ISSN 2278-8808
An International Peer Reviewed
SCHOLARLY RESEARCH JOURNAL
FOR INTERDISCIPLINARY STUDIES

Parallel Computing Implementation Using GPU

Mohammad Naeemullah
Department of Computer Science
Maulana Azad College, Aurangabad

Received: 22 February 2013
Reviewed & Received: 28 February 2013
Accepted: 28February 2013

Abstract

A few years, the programmable graphics processor unit has evolved into an absolute High performance computing. Simple data-parallel constructs, enabling the use of the GPU as a streaming coprocessor. A compiler and runtime system that abstracts and virtualizes many aspects of graphics hardware.

Commodity graphics hardware has rapidly evolved from being a fixed-function pipeline into having programmable vertex and fragment processors. While this new programmability was introduced for real-time shading, it has been observed that these processors feature instruction sets general enough to perform computation beyond the domain of rendering.

Proposed research work is a translation of share memory program to graphics processing unit for regular loop and irregular loop in parallelism. The theme of this translation is to make the efficiency to reduce the execution time for the huge amount of data processing for such an application. An analysis of the effectiveness of the Graphics Processing Unit as a computing device compared to the Central processing Unit, to determine when the GPU can produce outstanding result rather than the CPU for a particular algorithm for Application. To achieve good performance, our translation scheme includes efficient management of shared data as well as advanced handling of irregular accesses.

Key Words: *Parallel Computing, Implementation, GPU.*

1. INTRODUCTION

In today's wired world, person doesn't want to spend more time to execute the application on computer. Everyone is interested to get the fast response from computer for this purpose evolution of High Performance Computing. Less execution time is need of society whenever large amount of data processing. There are number of situations where only certain less execution time are require to allowed to use particular application, such as Drawing image after processing of data.

GPUs have recently emerged as powerful platform for general purpose High-performance computing. Programming for GPU is still complex, compared to programming general-purpose. CPUs and parallel programming models such as share memory programming model. The goal of this conversion is to further reduce execution time and make existing share memory programming applications amenable to execution on GPUs. Share memory programming model. OpenMP[1] has established itself as an important method and language extension for programming shared-memory parallel computers. 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.

There are several advantages of share memory programming model as a programming paradigm for GPUs.

• Share memory programming model is efficient at expressing loop-level parallelism in applications, which is a 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 share memory programming model 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.

• Parallelization of applications, which is one of share memory programming model features, can add the same benefit to GPU programming model.

The GPU 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 GPU[14,16] architectures for better programmability, its unique memory architecture is exposed to programmers to some extent. Therefore, the manual development of high-performance codes in GPU programming model is more involved than in other parallel programming models such as share memory programming model.

In this Research, developed a CPU parallel computing to GPU parallel computing converter to extend the ease of creating parallel applications with share memory programming to GPU architectures. Due to the similarity between share memory programming and GPU programming models, we were able to convert hare memory parallelism, basically loop-level parallelism, into the forms that best express parallelism in GPU. Performance gaps are due to architectural differences between traditional shared-memory multiprocessors (SMP), implemented by share memory programming, and stream architectures, accepted by most GPU [14,6]. Most existing share memory programs were tuned to more efficient for fast access to regular, consecutive elements of the data stream.

2. RELATED WORK

GPU programming model, programming GPUs was very difficult, requiring deep knowledge of the underlying hardware and graphics programming interfaces. Although the GPU programming model provides improved programmability, achieving high performance with GPU parallel programs is still difficult. Several studies have been conducted to develop the performance of GPU applications. In these contributions, optimizations were performed manually.

For the automatic optimization of GPU 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, in propose research framework performs well on actual benchmarks as well as on GPU functions. GPU-lite [18] is another translator, which generates codes for optimal tiling of global memory data. GPU-lite relies on information that a programmer provides via annotations, to perform transformations. Our approach is similar to GPU-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 Open MP directives, and the annotations provided by a programmer are used for fine tuning.

Open MP 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 share memory programming model has been ported to a variety of platforms. Previously, we have developed compiler techniques to translate share memory 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 share memory programs into MPI message-passing programs for execution on distributed memory systems [3]. Recently,

there have been several efforts to map share memory to Cell architectures [12, 19]. Our approach is similar to the previous work in that share memory parallelism, specified by work-sharing constructs, is exploited to distribute work among participating threads or processes, and share memory 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.

