OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization

Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann
School of ECE, Purdue University
West Lafayette, IN, 47907, USA
{lee222,smin,eigenman}@purdue.edu

Abstract

GPGPUs have recently emerged as powerful vehicles for general-purpose high-performance computing. Although a new Compute Unified Device Architecture (CUDA) programming model from NVIDIA offers improved programmability for general computing, programming GPGPUs is still complex and error-prone. This paper presents a compiler framework for automatic source-to-source translation of standard OpenMP applications into CUDA-based GPGPU applications. The goal of this translation is to further improve programmability and make existing OpenMP applications amenable to execution on GPGPUs. In this paper, we have identified several key transformation techniques, which enable efficient GPU global memory access, to achieve high performance. Experimental results from two important kernels (JACOBI and SPMUL) and two NAS OpenMP Parallel Benchmarks (EP and CG) show that the described translator and compile-time optimizations work well on both regular and irregular applications, leading to performance improvements of up to 50X over the unoptimized translation (up to 328X over serial on a CPU).

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

General Terms Algorithms, Design, Performance

Keywords OpenMP, GPU, CUDA, Automatic Translation, Compiler Optimization

1. Introduction

Hardware accelerators, such as General-Purpose Graphics Processing Units (GPGPUs), are promising parallel platforms for high-performance computing. While a GPGPU provides an inexpensive, highly parallel system to application developers, its programming complexity poses a significant challenge for developers. There has been growing research and industry interest in lowering the barrier of programming these devices. Even though the CUDA programming model [4], recently introduced by NVIDIA, offers a more user-friendly interface, programming GPGPUs is still complex and error-prone, compared to programming general-purpose CPUs and parallel programming models such as OpenMP.

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA

OpenMP [13] has established itself as an important method and language extension for programming shared-memory parallel computers. There are several advantages of OpenMP as a programming paradigm for GPGPUs.

- OpenMP is efficient at expressing loop-level parallelism in applications, which is an ideal target for utilizing GPU's highly parallel computing units to accelerate data-parallel computations.
- The concept of a master thread and a pool of worker threads in OpenMP 's fork-join model represents well the relationship between the master thread running in a host CPU and a pool of threads in a GPU device.
- Incremental parallelization of applications, which is one of OpenMP's features, can add the same benefit to GPGPU programming.

The CUDA programming model provides a general-purpose multi-threaded Single Instruction, Multiple Data (SIMD) model for implementing general-purpose computations on GPUs. Although the unified processor model in CUDA abstracts underlying GPU architectures for better programmability, its unique memory model is exposed to programmers to some extent. Therefore, the manual development of high-performance codes in CUDA is more involved than in other parallel programming models such as OpenMP [2].

In this project, we developed an automatic OpenMP to GPGPU translator to extend the ease of creating parallel applications with OpenMP to GPGPU architectures. Due to the similarity between OpenMP and CUDA programming models, we were able to convert OpenMP parallelism, especially loop-level parallelism, into the forms that best express parallelism in CUDA. However, the baseline translation of existing OpenMP programs does not always yield good performance. Performance gaps are due to architectural differences between traditional shared-memory multiprocessors (SMPs), served by OpenMP, and stream architectures, adopted by most GPUs. Even though the OpenMP programming model is platform-independent, most existing OpenMP programs were tuned to traditional shared-memory multiprocessors. We refer to *stream architectures* as those that operate on a large data space (or stream) in parallel, typically in an SIMD manner and tuned for fast access to regular, consecutive elements of the data stream. In GPU architectures, optimization techniques designed for CPU-based algorithms may not perform well [7]. Also, GPUs face bigger challenges in handling irregular applications than SMPs, because of the stream architectures' preference for regular access patterns.

To address these issues, we propose compile-time optimization techniques and an OpenMP to GPGPU translation system, consisting of two phases: The OpenMP stream optimizer and the OpenMP-to-GPGPU (O_2G) baseline translator with CUDA opti-

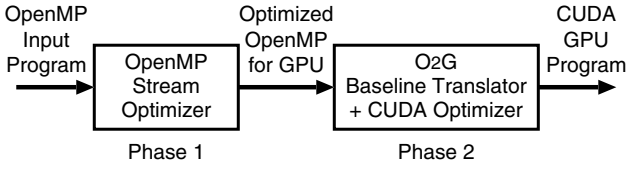


Figure 1. Two-phase OpenMP-to-GPGPU compilation system. Phase 1 is an OpenMP stream optimizer to generate optimized OpenMP programs for GPGPU architectures, and phase 2 is an OpenMP-to-GPGPU (O_2G) translator with CUDA optimizations.

mizer, shown in Figure 1. The OpenMP stream optimizer transforms traditional CPU-oriented OpenMP programs into OpenMP programs optimized for GPGPUs, using our high-level optimization techniques: *parallel loop-swap* and *loop-collapsing*. The O_2G translation converts the output of the OpenMP stream optimizer into CUDA GPGPU programs. The O_2G CUDA optimizer exploits CUDA-specific features.

We evaluate the baseline translation methods and the compile-time optimization techniques using two kernel benchmarks and two NPB3.0-OMP NAS Parallel Benchmarks, which have both regular and irregular applications. Experimental results show that the proposed compile-time optimization techniques can boost the performance of translated GPGPU programs up to 50 times over the unoptimized GPGPU translations.

This paper makes the following contributions:

- We present the first compiler framework for automatic source-to-source translation of standard OpenMP applications into CUDA-based GPGPU applications. It includes (1) the interpretation of OpenMP semantics under the CUDA programming model, (2) an algorithm to extract regions to be executed on GPUs, and (3) an algorithm for reducing CPU-GPU memory transfers.
- We have identified several compile-time transformation techniques to optimize GPU *global memory* access: (1) *parallel loop-swap* and *matrix transpose* techniques for regular applications and (2) *loop collapsing* for irregular applications.
- We have evaluated the proposed translator and optimizations on both applications (NAS OpenMP Benchmarks *EP* and *CG*) and kernels, resulting in performance improvements up to 50X (12X on average) over the unoptimized translations (up to 328X over serial on the CPU).

The rest of this paper is organized as follows: Section 2 provides an overview of the CUDA programming model, and Section 3 presents the baseline OpenMP to CUDA GPGPU translator. In Section 4, various compiler techniques to optimize the performance of GPGPU programs are explained. Experimental results are shown in Section 5, and related work and conclusion are presented in Section 6 and Section 7, respectively.

