An in-depth performance analysis of irregular workloads on VLIW APU

by

Matthew James Doerksen

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
May 2014

© Copyright 2014 by Matthew James Doerksen

Thesis advisor

Dr. Parimala Thulasiraman

Author

Matthew James Doerksen

An in-depth performance analysis of irregular workloads on VLIW APU

Abstract

Heterogeneous multi-core architectures have a higher performance/power ratio than traditional homogeneous architectures. Due to their heterogeneity, these architectures support diverse applications but developing parallel algorithms on these architectures can be difficult. In implementing algorithms for heterogeneous systems, proprietary languages are often required, limiting portability. Although general purpose graphics processing units (GPUs) have shown great promise in accelerating the performance of throughput computing applications, it is still limited by the memory wall. The memory wall can greatly affect application performance for problems that incorporate amorphous parallelism or irregular workload. Now, AMD's Fusion series of Accelerated Processing Units (APUs) attempts to solve this problem. The APU is a radical change from the traditional systems of a few years ago. This design change enables consumers to have a capable CPU connected to a powerful, compute-capable GPU using a Very Long Instruction Word (VLIW) architecture.

In this thesis, I address the suitability of irregular workload problems on APU architectures. I consider four scientific computing problems of varying characteristics and map them onto the architectural features of the APU. I develop several software

optimizations for each problem by making effective use of VLIW static scheduling through techniques such as loop unrolling and vectorization. Using AMD's OpenCL profiler, I analyze the execution of the various optimizations and provide an in-depth performance analysis using metrics such as kernel occupancy, ALUFetchRatio, ALUBusy Percentage and ALUPacking. Finally, I show the effect of register pressure due to vectorization and the limitations associated with the APU architecture for irregular workloads.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to begin by thanking my advisor, my committee, my parents, and all the people who have supported me along the way while completing my research.

Additionally, I would like to thank Michael S. Doyle for the Michael S. Doyle Graduate Fellowship award which aided in my research and furthering my education.

Chapter 1

Introduction

As computing has progressed, we have gone from simple, fixed function hardware to general purpose computing cores, capable of handling nearly any task given to them. However, today, these single cores simply aren't powerful enough to run the large simulations and calculations enterprises require. The first method of increasing performance was by adding more arithmetic hardware and increasing clock speed.

While performance did benefit, hardware constraints meant three walls were encountered: frequency, power and memory. First, as frequency increased, power consumption increased linearly, leading to physical limitations in terms of clock speeds (though research is being done with new materials such as graphene which may scale into terahertz frequencies (Zheng et al. [2013])) and total power consumption, the second wall. Given that these devices are limited in the simple aspects of form factor and heat dissipation, power consumption is typically limited to a few hundred watts (150W-300W), such as Intel's QX9775 quad-core Central Processing Unit (CPU)(Intel [2008]). Third is the memory wall which has formed because of the difference in

speed between the CPU (continually increasing in line with Moore's Law) and memory which doesn't follow the same trend for Moore's Law. Adding to the difficulty of increasing performance is that the hardware has been engineered to keep it busy at the instruction level by calculating multiple items simultaneously (via instruction level parallelism) as it was the lowest hanging fruit in terms of optimizations. This approach however affects algorithm performance when the hardware makes incorrect predictions, such as the case with incorrectly choosing a branch within the algorithm. Thus, these three walls force architecture designers to develop novel solutions to avoid algorithmic limitations when using traditional CPU-based architectures.

The first attempt at solving the three walls was the homogeneous multi-core CPU where each core is simplified and runs at a lower clock speed than a single core CPU. This method reduces heat output and power consumption at the expense of frequency and single threaded performance. The benefit of this approach is that multi-core systems require little to no code modification to enable existing software to run. However, this approach cannot be scaled indefinitely since, as the number of cores increases, so does the overall power consumption and heat dissapation. Diminshing returns are also encountered as Amdahl's Law states that the execution time of a program will be limited by the sequential portion of the algorithm (Amdahl [1967]). While homogeneous multi-cores have been promoted strongly by companies such as Intel (Lee et al. [2010]), heterogeneous multi-core architectures are moving at a very rapid pace into the general purpose computing market.

Heterogeneous multi-core architectures such as IBM's Cell Broadband Engine (Cell/B.E.), multi-core systems with Graphics Processing Units (GPUs) or any hard-

ware specialized accelerators have a higher performance/power ratio. Due to their heterogeneity, these architectures support diverse applications but programming these architectures is also very difficult. In implementing algorithms for heterogeneous systems, proprietary languages are often required, limiting portability. Now, OpenCL has become the new standard for ensuring cross-platform and device independent code and can be written using a "write once run anywhere" methodology. Although the GPU has shown great promise in accelerating the performance of data parallel applications, it is still limited by the memory wall as data must explicitly be transferred between the host CPU and the discrete GPU. This data transfer can greatly affect application performance for irregular programs where memory access is inconsistent or the number of work items may change during runtime.

Now, one such heterogenous multi-core architecture attempts to solve this problem, AMD's Fusion series of Accelerated Processing Units (APUs). The APU is a radical change from the traditional systems of a few years ago; instead of a separate CPU and GPU, they are combined onto a single die, reducing cost while increasing performance. This design change enables consumer systems to have a capable CPU connected to a powerful and compute-capable GPU using a Very Long Instruction Word (VLIW) architecture (and Graphics Core Next architecture in the future). The VLIW architecture provides a great amount of computational power in a small space, enabling high performance work to be done that, only a few years ago, would have been possible on a high end workstation.

1.1 Thesis Goal and Contributions

When I started this research three years ago, the APU architecture (Brazos and later, Llano) was still in its infancy. Now, AMD's next generation APU Kaveri has been released and their Carrizo APU is planned for release in 2015. A lot has changed in this progression: memory is no longer a subset of main memory, the CPU and GPU memory address space are now shared and coherency has been maintained, while in the future, AMD plans to implement GPU context switching for multitasking, quality of service, and other features.

In this work, I focus on the Llano APU's VLIW architecture. As well, a new standard parallel programming API for heterogeneous multicores has been proposed (OpenCL). The goal of my thesis is to study the APU architecture in depth. To do so, I consider four problems: 0-1 knapsack, Gaussian Elimination, Monte Carlo, and Binomial Lattice. Each of these problems exhibit different characteristics from the parallel computing perspective and are noted here. The 0-1 knapsack and Gaussian Elimination problems are compute bound, memory bound, and incorporate synchronization and communication latencies. The Binomial Lattice and Monte Carlo algorithms are two techniques in solving the option pricing problem in finance (Spiers and Wallez [2010]) and are compute bound. With the exception of the Monte Carlo problem, all three problems exhibit irregular workloads at runtime.

The 0-1 knapsack problem is a well-known problem in combinatorial optimization. Although there are many techniques to solve this problem, we use the basic dynamic programming algorithm. The dynamic programming algorithm divides the problem into smaller subproblems and each of these subproblems share subsubproblems. The

subdivisions are independent, but the sharing of solutions between subdivisions introduces some synchronicity. The algorithm uses a table to store the results. Since every subsubproblem is only solved once, the amount of workload may decrease during the course of the algorithm which creates further load imbalance. In our implementation, we take advantage of the on-chip memory to allocate data for sharing between the threads to increase performance. As the problem is memory bound, I investigate the use of global memory versus local memory, loop unrolling, scalability (in terms of APU compute units) and caching.

The Gaussian Elimination problem is a common technique used in linear algebra. It solves Ax=y where A is a matrix, and x and y are vectors. Though a fast technique by hand, there are problematic issues that occur when parallelizing the algorithm. First is that the matrix A could be sparse which introduces redundant computations and unnecessary usage of memory. Second is the pivot selection process at each iteration, leading to synchronization latencies. Last is the need for data sharing among the rows of the matrix which requires efficient use of memory.

The Binomial Lattice algorithm forms a tree structure with lots of concurrency within a level, but requires synchronization between levels. The larger the depth of the tree (time steps), the more accurate the results. This leads to a very fine-grained, computationally intensive problem, where each node in the tree can be thought of as a thread. The computation per node is minimal but there are many time steps and nodes in a tree depending on the level of accuracy required. Due to the regularity imposed in this problem, vectorization is a likely possibility. However, due to the irregular load at different iterations, the memory accesses are scattered which makes

optimization difficult.

The Monte Carlo simulation is a very embarassingly parallel problem. It is extremely compute intensive and exhibits regular workload. It provides a lot of parallelism which requires lots of registers for concurrent operations. I wanted to study this problem to see the net effect of register pressure on the GPU cores.

I provide different implementations for each of the problems in OpenCL on the APU, each one improving upon the previous variant. I use AMD's OpenCL profiler to analyze the execution of the various implementations and study performance metrics such as kernel occupancy, ALUFetchRatio, ALUBusy Percentage and ALUPacking. I make use of the VLIW static scheduling available in Llano through loop unrolling and vectorization and measure its effect on performance due to register pressure. I also study the effect of memory access versus local memory and caching.

The work from this thesis has resulted in a book chapter and three conference publications:

1. Matthew Doerksen, Steven Solomon, Parimala Thulasiraman, Designing APU Oriented Scientific Computing Applications in OpenCL, International Symposium on Advances of High Performance Computing and Networking in Conjunction with HPCC 2011, Banff, Alberta, Canada, Sept. 2-4, 2011.

2. Matthew Doerksen, Parimala Thulasiraman and Ruppa K. Thulasiram, Optimizing Option Pricing Algorithms and Profiling Power Consumption on VLIW APU Architecture, The 10th International Symposium on Parallel and Distributed Processing with Applications, Madrid, Spain, July 10-13, 2012.

3. Matthew Doerksen, Steven Solomon, Parimala Thulasiraman and Ruppa K.

Thulasiram, Financial Option pricing on APU, The 5th International Conference on Contemporary Computing (IC3), Noida, India, August 2012.

4. Matthew Doerksen, Parimala Thulasiraman and Ruppa K. Thulasiram, "Algorithm and Performance Analysis of Scientific Applications using OpenCL on Fusion APUs", Scalable computing and communications: theory and practice, John Wiley & Sons, Editors: Samee U. Khan, Lizhe Wang, and Albert Y. Zomaya, January 2013.

