A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs

Muthu Manikandan Baskaran
Department of Computer Science and Engg.
The Ohio State University
baskaran@cse.ohio-state.edu

Uday Bondhugula
Department of Computer Science and Engg.
The Ohio State University
bondhugu@cse.ohio-state.edu

Sriram Krishnamoorthy
Department of Computer Science and Engg.
The Ohio State University
krishnsr@cse.ohio-state.edu

J. Ramanujam
Dept. of Electrical & Computer Engg.
Louisiana State University
jxr@ece.lsu.edu

Atanas Rountev
Department of Computer Science and Engg.
The Ohio State University
rountev@cse.ohio-state.edu

P. Sadayappan
Department of Computer Science and Engg.
The Ohio State University
saday@cse.ohio-state.edu

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 their GPUs. However, manual development of high-performance parallel code for GPUs is still very challenging. In this paper, a number of issues are addressed towards the goal of developing a compiler framework for automatic parallelization and performance optimization of affine loop nests on GPGPUs: 1) approach to program transformation for efficient data access from GPU global memory, using a polyhedral compiler model of data dependence abstraction and program transformation; 2) determination of optimal padding factors for conflict-minimal data access from GPU shared memory; and 3) model-driven empirical search to determine optimal parameters for unrolling and tiling. Experimental results on a number of kernels demonstrate the effectiveness of the compiler optimization approaches developed.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors — Compilers, Optimization

General Terms: Algorithms, Design, Performance

Keywords: GPU, Polyhedral Model, Memory Access Optimization, Empirical Tuning

1. INTRODUCTION

Graphics Processing Units (GPUs) are now 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 [19]. 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) [8, 13, 12]. Until very recently, general-purpose computations on GPUs 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 provided a general-purpose multi-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 the 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 compiler support to facilitate the automatic transformation of sequential input programs into efficient parallel CUDA programs.

There has been significant progress over the last two decades in the development of powerful compiler frameworks for dependence analysis and transformation of loop computations with affine bounds and array access functions [1, 22, 16, 14, 9, 23, 21, 3]. For such regular programs, compile-time optimization approaches have been developed using affine scheduling functions with a polyhedral abstraction of programs and data dependencies. Although the polyhedral model of dependence abstraction and program transformation is much more powerful than the traditional model of data dependencies currently used in production optimizing compilers, early polyhedral approaches were not practically efficient. Recent advances in dependence analysis and code generation [23, 3, 28] have addressed many of these issues, resulting in polyhedral techniques being applied to code representative of real applications such as the spec2000fp benchmarks. CLooG [3, 7] is a powerful state-of-the-art code generator that captures most of these advances. Building on these developments, 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 [4, 5, 20]. The effectiveness of the transformation system has been demonstrated on a number of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'08, June 7–12, 2008, Island of Kos, Aegean Sea, Greece.
Copyright 2008 ACM 978-1-60558-158-3/08/06 ...$5.00.

non-trivial application kernels for multi-core processors. However, building such a framework for GPUs requires attention to several additional issues. In this paper we identify and characterize key factors that affect GPGPU performance and develop compile-time transformation approaches for GPGPU optimization.

The paper is organized as follows. In Section 2, we provide a brief overview of the NVIDIA GeForce 8800 GTX GPU. Section 3 develops an empirical characterization pertaining to 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 three sections, we discuss compilation techniques for GPGPUs that systematically address these performance-critical issues. Section 7 presents experimental performance results that demonstrate the effectiveness of the developed techniques. Related work is discussed in Section 8. We conclude in Section 9.

2. OVERVIEW OF GPU ARCHITECTURE AND CUDA PROGRAMMING

The NVIDIA GeForce 8800 GTX has 16 multiprocessor units, each consisting of 8 processor cores that execute in SIMD manner. The processors (SIMD units) within a multiprocessor unit communicate through a fast on-chip *shared memory*, 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 interface exposed by the NVIDIA Compute Unified Device Architecture (CUDA) model [18]. 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 16 KB of shared memory 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. In general, in a multiprocessor unit of a GPU device that has R registers and M KB shared memory, if the number of registers used per thread is r, the shared memory required per thread block is m KB, and the number of threads per thread block is p, the maximum number of active concurrent thread blocks in the multiprocessor at any time cannot exceed $min(\lfloor \frac{R}{p \times r} \rfloor, \lfloor \frac{M}{m} \rfloor)$.

