Fenceless Floating Point Algorithms

Charlie Jacobsen; Ganesh Gopalakrishnan, advising

Introduction

Parallel algorithms for numerical linear algebra and differential equations may involve the exchange or sharing of a large amount of floating point data among threads. For example, the Jacobi algorithm for solving the two-dimensional Poisson equation is typically parallelized by distributing blocks of the initial floating point matrix among threads. Each thread is responsible for carrying out a stencil operation on each element in its block for each iteration. Along the perimeter of a block, the stencil will overlap with neighboring blocks; hence, the thread will need to either share or obtain this floating point data from neighboring threads after each iteration. For a large matrix, the block boundaries can contain a significant amount of shared floating point data.

Memory barriers and synchronization operations are often used to enforce a strict ordering on shared memory accesses. We conjectured that some algorithms with shared floating point data may achieve acceptable accuracy and better performance when some memory barriers and synchronization operations are removed and the ordering of such accesses is more relaxed. Threads may compute similar results from stale or slightly inconsistent data.

Related Work

Synchronization relaxation is not new (although we were unaware of this idea when we began!). Chazan and Mirranker proved that iterative algorithms like the Jacobi and Gauss-Siedel algorithms still converge under certain conditions when values are updated in a periodic (but not serial) pattern [1]. Venkatasubramanian, et al. analyzed the behavior of the Jacobi algorithm on GPUs using asynchronous kernels [2].

Memory barriers and synchronization usually consume extra processor cycles and system resources, and hence, system power. This project is therefore related to approximate computing, an emerging area of research covering methods at all levels of the software and hardware design “stack”—algorithms, software systems, and hardware—for conserving energy while ensuring acceptable performance; see Han, et al. for a good overview [3].

Others have looked for opportunities to remove memory barriers from software that preserve correctness and enhance performance. Morrison, et al. found that certain memory barriers are unnecessary in the work-stealing algorithm that is part of the runtime for Intel Cilk, a parallel programming environment for multicore Intel chips [4].
Project Work Outline

Weak Memory Models Survey

In order to understand the consequences of removing memory barriers from parallel algorithms, we first surveyed some of the current weak memory specifications and literature, including the overview by Howells, et al. in the Linux kernel development tree and the memory ordering specification in the Intel IA-32/64 Architecture manual [5,6]. We found that the Intel architecture has strong memory consistency (i.e., less memory re-ordering): Loads and stores to the same memory location are never re-ordered, and memory ordering is transitive. However, stores can be re-ordered with loads to different memory locations.

We did not include GPU weak memory models in our survey, although current work is being done in this area [7].

Parallel Algorithms

Since parallel versions of some numerical linear algebra and differential equations algorithms use a significant amount of shared memory data, we first looked for current implementations of these algorithms in high-performance libraries. BLAS and LAPACK are high-quality linear algebra library “standards”, standardized by a reference implementation [8]. While most of the tuned parallel versions of BLAS and LAPACK are proprietary, we did find ATLAS, an open-source, highly-parameterized library that can be used to produce a parallelized version of BLAS [9]. As of release 3.10.1, ATLAS appears to primarily use synchronization (implemented with atomic operations that ensure memory ordering) for coordinating shared memory accesses.

From prior experience, we knew that the Jacobi algorithm converges regardless of the initial guess for the two-dimensional Poisson equation. We used it as an object of study in future phases of the project for experimenting with the trade off between accuracy and performance with respect to memory relaxation. We sought out a high-quality reference implementation that we could modify, but had trouble finding one, so we built our own.

Parallel Jacobi Algorithm: Implementations and Experiments

We implemented the Jacobi algorithm for GPUs and multicore CPUs (using Pthreads). Each implementation is parameterized to test various thread block dimensions, synchronization frequency, and so on. The code is available here: http://www.cs.utah.edu/formal_verification/civl/. (The tarball is large due to the size of the test matrices.)

We found that in both implementations, decreasing the synchronization frequency (i.e., less synchronization) gave comparable accuracy and improved performance. On a 512 x 512 matrix, with 8 x 16 blocks, 32 x 16 threads per block, and 2 x 2 elements per thread, the GPU Jacobi algorithm finished in 47.898 seconds with synchronization every 1,000 iterations versus 58.367 seconds with synchronization
every iteration (1,000,000 iterations total). The average absolute error in the former was 0.000385 and the latter was 0.000191.

On a 256 x 256 matrix, with 4 x 4 threads, each assigned to a core, and with 64 x 64 elements per thread, the CPU Jacobi algorithm finished in 98.342 seconds with synchronization every 1,000 iterations versus 125.506 seconds with synchronization every iteration (100,000 iterations total). The average absolute error in the former was 0.000194 and the latter was 0.000098.

Future Work

More testing needs to be done with these implementations (more combinations of grid and block sizes, etc.). In future projects, we may investigate the accuracy and performance of the ATLAS library when some synchronization operations are removed or relaxed. It may also be interesting to see an “approximate computing” implementation of the BLAS or LAPACK libraries.

Acknowledgments

I would like to thank Ganesh Gopalakrishnan and Tyler Sorensen for the helpful insights, suggestions, and guidance while carrying out this project.

References