Chapter 2

Parallel Computing and the APU

In this chapter we detail the systems that lead up to the APU, such as homogeneous CPU-based systems, the heterogeneous systems that followed to overcome the limitations of homogeneous setups and how the APU slides into the final equation.

2.1 Parallel Systems

2.1.1 Homogeneous Systems

Traditionally, parallel systems can be broken down into two categories, homogeneous and heterogeneous. Homogeneous parallel systems use computers with CPUs of the same architecture and are connected via some form of interconnection network. This type of system is very simple to build and scale: just add more machines. When dealing with simple, regular algorithms this approach works extremely well as the machines have very little need to communicate and can just work at their own speed, combining results at the end.

Figure 2.1: The Athlon X2, the world's first consumer level, and AMD's most successful, multi-core CPU, which was first shown in August of 2004 (AMD [2004]). Picture taken from CPU-World.com (Shvets [unknownb]).

Maximizing performance however can be a challenge for irregular problems since these systems often depend on "slow" interconnection networks (IN), such as Gigabit ethernet (high latency, low bandwidth), when compared to custom INs such as Infiniband which has a theoretical throughput of over 300Gb/s (Association [2013]). In these cases, it can often be that the communication time is greater than the time spent computing results, leading to results that take longer than computing them on a single machine. To deal with these sorts of problems and overcome the hardware constraints, heterogeneous systems were created.

2.1.2 Heterogeneous Systems

A heterogeneous system foregoes the traditional PC architecture to overcome one of the 3 walls seen earlier (frequency, power and memory) to achieve better performance (in terms of flops/watt, bandwidth or some other relevant benchmark) for a given type of problem. One of the very first examples of a heterogeneous architecture

were the floating point co-processors of the 1980s that could be paired with CPUs to give them floating point capabilities (Shvets [unknowna]).

Figure 2.2: Intel's 8087 floating point coprocessor which could be paired with the CPU to improve compute capabilities on floating point arithmetic. Image taken from (Shvets [unknowna]).

Eventually, these coprocessors would be integrated into the CPU, much like how we see other hardware such as the northbridge becoming part of CPUs today like AMD's Llano APUs and Intel's Sandy Bridge CPUs. A much more recent example of a heterogeneous architecture is IBM's Cell BE which was designed as a Single Instruction Multiple Data (SIMD) architecture. This design has a large master core (Power Processing Unit or PPU) to handle heavy tasks such as maintaining the operating system, while delegating computational work to multiple smaller, simpler cores (Synergistic Processing Units or SPUs) which run at higher frequency.

This design allows them to customize the hardware so each portion can excel at what it does best; the PPU has more branching hardware and is better at handling context switching between applications while the SPUs are much smaller and have more execution units for higher computational throughput (Gflops). With this, the Cell BE achieves better performance (230 Gflops) compared to a high end CPU such as Intel's i7-980x (109 Gflops (Williams [2010])).

The Cell BE also improves on the memory architecture, using a 4-ring bus (2 running in each direction) termed the Element Interconnect Bus (EIB) to connect the

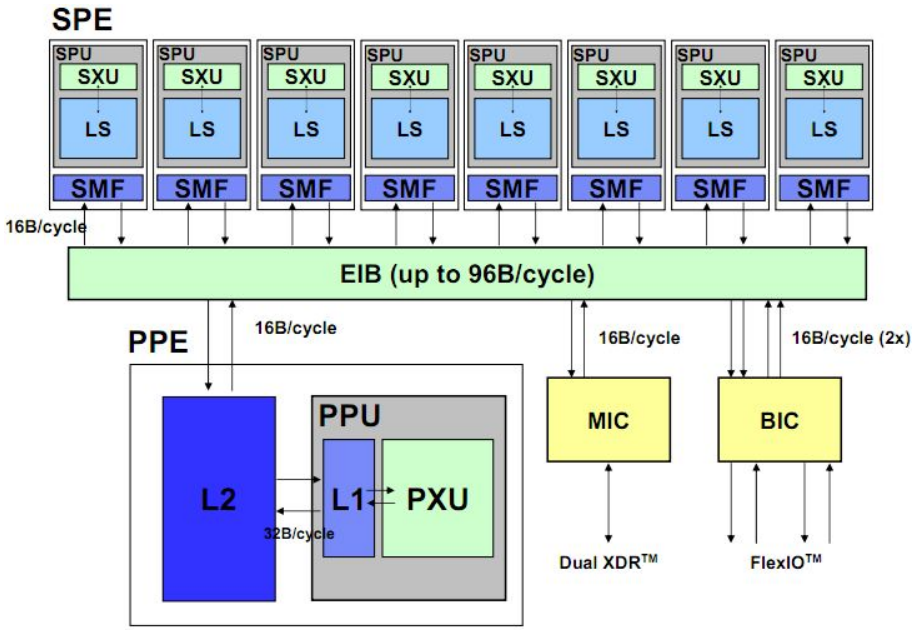


Figure 2.3: The IBM Cell Broadband Engine, a PowerPC based CPU which features a Power Processing Unit for OS tasks and Synergistic Processing Units for computation connected via the high bandwidth Element Interconnect Bus (Gschwind [2006]).

computing cores. This bi-directional bus has an internal bandwidth of 96 bytes/cycle/port adding up to 205 GB/s of sustainable bandwidth and over 300 GB/s of maximum peak bandwidth between ports. This is roughly 5 times the bandwidth than Intel's i7-4960X can provide with system memory, at only 60 GB/s (Intel [2013]). This advanced memory architecture allows the Cell BE to have much higher computational and memory throughput compared to a traditional system.

2.1.3 Homogeneous with a hint of heterogeneous

Yet another approach is the GPU which uses hundreds or thousands of cores (in a homogeneous setup) for computation. At the same time, it falls into the category of a heterogeneous device because it is, at least currently, a non-standard computing

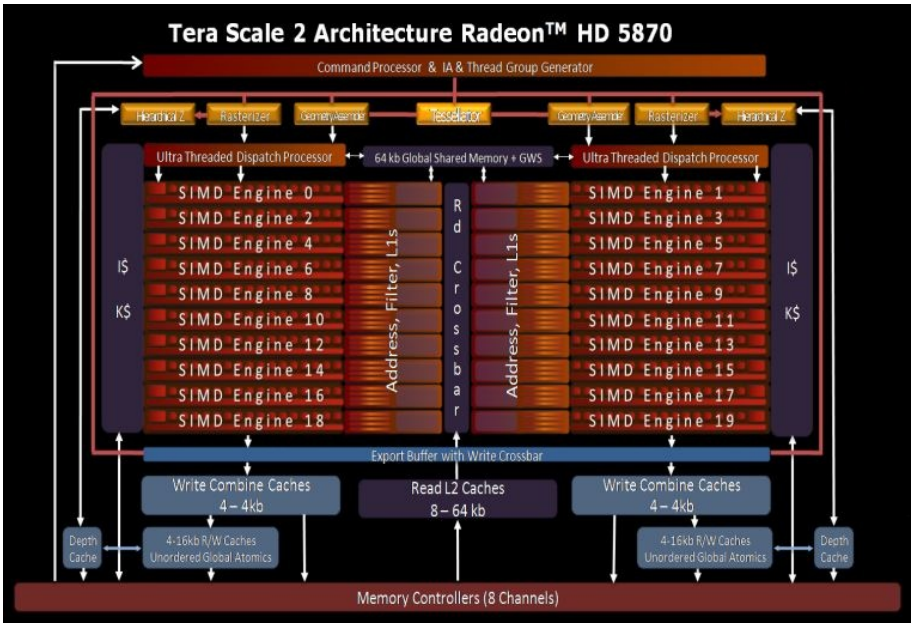


Figure 2.4: An overview of the Evergreen architecture used in AMD's Radeon 5870 GPU which contains 20 SIMD Engines (AMD [2010]).

device (compared to a CPU) and requires a different approach to programming and algorithm design. Traditionally, GPUs have been built to manipulate things like vertexes and pixels which are graphical operations that require minimal communication and can be done in a massively parallel manner. As such, GPUs have typically exhibited a SIMD-style architecture with large numbers of small, slow(er) cores which work in parallel to achieve optimal throughput. While this type of architecture is fantastic for computational throughput, it performs very poorly with tasks that are sequential, branchy, or randomly access memory. The APU attempts to resolve these issues by combining the sequential speed of the CPU with the parallel abilities of a GPU, the details and implementation of which are covered later in the APU Architecture section.

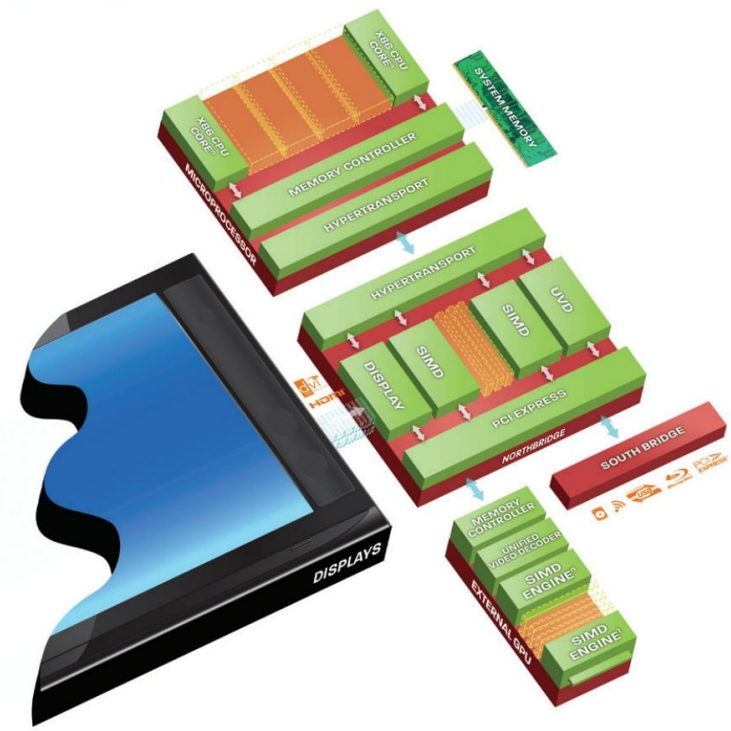


Figure 2.5: A typical computer, with a CPU and GPU which could be used for parallel computing (Brookwood [2010]).