2. Overview of the CUDA Programming Model

The CUDA programming model is a general-purpose multi-threaded SIMD model for GPGPU programming. In the CUDA programming model, a GPU is viewed as a parallel computing coprocessor, which can execute a large number of threads concurrently. A CUDA program consists of a series of sequential and parallel execution phases. Sequential phases have little or no parallelism, and thus they are executed on the CPU as host code. Parallel phases that exhibit rich data parallelism are implemented as a set of *kernel functions*, which are executed on the GPU. Each kernel function specifies GPU code to be executed in an SIMD fashion, by a number of threads invoked for each parallel phase.

In the CUDA model, threads are grouped as a grid of thread blocks, each of which is mapped to a multiprocessor in the GPU device. In CUDA-supported GPU architectures, more than one thread block can be assigned to a multiprocessor, and threads within each thread block are mapped to SIMD processing units in the multiprocessor. The number of thread blocks and the number of threads per thread block are specified through language extensions at each kernel invocation.

The CUDA memory model has an off-chip *global memory* space, which is accessible by all threads, an off-chip *local memory* space, which is private to each thread, a fast on-chip *shared memory* space, which is shared only by threads in the same thread block, and *registers*, which are private to each thread. The CUDA memory model also has separate memory spaces to exploit specialized hardware memory resources: *constant memory* with a dedicated small cache for read-only global data that are frequently accessed by many threads across multiple thread blocks and *texture memory* with a dedicated small cache for read-only array data accessed through built-in texture functions. The shared memory and the register bank in a multiprocessor are dynamically partitioned among the active thread blocks running on the multiprocessor. Therefore, register and shared memory usages per thread block can be a limiting factor preventing full utilization of execution resources.

In the CUDA programming model, a host CPU and a GPU device have separate address spaces. For a CPU to access GPU data, the CUDA model provides an API for explicit GPU memory management, including functions to transfer data between a CPU and a GPU.

One limitation of the CUDA model is the lack of efficient global synchronization mechanisms. Synchronization within a thread block can be enforced by using the *__syncthreads()* runtime primitive, which guarantees that all threads in the same thread block have reached the same program point, and data modified by threads in the same thread block are visible to all threads in the same block. However, synchronization across thread blocks can be accomplished only by returning from a kernel call, after which all threads executing the kernel function are guaranteed to be finished, and *global memory* data modified by threads in different thread blocks are guaranteed to be globally visible.

3. Baseline Translation of OpenMP into CUDA

This section presents a baseline translator, which performs a source-to-source conversion of an OpenMP program to a CUDA-based GPGPU program. The translation consists of several steps: (1) interpreting OpenMP semantics under the CUDA programming model and identifying *kernel regions* (code sections executed on the GPU), (2) outlining (extracting into subroutines) kernel regions and transforming them into CUDA kernel functions, and (3) analyzing shared data that will be accessed by the GPU and inserting necessary memory transfer calls. We have implemented these translation steps using the Cetus compiler infrastructure [8].

3.1 Interpretation of OpenMP Semantics under the CUDA Programming Model

OpenMP directives can be classified into four categories:

(1) *Parallel* constructs – these are the fundamental constructs that specify parallel regions. The compiler identifies these regions as candidate *kernel regions*, outlines them, and transforms them into GPU kernel functions.

(2) Work-sharing constructs (*omp for*, *omp sections*) – the compiler interprets these constructs to partition work among threads on the GPU device. Each iteration of an *omp for* loop is assigned to a thread, and each section of *omp sections* is mapped to a thread.

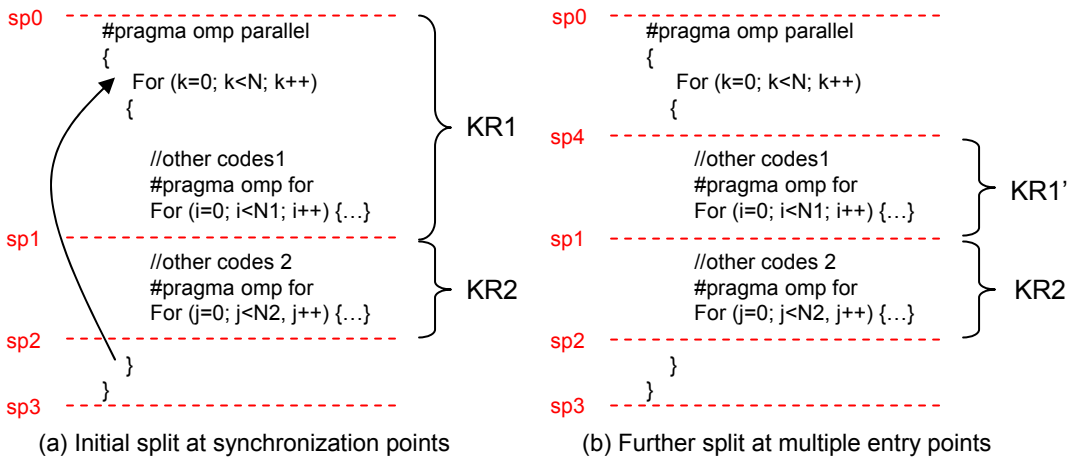


Figure 2. Parallel region example showing how multiple splits are applied to identify kernel regions. *sp0* - *sp4* are split points enforced to preserve OpenMP semantics, and *KR1'* and *KR2* are kernel regions to be converted into kernel functions.

(3) Synchronization constructs (*omp barrier*, *omp flush*, *omp critical*, etc.) – these constructs constitute *split points*, points where a parallel region must be split into two sub-regions; each of the resulting sub-regions becomes a kernel region. This split is required to enforce a global synchronization in the CUDA programming model, as explained in Section 2.

(4) Directives specifying data properties (*omp shared*, *omp private*, *omp threadprivate*, etc.) – these constructs are used to map data into GPU memory spaces. As mentioned in Section 2, the CUDA memory model requires explicit memory transfers for threads invoked for a kernel function to access data on the CPU. OpenMP *shared* data are shared by all threads, and OpenMP *private* data are accessed by a single thread. In the CUDA memory model, the shared data can be mapped to *global memory*, and the private data can be mapped to *registers* or *local memory* assigned for each thread. OpenMP *threadprivate* data are private to each thread, but they have global lifetimes, as do static data. The semantics of *threadprivate* data can be implemented by expansion, which allocates copies of the *threadprivate* data on *global memory* for each thread. Because the CUDA memory model allows several specialized memory spaces, certain data can take advantage of the specialized memory resources; read-only shared data can be assigned to either *constant memory* or *texture memory* to exploit temporal locality through dedicated caches, and frequently reused shared data can use fast memory spaces, such as *registers* and *shared memory*, as a cache.