The various memories available in GPUs for a programmer are as follows: (1) off-chip global memory (768MB on the 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.

M	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

3. PERFORMANCE CHARACTERIZATION OF GPGPU

In this section, micro-benchmarks are used to characterize key factors that affect GPGPU performance and the implications for compiler optimization are discussed.

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 maximizing data reuse helps to improve the performance of programs with temporal locality, reducing the latency in accessing data from global memory is critical for good performance.

The cost of global memory access was characterized by measuring the memory read bandwidth achieved for different data sizes, for blocked and cyclic distribution of computation amongst the threads. In the micro-benchmark used for bandwidth measurement, a one-dimensional array of size M (where $M = 16 \times N$) was accessed from global memory by 16 thread blocks (one mapped to each multiprocessor unit), where each thread block was a grid of T threads. Each thread in a thread block accessed N/T elements of the array (N was 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 next set of N/T elements, $\ldots$, 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, $\ldots$, 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 resulted 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 – *global memory access coalescing*. Accesses from adjacent threads in a half-warp to adjacent locations (that are aligned to 4, 8, or 16 bytes) in global memory are coalesced into a single contiguous aligned memory access. Interleaved access to global memory by threads in a thread block is essential to exploit this architectural feature.

Using cyclic data access by threads, the effect on achieved memory bandwidth was evaluated for different numbers of threads per thread block. In addition, the impact of strided data access on memory performance was evaluated. 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. It may be observed that non-unit strides across threads lead to significant degradation in performance. This is because global memory coalescing only happens

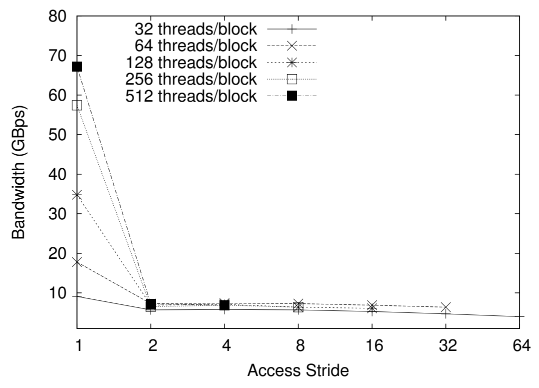


Figure 1: Global memory bandwidth for different access strides for varying number of threads per thread block

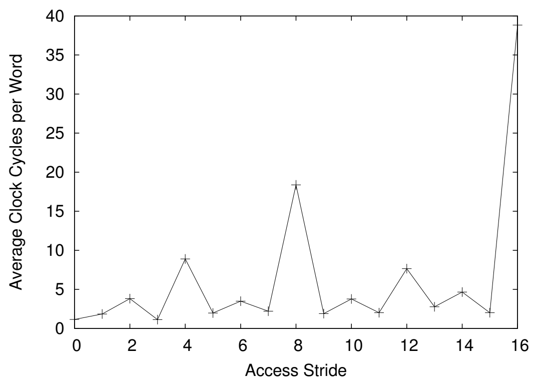


Figure 2: Shared memory access time for different access strides across threads

with unit stride access across threads. With non-unit access strides, all accesses are issued individually to memory, resulting in poor performance. With unit access stride, as the number of threads per thread block is increased, an increase in the memory bandwidth is observed, 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 make it one of the most important optimizations to be enabled by a compiler framework for GPGPUs. Also, the high latency of global memory access highlights the importance of reducing the number of global memory loads/stores.

3.2 Shared Memory Access

The shared memory is a fast on-chip software-managed memory space 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 addresses 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 simultaneous request to a bank). The bandwidth of shared memory access is inversely proportional to the degree of bank conflicts.

