A Compiler Framework for Optimization of Affine Loop Nests for General Purpose Computations on GPUs

Muthu Manikandan Baskaran[1] Uday Bondhugula[1] Sriram Krishnamoorthy[1]
J. Ramanujam[2] Atanas Rountev[1] P. Sadayappan[1]

[1]Dept. of Computer Science and Engineering
The Ohio State University
2015 Neil Ave. Columbus, OH, USA
{baskaran,bondhugu,krishnsr,rountev,saday}@cse.ohio-state.edu

[2]Dept. of Electrical & Computer Engg. and
Center for Computation & Technology
Louisiana State University
jxr@ece.lsu.edu

Technical Report OSU-CISRC-12/07-TR78
December 2007

Abstract

GPUs are a class of specialized parallel architectures with tremendous computational power. The new Compute Unified Device Architecture (CUDA) programming model from NVIDIA facilitates programming of general purpose applications on NVIDIA GPUs. However, there are various performance-influencing factors specific to GPU architectures that need to be accurately characterized to effectively utilize the parallel computing power of GPUs and improve the performance of applications on GPUs. Often these factors are tightly coupled, making their effective tuning a significant challenge. In addition, program-specific optimizations such as tiling, loop unrolling, etc. involve performance trade-offs on GPUs that are difficult to characterize accurately using performance models.

In this paper, we develop an automatic compiler framework for generating efficient parallel programs on GPUs for given input regular programs. The framework generates program transformations (using the general polyhedral model) that enable efficient execution over GPUs, and employs a model-driven empirical optimization approach to find optimal values for system parameters that maximize performance, as well as the best tile sizes and loop unroll factors. Experimental results show a significant improvement in performance for kernels generated using our framework, and provide new insights into performance optimizations for GPUs.

1. Introduction

Graphics Processing Units (GPUs) today are among the most powerful computational systems on a chip. For example, the NVIDIA GeForce 8800 GTX GPU chip uses over 680 million transistors and has a peak performance of over 350 GFlops [30]. In addition to the primary use of GPUs in accelerating graphics rendering operations, there has been considerable interest in General Purpose computation on GPUs (GPGPU) [13, 20, 19]. Until very recently, GPGPU computations were performed by transforming matrix operations into specialized graphics processing such as texture operations. The introduction of the CUDA (Compute Unified Device Architecture) programming model by NVIDIA in late 2006 provided a general-purpose threaded SIMD/MIMD architectural model for implementation of general-purpose computations on GPUs. Although more convenient than previous graphics programming APIs for developing GPGPU codes, the manual development of high-performance codes with the CUDA model is still much more complicated than use of parallel programming models such as OpenMP for general-purpose multi-core systems. It is therefore of great interest, for enhanced programmer productivity and for software quality, to develop a compiler infrastructure to facilitate the automatic transformation of sequential input programs into efficient parallel CUDA programs.

There has been significant progress over the last two decades on the development of powerful compiler frameworks for dependence analysis and transformation of loop computations with affine bounds and array access functions [2, 35, 27, 21, 14, 36, 33, 4]. For such regular programs, compile-time optimization approaches have been developed using affine scheduling functions with a polyhedral abstraction of programs and data dependences. Although the polyhedral model of dependence abstraction and program transformation is much more powerful than the traditional model of data dependences currently used in production optimizing compilers, until very recently polyhedral approaches were not considered practically efficient enough to handle anything but simple toy codes. But advances in dependence analysis and code generation [36, 4, 41] have solved many of these problems, resulting in the polyhedral techniques being applied to code representative of real applications like the spec2000fp benchmarks. CLooG [4, 10] is a powerful state-of-the-art code generator that captures most of these advances and is widely used. Building on these advances, we have recently developed the PLuTo compiler framework that enables end-to-end automatic parallelization and locality optimization of affine programs for general-purpose multi-core targets [6, 31]. The effectiveness of the transformation system has been demonstrated on a number of non-trivial application kernels on a multi-core processor. However, building such a system for GPUs requires the addressing of a number of additional issues. In this paper, we identify and characterize key factors that affect GPGPU performance and build on the PLuTo system to develop an affine compile-time transformation framework for GPGPU optimization.

The paper is organized as follows. A brief overview of the NVIDIA GeForce 8800 GTX GPU is provided in Section 2. Section 3 develops an empirical characterization of three significant performance issues: efficient global memory access, efficient shared memory access, and reduction of dynamic instruction count by enhancing data reuse in registers. In the next four sections, we discuss a general compiler framework for GPGPUs that systematically addresses these performance-critical issues. Section 8 presents experimental performance results that demonstrate the effectiveness of the proposed framework. Related work is discussed in Section 9. We conclude in Section 10.

2. Overview of the GPU Architecture and the CUDA Programming Model

The GPU parallel computing architecture comprises of a set of multiprocessor units, each one containing a set of processors executing instructions in a SIMD fashion. The NVIDIA GeForce 8800 GTX has 16 multiprocessor units, each consisting of 8 processor cores that execute in a SIMD fashion. The processors within a multiprocessor unit communicate through a fast on-chip *shared memory* space, while the different multiprocessor units communicate through a slower off-chip DRAM, also called *global memory*. Each multiprocessor unit also has a fixed number of *registers*. The GPU code is launched for execution in the GPU device by the CPU (host). The host transfers data to and from GPU's global memory.

Programming GPUs for general-purpose applications is enabled through an easy-to-use C language interface exposed by the NVIDIA Compute Unified Device Architecture (CUDA) technology [29]. The CUDA programming model abstracts the processor space as a grid of thread blocks (that are mapped to multiprocessors in the GPU device), where each thread block is a grid of threads (that are mapped to SIMD units within a multiprocessor). More than one thread block can be mapped to a multiprocessor unit, and more than one thread can be mapped to a SIMD unit in a multiprocessor. Threads within a thread block can efficiently share data through the fast on-chip shared memory and can synchronize their execution to coordinate memory accesses. Each thread in a thread block is uniquely identified by its thread block id and thread id. A grid of thread blocks is executed on the GPU by running one or more thread blocks on each multiprocessor. Threads in a thread block are divided into SIMD groups called *warps* (the size of a warp for the NVIDIA GeForce 8800 GTX is 32 threads) and periodic switching between warps is done to maximize resource utilization.

The shared memory and the register bank in a multiprocessor are dynamically partitioned among the active thread blocks on that multiprocessor. The GeForce 8800 GTX GPU has a 16 KB shared memory space and 8192 registers per multiprocessor. If the shared memory usage per thread block is 8 KB or the register usage is 4096, at most 2 thread blocks can be concurrently active on a multiprocessor. When any of the two thread blocks complete execution, another thread block can become active on the multiprocessor.

The various memories available in GPUs for a programmer are as follows: (1) off-chip global memory (768MB in 8800 GTX), (2) off-chip local memory, (3) on-chip shared memory (16KB per multiprocessor in 8800 GTX), (4) off-chip constant memory with on-chip cache (64KB in 8800 GTX), and (5) off-chip texture memory with on-chip cache.

3. Performance Characterization of GPU

In this section, we characterize key factors that affect GPGPU performance and provide insights into optimizations that a compiler framework must address to enhance performance of computation on GPUs. We use micro-benchmarks that can bring out the effect of the GPU architecture characteristics that are critical for efficient execution. The micro-benchmarks were run on a NVIDIA GeForce 8800 GTX GPU device.

3.1 Global Memory Access

The off-chip DRAM in the GPU device i.e. the global memory has latencies of hundreds of cycles. While optimizing for data locality helps improve the performance of programs with temporal locality, reducing the latency penalty incurred is critical for good performance.