3.2 OpenMP to CUDA Baseline Translation

The previous subsection described the interpretation of OpenMP semantics under the CUDA programming model. The next step performs the actual translation into a CUDA program. A simple translation scheme might convert all code sections specified by work-sharing constructs into kernel functions, since work-sharing constructs contain the only true parallel code in OpenMP. Other sub-regions, within an *omp parallel* but outside of work-sharing constructs, are executed by one thread (*omp master* and *omp single*), serialized among threads (*omp ordered* and *omp critical*), or executed redundantly among participating threads. However, our compiler includes some of these sub-regions into kernel regions, thus redundantly executing them; this method can reduce expensive memory transfers between the CPU and the GPU.

With these concepts in mind, we will explain the baseline translation schemes in the following subsections.

3.2.1 Identifying Kernel Regions

The compiler targets OpenMP parallel regions as potential *kernel regions*. As explained above, these regions may be split at synchronization constructs. Among the resulting sub-regions, the ones containing at least one work-sharing construct become kernel regions.

The translator must consider that split operations may break the control flow semantics of the OpenMP programming model, if the *split points* lie within control structures. In the OpenMP programming model, most directives work only on a *structured block* – a block of code with one entry and one exit point. If a parallel region is split in the middle of a control structure, the resulting kernel regions may become an unstructured block. Figure 2 (a) shows an example where a split operation would result in incorrect control flow. In the OpenMP programming model, a *flush* synchronization construct is implied at the entry to and exit from *parallel* regions (*sp0* and *sp3* in Figure 2 (a)) and at the exit from work-sharing regions (*sp1* and *sp2*), unless a *nowait* clause is present. The split operation identifies two kernel regions: *KR1* and *KR2*. Because the first kernel region, *KR1*, has multiple entry points, outlining this region will break control flow semantics. To solve this problem, our translator splits *KR1* further at multiple entry points. In Figure 2 (b), an additional split is applied at *sp4*, turning *KR1'* into a structured block with correct control flow.

The overall algorithm to identify kernel regions is shown in Figure 3. The idea behind this top-down splitting algorithm is to merge as many work-sharing regions as possible to reduce overheads by kernel invocations and CPU-GPU data transfers.

3.2.2 Transforming a Kernel Region into a Kernel Function

The translator outlines the identified kernel regions into CUDA kernel functions and replaces the original regions with calls to these functions.

At this stage, two important translation steps are involved: work partitioning and data mapping. For work partitioning, iterations of *omp for* loops are partitioned among threads using the rules of the OpenMP *schedule* clause, each section in *omp sections* is mapped to a thread, and remaining code sections in the kernel region are executed redundantly by all threads. The compiler decides the number of threads to be invoked for the kernel execution as the maximum number of threads needed for each work-sharing subregion contained in the kernel region. Once the compiler figures out the total number of threads, the number of thread blocks is also

Identifying kernel regions

Input: R /* a set of OpenMP parallel regions */
Output: KR /* a set of identified GPU kernel regions */

```
Foreach R(i) in R
    Foreach split point in R(i)
        Divide R(i) into two sub-regions at the split point
    Build cfg, a control flow graph for R(i)
    Foreach sub-region SR(j) in R(i)
        Foreach entry/exit points other than the one at the top/bottom of SR(j)
            Divide SR(j) into two sub-sub-regions at such entry/exit points
        Foreach sub-sub-region SSR(k) in SR(j)
            If SSR(k) contains an OpenMP work-sharing construct
                Add SSR(k) to KR set
```

Figure 3. Algorithm to identify kernel regions

calculated using default thread block size, which can be set through a command line option.

After work partitioning, the compiler constructs the sets of shared data and private data used in the kernel region, using the information specified by data property constructs. In the OpenMP programming model, data is shared by default, including data with global scope or with heap-allocated storage. For the data that are referenced in the region, but not in a construct, the compiler can determine their sharing attributes using OpenMP data sharing rules. Basic data mapping follows the rule explained in Section 3.1; shared data are mapped to *global memory*, *threadprivate* data are replicated and allocated on *global memory* for each thread, and private data are mapped to *register banks* assigned for each threads.

As a part of the data mapping step, the compiler inserts necessary memory transfer calls for the *shared* and *threadprivate* data accessed by each kernel function. A basic strategy is to move all the shared data that are accessed by kernel functions, and copy back the shared data that are modified by kernel functions. (*threadprivate* data transfers are decided by OpenMP semantics.) However, not all shared data are used by the CPU after the kernel completes. Also, data in the GPU *global memory* are persistent across kernel calls. Therefore, not all these data transfers are needed. The compiler optimization technique to eliminate redundant data transfers will be discussed in the following section.

Additionally, during this translation, if a *omp for* loop contains a *reduction* clause, the compiler replaces the reduction operation with the two-level tree reduction form proposed in [5]: a local parallel reduction within each thread block, followed by a host-side global reduction across thread blocks.

In the baseline translation scheme, *omp critical* regions are executed on the host CPU since the *omp critical* construct involves global synchronizations, which are expensive on GPU kernel executions due to kernel splits, and the semantic of *omp critical* requires serialized execution of the specified region. However, if the *critical* regions have reduction forms, the same transformation technique used to interpret a *reduction* clause [5] can be applied.

4. Compiler Optimizations

We describe our two-phase optimization system. The first is the OpenMP stream optimizer, and the second is O_2G (OpenMP-to-GPGPU) CUDA optimizer.

4.1 OpenMP Stream Optimizations

Both the OpenMP and GPGPU models are suitable for expressing data parallelism. However, there are important differences. GPUs are designed as massively parallel machines for concurrent execution of thousands of threads, each executing the same code on different data (SIMD). GPU threads are optimized for fine-grain data parallelism, which is characterized by regular memory accesses and regular program control flow. On the other hand, OpenMP threads are more autonomous, typically execute coarse-grain parallelism, and are able to handle MIMD computation.

The OpenMP stream optimizer addresses these differences by transforming the traditional CPU-oriented OpenMP programs into GPU-style OpenMP programs. One benefit of this high-level translation method is that the user can see the optimization result at the OpenMP source code level.

4.1.1 Intra-Thread vs. Inter-Thread Locality

In OpenMP, data locality is often exploited within a thread (intra-thread locality), but less so among threads that are executed on different CPU nodes (inter-thread locality). This is because spatial data locality within a thread can increase cache utilization, but, locality among OpenMP threads on different CPUs of cache-coherent shared-memory multiprocessors can incur false-sharing. Therefore, in OpenMP, parallel loops are usually block-distributed, rather than in a cyclic manner.

