Stack-Based Parallel Recursion on Graphics Processors

Ke Yang
Zhejiang Univ.
kyang@cad.zju.edu.cn

Bingsheng He
HKUST
saven@cse.ust.hk

Qiong Luo
HKUST
luo@cse.ust.hk

Pedro V. Sander
HKUST
psander@cse.ust.hk

Jiaoying Shi
Zhejiang Univ.
jyshi@cad.zju.edu.cn

Abstract

Recent research has shown promising results on using graphics processing units (GPUs) to accelerate general-purpose computation. However, today's GPUs do not support recursive functions. As a result, for inherently recursive algorithms such as tree traversal, GPU programmers need to explicitly use stacks to emulate the recursion. Parallelizing such stack-based implementation on the GPU increases the programming difficulty; moreover, it is unclear how to improve the efficiency of such parallel implementations. As a first step to address both ease of programming and efficiency issues, we propose three parallel stack implementation alternatives that differ in the granularity of stack sharing. Taking tree traversals as an example, we study the performance tradeoffs between these alternatives and analyze their behaviors in various situations. Our results could be useful to both GPU programmers and GPU compiler writers.

Categories and Subject Descriptors D.1.3 [**Programming Techniques**]: Concurrent Programming -- Parallel programming Language

General Terms Algorithms, Languages

Keywords Stack, Parallel Recursion, Graphics Processors

1. Introduction

Recursion is a fundamental programming construct. A recursive function consists of a *base case,* which can be solved directly, and a *recursive case,* which calls the function itself and reduces the problem domain towards the base case. At each recursion level, if the current function call becomes a base case, it will be solved and will return to the caller. Otherwise, it will partition the problem into sub-cases and go down to the next recursion level for each sub-case. Such execution forms a recursion tree, and can be converted into an iterative process using auxiliary data structures, in particular, a stack.

In this paper, we study parallel implementation alternatives for stack-based recursion on GPUs. The GPU can be viewed as a kind of massively threaded parallel hardware. **Figure 1** shows a typical organization of GPU threads. Multiple SIMD (Single-Instruction-Multiple-Data) GPU threads are grouped into a *warp*, and a batch of thread warps form a *block*. Thread blocks are the synchronization unit for GPU execution, and threads within a block share a small piece of on-chip local memory. Due to the massive threading parallelism and the SIMD nature of the GPU, GPU programs must exploit SIMD coherence, minimize thread communication and utilize on-chip local memory for efficiency. It is therefore challenging to use GPUs for parallel recursions [4], because (1) recursions are not directly data parallel since there are communications between each pair of recursion caller and callee, (2) the recursion tree may be irregular, and (3) data sizes of base cases may vary.

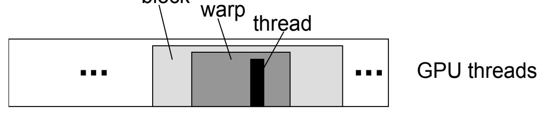

Figure 1. GPU thread organization.

Considering these challenges, we propose three GPU-based stack implementation alternatives, namely per-thread stack, per-warp stack and per-block stack, and study their performance. We have implemented these alternatives in a GPU-based tree traversal application. Our preliminary results show that stack-based recursion can be done on the GPU efficiently and that the relative performance of each alternative depends on the fanout of the recursion tree.

2. GPU-Based Parallel Stacks

We design three kinds of parallel stacks that differ in the granularity of stack sharing. We avoid write conflicts in sharing through software mechanisms [5]. It can also be done through hardware; either option can be expensive, and the cost generally increases with the number of conflicting threads.

2.1 *t_stack*

A per-thread stack (*t_stack*) is a local array owned by each thread; individual threads do not share stacks at all. Each thread independently handles a recursive task using stack operations similar to those on the CPU, and all these tasks can be executed in parallel [6]. However, if there are branches in a recursion case, the execution of individual threads will be divergent, and the SIMD hardware will be underutilized. Moreover, since concurrent writes occur among all threads, there will be intensive communication between threads. As a result, this *t_stack* is suitable for recursions with fine-grained parallelism.

2.2 *b_stack*

A per-block stack (*b_stack*) is a local array owned by a thread block. Each block of threads handles a recursive case, and multiple recursive cases are executed simultaneously among blocks.

In each recursive case, we first partition the input and generate related bookkeeping information in parallel. Then we perform a block-level synchronization operation, after which a single thread of each block uses the bookkeeping information to locate all the sub-cases and pushes them to the stack. Additionally, all base cases are parallelized among blocks.

PPoPP'09, February 14-18, 2009, Raleigh, North Carolina, USA.
ACM 978-1-60558-397-6/09/02.

Thread communications in *b_stack*s occur at two levels, namely intra-block and inter-block. Intra-block communication is efficient using the intrinsic barrier mechanism, and it is possible to cache block-level bookkeeping information on-chip. Furthermore, due to the limited number of blocks, we can pre-allocate a write buffer to each block, and the overhead of inter-block communication is much smaller than that of inter-thread one in *t_stack*s.

2.3 *w_stack*

A per-warp stack (*w_stack*) is a local array owned by a SIMD warp; each warp handles a task in parallel. The granularity of sharing a *w_stack* lies between that of *t_stack* and *b_stack*. This stack minimizes thread divergence within one warp. Compared with *b_stack*s, the warp-scope stacks and their bookkeeping structures are smaller and more likely to fit in on-chip stores. More importantly, the overhead of serializing stack updates by a single thread is confined by the warp width. Therefore, the communication overhead is generally less than that of *b_stack*.