N	Block (GBps)	Cyclic (GBps)
2048	4.11	22.91
4096	4.78	37.98
8192	5.11	48.20
16384	5.34	56.50
32768	6.43	68.51

Table 1. Global memory bandwidth for block and cyclic access patterns

We evaluated the cost of global memory access by measuring the memory read bandwidth achieved for different data sizes, for blocked and cyclic distribution of the computation amongst the threads of a single thread block.

In the micro-benchmark used for bandwidth measurement, a one-dimensional array of size N is accessed from global memory by a thread block with T threads. Each thread accesses N/T elements of the array (N is chosen as a multiple of T). Two different access patterns were compared: (1) blocked access, where thread 0 accesses the first N/T elements, thread 1 accesses the second set of N/T elements ..., and thread $T-1$ accesses the last N/T elements, and (2) cyclic access, where thread 0 accesses element 0, thread 1 accesses element 1, ... thread $T-1$ accesses element $T-1$, and the threads cyclically repeat the same access pattern. The bandwidth achieved is shown in Table 1. Although the threads in both cases accessed the same number of elements from global memory, cyclic access results in significantly higher memory bandwidth – up to 68.5GBps, improvement by a factor of 10, compared to blocked access.

The significant difference in performance of the two versions is due to a hardware optimization called *global memory coalescing* – accesses from adjacent threads in a half-warp to adjacent locations in global memory are coalesced into a single quad-word memory access instruction. Interleaved access to global memory from the threads in a thread block is essential to exploit this architectural feature.

Using cyclic data access by the threads, we measured the effect of the number of threads per thread block on the achieved memory bandwidth. In addition, we evaluated the impact of strided data access on the observed memory performance. The stride of access across threads was varied from 1 through 64, and the number of threads per thread block was varied from 32 through 512. The results from this experiment are shown in Figure 1.

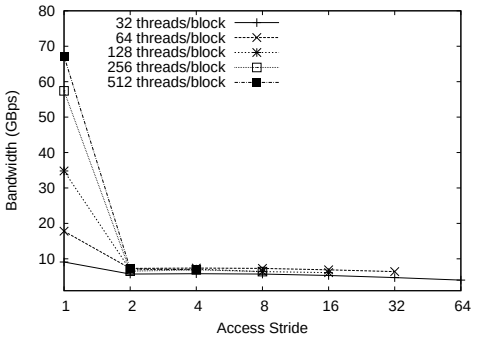


Figure 1. Global memory bandwidth for different access strides for varying number of threads per thread block

We observe that non-unit strides across threads lead to significant degradation in performance. This is because global memory coalescing only happens with unit stride access across threads. With non-unit access strides, all

memory accesses are issued individually to memory, resulting in similar poor performance. With unit access stride, as the number of threads per thread block is increased, we observe an increase in the memory bandwidth achieved, with the maximum bandwidth achieved for 512 threads. This is due to better ability to mask global memory access latencies with increase in the number of warps per multiprocessor.

The significant performance benefits due to coalesced access of memory makes it one of the most important optimizations to be enabled by a compiler framework. Also, the high latency of global memory access recommends the reduction of number of global memory loads/stores.

3.2 Shared Memory Access

The shared memory is a fast on-chip software-managed cache that can be accessed by all threads within a thread block. The shared memory space is divided into equal-sized memory modules called banks, which can be accessed in parallel. In the NVIDIA GeForce 8800 GTX, the shared memory is divided into 16 banks. Successive 32-bit words are assigned to successive banks. Hence, if the shared memory address accessed by a half-warp (i.e. the first 16 threads or the next 16 threads of a warp) map to different banks, there are no conflicting accesses, resulting in 16 times the bandwidth of one bank. However if n threads of a half-warp access the same bank at a time, there is an n-way bank conflict, resulting in n sequential accesses to the shared memory. In our further discussion, we refer to the number of simultaneous requests to a bank as *degree of bank conflicts*. Hence k degree of bank conflicts means a k-way bank conflict and 1 degree of bank conflicts means no bank conflicts (since there is only one request to a bank). The bandwidth of shared memory access is inversely proportional to the degree of bank conflicts.

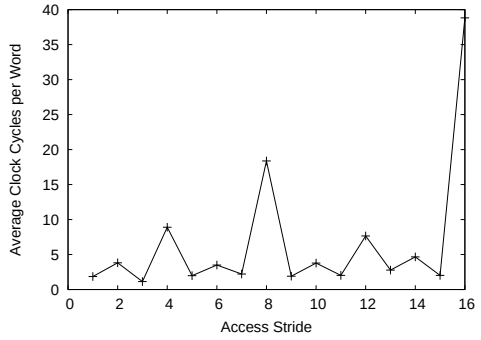


Figure 2. Shared memory access time for different access strides across threads

We conducted an experiment to study the effect of the degree of bank conflicts, by measuring the shared memory access time for different access strides (strides from 1 to 16) across threads. 32 threads per thread block were used for the experiment and each thread accessed 100 data elements. When access stride is 1, successive threads in a thread block access successive words in shared memory, which fall in different banks. Hence there are no bank conflicts between the thread accesses. When access stride is 2, the first thread accesses a word from bank i, the second thread accesses a word from bank $i+2$ and so on. Thus there is a 2-way conflict, i.e., conflict between the accesses of thread 1 and thread 9, thread 2 and thread 10, ..., thread 8 and thread 16. Figure 2 shows the observed behavior. There are no bank conflicts when the access stride is odd and hence the observed access time is fastest for odd access strides. From Figure 2, we clearly observe that shared memory access time depends on the degree of bank conflicts and access time is almost the same for access strides that lead to the

same degree of bank conflicts. This characterization suggests that minimizing shared memory bank conflicts is an important optimization to be handled by a compiler framework.

3.3 Degree of Parallelism vs Register Pressure

One of the important optimizations to be performed in the thread-level computation code is to reduce the number of dynamic instructions in the run-time execution. Loop unrolling is one of the techniques that reduces loop overhead and increases the computation per loop iteration. We studied the benefits of reducing dynamic instructions in a thread using loop unrolling through a micro-benchmark that runs over a simple computation loop of 1K iterations repeatedly adding a scalar to a shared memory data element. Figure 3 shows the effect of various unroll factors for the micro-benchmark code. Also, register-level tiling through unroll-and-jam to reduce number of loads/stores per computation is a well known program optimization when there is sufficient reuse in the data accessed. Though loop unrolling reduces dynamic instructions and register tiling reduces the number of loads/stores, they increase register usage. The number of threads that can be concurrently active in a multiprocessor unit depends on the availability of resources such as shared memory and registers. A thread block of threads can be launched in a multiprocessor unit only when the number of registers required by its threads and the amount of required shared memory are available in the multiprocessor. Clearly, increased register pressure may reduce the active concurrency in the system.

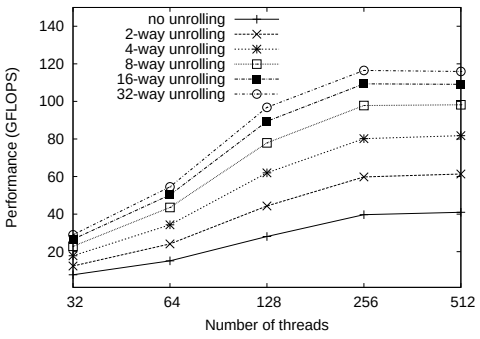


Figure 3. Characterization of the benefits of loop unrolling

For code for which performance is limited by memory access, having more threads can efficiently mask global memory access latency. Figure 1 clearly illustrates the impact of parallelism on the bandwidth of global memory access. Hence a memory-access-bound code requires more threads to efficiently overcome the global memory access latency.