In contrast to OpenMP, inter-thread locality between GPU threads plays a critical role in optimizing off-chip memory access performance. For example, in CUDA, GPU threads that access the off-chip memory concurrently can coalesce their memory accesses to reduce the overall memory access latency. These coalesced memory accesses can be accomplished if such GPU threads exhibit inter-thread locality where adjacent threads access adjacent locations in the off-chip memory. In the OpenMP form, a cyclic distribution of parallel loops can expose inter-thread locality.

The following subsections describe our two compile-time optimization techniques, *parallel loop-swap* and *loop-collapsing*, to enhance inter-thread locality of OpenMP programs on GPGPUs.

4.1.2 Parallel Loop-Swap for Regular Applications

In this section, we introduce a *parallel loop-swap* optimization technique to improve the performance of regular data accesses in nested loops. Previously, we have described how cyclic distribution can improve inter-thread locality in a singly nested loop via an

```
#pragma omp parallel for
for (i=1; i<=SIZE; i++) {
  for (j=1; j<=SIZE; j++)
    a[i][j] = (b[i-1][j] + b[i+1][j]
               + b[i][j-1] + b[i][j+1])/4;
}
```

(a) input OpenMP code

```
#pragma omp parallel for
for (i=1; i<=SIZE; i++) {
#pragma cetus parallel
  for (j=1; j<=SIZE; j++)
    a[i][j] = (b[i-1][j] + b[i+1][j]
               + b[i][j-1] + b[i][j+1])/4;
}
```

(b) Cetus-parallelized OpenMP code

```
#pragma omp parallel for schedule(static, 1)
for (j=1; j<=SIZE; j++)
  for (i=1; i<=SIZE; i++) {
    a[i][j] = (b[i-1][j] + b[i+1][j]
               + b[i][j-1] + b[i][j+1])/4;
}
```

(c) OpenMP output by OpenMP stream optimizer

```
// tid is a GPU thread identifier
for (tid=1; tid<=SIZE; tid++)
  for (i=1; i<=SIZE; i++) {
    a[i][tid] = (b[i-1][tid] + b[i+1][tid]
               + b[i][tid-1] + b[i][tid+1])/4;
}
```

(d) internal representation in O2G translator

```
// Each iteration of the parallel-for loop
// is cyclic-distributed to each GPU thread
if (tid<=SIZE) {
  for (i=1; i<=SIZE; i++) {
    a[i][tid] = (b[i-1][tid] + b[i+1][tid]
             + b[i][tid-1] + b[i][tid+1])/4;
  }
}
```

(e) GPU code

Figure 4. Regular application example from JACOBI, to show how *parallel loop-swap* is applied to a nested parallel loop to improve inter-thread locality

```
#pragma omp parallel for
 for (i=0; i<NUM_ROWS; i++) {
   for (j=rowptr[i]; j<rowptr[i+1]; j++)
     w[i] += A[j]*p[col[j]];
 }
```

(a) input OpenMP code

```
#pragma omp parallel for
 for (i=0; i<NUM_ROWS; i++) {
   #pragma cetus parallel reduction(+:w[i])
   for (j=rowptr[i]; j<rowptr[i+1]; j++)
     w[i] += A[j]*p[col[j]];
 }
```

(b) Cetus-parallelized OpenMP code

```
#pragma omp parallel
#pragma omp for collapse(2) schedule(static, 1)
 for (i=0; i<NUM_ROWS; i++) {
   for (j=rowptr[i]; j<rowptr[i+1]; j++)
     w[i] += A[j]*p[col[j]];
 }
```

(c) OpenMP output by OpenMP stream optimizer

```
 // collapsed loop
 for (tid1=0; tid1<rowptr[NUM_ROWS]; tid1++)
   l_w[tid1] = A[tid1]*p[col[tid1]];

 // For each GPU thread, a new thread-id, tid2,
 // is assigned for the reduction loop
 for (tid2=0; tid2<NUM_ROWS; tid2++) {
   for (j=rowptr[tid2]; j<rowptr[tid2+1]; tid2++)
     w[tid2] += l_w[j];
 }
```

(d) internal representation in O2G translator

```
 if (tid1<rowptr[NUM_ROWS]) {
   l_w[tid1] = A[tid1]*p[col[tid1]];
 }
 if (tid2<NUM_ROWS) {
   for (j=rowptr[tid2]; j<rowptr[tid2+1]; j++)
     w[tid2] += l_w[j];
 }
```

(e) GPU code

Figure 5. Example of an irregular application, CG NAS Parallel Benchmark. *Loop-collapsing* eliminates irregular control flow and improves inter-thread locality

OpenMP `schedule(static, 1)` clause. However, in a nested loop, we need advanced compile-time techniques to achieve this goal. For example, the input OpenMP code shown in Figure 4 (a) has a doubly nested loop, where the outer loop is parallelized with a block distribution of iterations, which prevents the O_2G translator from applying the coalesced memory optimization to the accessed arrays.

To solve this problem, our OpenMP stream optimizer performs a *parallel loop-swap* transformation. We define *continuous memory access* as a property of an array in a loop-nest, where the array subscript expression increases monotonically with a stride of one, and the adjacent elements accessed by the subscript expression are continuous in the memory. An array with *continuous memory access* is a candidate for the coalesced memory access optimization.

The OpenMP stream optimizer performs *parallel loop-swap* in five steps: (1) for a given OpenMP parallel loop nest with an OpenMP work-sharing construct on loop L, the compiler finds a set of *candidates*, arrays with *continuous memory access* within the loop-nest. (2) The compiler selects the loop, L^*, whose index variable increments the subscript expression of the array accesses in *candidates* by one. (3) The compiler applies parallelization techniques to find all possible parallel loops. (4) If all the loops between L and (including) L^* can be parallelized, these two loops are interchanged. (5) An OpenMP parallel work-sharing construct with cyclic distribution is added to loop L^*.

Figure 4 (b) shows the result of step 3, where a Cetus parallel pragma is added to the discovered parallel loop. Figure 4 (c) illustrates the result of the *parallel loop-swap* transformation per-

formed by the OpenMP stream optimizer. The O_2G translator converts this transformed OpenMP code into the internal representation, as shown in Figure 4 (d). Figure 4 (e) shows the CUDA GPU code.

4.1.3 Loop Collapsing for Irregular Applications

Irregular applications pose challenges in achieving high performance on GPUs because stream architectures are optimized for regular program patterns. In this section, we propose a *loop-collapsing* technique, which improves the performance of irregular OpenMP applications on such architectures.

