Dynamic Load Balancing for Heterogeneous Systems

Kenneth Lee, Kyle Morgan

***Abstract*—Leveraging maximimum performance from specialized hardware is one of the major problems facing computer scientists today. General-purpose computing on graphics processing units (GPGPU) is a technique that is becoming increasingly popular for handling computations typically performed by the CPU. However, many GPGPU algorithms place the entire workload on the Graphics processing unit (GPU) leaving the CPU idle for the duration of the computation. This paper investigates the effect various load-balancing schemes across the CPU and GPU have on overall application performance. Additionally, the discrete and fused GPU architectures, and their impact on performance, will be investigated.**

I. Introduction

LEVERAGING the maximum performance from specialized hardware is one of the major problems facing computer scientists today. The unique architecture of the GPU gives it a place as a massively parallel hardware accelerator. Programming frameworks such as OpenCL allow programmers to utilize the stream processors of the GPU for non-graphics data. However, because of the difficulty in programming for this hardware, most of the effort is spent on trying to achieve performance from the GPU, while the CPU is left unused. By also making use of CPU to process a chunk of the data, we can achieve higher utilization of resources and therefore increase overall program performance.

GPU+CPU co-processing hardly a new idea. Jimenez et al. [7] present a dynamic library which can be used in a system to dynamically schedule tasks for GPU computation across various processes. The method described only looks at optimizing usage of the GPU across the whole system, rather than optimizing a single applications performance. The StarPU system, on the other hand, presents a method of co-processing and load balancing on a per application basis [1]. They show that by using an appropriate model for the load balancing, they are able to achieve improvements in speed for applications which call for code acceleration multiple times. We intend to extend this work to improve the speed of each kernel execution by splitting the work of that kernel over the capable resources, similar to the schemes used by OpenMP [4]. Both of these papers address data transfer times as a major cost of GPU computing. Using the new AMD Fusion architecture, which fuses the CPU and GPU onto a single die, we may be able to eliminate or greatly reduce this cost, which will impact the overhead of GPU+CPU co-processing.

We expect that by using GPU+CPU processing we will improve the performance of the application, when compared to CPU-only or GPU-only. That is, consider a workload of size n, a GPU that requires t_g time to perform one block of computation, and a CPU that requires t_c time to perform one block of computation. We expect that the GPU would require time nt_g to perform a given computation, while the CPU would require nt_c time to perform the same computation. Optimally, we would expect the load-balancing scheme to require the following time to perform the computation.

$$\max\{(n-m)t_g, mt_c\} + t_d + t_o$$

Where m is the amount of the workload given to the CPU ($m \leq n$), t_d is the data transfer cost, and t_o is any overhead incurrent from synchronization costs of the load balancing scheme or otherwise. The above will be faster than just the GPU iff $t_g - \max\{(n-m)t_g, mt_c\} < t_d + t_o$.

Two different load balancing schemes were implemented for this paper. The first was a static load balancer, which partitions the workload in two, and transfers one chunk to the CPU, and the other to the GPU. The CPU and GPU work entirely independently, and their results are merged once each finish. The static load balancer should have no synchronization overhead, and transfers all of the data prior to the computation. The main caveat with a static load balancer is that the ratio of the workload given to the CPU is chosen by the load balancer. An optimal ratio would be one such that the time to perform the computation is equal for both the CPU and GPU. Unfortunately, this optimal ratio will be application-dependent, meaning that a fixed ratio cannot be optimal for every application. In order for a static load balancer to be optimal for a given application, a programmer must tune the amount of workload given to the CPU that works best for that application.

The second scheme is a dynamic load balancer, which partitions the data into many smaller chunks. One chunk is sent to the CPU and GPU each for computation. When one finished, it requests another chunk from the load balancer, and this process repeats until the entirely workload has been sent to either the CPU or GPU. The dynamic load balancer will incur some amount of synchronization overhead, because shared variables that track which chunks of the workload are left must be protected by a lock. Unlike the static load balancer, the dynamic load balancer transfers data throughout the computation. The dynamic load balancer is designed to mitigate the primary problem with the static load balancer - that the optimal ratio of computation performed on the CPU is application-dependent. A chunk size must be chosen such that the processing units are not requesting chunks so frequently that contention for shared variables rise and synchronization overhead increases, yet small enough to not

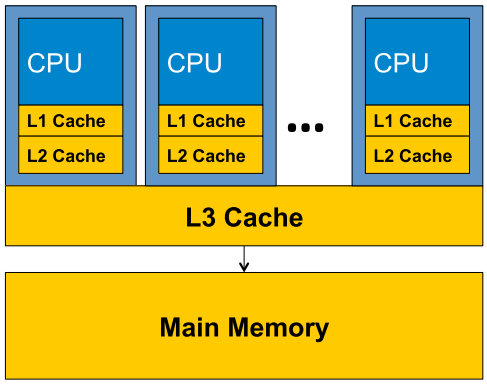

Fig. 1. CPU Architecture

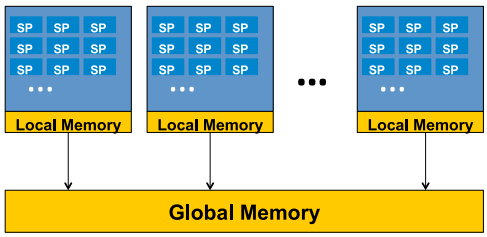

Fig. 2. GPU Architecture

allow one processing unit to receive a disproportionate amount of work.