We conducted an experiment to study the effect of bank conflicts, by measuring the shared memory access time for different access strides (strides from 0 to 16) across threads. 32 threads per thread block were used for the experiment and each thread accessed 100 data elements (100 32-bit words). When access stride is 0, all threads access the same word. 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 0 and thread 8, thread 1 and thread 9, ..., thread 7 and thread 15. 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. From Figure 2, we also observe that the access time when all threads access the same word is as fast as that when there are no bank conflicts. This is due to a hardware optimization in the shared memory that enables *broadcast* of a 32-bit word to several threads simultaneously when servicing one memory read request.

The importance of the fast on-chip shared memory space that is introduced in GPU architectures to improve the memory access performance elucidates the fact that minimizing shared memory bank conflicts is an important optimization to be handled by a compiler framework for GPGPUs.

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. 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 number of threads in the system.

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 large global memory access overhead requires higher concurrency and also demands 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.

Having identified the key performance-influencing characteristics of GPUs, we now discuss the compile-time optimization approaches developed to address these issues, towards the goal of developing a compiler framework for automatic parallelization and performance optimization of affine loop nests on GPGPUs.

4. OPTIMIZING GLOBAL MEMORY ACCESS

In this section, we develop an approach for performing program transformations that enable interleaved access to global memory by threads in a thread block which is necessary to facilitate coalesced global memory accesses and thereby improve global memory access performance. The approach is based on the polyhedral model, a powerful algebraic framework for representing programs and transformations [17, 21]. The 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 assumed to be 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.

4.1 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 is equivalently 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).

