MULTI-GPU PARALLELIZATION OF IRREGULAR ALGORITHMS

by

Kristi Belcher

Thesis Supervisor:

Martin Burtscher, Ph.D.
Department of Computer Science

Approved:

Heather C. Galloway, Ph.D.
Dean, Honors College

MULTI-GPU PARALLELIZATION OF IRREGULAR ALGORITHMS

HONORS THESIS

Presented to the Honors College of
Texas State University
in Partial Fulfillment
of the Requirements

for Graduation in the Honors College

by

Kristi Belcher

San Marcos, Texas
December 2016

ABSTRACT

All programs possess a certain degree of irregularity in their control flow and memory access patterns. The more irregular a program is, the harder it tends to be to parallelize and port to accelerators such as Graphics Processing Units (GPUs). Additionally, efficient accelerator-based computing devices are rapidly spreading since they provide more performance and better energy efficiency than conventional computers. Multi-accelerator systems are already on the horizon and will likely be commonplace in the near future. Hence, it is important to learn how to efficiently run irregular computations on multi-accelerator platforms. I have rewritten four single-GPU programs, each with different amounts of irregularity, so that they can exploit multiple GPUs simultaneously. By analyzing shared variables and data dependencies within the programs, I was able to create a general approach for parallelizing programs across multiple accelerators. I then compared the performance of these codes against their single-GPU counterparts to determine the performance benefit and how irregularity impacts that benefit. My results show that mostly regular programs and programs that display control flow irregularity tend to obtain a significant performance boost. However, programs that display memory access irregularity tend not to gain any speedup from multiple GPUs.

ACKNOWLEDGMENTS

I want to use this section to thank Dr. Martin Burtscher, my adviser, for allowing me access to use the lab machines that have the CPUs and GPUs I used for my experiments. Dr. Burtscher advised me throughout my research, helping me refine my research project for accuracy and conciseness. I would also like to thank the Department of Computer Science for the opportunities and resources provided and Dr. Heather Galloway, Dr. Ron Haas, and the Honors College for providing the opportunity to complete the Honors Thesis course.

INTRODUCTION

Recently, usage of GPUs as general purpose accelerators has sharply increased [15]. GPUs are used in high-performance computing because of their high computational ability and energy efficiency. Computation-heavy algorithms are able to be processed in a reasonable amount of time with GPUs, opening up further opportunities to solve important problems. Because GPUs tend to be more energy efficient than the main processor [14], they are also becoming more prevalent in both PCs and handheld devices. Additionally, multi-accelerator systems are starting to be increasingly common as scaling becomes simpler with these systems. GPUs can be used to solve many different problems, both regular and irregular. A more regular program tends to behave in a statically predictable manner, making it easier to program and optimize for performance. For example, without knowing any of the inputs of a matrix-vector multiplication algorithm, I can still easily predict the program behavior. This is because there will be uniform calculations across the data structure regardless of inputs. However, a more irregular algorithm like an operation on a binary search tree is very different. The program behavior is largely dependent on the inputs given to it, making it unpredictable and harder to port to an accelerator. Irregular algorithms are very common and include codes that build, traverse, and update irregular data structures such as trees and graphs. Examples of irregular algorithms include optimization theory [4], social networks [5], system modeling [6], compilers [7], discrete-event simulation [8], and meshing [9].

Irregularity can be described by two main measures. The first kind of irregularity depicts the control flow of the program. Control Flow Irregularity (CFI) makes an algorithm's behavior statically unpredictable as the results are data dependent. We see CFI in *while* loops and *if* statements. That means that, depending on the input, the algorithm may need to run more or fewer iterations to find a solution. The amount of CFI in GPU code can be calculated by the following equation [1]:

$$\text{Control-Flow Irregularity (CFI)} = \frac{\text{divergent\_branches}}{\text{executed\_instructions}}$$

The second kind of irregularity describes the memory accesses of the program. Memory Access Irregularity (MAI) also makes the algorithm’s behavior statically unpredictable because, again, the results are data dependent and vary from input to input. We see Memory Access Irregularity in pointer chasing and other means of accessing memory. The amount of MAI can be calculated by the following equation [1]:

$$\text{Memory-Access Irregularity (MAI)} = \frac{\text{replayed\_instructions}}{\text{issued\_instructions}}$$

MAI and CFI increase data dependencies and thus require careful analysis to determine how to program around these obstacles. Thus, programs that possess these kinds of irregularity are more difficult to handle with accelerators, and even more so with multiple accelerators. Different programs can possess varying degrees of one or both types of irregularity, so it is important to understand the behavior of the code before mitigating the effects of irregularity on performance.