Lastly, we have the APU which attempts to bridge the gap between both homogeneous systems and heterogeneous, bringing together the best of both worlds; homogeneous CPU cores for sequential tasks and GPU cores for the parallel components of an algorithm. Looking at Figure 2.5 we have a traditional system with a separate CPU, and if the system is built for compute or graphics works, a high-end GPU. Examining the image, the CPU has many layers which separate it from the GPU including the CPU memory controller, PCI Express bus and GPU memory controller. Each of these adds unnecessary transformation of data between protocols, reducing bandwidth and increasing latency; worst of which is the PCI Express bus

which limits communication between the CPU and GPU to just 16 GB/s in each direction (PCI-SIG [2010]). Compared to even system memory in all but the lowest end machines we can see this isn't fast enough, and even the PCI Express 4.0 specification to only be released late in 2015 will only double this to 32 GB/s (PCI-SIG [2014]). Knowing this, AMD decided to remove these performance hindering pieces of hardware, putting the CPU and GPU directly next to each other on the chip, the Accelerated Processing Unit.

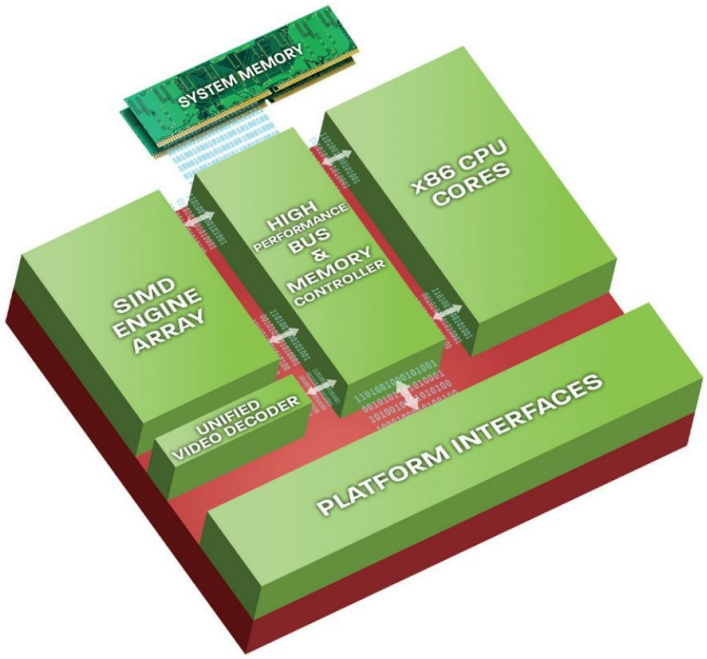


Figure 2.6: An APU system which moves the GPU onto the same chip as the CPU (Brookwood [2010]).

Comparing the two images (Figures 2.5 and 2.6) we see that the PCI Express bus has been removed from the equation, being replaced by a high performance on-chip bus. This has a few benefits; first, the performance regained by simply not translating information between protocols and not requiring it to leave the chip for information

to be communicated between the two, which saves time (latency) and bandwidth (the custom on-chip bus provides higher bandwidth compared to PCI Express). Second, the GPU now has faster access to system memory which is key as APUs do not currently have dedicated memory like a traditional GPU does. This has its own drawbacks however in that system memory is much slower than a high-end GPU's memory, but this can be mitigated in hardware by using the caching system already present and used by main memory. Last is the power savings found by not having to transform data between protocols and off the chip over the PCI Express bus. From this, the APU provides some key advantages over "traditional" heterogeneous systems using building blocks that are already available.

Chapter 3

APU architecture

In this chapter we examine the history of the APU, the implementations of AMD and Intel including strengths and weaknesses that resulted from design decisions and a comprehensive overview of AMD's APU architecture.

3.1 History of the APU

Similar to how heterogeneous computing wasn't a new idea, the idea of a fused CPU + GPU was not an entirely new concept (embedded systems could technically be labeled as APUs since they combine multiple systems using a System-on-a-Chip architecture). AMD first proposed the idea of a large, powerful, fused processor in 2006 when they acquired graphics manufacturer ATI (AMD [2006]). This was an attempt to bolster their intellectual property portfolio and to help them become more competitive with both Intel and Nvidia, their main rivals in the CPU and GPU fields respectively. AMD saw an opportunity for this new hardware in a world that was

becoming more media centric, with richer web applications and 3D graphics support and wanted to be the first to break the ice.

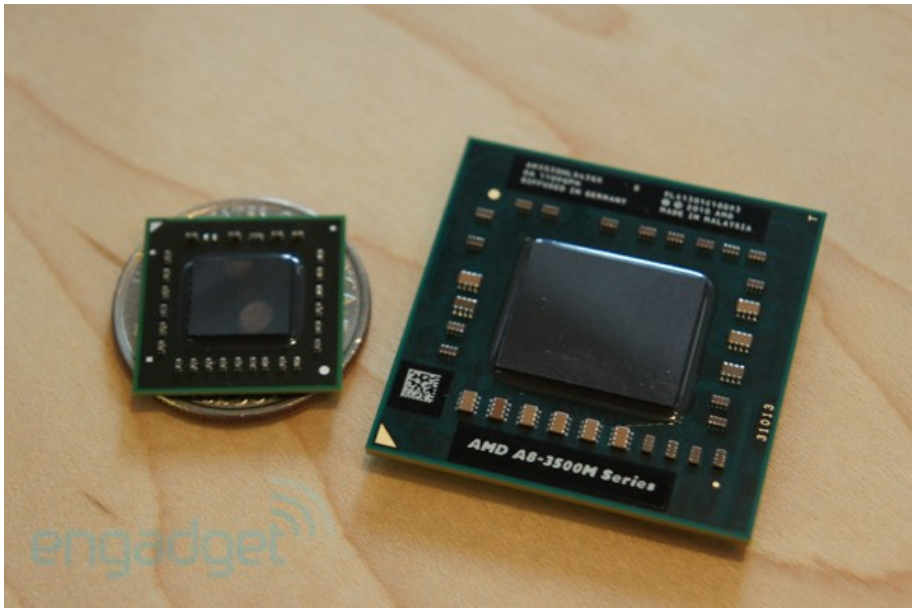


Figure 3.1: A chip from AMD's A8 series of APUs which integrated the GPU directly into the core of the CPU (Wollmann [2011]).

3.2 The Competition

While AMD was the first to propose the idea of an APU, they weren't the only one working on it. In fact, Intel was working on the same concept, which would incorporate their HD graphics onto the CPU. Due to delays with the manufacturing process and difficulties in porting the GPU onto CPU optimized silicon, AMD's APUs were pushed back until 2011 (Lal Shimpi [2009a]) while Intel was able to release their APUs in 2010 (Lal Shimpi [2009b]), making them the first publicly available. The approach Intel took however was different from AMD's; instead of a fused CPU +

GPU, they went with on-chip graphics. By using a 32nm CPU and a 45nm GPU they achieved a better time to market and encountered fewer frustrations in getting the GPU to work on the same process and manufacturing technology (compared to AMD), see Figures 3.1 and 3.2 to see how the resulting chips differ in appearance.

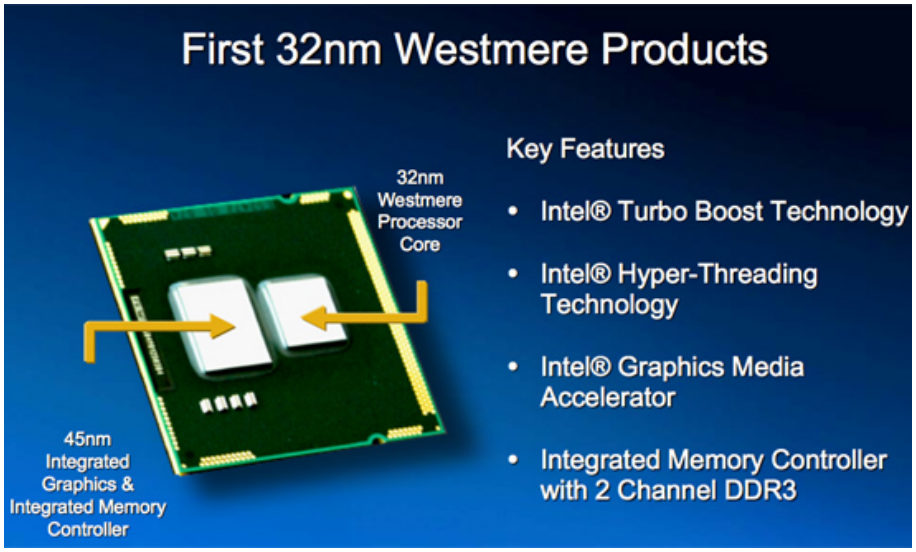


Figure 3.2: Intel's Westermere CPU, the first publicly available APU (Lal Shimpi [2009b]).

On the other end of the spectrum, we have Nvidia which is working to produce a GPU with a CPU built onto it (as opposed to Intel and AMD's approach of integrating the GPU onto the CPU). This processor, codenamed Denver, will use ARM CPU cores (and eventually their own custom ARM core) which provide very high performance per watt and will be paired with a single SMX containing 192 cores, based on the Kepler GPU architecture (Verduzco [2014]). One version, shown in Figure 3.3, details the four ARM CPU cores paired with a fifth "battery-saver" core to give the best balance of performance and battery life, and matched with a cutdown version of their Kepler GPU. This combination of disparate parts enables Nvidia to produce a very

high performance APU, but produce it at a low cost and have it run consuming minimal power. As well, as you'll see in future sections, enabling these modular designs across CPU and GPU combinations is the exact goal of the Heterogeneous Systems Architecture.

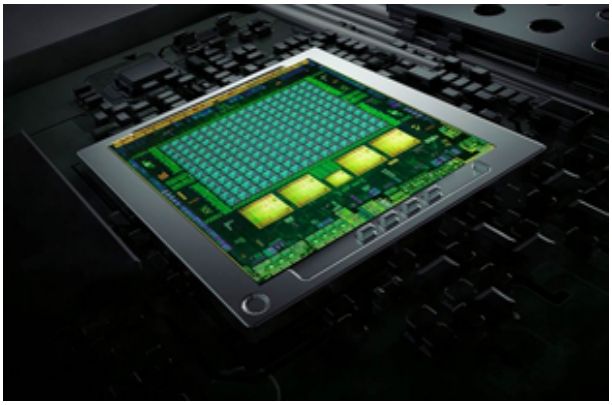


Figure 3.3: Nvidia's Tegra K1 APU which uses ARM CPU cores and their own Kepler GPU for high performance per watt at a low cost point (Smith [2014]).