Putting the issues together, a computation that has huge global memory access overhead requires higher concurrency and may also demand more registers to enable the benefits of loop unrolling such as loop overhead reduction and reduction of number of loads/stores. Hence there is a clear trade-off between number of active concurrent threads and number of registers available for a thread in a thread block to exploit the above benefits. Due to such a tight coupling of GPU resources, an empirical evaluation becomes necessary to select an optimal choice of program parameters such as unroll factors, and tile sizes and system parameters such as number of threads and thread blocks.

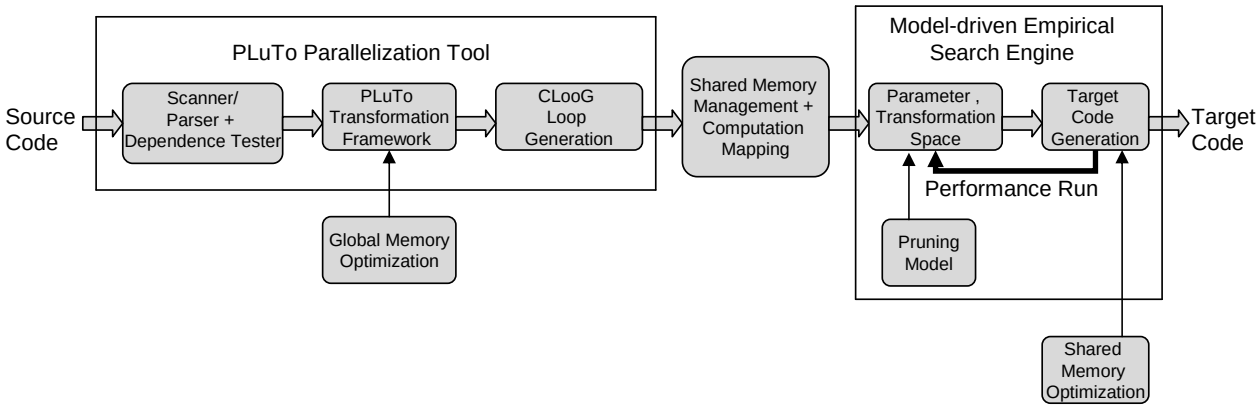


Figure 4. Compiler Framework for GPUs

4. Overview of Compiler Framework

Having identified several performance-influencing characteristics of GPUs, we now discuss the design of an end-to-end compiler framework that targets the generation of efficient parallel programs for GPUs, performing the optimizations required. We build the compiler framework for GPUs on top of PLuTo [31], an effective automatic parallelization system. PLuTo optimizes sequences of imperfectly nested loops, simultaneously for parallelism and locality through tiling transformations. Given an input sequential code, it automatically generates parallel OpenMP code for multi-core processors. PLuTo has an effective automatic transformation framework along with a source code scanner and parser, and dependence tester, and is integrated with the CLooG code generator tool.

Building a compiler framework for GPUs requires attention to several additional issues: (1) utilizing and managing on-chip shared memory, (2) exploiting multiple levels of parallelism (of varying granularities), (3) optimizing global memory access, (4) optimizing shared memory access, (5) optimizing and tuning several system and program parameters, and (6) generation of thread-centric code.

Issues (1) and (2) are addressed in our earlier work [3]. To address issue (3), we have developed a component that is integrated with PLuTo's transformation framework to generate program transformations that enable optimized global memory access. This is discussed in detail in Section 5.2. A module that optimizes shared memory access (issue (4)) is discussed in Section 6. A model-driven empirical search engine (discussed in Section 7 is used for finding optimal or near-optimal system and program parameters that are infeasible to characterize accurately with cost models alone, because of lack of knowledge/control over GPU register allocation by the compiler.

The various components of the compiler framework are depicted in Figure 4. The components can be broadly listed as, (1) the frontend PLuTo parallelization system, including the source code scanner and parser, dependence tester, transformation framework, and skeleton code generator, (2) component for generating transformations enabling global memory access, (3) component for managing on-chip shared memory and mapping computation on multiple parallel units, (4) component that optimizes shared memory access, and (5) model-driven empirical search engine to optimize system and program parameters.

In the following sections, we will describe each of the components addressed in this paper in detail.

5. Optimizing Global Memory Access

The results from Section 3.1 show that significant performance improvements for GPUs can be achieved through program transformations that optimize global memory accesses. This section develops an approach for performing such transformations. The proposed approach is based on the polyhedral model, a powerful algebraic framework for representing programs and transformations [28, 33]. Our focus is on loop-based computations where loop bounds are affine functions of outer loop indices and global parameters (e.g., problem sizes). Similarly, array access functions are also affine functions of loop indices and global parameters. Such code plays a critical role in many computation-intensive programs, and has been the target of a considerable body of compiler research.

mv kernel:

```
for (i=0;i<n;i++) {
  P: x[i]=0;
  for (j=0;j<n;j++) {
    Q: x[i]+=a[i][j]*y[j];
  }
}
```

tmv kernel :

```
for (i=0;i<n;i++) {
  S: x[i]=0;
  for (j=0;j<n;j++) {
    T: x[i]+=a[j][i]*y[j];
  }
}
```

Figure 5. mv and tmv kernels

5.1 Background

A statement S surrounded by m loops is represented by an m-dimensional polytope, referred to as an iteration space polytope. The coordinates of a point in the polytope (called the iteration vector $\vec{x_S}$) correspond to the values of the loop indices of the surrounding loops, starting from the outermost one. Each point of the polytope corresponds to an instance of statement S in program execution. The iteration space polytope is defined by a system of affine inequalities, $\mathcal{D}_S(\vec{x_S}) \geq \vec{0}$, derived from the bounds of the loops surrounding S. Using matrix representation in *homogeneous* form to express systems of affine inequalities, the iteration space polytope can equivalently be represented as

$$D_S . \begin{pmatrix} \vec{x_S} \\ \vec{n} \\ 1 \end{pmatrix} \geq \vec{0},$$

where D_S is a matrix representing loop bound constraints and $\vec{n}$ is a vector of global parameters (e.g., problem sizes).

Consider the Matrix Vector (mv) multiply and Transpose Matrix Vector (tmv) Multiply kernels in Figure 5. The iteration space polytope of statement Q is defined by $\{i, j \mid 0 \leq i \leq n-1 \ \wedge \ 0 \leq j \leq n-1\}$. In matrix representation, this polytope is given by

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & -1 \end{bmatrix} . \begin{pmatrix} \vec{x_Q} \\ n \\ 1 \end{pmatrix} \geq \vec{0}$$

where $\vec{x_Q} = \binom{i}{j}$ is the iteration vector of statement Q.

Affine array access functions are also represented using matrices. If $\mathcal{F}_{kAS}(\vec{x_S})$ represents the access function of the k^{th} reference to an array A in statement S, then

$$\mathcal{F}_{kAS}(\vec{x_S}) = F_{kAS}. \begin{pmatrix} \vec{x_S} \\ \vec{n} \\ 1 \end{pmatrix}$$

where F_{kAS} is a matrix representing an affine mapping from the iteration space of statement S to the data space of array A. Each row in the matrix defines a mapping corresponding to one dimension of the data space. When the rank of the access matrix of an array reference is less than the iteration space dimensionality of the statement in which it is accessed, the array is said to have an order of magnitude (or higher-order) reuse due to the reference. Thus, the condition for higher-order reuse of an array A due to a reference $\mathcal{F}_{kAS}(\vec{x_S})$ is

$$rank(F_{kAS}) < dim(\vec{x_S}) \tag{1}$$

Loops whose iterators do not occur in the affine access function of a reference are said to be *redundant loops* for the reference.