II. Architecture Overview

In this section we discuss the different architectures of the experimental machines under test. We will discuss CPU, GPU, and APU architectures in terms of major differences that affect the performance of the device.

A. CPU Architecture

The CPU architecture is designed as a low latency architecture. To facilitate the needs for low latency application performance, a series of hierarchical caches have been designed to hide memory access costs, the most common cause of latency in single-threaded applications on modern architectures. Each core of the CPU typically only runs one or two hardware-enabled threads to reduce cache pollution, thus avoiding latency. An example CPU architecture is given by Figure 1.

The use of the Single Instruction Multiple Data (SIMD) paradigm enables increased performance on these devices by increasing data throughput for data parallel applications.

B. GPU Architecture

The GPU almost acts a foil to the CPU architecture. Instead of optimizing the architecture of the device to increase performance of a single thread, the GPU architecture is specifically designed for large throughput performance.

The memory hierarchy for the GPU architecture is given by Figure 2.

C. APU Architecture

The APU is a relatively new, novel architecture that attempts to improve on the the discrete CPU+GPU system. The CPU die is split between the typical CPU processor as well as allocating space for a SIMD engine, the basis for a GPU processor. The APU has a shared memory between the CPU and SIMD engine, in which accesses are facilitated through a high-speed memory controller. The AMD Fusion technology is an example of this type of architecture [**?**].

One of the main benefits of the APU is the elimination of the PCI-e bus, which allows for much faster transfer between CPU and GPU. Daga et al. show that the AMD Fusion architecture allows for much faster data transfers between CPU and GPU, allowing for an overall improvement in runtime, despite reduced compute power on the GPU [3].

III. Load Balancing Benchmarks

For this paper, three different benchmarks were used to assess the effectiveness of the load balancer. The first is a reduction benchmark, in which the sum of variably-sized array is computed. The reduction benchmark is unique in that different code is being run on the CPU and the GPU. The reason for this is that the GPU cannot efficiently compute a reduction the way a CPU typically would – due to the relative innefficiency of backward branches on the GPU. A reduction on the GPU is a multi-stage process, in which the output of one step is used as the input of the next. Each steam processor computes the reduction of two elements during each iteration, reducing the amount of work during each step by a factor of two. As a result, each step utilizes half the stream processors of the previous step. This continues until the reduction operation is only done by a single stream processor. A natural consequence of this algorithm is that not all of the stream processors will be utilized during the computation.

The next benchmark is a simple vector addition operation that computes the sum of two input vectors, and stores the result into an output vector. The third benchmark, referred henceforth as the "Vector Add+" benchmark, is similar to the Vector Add benchmark, but it computes 1000 additions before storing the result in the output vector. The motivation behind this benchmark is to reduce memory access overhead with more computation.

IV. Results

Consider Figure 3, where the amount of data executed on the GPU is varied between 90% and 100%. The total execution time including data transfers has an obvious downward trend as more data is executed on the GPU. However, there are a couple of static ratios which yielded better performance than executing all of the data on the GPU. Most notably, when only 98.6% of the data was executed on the GPU. In this instance, the overall execution time was 325.27 seconds, compared to 333.31 seconds with only the GPU – a 2.4% speedup.

Next, consider Figures 4, 5, and 6. The figures demonstrate a variety of different load balancing options for the Reduction, Vector Add, and Vector Add+ benchmarks, across data sizes ranging between 1 MB and 16 MB. Each figure depicts

