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

AN EVOLUTIONARY APPROACH TO PARALLEL COMPUTING USING GPU

Mohammad Naeemullah
Maulana Azad College of Arts Science & Commerce
Rauza Bagh, Aurangabad

Received: 20 August 2012
Accepted: 28 August 2012

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 run time 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 them of this translation is to make the efficient for reduce the execution time for the huge amount of data processing for such a 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.

Keywords: *Share Memory Programming Model, GPU, Parallel Computation, Load transfer*

INTRODUCTION

In today’s wired world, person don'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 .

GPU have recently Known as general purpose High-performance computing Component . Programming for GPU is to difficult ,compared to programming general-purpose CPU 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 GPU. 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 benefits of share memory programming models a programming paradigm for GPU.

• Share memory programming mode is efficient at expressing loop-level parallelism in applications, which is an target for utilizing GPU is efficiently highly parallel computing units for data parallel computations.

• The concept of a leader thread and a group of flower threads in share memory programming model fork-join model represents well the relationship between the leader thread running in a CPU and a group 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 an 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[14,16].

Most existing share memory programs were tuned to more efficient 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, GPU having big problem in handling irregular applications than SMPs, because of the stream architectures' preference for regular access patterns.

RELATED WORK

GPU programming model , programming GPU 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 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 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 a 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 e xchange* 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.

PROBLEM STATEMENT

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 to 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. In 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 work revolve on totally based compilation system architecture. Our proposed block diagram shows that how it works.

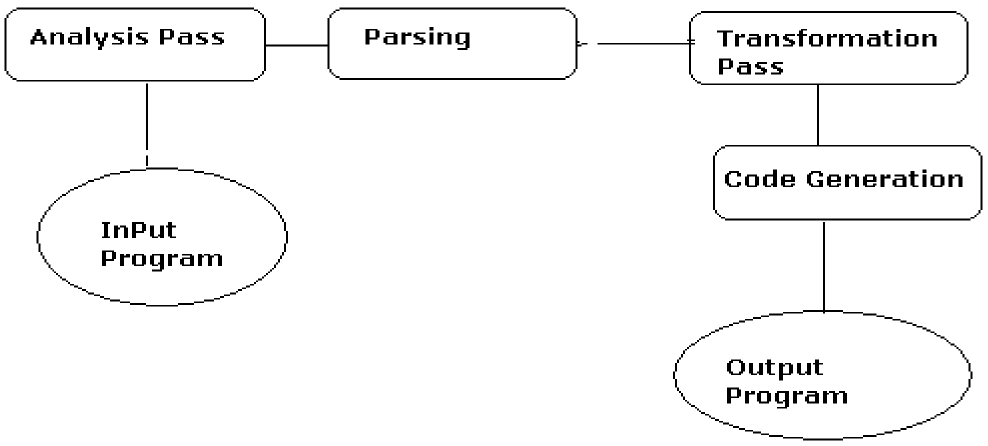


Fig 1: ***Block Diagram of Transfer the source code***

For application such as large data processing , it would be useful to have a reduce the execution time for getting the result as fast as possible compare to serial executation. A few years, the programmable graphics processor unit has evolved into an absolute High performance computing. Share memory programming is an Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for developers of shared memory parallel applications.

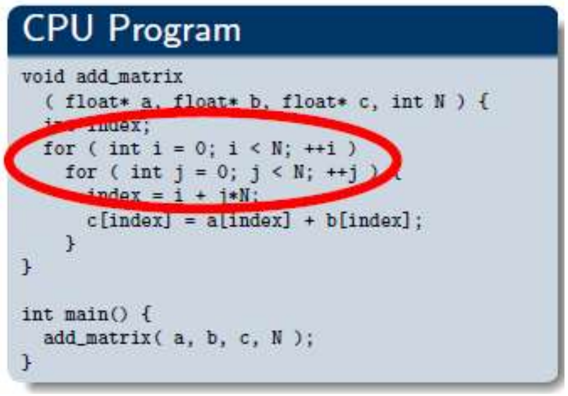
Geared to those who are new to parallel programming with Share memory programming. Basic understanding of parallel programming in C assumed. For those who are unfamiliar with Parallel Programming in general, they have to use GPU as a parallel programming.

Share memory programming provides a standard among a variety of shared memory architectures/platforms. Its very difficult to write a program for graphics processing unit for that purpose , our proposed research work helpful for use the GPU.

PROBLEM SOLUTION

Solution of our problem statement ,we have uses following steps. , method based on compilation system. More focus on loop level parallelism and data parallelism. For data parallelism we have to use Array privatization concept, because OpenMP is share memory programming model.

Figure 1 shows the example of CPU source code which is execute on CPU , Bu

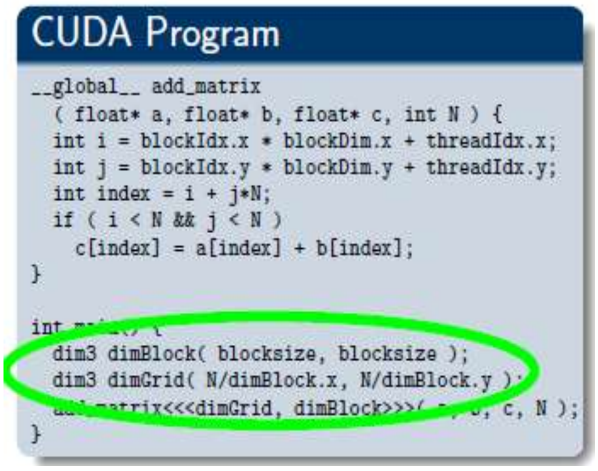


CPU Program

```
void add_matrix
  ( float* a, float* b, float* c, int N ) {
  int index;
  for ( int i = 0; i < N; ++i )
    for ( int j = 0; j < N; ++j ) {
      index = i + j*N;
      c[index] = a[index] + b[index];
    }
}

int main() {
  add_matrix( a, b, c, N );
}
```

Fig 1: Example of CPU source code



```
__global__ add_matrix
  ( float* a, float* b, float* c, int N ) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}

int main() {
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

Fig 2: Example of GPU source code

CONCLUSION

Translating standard OpenMP programs into OpenGL based GPGPU programs. The proposed translation aims at offering an easier programming model for general computing on GPGPU. 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.Efficient global memory access is one of the most important targets of GPU optimizations, but simple transformation techniquesare effective in optimizing global memory accesses. 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; hence, OpenMP appears to be a good t for GPGPUs. Future 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.

References

1. Open MP [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 PerformanceComputing, 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.