In Figure 5, the access function of the reference to array a in statement T can be represented as

$$\mathcal{F}_{1aT}(\vec{x_T}) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} . \begin{pmatrix} \vec{x_T} \\ n \\ 1 \end{pmatrix}$$

The rank of the access matrix is 2 and the iteration space dimensionality is 2, indicating that the array has no higher-order reuse due to this reference.

Affine transformation of a statement S is defined as an affine mapping that maps an instance of S in the original program to an instance in the transformed program. The affine mapping function of a statement S is given by

$$\phi_S(\vec{x_S}) = \mathcal{C}_S. \begin{pmatrix} \vec{x_S} \\ \vec{n} \\ 1 \end{pmatrix}.$$

When $\mathcal{C}_S$ is a row vector, the affine mapping ϕ_S is a one-dimensional mapping. An m-dimensional mapping can be represented as a combination of m (linearly independent) one-dimensional mappings, in which case $\mathcal{C}_S$ is a matrix with m rows. In further discussion, we use θ_S to denote affine time mapping (to time points) and π_S to denote affine space mapping (to virtual processors).

There has been significant and advanced work on dependence analysis in the polyhedral model [14, 35]. An affine transformation, found to improve performance of programs, is valid only if it preserves the dependences in the original program. Affine transformations using polyhedral model are widely used for improvement of sequential programs (source-to-source transformation) [15, 16] as well as automatic parallelization of programs [27, 21, 17, 11].

Lim et al. [27] use an affine transformation framework for automatic parallelization. They define two different categories of constraints on the transformations: *space partition constraints* and *time partition constraints*. The space partition constraints ensure that any two dependent statement instances are always placed in the same space partition, i.e., they are mapped to the same virtual processor in processor space. The time partition constraints ensure that if a statement instance r depends on a statement instance s, then r is executed at the same time point as s or at a later time point than s. Feautrier [15, 16] defines affine *time schedule*, which is one-dimensional (single sequential loop in the transformed program) or multi-dimensional (nested sequential loops in the program). The schedule associates a timestamp with each statement instance. Statement instances are executed in increasing order of timestamps to preserve data dependences. Two statement instances that have the same timestamp can be executed in parallel. The *time schedule constraint* in Feautrier's framework, needed to preserve a data dependence, is as follows

$$\forall \vec{x_s} \in \mathcal{D}_s, \forall \vec{y_t} \in \mathcal{D}_t \text{ s.t. } \vec{y_t} \text{ depends on } \vec{x_s}, \qquad \theta_t(\vec{y_t}) - \theta_s(\vec{x_s}) > 0 \tag{2}$$

Using such constraints, one can define a system that characterizes the time schedule coefficients, taking into account all dependencies. The system is then solved to find the legal time schedules. There has been a significant body of work (e.g., [15, 27]) on the procedure to solve a system of constraints for affine partition mappings, using the affine form of Farkas' Lemma and Fourier-Motzkin projection algorithm.

5.2 Global Memory Coalescing

In GPUs, execution of a program proceeds by distributing the computations across thread blocks and across threads within a thread block. In a thread block, data required for computation can be either accessed directly from global memory or copied to shared memory and then accessed. We focus on the code executed in a thread block to optimize for global memory access. We use PLuTo-generated tiling hyperplanes to distribute tiles across thread blocks and a tile (with iteration space specified by the shape of the tile) is given as input to the global memory access optimization component of our framework.

We first determine arrays that have either higher-order reuse (using condition defined by (1) or sufficient constant reuse and mark them as arrays that have to be copied from global memory to shared memory for efficient performance. Arrays that have no reuse due to any of its reference are candidates for direct access from global memory. But inefficient access to global memory may degrade the performance as illustrated in Section 3.1. We apply the framework to find program transformations that can lead to efficient global memory access of as many array references (that have insufficient reuse) as possible. If a resulting program transformation does not optimize access of an array reference, then the data accessed by the reference is copied (efficiently) to shared memory.

To enable global memory coalescing for an array reference in a statement, iterations accessing adjacent elements of the array (along the fastest varying dimension) have to be executed simultaneously (in time) by distinct threads that are consecutive in thread (processor) space. In a timepoint, different statement instances in the timepoint are executed collectively by all the threads. Hence iterations accessing adjacent data elements of an array should have the same timestamp to ensure simultaneous access by adjacent threads. This is enforced by the *time schedule adjacency constraint* which enforces two statement instances that access adjacent elements of an array (that is not reused) to be executed at the time instance. The *time schedule adjacency constraint* is

defined (assuming row major storage of arrays) as:

$$\forall \vec{x_s} \in \mathcal{D}_s, \forall \vec{y_s} \in \mathcal{D}_s \text{ s.t. } \mathcal{F}_{rzs}(\vec{x_s}) + (0 \ldots 1)^T = \mathcal{F}_{rzs}(\vec{y_s}), \qquad \theta_s(\vec{x_s}) = \theta_s(\vec{y_s}) \tag{3}$$

In a space partition, each instance is processed by distinct threads. Thus iterations accessing adjacent data elements of an array have to be in adjacent space partitions so that adjacent data elements are accessed by adjacent threads. This is enforced by the *space partition adjacency constraint* which enforces two statement instances that access adjacent elements of an array (that is not reused) to be executed by adjacent processors in the processor space. The *space partition adjacency constraint* is defined as:

$$\forall \vec{x_s} \in \mathcal{D}_s, \forall \vec{y_s} \in \mathcal{D}_s \text{ s.t. } \mathcal{F}_{rzs}(\vec{x_s}) + (0 \ldots 1)^T = \mathcal{F}_{rzs}(\vec{y_s}), \qquad \pi_s(\vec{y_s}) = \pi_s(\vec{x_s}) + 1 \tag{4}$$

The space adjacency constraint also enforce cyclic distribution of virtual processors to physical processors as block distribution may nullify the effect of optimization achieved by the transformation satisfying space adjacency constraint.

We now explain the procedure used to determine transformations, for code executed in a thread block, that enable optimal global memory access. In our approach, we solve for a time schedule (for each statement) that preserves all dependences and satisfies *time schedule adjacency constraint* (Equation 3) for all array accesses (that do not have enough reuse) in the program. If there does not exist a solution, we combinatorially try all subsets of array accesses and generate time schedules that satisfy the *time schedule adjacency constraint* and potentially would generate space partitions that satisfy *space partition adjacency constraint* (Equation 4). Once a t dimensional time schedule is determined for a statement with m loops surrounding it, we find a $m-t$ dimensional space partition mapping such that each of the $m-t$ mappings are linearly independent of each other and the time schedule and one of the space mappings (treated as the innermost space partition) satisfy the *space partition adjacency constraint*. All transformations that have legal time schedules and also have valid space partition mappings enabling coalesced global memory access are considered as candidate transformations for the empirical search engine (discussed in Section 7). If there is no valid 'space-time partition' solution for all statements, for any non-empty subset of array accesses considered, then we find, for each statement, a time schedule that preserves all dependences and space partitions that are linearly independent to each other and the time schedule. The procedure is summarized in Algorithm 1.

5.3 Examples

Consider the kernels in Figure 5. Array a in mv and tmv kernels has no reuse and is considered for coalesced global memory access. Without applying the constraints defined by Equations 3 and 4, we get the following valid time schedule and space partition mapping for statement Q in mv kernel and statement T in tmv kernel.

$$\theta_Q(\vec{x_Q}) = j \text{ and } \pi_Q(\vec{x_Q}) = i$$

$$\theta_T(\vec{x_T}) = j \text{ and } \pi_T(\vec{x_T}) = i$$

where $\vec{x_Q} = \binom{i}{j}$ and $\vec{x_T} = \binom{i}{j}$.