3.3 APU Design Tradeoffs

Why the AMD and Intel produced different results was due to the strengths that each posessed. AMD envisioned a die with a great emphasis on GPU computing capabilities which could be used for general purpose computation as the world became more media centric. Intel on the other hand, focused heavily on CPU performance with the GPU being a second class citizen, used primarily for 2D graphics, instead of media consumption, which they believed could be offloaded to the CPU. This was reflected in each company's allocation of hardware resources, where AMD dedicated much more resources to the GPU than Intel, as shown in Figure 3.4. From Intel's

reaction, we see that AMD had predicted correctly, media consumption was becoming a more important factor for consumers and Intel was forced to increase the die space dedicated to the GPU to keep up with AMD's APUs and consumer's demands. We'll cover more of these design decisions in the next section where we take an in-depth look at AMD's A10-5800K APU.

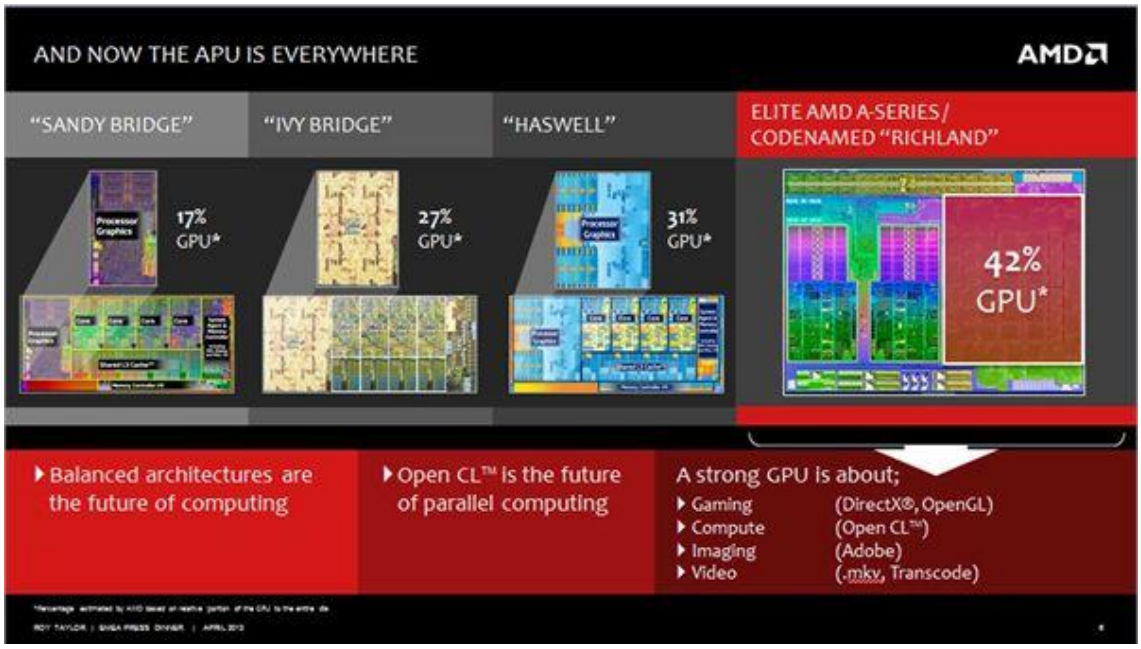


Figure 3.4: The percentage of each APU die reserved for the GPU. Intel's APUs are shown on the left while AMD's Richland APU appears on the right (Magee [2013]).

3.4 AMD's APU Architecture

As an AMD A10-5800K APU was used to gather the results which are covered in Chapters 5 through 8, we cover its implementation here, how the pieces come together to create a powerful heterogeneous computing device, and what AMD is working towards.

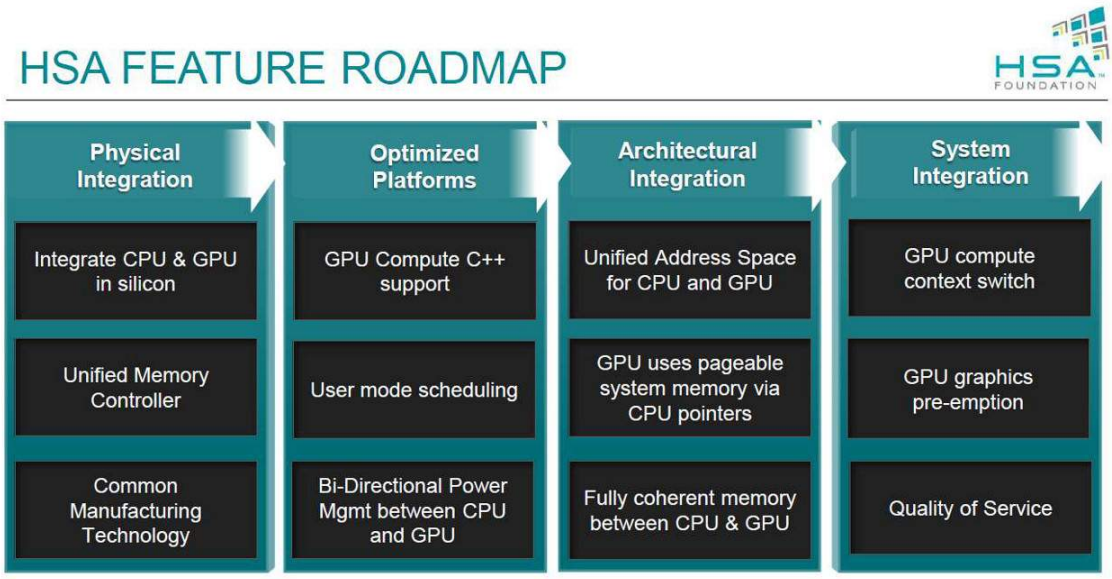


Figure 3.5: AMD's roadmap for the Heterogeneous Systems Architecture which moves from combining the CPU and GPU at the die level, to transparently map memory between devices and finally, autonomously full use of the GPU for computing (Clarke [2013]).

3.4.1 The CPU

AMD's first step in their Heterogeneous Systems Architecture (Figure 3.5) roadmap was to combine the CPU and GPU into the same chip. That goal was accomplished with the release of the Llano architecture in 2011. The A10-5800K is the result of the second phase, platform optimization, and is a chip that contains 4 CPU cores and 384 VLIW4 GPU shaders clocked at 3.8 GHz and 800 MHz respectively which runs at a 100W TDP (see Figure 3.7). The CPU cores used belong to the Piledriver family (the next iteration of Bulldozer) and come in groups of two that are collectively called a module. These modules have a lower instructions per clock throughput than current Intel CPU cores and instead use high clock rates to boost performance. The modular

design, based in cluster multithreading (see Figure 3.6), combines two cores to share some hardware, saving die area, lowers power consumption and increases utilization. This is a different approach to simultaneous multithreading (termed HyperThreading) that Intel employs in its CPUs, which doubles only the frontend hardware (as well as internal registers) to try and get better utilization of hardware resources. In practice, cluster multithreading works best at highly threaded workloads which are heavier on integer operations while simultaneous multithreading is better when there is a larger number of varying instructions which don't cause resource contention.

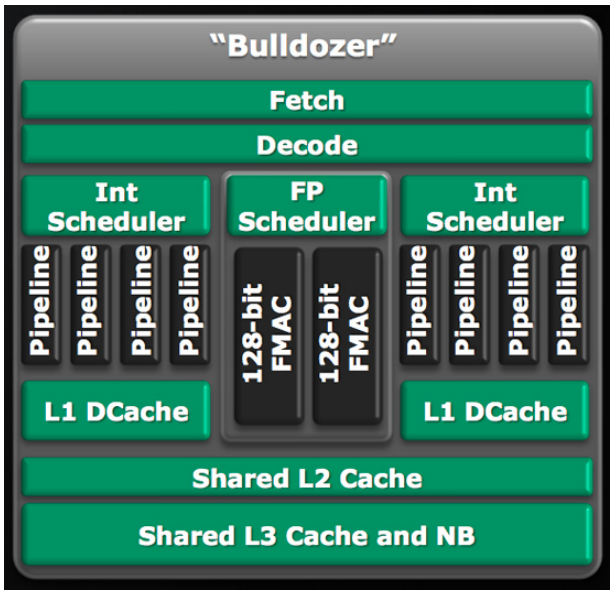


Figure 3.6: A module used in the "Bulldozer" architecture which has separate integer pipelines but a shared floating point pipeline which enables a "full" second core for only 50% more die area (Lal Shimpi [2009c]).

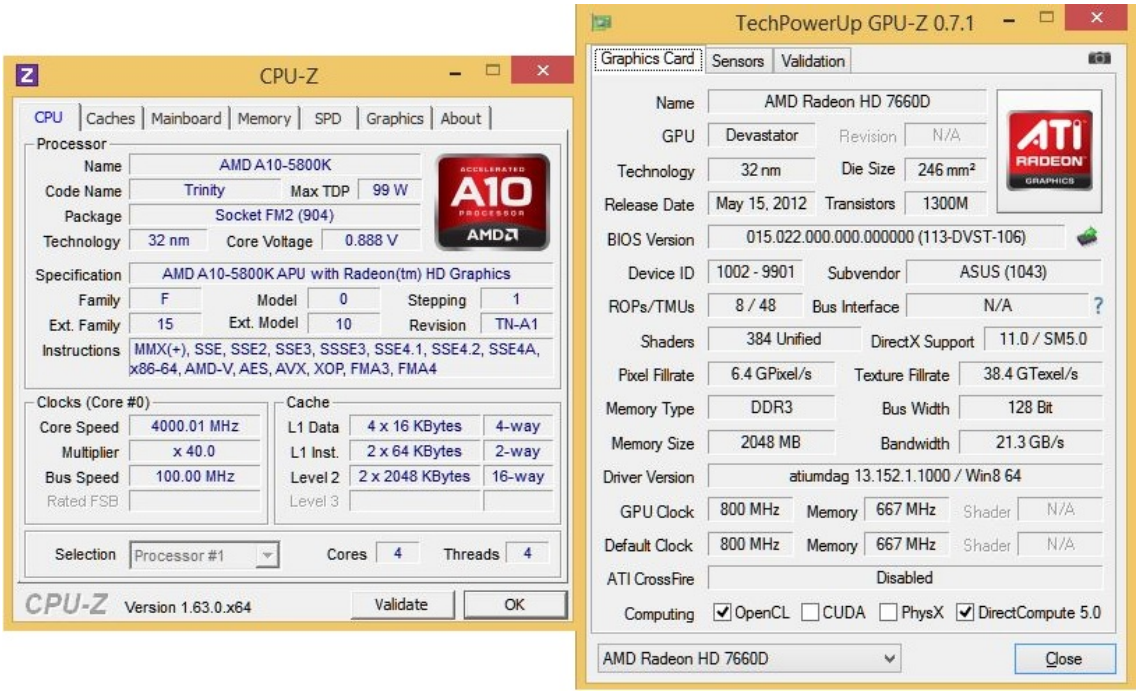


Figure 3.7: Two screen captures of the A10-5800K APU showing the 4 cores backed by 384 VLIW4 shaders.

