Automatic C-to-CUDA Code Generation for Affine Programs

Muthu Manikandan Baskaran[1], J. Ramanujam[2], and P. Sadayappan[1]

[1] The Ohio State University, USA
[2] Louisiana State University, USA

Abstract. Graphics Processing Units (GPUs) offer tremendous computational power. CUDA (Compute Unified Device Architecture) provides a multi-threaded parallel programming model, facilitating high performance implementations of general-purpose computations. However, the explicitly managed memory hierarchy and multi-level parallel view make manual development of high-performance CUDA code rather complicated. Hence the automatic transformation of sequential input programs into efficient parallel CUDA programs is of considerable interest.

This paper describes an automatic code transformation system that generates parallel CUDA code from input sequential C code, for regular (affine) programs. Using and adapting publicly available tools that have made polyhedral compiler optimization practically effective, we develop a C-to-CUDA transformation system that generates two-level parallel CUDA code that is optimized for efficient data access. The performance of automatically generated code is compared with manually optimized CUDA code for a number of benchmarks. The performance of the automatically generated CUDA code is quite close to hand-optimized CUDA code and considerably better than the benchmarks' performance on a multicore CPU.

1 Introduction

Graphics Processing Units (GPUs) represent the most powerful multi-core systems currently in use. For example, the NVIDIA GeForce 8800 GTX GPU chip has a peak performance of over 350 GFLOPS and the NVIDIA GeForce GTX 280 chip has a peak performance of over 900 GFLOPS. There has been considerable recent interest in using GPUs for general purpose computing [8,13,12]. Until 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 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.

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 244–263, 2010.

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 [1,5,6,24,18,14,9,25,23,4]. For such regular programs, compile-time optimization approaches have been developed using affine scheduling functions with a polyhedral abstraction of programs and data dependencies. CLooG [4,7] is a powerful open-source state-of-the-art code generator that transforms a polyhedral representation of a program and affine scheduling constraints into concrete loop code. The Pluto source-to-source optimizer [5,6,22] enables end-to-end automatic parallelization and locality optimization of affine programs for general-purpose multi-core targets. The effectiveness of the transformation system has been demonstrated on a number of non-trivial application kernels for multi-core processors, and the system implementation is publicly available [22].

In this paper we describe an end-to-end automatic C-to-CUDA code generator using a polyhedral compiler transformation framework. We evaluate the quality of the generated code using several benchmarks, by comparing the performance of automatically generated CUDA code with hand-tuned CUDA code where available and also with optimized code generated by the Intel icc compiler for a general-purpose multi-core CPU.

The rest of the paper is organized as follows. Section 2 provides an overview of the polyhedral model for representing programs, dependences, and transformations. Section 3 provides an overview of the NVIDIA GPU architecture and the CUDA programming model. The design and implementation of the C-to-CUDA transformer is presented in Section 4. Experimental results are provided in Section 5. We discuss related work in Section 6 and conclude with a summary in Section 7.

2 Background

This section provides background information on the polyhedral model. A hyperplane in n dimensions is an $n-1$ dimensional affine subspace of the n-dimensional space and can be represented by an affine equality. A halfspace consists of all points of an n-dimensional space that lie on one side of a hyperplane (including the hyperplane); it can be represented by an affine inequality. A polyhedron is the intersection of finitely many halfspaces. A polytope is a bounded polyhedron.

In the polyhedral model, 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 (referred to as the iteration vector i_s) correspond to the values of the loop indices of the surrounding loops, starting from the outermost. In this work we focus on programs 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. Hence the iteration space polytope $\mathcal{D}_s$ of a statement s can be defined by a system of affine inequalities derived from the bounds of the loops surrounding s. Each point of the polytope corresponds to an instance of statement s in program execution. Using matrix representation to express systems of affine inequalities, the iteration space polytope is defined by $D_s \begin{pmatrix} i_s \\ n \\ 1 \end{pmatrix} \geq 0$, where D_s is a matrix representing loop bound constraints and n is a vector of global parameters.

Affine array access functions can also be represented using matrices. Let $a[\mathcal{F}_{ras}(i_s)]$ be the r^{th} reference to an array a in statement s whose corresponding iteration vector is i_s. Then $\mathcal{F}_{ras}(i_s) = F_{ras}\begin{pmatrix} i_s \\ n \\ 1 \end{pmatrix}$, where F_{ras} is a matrix representing an affine mapping from the iteration space of statement s to the data space of array a. Row i in the matrix F_{ras} (often referred to as the access matrix) defines a mapping corresponding to the ith 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 that reference.

Given an iteration space polytope $\mathcal{D}$ 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 or the *accessed data space* is given by $\mathcal{DS} = \bigcup_{j=1}^{k} \mathcal{F}_j\mathcal{D}$, where $\mathcal{F}_j\mathcal{D}$ is the image of the iteration space polytope $\mathcal{D}$ formed by the affine access function $\mathcal{F}_j$ and it gives the set of elements accessed by the reference $\mathcal{F}_j$ in $\mathcal{D}$.

Dependences. There has been a significant body of work on dependence analysis in the polyhedral model [9,24,29]. An instance of statement s, corresponding to iteration vector i_s within iteration domain D_s, depends on an instance of statement t (with iteration vector i_t in domain D_t), if (1) i_s and i_t are valid points in the corresponding iteration space polytopes, (2) they access the same memory location, and (3) i_s is executed before i_t. Since array accesses are assumed to be affine functions of loop indices and global parameters, the constraint that defines conflicting accesses of memory locations can be represented by an affine equality (obtained by equating the array access functions in source and target statement instances). Hence all constraints to capture a data dependence can be represented as a system of affine inequalities/equalities with a corresponding polytope (referred to as a *dependence polytope*).

Affine Transforms. The polyhedral model has been effectively used to find good affine program transformations that are aimed at either improvement of sequential programs (source-to-source transformation) or automatic parallelization of programs or both [10,18,14,11,14,23,6].

