MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs

John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu
Center for Reliable and High-Performance Computing and
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{stratton,ssstone2,hwu}@crhc.uiuc.edu

Abstract. CUDA is a data parallel programming model that supports several key abstractions - thread blocks, hierarchical memory and barrier synchronization - for writing applications. This model has proven effective in programming GPUs. In this paper we describe a framework called MCUDA, which allows CUDA programs to be executed efficiently on shared memory, multi-core CPUs. Our framework consists of a set of source-level compiler transformations and a runtime system for parallel execution. Preserving program semantics, the compiler transforms threaded SPMD functions into explicit loops, performs fission to eliminate barrier synchronizations, and converts scalar references to thread-local data to replicated vector references. We describe an implementation of this framework and demonstrate performance approaching that achievable from manually parallelized and optimized C code. With these results, we argue that CUDA can be an effective data-parallel programming model for more than just GPU architectures.

1 Introduction

In February of 2007, NVIDIA released the CUDA programming model for use with their GPUs to make them available for general purpose application programming [1]. However, the adoption of the CUDA programming model has been limited to those programmers willing to write specialized code that only executes on certain GPU devices. This is undesirable, as programmers who have invested the effort to write a general-purpose application in a data-parallel programming language for a GPU should not have to make an entirely separate programming effort to effectively parallelize the application across multiple CPU cores.

One might argue that CUDA’s exposure of specialized GPU features limits the efficient execution of CUDA kernels to GPUs. For example, in a typical usage case of the CUDA programming model, programmers specify hundreds to thousands of small, simultaneously active threads to achieve full utilization of GPU execution resources. However, a current CPU architecture currently supports only up to tens of active thread contexts. On the other hand, some language features in the CUDA model can be beneficial to performance on a CPU, because these features encourage the programmer to use more disciplined
control flow and expose data locality. Section 2 describes in more detail the key CUDA language features and a deeper assessment of why we expect many CUDA features to map well to a CPU architecture for execution. We propose that if an effective mapping of the CUDA programming model to a CPU architecture is feasible, it would entail translations applied to a CUDA program such that the limiting features of the programming model are removed or mitigated, while the beneficial features remain exposed when possible.

Section 3 describes how the MCUDA system translates a CUDA program into an efficient parallel C program. Groups of individual CUDA threads are collected into a single CPU thread while still obeying the scheduling restrictions of barrier synchronization points within the CUDA program. The data locality and regular control encouraged by the CUDA programming model are maintained through the translation, making the resulting C program well suited for a CPU architecture.

The implementation and experimental evaluation of the MCUDA system is presented in Section 4. Our experiments show that optimized CUDA kernels utilizing MCUDA achieve near-perfect scaling with the number of CPU cores, and performance comparable to hand-optimized multithreaded C programs. We conclude this paper with a discussion of related work in Section 5 and some closing observations in Section 6.

2 Programming Model Background

On the surface, most features included in the CUDA programming model seem relevant only to a specific GPU architecture. The primary parallel construct is a data-parallel, SPMD kernel function. A kernel function invocation explicitly creates many CUDA threads (hereafter referred to as logical threads.) The threads are organized into multidimensional arrays that can synchronize and quickly share data, called thread blocks. These thread blocks are further grouped into another multidimensional array called a grid. Logical threads within a block are distinguished by an implicitly defined variable threadIdx, while blocks within a grid are similarly distinguished by the implicit variable blockIdx. At a kernel invocation, the programmer uses language extensions to specify runtime values for each dimension of threads in a thread block and each dimension of thread blocks in the grid, accessible within the kernel function through the variables blockDim and gridDim respectively. In the GPU architecture, these independent thread blocks are dynamically assigned to parallel processing units, where the logical threads are instantiated by hardware threading mechanisms and executed.

Logical threads within CUDA thread blocks may have fine-grained execution ordering constraints imposed by the programmer through barrier synchronization intrinsics. Frequent fine-grained synchronization and data sharing between potentially hundreds of threads is a pattern in which CPU architectures typically do not achieve good performance. However, the CUDA programming model does restrict barrier synchronization to within thread blocks, while different thread blocks can be executed in parallel without ordering constraints.