OpenMP Optimization and its Translation to OpenGL

Santosh Kumar
SITRC-Nashik, India

Dr. V.M.Wadhai
MAE-Pune, India

Prasad S.Halgaonkar
MITCOE-Pune, India

Kiran P.Gaikwad
GHRIEC-Pune, India

ABSTRACT
For general purpose high-performance computing, recently GPGPUs have emerged as powerful vehicles. Programming GPGPUs is complex when compared to programming general purpose CPUs and parallel programming models such as OpenMP. Goal of our translation is to improve programmability and make existing OpenMP applications to be able to execute on GPGPUs. OpenMP has established itself as an important method and language extension for programming shared-memory parallel computers. Our translator works well on regular applications, leading to performance improvements of up to 50X over the un-optimized translation.

Keywords
OpenMP, GPU, Brook+, Automatic translation

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. Programming GPGPUs is still complex and error-prone, compared to programming general-purpose CPUs and parallel programming models such as OpenMP.

OpenMP [1] 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 GPUs highly parallel computing units to accelerate data parallel computations.
- The concept of a master thread and a pool of worker threads in OpenMPs fork-join model represent 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 OpenMPs features, can add the same benefit to GPGPU programming.
- Its programming complexity poses a significant challenge for developers, converting the source code to GPGPU.
- To convert OpenMP parallelism, especially loop-level parallelism, into the forms that optimal parallelism in GPGPU.
- The interpretation of OpenMP semantics under the GPGPU programming model.
- To extract regions to be executed on GPU's i.e. kernel function.
- Baseline translation of existing OpenMP programs does not always yield good performance.

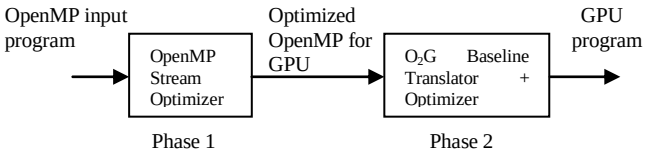

Figure 1: Two-phase OpenMP-to-GPGPU compilation system. Phase 1 is an OpenMP stream optimizer to generate optimized OpenMP programs for GPGPU architectures; phase 2 is an OpenMP-to-GPGPU (O_2G) translator

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.

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 O2G translation converts the output of the OpenMP stream optimizer into OpenGL GPGPU programs.

This paper makes the following contributions:

- Our framework is automatic source-to-source translation of standard OpenMP applications into OpenGL-based GPGPU applications. It includes (1) the interpretation of OpenMP semantics under the OpenGL programming model, (2) an algorithm to extract regions to be executed on GPUs, and (3) an algorithm for reducing CPU-GPU memory transfers.

- Identified the several compile-time transformation techniques to optimize GPU global memory access: parallel loop-swap and matrix transpose techniques for regular applications.

2. RELATED WORK

Prior to the advent of the CUDA programming model [2], 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 [3] 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 [4]. In these contributions, optimizations were performed manually.

For the automatic optimization of CUDA programs, a compile time transformation scheme [5] 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.

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 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 [6, 7] and another compile-time translation scheme to convert OpenMP programs into MPI message-passing programs for execution on distributed memory systems [8]. Recently, there have been several efforts to map OpenMP to Cell architectures [9, 10]. 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 [11] is an opposite approach, which maps the CUDA programming model onto 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 [12]. 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. 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 offchip memory performance. On the other hand, loop interchange in vectorizing compilers is to enable vectorization of certain loops within a single thread.

3. OVERVIEW OF THE BROOK+ PROGRAMMING MODEL

The Brook+ programming model is a general-purpose multi-threaded SIMD model for GPGPU programming. In the Brook+ programming model, a GPU is viewed as a parallel computing coprocessor, which can execute a large number of threads concurrently. A Brook+ program consists of a series of parallel execution phases. Parallel phases that exhibit rich data parallelism are implemented as a set of kernel functions, which are executed on the GPU. Brook GPU, as it normally called, is compiler and runtime implementation for General Purpose Stream Computing. With brook GPU, General purpose computing can be done on graphics card. Brook can be compiled for ATI-AMD graphics card as well as for nvidia card.

4. BASELINE TRANSLATION OF OPENMP INTO BROOK+

This section presents a baseline translator, which performs a source-to-source conversion of an OpenMP program to a Brook based GPGPU program. The translation consists of several steps: (1) interpreting OpenMP semantics under the Brook+ programming model and identifying kernel regions (code sections executed on the GPU), (2) outlining (extracting into subroutines) kernel regions and transforming them into kernel functions.

4.1. Interpretation of OpenMP Semantics under the Brook+ 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.
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.
4) Directives specifying data properties (omp shared, omp private, omp thread private, etc.) these constructs are used to map data into GPU memory spaces.