A one-dimensional affine transformation of a statement s is represented in the polyhedral model as $\phi_s(i_s) = \mathcal{C}_s.\begin{pmatrix} i_s \\ n \\ 1 \end{pmatrix}$, where $\mathcal{C}_s$ is a row vector and the affine mapping ϕ_s represents an affine hyperplane that maps each instance of statement s to a point in a dimension of the transformed iteration space. An affine transformation is valid only if it preserves the dependences in the original program. An m-dimensional affine mapping can be represented using a matrix with m rows, where each row represents a one-dimensional mapping. A set of linearly independent one-dimensional affine functions $(\phi_s^1, \phi_s^2, \ldots, \phi_s^k)$ maps each instance of statement s into a point in the multi-dimensional transformed space. The transformation matrix captures a composition of transformations like fusion, skewing, reversal and shifting.

It has been shown (in automatic transformation systems like Pluto) that key compiler transformations like tiling can be effectively performed using the polyhedral model. When tiling is performed, in the tiled iteration space, statement instances are represented by higher dimensional statement polytopes involving *supernode* or *inter-tile*

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

(a) Original code

$$D_Q^{orig}.\begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq 0 \qquad D_Q^{tiled}.\begin{pmatrix} it \\ jt \\ i \\ j \\ N \\ 1 \end{pmatrix} \geq 0$$

(c) Original and tiled iteration space

```
for (it=0;it<=floord(N-1,32);it++) {
  for (jt=0;jt<=floord(N-1,32);jt++) {
    if (jt == 0) {
      for (i=max(32*it,0);
      i<=min(32*it+31,N-1); i++) {
        P: x[i]=0;
        Q: x[i]=x[i]+a[0][i]*y[0];
      }
    }
    for (i=max(32*it,0);
     i<=min(32*it+31,N-1); i++) {
      for (j=max(32*jt,1);
       j<=min(32*jt+31,N-1);j++) {
        Q: x[i]=x[i]+a[j][i]*y[j];
      }
    }
  }
}
```

(b) Tiled code

Fig. 1. Example to illustrate Tiling: Transpose matrix vector multiply (tmv) kernel

iterators and *intra-tile* iterators. The code in Figure 1(b) represents the tiled version of the code in Figure 1(a). The original iteration space and the transformed iteration space are illustrated in Figure 1(c).

3 GPU Architecture and the CUDA Programming Model

In this Section, we provide an overview of the GPU parallel computing architecture, the CUDA programming interface, and the GPU execution model.

3.1 GPU Computing Architecture

NVIDIA GPUs comprises of a set of multiprocessor units called *streaming multiprocessors (SMs)*, each one containing a set of processor cores (called *streaming processors* (SPs)). The NVIDIA GeForce 8800 GTX has 16 SMs, each consisting of 8 SPs. The NVIDIA GeForce GTX280 has 30 SMs with 8 SPs in each SM. The SPs within an SM communicate through a fast explicitly managed on-chip local store, also called the *shared memory*, while the different SMs communicate through slower off-chip DRAM, also called the *global memory*. Each SM unit also has a fixed number of *registers*.

Different types of memory in the GPUs are addressable in CUDA programming model. The memories are organized in a hybrid cache and local-store hierarchy. The memories 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.

The off-chip DRAM in the GPU device (i.e., the global memory) has a very high latency (about $100-200$ cycles). Hence reducing the latency in accessing data from global

memory is critical for good performance. The global memory accesses in NVIDIA GPU chips are characterized by 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 and is therefore an important optimization for a C-to-CUDA compiler.

The shared memory in each SM is organized into banks. When multiple addresses belonging to the same bank are accessed at the same time, bank conflict occur. Each SM has a set of registers. The constant and texture memories are read-only regions in the global memory space and they have on-chip read-only caches. Accessing constant cache is faster, but it has only a single port and hence it is beneficial when multiple processor cores load the same value from the cache. Texture cache has higher latency than constant cache, but it does not suffer greatly when memory read accesses are irregular and it is also beneficial for accessing data with 2D spatial locality. It is extremely important to reduce the number of accesses to off-chip memory and maximize utilization of the on-chip memories.

3.2 CUDA Programming Model

Programming GPUs for general-purpose applications is enabled through a C/C++ language interface exposed by the NVIDIA Compute Unified Device Architecture (CUDA) technology [20]. The CUDA programming model provides an abstraction of the GPU parallel architecture using a minimal set of programming constructs such as hierarchy of threads, hierarchy of memories, and synchronization primitives. A CUDA program comprises of a host program which is run on the CPU or host and a set of CUDA kernels that are launched from the host program on the GPU device. The CUDA kernel is a parallel kernel that is executed on a set of threads. The threads are organized into groups called *thread blocks*. The threads within a thread block synchronize among themselves through barrier synchronization primitives in CUDA and they communicate through shared memory. A kernel comprises of a *grid* of one or more thread blocks. Each thread in a thread block is uniquely identified by its thread id (threadIdx) within its block and each thread block is uniquely identified by its block id (blockIdx). The dimensions of the thread and thread block are specified at the time of launching the kernel, through the identifiers *blockDim* and *gridDim*, respectively.

Each CUDA thread has access to the different memories at different levels in the hierarchy. The threads have a private local memory space and register space. The threads in a thread block share a shared memory space. The GPU DRAM is accessible by all threads in a kernel.

3.3 GPU Execution Model

NVIDIA GPUs use a Single Instruction Multiple Threads (SIMT) model of execution. The threads in a kernel are executed in groups called *warps*, where a warp is a unit of execution. The scalar SPs within an SM share a single instruction unit and the threads of a warp are executed on the SPs. All the threads of a warp execute the same instruction and each warp has its own program counter. The SM hardware employs zero-overhead

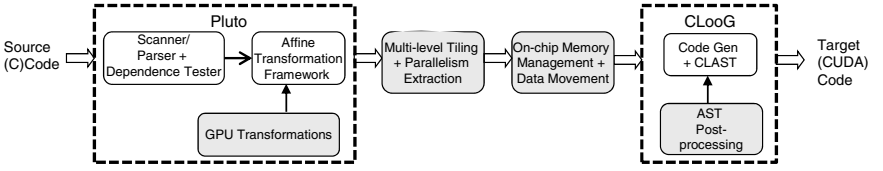

Fig. 2. The C-to-CUDA Code Generation Framework

warp scheduling through the CUDA runtime scheduler. Any warps whose next instruction has ready operands is eligible for execution. Eligible warps are selected for execution by a prioritized scheduling policy. The warp scheduling is completely transparent to the CUDA programmer.