3.4.2 The GPU

The AMD Radeon 7660D is a $246mm^2$ GPU which has 384 shaders and runs at up to 800 MHz and has up to 2048 MB of shared memory which runs at the speed of main memory. This particular GPU is based on the Northern Islands architecture which moved from VLIW5 to VLIW4 to obtain better utilization of the hardware. This was done by removing one of the execution units as AMD found it was unused in the majority of cases, leading to unused hardware and increased power usage. In turn, they were able to use those extra transistors to create more SIMD Engines, similar to Figure 2.4, and increase overall computing power. At this time however, a newer architecture exists, Graphics Core Next (GCN), which moves the scheduling

to hardware which could better determine dependencies at runtime as opposed to during compilation, and as such, schedule instructions on the hardware for better utilization. AMD's third generation APU, codenamed Kaveri which was released in January 2014, includes CPU cores based on the Steamroller architecture and GPU cores rooted in the GCN-based Volcanic Islands architecture which was released in Q4 of 2013. These changes provide a much needed boost on the CPU side and further AMD's advantage over rivals Intel and Nvidia on the GPU side (in the low-mid range market targeted at APUs).

3.4.3 The Magic

Combining a CPU and GPU is not all that difficult; take one part CPU, one part GPU, one part silicon and mix them together. This basic approach however won't yield very good performance as both the CPU and GPU will be performance limited since they now have to share the same die area that a single chip previously did. The real magic of the APU lies in the hard-to-measure category of the uncore, the glue that combines the two in order to maximize performance.

To enable GPU communication with memory, Llano has a dedicated interface, the Radeon Memory Bus, which is 256 bits wide in each direction and replicated for each memory channel (Kanter [2011]). For memory accesses to coherent system memory it has the Fusion Control Link which is 128 bits wide in each direction. The significance of these are to simplify memory accesses from the GPU to CPU memory and vice versa since the two are built for different use cases; the CPU for latency and the GPU for throughput. Based on this, we can see that the limiting factor is going to be the

throughput of main memory for the GPU (the CPU already uses caching).

To aid in the pain point of bandwidth, AMD has two technologies, Pin In Place and Zero Copy (Demerjian [2011]). Pin In Place enables placing a chunk of memory in a static location which prevents having to search for memory locations at runtime, saving time, power and resources. More important is Zero Copy, which enables memory sharing between devices by simply passing a pointer from one to the other instead of copying data between devices. These technologies, enabled by the on-chip buses, allow programmers to finally extract maximum performance, removing hardware limitations, saving massive amounts of bandwidth and lowering latency dramatically.

Regardless of these improvements, AMD's APU still relies heavily on main memory speed, which is why they achieve roughly a 20% performance boost over DDR3-1333 memory when using DDR3-1866 (Lal Shimpi [2011a]), an increase of approximately 40%. Faster RAM is not the end all due primarily to the cost associated with faster memory, which is an issue in a budget oriented market. That is why companies like Intel are working in tandum with Micron to develop memory which would be placed directly onto the chip, and using a very wide bus for extremely high memory bandwidth, up to 1 Tb/s (Lal Shimpi [2011b]).

3.4.4 Implementation comparison of APUs, Intel vs. AMD

While we've already detailed the differences in how AMD and Intel design their APUs (in terms of transistor allocation), we haven't yet covered some of the lower level details regarding the implementation. One difference is that Intel has gone with an L3 cache that is shared between both the CPU and GPU (Lal Shimpi [2010]).

This enables higher performance for both the CPU and GPU since data can be easily shared, and, since it is a cache, it can prefetch new data (unlike a traditional GPU), so it becomes available before it is required. This gives it very good performance when either the CPU or GPU is being stressed as shown by benchmarks run by the website Anandtech (Lal Shimpi [2010]). However, when there is a blend of CPU and GPU use (i.e., there is no heavy imbalance in the processing power used by each component; it's split approximately 50/50 or 60/40) such as the Dawn of War II and HAWX games, the solution loses to a low-end dedicated GPU since the L3 cache cannot be fully utilized due to conflicts between devices and because of the second design difference, how the hardware is provisioned.

Intel went with the approach of scaling the CPU and GPU independently within the TDP limit and a priority to the CPU and AMD set minimum TDP limits for both the CPU and GPU with a priority set more towards the GPU (Demerjian [2012]). This gives Intel the benefit of having excellent performance when one or the other is used (most traditional benchmarks) but has lower average performance since neither the CPU or GPU can maximize their potential (they now have to share the maximum TDP instead of having it allocated to a single one) (Lal Shimpi [2009d]). AMD on the other hand took the other approach and guarantees both the CPU and GPU run at a minimum level, even when stressed which gives them better mixed performance at the cost of single benchmark maximum performance. Each implementation has its own advantages and disadvantages but as APUs have progressed, the methods of maximizing performance have improved and APUs from different companies seem to look more and more alike with every iteration.

3.5 Future APUs

We've covered how APUs from competitors bear more and more resemblance with each iteration. Starting in 2011 with the Llano architecture, AMD put forth the first of many pieces towards their heterogeneous systems architecture, integrating the CPU and GPU onto the same silicon. They have since moved to phase 2, optimizing the components and enabling better resource allocation and maximum performance from each subsystem (using the Trinity architecture). Now, in 2014 they will complete stage 3 with the release of Kaveri, architectural integration between the systems, enabling simple, clean communication between the CPU and GPU and removing the bottlenecks of memory passing. This leads into the last phase, system integration which I will cover later in Chapter 9 where I examine future APUs and technologies currently being developed and what they mean for the computing industry.

Chapter 4

GPU Computing: Shaders, CUDA and OpenCL

GPU computing isn't exactly a brand new idea; it has been around since GPUs had vertex and pixel shaders which were used to manipulate objects. The problem with this approach was that there was no standardized application programming interface (API) used to specify data and how it should be accessed. Instead, programmers provided vertices and vectors with which the GPU performed various calculations and when finished, the programmer would convert the data back into a usable form such as the work done by Fung and Mann [2005]. This roundabout method required inefficient programming methods and was not programmer friendly, meaning it was very error prone and not easily scalable to very large projects. Nvidia later saw the potential of the GPU for more than just graphics and in 2006 would release the Compute Unified Device Architecture (CUDA) which would enable general purpose computations to be performed on a standard GPU (Nvidia [2006]). Similarly, ATI

(acquired by AMD in 2006) was at the same time working on their own project, Close to Metal, which has since been discontinued in favor of OpenCL.

CUDA started out as a framework to reduce the effort required to use the GPU for general purpose computation. Nvidia started by using a C-based language and added extensions on top of it in both the GPU driver and programming SDK to handle the tough work like setting up the device, transferring data and computing the results. With such a simple model and general availability (it could be used on any GPU starting with the GeForce 8800 series, also released in 2006), CUDA positioned itself to allow GPU computing for the masses. CUDA however would not meet this end goal, being far too limiting with regards to the hardware it could be run on (solely Nvidia GPUs), never gaining the traction its creators had envisioned; it did however pave the way for a more open computing language.

4.1 OpenCL, the Open Computing Language

OpenCL, originally developed by Apple and now owned by the Khronos group is an open, cross-platform API enabling programming for heterogeneous devices. This abstraction of hardware gives OpenCL a write once, run anywhere model and means it can better handle future changes in the computing landscape, such as the inclusion of OpenCL in ARM CPUs, as well as Application Specific Integrated Circuits (ASICs) built in hardware to perform a single type of calculation, such as bitcoin mining. With OpenCL's vast reach, it can be used everywhere, giving it many advantages over its competitors as listed in Table 4.1.

As we can see from the chart, OpenCL has the benefit of being fully open regarding

	OpenCL	CUDA	DirectCompute
Operating System Agnostic	Y	Y	N
Hardware Agnostic	Y	N[1]	N[2]
Open standards API	Y	N	N
Future-proof (APU ready)	Y	N[3]	N
Gaining support (e.g., ARM)	Y[4]	N	N
Easily integrated with graphics	Y	N	Y

[1]Nvidia devices only,
[2]DirectX 10+ GPUs only
[3]No APU support, only compatible with Nvidia's ARM products
[4]Multiple vendors are adding OpenCL support to their devices (Steele [2011]; Lokhmotov [2010])

Table 4.1: OpenCL vs CUDA, hardware and software support.

hardware and software, making it the perfect API for developing high performance computing applications as hardware changes rapidly. Moving forward we look to the details of OpenCL from a programmer's perspective, including the threading model, memory model and data partitioning.

4.1.1 OpenCL Threading Model

The OpenCL threading model is quite simple. Each element to be computed is a work item. These work items are then blocked into local workgroups (whose size may vary between devices and is used to target optimal device use) which then execute on the device (and combined make up the global workgroup). For example, say that in Figure 4.1, the NDRange size, or global workgroup is 768x768. This would represent a total of 589,824 work items which will be computed by the device. Looking at the global workgroup, we see it's broken down into 9 chunks that are deemed local workgroups. Each of these 9 local workgroups would then contain

256x256 (65,536) work items, though the number can and will vary based on hardware support. The hardware then independently schedules the local workgroups to best utilize the hardware; an important factor in maximizing algorithm performance due to hardware constraints (such as running out of hardware registers by choosing too large a local workgroup size).

