Hybrid CPU-GPU scheduling and execution of tree traversals

Jianqiao Liu, Nikhil Hegde, Milind Kulkarni

School of Electrical and Computer Engineering, Purdue University
liu1274, hegden, milind@purdue.edu

Abstract

GPUs offer the promise of massive, power-efficient parallelism. However, exploiting this parallelism requires code to be carefully structured to deal with the limitations of the SIMT execution model. In recent years, there has been much interest in mapping *irregular* applications to GPUs: applications with unpredictable, data-dependent behaviors. While most of the work in this space has focused on *ad hoc* implementations of specific algorithms, recent work has looked at generic techniques for mapping a large class of *tree traversal* algorithms to GPUs, through careful restructuring of the tree traversal algorithms to make them behave more regularly. Unfortunately, even this general approach for GPU execution of tree traversal algorithms is reliant on *ad hoc*, hand-written, algorithm-specific scheduling (*i.e.*, assignment of threads to warps) to achieve high performance.

The key challenge of scheduling is that it is a highly irregular process, that requires the inspection of thread behavior and then careful sorting of the threads into warps. In this paper, we present a novel scheduling and execution technique for tree traversal algorithms that is both general and automatic. The key novelty is a hybrid approach: the GPU partially executes tasks to inspect thread behavior and transmits information back to the CPU, which uses that information to perform the scheduling itself, before executing the remaining, carefully scheduled, portion of the traversals on the GPU. We applied this framework to five tree traversal algorithms, achieving significant speedups over optimized GPU code that does not perform application-specific scheduling. Further, we show that in many cases, our hybrid approach is able to deliver better performance even than GPU code that uses hand-tuned, application-specific scheduling.

Categories and Subject Descriptors D.1.3 [Concurrent Programming]: Parallel Programming; I.2.8 [Problem Solving, Control Methodsm and Search]: Scheduling

Keywords Heterogeneous architectures, Scheduling, Irregular applications, Tree traversal

1. Introduction

GPUs offer the promise of massive, energy-efficient parallelism, providing hardware that can execute hundreds of simultaneous parallel threads. Unfortunately, achieving such highly efficient parallelism requires compromises: GPUs present a somewhat lim-

ited SIMT (single-instruction, multiple-thread) execution model, where, to take full advantage of the parallel execution resources, threads that are executing simultaneously must perform the same computation (avoiding *control divergence*) and access memory in a predictable way (avoiding *memory divergence*). In other words, while GPUs appear to be well-suited to executing data-parallel algorithms, the full power of the GPU cannot be exploited unless the data-parallel tasks are similar to each other. As a result, most successful GPU algorithms are *regular*, with predictable control flow and memory access patterns.

There has been considerably less success tackling *irregular* algorithms. These algorithms, which feature data structures such as trees and graphs and input-dependent behavior, are much more difficult to map to GPUs, as the input-dependence precludes grouping together threads to minimize control divergence and the pointer-based data structures means that even if threads are performing similar operations, their memory accesses are likely not predictable, increasing memory divergence. As a result of these difficulties, most attempts to map irregular applications to GPUs have been one-off implementations: for each new algorithm, a new, *ad hoc* implementation for GPUs must be developed [1, 3–6].

Recently, Goldfarb *et al.* developed a general framework that took advantage of two transformations, *autoroping* and *lockstepping*, to restructure tree traversal algorithms so that they execute effectively on GPUs. Unfortunately, to achieve maximum performance using their framework, Goldfarb *et al.* used application-specific “sorting” optimizations to reorganize the set of traversals to be performed so that the traversals that were grouped into SIMT thread groups were likely to touch similar portions of the tree. What is missing is an approach to tackling these problems that does not rely on application-specific knowledge to be effective.

In this work, we take advantage of a key insight about the behavior of tree traversal algorithms that allows us to perform effective scheduling without performing application-specific sorting: even though traversals are input-dependent and hence inherently unpredictable, the depth-first nature of recursive traversals means that the behavior of a traversal as it explores the “lower half” of the tree (*i.e.*, nodes at depth more than half the tree height) is largely determined by its behavior in the “upper half” of the tree (*i.e.*, which nodes in the upper half of the tree the traversal visits, and in which order). In other words, two traversals that have similar behaviors in the upper half of the tree will behave similarly in the lower half of the tree, as well. However, the vast majority of the *work* performed by the traversal occurs in the lower half of the tree. As a result, it is possible to examine the behavior of traversals as they visit the upper half of the tree, and use that behavior to *reschedule the traversals* before they visit the lower half of the tree. Hence, we can dynamically schedule traversals based on their history to achieve effective scheduling.

We introduce a novel *hybrid scheduling algorithm*. The key insight is that tracking traversal behavior is straightforward to parallelize, while the reorganizing and scheduling process is highly

irregular and difficult to optimize for GPUs. Hence, we track the behavior of traversals in the “top half” of the tree on the GPU, then send the results back to the CPU to perform the scheduling itself. The CPU then uses the new schedule to drive the execution of the “bottom half” of the traversals on the GPU, with the dynamic schedule providing substantially improved performance.