In order to use multi-accelerator systems and obtain good performance from irregular codes, I must program around the irregularity obstacles. Since GPUs are high-throughput devices, they are built for large numbers of computations with high data reuse [16]. Even though GPUs provide performance advantages through computation, sending data between the CPU and the GPU and accessing memory on the GPU remain the biggest bottlenecks. By optimizing the code to minimize communication and to use memory as efficiently as possible, it may be possible to obtain good performance on these systems.

APPROACH

The first algorithm I used is a fractal algorithm that calculates a picture (bitmap of pixels). This fractal code is fairly regular, except for some CFI. The CFI results in the load imbalance (when some threads have more work than others) because the darker pixels take more computation than the lighter pixels. The second algorithm I picked is the N-Body algorithm, which calculates the movement of each star in a cluster. For every "timestep" (iteration in the loop), I need the current position, mass, velocity, and acceleration of each star in the cluster and then I can calculate where that particular star will move next (and update its position) based on the gravitational force of the other stars. This algorithm is mostly regular. Next, I used the Maximally Independent Set (MIS) algorithm, which helps, for example, in the parallelization of other codes by determining a set of independent tasks that the threads can work on without the need to synchronize with other threads. The MIS algorithm has both MAI and CFI. Finally, I have the Connected Component (CC) algorithm, which is used as a pre-processing step for many other graph algorithms. The CC algorithm finds all reachable vertices for every vertex. This algorithm also displays irregularity in both dimensions (MAI and CFI).

For each algorithm, I have a certain input, or group of inputs, that I picked to ensure that the GPUs have enough data to be fully loaded. It is important for the GPUs to be fully loaded to get the best performance. For the Fractal algorithm, I compute a subset of the Mandelbrot set with 39357 by 39357 pixels and 256 levels of gray. I used that particular width because that maximally loads the GPUs I am working on. Additionally, 256 shades of gray gives a nice distribution of light and dark pixels for computation. For the N-Body algorithm, I use 200,000 bodies (or stars) and 10 time steps. For both the MIS and the CC algorithms, I use a USA road map with 23 million vertices and 58 million edges. This input has one connected component with 45% of the vertices in the MIS. I also use a UK web domain graph with 18 million vertices and 523 million edges. I determined that this input has 1,059 connected components and 56.9% of the nodes are in the MIS.

Since the GPU is a co-processor and not a standalone processor like the CPU, GPUs cannot operate by themselves. Instead, they need to interact with the CPU to be given data and to send data back. Figure 1 illustrates how a typical multi-GPU system sends data. If one of the GPUs of the system needs to send data to the other GPU, it must first go through the PCIe bus to get to the CPU. Then the CPU will send the data back through the PCIe bus to the other GPU.

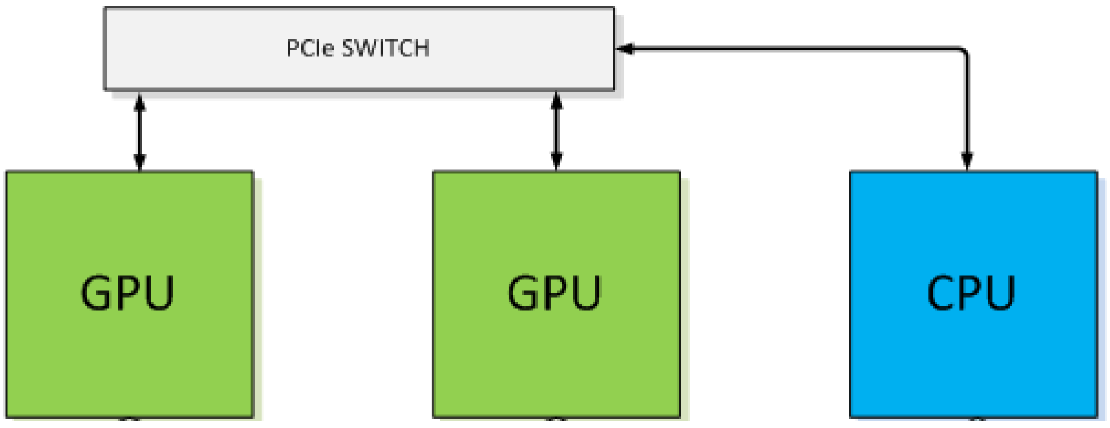

Fig. 1: A multi-GPU system

With multiple accelerators, there will be a certain amount of unavoidable communication, since data has to be sent from the CPU to the GPU and then from one GPU to another for certain algorithms. Since communication is relatively very time consuming, it is important to minimize and hide the communication as much as possible. Additionally, sending the data from one GPU to the other is much slower than accessing the data once it is there, which increases the need to optimize the codes around it.