The computational resources on a multiprocessor unit, i.e., the shared memory and the register bank, are shared among the active thread blocks on that unit. For example, an application abstracted as a grid of 64 thread blocks can have 4 thread blocks mapped on each of the 16 multiprocessors of the NVIDIA GeForce 8800 GTX. The GeForce 8800 GTX GPU has a 16 KB shared memory space and 8192 registers. If the shared memory usage per thread block is 8 KB and the register usage is 4096, at most 2 thread blocks can be concurrently active on a multiprocessor; when one of the two thread blocks completes execution, another thread block can become active on the multiprocessor.

4 Design of C-to-CUDA Generator

In this section, we describe the C-to-CUDA code generator. Before providing details on the various transformation aspects, we first outline the general steps involved in source-to-source code generation using a polyhedral compiler framework.

1. The input program is run through a scanner and parser that constructs an abstract syntax tree (AST) for the input program. From the AST, iteration space polytopes and array access functions are extracted.
2. Data dependences are analyzed and dependence polytopes (described in Section 2) are generated.
3. After analyzing the dependences, affine statement-wise transforms are determined. The affine transforms provide the new lexicographic ordering of the statements in the transformed program.
4. When tiling has to be performed, the affine statement-wise transforms are used as tiling hyperplanes to generate higher-dimensional statements domains (involving supernode iterators and intra-tile iterators).
5. The transformed statement polytopes along with the affine transformations are provided to a polyhedral code generator such as CLooG to generate transformed code.

As described in Section 3, the GPU architecture represents a multi-level parallel architecture. It has various memory units (with different access properties) that are at different proximity with respect to the chip (on-chip and off-chip) and have very different access latencies. We now discuss the various issues that are addressed by our

code generation system for generating effective CUDA code along the lines of the code generation process described above. There are several publicly available polyhedral transformation frameworks and tools. We used the Pluto [22] polyhedral parallel tiling infrastructure and CLooG [4,7], a state-of-the-art polyhedral code generator. The sequence of steps in the implemented system is shown in Fig. 2.

1. One of the key optimizations is to generate efficient access pattern for global (off-chip) memory access. Pluto finds affine transforms that are (1) communication-optimized, and (2) locality-optimized. At Step 3 of the code generation process (outlined above), our framework finds affine transforms that enable global memory coalescing in addition to being communication-optimized and locality-optimized. (detailed in [2]).
2. Two levels of parallelism must be extracted to exploit parallelism at the thread block level and the thread level for GPUs. At Step 4, we use the affine transforms determined at Step 3 to find multi-level tiled statement domains and identify and extract parallelism.
3. A critical optimization for GPUs is the utilization of on-chip memories. It is beneficial to move repeatedly reused data from off-chip memory to on-chip memory before the first use and move it back after the last use. At Step 4, our framework generates iteration space polytopes of data movement statements using polyhedral techniques, in addition to generating the transformed statement domains. (detailed in [3]).
4. At Step 5, we use the CLooG polyhedral code generator to generate the target code structure. Suitable input, in the form of a description of all statements (computation and data movement), together with their iteration spaces (as polytopes) as well as the transformations (as scheduling functions) specifying the new execution order for each statement instance, is input to the CLooG code generator. The union of all input iteration space polytopes is scanned by CLooG according to the specified scheduling functions, in order to generate loop nests in the target program that execute the statement instances in this new execution order.
5. After Step 5, the AST of the generated parallel tiled code is post processed to generate compilable CUDA code. The post processing is primarily (1) to introduce thread-centricity in the parallel code, i.e., to add thread identifier and thread block identifier, and (2) add inter-thread and inter-thread-block synchronizations at appropriate execution points.

In the rest of this section, we provide details on the following three aspects of the C-to-CUDA generator:

1. generation of multi-level tiled parallel code,
2. generation and placement of code to move data between on-chip and off-chip memories, and
3. generation of thread-centric parallel code.

4.1 Multi-level Parallel Tiled Code Generation

Tiling Hyperplanes and Tiling Legality Condition. In order to generate tiled code, Pluto finds affine transforms that satisfy the following tiling legality condition [6] in a

multi-statement imperfectly nested program and use them as tiling hyperplanes which constitute the loops in the transformed program:
A set of one-dimensional affine transformation functions (one corresponding to each statement in a imperfectly nested multi-statement program), $\{\phi_{s_1}, \phi_{s_2}, \ldots, \phi_{s_n}\}$, represents a valid tiling hyperplane if for each pair of dependent statement instances (i_{s_p}, i_{s_q}) $\phi_{s_q}(i_{s_q}) - \phi_{s_p}(i_{s_p}) \geq 0$. This condition guarantees that any inter- or intra-statement affine dependence is carried in the forward direction along the tiling hyperplane. Hence if a program is transformed using the affine transforms satisfying the above condition, then rectangular tiling is legal in the transformed program.

Affine transformations for CUDA. With CUDA, execution of a program involves distributing the computation across thread blocks and across threads within a thread block. For tiling at the outer level (at the level of thread blocks), our framework uses the affine transforms generated by Pluto. For finding tiling hyperplanes to generate tiled code at the inner level (at the level of threads), we modify Pluto to generate program transformations that enable interleaved access to global memory by threads in a thread block - this is necessary to facilitate coalesced global memory accesses that improve global memory access bandwidth. An approach to achieve this was developed in [2]. We incorporated that approach in our system by framing additional constraints to feed to Pluto while finding affine statement-wise transforms. The additional constraints are:

- If two statement instances access adjacent elements of an array (based on the actual array layout), then the statement instances are scheduled to execute at the same time; (and)
- If two statement instances access adjacent elements of an array (based on the actual array layout), then the statement instances are scheduled to execute on adjacent processors.