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: $rank(F_{kAS}) < dim(\vec{x_S})$. Loops whose iterators do not occur in the affine access function of a reference are said to be *redundant loops* for the reference.

```
mv kernel:                              tmv kernel  :

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

Figure 3: mv and tmv kernels

Given an iteration space polytope I and a 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 set of array 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 set of elements accessed by the reference $\mathcal{F}_j$ in I.

Consider the Matrix Vector (mv) multiply and Transpose Matrix Vector (tmv) Multiply kernels in Figure 3. The iteration space polytope of statement T is defined by $\{i, j \mid 0 \leq i \leq n-1 \ \wedge \ 0 \leq j \leq n-1\}$. The access function of the reference to array a in statement T is represented as

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

where $\vec{x_T} = \binom{i}{j}$ is the iteration vector of statement T. 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 much work on dependence analysis in the polyhedral model [9, 22]. Feautrier [10, 11] 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 dependencies. 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.

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

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., [10, 16]) 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.

4.2 Global Memory Access 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 develop the global memory access optimization approach on top of PLuTo [20], an effective automatic transformation framework. PLuTo optimizes sequences of imperfectly nested loops, simultaneously for parallelism and locality, through tiling transformations. We use PLuTo to generate tiling transformations to distribute tiles across thread blocks and a tile (with iteration space defined by the shape of the tile) is given as input to the global memory access optimization approach.

We first determine array references whose accessed data space have either higher-order reuse or sufficient constant reuse and mark them as candidates that have to be copied from global memory to shared memory for efficient performance. Array references whose accessed data space have no reuse 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 find program transformations that can lead to efficient direct global memory access of as many 'candidate' array references 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 (along the fastest varying dimension) of the array due to the reference have to be executed simultaneously (in time) by distinct virtual processors that are consecutive in processor space. This is enforced by the *time schedule adjacency constraint* which enforces two statement instances that access adjacent elements of an array to be executed at the time instance. The *time schedule adjacency constraint* is defined (assuming row major storage of arrays) as:

$$\forall \vec{x_s} \in \mathcal{D}_s, \forall \vec{y_s} \in \mathcal{D}_s \text{ s.t. } \mathcal{F}_{rzs}(\vec{x_s}) + (0\ldots1)^T = \mathcal{F}_{rzs}(\vec{y_s}),$$
$$\theta_s(\vec{x_s}) = \theta_s(\vec{y_s}) \qquad (1)$$

In addition to the above constraint, iterations accessing adjacent data elements of an array have to be in adjacent space partitions so that they are accessed by adjacent virtual processors. This is enforced by the *space partition adjacency constraint* which enforces two statement instances that access adjacent elements of an array to be executed by adjacent processors in the processor space. The *space partition adjacency constraint* is defined as:

$$\forall \vec{x_s} \in \mathcal{D}_s, \forall \vec{y_s} \in \mathcal{D}_s \text{ s.t. } \mathcal{F}_{rzs}(\vec{x_s}) + (0\ldots1)^T = \mathcal{F}_{rzs}(\vec{y_s}),$$
$$\pi_s(\vec{y_s}) = \pi_s(\vec{x_s}) + 1 \qquad (2)$$

The space adjacency constraint also enforces 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, that enable interleaved global memory access by threads, for code executed in a thread block. In our approach, we solve for a time schedule (for each statement) that preserves all dependencies and satisfies *time schedule adjacency constraint* (Equation 1) for all candidate array references whose accessed data space do not have enough reuse in the program. If there does not exist a solution, we try all subsets of those array references and generate time schedules that satisfy the *time schedule adjacency constraint* and potentially generate space partitions that satisfy the *space partition adjacency constraint* (Equation 2). 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) satisfies the *space partition adjacency constraint*. If there is no valid space-time transformation satisfying dependencies and adjacency constraints, for all statements, for any non-empty subset of array references considered, then we use PLuTo-generated tiling transformation (space-time transformation generated without enforcing adjacency constraints) for each statement. The procedure is summarized in Algorithm 1. All valid transformations that are determined by the procedure are considered as candidate transformations for an empirical search (discussed in Section 6).

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 (whose accessed data space do not have reuse) - $\{\mathcal{F}_{rzs}\}$, Set of Dependencies - $\mathcal{R}$
1:  for all non-empty subsets $G$ of array references do
2:      Find a time schedule $\theta$ for each statement $s$ that preserves all dependencies in $\mathcal{R}$ and satisfies time schedule adjacency constraint (1) 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 $\pi_1$ that is linearly independent to $\theta$ and satisfies space partition adjacency constraint (2) 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 $\pi_1$ and $\theta$.
6:      end for
7:  end for
8:  if no valid space-time transformation (satisfying adjacency constraints) exists for all statements, for any non-empty subset of array references considered then
9:      Use tiling transformation (space-time transformation) generated by PLuTo without enforcing adjacency constraints, for each statement.
10: end if
Output  Transformations enabling coalesced global memory access along with marking of references for which copy to shared memory is needed
```

4.3 Examples

Consider the kernels in Figure 3. Array a in mv and tmv kernels has no reuse and is considered for direct global memory access. Without applying the constraints defined by Equations 1 and 2, 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 $\vec{x_T} = \binom{i}{j}$.

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 dependencies. 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 dependencies, and hence results in a valid transformation that can enable coalesced global memory access.

4.4 Effective Use of Register and Non-register Memories

The approach 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 decisions on effectively using register memory and non-register memories such as constant memory, and thereby reduces the number of global memory accesses. 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, and consider them as 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 candidates for storing in registers in each thread.

4.5 Optimized Copy from Global Memory to Shared Memory

Array references that have sufficient reuse and 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. A detailed discussion on the approach to determine accessed data spaces of array references, automatically allocate storage space in the form of arrays in shared memory, and generate code to move data between global and shared memories is presented in one of our earlier works [2]. The loop structure of the data movement code (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.

4.6 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 6). 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 tiles in the tiled iteration space.

5. 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)*. When the stride of array access across threads is zero, i.e. when all threads access the same word, there is a special hardware optimization in the GPU architecture that enables *broadcast* of the word to all threads. Hence in that case, the degree of bank conflicts is considered as one as there is only one simultaneous bank request.

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

$$C(n) = t_{start} + t_{request} \times n \tag{3}$$

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 4 shows the trend of the linear shared memory bank access cost function (plotted using data obtained from the experiment described in Section 3.2).