When converting an algorithm from single-GPU to multi-GPU, there are certain things that must be considered. For example, I must determine which variables need to be shared between the GPUs. These variables need to be sent from one GPU to the other for correctness and therefore we will need GPU to GPU communication. As I studied the shared variables within the programs, I determined if they were read-only, write-only, or read/write variables. Depending on what kind of shared variable it is, I handle it differently. For example, a shared variable that is read-only will need to be handled differently than a variable that is read/write and might require synchronization, etc. In Figure 2, I show a detailed diagram of my implementation for parallelizing the MIS code on multiple GPUs. Let *k* be equal to the number of nodes in the input graph

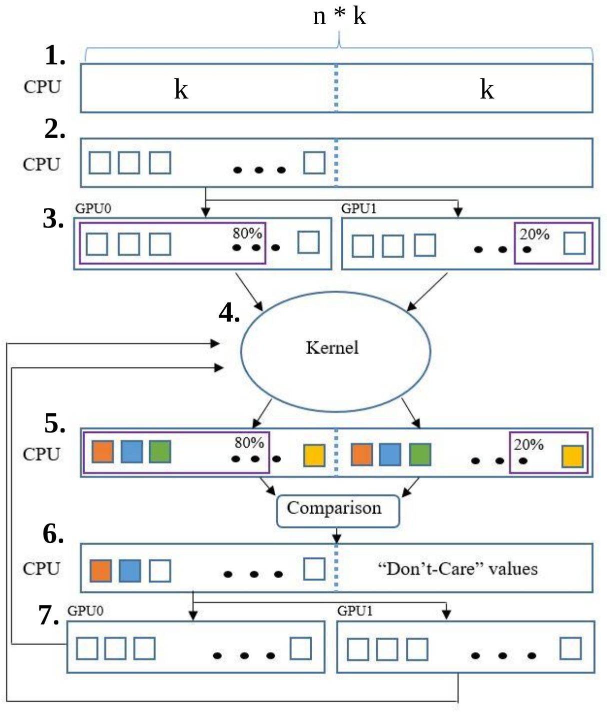

Fig. 2: Detailed depiction of the implementation to parallelize the MIS code on multiple GPUs. In this case, there are 2 GPUs with the first getting 80% of the work and the second getting 20%.

and let *n* be the number of GPUs I am working with. Then, to correctly parallelize the MIS code, I implemented the following steps as outlined in Figure 2:

1. Allocate an array on the CPU with *k* * *n* elements. However, each GPU just gets an array that is *k* elements long.
2. Initialize the data for all *k* nodes.
3. Send the initialized data to each GPU.
4. Each GPU runs a kernel to calculate its own portion of the data. In this example, one GPU processes 80% of the data and the other processes 20%.
5. Each GPU sends its entire array to the CPU.
6. The CPU merges the data. In the merge, the CPU compares all the values sent from one GPU to the values from the other GPU, looking to see if any GPU determined a node to be "in" or "out" of the set.
7. The consolidated data of *k* elements in size are then sent back to each GPU for the next round of kernel calls. This process repeats until no more updates are made.

I similarly created implementation models for the remaining codes that I worked with. Additionally, there are other considerations when converting to multi-GPUs such as how and when to switch between the GPUs and whether or not to use asynchronous operations and wait statements.

RELATED WORK

Previous work by Burtscher *et al.* [1] gave a formula to calculate MAI and CFI and was the first to show were programs lie in that 2D space. Using this work, I was better able to understand the performance of the four algorithms that my research dealt with specifically. However, a number of papers study irregularity in GPU codes and suggest optimizations to reduce that irregularity. Wu *et al.* studied the source of control flow irregularity [18]. Zhang *et al.* proposed techniques to eliminate CFI and MAI [19]. Hetherington *et al.* demonstrates that even with the presence of CFI, algorithms can achieve good performance on GPUs [21]. Through my experiments, I found the same to be true on my multi-GPU codes as well. Additionally, some papers study the hardware improvements that can be made to be able to better handle irregularity on GPUs [17, 20, 22, 23].

Other works [2, 3] have studied multi-GPU implementations that differ from this approach in a few key ways. Yang *et al.* [2] studied a hybrid implementation of a mostly regular algorithm, matrix multiplication, by experimenting with using MPI and/or OpenMP in addition to the GPU. Zhai *et al.* [3] studied a non-oscillatory central mesh computation using 1 to 3 GPUs and then comparing it to using a CPU with OpenMP. Although the implementation is given explicitly in a list, there is no mentioning of source code optimizations to get the most out of GPU parallelization. The paper concludes that in some experiments the multiple GPU approach was better and in others the CPU approach was better, but did not explain why. It is clear from previous works that there is much work to be done to fully understand multi-GPU behavior.