Extracting Parallel Loops. The affine transformations may or may not result in synchronization-free parallel tile loops (*doall* loops). If doall loops exist in the tile space, they are used as parallel loops. However when no synchronization-free parallelism exists, parallel code generation needs additional processing. There may be one or more loops that carry dependences (*doacross* loops). Since the tiling legality condition assures that the dependences are always carried in the forward direction, pipelined parallelism with synchronization can be exploited in such cases.

If $\{\phi^1, \phi^2, \ldots, \phi^n\}$ represent the doacross loops in the tile space, then the sum $\phi^1 + \phi^2 + \cdots + \phi^n$ carries all dependences that are carried by each $\phi^i, 1 \leq i \leq n$, and represents a legal wavefront of tiles such that all tiles in the wavefront are parallel [15]. In other words, the set of loops are transformed (using a unimodular skewing transformation) as follows:

$$\begin{pmatrix} \phi'^1 \\ \phi'^2 \\ \phi'^3 \\ \vdots \\ \phi'^n \end{pmatrix} = \begin{pmatrix} 1\ 1 \ldots 1\ 1 \\ 1\ 0 \ldots 0\ 0 \\ 0\ 1 \ldots 0\ 0 \\ \vdots\ \vdots\ \ddots\ \vdots\ \vdots \\ 0\ 0 \ldots 1\ 0 \end{pmatrix} \begin{pmatrix} \phi^1 \\ \phi^2 \\ \phi^3 \\ \vdots \\ \phi^n \end{pmatrix}.$$

This ensures that ϕ'^1 is sequential and $\phi'^2, \phi'^3, \ldots, \phi'^n$ represent the parallel loops. This is the approach we employ to extract parallel loops at one level. A synchronization call to

synchronize across the parallel units has to be placed at each iteration of the sequential loop. Handling the placement of synchronization calls is discussed later in the Section.

Pluto generates parallel code for general purpose multi-core architectures; it generates multi-level tiled code with parallelism only at the outer level. However for multi-level parallel architectures like GPGPUs, parallelism has to be extracted at multiple levels (two levels for CUDA - thread block level and thread level). Algorithm 1 provides details on the approach to generate multi-level tiled transformed statement domains (which are later fed to CLooG for code generation) along with the identification of parallel loops at thread block level and thread level.

Using CLooG for Multi-level Tiled Code Generation. As described earlier, CLooG scans a union of statement (iteration space) polyhedra using an optionally provided global lexicographic ordering specified through statement-wise scheduling functions or *scattering functions*, and generates loop nests in the target program that execute the statement instances in the new lexicographic order. CLooG does not include any data dependence information and hence the legality of scanning the statement polyhedra should be guaranteed by the user specifying the scattering functions. In our framework, the statement-wise affine transforms provided as scattering functions to CLooG ensure effective and correct execution of the transformed program. Tiled code is generated using CLooG by specifying a modified higher dimensional statement domain for each statement and also specifying the scheduling or scattering functions (using the affine statement-wise transforms) to generate the correct ordering of inter-tile and intra-tile loops.

Algorithm 1. Multi-level Parallel Tiled Code Generation

Input Set of statements - S, Iteration Space Polytopes of all statements $\mathcal{D}_s, s \in S$, Statement-wise affine transforms for each level k: $\phi k_s^1, \phi k_s^2, \ldots, \phi k_s^n, s \in S$, Tile sizes $t_1, t_2, \ldots, t_n$ for each level

1. **for** each level **do**
2. **for** each statement $s \in S$ **do**
3. **for** each transform $\phi_s = C_s(i_s)$ **do**
4. Increase the statement domain's dimensionality so that the domain includes the supernode iterators
5. Add constraints involving supernode iterators (ϕT_s) and tile sizes that represent a statement instance in a supernode $t \times \phi T_s \leq C_s(i_s) \leq t \times \phi T_s + t - 1$
6. **end for**
7. Add scattering functions corresponding to supernodes. (The scattering functions are identity functions involving the supernode iterators)
8. **if** level to be parallelized **then**
9. **if** there exists *doall* loops **then**
10. Mark them as parallel
11. **else**
12. Transform the first non-sequential loop ϕ^i as follows: $\phi^i \leftarrow \phi^i + \phi^{i+1} + \cdots + \phi^n$
13. Mark ϕ^i as sequential and remaining subsequent loops in the band as parallel
14. **end if**
15. **end if**
16. **end for**
17. **end for**

Output Transformed computation statement domains and scattering functions

4.2 Data Movement between Off-Chip and On-Chip Memories

As discussed in Section 3, it is very important to reduce the accesses to off-chip memory and utilize the on-chip memories. Array references that have sufficient data reuse are good candidates to be copied to shared memory since the repeated accesses would be made in low-latency on-chip memory instead of off-chip memory. Array references, for which there exists no suitable affine scheduling that supports coalesced memory access, are also treated as candidates to be copied to shared memory. This is because of the fact that non-coalesced accesses incur very high memory access cost.

Given a program block or tile (having one or more statements), the data spaces accessed by array references within the block are determined using the iteration space of each statement and the array access function of each reference in each statement (as mentioned in Section 2). The data spaces accessed by the read and write references of each array are represented as separate polytopes and are then used to determine the size of storage buffer needed to host the required data. The code for data movement is then generated by scanning the data space polytopes using CLooG. 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. The data movement statements are of two types: (1) those that move data in to shared memory (further referred to as copy-in statements) and (2) those that move data out of shared memory (further referred to as copy-out statements).

The target code should encompass the data movement statements and computation statements in proper order so that the parallel code results in correct program execution. At the level of thread blocks, the data movement statements are placed such that they respect the following order: *copy-in, computation, copy-out*. We utilize the scattering functions in CLooG to achieve the proper placement of data movement and computation statements. The scattering functions provide a multi-level multi-dimensional schedule. The basic idea is to introduce an additional 'constant' dimension in the original schedule at the level of thread blocks to define the order of statements. Suppose that in the transformed program, the computation and data movement statements are defined at the outer level by a schedule using the iterators $(c_1, c_2, \ldots, c_n)$. We modify the schedule of the copy-in, computation, copy-out statements as $(c_1, c_2, \ldots, c_n, 0)$, $(c_1, c_2, \ldots, c_n, 1)$, and $(c_1, c_2, \ldots, c_n, 2)$, respectively, to achieve the required order.