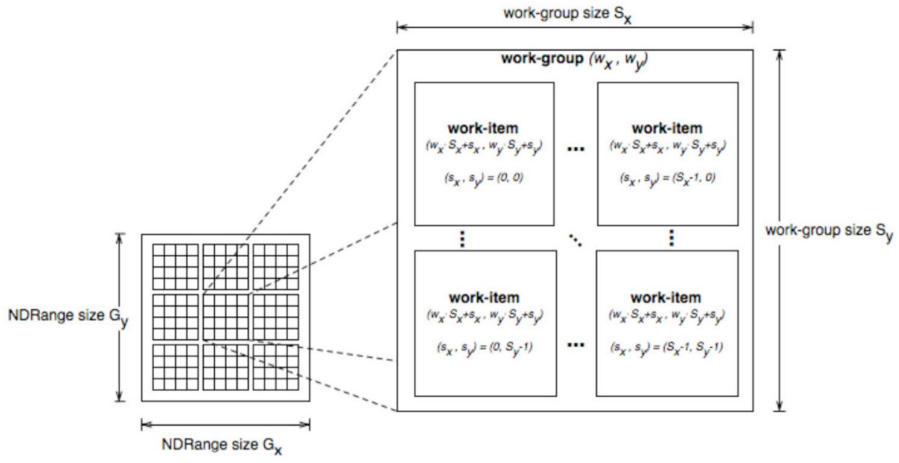


Figure 4.1: All elements to be computed are members of the global workgroup (shown on the left). On the right is a single local workgroup which is a subset of the global workgroup and contains many work items that share a common hardware resource (compute block) (Gaster and Howes [2010]).

4.1.2 OpenCL Memory Model

OpenCL's memory model is very similar to that of CUDA, and, at a high level, that of a regular compute device with multi-level caches to speed up operations (though here we have some control over how and where our data is moved and located). Looking at Figure 4.2 we see 4 types of memory: private, local, global/constant and host. Private memory is the lowest level of memory available, registers which

provide the fastest access to memory and are allocated to each thread (i.e., each work item) in the same manner as memory is allocated to a CPU. As registers are an entirely hardware managed memory, programmers cannot directly access it, but instead declare variables within the OpenCL kernel that are placed into registers via the compiler (if the requested resources are available at the time).

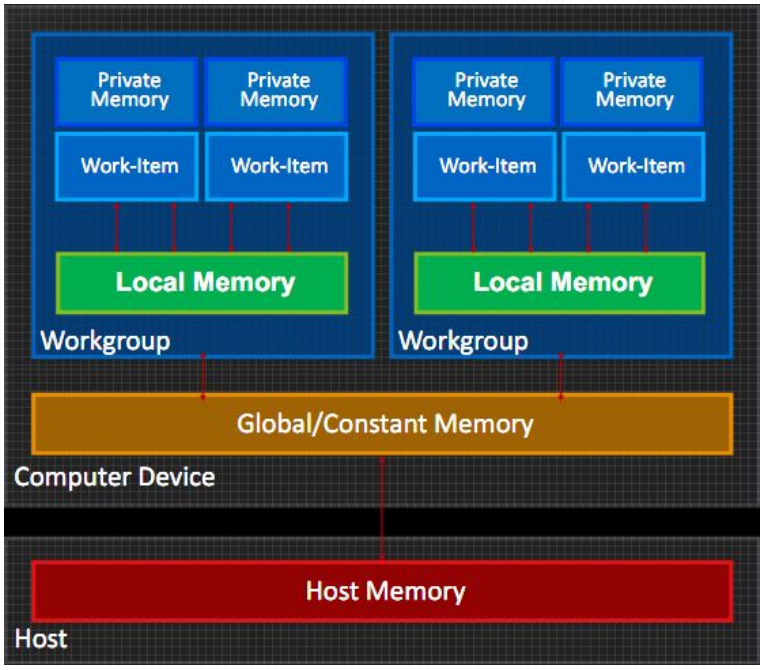


Figure 4.2: The OpenCL memory model involves four different types of memory provided by the GPU. These include the global memory, constant memory, local memory and private memory. Each of these has a particular purpose which will be explained in the GPU Memory Types section (Gaster and Howes [2010]).

Local Memory

Local memory is one step above registers and is typically around the size of a CPU's L1 cache. This piece of memory is included in each compute block on the device, meaning we actually have a few MegaBytes worth to use for the entire device.

Local memory is both read and writeable from all threads in a local workgroup (which in turn is assigned to a single compute block). The setup of local memory in hardware makes it so that as programmers, we don't have to perform manual coalescing of memory reads/writes (ordered by memory address; without which, memory accesses would be performed sequentially) to get good performance. We also have full control over this portion of memory so we can use it in any way we like and keep data around as long as we like (as opposed to CPUs which typically use a least recently used setup to remove old lines from the cache). However, local memory does have the limitation of not being able to communicate directly with other local workgroups (like the cache sharing model employed in multi-core CPUs). Instead, we as programmers must explicitly move information from the one workgroup's local memory into global memory at which point we can then read it into the other local workgroup's local memory.

Why not just one memory?

Now you might be asking yourself, why not just add more hardware to replace global memory with local memory since it's so much better? The answer to that lies in hardware limitations. Replacing global memory with local memory would require replacing the separate memory chips currently on a GPU and integrating those transistors into the chip itself. This would in turn increase the power use of the chip (high end GPUs already consume 300 watts, as defined by their thermal design power) and would require more die area for the chip itself, and GPUs are already bumping into the upper limits of manufacturing capability due to silicon

wafer defects. Due to this limitation, just like CPUs, GPUs have added memory levels for specific use cases to try and reduce the amount of time the device must wait on input/output.

Memory Type	Size per Compute Unit	Total Size on GPU	Peak Bandwidth
Private (registers)	256 kB	8 MB	22.2 TB/sec
Local	64 kB	2 MB	3.7 TB/sec
Constant	-	128 kB	474 GB/sec
Global	-	3+ GB	264 GB/sec
PCI Express v3.0	-	-	16 GB/sec
DDR3-1600	-	-	12.8 GB/sec

Table 4.2: GPU Memory Types and Bandwidth for AMD Radeon 7970 (AMD [2012]).

Constant Memory

On AMD hardware, constant memory (image memory is similar but for textures) is a small portion of dedicated memory. It is smaller than the Local Data Store (LDS or local memory) at only 48kB for the entire GPU, but is meant to hold values that will not change during runtime, whereas the LDS is a scratch pad for information that will be passed around and updated constantly, requiring the ultra high bandwidth and low latency access LDS can provide. As it is implemented directly in hardware, it has very high bandwidth at almost 500 GB/s which is extremely slow compared to registers, but nearly twice as fast as global memory, the highest level.

Global Memory

Global memory is slow compared to all other memories (registers, local, constant/texture) but has the advantage of being very large (4+ GB is becoming standard

on the top models of current GPUs) and read/write accessible from all workgroups in a program. This makes it the best storage location for the GPU for large data that needs to be simulatenously worked on by all threads and which is written or read infrequently; this approach sacrifices latency and bandwidth for the benefit of being able to globally synchronize across all workgroups. As well, in traditional devices, such as the GPU, it's also the end point for communicating between the CPU and GPU where data must be explicitly copied from the CPU to the GPU and back. What we'll see later with the APU however, is this model being turned upside down, where no data needs to actually be transferred.

4.1.3 OpenCL Data Partitioning

Data partitioning with OpenCL is key to achieving good performance and there are two things we need to keep in mind. One is that all work items are calculated in parallel (pending hardware scheduling restrictions) and there is a balance between having each individual work item have the resources that it needs while simultaneously maximizing the utilization of the device. First, we need to remember that all work items are calculated in parallel with no guarantees regarding the order of completion (assuming no synchronization is required in the kernel). This leads to programmers being responsible for partitioning data correctly and at a level where we can synchronize/transfer data in the most efficient manner, either at the local workgroup level (most efficient; fewer threads) or at the global workgroup level (all threads). While synchronization hinders maximum performance, without it, results can not be guaranteed due to race conditions or improperly ordered memory accesses

(compared to a sequential version).

Maximizing Efficiency

Given this, we need to then look at how to best optimize our available resources, of which there are a few ways: workgroup sizes, locality and optimizing the usage of available hardware. The most simple method of data partitioning is to split the data across workgroups, which can vary in size (as supported by the device). There are tradeoffs with the local workgroup size, where smaller workgroups should provide good device utilization. However, at the same time, having too many small workgroups may cause the hardware to spend a lot of time context switching workgroups as they wait for, or complete work, hindering overall performance. Thus, we need to find the optimal size for the given hardware we have (e.g., Are we limited by register usage, the number of work groups, or something else?) which can be found via AMD's OpenCL Kernel Analyzer. Not unrelated is the hardware present such as the constant and image memories. While it may have a lower total bandwidth than LDS, using constant memory instead may free up our usage of LDS (if pushing against the 64kB upper limit per local workgroup) allowing us to add more data to it, increasing our data locality and making sure it is as readily accessible as possible.

4.1.4 Optimization Techniques

To obtain the best performance, we need to look at multiple techniques to optimize our algorithm for our hardware. These range from optimizing control between devices and eliminating memory transfers, to packing instructions to take advantage of the

APU's VLIW architecture.

Loop Unrolling

A relatively simple technique to reduce idle time is to keep more iterations on the APU before transferring control back to the CPU (for memory transfers, synchronization, sequential calculations, or other reasons). While not possible for every algorithm, when it is possible to be used, it can have a very dramatic effect on performance.

Memory Optimization

To optimize memory accesses we can use local memory (covered in more detail later) as a programmer-managed cache to avoid having to make calls to the slow global memory. A second option is to take advantage of the APU's shared memory space between the CPU and GPU by using host memory which transfers data between the two by simply moving a pointer. Using this approach, transferring data no longer involves copying memory and moving it off the device, making it incredibly efficient when memory is read or written infrequently.

Vectorization

The final optimization I examine for the APU is vectorization. With vectorization, we can pack instructions and data together (if there are no conflicts) which can be executed simultaneously on the hardware (provided it also supports parallel calculations by each execution core). This organization of data enables better scheduling of instructions to improve device utilization and optimize memory access.