Due to the increasing utilization of GPUs to parallelize different algorithms, understanding the implications of irregularity on multi-GPU performance is very important to study. In turn, such insights can lead to more scientific knowledge. To the best of my knowledge, this is the first in-depth analysis of the effect of irregularity on the multi-GPU performance of different algorithms. I analyze programs with a broad range of irregularity (from mostly regular to very irregular) to understand the resulting impact on performance.

METHODOLOGY

In order to get a variety of results, I tested on three different machines to obtain our measurements. One machine has two Titan X GPUs [10]. The second machine has two K20 GPUs [11], and the third has GTX 680 [13] and GTX 480 [12] GPUs. Hence, some of my measurements were on two identical GPUs like the two K20s or the two Titan Xs. Other measurements were on different GPUs, which I could get from using the GTX 680 and GTX 480 or the Titan X and K40 GPUs. I then evaluated various combinations of percentages of the workload assigned to each GPU. When using two GPUs, I ran the programs using 100% and 0%, 99% and 1%, 98% and 2%, etc. to test every possible combination. The user can determine the percentages (for one, two, or more GPUs, assuming the system is appropriately equipped with that particular number of GPUs) simply by specifying appropriate command-line parameter. For example, to run the fractal code with 39357 width, 256 depth, and 50% on one GPU and 50% on the other, you could run it with "./fractal 39357 256 50 50". In the codes, I have error checks to ensure that the percentages add up to 100.

For every program, I must first determine how to successfully partition the data between the GPUs. To do that, I look at shared variables between GPUs and what data must be sent from one GPU to the other. Once I have distinguished which variables are read-only, write-only, and read-write, I can analyze how much data needs to be sent via these variables. In case of the fractal code, since the algorithm is determining pixel values on a bit-map, no communication between GPUs is required. The shared variable is write-only and each thread will write to its own spot in the array. For the N-Body code, since in every time step the GPUs need access to all current position data to calculate position values for its particular portion of the work, I send the entire position data for all the bodies to each GPU. As they calculate updated position values for the bodies, I have to resend all the position data for the next time step. In the MIS code, the entire node status array – which holds information about whether the node is in the maximal independent set or not – is given to each GPU since it needs to be able to access that information to calculate updated values. After each GPU has the updated data, it sends the entire node status array to the CPU. The CPU then consolidates the array to where it contains all the updated values from each GPU. From here the entire updated node status array is sent back to each GPU for the next round of computations. Similar to the MIS code, the CC code also must have access to all the node status values (for the CC code, the node status array contains information about whether or not a particular node is in the connected component or not). With each iteration, the GPUs will compute their portion of the data and then send those updated values to be merged. If there is still work to be done, the GPUs will receive the entire updated node status values for the next iteration of computation.

EXPERIMENTAL RESULTS

In the charts below I show results of running the four algorithms with two GPUs. For each algorithm, I have a corresponding input. Figure 3 shows the results of running the Fractal with 39357 width and 256 depth. Next, in Figure 4, I show the results of running the N-Body code with 200,000 bodies and 10 time steps on two GPUs. Again, I choose these numbers because they fully load the GPUs. In Figure 5 and Figure 6, I show the results of running the MIS code with Figure 5 showing the UK input graph and Figure 6 showing the USA input graph results. In Figure 7, I show the results of using three GPUs to compute the MIS code. In this chart, I use the WEST input, another road map graph with 6 million nodes and 15 million edges. Here we see that the multi-GPU codes can work for more than just two GPUs, but we still see a similar trend with the results. There is no optimal workload distribution as the single GPU implementation is the best. In Figure 8 and Figure 9, I similarly show the results of running the CC code with Figure 8 displaying the UK input graph and Figure 9 showing the USA input graph.