The algorithm to generate data movement statement domains and scattering functions to properly place data movement code in the target CUDA code structure is outlined in Algorithm 2.

Exploiting constant memory and registers. In addition to handling data movement to the on-chip shared memory, we handle on-chip constant memory and registers. 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. If threads in a warp access different values in constant memory, the requests get 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 used for distributing computation across threads. Such arrays

Algorithm 2. Generation and Placement of Data Movement Code

Input Set of statements - $\mathcal{S}$, Transformed Statement Domains of all statements $\mathcal{D}_s, s \in S$ from Algorithm 1, Affine array access functions

1. **for** each array A **do**
2. **for** all references of the array **do**
3. Find the data space accessed by the references
4. **end for**
5. Partition the set of all data spaces into maximal disjoint sets such that each partition has a subset of data spaces each of which is non-overlapping with any data space in other partitions
6. For each partition, find the convex union of its data spaces and the bounding box of the convex union gives the storage buffer needed for the partition
7. **for** each statement $s \in S$ **do**
8. **for** all read references of the array **do**
9. Find the data space accessed by the references and use them as domains of copy-in statements
10. Use "identity" scattering functions
11. **end for**
12. **for** all write references of the array **do**
13. Find the data space accessed by the references and use them as domains of copy-out statements
14. Use "identity" scattering functions
15. **end for**
16. **end for**
17. **end for**
18. Let the number of copy-in and copy-out statements be c and d, respectively
19. Add a new dimension in all scattering functions (those of copy-in, computation, and copy-out statements) with just a constant value; the constant being 0 to $c-1$ for copy-in statements, c for computation statements, $c+1$ to $c+d$ for copy-out statements

Output Data movement statement domains and updated scattering functions

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

4.3 Syntactic Post-processing

The transformed multi-level tiled computation statement domains and data movement statement domains along with the scattering functions (generated by Algorithms 1 and 2) are fed to CLooG to generate multi-level tiled code. Syntactic post processing of the multi-level tiled code generated by CLooG is needed to generate a final compilable CUDA code. The primary tasks of the post processing are (1) to generate thread-centric code and (2) to place synchronization calls for correct parallel execution.

An important aspect of CUDA code generation is thread-centric code generation, i.e. generation of code where the computation is distributed across the threads in the system. A thread in the system is uniquely identified by a combination of its "thread block identifier" and "thread identifier" within the thread block. We take a syntactic approach to introduce thread-centricity in the parallel code generated using the above technique. The CLooG tool has its own AST representation called the CLAST. The

CLAST generated for the parallel tiled code is parsed to introduce "thread block and thread identifiers" in the parallel loops (identified in Algorithm 1) such that the parallel tiles at the outer level are cyclically distributed across the *thread blocks* and that at the inner level are cyclically distributed across the *threads*. The data movement code is also parsed to place "thread identifier" in the data movement loops.

CUDA offers a synchronization primitive to synchronize across threads within a thread block, but no built-in synchronization primitives to synchronize across thread blocks. We introduce a primitive through a code segment that uses a "single-writer multiple-reader" technique to achieve synchronization across thread blocks using the global memory space. It is necessary to place barrier synchronizations at each iteration of a sequential loop (if any) that precedes parallel loops, and at the end of data movement loops. It is done syntactically by modifying the CLAST. Algorithm 3 summarizes the CUDA code generation steps after applying Algorithms 1 and 2.

It should be noted that the tile sizes used for tiling are fixed at compile time and provided by the user. The code generated by our framework represents the number of threads and thread blocks as symbolic constants, which the user sets before the actual execution. Our framework also syntactically inserts an "unroll" pragma - *#pragma unroll unroll_factor* - which enables the CUDA compiler to perform inner loop unrolling.

Algorithm 3. Parallel CUDA Code Generation

Input Computation statement domains, Data movement statement domains, Scattering functions

1. Feed the computation and data movement statement domains and scattering functions to CLooG to generate CLAST
2. Parse CLAST to change the lower bounds and loop increments of (outer and inner level) parallel loops to make them thread-centric
3. Parse CLAST to change the lower bounds and loop increments of data movement loops to make them thread-centric
4. Place barrier synchronization at each iteration of sequential loop (if any) that precedes parallel loops, and at the end of data movement loops
5. Print the modified CLAST to generate CUDA code

Output Multi-level parallel tiled CUDA code with data movement

5 Experimental Results

In this section, we present experimental results to assess the effectiveness of the CUDA code generated by the implemented C-to-CUDA transformation system. We present results on seven benchmarks. Where available, we compare the performance of the automatically generated CUDA code with hand-tuned CUDA code. We also compare the performance of the generated CUDA code on the GPU with the performance of input C code (optimized by the Intel icc compiler), on a multi-core CPU.

The GPU device used in our experiments was an NVIDIA GeForce 8800 GTX GPU. 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. 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

```
for (t1=0; t1<VOLY; t1++) {
   for (t2=0; t2<VOLX; t2++) {
       for (t3=0;t3<NATOMS;t3++) {
           energy[zDim*VOLX*VOLY + t1*VOLX + t2] =
           atoms[3+4*t3]/ ... atoms[2+4*t3] ...
           atoms[1+4*t3] ... atoms[4*t3];
       }
   }
}
```

Fig. 3. Original code structure for Coulombic Potential (cp) benchmark

processor with 2 MB L2 cache. The GPU device was connected to the CPU through a 16-x PCI Express bus. We used CUDA version 2.1 for our experiments.