Chapter 5

0-1 Knapsack

5.1 Problem Definition

The basic 0-1 knapsack problem is a staple of computer science. The goal of the problem is to choose a set of items, each with a given profit and weight, so as to maximize profit within a given weight restriction. This is known as an optimization problem, and more specifically, resource allocation and is applicable to many real world situations such as bin packing, route choice (similar to to the traveling salesman problem) and cutting an item to minimize waste (see Figure 5.1). This means large real world cost savings when a solution can be even just 1% better, which is why this problem has been studied extensively, as I'll cover later under related works.

Item	1	2	3	4	5
Profit	5	3	7	2	8
Weight	4	1	3	2	8

Capacity/Item	1	2	3	4	5	6	7	8	9	10
1	0	0	0	5	5	5	5	5	5	5
2	3	3	3	5	8	8	8	8	8	8
3	3	3	7	10	10	10	10	15	15	15
4	3	3	7	10	10	12	12	15	15	17
5	3	3	7	10	10	12	12	15	15	17

Figure 5.1: A short example of the 0-1 knapsack problem using the dynamic programming approach. Here we have 5 items to choose from, each with a given profit and weight, and a set bag capacity (up to 10). In this case, the best profit achievable is 17 using items 1, 2, 3, and 4.

5.2 Related Works

To solve the 0-1 knapsack problem there are a number of algorithms that can be used, each of which takes a different approach. These include greedy algorithms which simply choose the "best" items (be it in terms of profit or some other maximizer), branch and bound algorithms which select items based on some factor (e.g., profit/weight ratio), approximation algorithms, or dynamic programming which builds a 2D item/weight matrix filled with the best profit achievable using items x..y..z.. For my work, I examined the dynamic programming approach as it provides an optimal solution every time and, based on the matrix's structure, should map well to the APU architecture.

Examining related works, Boyer et al. [2012] created a solution to reduce the memory footprint of the dynamic programming 0-1 knapsack algorithm. The benefit of this is reducing the amount of data that needs to be passed to the GPU, which should in theory speed up the alogrithm. To do this, they removed continuous duplicated

values within a row of the matrix with a single value. This removes unnecessary data transfer between the CPU and GPU as well as eliminates the duplicated processing of these values since they will be the same after the iteration completes. The result of their optimization techniques is a matrix which took up less than one percent of the space of the original matrix and a speedup ranging from 19 times for a 10,000 item matrix to 26 times for a 90,000 item matrix. Quantifying these values, the CPU took nearly 59 seconds for 10,000 items while the GPU took 3.06 seconds. Moving to the larger matrix of 90,000 items, the CPU took over 100 minutes while the GPU completed the same work in under 4 minutes.

A second approach is branch and bound shown in Figure 5.2 and researched by Boukedjar et al. [2012], whose work was continued by Lalami and El-Baz [2012]. The branch and bound 0-1 knapsack algorithm attempts to prune solutions that appear at a high level to be less than optimal so as to avoid having to traverse those branches running calculations which cannot improve the solution. The later work by Lalami and El-Baz [2012] showed that this type of algorithm could work on the GPU (which traditionally does not do well with branching due to the random memory access pattern), providing a speedup over a CPU-based algorithm. In their results, for a matrix size of 100, the GPU was able to achieve a speedup of 8.48 times, with an execution time of 0.18 seconds, compared to the CPU's 1.59 seconds. At the upper end, choosing from 500 items, the GPU completed in 0.65 seconds with respect to the CPU's 13.39 seconds, for a speedup of 20.48 times.

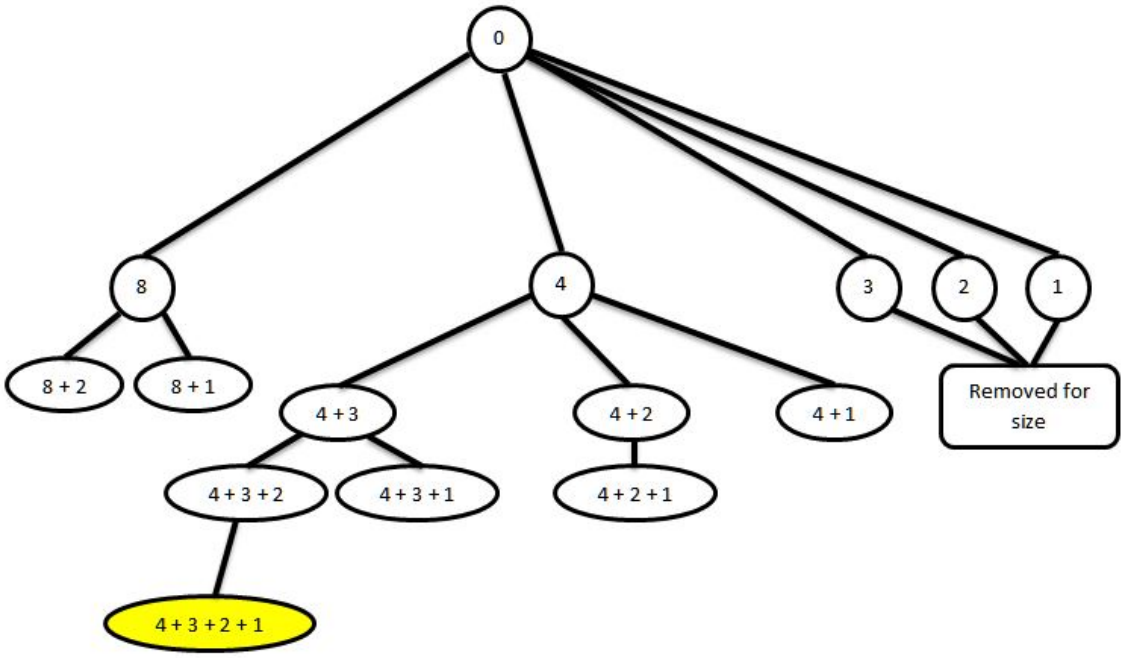


Figure 5.2: A short example of the 0-1 knapsack problem using a greedy branching approach (over capacity solutions not shown). We start by looking at the highest profit items and examining all possible solutions. As we move through the tree, we examine all solutions except ones containing items already searched (to the left). For example, in the branch starting at 4, we don't look at any solutions containing 8 as we would have covered the 8+4 solution in branch 8 and found it to be invalid.

5.3 Implementation and Results

5.3.1 Algorithm 1: Naïve

In my first algorithm (Doerksen et al. [2011]), I used a single compute unit of the GPU to obtain a basic working implementation of the 0-1 knapsack algorithm. This approach passed control back and forth between the CPU and GPU after each row had been computed. The next line was then transferred to the GPU, computed and returned, etc. As the profiler was not available at the time of publication (Doerksen et al. [2011]), I was unable to analyze the performance of the algorithm directly.

However, with my knowledge of how the APU works and GPU computing in general, I knew that the bottleneck was the memory transfer over the PCI Express bus and the context switching that occurs when transferring control from the CPU to the GPU and vice versa. As shown in Figure 5.3 (which was captured using the OpenCL profiler), we have over 1000 read/writes, one for each time the CPU had to pass information to the GPU. The inefficiency of data transfer manifests itself in the allocation of time spent computing vs. communicating, 1:99 percent in this first algorithm, and a very slow execution time of over 2.5 seconds compared to the sequential algorithm's execution time of 0.1 seconds (see Figure 5.4). At the same time, the APU also ends up slower than our parallel implementation which used OpenMP on the CPU. Using this profiling information, I was able to perform guided optimizations (examining both software and hardware counters) to iteratively improve the execution time of my algorithms.

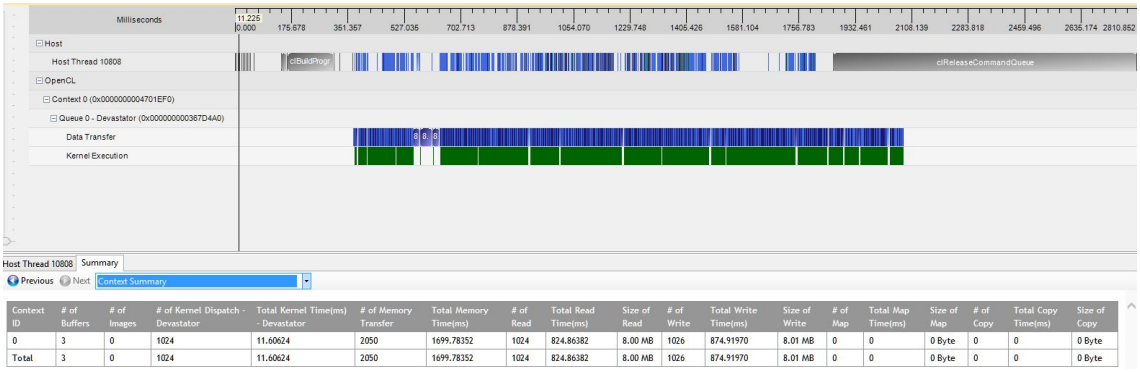


Figure 5.3: Algorithm 1 of the dynamic programming algorithm for 0-1 knapsack. We can see many data transfers are being done and that our time spent executing the kernel is less than one percent of the overall execution time.