Algorithm 1 Finding transformations enabling coalesced global memory access

Input Set of statements - $\mathcal{S}$, Iteration Space Polytopes of all statements $I_s, s \in S$, Array references (that do not have reuse) - $\{\mathcal{F}_{kzr}\}$, Set of Dependences - $\mathcal{R}$

1: **for** all non-empty subsets G of array references **do**
2: Find a time schedule θ for each statement s that preserves all dependences in $\mathcal{R}$ and satisfies *time schedule adjacency constraint* (3) for all references in G.
3: **for** each statement s (with dimensionality of iteration space being m and dimensionality of time schedule being t) **do**
4: Find a space partition π_1 that is linearly independent to θ and satisfies *space partition adjacency constraint* (4) for all references in G. Mark this space partition as the innermost space partition.
5: Find $m-t-1$ space partitions that are linearly independent to each other and also to π_1 and θ.
6: **end for**
7: **end for**
8: **if** no valid 'space-time partition' solution exists for all statements, for any non-empty subset of array references considered **then**
9: Find a time schedule θ for each statement s that preserves all dependences in $\mathcal{R}$.
10: For each statement s (with dimensionality of iteration space being m and dimensionality of time schedule being t), find $m-t$ space partitions that are linearly independent to each other and also to θ.
11: **end if**

Output Transformations enabling coalesced global memory access along with marking of references for which copy to shared memory is needed

Applying adjacency constraints for the mv kernel (in a system with row major storage) yields no valid transformation. Adjacent global memory access by distinct threads is possible only across different j-loop iterations of an i-loop iteration. Hence the time schedule adjacency constraint results in a time schedule $\theta_Q(\vec{x_Q}) = i$, which does not dismiss all dependences. Hence there is no valid transformation possible that can enable coalesced global memory access. Hence the transformation (time schedule and space partition mapping) obtained without applying adjacency constraints is used and array a in mv kernel is copied to shared memory and accessed, but not accessed directly from global memory.

On the other hand, applying adjacency constraints for the tmv kernel, yields a time schedule $\theta_T(\vec{x_T}) = j$ (as adjacent global memory access by distinct threads is possible across different i-loop iterations of an j-loop iteration) and a space partition mapping $\pi_T(\vec{x_T}) = i$, which preserve data dependences and hence resulting in a valid transformation that has optimal coalesced global memory access.

5.4 Effective Use of Register and Non-register Memories

The compiler framework not only makes decision on what data needs to be moved to shared memory and what needs to be accessed directly from global memory, but also makes optimal decisions on effectively using register memory and non-register memories such as constant memory. Constant memory has an on-chip portion in the form of cache which can be effectively utilized to reduce global memory access. Access to constant memory is useful when a small portion of data is accessed by threads in such a fashion that all threads in a warp access the same value simultaneously. When threads in a warp access different values in constant memory, then the requests are serialized.

We determine arrays that are read-only and whose access function does not vary with respect to the loop iterators corresponding to the parallel loops that are used for distributing computation across threads, as potential

candidates for storing in constant memory. Similarly arrays whose access function varies only with respect to the loop iterators corresponding to the parallel loops are considered as potential candidates for storing in registers in each thread.

5.5 Optimized Copy from Global Memory to Shared Memory

Arrays that have reuse and data accessed by array references that are marked to be copied to shared memory because of infeasible transformation for coalesced global memory access, have to be efficiently copied from/to shared memory. Given an iteration space polytope I and set of array access functions $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k$ of k references to an array in the iteration space, the elements accessed in the iteration space (further referred to as *accessed data space*) is given by

$$\mathcal{DS} = \bigcup_{j=1}^{k} \mathcal{F}_j I$$

where $\mathcal{F}_j I$ is the image of the iteration space polytope I formed by the affine access function $\mathcal{F}_j$ and it gives the elements accessed by the reference $\mathcal{F}_j$ in I.

For each array to be copied in a tile executed in a thread block, the accessed data space is determined for a given tiled iteration space using Polylib [32], a tool providing library functions for operations done over polyhedra. Using CLooG to scan the accessed data space polytope, we generate code to move data between global memory and shared memory. The loop structure of the copy code is a perfect nest of n loops, where n is the dimensionality of the accessed data space. By using a cyclic distribution of the innermost loop across threads of a warp, we enable interleaved access of global memory by threads. This results in adjacent words from global memory being accessed by adjacent threads, resulting in coalesced global memory access. The copy code, by default, is placed at the start of the tile to copy data required in a tile, and at the end of the tile to move modified data back to global memory. The copy code position in the loop structure of the program can be optimized by moving it across any redundant loops.

A detailed discussion on copy code generation is presented elsewhere [3].

5.6 Model to Estimate Memory Traffic

In this subsection, we discuss a model to estimate memory traffic expected during the execution of a tile. This is then used to guide the empirical search on tile sizes and unroll factors (as explained later in Section 7). Consider a tile to be executed by a thread block or a thread. The iteration space of statements in the tile is parameterized by the tile sizes of the loops defining the tile. Consider a tile of n loops with tile sizes being $t_1, t_2, \ldots, t_n$. Consider k arrays $(a_1, a_2, \ldots, a_k)$ being accessed in the tile. Let r_i be the number of read references and w_i be the number of write references of array a_i. Let $\mathcal{F}_{i1}, \mathcal{F}_{i2}, \ldots, \mathcal{F}_{ir_i}$ be the read accesses of array a_i in the tile and $\mathcal{G}_{i1}, \mathcal{G}_{i2}, \ldots, \mathcal{G}_{iw_i}$ be the write accesses of array a_i in the tile. Let I be the iteration space of the tile parameterized by the tile sizes. Let f be a function that counts the number of integer points in a polytope given the parameters. Let $\mathcal{DS}_{l_i}$ denote the accessed data space of read references of array a_i. The number of integer points in polytope $\mathcal{DS}_{l_i}$ gives the number of loads due to array a_i. Let $\mathcal{DS}_{s_i}$ denote the accessed data space of write references of array a_i. The number of integer points in $\mathcal{DS}_{s_i}$ gives the number of stores due to array a_i.

The model to estimate memory loads and stores in a tile can be characterized as follows.

$$\mathcal{DS}_{l_i} = \bigcup_{j=1}^{r_i} \mathcal{F}_{ij} I \quad \text{and} \quad \mathcal{DS}_{s_i} = \bigcup_{j=1}^{w_i} \mathcal{G}_{ij} I$$

The number of loads and stores in a tile =

$$\sum_{i=1}^{k} f(\mathcal{DS}_{l_i}, t_1, t_2, \ldots, t_n) + f(\mathcal{DS}_{s_i}, t_1, t_2, \ldots, t_n)$$

Having modeled the number of loads and stores in a tile, the total memory traffic is estimated based on the number of iterations in the tiled iteration space, i.e., the number of tiles.

6. Optimizing Shared Memory Access

This section describes our approach to optimize access of on-chip shared memory in GPU multiprocessor units. Following the observation from Section 3.2, optimization of shared memory access can be equivalently viewed as minimization of bank conflicts. The strategy to minimize bank conflicts in shared memory access is to *pad* the arrays copied into shared memory. However, finding a suitable padding factor for an array in shared memory is not trivial. The procedure of finding a padding factor for an array in order to minimize bank conflicts has to consider the effects of padding on all references made to the array. Padding to minimize bank conflict with respect to one reference might have a negative impact with respect to another reference.

We define a formal relation between the degree of bank conflicts and the access stride across threads in a half warp that determine the degree of bank conflicts and hence the shared memory access bandwidth. With shared memory organized into banks and successive words stored in successive banks in a cyclic pattern, the degree of bank conflicts is given by *GCD(stride of array access across threads of a half warp, number of bank modules)*.