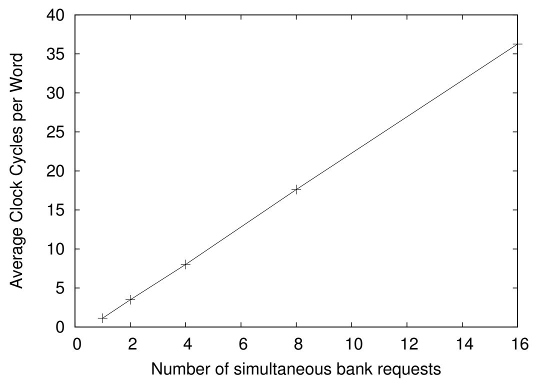


Figure 4: Shared memory bank access time for varying simultaneous bank requests

The algorithm to find optimal padding factors for arrays in shared memory focuses on the code to be executed in a thread block which has been transformed for global memory coalescing using the approach described in Section 4.2. The algorithm has information regarding arrays that need to be copied into shared memory, as well as the space and time partitions of each statement in the code. For each reference, based on the distribution of innermost space loop across threads (block or cyclic), the distance between data points accessed by successive threads is calculated and it defines the access stride across threads for that reference. For each array in shared memory, the algorithm enumerates all padding factors and finds the optimal one that minimizes the total number of bank conflicts caused by all the references to the array.

6. 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. Finally, if there is enough reuse to be exploited, register-level tiling is done to reduce the number of loads/stores from global/shared memory.