MCUDA [16] is an opposite approach, which maps the CUDA programming model onto the conventional shared-memory CPU architecture. MCUDA can be used as a tool to apply the GPU programming model[8,9] 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 specification gap between domain-specific algorithms and current GPU programming models such as Brook, a framework for scalable execution of domain-specific templates on GPUs has been proposed. This research work is the problem of partitioning the computations that do not fit into GPU memory.

However, the architectural differences between GPU and vector systems [9] different challenges in applying these techniques, leading to different directions; *parallel loop exchange* and *loop overlapping* transformations are 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.

3. OVERVIEW OF THE CUDA PROGRAMMINGMODEL

CUDA stands for Compute Unified Device Architecture and is a new hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API. It is available for the GeForce 8 Series, the Tesla solutions, and some Quadro solutions. The operating system's multitasking mechanism is responsible for managing the access to the GPU by several CUDA and graphics applications running concurrently.

The CUDA software stack is composed of several a hardware driver, an application programming interface (API) and its runtime, and two higher-level mathematical libraries of common usage, CUFFT and CUBLAS that are both described in separate documents. The hardware has been designed to support lightweight driver and runtime layers, resulting in high performance.

CUDA provides general DRAM memory addressing for more programming flexibility: both scatter and gather memory operations. From a programming perspective, this translates into the ability to read and write data at any location in DRAM, just like on a CPU.

CUDA features a parallel data cache or on-chip shared memory with very fast general read and write access, that threads use to share data with each other. The applications can take advantage of it by minimizing over fetch and round-trips to DRAM and therefore becoming less dependent on DRAM memory bandwidth.

3.1 HARDWARE IMPLEMENTATION

3.1.1 A Set of SIMD Multiprocessors with On-Chip Shared Memory

The device is implemented as a set of multiprocessors, Each multiprocessor has a Single Instruction, Multiple Data architecture (SIMD): At any given clock cycle, each processor of the multiprocessor executes the same instruction, but operates on different data.

Each multiprocessor has on-chip memory of the four following types:

- One set of local 32-bit registers per processor,
- A parallel data cache or shared memory that is shared by all the processors and implements the shared memory space,
- A read-only constant cache that is shared by all the processors and speeds up reads from the constant memory space, which is implemented as a read-only region of device memory,
- A read-only texture cache that is shared by all the processors and speeds up reads from the texture memory space, which is implemented as a read-only region of device memory.

4. EXPERIMENTAL METHOD AND PROPOSED ALGORITHM

4.1 PROPOSED SYSTEM

This research introduces methods for transfer the load from Central processing Unit to Graphics processing unit for High Performance Computing. Parallel Computing is an area of High Performance Computing that has reduce the execution time of Algorithm. Parallel computing always produce the less execution time compare to serial computing.

To isolate CPU-intensive parallelization functionality into mostly independent logical threads, tasks, or jobs, so that each core or CPU can get its thread(s),spreading the overall load that load transfer from CPU to GPU.

A source to source transformation of share memory programming model to graphics processing unit programming model.

A growing demand for High performance computing is operation of huge amount of data processing. This work is to carry out the High Performance Computing using the parallel programming model (share memory programming model) iterated execution of individual thread on separate core, which is in GPU.

This Research work revolve on compilation system

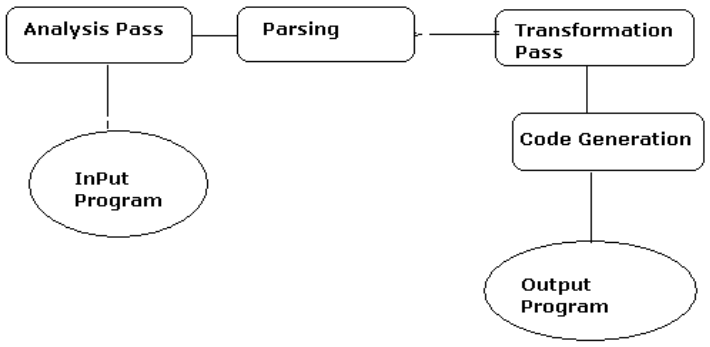

Figure: 1 Block Diagram of Transfer the source code

To bridge the specification gap between domain-specific algorithms and current GPU programming models such as Brook, a framework for scalable execution of domain-specific templates on GPUs has been proposed. This research work is the problem of partitioning the computations that do not fit into GPU memory.