There are two main types of irregular behavior in parallel programs: (1) data access patterns among threads due to indirect array accesses and (2) different control flow paths taken by different threads, such as in conditional statements. Irregular data accesses prevent the compiler from applying the memory coalescing technique because it cannot prove *continuous memory access* in the presence of indirect references. Irregular control flow prevents the threads from executing fully concurrently on the GPU's SIMD processors.

Figure 5 (a) shows one of the representative irregular program patterns in scientific applications. It exhibits both irregular data access patterns caused by the indirect array accesses in `A`, `p`, and `col` and control flow divergence because the inner loop depends on the value of array `rowptr`; different GPU threads will execute different numbers of inner loop iterations. Our first optimization technique, *parallel loop-swap*, cannot be applied to this irregular case because the dependency prevents loop interchange.

In this example, the compiler tries to collapse the doubly nested loop into a single loop and prove that the accesses to array `A` and `col` are *continuous memory accesses*. This technique eliminates both irregular data accesses to arrays `A` and `col` as well as control flow divergence in the inner loop. To prove the *continuous memory access* property to array `A` and `col`, one needs to prove that `i`, `rowptr`, and `j` are all monotonically increasing, and `i` and `j` are increasing with stride one. Since both `i` and `j` are loop index variables, this proof can be done at compile-time. However, the monotonicity of array `rowptr` cannot be proven at compile-time if it is read from a file. In that case, the compiler inserts a runtime check, after the array `rowptr` is defined, to verify monotonicity.

The *loop-collapsing* optimization is implemented in three steps. First, for each perfectly nested loop in the OpenMP source, the OpenMP stream optimizer finds all possible parallel loops in a given nest (Figure 5 (b)). Second, among parallel loops, the optimizer performs monotonicity checks to identify the parallel loops eligible for loop-collapsing. Third, the optimizer annotates an OpenMP clause *collapse (k)* to indicate that the first *k* nested loops can be collapsed (Figure 5 (c)). The O_2G translator reads this optimized OpenMP program and transforms the loops into a single loop. Figure 5 (d) shows two loops in the internal representation of the O_2G translator. The first loop is the output of *loop-collapsing*, where each GPU thread executes one iteration of the collapsed loop. The second loop is for handling the reduction pattern recognized in Figure 5 (b).

The *loop-collapsing* transformation improves the performance of irregular OpenMP applications on GPUs in three ways:

- The amount of parallel work (the number of iterations, to be executed by GPU threads) is increased.
- Inter-thread locality is increased, especially for cases where *parallel loop-swap* cannot be applied.
- Control flow divergence is eliminated, such that adjacent threads can be executed concurrently in an SIMD manner.

4.2 O_2G CUDA Optimization Techniques

This section describes the CUDA optimizer, a collection of optimization techniques for translating OpenMP programs into actual GPGPU programs. These transformations differ from those in the OpenMP stream optimizer in that they are specific to features of the CUDA memory architecture.

4.2.1 Caching of Frequently Accessed Global Data

In the CUDA model, the global memory space is not cached. Therefore, to exploit temporal locality, frequently accessed global data must be explicitly loaded into fast memory spaces, such as *registers* and *shared memory*. In CUDA, exploiting temporal locality both within threads (intra-thread locality) and across threads (inter-thread locality) is equally important. *Shared memory* is shared by threads in a thread block, and the dedicated caches for *texture memory* and *constant memory* are shared by all threads running on the same multiprocessor.

In traditional shared-memory systems, temporal locality across threads is automatically exploited by the hardware caches. Therefore, in the OpenMP programming model, most existing caching-related optimizations focus on temporal locality within a thread only. However, under the CUDA memory model, software cache management for both intra- and inter-thread locality is important.

Our compiler performs the requisite *data flow analysis* to identify temporal locality of global data and inserts the necessary caching code. Table 1 shows caching strategies for each global data type. The baseline translator maps globally shared data into *global memory*, but depending on the data attributes, they can be cached in specific fast memory spaces, as shown in the table.

Table 1. Caching strategies for globally shared data with temporal locality. In *A(B)* format, *A* is a primary storage for caching, but *B* may be used as an alternative. *Reg* denotes *Registers*, *CC* means *Constant Cache*, *SM* is *Shared Memory*, and *TC* represents *Texture Cache*.

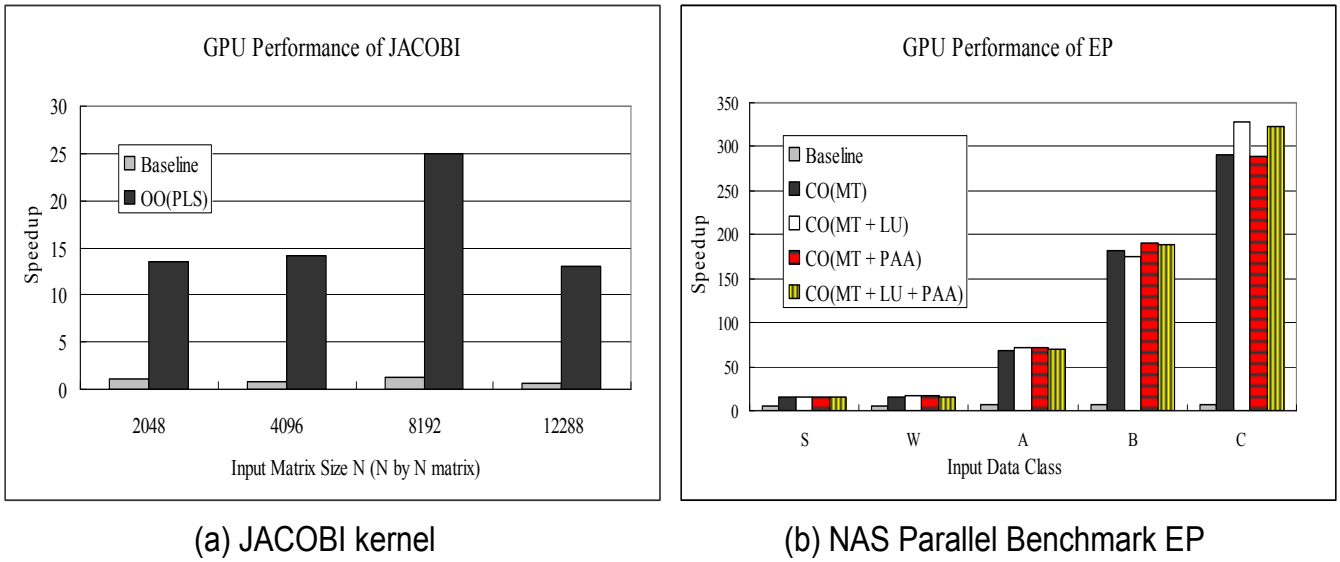
	Temporal locality	
	Intra-thread	Inter-thread
R/O shared scalar	Reg (CC or SM)	CC (SM)
R/W shared scalar	Reg (SM)	SM
R/O shared array	TC (SM)	TC (SM)
R/W shared array	SM	SM

Another CUDA-specific caching optimization is to allocate a CUDA-private array in *shared memory* if possible. In the CUDA memory model, a device-local array is allocated in *local memory*, which is a part of the off-chip DRAM; accessing the *local memory* is as slow as accessing *global memory*. If the private array size is small, it may be allocated on *shared memory* using an array expansion technique.

