Implementing the Parallel Iterative Improvement Algorithm for the Stable Marriage Problem on GPUs

Andrew A. Barkley[1], Justin A. Martin[2]

[1]State University of New York, Potsdam, New York, [2]Lock Haven University, Lock Haven, Pennsylvania

Abstract

We implemented a parallel algorithm for finding a stable matching on massively parallel graphics processing units (GPUs). The algorithm is the Parallel Iterative Improvement (PII) Algorithm for the Stable Matching Problem [5]. It has applications in real-time communication switching networks [4].

Introduction

The stable matching problem is often similarly stated as follows: "Given n men, n women, and $2n$ ranking lists in which each person ranks all members of the opposite sex in order of preference, a *matching* is a set of n pairs of men and woman with each person in exactly one pair. A matching is *unstable* if there are two persons who are not matched with each other, and each of whom strictly prefers the other to his/her partner in the matching; otherwise, the matching is *stable*" [5]. The PII algorithm has $O(n \log n)$ complexity [5]. Korakakis simulated the Gale-Shapley (GS) and PII algorithms on MPI clusters [3]. Our claim is that these implementations are bottlenecked by the communication network between cluster nodes. The GPU should provide a significant speedup over these implementations by eliminating the bottleneck.

1. The PII Algorithm

The PII algorithm has two phases: the initialization phase and the iteration phase. The initiation phase entails generating a random matching. The iteration phase iteratively searches for new matchings in the hope of finding one with fewer unstable pairs. The goal is to iterate until the algorithm converges with no unstable pairs. The algorithm incorporates alternating between the two phases, to counteract cycling, until a stable match is found. The algorithm can be stopped at any point to output the current matching. Each step in the PII algorithm is parallel, so the algorithm can be implemented on parallel architectures such as MPI clusters and GPUs [5, 4, 3, 2].

2. PII on GPUs

Since the PII algorithm is parallel, it is natural to implement it on parallel architectures such as MPI clusters and the GPU. The GPU provides an advantage over the MPI cluster. MPI clusters are often constrained by the communication network bottleneck between nodes. Since the PII requires frequent communication between execution units, this bottleneck would be significant. In contrast, and to our advantage, the GPU has a large pool of high speed, low latency, global memory which is available to all execution units. For example, the effective memory clock rate of a NVIDIA GeForce GTX 770 is about 7000 MHz. This translates to a latency of 1.4×10^{-10} seconds (0.14 nanoseconds). A typical InfiniBand MPI cluster interconnect may have latencies on the order of milliseconds (10^{-6} seconds). This illustrates an advantage of four orders of magnitude with the GPU. This high speed, low latency pool of global memory should provide a low-latency means of communication between execution units [1]. This feature of the GPU, coupled with the large number of execution units on chip, makes us anticipate a significant speedup for the PII algorithm from MPI clusters to the GPU.

3. The CUDA Implementation

NVIDIA's Compute Unified Device Architecture (CUDA) provides an elegant API for implementing the PII algorithm [6]. Each thread of execution running on the card can be uniquely referred to in the kernels, the C functions which run on the card. This allows us to easily program the processor elements referred to in the algorithm to individual threads in the API. Pointers can easily be kept between processor elements and communicated in a low-latency manner, as described in the algorithm.

We implemented each step in the PII algorithm with one or more separate CUDA kernels. Most of these are launched with n^2 threads, while some are launched with n, depending on the constraints of the algorithm at that step. The CUDA C API organizes threads in to groups called blocks and the blocks in turn are grouped into grids. We implemented the kernels that were launched with n^2 threads by launching the kernel with n blocks per grid and n threads per block. While this provides a convenient way to program in the API, it limits the scalability of the implementation to $n = 1,024$. This is a result of the limit of $1,024$ threads per block by NVIDIA [6]. However, this is not a huge concern as NVIDIA's GPUs usually have between one and two thousand simultaneous executing units. That means that beyond a certain problem size, the GPU scheduler will have to begin serializing access for the logical threads to the physical execution units. The algorithm requires n^2 processors to achieve the $O(n \log n)$ complexity. However, in this scenario, the number of execution units is smaller than n^2, resulting in serialization. A real limit to the scalability of the GPU for this problem exists, due the limited number of execution units on the GPU. Our implementation should utilize the card quite well.

4. Results

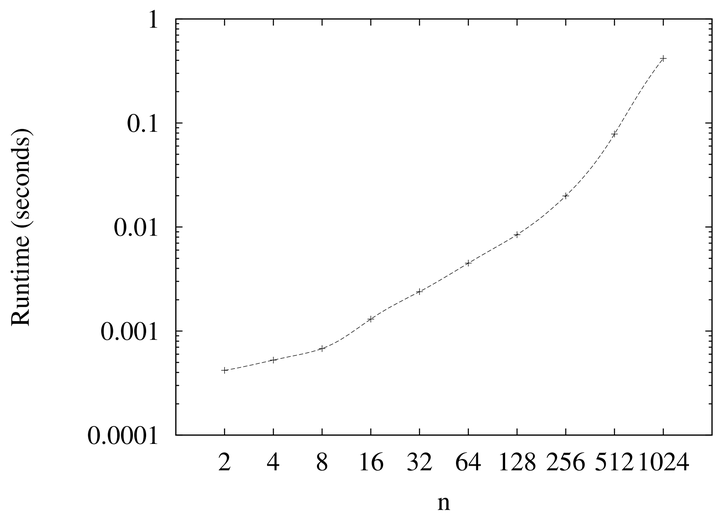

Figure 1: *n versus runtime (seconds)*

5. Conclusion

GPUs are readily available and cost-efficient ways to compute lots of data for throughput intensive applications. The massively parallel architecture does so much work that latency can even be improved by reducing the computation time of complex algorithms. The PII algorithm demands this kind of efficiency. It is not unreasonable to imagine a variants of PII algorithm running on a GPU-enabled router. CUDA provides a conveniently simple and powerful API to the hardware. OpenCL is another API to program chips, including those other than NVIDIA's and GPUs. Doubtlessly, there is much optimization to be made to the implementation we created. This may take considerably more time to develop. Map/Reduce/Hadoop might be a possible implementation platform for an adaption of the PII algorithm. Natural modifications to this implementation would to implement smart initialisation and cycle detection and limited preference lists [7].

Acknowledgment

The authors would like to thank the faculty at Salisbury University and the NSF for both time and money to support our experience this summer. We would like to express particular gratitude to Drs. Enyue Lu, Donald Spickler, Yuanwei Jin, and Arthur Lembo for their dedication to this summer's success. Funding from NSF CCF-1156509 under Research Experiences for Undergraduates Program (REU).

References

[1] Cook, S., *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, Morgan Kaufmann, 2012.

[2] Gale, D. and Shapley, L.S. *College Admissions and the Stability of Marriage*, American Mathematical Monthly 69, 9-14, 1962.

[3] Korakakis, E., *Examining the Parallelization Limits of the Stable Matching Problem*, Master's Thesis, University of Edinburgh, 2005

[4] Lu, E. *Parallel Algorithms for High Performance Switching in Communication Networks*, Dissertation, The University of Texas at Dallas, 2004.

[5] Lu, E. and Zheng, S. Q. *A Parallel Iterative Improvement Stable Matching Algorithm*, HiPC 213, LNCS 2913, 55-65, 2003.

[6] NVIDIA Corporation, *CUDA C Programming Guide*, `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, February 2014.

[7] White, C., Lu, E., *An Improved Parallel Iterative Improvement Algorithm for Stable Matching*, Extended Abstract, Companion of IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SuperComputing), Denver, CO, Nov. 2013.