Data parallel loop statement extension to CUDA: GpuC

Zeki Bozkus[1], Rajeev Thakur[2], William Gropp[3], Ewing Lusk[2]
[1] Department of Computer Engineering
Kadir Has University, Istanbul, 34083 Turkey
[2]Matematics and Computer Science Division
Argonne National Laboratory, Argonne IL 60439 USA
[3]Department of Computer
University of Illinois, Urbana, IL 61801 USA

***Abstract*—In recent years, Graphics Processing Units (GPUs) have emerged as a powerful accelerator for general-purpose computations. GPUs are attached to every modern desktop and laptop host CPU as graphics accelerators. GPUs have over a hundred cores with lots of parallelism. Initially, they were used only for graphics applications such as image processing and video games. However, many other applications are starting to be ported to GPUs to extend the power of the GPU beyond graphics. Current approaches to program GPUs are still relatively low-level programming models such as Compute Unified Device Architecture (CUDA), a programming model from NVIDIA, and Open Compute Language (OpenCL), created by Apple in cooperation with others. These two programming models have all the complexity of parallel programming such as breaking up the task into smaller tasks, assigning the smaller tasks to multiple CPUs to work on simultaneously, and coordinating the CPUs. There is a growing need to lower the complexity of programming these devices. In this paper, we propose a data-parallel loop (f*orall*) extension to the CUDA programming model. We describe our prototype compiler named GpuC. The compiler takes data-parallel f*orall* loops along with the other CUDA statements as input and generates CUDA code as output. We present compilation steps, optimizations, and code generations. We identified several key optimizations for the compiler. We present experimental results from four NAS benchmarks to show performance gains.**

Keywords- GPGPU, Data Parallel, CUDA, Compiler Optimizations

I. INTRODUCTION

Single CPU speeds are slowing down because of physical limits of light waves used in the chip-manufacturing process. Every major microprocessor manufacturer has introduced processor chips with multiple processor cores for desktops and laptops. Over a hundred cores are available in some **G**raphics **P**rocessing **U**nits (GPUs). The common belief is that the numbers of cores per chip will double every two years while processor clock speeds will remain relatively flat. This makes parallel programming a concern for the entire computer industry. However, parallel programming is an extremely difficult task. In fact, it adds extra complexities over regular sequential programming, such as breaking up the task into smaller tasks, assigning the smaller tasks to multiple CPUs to work on simultaneously and coordinating the CPUs.

There have been multiple attempts to make parallel programming available to a wider range of programmers for different parallel architectures. Shared-memory parallel architectures have two different programming models. The first model is a low-level programming model with a standardized thread library such as Portable Operating System Interface (POSIX) threads [1]. The second model is a high-level programming model with Open Multi-Processing (OpenMP) [2], which consists of a set of compiler directives, library routines, and environment variables that help programmers write parallel programs at a higher level of abstraction. The other alternative is to use Message Passing Interface (MPI) [3], which enables many processes of a parallel program to communicate with one another. All of these developments belong to the early generation of parallel machines. New parallel platforms such as GPUs need similar tools to make them convenient to program.

In recent years, GPUs have emerged as a powerful accelerator for general purpose computations. GPUs are attached to every modern desktop and laptop host CPU as graphics accelerators. GPUs have over a hundred cores with lots of parallelism. Initially, they were used only for graphics applications such as image processing and video games. However, many other applications are starting to be ported to GPUs to extend the power of the GPU beyond graphics. This is due to the high computational power, easy availability, and low cost of GPUs. Current approaches to program GPUs are still low-level programming models, such as **C**ompute **U**nified **D**evice **A**rchitecture (CUDA) [4], a programming model from NVIDIA and **O**pen **C**ompute **L**anguage (OpenCL) [5], created by Apple in cooperation with others. These two programming models have all the complexity of parallel programming, such as breaking up the task into smaller tasks, assigning the smaller tasks to multiple CPUs to work on simultaneously, and coordinating the CPUs. There is a growing need to lower the complexity of programming these devices. In this paper, we present our prototype compiler, GpuC, which takes data-parallel loops along with other CUDA code, and generates CUDA code that the NVCC compiler from NVIDIA can translate into code for both GPU and host CPU. We combine a high-level parallel construct, namely, a *forall* loop with no data dependency, with low-level CUDA programming. We present our compiler technology for compiling *forall* loops for the GPU architecture. We believe compilers are very good for generating code for *forall* loops and keeping all the data transfers and loop-scheduling details away from programmers. But, at the same time, we want to retain the relatively low-level CUDA programming for the experienced programmers to have better control for more complex applications. Basically, we extend CUDA with *forall* loops. Programmers can use high-level and low-level constructs in the same CUDA programming model. We believe this combination presents an iterative parallelization method for parallelizing an application for GPUs. For example, a programmer will first identify the *forall* loops as a low-hanging fruits and then parallelize the rest with CUDA language. This iterative approach will increase the widespread adoption of CUDA as a successful parallel programming environment.