2.4 Discussion

Stack depth. All three kinds of stacks can be efficiently implemented in CUDA [1] by allocating a sufficiently large, fixed-sized array. In the uncommon cases of stack overflow, the thread dumps the stack to the GPU memory and resumes the execution in a second kernel.

Hybrid alternatives. Since the granularity of parallelism may vary in a recursion tree, it might be better to switch among different stack models during execution. For example, we may use *b_stack*s for a small number of sub-tasks at the beginning, and then use *w_stack*s or *t_stack*s for a large number of sub-tasks approaching the base cases. The challenge of developing an efficient hybrid scheme on the GPU is to reduce the switching overhead, especially the book-keeping.

Applications versus compilers. Given the strengths and weaknesses of the three kinds of parallel stacks, GPU developers have the flexibility to choose individual or hybrid alternatives suitable for their own algorithms, presumably having a better knowledge of their algorithms than the compiler. On the other hand, if the compiler natively supports recursion, it will significantly ease programming and possibly improves the efficiency.

3. Preliminary Results

We have applied our GPU-based parallel stacks to a representative recursive problem, tree traversal [3]. The tree index is a two-dimensional R-tree [2] on 4M records amount to 64MB, and the workloads are 100K two-dimensional range queries. We use CUDA to implement the programs on a GeForce 8800GTX GPU. For comparison, we have also implemented a CPU-based parallel traversal routine using OpenMP with two threads running on an Athlon Dual Core CPU. This routine uses native recursion.

We study the query performance at various degrees of parallelism, specifically, the number of input partitions of each recursive case, or the fanout of the recursion tree. For tree traversal, this corresponds to the node size in number of entries (denoted as *N*). With *N* varied under the same workload, we measure the execution time using the three types of stacks on the GPU, in comparison with the CPU time.

Figure 2 shows the time in log scale with node size varied. The highest speedup of GPU over CPU using *t_stack*, *b_stack* and *w_stack* is 5X, 5.9X and 6.3X, respectively. When *N* is smaller than four, parallelism among nodes is limited, especially near the root of the recursion tree. As a result, most threads in *b_stack* and *w_stack* are underutilized, and *t_stack* becomes the fastest. As the

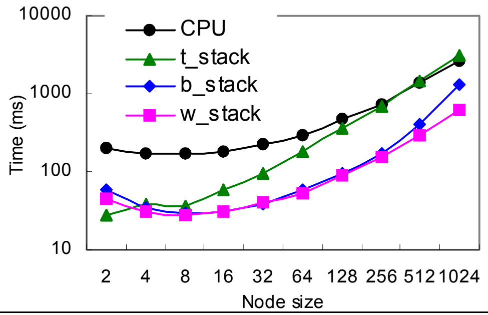

Figure 2. Traversal performance with node size varied.

node size grows, the divergence in *t_stack* becomes significant. When *N* is larger than 1024, *t_stack* becomes even slower than the CPU routine. For *N* less than the warp size (32), *b_stack* performs similarly to *w_stack*. When the node size is larger than 256, *b_stack's* communication overhead becomes more significant, and *w_stack* performs faster. Therefore, *t_stack* is the first choice for small nodes (e.g., $N < 4$), and *w_stack* is the best alternative for large nodes. Since our preliminary results are limited to tree traversals, we speculate *b_stack* might be more apt at more coarse-grained tasks such as sorting, and might outperform *w_stack* in such cases. Such further comparison is in our ongoing work.

4. Conclusions

Graphics processors have become an attractive alternative for general-purpose high performance computing on commodity hardware. In this study, we have designed three stack implementation alternatives for emulating recursions on GPUs. These parallel stacks differ in the granularity of stack sharing and are suitable for different situations. We have implemented these alternatives for tree traversal on the GPU and have compared the performance with the node size varied. Our results could be useful to both GPU programmers and GPU compiler writers.

As ongoing work, we are applying our techniques to other recursive algorithms, such as quick sort, on the GPU, are investigating the relative performance of these alternatives, and are exploring a hybrid approach that utilizes multiple kinds of stacks.

Acknowledgments

The authors thank the anonymous reviewers for their insightful suggestions. This work was supported by grant 616808 from the Hong Kong Research Grants Council. Ke Yang and Bingsheng He are currently with Microsoft Corp.

References

[1] CUDA, http://developer.nvidia.com/object/cuda.html.

[2] A. Guttman, R-trees: A dynamic index structure for spatial searching. In Proc. ACM SIGMOD, pp. 47-54. 1984.

[3] S. Popov, J. Günther, S. Hans-Peter et al, Stackless KD-Tree Traversal for High Performance GPU Ray Tracing In: Computer Graphics Forum 26(3), pp. 415–424, 2007.

[4] L. Prechelt, S. U. Hänßgen, Efficient Parallel Execution of Irregular Recursive Programs, IEEE Transactions on Parallel Distributed Systems 2002, 13(2):167 - 178.

[5] B. He, K. Yang, R. Fang et al, Relational Joins on Graphics Processors, SIGMOD 2008.

[6] K. Zhou, Q. Hou, R. Wang, B. Guo, Real-Time KD-Tree Construction on Graphics Hardware, SIGGRAPH Asia 2008.