The multi-core system used for our experiments was a quad-core Intel Core 2 Quad Q6600 CPU clocked at 2.4 GHz (1066 MHz FSB) with a 32 KB L1 D cache, 8MB of L2 cache (4MB shared per core pair), and 2 GB of DDR2-667 RAM, running Linux kernel version 2.6.22 (x86-64). ICC 10.x was the primary compiler used to compile the code on the multi-core system; it was run with -fast -funroll-loops (-openmp for parallelized code); the -fast option turns on -O3, -ipo, -static, -no-prec-div on x86-64 processors; these options also enable auto-vectorization in icc.

5.1 Coulombic Potential (cp)

This benchmark is used for the computation of electric potential in a volume containing point charges. It is one of the codes in the *parboil* benchmark suite from UIUC [21]. Fig. 4 presents the performance data - performance of the generated CUDA code with different optimizations is compared with the hand-tuned code from the parboil benchmark suite and icc optimized C code. The CUDA code generated by our framework performs better than the optimized version on general-purpose multi-core system. The

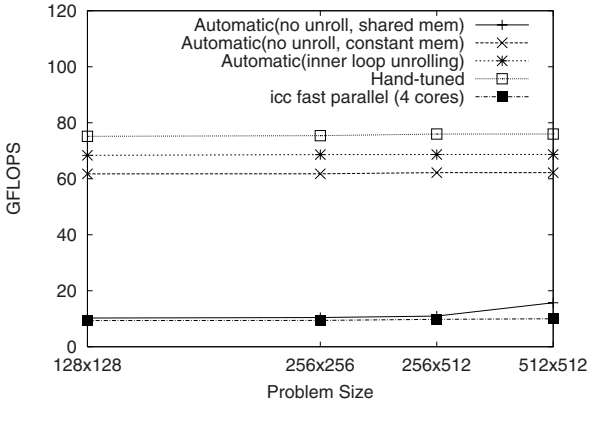

Fig. 4. Performance of cp benchmark

performance of the code generated by turning on all optimizations is very close to that of the hand-tuned code. In addition to extracting "doall' parallelism across threads and thread blocks, the code has optimized off-chip access in one of the two ways - (1) utilizing shared memory or (2) utilizing constant memory. Fig. 4 shows the performance measurements for both the cases and it can be seen that the performance when constant memory is used is significantly higher than that when shared memory is used. This is because the use of constant memory significantly reduces global memory traffic in comparison to accessing data after moving from global memory to shared memory. Inner loop unrolling was performed using NVIDIA's *#pragma unroll* option.

Figures 3 and 9 illustrate the CUDA code generation. Fig. 3 shows the structure of sequential code (along with the array accesses) for Coulombic Potential (cp) benchmark. Fig. 9 shows the structure of two-level tiled parallel code that is thread-centric where the parallelism is across thread blocks at the outer level and across threads at the inner level (Note the modified lower bounds and loop increments of parallel loops). Fig. 9 also shows the proper placement of data movement and computation statements.

5.2 N-Body Simulation (nbody)

N-body simulation is an important computation that arises in many computational science applications. It approximates the evolution of a system of bodies in which each

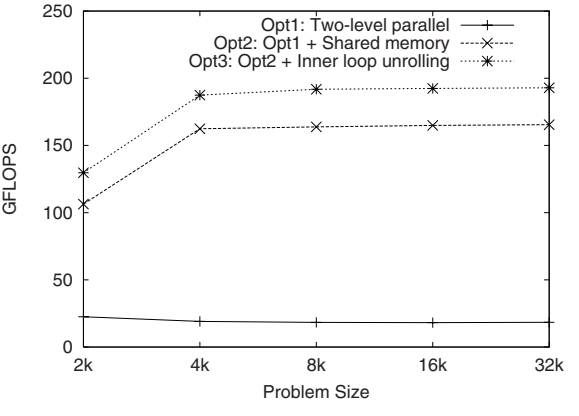

Fig. 5. Performance variation of nbody benchmark w.r.t Optimizations

Table 1. Performance of nbody benchmark (in GFLOPS)

N	Auto-CUDA	Hand-tuned	icc
2048	129.67	157.34	1.00
4096	187.41	182.31	1.10
8192	191.81	188.78	1.42
16384	192.45	198.43	1.47
32768	192.91	200.35	1.50

body continuously interacts with every other body. The CUDA code generated by our framework performs much better than the optimized version on general-purpose multi-core system and performs very comparably to the hand-tuned CUDA code, as illustrated in Table 1. The code generated by our framework exploited "doall" parallelism across threads and thread blocks. It effectively moved data from arrays that exhibited data reuse from global memory to shared memory, thereby enabling coalesced global memory access and also reduction in off-chip memory access latency, by exploiting data reuse in on-chip shared memory. Further, inner loop unrolling was performed using NVIDIA's *#pragma unroll* option. Fig. 5 depicts incremental performance improvement when different optimizations are applied. The importance of shared memory utilization and inner loop unrolling (to reduce loop overhead and dynamic loop instruction count) are illustrated by this benchmark.

Table 2. Performance of MRI-Q (in GFLOPS)

N	Auto CUDA (2)		Auto CUDA (1)		Hand	icc
	no unroll	unroll	no unroll	unroll	tuned	
32768	87.11	122.19	137.1	176.50	178.98	0.91
65536	88.27	121.87	141.7	179.32	179.12	1.14
131072	88.53	123.11	142.3	181.23	179.32	1.14
262144	89.16	122.12	142.6	183.32	180.91	1.15

Table 3. Performance of MRI-FHD (in GFLOPS)

N	Auto CUDA (2)		Auto CUDA (1)		Hand	icc
	no unroll	unroll	no unroll	unroll	tuned	
32768	57.91	90.91	112.52	142.52	143.11	1.37
65536	61.27	91.2	116.12	143.15	142.27	1.68
131072	62.13	91.6	116.22	144.21	144.39	2.19
262144	62.67	91.52	116.67	142.61	144.43	2.21

