1	Linear, Shifting	Duplicate, Shifting	100%	2.36×	negligible
Livermore 19	Linear, Shifting, Reverse Linear	Duplicate, Shifting, Reversing Column	100%	11.4×	3.39%
NAS Cholesky	Linear, Non-unit Stride	Dimension Switch (3D)	100%	1.39×	1.36 %
NAS Vpenta	Linear, Non-unit Stride	Dimension Switch (2D)	100%	2.32×	23.16 %

** The speedup shown includes data transformation overhead and is measured at input size 3,072 for each dimension of the array.*

TABLE 4
Summary of Benchmarks for Memory Selection on NVIDIA Platform

Benchmark	Kernel	Memory Access Patterns and Additional Memory Access Information	Memory Selections	Speedup over Default Space*
MRI-Q	ComputePhiMag	Linear read, Linear write, No temporal reuse	Global	N/A
	ComputeQ	Linear read, Linear write, Same address read	Global, Constant	3.03×
MRI-FHD	ComputeRhoPhi	Linear read, Linear write, Not enough temporal reuse	Global	N/A
	ComputeFH	Linear read, Linear write, Same address read	Global, Constant	5.34×
CP	cpuenergy	Non-unit stride write, Same address read	Global, Constant	1.3×
SAD	sad4_cpu	Linear read, Linear write, Random read, Temporal reuse	Global, Texture, Shared	13.5×
	larger_sads	Random read-write	Texture	1.4×
PhysBAM	ClampParticle	Linear write, Random read, Temporal reuse	Global, Texture, Shared	8.89×

The speedup shown is measured using the largest input size provided by benchmark suites.

We investigated seven kernels from four Parboil benchmark applications and one kernel from the PhysBAM library. For each serial kernel, the memory access models are built using our model and the memory space is selected using Algorithm 3. Table 4 summarizes the benchmark kernels used for the evaluation of our proposed memory selection methodology on the NVIDIA platform. The kernels, *ComputePhiMag* and *ComputeRhoPhi*, are small kernels and all arrays are accessed with linear reads, linear writes, and possess no temporal reuse. These three characteristics guide our algorithm to select global memory space for all of the arrays. The kernel *larger_sads* has a large read-write array whose memory access pattern is random for both reads and writes. Due to this random write access pattern, the only possible selection is global memory. For these three kernels (whose memory selection is only global), a more formal performance comparison is not available because our baseline implementation is global memory.

The memory access patterns in the arrays in the *ComputeQ* and *ComputeFH* kernels are characterized by a linear writes, linear reads, and same address reads in our model. The read-only input array in both kernels is accessed by all threads with the same access pattern, regardless of the thread configuration (denoted by the "same address reads" in our model) and placed in constant memory by our algorithm after checking the size requirement. Figs. 10a and 10b show the performance impact of these selections on each kernel, respectively.

The kernel *cpuenergy* exhibits similar characteristics as found in the two kernels above, except that it has a nonunit stride write. The memory selected for this kernel is the same and the impact of this choice is seen in Fig. 10e. The kernel, *sad4_cpu*, exhibits various kinds of memory access behavior. There are multiple arrays used in this kernel and our model characterizes those accesses as linear reads, linear writes, random reads, and temporal reuse. Our algorithm selects global, texture, and shared memory spaces for those arrays to be placed. Fig. 10d shows that the performance improvement is significant when using all three memory spaces.

The kernel *ClampParticle* from the PhysBAM benchmark is a representative function that frequently arises in Physics simulations. It determines the new position for a subset of particles using a weighted sum of the positions of its neighbors. Using our model, its memory access pattern is characterized as linear writes, random reads, with temporal reuse. Our algorithm selects global, texture, and shared memory spaces for the arrays used in the kernel. Fig. 10e illustrates the performance improvement of using the selected memory spaces. For a large input size, the speedup achieved is $8.89\times$. Table 4 summarizes our experimental results.

