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

Andrew Barkley[1], Justin Martin[2]
[1] *Undergraduate, SUNY Potsdam*, [2] *Undergraduate, Lock Haven University*

***Abstract*—We implemented a parallel algorithm for finding a stable marriage on massively parallel graphics processing units (GPUs). The algorithm is the Parallel Iterative Improvement (PII) Algorithm for the Stable Matching Problem proposed by Lu and Zheng in 2003 [5]. The algorithm has applications in real-time communication switching networks [4]. Global memory access for the many execution units on GPUs produces a speedup over what was possible on previously used achitectures.**

***Keywords*—parallel prcoessing, stable matching, stable marriage, graphics processing units, computer networks.**

I. Introduction

THE stable marriage 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]. In 1962, David Gale and Lloyd S. Shapley articulated an algorithm to solve the stable matching problem [2]. That algorithm works well for uniprocessor architecures, but due to the dependencies in its steps, it is difficult to adapt it for a parallel architecture. It has $O(n^2)$ complexity on one processor, which is not fast enough for applications like real-time network switching.

In 2003, Enyue Lu and S. Q. Zheng developed the PII algorithm for fully connected multiprocessors with $O(n \log n)$ complexity with n^2 processors [5]. Korakakis simulated the Gale-Shapley (GS) and PII algorithms on MPI clusters [3]. These implementations are bottlenecked by the communication network between cluster nodes. The GPU provides a significant speedup over these implementations by eliminating the bottleneck.

August, 2014

II. The PII Algorithm

The PII algorithm was originally intended for use in time-sensitive, communication network switch scheduling but might be useful in other applications where latency is a concern. It has two phases: the initiation 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 ultimate goals is to iterate until the algorithm converges with no unstable pairs. There exists the potential to cycle during the iteration phase. This would result in the failure to converge, so the algorithm incorporates going back and forth between the two phases until a stable match is found. The algorithm can be stopped at any point to output the current matching. The algorithm finishes in $O(n \log n)$, which is considerably better than the $O(n^2)$ of the GS algorithm. Each step in the PII algorithm is parallel, so the algorithm can be implemented on parallel architectures such as MPI clusters and GPUs [3]–[5].

Funded by NSF CCF-1156509 under Research Experiences for Undergraduates Program (REU). A. Barkley and J. Martin are REU students at Salisbury University.

III. 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, or 0.14 nanoseconds. A typical InfiniBand MPI cluster interconnect may have latencies on the order of milliseconds (10^{-6} seconds). This illustrates a potential 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], [6].

IV. The CUDA Implementation

NVIDIA's Compute Unified Device Architecture (CUDA) provides an elegant API for programming the GPU, which makes it easy to imlement the PII algorithm on it [7]. CUDA C is a set of extensions to the C programming language. CUDA developers write functions in CUDA C called kernels, which are `__global__` qualified functions which run on all execution units on a GPU at a time. Each thread of execution needed by the developer can be uniquely referred to in the

kernels. 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.

A CUDA thread is a single thread of execution, which is scheduled by the GPU at runtime. The API groups threads into groups called blocks and blocks into groups called grids. When the developer calls a kernel in their source code, they specify how many threads and blocks they want in these two groups. This can organize threads in a way that fits the thread model used in the body of the kernel definition, which will match the model of the problem being solved in some way.

We implemented each step in the PII algorithm with one or more separate CUDA kernels. Most of these are launched with n^2 threads, with n threads in a block and n blocks in a grid. This models the problem being solved well, even though limits the size of n that our implementation can solve. More precisely, this implementation will solve the problem to $n = 1,024$. This is a result of the limit of $1,024$ threads per block by NVIDIA [7]. Some kernels are launched with n threads. The number of threads launched depends on the constraints of the algorithm at that step.

The limit on the size of n 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, when n is sufficiently large, the number of execution units is smaller than n^2, resulting in serialization. A real limit ($n = \lfloor \sqrt{N} \rfloor$, where N is the number of cores, or execution units, on the GPU) to the scalability of the GPU for this problem exists, due the limited number of execution units on the GPU. It remains to be seen how much this limit will effect runtime for $n > 1,024$. Our implementation should utilize the card quite well, even if only for the brief moment it takes to solve the problem when $n = 1,024$.

