CS575 Paper Analysis

Yunfan Li
liyunf@oregonstate.edu

Analysis of "General Transformations for GPU Execution of Tree Traversals"

1. Authors' Background and Brief Introduction

There are three authors in this paper, Michael Goldfarb, Youngjoon Jo and Milind Kulkarni. Let's take a look at these three authors respectively.

Michael Goldfarb: He was a Master of Science in Perdue University from 2010 to 2013, and he also graduated from Perdue University at 2010 and got his bachelor degree. He has a three-months intern in BAE Systems Company from June 2009 to August 2009. He worked on Perl script developing, specifically, developed a Perl tool that can generate C++ code and XML document creation. He also had an internship in Qualcomm for about four months from May 2012 to August 2012, this time, he was working on Qualcomm's CDMA Technologies Group. From July 2013 to November 2015, he worked on Qualcomm Company as a Software Engineer at Qualcomm's Research Center, majorly working on high-performance machine learning for Snapdragon processor which produced by Qualcomm. He has been promoted to Senior Software Engineer since November 2015, and currently working SoC architecture research team at Qualcomm's Research Center. The only publication of him is "General Transformations for GPU Execution fo Tree Traversals" at 2013.

Youngjoon Jo: He got his bachelor degree from Seoul National University at 2009 and then joined in Purdue University as a Ph.D. student, and finally graduated from Purdue University at 2013 majored in Electrical and Computer Engineering. He was a Game Developer in Nexon from March 2006 to January 2008. He got an internship in Google as a Software Engineering Developer from May 2011 to August 2014. In the summer of 2012, he joined Microsoft as a Software Developer for another internship. From November 2013 to present, he has worked in Google as a Software Engineer. During his Ph.D. degree, he got a lot of publications, mostly focused on tree traversal transformation on GPUs. For example, at 2013, on PACT, he published a paper named "Automatic Vectorization of Tree Traversal", at 2012, on OOPSLA, he published a paper named "Automatically Enhancing Locality for Tree Traversals with Traversal Splicing".

Milind Kulkarni: He is an associate professor with the School of Electrical and Computer Engineering at Purdue University. His main research area is Programming Languages and Compilers, he is specifically focusing on developing languages, compilers, and runtimes that support efficient programming and high performance on emerging complex architectures. He graduated in 2002 with a B.S in both Computer Science and Computer Engineering from North Carolina State University, and then received his Ph.D. degree from Cornell University. After he got his Ph.D. degree, he joined Institute of Computational Sciences and Engineering at the University of Texas at Austin as a postdoc, where he worked in Intelligent Software System (ISS) group. He currently has 8 students, all of them are Ph.Ds. He teaches ECE468 – "Introduction to Compiler and Translation Systems Engineering" and ECE 573 – "Compilers and Translation Systems Engineering" in Purdue University. Formerly he taught ECE 663 – "Advanced Optimizing Compilers" at Spring term of 2010. His current research fields are "Automatically optimizing irregular applications", "Optimizing computational science applications by exploiting semantics", "Detecting and diagnosing bugs in large-scale distributed systems", and "Effective computation offloading". Most of the research projects are funded by NSF or DOE.

2. General Theme of this paper

In this paper, the authors try to figure out a useful method or algorithm to effectively transform the irregular algorithms from CPU-based code to GPU-based code. For now, there are few works focus on pointer-based dynamic data structures are involved in irregular algorithms. There are some works that provide implementations that can transform some of these algorithms to GPU-based code, but all of them rely on exploiting application-specific semantics so that they can get an acceptable performance. The authors therefore, propose a novel method that can implement irregular algorithms, especially those which contains pointer-based data structures, on GPUs in a general-purpose technique. They demonstrate these techniques on several tree traversal algorithms, and shows that they achieving speedups of up to 38x over 32-thread CPU versions code.

Currently, most of the works which are focusing on transform algorithms on GPUs are using regular algorithms that has dense linear data storage fashion. These data are organized by predictable, structured accesses to memory and can exploiting large amounts

of data parallelism on GPUs. The irregular applications which perform unpredictable, data-driven accesses are not in a spot light because it is too complicated to do hand-tuned work on these programs to make their memory access pattern adapt to the GPUs memory access pattern. In this paper, the authors show that tree traversal algorithms have several common properties that can be exploited to produce efficient GPU implementations. And these properties are not relying on the semantics of different algorithms but instead emerge from common structural features of tree traversal algorithms. In general, the authors can develop a catalog of techniques to optimize traversal algorithms for GPUs and provide a set of easily-checked structural constrains that govern when and how to apply these techniques.