5.3 Limitations of Our Approach

Although our experimental results demonstrate that the proposed data transformations can significantly increase the

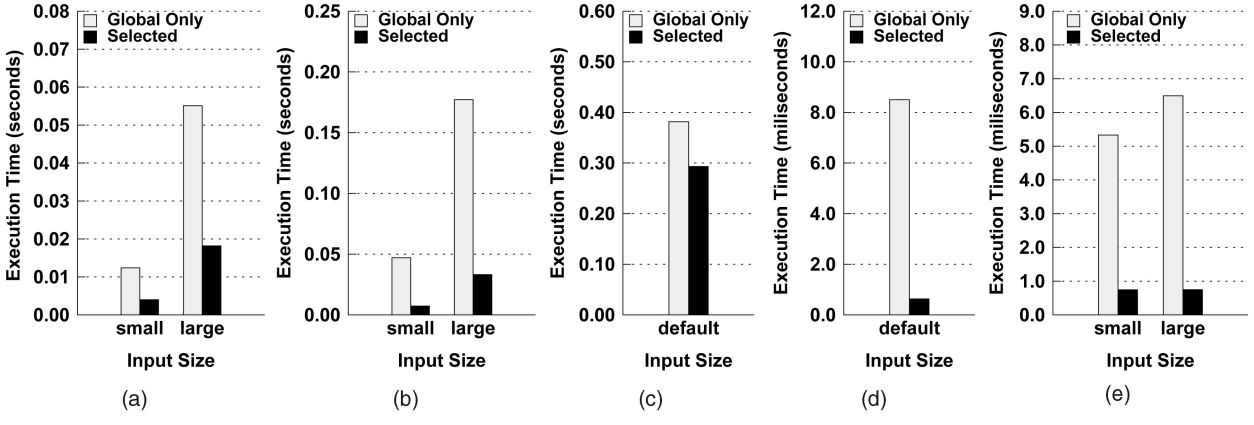


Fig. 10. Performance comparison between using default memory (global only) and using selected memory on the NVIDIA platform. Note that we used the input data size provided by benchmark suites (small, large, and default). (a) ComputeQ, (b) ComputeFH, (c) cpuenergy, (d) sad4_cpu, and (e) ClampParticle.

number of loops that can be vectorized, and that the overhead associated with our data transformations on massively multithreaded vector GPU architectures can be easily amortized, the overall overhead (including data transformation time and storage requirements) can become an issue since the overhead of data duplication is heavily dependent on the particular memory access pattern present. For example, a multipoint stencil code (e.g., 9-point stencil) would require multiple duplications of an array using our approach. Given the overhead of data duplication and the potential impact on cache utilization, data transformations on more complex reference patterns remain an open problem (note that all data are placed in texture memory on the AMD Brook+ platform and utilize the texture cache). An exact analysis of the trade-offs between the benefits of vectorization and the associated transformation overhead remains as future work.

Another issue with our data duplication approach arises when the data array is both read and written in a loop body. Consider a loop nest in which an array is read twice, each time with a different access pattern, and with an intervening write between the two reads. If the data are duplicated based on an analysis of the two different read access patterns, data consistency issues can arise due to the intervening write. Although these cases can be easily detected, special care should be taken when duplicating arrays that involve read/write access.

In the proposed memory selection scheme, we do not provide the details of shared memory usage because finding the best usage of shared memory is directly related to the thread block size. The best shared memory mapping usually requires significant programmer intervention. Given that efficient share memory usage is highly data dependent and may require frequent code changes, (e.g., introducing data prefetching code or implementing loop transformations such as loop tiling), detailing shared memory usage is beyond the scope of this work. However, as a number of previous research studies have shown, effective utilization of shared memory is a key step to achieving high performance on NVIDIA GPUs [8], [11], [14], [24]. We plan to consider these trade-offs in our future work.

6 Conclusion and Future Work

Many general-purpose programs running on data-parallel architectures (such as GPUs) suffer from memory bottlenecks. The hardware design of data-parallel architectures foregoes versatility in the memory subsystem for high computational power, resulting in significant delays when faced with irregular memory access patterns. Unless we handle these problematic memory accesses patterns properly, the performance of general-purpose applications on GPUs can suffer greatly.