V. Results

The timing mechanism we used to measure runtime used by the PII algorithm in our code was the `clock_gettime` function from the standard c header `time.h`, which has reliable nanosecond precision on a simetric multiprocessor if the processor affinity of the running process is locked to one execution unit. The form of timing we used with this function was monotonic time. We used this to measure the total elapsed time the algorithm took. We did not measure file I/O time in our measurement as this will vary greatly according to the hardware and implementation used and does not represent the execution time of the algorithm. Output was converted to seconds to give a relatable result.

The results show that as the problem size increases to the point where the demand for execution units exceeds the number of execution units available on the card the runtime dramatically increases, regressing to quadratic. This can be seen in 1 as the slope of the plot for t approaches the slope of the plot x^2.

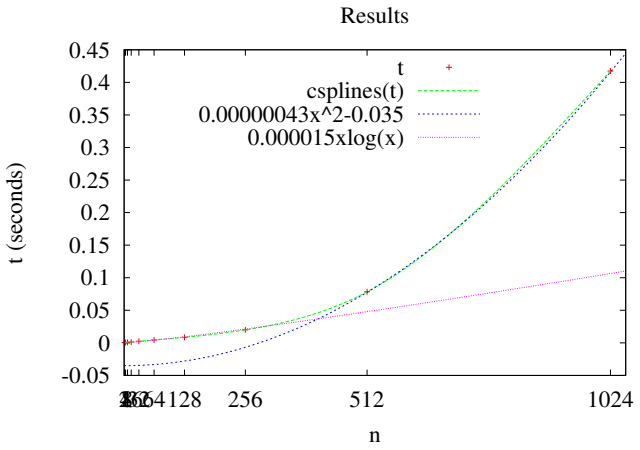

Fig. 1. n versus t (seconds)

At the problem size of 32, the algorithm demands $32^2 = 1,024$ execution units. The NVIDIA GeForce GTX 770 we used has $1,536$ cores, some of which are used by the card for it to do its job and some of which are being used to run the display. We expect there are $1,024$ cores available for the calculation though. If not, the card would be pretty inefficient, which it is not [1]. So, the number of execution units demanded by the algorithm at the problem size of 32, $1,024$, comes in quite under $1,536$, the number execution units available the card.

However, when $n = 64$, the algorithm demands $64^2 = 4,096$ execution units. This is far more than the number execution units on the card. This would explain a results jump in the runtime at $n = 64$, but that is surprisingly not the case, as 1 shows.

At $n = 32$ and needing $1,024$ execution units, we should just be saturating the card. Due to the fact that threads gets executed in warps of 32 parallel threads, the smaller problem sets would still be saturating the card, but many threads would be wasted not doing work [7].

We expected the execution time to increase dramatically when the problem size demanded more execution units than are available on the card, precisely when $n > \lfloor \sqrt{N} \rfloor$ where N is equal to the number of parallel execution units on the GPU. This would be the result of the necessity for the scheduler on the GPU to serialize access to the execution units because the algorithm is using more logical threads than execution units. Despite this, the incredible speedup provided by the GPU offset this situation between $256 <= n <= 512$.

Beyond $n = 256$, the card appears to be oversaturated with a demand for more exeucution units than it has. We know this because, as 1 shows, the slope of t becomes very close to the slope of $f(x) = x^2$. The runtime is now growing quadratically.

VI. 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 latencies 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. It is likely that future research could involve optimization to the implementation we created. This may take considerably more time to develop. An implementation that scales beyond $n = 1,024$ might also be of interest. Other natural modifications to this implementation are implementing smart initiation and cycle detection. It may also be possible to reformulate the entire problem into one which involves more limited preference lists [8].

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, Arthur Lembo, and Yuanwei Jin for their dedication to this summer's success.

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] Rehman, M. S., Kothapalli, K., Narayanan, P. J., *Fast and Scalable List Ranking on the GPU* ICS'09, Yorktown Heights, New York, USA, June 8–12, 2009.

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

[8] 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.