A Study of Persistent Threads Style Processing on the GPU

Kshitij Gupta Jeff A. Stuart John D. Owens
University of California, Davis

1 Introduction

GPU computing applications are built around a data-parallel programming model (PM) that cleanly separates the concept of how a software thread maps to hardware. Typical GPU computing applications launch a large number of lightweight threads, grouped into software blocks, which are then efficiently mapped onto the hardware's compute resources. While this PM is well-suited to implement algorithms with regular structure having fixed compute and memory access patterns, we believe that more complex, poorly structured workloads that are irregular in nature are a poorer fit.

2 Overview

In this poster we will present an analysis of the usefulness of a style of programming for the GPU called Persistent Threads (PT). Although the concept of PT itself is not new with multiple research efforts utilizing it for accelerating irregular workloads, the topic lacks a formal treatment. In this poster we will provide a formal definition for PT, and distinctly categorize all usage scenarios we came across in literature into four *use case* categories. We being by identifying limitations of the current GPGPU programming model. We then present two important distinctions of how PT style programming is different from the traditional (CUDA/OpenCL) approach. We then present a concise table of our use cases, under what circumstances they would be used, and what their advantages are, followed by results of our experiments from each use case. Finally, we conclude by identifying the changes that would be required for extending native support for PT style programming.

Use Case	Scenario	Advantage of Persistent Threads
CPU-GPU Synchronization	Kernel A produces a variable amount of data that must be consumed by Kernel B	nonPT implementations require a round-trip communication to the host to launch Kernel B with the exact number of blocks corresponding to work items produced by Kernel A.
Load Balancing	Traversing an irregularly-structured, hierarchical data structure	PT implementations build an efficient queue to allow a single kernel to produce a variable amount of output per thread and load balance those outputs onto threads for further processing.
Maintaining Active State	A kernel accumulates a single value across a large number of threads, or Kernel A wants to pass data to Kernel B through shared memory or registers	Because a PT kernel processes many more items per block than a nonPT kernel, it can effectively leverage shared memory across a larger block size for an application like a global reduction.
Global Synchronization	Global synchronization within a kernel across workgroups	In a nonPT kernel, synchronizing across blocks within a kernel is not possible because blocks run to completion and cannot wait for blocks that have not yet been scheduled. The PT model ensures that all blocks are resident and thus allows global synchronization.

Table 1: *Summary of Persistent Threads Use Cases*

3 Looking Forward

GPU computing is still a nascent field, providing speedup obtained previously only by supercomputers. While GPU architectures have made steady strides since the adoption of unified shaders for supporting GPGPU, there is a long list of potential advancements from the CPU world that may be required to unlock its full potential. Support for features like recursion, memory management, preemption and others will eventually make its way to the GPU, but only after potentially significant redesigns of the architecture.

One of the most exciting aspects of GPU computing to-date has been how promising software techniques developed first on the GPU quickly become system and hardware features. These features, typically, have not required significant re-design of GPU architecture in the past. We believe that on both these fronts—extending PT which exists as a software hack today with the minimal hardware, software and API extensions identified by us—the PT model is one such instance, and fits well into the GPU programming paradigm. We hope that our work toward characterization and understanding of PT will play an important role in simplifying and enhancing general-purpose applications on the GPU.