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. (3) - $C$
 1: $MinAccessCost = \infty$
 2: $OptPadding = 0$
 3: for $pad = 0$ to $NB - 1$ 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 padding factor $pad$
 7:         if $AS_{new}[ref] = 0$ then
 8:             $BankConflict[ref] = 1$
 9:         else
10:             $BankConflict[ref] = \mathrm{GCD}(AS_{new}[ref], NB)$
11:         end if
12:         $AccessCost[ref] = C(BankConflict[ref])$
13:         $TotalAccessCost += AccessCost[ref]$
14:     end for
15:     if $TotalAccessCost < MinAccessCost$ then
16:         $OptPadding = pad$
17:         $MinAccessCost = TotalAccessCost$
18:     end if
19: end for
Output  Optimal padding factor for $A$ - $OptPadding$
```

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 4.6, we can obtain an accurate estimate of memory loads/-stores. However, because of the lack of control on 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 we employ an empirical search to pick the optimal code variant 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 loads/stores.

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

- For each valid program transformation structure obtained by the approach described in Section 4.2, 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 4.5).
- For each tiled loop structure, determine the register usage r and determine the maximum concurrency (L threads) possible within a multiprocessor. (There is an option in NVCC

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 equally among thread blocks and 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 4.6. 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.
- In all selected code versions, pad the arrays in shared memory with optimal padding factor determined using Algorithm 2.
- Search empirically among the selected code versions by explicitly running them and timing the execution time, and select the best one.

7. EXPERIMENTAL RESULTS

Experiments were conducted on an 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 processor cores (SIMD units) running at twice the clock frequency of the multiprocessor and has 16 KB of shared memory. CUDA version 1.0 was used for the experiments. The CUDA code was compiled using the NVIDIA CUDA Compiler (NVCC) to generate the device code that is launched from the CPU (host). The CPU was a 2.13 GHz Intel Core2 Duo processor with 2 MB L2 cache. The GPU device was connected to the CPU through a 16-x PCI Express bus. The host programs were compiled using the icc compiler at -O3 optimization level.

7.1 Performance Evaluation on Kernels

Figure 5 shows the performance of several kernels — Matrix Vector multiply (mv), Transposed Matrix Vector multiply (tmv), Matrix Vector Transpose (mvt), and Matrix Matrix multiply (mm) — that were optimized using the compile-time techniques developed in this paper. The comparison was done with the vendor-optimized CUBLAS library (version 1.0) supplied by NVIDIA. The effectiveness of the compile-time optimizations developed in this paper is evident from these results: for mv, tmv and mvt kernels, the performance achieved is better than that of the vendor-optimized CUBLAS implementation, and for mm kernel, the performance is close to that of CUBLAS implementation.

7.2 Optimized Global and Shared Memory Access

The mv and tmv kernels (Figure 3) are used to illustrate the benefits of global and shared memory access optimization. Table 2 shows the performance of mv kernel implemented using space and time partition mappings, as discussed in Section 4.3. Our approach makes a decision to copy the elements of array a in to shared memory though there is no reuse of the elements copied. An implementation with efficient copy of elements of array a from global memory to shared memory (column "Non-optimized Shared") provides an order of magnitude better performance than the version implemented with direct access of a from global memory (column "Direct Global"). The implementation with copy to shared memory is further enhanced by minimizing the shared memory bank conflicts through effective padding of the shared memory buffer created to hold the elements of array a. A further 2x improvement in performance is achieved (column "Optimized Shared") due to this shared memory access optimization.

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

Table 3 shows the performance of the tmv kernel implemented using the space and time partition mappings discussed in Section 4.3. 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 by our approach.

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

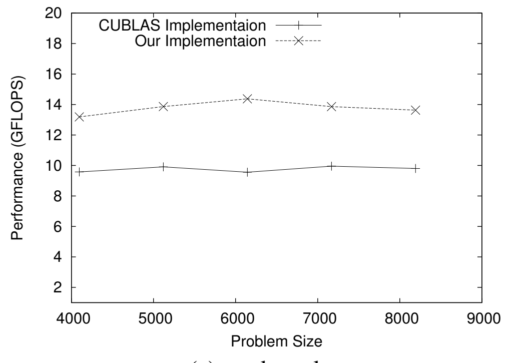
Matrix Vector Transpose (mvt) is a kernel that involves two matrix-vector multiplies, where one matrix is the transpose of the other, and hence it encompasses the computations involved in mv and tmv kernels. Our approach identifies the data space accessed by array reference involved in mv kernel computation as a candidate to be copied to shared memory and on the other hand identifies the data space accessed by array reference involved in tmv kernel computation as a candidate for direct global memory access, and results in optimized global memory access.