We used GCC, which is an open-source compiler that is used by many researchers as the platform for implementing application-specific compilers. We added an extension to GCC to accept *forall* loops and the other CUDA constructs. *Forall* loop computations are more suitable for the Single Instruction Multiple Data (SIMD) model of the GPU architecture.

This work will contribute as follows:

1. The GPU architecture has a memory hierarchy comprising global memory reachable from each core of the GPU, local shared memory reachable from a group of threads, and registers for each thread. This paper investigates compiler optimizations to use the memory hierarchy efficiently by improving data locality.
2. A data-parallel loop has no data dependency. Each iteration of the loop can be executed concurrently. This paper investigates loop optimizations, such as *loop unrolling*, *loop swapping*, and *loop peeling*, for better performance.
3. GPU kernel calls are asynchronous. After a kernel launch, the control immediately returns to the host CPU. This paper will explore the optimizations to overlap computation on the host and GPU device to get better performance. Depending on the amount of time the GPU computation takes to complete, it is possible for both host and the GPU device to compute concurrently.
4. Once data is moved to the GPU global memory, there are several alternatives to assign computations to threads for each instance of the data-parallel loop statement. This paper investigates algorithms for the distribution of computation among GPU threads.

The compiler performs source-to-source translation from CUDA with data-parallel loops (*forall*) to CUDA C language for NVIDIA GPU. This programming model will make GPU programming accessible and allow many real-world applications that are easily implemented on GPUs to run significantly faster than on the regular multi-core systems without GPU. The results will help many other interdisciplinary applications such as molecular simulations, computational chemistry, and medical imaging.

II. OVERVIEW OF THE COMPILER

A data-parallel loop has no data dependency. Each iteration of the loop can be executed concurrently. The goal of this prototype compiler is to hide most of the hardware details from the programmers when they need to use very common *forall* loops. The current CUDA program has to pay attention to two different memories, one for GPU memory and one for CPU memory. Our proposed model is to use *forall* loops along with other CUDA statements. These *forall* loops can be marked with *forall* keywords as part of the syntax, or the programmer can mark those loops as data-parallel loop with a compiler directive just like OpenMP. Our GpuC compiler will hide all the hardware detail to the programmer for data-parallel *forall* loops as shown in the Figure 2.