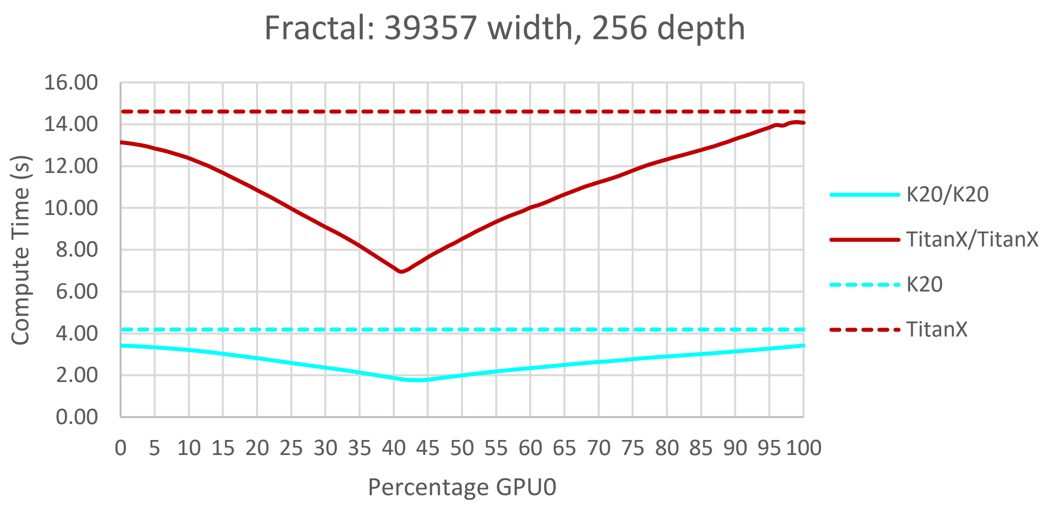

Fig. 3: Experimental results of running the Fractal code with 2 GPUs

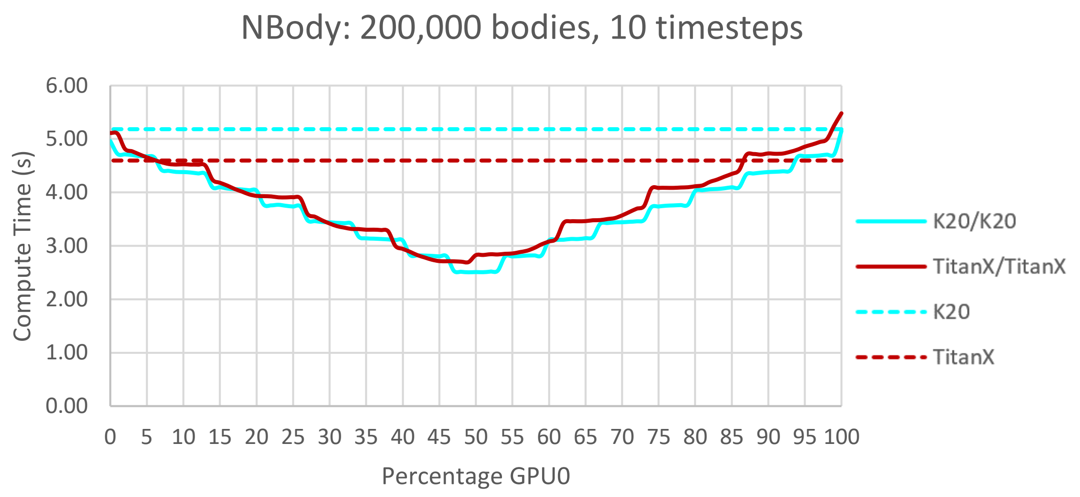

Fig. 4: Results of running N-Body code with 2 GPUs

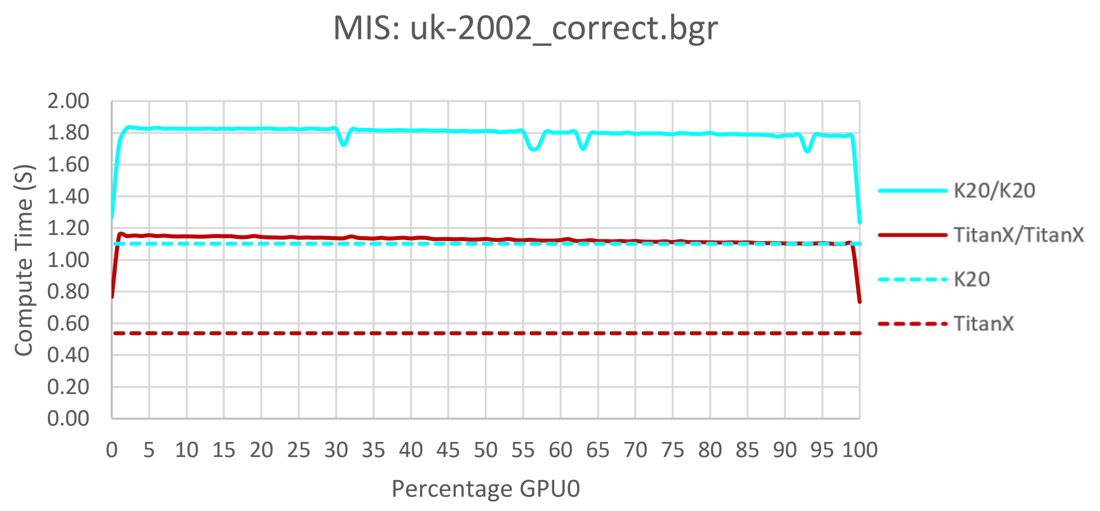