In this work, we present an algorithmic approach to enhance the memory efficiency on the two major GPU platforms, AMD and NVIDIA. Our analysis and structured memory optimization techniques are based on a mathematical model that captures loop-based memory access patterns, and can account for the differences in the underlying hardware and programming language constructs of the two platforms.

We introduce data transformations that enable vectorization on AMD GPUs, and an algorithmic memory selection technique for NVIDIA GPUs. The impact of our techniques on performance is demonstrated using a range of benchmarks. Although we show two applications of our memory access pattern model on two different GPU platforms, the techniques are applicable to data-parallel architectures in general.

The experiments in this work are performed in a semiautomated fashion following the steps exactly as shown in the pseudocode provided. These algorithms can be further embedded into a compiler framework as an optimization stage, or implemented as a stand-alone toolset that guides a programmer on how best to utilize the available memory subsystem on the GPU.

Acknowledgments

This research was supported by the US National Science Foundation (NSF) EEC Innovations Program Award number EEC-0946463 and an AMD/ATI Fellowship Award. The authors would like to thank AMD and NVIDIA for their generous hardware donations and Jenny Mankin for her thorough proofreading. The authors would also like to thank the anonymous contributors on the AMD stream computing and CUDA forums for their valuable discussions and clarification of some subjects.

References

[1] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro,* vol. 28, no. 2, pp. 39-55, Mar./Apr. 2008.

[2] D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn, "Gpgpu: General Purpose Computation on Graphics Hardware," *Proc. ACM SIGGRAPH,* p. 33, 2004.

[3] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, "GPU Computing," *Proc. IEEE,* vol. 96, no. 5, pp. 879-899, May 2008.

[4] "GPGPU Website," www.gpgpu.org, 2010.

[5] H. Chang and W. Sung, "Efficient Vectorization of SIMD Programs with Non-Aligned and Irregular Data Access Hardware," *Proc. 2008 Int'l Conf. Compilers, Architectures and Synthesis for Embedded Systems (CASES '08),* pp. 167-176, 2008.

[6] H. Chang and W. Sung, "Access-Pattern-Aware On-Chip Memory Allocation for SIMD Processors," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 28, no. 1, pp. 158-163, Jan. 2009.

[7] C.G. Lee and M.G. Stoodley, "Simple Vector Microprocessors for Multimedia Applications," pp. 25-36, Nov./Dec. 1998.

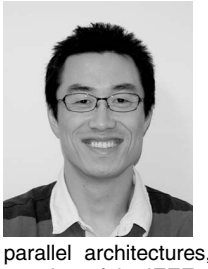
[8] M.M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic Data Movement and Computation Mapping for Multi-Level Parallel Architectures with Explicitly Managed Memories," *Proc. 13th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '08),* pp. 1-10, 2008.

[9] B. Jang, S. Do, H. Pien, and D. Kaeli, "Architecture-Aware Optimization Targeting Multithreaded Stream Computing," *Proc. Second Workshop General Purpose Processing on Graphics Processing Units (GPGPU-2),* pp. 62-70, 2009.

[10] B. Jang, D. Kaeli, S. Do, and H. Pien, "Multi GPU Implementation of Iterative Tomographic Reconstruction Algorithms," *Proc. Sixth IEEE Int'l Symp. Biomedical Imaging: From Nano to Macro (ISBI '09),* June 2009.

[11] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J.D. Owens, "Efficient Computation of Sum-Products on GPUs through Software-Managed Cache," *Proc. 22nd Ann. Int'l Conf. Supercomputing (ICS '08),* pp. 309-318, 2008.