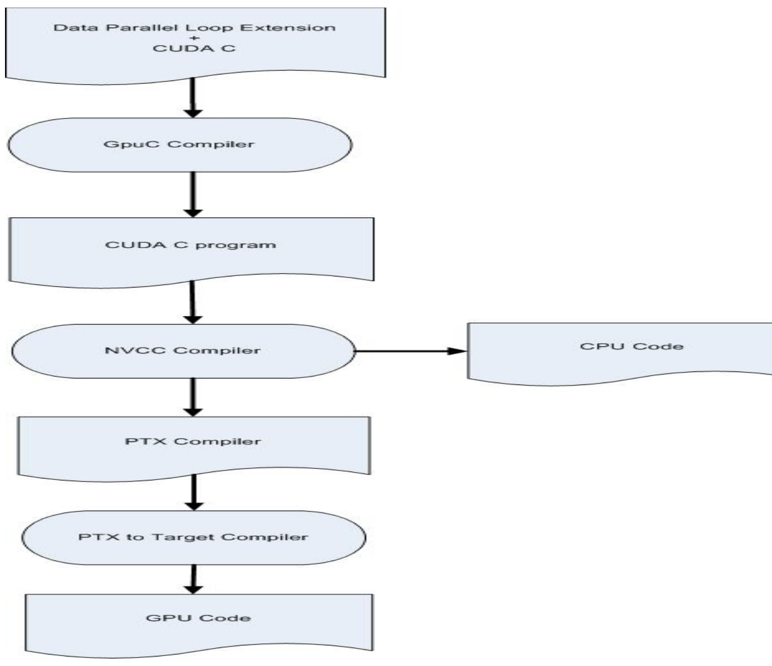

Figure 2: The prototype compiler flow chart

GpuC translates data-parallel loop to CUDA program. NVCC compiler from NVIDIA will take the CUDA program and generate the machine code for the parallel GPU and generate sequential code for the CPU.

The prototype compiler uses GCC, which is an open-source compiler that is used by many researchers as a platform to implement application-specific compilers. The compiler translates data-parallel *forall* loop to the CUDA program. The project will investigate many loop optimizations for a better code-generation phase.

The compiler will exploit the memory hierarchy of GPUs to improve data locality. This project will explore the optimizations to overlap computation on the CPU host and the GPU device to get better performance. The project will gather a lot of experimental results that will be useful to a commercial data-parallel compiler for GPUs.

A data-parallel loop has no data dependency. Each iteration of the loop can be executed concurrently. The goal of this prototype compiler is to hide most of the hardware details from the programmers when they need to use very common *forall* loops. The current CUDA program has to pay attention to two different memories, one for GPU memory and one for CPU memory. Our proposed model is to use *forall* loops along with other CUDA statements. These *forall* loops can be marked with *forall* keywords as part of the syntax, or the programmer can mark those loops as data-parallel loop with a compiler directive just like OpenMP. Our GpuC compiler will hide all the hardware detail to the programmer for data-parallel *forall* loops as shown in the Figure 2.

III. FORALL STATEMENT

High Performance Fortran [9] introduced a statement *forall* as an alternative to the DO-loop. The intent with the *forall* statement is that its content can be executed in any order, independent of the index. It therefore gives the possibility of a parallel implementation. GpuC has adapted a similiar forall statement with C style syntax shown at Figure 3.a. However, some rules are introduced in order to avoid side effects such as function calls inside forall cannot be recursive and cannot have side effects on global data or on its arguments. Figure 3.b shows a simple example on forall which inverts the elements of a matrix, avoiding division with zero.

```
      forall ( index0 = index0_expression1;
               index0_expression2,
               index1 = index1_expression1;
               index1_expression2,
                             ...
               indexN = indexN_expression1;
               indexN_expression2)
                        statements;

                 (a)forall statement syntax

  forall ( int i = 0; i < N; i++, int j = 0; j
                  < N; j++)  {
             if( x[i][j] !=  0) X[i][j] = 1.0 /
                    Y[i][j];
                        }
                (b) an simple forall example
```

Figure 3 Data parallel forall statements

GpuC tries to execute forall statements on GPU device by efficent parallel implementation . If it can not execute on GPU for some reasons, the compiler inform the user with a warning attached with a reason at compile time. In this sequential execution case, the compiler replace the forall with a series of regular *for* statements. The forall can have a scalar variables at the left hand side assignment. GpuC can anaylze these cases and recgonize the reduction pattern which will be explanied the more detail on later section.

The compiler is free to execute the iteration space with any order. Block forall statemet is interpreted assentially as a series of single statement forall. The values of forall index variables can not be used after the termination of forall.

REFERENCES

[1] Scott J. Norton, Mark D. Dipasquale. TREADTIME: The Multithread Programming Guide. 1997. Prentice Hall.

[2] OpenMP online. Available: http://openmp.org

[3] Message Passing Interface Forum. MPI-2 Extensions to the Message-Passing Interface, July 1997. Available: http://www.mpi-forum.org/docs/docs.html.

[4] NVIDIA CUDA online. Available: www.developer.nvidia.com/object/cuda.home.html.

[5] OpenCL online. Available: http://www.khronos.org/opencl/

[6] Michael Wolfe at the Portland Group. A GPU and Accelerator Programming Model. Available www.pgroup.com.

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

[8] Seyong Lee, Seung-Jai Min and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming.* Pages 101-110, 2009.