2. Design

This section describes our hybrid CPU-GPU scheduling strategy, a novel scheduling and execution technique for tree traversal algorithms that is both general and automatic. We do not rely on any application-specific or semantic knowledge. Instead, our technique uses two GPU kernels: one that runs a portion of the traversal code on the GPU while inspecting the behavior of individual traversals (*inspecting*). The CPU then uses this information to dynamically reorder the traversals (*Scheduling*) so that when the second kernel is called (*Execution*), threads grouped into warps perform similar work, improving SIMT efficiency.

The inspecting step is essentially partial execution of the whole computation task but truncated at certain depth. The deeper this truncation depth, the more information is collected and a better scheduling result can be expected, however, with a higher overhead. In our case, we choose an inspecting depth of one third or half of the maximum depth of the tree. Because the bottom half of the tree contains the bulk of the tree nodes, the inspecting overhead is negligible. The low time consumption in hybrid scheduling contributes a lot on its performance advantage in evaluation.

During inspecting, the traversal of each point is traced and recorded into a *scheduling matrix*. The *scheduling matrix* has one row for each point, and one column for each node in the top tree. When a point visits a node, the appropriate cell in the table is marked. This matrix is transferred back to the CPU for use during scheduling.

The final stage of our process is the execution phase. We simply run the original GPU kernel (with the addition of our optimized lockstep skeleton) for the traversal algorithm using the order of points determined during scheduling. Since this order of points is based on points that we expect to behave similarly, this has an analogous effect to when Goldfarb *et al.* ran their kernels on *sorted* input points.

Guided and unguided traversals In some algorithms, such as nearest neighbor, the particular order a point visits nodes is governed by point-specific data. For example, based on characteristics of a point, one point might visit the tree root’s left child before its right, while another might visit the right child before the left. Following Goldfarb *et al.*’s terminology, we call algorithms that have this property *guided* traversals, in contrast to algorithms like point correlation that are *unguided* [2]. Note that whether a traversal is guided or not can be determined by a simple static analysis that determines whether the order of recursive calls is control dependent on any point-specific data. For unguided traversals, all the traversals follow the same order, so the *scheduling matrix* only records the set of visited nodes, but for guided traversals, the *scheduling matrix* is restructured to include traversal order in addition to visited nodes.

The CPU then analyzes the *scheduling matrix* and groups points into different buckets according to the nodes they visited. Points in one bucket are more likely to access same branches of the tree and hence exhibit better locality. For guided traversal, since the scheduling matrix contains both the traversal nodes and their order information, above scheduling is enough to achieve good performance. However, for unguided traversal, the scheduling can only partition points into limited number of buckets, while each bucket accumulates large number of threads with considerable control divergence. Therefore, intra-bucket scheduling is introduced along

Figure 1. Speedup comparison

with bucket generation. By continuously refining the schedule as we build the scheduling buckets, a final schedule capturing similarity of points’ traversals is obtained for every bucket.

3. Evaluation

To evaluate the performance of Hybrid CPU-GPU scheduling strategy, we study five tree traversal benchmarks: k-Nearest Neighbor (kNN), Vantage Point (VP), Nearest Neighbor (NN), Point Correlation (PC) and Barnes Hut (BH). We evaluate the original kernels (Goldfarb *et al.*’s original lockstep code) with both unsorted and sorted inputs, and our new hybrid scheduling strategy with unsorted input. We evaluate our benchmarks on a server with two AMD Opteron 6164 HE Processors and an nVidia Tesla K20C GPU. The system has 32GB system memory and run on Red Hat 6.6 with Linux kernel v2.6.32. We take Goldfarb *et al.*’s original implementation [2] with unsorted input as the baseline. Each evaluation is run 15 times, and the resulting confidence intervals are small.

Due to the lack of space, we only show the first two benchmark results in figure 1. Across all of the inputs in kNN and VP, we see that our hybrid scheduling is significantly better than hand-written sorting methods. **The geometric mean speedup for all five benchmarks for hybrid scheduling is $6.37\times$ over the baseline, and $1.82\times$ over the optimized sorted version.** In other words **our automatic, hybrid scheduler outperforms hand-written, application-specific sorting!**

References

- [1] M. Burtcher and K. Pingali. An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. In *GPU Computing Gems Emerald Edition*, pages 75–92. Elsevier Inc., 2011.
- [2] M. Goldfarb, Y. Jo, and M. Kulkarni. General transformations for gpu execution of tree traversals. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’13, pages 10:1–10:12, New York, NY, USA, 2013. ACM.
- [3] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek. Realtime ray tracing on gpu with bvh-based packet traversal. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 113–118, 2007.
- [4] M. Hapala, T. Davidovic, I. Wald, V. Havran, and P. Slusallek. Efficient Stack-less BVH Traversal for Ray Tracing. In *Proceedings 27th Spring Conference of Computer Graphics (SCCG) 2011*, pages 29–34, 2011.
- [5] M. Méndez-Lojo, M. Burtcher, and K. Pingali. A gpu implementation of inclusion-based points-to-analysis. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 107–116. ACM, 2012.
- [6] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 117–128, 2012.