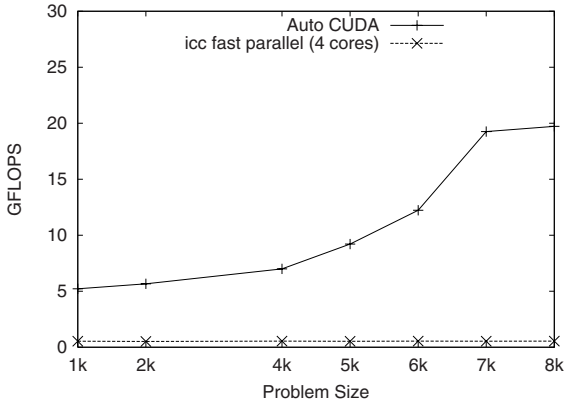

Fig. 6. Performance of 2D Jacobi

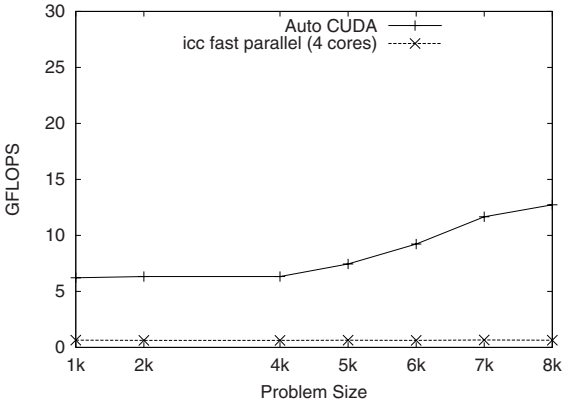

Fig. 7. Performance of 2D FDTD

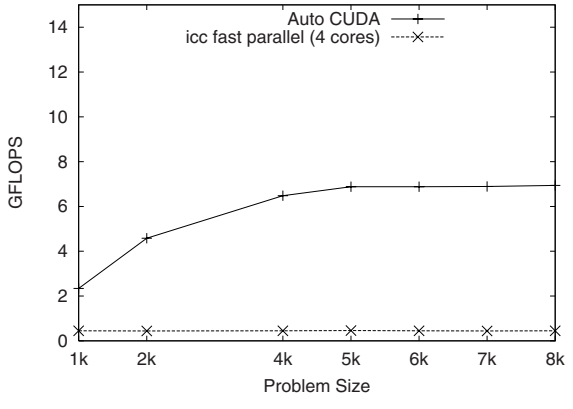

Fig. 8. Performance of Gauss Seidel

The hand-tuned version was taken from the NVIDIA CUDA SDK, the code being based on the article in [16]. The code generated by our framework represents the number of threads and thread blocks as symbolic constants, which the user sets before the actual execution.

5.3 MRI Kernels

We employed our framework to generate code for two kernels used in Magnetic Resonance Imaging, MRI-Q and MRI-FHD [21]. Both the kernels involve two computational blocks such that data computed in the first computation block is used as "read-only" data in the second computational block. The hand-tuned code from parboil optimizes the two computational blocks independently and executes them as separate GPU kernels. We used our framework to generate two versions of code for each of the two MRI kernels - version (1) in which CUDA code is generated independently for the

two computational blocks (first block pre-computes data for second block) and version (2) in which unified CUDA code is generated for both blocks.

Tables 2 and 3 summarize the performance measures of the code versions of MRI-Q and MRI-FHD, respectively. The code version (1) generated as two separate GPU kernels outperforms the code version (2) generated as single GPU kernel because of the fact that in version (1) the data precomputed in the first GPU kernel is stored in constant memory and accessed in the second kernel. However both the versions identified various data arrays as candidates for constant memory and thereby optimized off-chip memory

```
int by = blockIdx.y;
int bx = blockIdx.x;
int ty = threadIdx.y;
int tx = threadIdx.x;

int t1,t2,t3,t4,t5,t6;
// Parallel loops distributed across thread blocks
// Loops modified syntactically for thread block identifiers
for (t1=by; t1<=floord(VOLY-1,16); t1+=NBLKSY) {
  for (t2=bx; t2<=floord(VOLX-1,16); t2+=NBLKSX) {
    for (t3=0;t3<=NATOMS-1;t3+=256) {
      // Data movement code
      __shared__ float atomsS[1024];
      for (t6=4*t3+THREADY*NTHRDSX+THREADX;
           t6<=min(4*NATOMS-1,4*t3+1023);
           t6+=NTHRDSX*NTHRDSY)
         atomsS[t6-4*t3] = atoms[t6];
       __syncthreads();
      // Parallel loops distributed across threads
      // Loops modified syntactically for thread identifiers
      for (t4=max(0,16*t1)+ty;
           t4<=min(VOLY-1,16*t1+15);t4+=NTHRDSY) {
        for (t5=max(0,16*t2)+tx;
             t5<=min(VOLX-1,16*t2+15);t5+=NTHRDSX) {
               ...

          // Computation code
          for (t6=t3; t6<=min(NATOMS-1,t3+255);
               t6++) {
            energy[zDim*VOLX*VOLY + t4*VOLX + t5] =
            atomsS[3+4*t6-4*t3]/ ... atomsS[2+4*t6-4*t3] ...
            atomsS[1+4*t6-4*t3] ... atomsS[4*t6-4*t3];
          }
        }
      }
    }
  }
}
```

Fig. 9. Parallel tiled code structure (with data movement) for cp benchmark

access. The code version (1) generated by our framework performs as well as the hand-tuned version.

5.4 Stencil Computation Kernels

We used two stencil computation kernels, 2D Jacobi and 2D Finite Difference Time Domain (FDTD). The code generated using our framework performs better than the optimized version on the Intel multi-core system, as illustrated in Figures 6 and 7. For these two kernels, we were unable to find any hand-tuned CUDA code to compare against. The code generated by our framework exploits parallelism across threads and thread blocks and effectively utilizes shared memory and exploits data reuse. The parallel execution of stencil computations is characterized by synchronization overhead at every time step across the processors. This overhead is particularly costly in GPUs where the thread blocks have to synchronize using the slow off-chip memory. This is the reason for the lower absolute performance of these kernels on GPUs, relative to the previous benchmarks. The performance of the stencil kernels is very low for smaller problem sizes for the same reason.