Fig. 5: Results of running the MIS code with the UK input

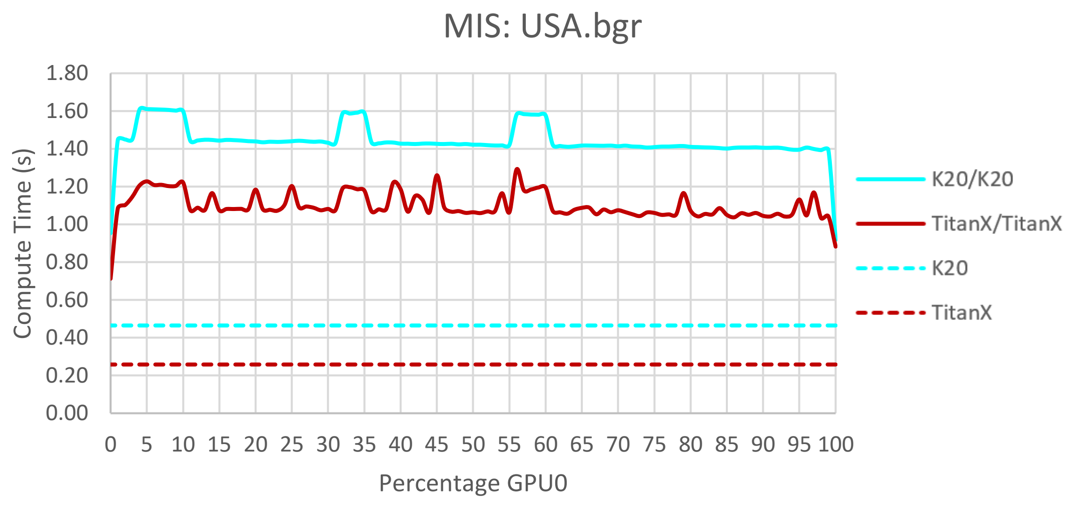

Fig. 6: Results of running MIS code with the USA input file

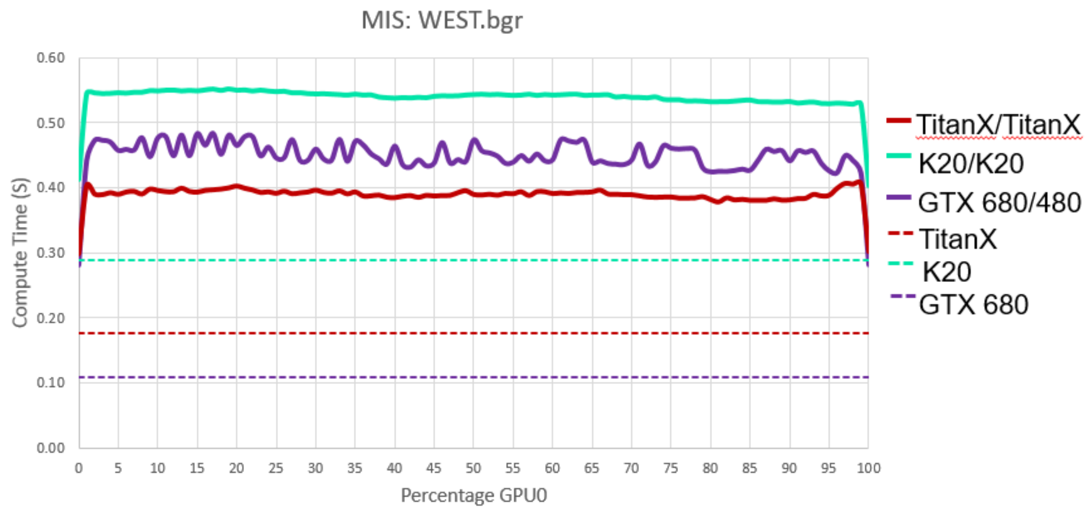

Fig. 7: Results of running the MIS code with the WEST input, another road map, with three GPUs