OpenMP shared data are shared by all threads and OpenMP private data are accessed by a single thread. In the GPU 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 thread-private data are private to each thread, but they have global lifetimes, as do static data. The semantics of thread-private data can be implemented by expansion, which allocates copies of the thread-private data on global memory for each thread. Because the GPU 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.

4.2. OpenMP to Brook Baseline Translation

The previous subsection described the interpretation of OpenMP semantics under the Brook programming model. The next step performs the actual translation into a brook 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.

4.2.1. 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.

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. OpenMP threads are more autonomous, typically execute coarse-grain parallelism, and are able to handle MIMD computation.

4.2.2. 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.

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 figure 2. Parallel region example shows 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.

The OpenMP schedule clause, each section in omp sections

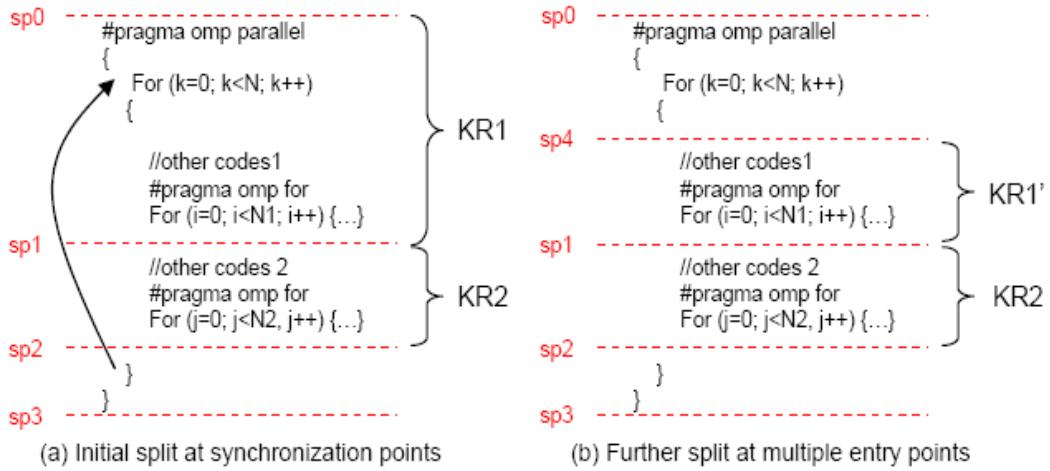

Figure 2. Parallel region example showing how multiple splits are applied to identify kernel regions. *sp0 - sp4* are split points enforced to preserve OpenMP semantics, and *KR1'* and *KR2* are kernel regions to be converted into kernel functions.

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 sub-region contained in the kernel region. Once the compiler figures out the total number of threads, the number of thread blocks is also 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. Shared data are mapped to global memory, thread-private data are replicated and allocated on global memory for each thread, and private data are mapped to register banks assigned for each thread.

As a part of the data mapping step, the compiler inserts necessary memory transfer calls for the shared and thread-private 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. (Thread-private 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 [1]: 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 [1] can be applied.

TABLE I
GPU AND OPENMP PROGRAM EXECUTION TIME OF MATRIX MULTIPLICATION

Size of matrix	**Execution time of GPU program**	**Execution time of openMP program**
256 by 256	0.960201	1.184087 sec
512 by 512	3.102356	8.817162 sec
256 by 512	1.099560	1.706772 sec
256 by 712	1.965801	4.103025 sec

5. PERFORMANCE EVALUATION

This section presents the performance of the presented OpenMP to GPGPU translator and compiler optimizations. In our experiments, regular OpenMP programs (Matrix Multiplication). The baseline translations were performed automatically by the compiler framework. **We used an ATI Fire GL V5600 GPU as an experimental platform. The device has a clock rate of 1.35 GHz and 512 MB 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 Brook programs with the Brook Compiler (brcc) to generate device code.

5.1. Performance of Regular Applications

Matrix Multiplication is a widely used kernel containing the main loop of an iterative solver for regular scientific applications. Due to its simple structure, the Matrix kernel is easily parallelized in many parallel programming models. Baseline in the figure represents the execution 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. These results demonstrate that, in regular programs, uncoalesced global memory accesses may be converted to coalesced accesses by loop transformation optimizations.

After making the comparison between serial and parallel, we have to make the comparison between OpenMp program and GPU (brook+) program. Table 1 shows the execution time of both the programs.

6. CONCLUSION

OpenMP appears to be a good fit for GPGPUs. It 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.

Our 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 OpenGL programming model in a natural way. Ongoing work focuses on transformation techniques for efficient GPU global memory access which includes automatic tuning of optimizations to exploit shared memory and other special memory units.

7. REFERENCES

[1] OpenMP [online]. available: http://openmp.org/wp/NVIDIA CUDA [online]. Available: http://developer.nvidia.com/object/cuda home.html

[2] 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.

[3] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, andW.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.

[4] 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.

[5] 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.

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

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

[8] 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.

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

[10] 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.

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