However, complex interactions among limited hardware resources may cause performance effects that are difficult to control statically. Therefore, our compiler framework provides language extensions and command line options for a programmer or an automatic tuning system to guide these optimizations.

4.2.2 Matrix Transpose for Threadprivate Array

The OpenMP-to-GPGPU translator must also handle the mapping of *threadprivate* data. When *threadprivate* array is placed into *global memory*, the translator implements correct semantics by expanding the *threadprivate* array for each thread. Row-wise expansion is a common practice to preserve intra-thread locality in traditional shared-memory systems. However, row-wise array expansion would cause uncoalesced memory accesses in GPUs. In this case, *parallel loop-swap* can not be applied due to dependencies or *threadprivate* semantics.



(a) JACOBI kernel (b) NAS Parallel Benchmark EP

Figure 6. Performance of Regular Applications (speedups are over serial on the CPU). *Baseline* is the baseline translation without optimization; the other bars measure the following OpenMP optimizations (*OO*) or CUDA optimizations (*CO*): *Parallel Loop-Swap* (*PLS*), *Matrix Transpose* (*MT*), *Loop Unrolling* (*LU*), and private array allocation on *shared memory* using array expansion (*PAA*). The performance irregularity shown in the left figure is caused by reproducible, inconsistent behavior of the CPU execution; the average CPU execution time per array element is longer at $N = 8192$ than others.

The *matrix transpose* transformation solves this problem by converting the array into a form where the *threadprivate* array is expanded in a column-wise manner. *Matrix transpose* changes intra-thread array access patterns from row-wise to column-wise, so that adjacent threads can access adjacent data, as needed for coalesced accesses.

4.2.3 Memory Transfer Reduction

A final, important step of the actual OpenMP to GPGPU translation is the insertion of CUDA memory transfer calls for the *shared* and *threadprivate* data accessed by each kernel function. The baseline translation inserts memory transfer calls for all shared data accessed by the kernel functions. However, not all of these data moves between the CPU and the GPU are needed. To remove unnecessary transfers, our compiler performs a *data flow analysis*; for each kernel region, (1) the compiler finds a set of shared data read in the kernel (*UseSet*) and a set of shared data written in the kernel (*DefSet*), (2) for each variable in the *UseSet*, if its reaching definition is in the host code, it should be transferred from the CPU, and (3) for each variable in the *DefSet*, if the variable is used in the host code, it should be copied back to the CPU after the kernel returns. The algorithm reduces unnecessary data transfers between the CPU and the GPU for each kernel function. Currently, the algorithm performs conservative array-name only analysis for shared array data. Our experimental results show that this analysis is able to reduce memory transfers sufficiently.

5. Performance Evaluation

This section presents the performance of the presented OpenMP to GPGPU translator and compiler optimizations. In our experiments, two regular OpenMP programs (*JACOBI* kernel and NAS OpenMP Parallel Benchmark *EP*) and two irregular OpenMP programs (*SPMUL* kernel and NAS OpenMP Parallel Benchmark *CG*) were transformed by the translator. The baseline translations and optimizations were performed automatically by the compiler framework, but some advanced compiler optimizations using interprocedural analysis were applied manually.

We used an NVIDIA Quadro FX 5600 GPU as an experimental platform. The device has 16 multiprocessors with a clock rate of 1.35 GHz and 1.5GB of DRAM. Each multiprocessor is equipped with 8 SIMD processing units, totaling 128 processing units. The device is connected to a host system consisting of Dual-Core AMD 3 GHz Opteron processors. Because the tested GPU does not support double precision, we manually converted the OpenMP source programs into single precision before feeding them to our translator. (NVIDIA recently announced GPUs supporting double precision computations.) We compiled the translated CUDA programs with the NVIDIA CUDA Compiler (NVCC) to generate device code. We compiled the host programs with the *GCC* compiler version 4.2.3, using option *-O3*.

5.1 Performance of Regular Applications

JACOBI is a widely used kernel containing the main loop of an iterative solver for regular scientific applications. Due to its simple structure, the *JACOBI* kernel is easily parallelized in many parallel programming models. However, the base-translated GPU code does not perform well, as shown in Figure 6 (a); *Baseline* in the figure represents the speedups of the unoptimized GPU version over serial on the CPU. This performance degradation is mostly due to the overhead in large, uncoalesced global memory access patterns. These uncoalesced access patterns can be changed to coalesced ones by applying *parallel loop-swap* (*PLS* in Figure 6 (a)). These results demonstrate that, in regular programs, uncoalesced global memory accesses may be converted to coalesced accesses by loop transformation optimizations.

EP is one of the NAS OpenMP Parallel Benchmarks. *EP* has abundant parallelism with minimal communication; it is often used to demonstrate the performance of parallel computing systems. However, the baseline translation of *EP* shows surprisingly low speedups on the tested GPU (*Baseline* in Figure 6 (b)).

