Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 352–361

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

APTCC : Auto Parallelizing Translator From C To CUDA

Takehiko Nawata[a,*], Reiji Suda[a,b]

[a]*Department of Computer Science, Graduate School of Information Science, the University of Tokyo*
[b]*Japan Science and Technology Agancy, Core Research for Evolutional Science and Technology*

Abstract

This paper proposes APTCC, Auto Parallelizing Translator from C to CUDA, a translator from C code to CUDA C without any directives. CUDA C is a programming language for general purpose GPU (GPGPU). CUDA C requires us a special programming manner differently from C. Although there are several pieces of research to reduce this difficulty, the result of those researches still compels us to beware of GPU architecture. It is better however that we are able to concentrate on the algorithm. Hence we propose translation of C code into CUDA C optimized to the target GPU architecture without directives, where the complexity of the GPU hardware is transparent to the programmer.

In translating a C code to a CUDA C code, two questions have to be answered. The first question is how to select the code fragments which should be translated into CUDA C, and the second question is how to translate the selected code fragments into CUDA C. To the first question, this paper proposes a heuristic selection scheme based on the loop structure of the source code. The current implementation of APTCC selects nested loops for the target of translation. To the second question, APTCC translate all the statements in the body of outermost loop into CUDA C.

This paper explains the detailed implementation of APTCC and compares the results of performance comparison of a few experimental input source codes.

Keywords: GPGPU, CUDA, Translation, Auto-Parallelization

1. Introduction

In recent years, GPU comes to be used for general computation rather than only for graphics processing. Because the GPU is designed to process a huge amount of pixels in a short time, it is suitable for applying the same sequence of operations to a large number of data. Stencil computation is a typical example of this kind of computations from the areas other than graphics processing [1].

Although in old days there were no standard language to control GPU, a C-extension language for general purpose GPU (GPGPU) called CUDA C has been released by NVIDIA in 2007[2]. CUDA stands for Compute Unified Device Architecture. CUDA C is not the first high-level language to control GPU. Direct3D has a high-level programming language called High Level Shader Language (HLSL)[3] to write shader programs and HLSL became available from DirectX3D versions DirectX9 and later. OpenGL also has a high-level shader language called OpenGL Shading Language (GLSL)[4]. Cg [5] is a high-level language to write shader programs which was offered by NVIDIA before

*Corresponding author
Email addresses: samugari@is.s.u-tokyo.ac.jp (Takehiko Nawata), reiji@is.s.u-tokyo.ac.jp (Reiji Suda)

doi:10.1016/j.procs.2011.04.037

CUDA C. General purpose programming were tried with those shader languages, but it was required to program with strong restrictions and tricky usage of Graphics API for general purpose computing. Comparing with these high-level languages, CUDA C is a language for GPGPU. There is a new specification of a programming language for heterogeneous programming called OpenCL[6]. GPU is included as one of the target of OpenCL. The core programming language of OpenCL is called OpenCL C and it is similar to CUDA C. But CUDA C is the only target of this paper.

To learn and to write CUDA C is easy because CUDA C is a simple extension of C language. However, even using CUDA C, getting good performance — it must be the main reason for using GPUs for general purpose computing — is still difficult. The cause for this difficulty is the fact that CUDA C is too close to GPU hardware. We are forced to consider about the hardware implementation of GPU whenever we write high performance programs using CUDA C. We want to make it easier to get a CUDA C code that achieves good performance.

This paper proposes a translator named APTCC which takes C language source codes as inputs and outputs a CUDA C code which carries out the same calculations on GPU. In our approach, the knowledge of characteristics of GPU is included to APTCC and APTCC outputs the optimized CUDA C code based on that knowledge. We selected the programming language C to write source code because it is widely used.

There are many other approaches to make CUDA C easier: Wrapping typical functions into a library[7, 8], making a new language[9], defining a Domain Specific Language[10, 11], directive-based approach[12] and so on. Some are for writing CUDA C easier and some are for getting good performance CUDA C code easier. Some of these approaches are not targeted to only at CUDA C but also at OpenCL.

When we compare APTCC with these researches, the biggest advantage of APTCC is the elimination of any additions to the C language. APTCC does not introduce any new function, syntax, class and directive. APTCC accepts the standard C code and translates it into CUDA C code.

In this paper, the preliminary implementation of APTCC is discussed. Section 2 is for brief introduction of CUDA C and in section 3 we outline our proposal. The detail of current APTCC implementation is explained in section 4, and the performance of this preliminary implementation is evaluated in section 5.

2. CUDA C

This section is for brief introduction of CUDA C. To get more detailed information, programming guide of CUDA C[13] issued by NVIDIA is a good reference.

The syntax of CUDA C is almost same as C language but a few specifiers are added. In a program written in CUDA C, we call the CPU “host” and the GPU “device.” To use one GPU, we have to write the data transfer codes and the definition of functions that are carried out computations on GPU. First the memory on the “device” must be allocated and the data for succeeding calculation on the “device” must be sent to the “device.” The calculation on the device must be expressed as a function whose definition is given with the special specifier `__global__`. This function is called “kernel” and the “device” executes the “kernel” when the “host” invokes it. A GPU has several Multi Processors (MPs) and each MP has many Stream Processors (SPs). In the the latest product GTX580, the numbers of MP and SP is 16 and 32, respectively.

“Block” and “thread” in CUDA C are the abstraction of MP and SP, respectively. When the “host” invokes a “kernel,” the “host” must specify the number of “block” and the “threads” per “block.” One “block” runs on one MP and one “thread” runs on one SP. From the latest architecture called Fermi, we can invoke another “kernel” before preceding “kernel” invocation finishes.

The parallelism is important to get a good performance in CUDA C. At least, 16 “blocks” for fully use of the MPs and 256 “threads” for hiding the latency of memory accesses are required according to the recommendation from NVIDIA. SPs that belongs to the same MP can cooperate by taking the synchronization and sharing data through the small, high-speed MP-local memory called shared memory. Before Fermi architecture comes out, coalescing the memory access pattern and timing, and using the shared memory efficiently were recognized as the most important factors to get the good performance. However, nowadays importance of these techniques are declining because Fermi architecture has cache memory and the hardware for memory access has been improved.