5.5 Gauss Seidel Successive over Relaxation

The Gauss Seidel benchmark illustrates the effect of exploiting wavefront or pipelined parallelism on GPUs. We achieve better performance than the optimized version on multi-core system, as illustrated in Fig. 8. However, the absolute performance is rather low because of (1) low processor utilization during the starting and draining of pipeline and (2) synchronization overhead across thread blocks at every time step.

6 Related Work

In this Section, we review prior work on optimizations and code generation for GPUs.

Ryoo et al. [27,26] presented experimental studies of 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. [28] presented performance metrics such as *efficiency* and *utilization* 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. The end-to-end system described in this paper builds on our prior work [2,3] that developed some of the compiler optimizations - optimizing global memory and shared memory access, and utilizing and managing on-chip shared memory. Recently, Lee et al. [17] developed a compiler framework for automatic translation from OpenMP to CUDA. The system handles both regular and irregular programs parallelized using OpenMP primitives. Work sharing constructs in OpenMP are translated into distribution of work across threads in CUDA. However the system does not optimize data access costs for access in global memory and also does not make use on-chip shared memory. Thus the optimizations implemented in our system can complement and enhance the effectiveness of their system.

Recently, Liu et al. [19] developed a GPU adaptive optimization framework (G-ADAPT) for automatic prediction of near-optimal configuration of parameters that affect GPU performance. They take unoptimized CUDA code as input and traverse an

optimization space search to determine optimal parameters to transform the unoptimized input CUDA code into an optimized CUDA code. Using our framework, a user can automatically generate CUDA code for any arbitrary input affine C code, hand-parallelization of which is very cumbersome in many cases. The user may then use G-ADAPT to further tune the CUDA code generated from our system.

7 Conclusions

In this paper, we have described an automatic source-to-source transformation framework that can take an arbitrarily nested affine input C program and generate an efficient CUDA program. Experimental results demonstrated the performance improvements achieved using the framework. We are in the process of creating a publicly available release of the C-to-CUDA transformation software.

Acknowledgments. This work was supported in part by the U.S. National Science Foundation through awards 0403342, 0508245, 0509442, 0509467, 0541409, 0811457, 0811781, 0926687 and 0926688, and by the Department of the Army through contract W911NF-10-1-0004.

References

1. Ancourt, C., Irigoin, F.: Scanning polyhedra with do loops. In: PPoPP 1991, pp. 39–50 (1991)
2. Baskaran, M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In: ACM ICS (June 2008)
3. Baskaran, M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories. In: ACM SIGPLAN PPoPP (February 2008)
4. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT 2004, pp. 7–16 (2004)
5. Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 132–146. Springer, Heidelberg (2008)
6. Bondhugula, U., Hartono, A., Ramanujan, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: ACM SIGPLAN Programming Languages Design and Implementation, PLDI 2008 (2008)
7. CLooG: The Chunky Loop Generator, `http://www.cloog.org`
8. Fatahalian, K., Sugerman, J., Hanrahan, P.: Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In: ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, pp. 133–137 (2004)
9. Feautrier, P.: Dataflow analysis of array and scalar references. IJPP 20(1), 23–53 (1991)
10. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part I: one-dimensional time. IJPP 21(5), 313–348 (1992)
11. Feautrier, P.: Automatic parallelization in the polytope model. In: Perrin, G.-R., Darte, A. (eds.) The Data Parallel Programming Model. LNCS, vol. 1132, pp. 79–103. Springer, Heidelberg (1996)
12. Govindaraju, N.K., Larsen, S., Gray, J., Manocha, D.: A memory model for scientific algorithms on graphics processors. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089. Springer, Heidelberg (2006)

13. General-Purpose Computation Using Graphics Hardware, http://www.gpgpu.org/
14. Griebl, M.: Automatic Parallelization of Loop Programs for Distributed Memory Architectures. Habilitation Thesis. FMI, University of Passau (2004)
15. Irigoin, F., Triolet, R.: Supernode partitioning. In: Proceedings of POPL 1988, pp. 319–329 (1988)
16. Nyland, L., Harris, M., Prins, J.F.: Fast N-body Simulation with CUDA. GPU Gems 3 article (August 2007)
17. Lee, S., Min, S.-J., Eigenmann, R.: Openmp to gpgpu: A compiler framework for automatic translation and optimization. In: PPoPP 2009, pp. 101–110 (2009)
18. Lim, A.: Improving Parallelism And Data Locality With Affine Partitioning. PhD thesis, Stanford University (August 2001)
19. Liu, Y., Zhang, E.Z., Shen, X.: A cross-input adaptive framework for gpu programs optimizations. In: IPDPS (May 2009)
20. NVIDIA CUDA, http://developer.nvidia.com/object/cuda.html
21. Parboil Benchmark Suite, http://impact.crhc.illinois.edu/parboil.php
22. Pluto: A polyhedral automatic parallelizer and locality optimizer for multicores http://pluto-compiler.sourceforge.net
23. Pouchet, L.-N., Bastoul, C., Cohen, A., Vasilache, N.: Iterative optimization in the polyhedral model: Part I, one-dimensional time. In: CGO 2007, pp. 144–156 (2007)
24. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM 8, 102–114 (1992)
25. Quilleré, F., Rajopadhye, S.V., Wilde, D.: Generation of efficient nested loops from polyhedra. IJPP 28(5), 469–498 (2000)
26. Ryoo, S., Rodrigues, C., Baghsorkhi, S., Stone, S., Kirk, D., Hwu, W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: ACM SIGPLAN PPoPP 2008 (February 2008)
27. Ryoo, S., Rodrigues, C., Stone, S., Baghsorkhi, S., Ueng, S., Hwu, W.: Program optimization study on a 128-core GPU. In: The First Workshop on General Purpose Processing on Graphics Processing Units (October 2007)
28. Ryoo, S., Rodrigues, C., Stone, S., Baghsorkhi, S., Ueng, S., Stratton, J., Hwu, W.: Program optimization space pruning for a multithreaded GPU. In: CGO (2008)
29. Vasilache, N., Bastoul, C., Girbal, S., Cohen, A.: Violated dependence analysis. In: ACM ICS (June 2006)