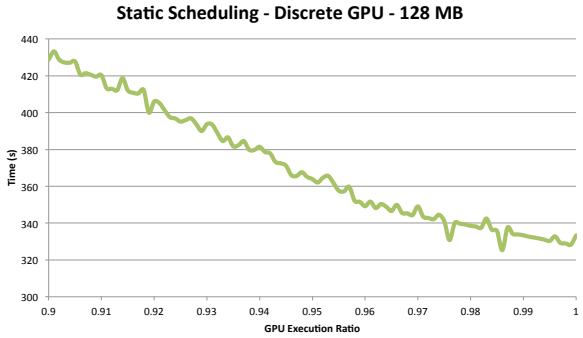

Fig. 3. Varying Data Ratio with a Discrete GPU

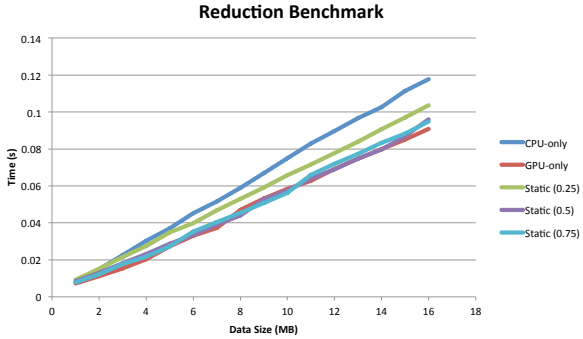

Fig. 4. Reduction Benchmark on an APU Architecture

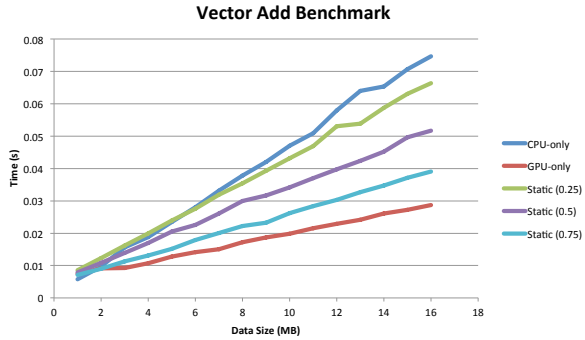

Fig. 5. Vector Add Benchmark on an APU Architecture

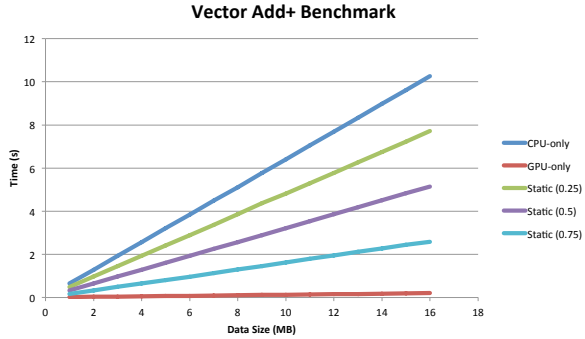

Fig. 6. Vector Add+ Benchmark on an APU Architecture

the total execution time for the CPU-only and GPU-only computations, as well as static load balancing with 25%, 50%, and 75% of the data on the GPU. In each of the figures, it's evident that the GPU performs significantly faster than the CPU for the given benchmarks, with the static partitioning schemes lying somewhere between the two.

V. Analysis

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

VI. Related Work

Ryoo et al. discuss optimization strategies for GPU versus CPU runtime systems [8]. However, their research is based on a GPU-only or CPU-only implementation instead of a heterogeneous approach. The work presented in [6] describes a runtime and compiled framework for improving memory access latency and reducing data transfer overhead for GPU kernels. Our work differs from this, because the kernel execution was still being performed soley on the GPU, whereas our work investigates kernel execution on multiple devices.

Computation via heterogeneous systems is a relatively new research area for high performance computing. The various frameworks and large number of different accelerator devices can make this quite difficult for application programmers to manage. The StarPU system [1] presents a framework to allow for programmers to write numerical codelets and have them automatically run on a heterogeneous system. Scheduling work for this system is one of the most difficult parts of the system, because of the various overheads for work-sharing schemes. We simplify this work, by looking at a two-device system, and focus on the performance characteristics of different scheduling scheme. Verner et al. use this approach to create a high-throughput low-lantecy heterogeneous encryption system [9].

Jimenez et al. [7] present a dynamic library which can be used in a system to dynamically schedule tasks for GPU computation across various processes. The method describes only looks at optimizing usage of the GPU across the whole system, instead of optimizing a single application's performance.

VII. Conclusion

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

References

[1] C. Augonnet, S. Thibault, R. Namyst, and P.A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures" in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, 2009, pp. 863-874.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing" in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44-54.

[3] M. Daga, A.M. Aji, and W. Feng, "On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing" in *Application Accelerators in High-Performance Computing (SAAHPC)*, 2011, pp. 141-149.

[4] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming" in *IEEE Computational Science and Engineering* 1998, pp. 4655.

[5] A. Danalis, G. Marin, C. McCurdy, J.S. Meredith, P.C. Roth, K. Spafford, V. Tipparaju, and J.S. Vetter, "The Scalable Heterogeneous Computing (SHOC) benchmark suite" in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 63-74.

[6] J.A. Jablin, P. McCormick, and M. Herlihy, "Scout: High-Performance Heterogeneous Computing Made Simple" in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, 2011, pp. 2093-2096.

[7] V.J. Jimnez, L. Villanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro, "Predictive Runtime Code Scheduling for Heterogeneous Architectures" in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, 2009, pp. 19-33.

[8] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA" in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2007, pp. 73-82.

[9] U. Verner, A. Schuster, and M. Silberstein, "Processing data streams with hard real-time constraints on heterogeneous systems" in *Proceedings of the international conference on Supercomputing*, 2011, pp. 120-130.

APPENDIX
TEAM MEMBER CONTRIBUTIONS

Kenneth Lee did A, B, C. Kyle Morgan did X, Y, Z.