Summary: Efficient Deployment of Irregular Computations on Multi- and Many-core Architectures

Hancheng Wu
North Carolina State University
hwu16@ncsu.edu

1 Introduction

Multi- and Manycore processors have been advancing High Performance Computing with their high throughput and power efficiency. There has been an increasing interest in accelerating irregular computations on these devices that offer massive parallelism. My thesis focuses on compiler techniques and code transformations that facilitate the deployment of irregular computations on multi- and many-core processors, aiming to achieve higher performance and provide better programmability. I summarize my contributions below.

- We propose a compiler-based consolidation framework to improve the efficiency of irregular graph and tree computations written with Dynamic Parallelism on GPUs.
- We analyze and categorize parallel recursive tree traversal patterns, then provide insights on how to select the platform and code template based on identified traversal patterns.
- We propose compiler techniques to support a SIMT programming model on Intel multi- and many-core architectures with wide vector units, and point out the main challenges in supporting the SIMT model, especially for irregular computations.

2 Compiler-Assisted Workload Consolidation for Efficient Dynamic Parallelism on GPU (ICPP'15 and IPDPS'16)

Many irregular applications such as graphs and tree algorithms involve patterns of parallel irregular loops and parallel recursion and expose runtime nested parallelism, which is difficult to be handled at compile time. Recently, Nvidia has introduced Dynamic Parallelism (*DP*) in its GPUs. By making it possible to launch kernels directly from GPU threads, this feature enables nested parallelism at runtime. Programmer may achieve better load balancing by offloading the workload of nested parallelism met at runtime to a child GPU kernel. However, we find out that the naïve use of this feature often suffers from significant runtime overhead and leads to GPU underutilization, resulting in poor performance.

In this work, we address this problem. We propose a compiler-based workload consolidation approach to improve the performance of applications written with DP. Specifically, we consolidate into a single nested kernel the workload belonging to kernels that would be spawned by multiple GPU threads. We consider performing kernel consolidation at three granularities: *warp-*, *block-* and *grid-level*, whereby the consolidation involves the kernels launched by all the threads within a warp, all the threads within a block or the entire grid, respectively. We integrate our consolidation schemes in a directive-based compiler. By automating our code transformations, we allow programmers to write simple code focusing on functionality rather than on performance. We observe that static methods to configure the degree of multithreading of GPU kernels (e.g. CUDA occupancy calculator) are ineffective in the presence of DP, therefore we propose a systematic way to configure dynamic kernel launches. We evaluate our consolidation mechanisms on graph and tree algorithms that involve either parallel irregular loops or parallel recursion. Our results show that the grid-level consolidation yields the best performance. **We significantly reduce runtime overhead of DP and improve the GPU utilization, leading to speedup factors from 90x to 3300x over basic DP-based codes and speedups from 2x to 6x over flat implementations.**

3 An Analytical Study of Recursive Tree Traversal Patterns on Multi- and Many-core Platforms (ICPADS'17)

The irregular applications we have addressed in our previous work involve a single traversal over the underlying graphs or trees. For example, we have shown how to improve the DP-based recursive tree algorithms - *Tree Heights* and *Tree Descendants* (IPDPS'16). They perform a single parallelizable recursive tree traversal (DP implements the recursive tree traversal). Thus, we call this recursive traversal pattern *Parallel Traversal Single Data* (*PTSD*). A couple of recent efforts have focused on optimizing the execution of multiple serial tree traversals on GPU. These serial traversals are independent of each other, therefore can be executed in parallel. We call this pattern *Serial Traversal Multiple Data* (*STMD*). While both patterns exhibit parallelism, they have distinct behaviors.

In this work, we focus on irregular recursive tree traversals that are found in many application domains, such as data mining, graphics, machine learning and scientific simulations. We aim to understand how to select the implementation and platform most suited to a given tree traversal pattern and dataset. To this end, we identify *PTSD* and *STMD* patterns and perform a systematic study of them on CPU, GPU and the Intel Phi coprocessor. In fact, the traversals associated to STMD and PTSD patterns are depth-first and level-ordered, respectively. We further break down the STMD category into three patterns: STMD-*static*, STMD-*dynamic* and STMD-*top-down*, depending on the nature of the traversals. For

STMD-static, all traversals visit the child nodes following the same sequence (different traversals may skip different nodes and subtrees in the sequence). For STMD-dynamic, the order in which the children are visited differs from node to node and is query-dependent. For STMD-topdown, the traversals are limited to one child per node, and they always move in the direction of increasing depth. For each tree traversal pattern, we consider a set of recursive and iterative code variants, and implement them on CPU, GPU and the Intel Phi. For STMD patterns, we revisit existing optimization techniques (*dynamic ropes* and *lockstep execution*), and propose two additional techniques: *static ropes* and *greedy-dynamic-ropes*. For the PTSD pattern, we evaluate a basic and an enhanced recursive implementation as well as an iterative implementation based on a flattening transformation. We evaluate and analyze these code variants on CPU, GPU and the Intel Phi using datasets with different characteristics (trees with different shapes, and sorted and unsorted inputs for the STMD pattern).