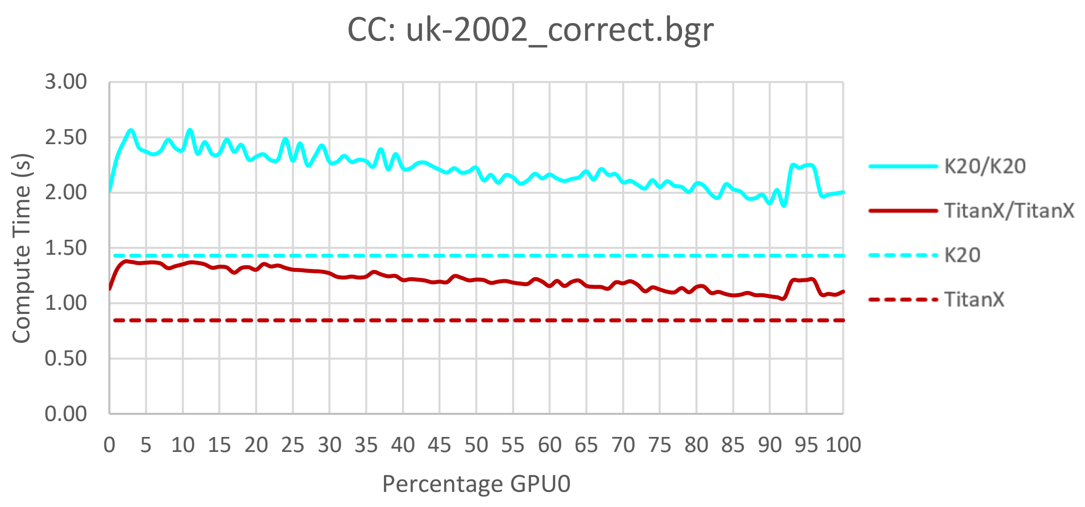

Fig. 8: Results of running the CC code with the UK input

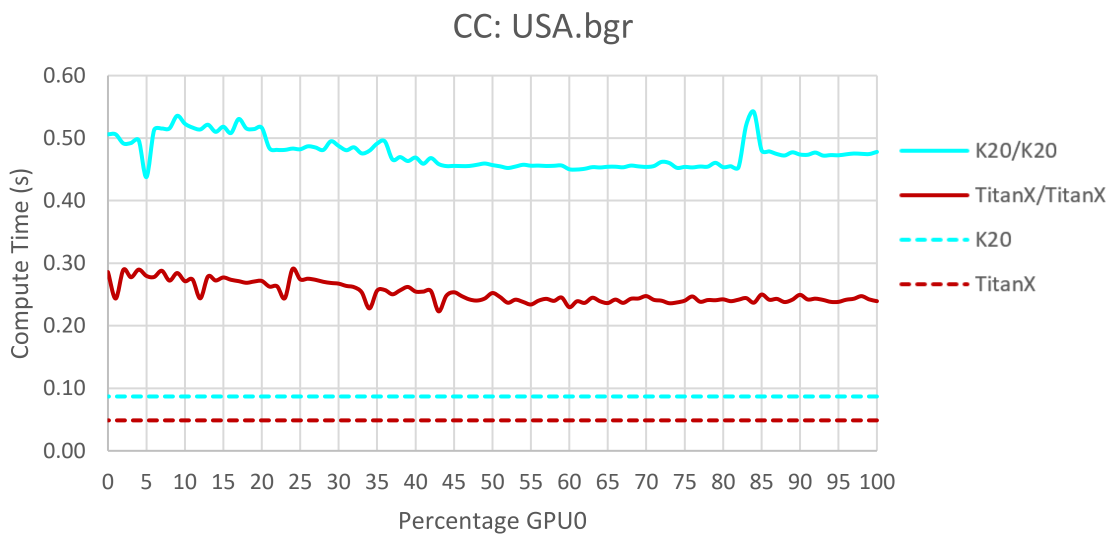

Fig. 9: Results of running the CC code with the USA input file

From the Fractal and N-Body results, we see that there is a clear “sweet-spot” for work-load distribution between GPUs. I get considerable performance gain with both of these algorithms with 1.90 speedup on the Titan X and 1.98 on K20 with the N-Body code. With the Fractal code, I get 1.89 speedup on the Titan X and 1.94 on K20. In fact, there is an optimal workload distribution at 49% and 51% between two GPUs for the N-Body code, but the optimal distribution for the Fractal is at 44% and 56% or 41% and 59% (depending on which GPU is used). Having an optimal distribution close to 50% and 50% with the N-Body makes sense since it is a mostly regular algorithm. However, the reason why the distribution is not also closer to 50% and 50% for the Fractal is because of the Control Flow Irregularity the code possesses.

In contrast, we see that there is no clear optimal workload distribution for the MIS and CC codes. Additionally, I obtain no speedup for these codes using two GPUs. No matter if I am using two GPUs or three, we see the similar behavior that there is no performance gain. The Memory Access Irregularity that both of these codes exhibit seems to be too much of a burden on performance and, therefore, I do not gain any performance when using two GPUs.

Since the Fractal code possesses Control Flow Irregularity but still achieves a considerable speedup, it seems that CFI does not hurt performance that much. However, the CC and MIS codes possess both CFI and MAI, and do not show any speedup nor optimal workload distribution. Therefore, it appears that MAI has more of an impact on performance than CFI because it increases the need for expensive and slow data transfers between GPUs.