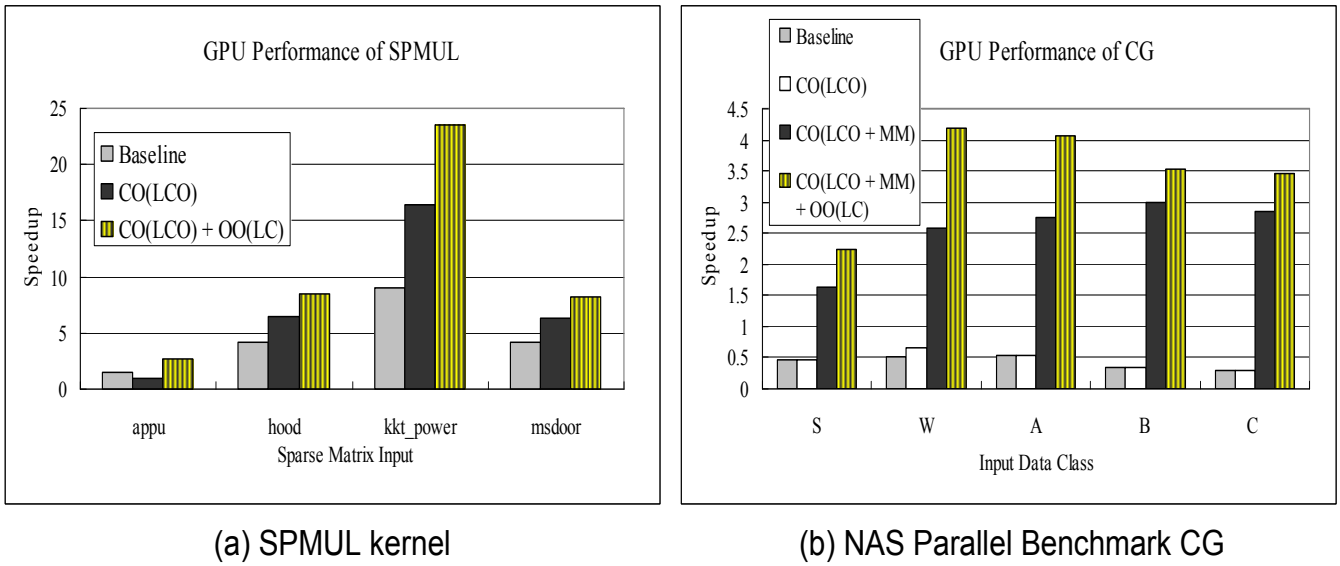


Figure 7. Performance of Irregular Applications (speedups are over serial on the CPU). *Baseline* is the baseline translation without optimization; the other bars represent the following OpenMP optimizations (*OO*) or CUDA optimizations (*CO*): *LCO* is a local caching optimization for global data, *LC* means the *loop collapsing* transformation technique, and *MM* represents a redundant memory transfer minimizing optimization.

At the core of *EP* is a random number generator; each participating thread or process performs some computations based on the chosen numbers. In an OpenMP version, each thread stores the random numbers in a *threadprivate* array. The baseline translation scheme allocates the memory space for the *threadprivate* array by expanding the array in a row-wise fashion. However, this row-wise array expansion causes uncoalesced memory access problems under the CUDA memory model.

The *Matrix transpose* transformation in Section 4.2.2 resolves this limitation; it changes the access patterns of the *threadprivate* arrays into coalesced accesses. *MT* in Figure 6 (b) shows the speedups when the *matrix transpose* transformation is applied. The results show that *matrix transpose* increases the performance tremendously. To increase the performance of the translated *EP* further, additional CUDA-specific optimizations are applied; if the size of a private array, which is normally mapped to *local memory* in the GPU device, is small, it can be allocated on *shared memory* by using array expansion (*PAA* in Figure 6 (b)). Also, *loop unrolling* can be applied to reduction computations (*LU* in Figure 6 (b)). Performance variations in Figure 6 (b) indicate that complex interactions among the hardware resources may need fine tuning for optimal performance. *EP* is a representative case showing the importance of compile-time transformations.

5.2 Performance of Irregular Applications

Sparse matrix-vector (SpMV) multiplication is an important representative of irregular applications. In our experiments, we have translated two codes using SpMV, the *SPMUL* kernel and the NAS OpenMP Parallel Benchmark *CG*, to measure the performance of the proposed system on irregular applications.

In *SPMUL*, a baseline translation without optimizations gives reasonable speedups (*Baseline* in Figure 7 (a)) on several real sparse matrices in the UF Sparse Matrix Collection [6]. Simple *local caching* optimizations, such as caching frequently accessed global data on registers and using *texture memory* to exploit a dedicated cache for read-only global data, work well (*LCO* in Figure 7 (a)), since some data in the *global memory* are accessed repeatedly either within each thread or across threads.

From an algorithmic perspective, the translated *SPMUL* conducts SpMV multiplications by assigning each row to a thread and letting each thread compute the inner product for the assigned row, which takes a reduction form. This algorithmic structure is implemented as a nested loop, which exhibits blocked access patterns, where each thread accesses a block of continuous data in *global memory*. Blocked access results in significantly lower memory bandwidth than cyclic access due to uncoalesced memory access. The *loop collapsing* technique in Section 4.1.3 can change the blocked access patterns to a cyclic access pattern, increasing the effective bandwidth on *global memory* (*LC* in Figure 7 (a)).

CG is another sparse linear solver program. While both *CG* and *SPMUL* use similar SpMV multiplication algorithms, *CG* poses additional challenges. In *CG*, each parallel region contains many work-sharing regions including *omp single*. Depending on kernel-region-extracting strategies, the amount of overheads related to kernel invocations and memory transfers will be changed. The kernel-region-identifying algorithm in Section 3.2.1 merges as many work-sharing regions as possible to minimize the kernel-related overheads. As indicated in Figure 7 (b), however, the base GPU version (*Baseline*) still incurs very large memory transfer overheads, degrading the performance below the serial version. Eliminating redundant memory transfers with the algorithm described in Section 4.2.3 reduces these overheads. In *CG*, interprocedural data flow analysis is needed to identify this redundancy, since work-sharing regions are distributed among several subroutines. The results show that eliminating redundant data transfers increases the performance significantly (*MM* in Figure 7 (b)), and *loop collapsing* transformation, used in *SPMUL*, also works well on *CG* (*LC* in Figure 7 (b)).

6. Related Work

Prior to the advent of the CUDA programming model [4], programming GPUs was highly complex, requiring deep knowledge of the underlying hardware and graphics programming interfaces.

Although the CUDA programming model provides improved programmability, achieving high performance with CUDA programs is still challenging. Several studies have been conducted to optimize the performance of CUDA-based GPGPU applications; an optimization space pruning technique [15] has been proposed, using a Pareto-optimal curve, to find the optimal configuration for a GPGPU application. Also, an experimental study on general optimization strategies for programs on a CUDA-supported GPU has been presented [14]. In these contributions, optimizations were performed manually.

For the automatic optimization of CUDA programs, a compile-time transformation scheme [2] has been developed, which finds program transformations that can lead to efficient global memory access. The proposed compiler framework optimizes affine loop nests using a polyhedral compiler model. By contrast, our compiler framework optimizes irregular loops, as well as regular loops. Moreover, we have demonstrated that our framework performs well on actual benchmarks as well as on kernels. CUDA-lite [18] is another translator, which generates codes for optimal tiling of global memory data. CUDA-lite relies on information that a programmer provides via annotations, to perform transformations. Our approach is similar to CUDA-lite in that we also support special annotations provided by a programmer. In our compiler framework, however, the necessary information is automatically extracted from the OpenMP directives, and the annotations provided by a programmer are used for fine tuning.