Our analysis shows that there is no single solution that performs best on all tree traversal patterns and input datasets. CPU code variants (either recursive or iterative) are generally preferable for STMD applications where different queries originate different traversal behaviors at runtime. For STMD applications where the traversal sequence is fixed and known a priori and parallel traversals differ only in the nodes they skip in that sequence, **GPU implementations with explicit stack and lockstep execution tend to report the best performance, especially on sorted datasets. For PTMD algorithms, the optimized recursive GPU code variant outperforms the other implementations significantly, and the execution on CPU and Phi suffers from the serialization overhead of atomic operations.**

4 Compiling SIMT Programs on Multi- and Many-core Processors with Wide Vector Units (HiPC'18)

Although GPUs and *Intel hybrid architectures* (Intel multi- and many-core (co)processors with wide vector extensions) are both designed to provide massive parallelism, their architectural organization and programming paradigms differ in many ways. GPUs can be programmed with SIMT programming model such as CUDA. In contrast, high performance coding on Intel hybrid architectures requires using both their x86-compatible cores (MIMD model) and SIMD vector units (SIMD model). Efficient use of both MIMD and SIMD models on hybrid architectures is not feasible. While our previous work mainly targets GPUs, in this work, we explore how to deploy irregular applications on *Intel hybrid architectures* efficiently. Specifically, we ask ourselves two questions. First, how to develop a SIMT programming model to allow the simultaneous use of x86 cores and their vector units, and to provide programmability and code portability on hybrid architectures? Second, what are the performance bottlenecks of such SIMT model, especially for irregular applications? To this end, we first propose compiler techniques that enables the transformation of programs written using a SIMT programming model into code that leverages both the x86 cores and the vector units of a hybrid architecture. We then evaluate the SIMT model with microbenchmarks and real-world applications.

We consider a subset of CUDA-C as the SIMT programming language, code based on Pthreads and vector intrinsics as compilation target, and three Intel processors with 512-bit vector extensions as our destination platforms. The key compiler techniques are the transformations that support **Control Flow** and **Function Calls** on SIMD vector lanes. On Intel hybrid architectures, the control flow is maintained by the threads running on x86 cores, thus, enabling SIMT model on vector lane level requires handling the case where different VPU lanes take different branches, especially for nested control flow statements. To support this, we associate a mask variable to each **block scope** and issue all vector instructions inside the scope with that mask. Child scopes first naturally inherit their mask variables from the parent scope, then have their mask variables refined based on conditional statements. To transform a function written in SIMT style into a function that executes on VPUs, each device function is treated as a block scope and its mask variable is associated with an extra mask parameter inserted to receive the mask variable passed from the caller scope. Thus, vector lanes inactive at the caller scope will stay inactive in the callee scope. In addition, our transformation makes sure that the function will not return until all vector lanes have terminated at the function.

To evaluate the framework, we implement various microbenchmarks and real-world applications to characterize the performance bottlenecks. **We have found out that several performance-limiting factors will result in poor performance of existing CUDA applications on hybrid systems. They are iterative kernel launches, thread-block synchronization, and the use of shared memory on large thread-block configurations.** On one hand, we find out traditional GPU implementations of irregular computations such as BFS and SSSP can hardly benefit from hybrid architectures, either suffering from iterative kernel launches or heavy synchronization points. The high cost of synchronization overhead on hybrid architectures also makes our GPU consolidation method (IPDPS'16) not beneficial. On the other hand, we find out Intel Phi processors outperform Pascal GPUs on certain CUDA applications which exhibit regular memory accesses, incur no iterative kernel invocations and avoid synchronization.

4 Future Work

We are trying to improve the performance of irregular computations on Intel hybrid architectures by removing the bottlenecks. First, we are looking into hybrid synchronization implementations to reduce the synchronization cost. Second, we will study synchronization-free implementations of irregular computations. (e.g., avoiding using atomics for graph applications by preprocessing the input graph.)

Due to the 2-page space limitation, please refer to each publication for the detailed reference list.