CONCLUSIONS & FUTURE WORK

As my results illustrate, the irregularity that a program may or may not possess greatly influences the speedup and the best GPU percentage. For example, the N-Body code, which is mostly regular, has a best percentage of 49%/51% on two GPUs. The Fractal, however, has a best GPU percentage of 44%/56% and 41%/59% because of the Control Flow Irregularity present in the code. For both the MIS and the CC codes, the single GPU works best. It can therefore be inferred that Memory Access Irregularity, which both the MIS and CC algorithms possess, increases the need for slow data transfers and thus makes it difficult to obtain speedup. However, the speedup attained on the regular codes is very substantial. On N-Body, I got 1.90 speedup on the Titan X's and 1.98 on the K20's. For the Fractal, I got 1.89 speedup on the Titan X's and 1.94 on the K20's. Again, this suggests that Control Flow Irregularity does not impact performance as much as Memory Access Irregularity.

My codes work for any number of GPUs and also for different types of GPUs within the same system. Additionally, for big problem sizes, regular codes could definitely benefit, as can be seen by the speedups I obtained. However, using multiple GPUs for irregular codes may not be worth it because of the effects of Memory Access and Control Flow Irregularity.

Future work can be done to try to mitigate the effects of communication on those algorithms for speedup. For example, Peer-to-Peer (P2P) direct GPU communication may help reduce the impact of communication and MAI. Additionally, using compressed messages (where I use data compression to send less data), and/or a fixed-point approach (where I hold off on communicating until it is absolutely necessary) may also help. This future work can serve as a way to better understand how to use these approaches for multi-GPU programming – particularly P2P programming, which is a relatively new technique. I hope that my findings will help others better understand the behavior of irregular codes when applied to multi-GPU systems.

REFERENCES

[1] M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. 2012 IEEE International Symposium on Workload Characterization, pp. 141-151. November 2012.

[2] C-T. Yang, C-L. Huang, C-F. Lin. Hybrid CUDA, OpenMP, and MPI Parallel Programming on Multicore GPU Clusters. 2011 Elsevier Computer Physics Communications, pp. 266-269. July 2010.

[3] J. Zhai, W. Liu, L. Yuan. Solving Two-Phase Shallow Granular Flor Equations with a Well-Balanced NOC Scheme on Multiple GPUs. 2016 Elsevier Computers and Fluids, pp. 90-110. May 2016.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, McGraw Hill, 2001.

[5] Kirsten Hildrum and Philip S. Yu. Focused Community Discovery. In *International Conference on Data Mining*, 2005.

[6] James L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3):223–252, 1977.

[7] A. Aho, R. Sethi, and J. Ullman. *Compilers: principles, techniques, and tools*. Addison Wesley, 1986.

[8] Jayadev Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, 1986.

[9] L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the Symposium on Computational Geometry* (SCG), 1993.

[10] GeForce GTX Titan X. http://www.geforce.com /hardware/desktop-gpus/geforce-gtx-titan x/specifications. 2016.

[11] Tesla K20. https://www.nvidia.com/content/PDF /kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf. 2012.

[12] GeForce GTX 480. http://www.nvidia.de/docs/ IO/90203/GTX-480-470-Web-Datasheet-Final4.pdf. 2010.

[13] GeForce GTX 680. http://www.nvidia.com/ content/PDF/product-specifications/Ge-Force_ GTX_680_Whitepaper_FINAL.pdf. 2012.

[14] Song Huang, Shucai Xiao, and Wu chun Feng. On the energy efficiency of graphics processing units for scientific computing. In 23rd IEEE International Symposium on Parallel and Distributed Processing, pages 1–8, 2009.

[15] David B. Kirk and Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2010.

[16] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro, 28:39–55, 2008.

[17] Wilson W. L. Fung and Tor M. Aamodt. Thread block compaction for efficient SIMT control flow. In Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture, pages 25–36, Washington, DC, USA, 2011. IEEE Computer Society.

[18] Haicheng Wu, Gregory Diamos, Si Li, and Sudhakar Yalamanchili. Characterization and Transformation of Unstructured Control Flow in GPU Applications. In First International Workshop on Characterizing Applications for Heterogeneous Exascale Systems, 2011.

[19] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 369–380, New York, NY, USA, 2011. ACM.

[20] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[21] T.H. Hetherington, T.G. Rogers, L. Hsu, M. O'Connor, and T.M. Aamodt. Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. In IEEE International Symposium on Performance Analysis of Systems and Software, pages 88–98, April 2012.

[22] Roman Malits, Evgeny Bolotin, Avinoam Kolodny, and Avi Mendelson. Exploring the limits of GPGPU scheduling in control flow bound applications. ACM Transactions on Architecture and Code Optimization, 8(4):29:1–29:22, January 2012.

[23] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In Proceedings of the 37th Annual

International Symposium on Computer Architecture, pages 235–246, New York, NY, USA, 2010. ACM.