However, the architectural differences between GPU and vector systems [9] different challenges in applying these techniques, leading to different directions; *parallel loop exchange* and *loop overlapping* transformations are 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.

4.2 PERFORMANCE EVALUATION

This section presents the performance of the presented share memory programming model to graphics processing unit. I have 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). compiled the translated CUDA programs with the NVIDIA CUDA Compiler (NVCC) to generate device code. Compiled the host programs with the GCC compiler version 4.2.3, using option -O3.

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 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 2 shows the execution time of both the programs.

5. RESULTS AND DISCUSSION

Table 1: GPU AND OPENMP PROGRAM EXECUTION TIME OF MATRIX MULTIPLICATION

Size of matrix	**Execution time of GPU program Before Memory Tuning**	**Execution time of GPU program After Memory Tuning**
256 by 256	1.02	1.184087 sec
256 by 512	2.01	1.706772 sec
256 by 712	2.77	4.103025 sec
512 by 512	3.99	8.817162 sec
1024 by 1024	16.03	26.23562 sec
2048 by 2048	29.15	56.3621 sec

Table 2: GPU AND OPENMP PROGRAM EXECUTION TIME OF MATRIX MULTIPLICATION AFTER MEMORY TUNING

Size of matrix	**Execution time of GPU program**	**Execution time of OpenM program**
256 by 256	0.960201	1.184087 sec
256 by 512	1.099560	1.706772 sec
256 by 712	1.965801	4.103025 sec
512 by 512	3.102356	8.817162 sec
1024 by 1024	12.03	26.23562 sec
2048 by 2048	20.15	56.3621 sec

Table: 3 COMPARISION OF GPU EXECUTION TIME OF MATRIX MULTIPLICATION BEFORE AND AFTER MEMORY TUNNING

Size of matrix	**Execution time of GPU program Before Memory Tuning**	**Execution time of GPU program After Memory Tuning**
256 by 256	1.02	0.960201
256 by 512	2.01	1.099560
256 by 712	2.77	1.965801
512 by 512	3.99	3.102356
1024 by 1024	16.03	12.03
2048 by 2048	29.15	20.15

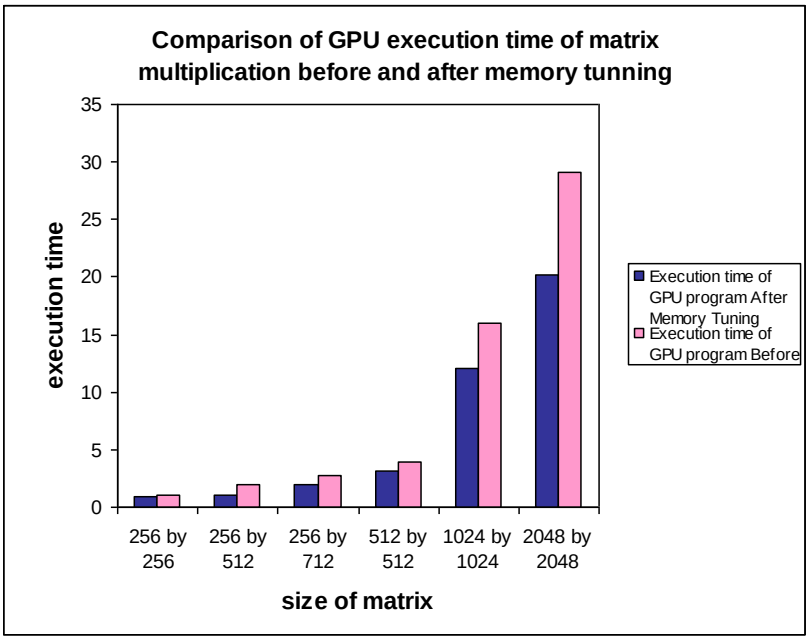

Figure 2 Shows Comparison of GPU execution time of matrix multiplication before and after memory tuning

The table 3 shows the information about comparison between the memory of GPU program, while execution is going on. In which the execution time reduces after tuning the memory operation (data transmission of GPU to CPU).