OpenMP [13] is an industry standard directive language, widely used for parallel programming on shared memory systems. Due to its well established model and convenience of incremental parallelization, the OpenMP programming model has been ported to a variety of platforms. Previously, we have developed compiler techniques to translate OpenMP applications into a form suitable for execution on a Software Distributed Shared Memory (DSM) system [10, 11] and another compile-time translation scheme to convert OpenMP programs into MPI message-passing programs for execution on distributed memory systems [3]. Recently, there have been several efforts to map OpenMP to Cell architectures [12, 19]. Our approach is similar to the previous work in that OpenMP parallelism, specified by work-sharing constructs, is exploited to distribute work among participating threads or processes, and OpenMP data environment directives are used to map data into underlying memory systems. However, different memory architectures and execution models among the underlying platforms pose various challenges in mapping data and enforcing synchronization for each architecture, resulting in differences in optimization strategies. To our knowledge, the proposed work is the first to present an automatic OpenMP to GPGPU translation scheme and related compile-time techniques.

MCUDA [16] is an opposite approach, which maps the CUDA programming model onto a conventional shared-memory CPU architecture. MCUDA can be used as a tool to apply the CUDA programming model for developing data-parallel applications running on traditional shared-memory parallel systems. By contrast, our motivation is to reduce the complexity residing in the CUDA programming model, with the help of OpenMP, which we consider to be an easier model. In addition to the ease of creating CUDA programs with OpenMP, our system provides several compiler optimizations to reduce the performance gap between hand-optimized programs and auto-translated ones.

To bridge the abstraction gap between domain-specific algorithms and current GPGPU programming models such as CUDA, a framework for scalable execution of domain-specific templates on GPUs has been proposed [17]. This work is complementary to our work in that it addresses the problem of partitioning the computations that do not fit into GPU memory.

The compile-time transformations proposed in this paper are not fundamentally new ones; vector systems use similar transformations [1, 9, 20]. However, the architectural differences between GPGPUs and vector systems pose different challenges in applying these techniques, leading to different directions; *parallel loop-swap* and *loop collapsing* transformations are enabling techniques to expose stride-one accesses in a program so that concurrent GPU threads can use the coalesced memory accesses to optimize the off-chip memory performance. On the other hand, *loop interchange* in vectorizing compilers is to enable vectorization of certain loops within a single thread.

7. Conclusion

In this paper, we have described a compiler framework for translating standard OpenMP shared-memory programs into CUDA-based GPGPU programs. For an automatic source-to-source translation, several translation strategies have been developed, including a *kernel region* identifying algorithm. The proposed translation aims at offering an easier programming model for general computing on GPGPUs. By applying OpenMP as a front-end programming model, the proposed translator could convert the loop-level parallelism of the OpenMP programming model into the data parallelism of the CUDA programming model in a natural way; hence, OpenMP appears to be a good fit for GPGPUs. We have also identified several key transformation techniques to enable efficient GPU *global memory* access: *parallel loop-swap* and *matrix transpose* techniques for regular applications, and *loop collapsing* for irregular ones.

Experiments on both regular and irregular applications led to several findings. First, a baseline translation of existing OpenMP applications does not always yield good performance; hence, optimization techniques designed for traditional shared-memory multiprocessors do not translate directly onto GPU architectures. Second, efficient *global memory* access is one of the most important targets of GPU optimizations, but simple transformation techniques, such as the ones proposed in this paper, are effective in optimizing global memory accesses. Third, complex interaction among hardware resources may require fine tuning. While automatic tuning may deal with this problem, we believe that the performance achieved by the presented OpenMP-to-GPGPU translator comes close to hand-coded CUDA programming.

Our ongoing work focuses on transformation techniques for efficient GPU *global memory* access. Future work includes automatic tuning of optimizations to exploit *shared memory* and other special memory units more aggressively.

Acknowledgments

This work was supported, in part, by the National Science Foundation under grants No. 0429535-CCF, CNS-0751153, and 0833115-CCF.

References

[1] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[2] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. *ACM International Conference on Supercomputing (ICS)*, 2008.

[3] Ayon Basumallik and Rudolf Eigenmann. Towards automatic translation of OpenMP to MPI. *ACM International Conference on Supercomputing (ICS)*, pages 189–198, 2005.

[4] NVIDIA CUDA [online]. available: http://developer.nvidia.com/object/cuda_home.html.

[5] NVIDIA CUDA SDK - Data-Parallel Algorithms: Parallel Reduction [online]. available: http://developer.download.nvidia.com/compute/cuda/1_1/Website/Data-Parallel_Algorithms.html.

[6] Tim Davis. University of Florida Sparse Matrix Collection [online]. available: http://www.cise.ufl.edu/research/sparse/matrices/.

[7] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. *International Conference for High Performance Computing, Networking, Storage and Analysys (SC)*, 2006.

[8] Sang Ik Lee, Troy Johnson, and Rudolf Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2003.

[9] David Levine, David Callahan, and Jack Dongarra. A comparative study of automatic vectorizing compilers. *Parallel Computing*, 17, 1991.

[10] Seung-Jai Min, Ayon Basumallik, and Rudolf Eigenmann. Optimizing OpenMP programs on software distributed shared memory systems. *International Journel of Parallel Programming (IJPP)*, 31:225–249, June 2003.

[11] Seung-Jai Min and Rudolf Eigenmann. Optimizing irregular shared-memory applications for clusters. *ACM International Conference on Supercomputing (ICS)*, pages 256–265, 2008.

[12] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting OpenMP on Cell. *International Journel of Parallel Programming (IJPP)*, 36(3):289–311, June 2008.

[13] OpenMP [online]. available: http://openmp.org/wp/.

[14] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 73–82, 2008.

[15] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. Program optimization space pruning for a multithreaded GPU. *International Symposium on Code Generation and Optimization (CGO)*, 2008.

[16] J. A. Stratton, S. S. Stone, and W. W. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2008.

[17] Narayanan Sundaram, Anand Raghunathan, and Srimat T. Chakradhar. A framework for efficient and scalable execution of domain-specific templates on GPUs. *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2009.

[18] S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu. CUDA-lite: Reducing GPU programming complexity. *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2008.

[19] Haitao Wei and Junqing Yu. Mapping OpenMP to Cell: An effective compiler framework for heterogeneous multi-core chip. *International Workshop on OpenMP (IWOMP)*, 2007.

[20] Peng Wu, Alexandre E. Eichenberger, Amy Wang, and Peng Zhao. An integrated simdization framework using virtual vectors. *ACM International Conference on Supercomputing (ICS)*, pages 169–178, 2005.