We model the cost of accessing a word from a shared memory bank as a linear function of the degree of bank conflicts. Let $C(n)$ be the cost of accessing a word from a shared memory bank when there are n simultaneous requests to the bank (possibly by different threads of a half warp). The cost function is given by

$$C(n) = t_{start} + t_{request} \times n \tag{5}$$

where t_{start} is the startup time to access a bank when there is one or more requests to the bank and $t_{request}$ is the time to service a request. Figure 6 shows the trend of the linear shared memory bank access cost function (plotted using data obtained from the experiment described in Section 3.2).

The algorithm to find optimal padding factors for arrays in shared memory takes as input a sub-program to be executed in a thread block which has been transformed for global memory coalescing using the framework from Section 5.2. Thus, the algorithm has information regarding arrays that would be copied into shared memory, as well as the space and time partitions in the transformed sub-program. For each reference, the distance between successive points accessed by successive iterations of the innermost space loop provides the access stride across threads for that reference. This information is provided as input to the algorithm that finds optimal padding factor for each array in shared memory (shown in Algorithm 2). For each array in shared memory, the algorithm

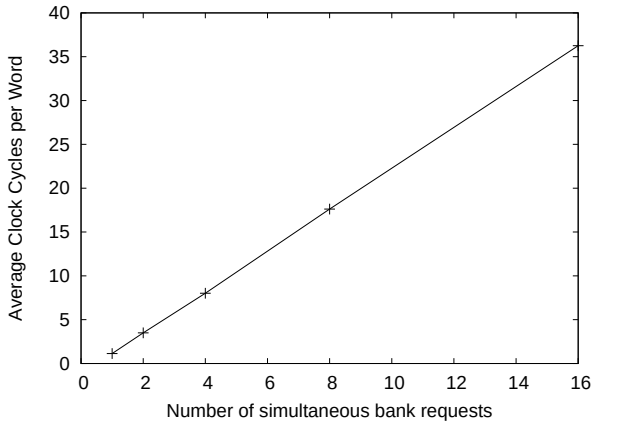


Figure 6. Shared memory bank access time for varying simultaneous bank requests

enumerates all effective padding factors (which vary from 1 to number of shared memory bank modules) to find the optimal one that minimizes the total number of bank conflicts caused by all the references of the array.

Algorithm 2 Finding Optimal Padding Factor

Input Input array for which padding is to be determined - A, Number of references to A in the sub-program - N_{ref}, Original access strides across threads for the N_{ref} references - $AS[N_{ref}]$, Number of bank modules - NB, Cost function from Eq. (5) - C

1: $MinAccessCost = 0$
2: $OptPadding = 1$
3: **for** $pad = 1$ to NB **do**
4: $TotalAccessCost = 0$
5: **for** $ref = 1$ to N_{ref} **do**
6: Calculate new access stride $AS_{new}[ref]$ for reference ref using original access stride $AS[ref]$ and new padding factor pad
7: $BankConflict[ref] = \text{GCD}(AS_{new}[ref], NB)$
8: $AccessCost[ref] = C(BankConflict[ref])$
9: $TotalAccessCost += AccessCost[ref]$
10: **end for**
11: **if** $TotalAccessCost < MinAccessCost$ **then**
12: $OptPadding = pad$
13: $MinAccessCost = TotalAccessCost$
14: **end if**
15: **end for**

Output Optimal padding factor for A - $OptPadding$

7. Model-driven Empirical Search for Optimal Tile sizes and Unroll factors

In this section, we discuss optimization of program parameters such as tile sizes and unroll factors that are closely linked with the choice of system parameters such as number of threads and number of thread blocks used for execution, and the availability of GPU local resources such as shared memory and registers.

We perform multiple levels of tiling for exploiting parallelism across thread blocks and threads, and also perform register-level tiling through unroll-and-jam to optimize thread-level code. The first level of tiling is

done to exploit parallelism across thread blocks. In GPUs, the size of a tile executing in a thread block at a time instance depends on the amount of shared memory available for execution of the thread block. The second level of tiling within a thread block is done, if needed, to bound shared memory usage within available limits. When the number of iteration points in a loop executed within a thread block is more than the number of threads, one more level of tiling is needed to distribute the computation across threads. Tiling to distribute computation across thread blocks and threads is performed according to the space partition mapping derived by the framework, as explained in Section 5.2. Finally, if there is enough reuse to be exploited, register-level tiling is done to reduce the number of loads from global/shared memory.

For a GPU architecture, performance is enhanced by optimizing memory access and exploiting parallelism, as illustrated in Section 3. Hence it would be ideal to characterize and model tile size determination based on the number of loads/stores between global and shared memory, and the number of loads/stores between shared memory and registers. Using the polyhedral model discussed in Section 5.6, we can obtain an accurate estimate of memory loads/stores. However, because of the lack of control on the the number of registers actually used by NVIDIA CUDA C Compiler (NVCC), and because of the tight coupling of the GPU resources, optimal tile sizes and unroll factors cannot be determined by a cost model alone. An empirical search is needed to find an optimal set of tile sizes for the tiled loops and optimal unroll factors for the loops that are unrolled. Hence in our framework, we employ an empirical search to pick the optimal code among various code variants resulting due to different transformations enabling efficient global memory access, different tile sizes at multiple levels, and different unroll factors. The search space due to different choices of tile sizes and unroll factors are pruned with the help of the cost model that estimates memory laods/stores.

The model-guided empirical search procedure used in our compiler framework is outlined below.

- For each valid program transformation structure obtained by the framework described in Section 5.2, perform multi-level tiling (except register-level tiling).
- Generate optimal copy code for arrays that need to be copied to shared memory (as explained in Section 5.5).
- For each tiled loop structure, determine the register usage r and determine the maximum concurrency (L threads) possible within a multiprocessor. (NVCC has an option to generate a low-level object code file called the *cubin* file that provides information on the amount of shared memory used by a thread block and the number of registers used by a thread in a thread block). Set the exploration space of number of threads in a thread block to be $T, T/2, T/4$, where T is the nearest multiple of *warp size* of the GPU device less than L and 512.
- For all valid tile sizes that distribute computation almost equally among thread blocks and also among threads within a thread block, and satisfy shared memory limit constraint, estimate the total number of global memory loads/stores using the polyhedral model in Section 5.6. Discard loop structures that have p% more loads/stores than the structure with lowest number of loads/stores.
- For all selected loop structures, do register-level tiling and explicit unrolling, instrument the register usage and discard those for which register pressure is increased to an extent where concurrency is reduced to less than 25% of maximum possible concurrency.

N	Direct Global	Optimized Shared	Non-Optimized Shared
4K	0.43	13.18	5.61
5K	0.48	13.87	5.79
6K	0.35	14.37	6.04
7K	0.30	13.86	5.78
8K	0.24	13.63	5.52

Table 2. Performance comparison (in GFLOPS) of mv kernel

N	Non-optimized Global	Optimized Global
4K	4.22	25.21
5K	3.09	28.90
6K	3.24	33.47
7K	3.70	33.58
8K	4.13	34.93

Table 3. Performance comparison (in GFLOPS) of tmv kernel

- In all selected code versions, pad the arrays in shared memory with optimal padding factor determined using Algorithm 2.
- Search empirically among the remaining candidate loop structures by explicitly running them and timing the execution time and select the optimal one.