7.3 Model-driven Empirical Search on Matrix-Matrix Multiply (MM) kernel

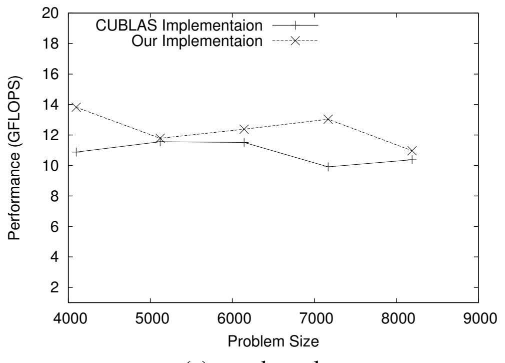
The MM kernel is used to illustrate the steps involved in the model-driven empirical search procedure explained in Section 6. A problem size of $4K \times 4K$ (that was barely able to fit in the GPU DRAM) was used.

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

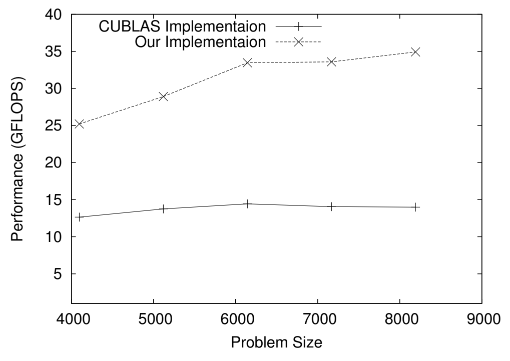
For various tile sizes that distribute the computation equally among thread blocks and among threads within a thread block, and satisfy the shared memory limit constraint, the total global memory loads varied from the order of $4K^3/2^7$ to $4K^3/2^4$. All code versions which had loads in the order of $4K^3/2^7$ to $4K^3/2^6$ were



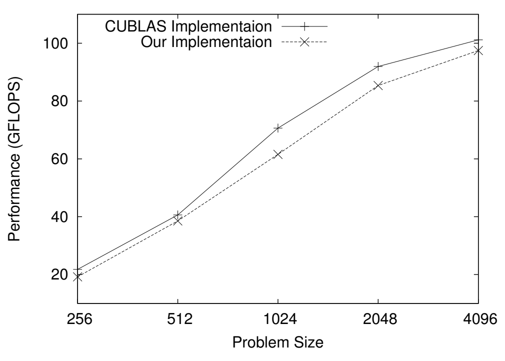
(a) mv kernel



(c) mvt kernel



(b) tmv kernel



(d) mm kernel

Figure 5: Performance of Matrix kernels

considered and various combinations of loop unrolling and register-level tiling were performed 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 determined, and those versions with excessive register usage (those which restricted the number of concurrent threads to below 128) were eliminated. Figure 6 illustrates the performance of the selected candidates that were run empirically to select the best one. The code version that was selected by the search procedure resulted in a performance of around 97 GFLOPS – compared to vendor-optimized MM kernel performance of around 101 GFLOPS.

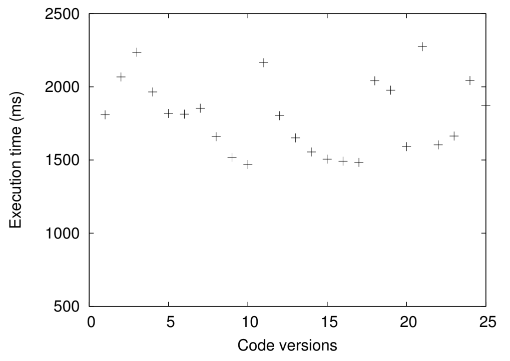


Figure 6: Performance of MM kernel for various code versions

8. RELATED WORK

Prior to the introduction of CUDA [18], GPU programming systems have relied on graphics API-based implementations, which have limited the size and kind of codes that are 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 [27] does not allow access to an individual element in parallel arrays and operations are performed on all array elements. Brook [6] 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 general purpose applications. Fatahalian et al. [8] mention that low-bandwidth cache designs on GPUs prevent general purpose applications from benefiting from the available computational power. Govindaraju et al. [12] 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.

Liao et al. [15] have developed a framework that works with Brook [6] to perform aggressive data and computation transformations. Recently, Ryoo et al. [25, 24] 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. [26] 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.

9. CONCLUSIONS

In this paper, critical performance-influencing factors on GPUs were characterized and techniques were developed to address the issues, that include 1) generation of effective program transformations for GPUs that enable efficient global memory access, 2) determination of optimal padding factors for conflict-minimal data access from shared memory, and 3) model-driven empirical optimization approach to optimize values for system and program parameters. The effectiveness of the developed techniques was demonstrated with various kernels.

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

10. REFERENCES

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

[2] 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.

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

[4] 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.

[5] U. Bondhugula, A. Hartono, J. Ramanujan, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Programming Languages Design and Implementation (PLDI '08)*, 2008.

[6] 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.

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

[8] 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.

[9] P. Feautrier. Dataflow analysis of array and scalar references. *IJPP*, 20(1):23–53, 1991.

[10] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I: one-dimensional time. *IJPP*, 21(5):313–348, 1992.

[11] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *IJPP*, 21(6):389–420, 1992.

[12] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *SC '06*, 2006.

[13] General-Purpose Computation Using Graphics Hardware. http://www.gpgpu.org/.

[14] M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. FMI, University of Passau, 2004. Habilitation Thesis.

[15] 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.

[16] A. Lim. *Improving Parallelism And Data Locality With Affine Partitioning*. PhD thesis, Stanford University, Aug. 2001.

[17] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL*, pages 201–214, 1997.

[18] NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html.

[19] NVIDIA GeForce 8800. http://www.nvidia.com/page/geforce_8800.html.

[20] PLuTo: A polyhedral automatic parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net.

[21] 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.

[22] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, Aug. 1992.

[23] F. Quilleré, S. V. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *IJPP*, 28(5):469–498, 2000.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.