The primary transformation they developed is called autoroping. “Rope” is used to preprocess the tree itself that installs some extra pointers that connect a node in the tree not to its children, but instead to the next new node that a point would visit if its children are not visited. Every node, including interior nodes, processes such pointers. With these “ropes” installed into a tree, a point never needs to return to interior nodes to perform its traversal. However, a tree must be preprocessed to install the ropes, which will consume a lot of time to do this on GPUs. Therefore, the authors think that we can perform the preprocessing step during tree traversal and store them on stack instead of preprocessing the tree data structure and store the “ropes” in the tree.

In the Figure 4 and Figure 5 of this paper, the authors show us the traditional CPU version of tree traversal pseudo code, and the Figure 6 and Figure 7 show us how these code transformed after applied the autoroping technique. Autoropes saves the ropes mentioned before to the stack dynamically at runtime, obviating the need for additional preprocessing steps or semantic knowledge about the traversal. After apply to the autoropes, the traversal is facilitated by a loop that repeatedly pops the address of the next node in the traversal from the top of the stack until the stack is empty, indicating there are no more nodes to visit. Thus, the recursive calls are replaced with stack push operations, but the order in which nodes are pushed is the reverse of the original order of recursive calls, ensuring the order that nodes are visited remains unchanged. In other words, the implicit function stack is replaced with an explicit rope stack. They also assume that the recursive traversal function does not return any values.

3. Experiments in this paper

The authors transformed several important tree traversal algorithms from different application domains. For each benchmark, they evaluate non-lockstep and lockstep versions of the algorithm which are discussed in Section 4.3. They firstly compare a naïve GPU implementation that uses CUDA compute capability 2.0's support for recursion to direct map the recursive algorithm to the GPU. The only difference between naïve implementation and theirs is the use of autoropes. The also compare against parallel CPU implementations of the same traversal algorithms. In their experiments, the CPU benchmarks were compiled using gcc 4.4.7 with optimization level –O3, and GPU benchmarks were compiled using nvcc for compute version 2.0 and with CUDA version 5.0. Their platforms are as follows:

CPU system contains four AMD Opteron 6176 processors that contain 12 cores running at 2300MHz. Each CPU has 64Kb L1 data cache per core, 512kB L2 cache per core, two 6MB shared L3 caches and 256GB of main memory.

GPU system contains one nVidia Tesla C2070 GPU which contains 6GB of global memory, 14SMs with 32 cores per SM. Each SM contains 64KB of configurable shared memory.

4. Conclusions of this paper

In this paper, the authors described a series of transformations that enable the efficient execution of tree traversal algorithms on GPUs. These techniques unlike in most prior work on GPU implementations of irregular algorithms, do not rely on application-specific semantic knowledge, instead leveraging only structural properties of traversal algorithms. In the methodology section, the authors show that the speedup of CPU performance vs. GPU is over 30x in particular benchmarks. And even better if applying the point-sort technique. All of these can be achieved without taking advantage of application-specific knowledge.Which means that autoropes and lockstep are effectively improve the efficiency of irregular algorithms which contains sparse, pointer-based dynamic data structures on GPUs. These techniques can help researchers figure out some novel fields that can be researched to implement irregular algorithms on GPUs.

5. Insights from this paper that I can get

From this paper, I think I have learned that a new technique called “autorope” that can effectively transform an implicit function recursive calls stack to an explicit rope storage stack. Which can turn recursive version algorithm to a while-loop fashion algorithm. Thus our CPU-based irregular tree traversal algorithm can be transformed to a GPU-based traversal algorithm.

I also learned another technique which is “lockstep”. This technique can be used on GPU to increase the memory coalescing and eliminate the Thread divergence in GPU.

6. Flaws or short-sightedness of this paper

I think the only problem in this paper is they lack of the data which come from hand-tuned irregular algorithms. As authors mentioned in Section 3, in order to make the whole techniques more general, they have to sacrifice some of efficiency compared to hand-tuned algorithms. I would like to know how much efficiency do they have to sacrifice?

7. Future work of this paper

I think in the future, the authors should focus on find out more general property that exits in most of irregular algorithms, so that we can immigrating more and more algorithms from CPUs to GPUs that can exploiting data parallelism in GPUs.