8. Experimental Results

The experiments were conducted on a NVIDIA GeForce 8800 GTX GPU device. The device has 768 MB of DRAM and has 16 multiprocessors (MIMD units) clocked at 675 MHz. Each multiprocessor has 8 SIMD units running at twice the clock frequency of the multiprocessor and has 16 KB of shared memory per multiprocessor. We used CUDA version 1.0 for our experiments. The CUDA code is compiled using the NVIDIA CUDA Compiler (NVCC) to generate the device code that is launched from the CPU (host). The CPU is an Intel Core2 Duo processor at 2.13 GHz with 2 MB L2 cache. The GPU device is connected to the CPU through a 16-x PCI Express bus. The host programs are compiled using the icc compiler at -O3 optimization level.

8.1 Illustration of the Benefits of Optimized Global Memory Access

We discuss the performance of the mv and tmv kernels (Figure 5) to illustrate the benefits of global memory access optimization. Table 2 shows the performance of mv kernel implemented using space and time partition mappings, as discussed in Section 5.3. An implementation with efficient copy of elements of array *a* from global memory to shared memory (column “Optimized Shared”), as deduced by the framework, provides an order of magnitude better performance than the version implemented with direct access of *a* from global memory (column “Direct Global”). In addition, an implementation without optimized padding of array *a* in shared memory (column “Non-Optimized Shared”) leads to a 2x degradation in performance.

Table 3 shows the performance of the tmv kernel implemented using the space and time partition mappings discussed in Section 5.3. When tiling along space loops is done in a blocked fashion to map virtual processors in a space partition mapping to threads, it violates the coalesced memory access constraints and performance degrades (column “Non-Optimized Global”). Hence tiling along space loops is done in a cyclic fashion (column “Optimized Global”), as inferred with the optimization framework.

8.2 Illustration of the Model-driven Empirical Search using Matrix-Matrix Multiply (MM) kernel

We use the MM kernel to illustrate the steps involved in the model-driven empirical search procedure explained in Section 7. We used a problem size of $4K \times 4K$, a size that is large enough to just fit in the GPU DRAM.

For the multi-level tiled code generated using the program transformations (without loop unrolling and register-level tiling), the register usage per thread varied was estimated using *cubin* as 13, leading to a possibility of 512 concurrent threads. Further experiments were done for a thread space of 128, 256 and 512 threads per thread block. The number of thread blocks was varied between 16, 32 and 64.

For various tile sizes that distribute computation almost equally among thread blocks and also equally among threads within a thread block, and satisfy shared memory limit constraint, the total global memory loads varied from the order of $\frac{4K^3}{2^7}$ to $\frac{4K^3}{2^4}$. The framework considered all code versions that had loads in the order of $\frac{4K^3}{2^7}$ to $\frac{4K^3}{2^6}$ and various combinations of loop unrolling and register-level tiling were done for the selected code versions. Since the choices of register-level tiling depend on the size of the tile being executed in a thread, the choices were limited. The register usage of each unrolled, register-tiled version was found to eliminate those that had excessive register usage that reduces the number of concurrent threads to below 128. Figure 7 illustrates the performance of the selected candidates that were run empirically to select the best one. Our approach resulted in a performance of around 97 GFLOPS as compared to vendor-optimized MM kernel's performance of around 101 GFLOPS.

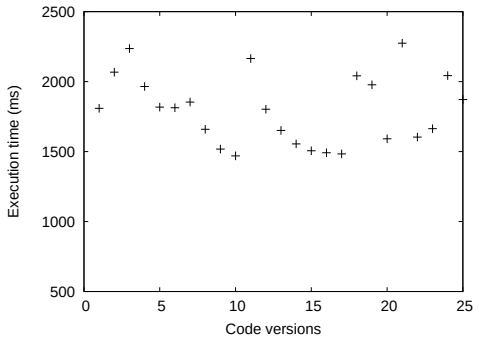
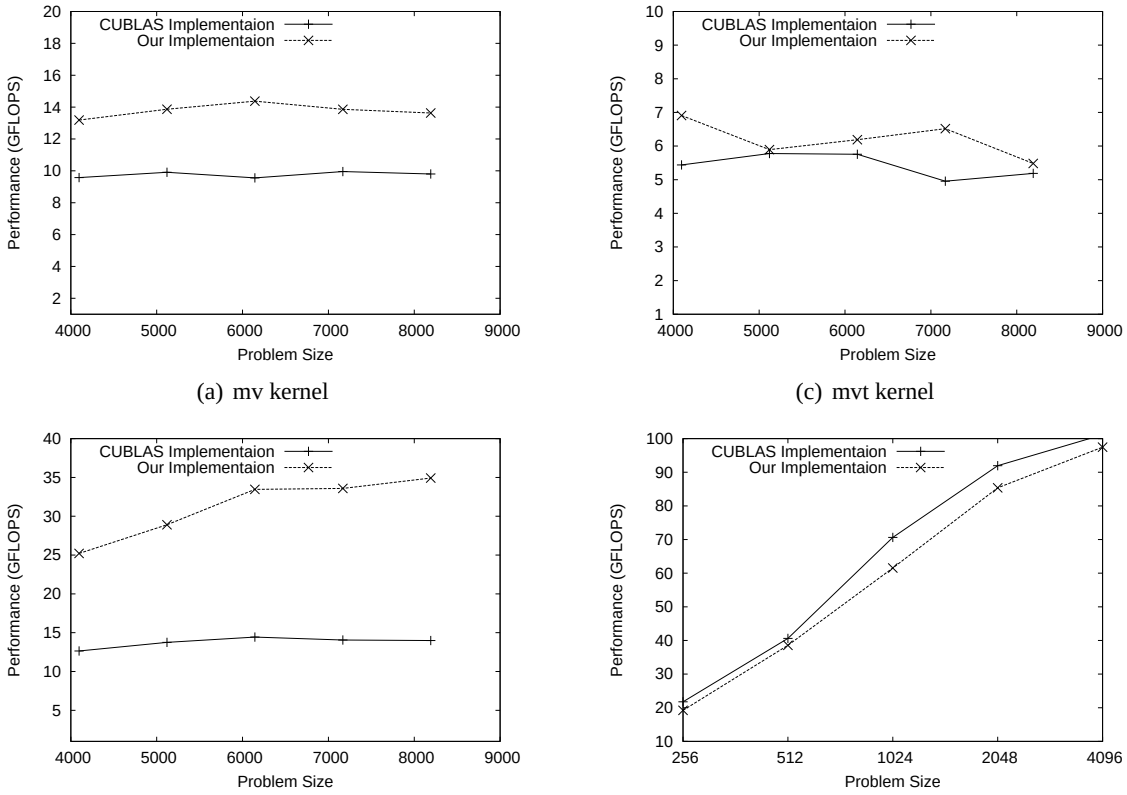


Figure 7. Performance of MM kernel for various code versions

8.3 Performance Evaluation on Kernels

8.3.1 Matrix Kernels

Figure 8 shows the performance of several kernels: Matrix Vector multiply (mv), Transpose Matrix Vector multiply (tmv), Matrix Vector Transpose (mvt), which involves both mv and tmv kernels, and Matrix Matrix multiply (mm). The comparison is done with the vendor-optimized CUBLAS library from CUDA. The strength of the proposed optimization framework is evident from these results: for mv, tmv and mvt kernels, our approach achieves better performance than the CUBLAS implementation, and for mm kernel, the performance is close to CUBLAS.



(a) mv kernel

(c) mvt kernel

(b) tmv kernel

(d) mm kernel

Figure 8. Performance of Matrix kernels

Version	MRI-Q	MRI-FHD
CPU	0.17	0.18
Non-optimized GPU	9.62	10.14
Optimized GPU	159.79	154.93

Table 4. Performance of MRI kernels

8.3.2 Magnetic Resonance Imaging Kernels

We employed our framework for few important kernels used in Magnetic Resonance Imaging [38, 37] application, namely, MRI-Q and MRI-FHD. The performance improvement over the non-optimized versions of the kernels are clearly illustrated in Table 4. The performance improvement is primarily due to efficient tile sizes and unroll factors determined, and utilization of constant memory. Ryoo et al. [38, 37] report performance improvement of these kernels in their work, but our approach detects the optimization to be performed in an automatic fashion unlike their approach that is aimed at manually improving the performance of the kernels.