[12] B. Jang, P. Mistry, D. Schaa, R. Dominguez, and D. Kaeli, "Data Transformations Enabling Loop Vectorization on Multithreaded Data Parallel Architectures," *Proc. 15th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '10)*, 2010.
[13] N.K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A Memory Model for Scientific Algorithms on Graphics Processors," *Proc. ACM/IEEE Conf. Supercomputing (SC '06)*, p. 89, 2006.
[14] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication," *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. Graphics Hardware (HWWS '04)*, pp. 133-137, 2004.
[15] AMD, "ATI Stream Computing User Guide, V 1.4 Beta, Brook+ SDK," http://developer.amd.com/gpu/ATIStreamSDK/, 2010.
[16] AMD, "Stream Computing Forum," http://forums.amd.com/devforum/, 2010.
[17] NVIDIA, "CUDA Programming Guide 2.3," http://www.nvidia.com/cuda/, July 2009.
[18] NVIDIA, "CUDA Forum," http://forums.nvidia.com/, 2010.
[19] R. Dominguez, D. Kaeli, J. Cavazos, and M. Murphy, "Improving the Open64 Backend for GPUs," *Proc. GPU Technology Conf.*, 2009.
[20] S.T. Leung and J. Zahorjan, "Optimizing Data Locality by Array Restructuring," Technical Report TR 95-09-01, Univ. of Washington, 1995.
[21] S. Ghosh, M. Martonosi, and S. Malik, "Cache Miss Equations: An Analytical Representation of Cache Misses," *Proc. 11th Int'l Conf. Supercomputing (ICS '97)*, pp. 317-324, 1997.
[22] "BLAS (Basic Linear Algebra Subprograms)," http://www.netlib.org/blas/, 2010.
[23] D. Levine, D. Callahan, and J. Dongarra, "A Comparative Study of Automatic Vectorizing Compilers," *Parallel Computing*, vol. 17, nos. 10/11, pp. 1223-1244, 1991.
[24] NVIDIA, "NVIDIA CUDA C Programming Best Practices Guide 2.3," http://www.nvidia.com/cuda/, July 2009.
[25] Impact Research Group, "The Parboil Benchmark Suite," http://impact.crhc.illinois.edu/parboil.php, 2007.
[26] "Physics Based Modeling (PhysBAM)," http://physbam.stanford.edu/, 2010.
[27] J. Teran, E. Sifakis, G. Irving, and R. Fedkiw, "Robust Quasistatic Finite Elements and Flesh Simulation," *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation (SCA '05)*, pp. 181-190, 2005.



Byunghyun Jang received the BS degree in biomechatronic engineering from Sungkyunkwan University, South Korea, and the MS degree in computer science from Oklahoma State University. He is currently working toward the PhD degree in computer engineering at Northeastern University. Prior to joining Northeastern, he was a research engineer at Samsung Electronics, South Korea. His research interests include GPU computing, compiler techniques for data-parallel architectures, and program parallelization. He is a student member of the IEEE.

Dana Schaa received the BS degree in computer engineering from California Polytechnic State University, San Luis Obispo, and the MS degree in computer engineering from Northeastern University, where he is currently working toward the PhD degree. His research interests include parallel computing, multicore and many-core systems (esp., GPUs), and virtualization. He is a student member of the IEEE.

Perhaad Mistry received the BS degree in electronics and communication engineering from the University of Mumbai and the MS degree in computer engineering from Northeastern University. He is currently working toward the PhD degree at Northeastern University, where he is a member of the Northeastern University Computer Architecture Research Laboratory (NUCAR). His research interests include the design of architecture-aware data structures for physics simulation, parallel programming models, and GPU computing. He is a student member of the IEEE.

David Kaeli received the BS and PhD degrees in electrical engineering from Rutgers University and the MS degree in computer engineering from Syracuse University. He is a full processor in the ECE Faculty at Northeastern University, Boston, Massachusetts, where he directs the Northeastern University Computer Architecture Research Laboratory (NUCAR). Prior to joining Northeastern in 1993, he spent 12 years at IBM, the last seven years at T.J. Watson Research Center, Yorktown Heights, New York. He has published more than 200 critically reviewed publications, six books, and eight patents. His research spans a range of areas including microarchitecture to back-end compilers and database systems. He is an associate editor of the *Journal of Instruction-Level Parallelism* and on the Executive Board of both the ACM SIGMicro and the IEEE TCCA. He is also a member of CRA's Computing Consortium Council. He is a fellow of the IEEE.