The matrix multiplication program has been executed before and after memory tuning the significant time difference has been observed and noted. The matrix multiplication program of size 256 by 256 has been executed before memory tuning the time taken is 1.02 and after memory tuning execution time noted is 0.960201. The execution time for matrix multiplication of size 256 by 512 before memory tuning is 2.01 and after memory tuning it has noted 1.099560. The execution time for matrix multiplication of size 256 by 712 before memory tuning is 2.77 and after memory tuning it has noted 1.965801. The execution time for matrix multiplication of size 512 by 512 before memory tuning is 3.99 and after memory tuning it has noted 3.102356. The execution time for matrix multiplication of size 1024 by 1024 before memory tuning is 16.03 and after memory tuning it has noted 12.03. The execution time for matrix multiplication of size 2048 by 2048 before memory tuning is 29.15 and after memory tuning it has noted 20.15.

The significant difference has been observed in execution time before and after memory tuning. As the size of matrix increases the execution time difference before and after memory tuning increases. The execution time observed after memory tuning is less than before memory tuning.

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. 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.

Translating standard OpenMP shared-memory programs into MPI message passing variants, based on a novel model of partial replication. The contributions include a new algorithm to compute message sets, techniques for statically handling irregular accesses, and optimizations based on collective communication. Partial replication contrasts with data distribution models, as used in HPF-like languages, in which a single node owns a (partition of a) shared array. Partial replication simplifies the generation of messages, which holds for irregular accesses. Exploiting monotonicity properties of index arrays, our techniques were able to handle all message generation for irregular accesses in our program suite statically.

REFERENCES

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

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. 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.

4. Seung-Jai Min, Ayon Basumallik, and Rudolf Eigenmann. Optimizing OpenMP programs on software distributed shared memory systems. International Journel of Parallel Programming (IJPP), 31:225249, June 2003.

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

6. Data-Parallel Algorithms: Parallel Reduction [online]. available: http://developer.download.nvidia.com/cuda/1 1/Website/Data-Parallel Algorithms.html.
7. ATI, 2004. Hardware image processing using ARB fragment program. http://www.ati.com/developer/sdk/RadeonSDK/Html/Samples/ OpenGL/HW Image Processing.html.
8. Brook, 2004. Brook project web page. http://brook.sourceforge.net
9. NVIDIA CUDA SDK - Data-Parallel Algorithms: Parallel Reduction [online]. available: http://developer.download.nvidia.com/compute/ cuda/1 1/Website/Data-Parallel Algorithms.html.
10. Tim Davis. University of Florida Sparse Matrix Collection [online] available: http://www.cise.ufl.edu/research/sparse/matrices/.
11. 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.
12. David Levine, David Callahan, and Jack Dongarra. A comparative study of automatic vectorizing compilers. Parallel Computing, 17, 1991.
13. 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.
14. ATI, 2004. Hardware image processing using ARB fragment program.

    http://www.ati.com/developer/sdk/RadeonSDK/Html/Samples OpenGL/HW Image Processing.html.
15. England, N. 1986. A graphics system architecture for interactive application-specic display functions. In IEEE CGA, 60-70.
16. Fuchs, H., Poulton, J., Eyles, J., Greer, T., Gold- feather, J., Ellsworth, D., Molnar, S., Turk, G., Tebbs, B., and Israel, L. 1989. Pixel-Planes 5: a heterogeneous multiprocessor graphics system using processorenhanced memories. In Computer Graphics (Proceedings of ACM SIGGRAPH 89), ACM Press, 79 – 88.
17. Intel, 2004. Intel math kernel library. http://www.intel.com/software/products/mkl
18. J. Li andM. Chen. Compiling communication-efficient programs for massively parallel machines. IEEE Trans. Parallel Distrib. Syst., 2(3):361–376, 1991.
19. P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. IEEE Transactions on Parallel and Distributed Systems, 2(3):350–360, 1991.