8.3.3 FDTD Kernel

Figure 9 shows the performance of 2D Finite-Difference Time-Domain (FDTD) kernel (a realistic scientific kernel characterized by a Jacobi-like stencil computation) generated using our approach.

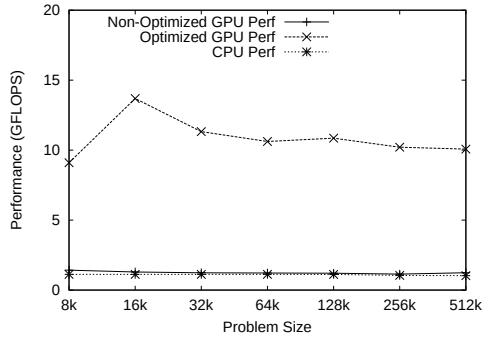


Figure 9. Performance of FDTD kernel

9. Related Work

Prior to the introduction of CUDA [29], GPU programming systems have relied on graphics API-based implementations, which have limited the size and kind of codes that have been implementable on GPUs. In addition, CUDA has significantly enhanced programmer productivity by relieving the programmer of the burden of thinking in terms of graphics operations.

Previous GPU generations and their APIs had restrictive memory access patterns such as allowing only sequential writes to a linear array. For example, Accelerator [40] does not allow access to an individual element in parallel arrays and operations are performed on all array elements. Brook [8] is a stream-based model that executes its kernel for every element in the stream with restrictions. The GeForce 8800 allows for general addressing of memory by each thread, which supports a much wider variety of algorithms. With this general addressing, it is important to apply data locality optimizations in order to exploit high bandwidth and hide memory latency.

Traditional GPUs also provided limited cache bandwidth for GPGPU applications. Fatahalian et al. [13] mention that low-bandwidth cache designs on GPUs prevent general purpose applications from benefiting from the available computational power. Govindaraju et al. [19] use an analytical cache performance prediction model for GPU-based algorithms. Their results indicate that memory optimization techniques designed for CPU-based algorithms may not directly translate to GPUs. New graphics architectures provide a variety of storage resources, which need to be exploited for getting good performance. In the context of architectures with explicitly managed memory hierarchies, Fatahalian et al. [12] and Knight et al. [25] present respectively a language and an optimizing compiler system.

In the context of getting good performance on graphics applications running on CPUs, two works have developed compiler solutions. Breternitz et al. [7] have developed a compiler to generate efficient code on a CPU for SIMD graphic workloads by extending the base ISA to SSE2. Liao et al. [26] have developed a framework that works with Brook [8] to perform aggressive data and computation transformations. Recently, Ryoo et al. [38, 37] have presented experimental studies on program performance on NVIDIA GPUs using CUDA; they do not use or develop a compiler framework for optimizing applications, but rather perform the optimizations manually. Ryoo et al. in [39] have presented performance metrics to prune the optimization search space on a pareto-optimality basis. However, they manually generate the performance metrics data for each application they have studied.

A number of efforts have focused on automatically generating efficient implementations of programs for different architectures, though none of these have addressed GPUs. A significant example in this space is the SPIRAL project which is aimed at the design of a system to generate efficient libraries for digital signal processing algorithms [34]. Other efforts include FFTW [18], the telescoping languages project [23], works on iterative compilation [1, 9, 24], ATLAS [42] for deriving efficient implementation of BLAS routines, and the PHIPAC [5] and the OSKI [22] projects. All these efforts use search-based approaches for performance tuning of codes. A comparison of model-based and search-based approaches for matrix-matrix multiplication is reported in [43].

10. Conclusions

We have characterized critical performance influencing factors on GPUs and have developed an automatic compiler framework to efficiently enable an optimized execution over GPUs. We have developed techniques to generate effective program transformations for GPUs, and have employed a model-driven empirical optimization approach to find optimal values for system and program parameters that maximize performance. The effectiveness of the developed approach is demonstrated with various kernels.

Acknowledgments This work is supported in part by the U.S. National Science Foundation through awards 0121676, 0121706, 0403342, 0508245, 0509442, 0509467 and 0541409.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, 2006.

[2] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *PPoPP'91*, pages 39–50, 1991.

[3] 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 *ACM SIGPLAN PPoPP 2008*, Feb. 2008.

[4] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'04*, pages 7–16, 2004.

[5] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC. In *ICS*, pages 340–347, 1997.

[6] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction (ETAPS CC)*, Apr. 2008.

[7] M. Breternitz, H. Hum, and S. Kumar. Compilation, Architectural Support, and Evaluation of SIMD Graphics Pipeline Programs on a General-Purpose CPU. In *PACT 2003*, 2003.

[8] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04*, pages 777–786, 2004.

[9] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–197, 2007.

[10] CLooG: The Chunky Loop Generator. http://www.cloog.org.

[11] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *IJPP*, 25(6):447–496, Dec. 1997.

[12] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06*, 2006.

[13] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 133–137, 2004.
[14] P. Feautrier. Dataflow analysis of array and scalar references. *IJPP*, 20(1):23–53, 1991.
[15] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I: one-dimensional time. *IJPP*, 21(5):313–348, 1992.
[16] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *IJPP*, 21(6):389–420, 1992.
[17] P. Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103, 1996.
[18] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".
[19] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *SC '06*, 2006.
[20] General-Purpose Computation Using Graphics Hardware. http://www.gpgpu.org/.
[21] M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. FMI, University of Passau, 2004. Habilitation Thesis.
[22] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, 2004.
[23] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, S. Johnsson, J. Mellor-Crummey, and L. Torczon. Telescoping Languages: A Strategy for Automatic Generation of Scientific Problem-Solving Systems from Annotated Libraries. *JPDC*, 61(12):1803–1826, 2001.
[24] T. Kisuki, P. Knijnenburg, and M. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 237, 2000.
[25] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan. Compilation for explicitly managed memory hierarchies. In *PPoPP '07*, pages 226–236, 2007.
[26] S.-W. Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and computation transformations for Brook streaming applications on multiprocessors. In *CGO '06*, pages 196–207, 2006.
[27] A. Lim. *Improving Parallelism And Data Locality With Affine Partitioning*. PhD thesis, Stanford University, Aug. 2001.
[28] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL*, pages 201–214, 1997.
[29] NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html.
[30] NVIDIA GeForce 8800. http://www.nvidia.com/page/geforce_8800.html.
[31] PLuTo: A polyhedral automatic parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net.
[32] PolyLib - A library of polyhedral functions. http://icps.u-strasbg.fr/polylib/.
[33] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *CGO '07*, pages 144–156, 2007.
[34] M. Pueschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, February 2005.
[35] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, Aug. 1992.
[36] F. Quilleré, S. V. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *IJPP*, 28(5):469–

498, 2000.

[37] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *ACM SIGPLAN PPoPP 2008*, Feb. 2008.

[38] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, and W. Hwu. Program optimization study on a 128-core GPU. In *The First Workshop on General Purpose Processing on Graphics Processing Units*, October 2007.

[39] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, J. Stratton, and W. Hwu. Program optimization space pruning for a multithreaded GPU. In *CGO*, 2008.

[40] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ASPLOS-XII*, pages 325–335, 2006.

[41] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *International Conference on Compiler Construction (ETAPS CC'06)*, pages 185–201, Mar. 2006.

[42] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proc. Supercomputing '98*, 1998.

[43] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, Feb 2005.