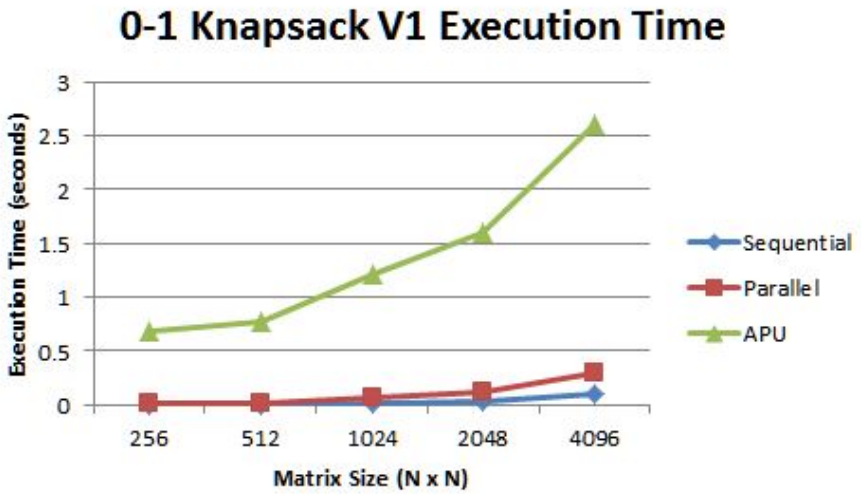
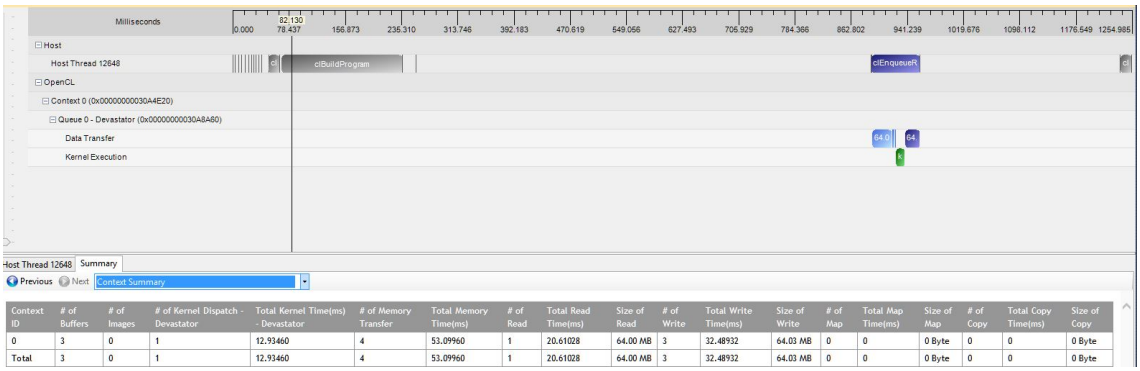


Figure 5.4: Execution times for the naïve algorithm where we see the penalty the APU pays for this unnecessary data transfer.

5.3.2 Algorithm 2: Loop Unrolling

The goal of the second algorithm that I created was to remove the PCI Express bottleneck in the system by attempting to keep data on the GPU for as long as possible. This was done by passing multiple rows at each iteration to reduce the number of context switches between computing on the CPU and GPU. Figures 5.5 and 5.6 show the result of eliminating these data transfers done at the beginning/end of the GPU's calculations. We now have only four data transfers in our algorithm but this change now necessitated row-level synchronization (between local workgroups) within the OpenCL kernel to avoid race conditions. We can also see that we are transferring a relatively large amount of data, nearly 128 MB in total which takes 53 milliseconds. Compared to the kernel which only takes 13 ms, we've gone from 1% computation time, up to 20% of total time, with an execution time just over 0.05

seconds, approximately twice as fast as the sequential algorithm. While an excellent improvement, it would be better if we could take that memory time and reduce that further.



Context ID	# of Buffers	# of Images	# of Kernel Dispatch - Devastator	Total Kernel Time(ms) - Devastator	# of Memory Transfer	Total Memory Time(ms)	# of Read	Total Read Time(ms)	Size of Read	# of Write	Total Write Time(ms)	Size of Write	# of Map	Total Map Time(ms)	Size of Map	# of Copy	Total Copy Time(ms)	Size of Copy
0	3	0	1	12.93460	4	53.09960	1	20.61028	64.00 MB	3	32.48932	64.03 MB	0	0	0 Byte	0	0	0 Byte
Total	3	0	1	12.93460	4	53.09960	1	20.61028	64.00 MB	3	32.48932	64.03 MB	0	0	0 Byte	0	0	0 Byte

Figure 5.5: Algorithm 2 of the dynamic programming algorithm for 0-1 knapsack. I've removed the thousands of write/read cycles present in the first algorithm, enabling data to stay on the GPU for as long as possible.

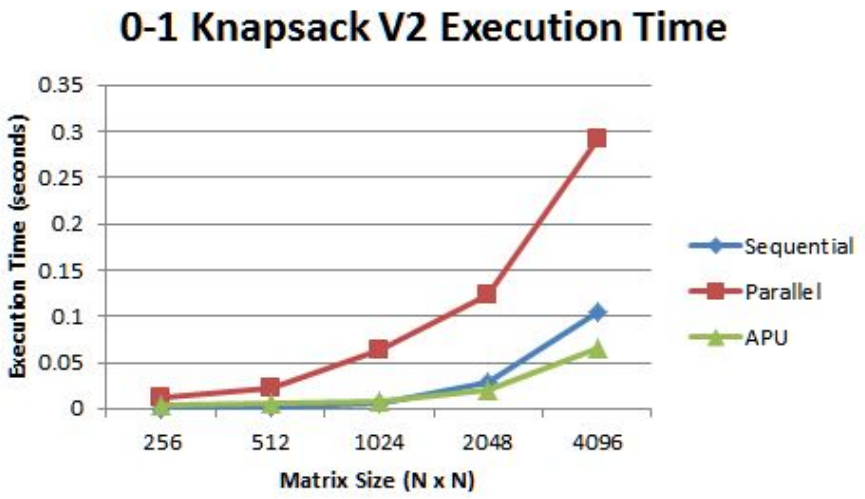


Figure 5.6: Execution times for the second algorithm where we kept the data on the GPU as long as possible by not returning control to the CPU after each iteration.

5.3.3 Algorithm 3: Scaling and Optimization

With the third algorithm, I implemented host memory and enabled the algorithm to use all compute units in the APU. From Figure 5.7 we see that execution time hasn't actually changed much, even though we cut the memory transfer time to almost zero as seen in Figure 5.8. Looking at this result doesn't make much sense since, as we've decreased the transfer time, shouldn't the overall execution time be lowered to the kernel time? To see why this happens we need to look closer, at Figure A.1 where we see that the ALUBusy, FetchUnitBusy and WriteUnitStalled counters have gotten worse.

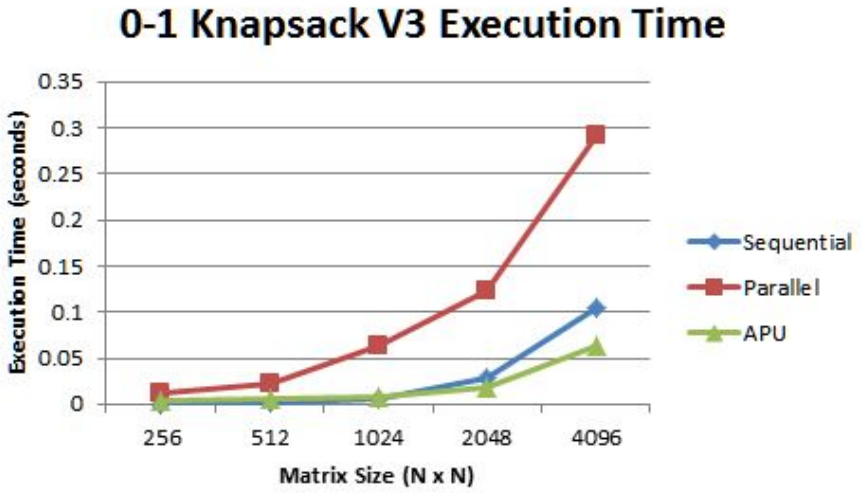
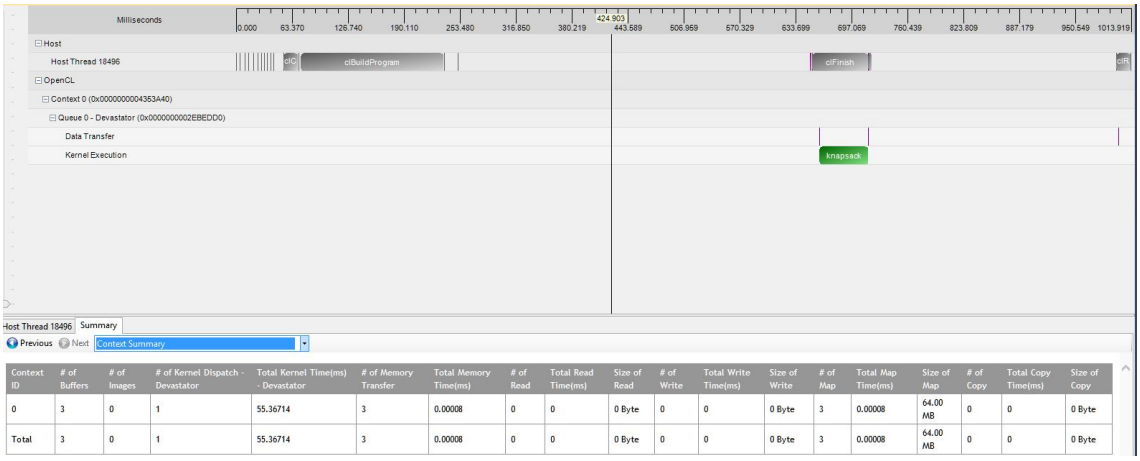


Figure 5.7: Execution times for the third algorithm which uses all computation units in the GPU and removes all GPU-located data buffers in favor of using host memory.



Context ID	# of Buffers	# of Images	# of Kernel Dispatch - Devastator	Total Kernel Time(ms) - Devastator	# of Memory Transfer	Total Memory Time(ms)	# of Read	Total Read Time(ms)	Size of Read	# of Write	Total Write Time(ms)	Size of Write	# of Map	Total Map Time(ms)	Size of Map	# of Copy	Total Copy Time(ms)	Size of Copy
0	3	0	1	55.36714	3	0.00008	0	0	0 Byte	0	0	0 Byte	3	0.00008	64.00 MB	0	0	0 Byte
Total	3	0	1	55.36714	3	0.00008	0	0	0 Byte	0	0	0 Byte	3	0.00008	64.00 MB	0	0	0 Byte

Figure 5.8: Algorithm 3 of the dynamic programming algorithm for 0-1 knapsack. Here I've switched to using host memory instead of global memory and enabled the algorithm to scale across all compute cores.

Interpreting Performance

What Figure A.1 means is that our hardware is now less utilized, 27% for the ALU, 22% for the FetchUnit, and our write unit is stalled (albeit to an unnoticeable degree). Comparing the two we see that the proportions haven't changed, as the ALU:Fetch Percentage ratio is still approximately 1.2:1, so what could have caused hardware utilization to decrease? This is answered in the other change that was made to the algorithm, scaling it across all compute units. Remember from the section on OpenCL, we use groups of 256 items (the optimal size for the APU for this problem) and if we are using more, synchronization must occur to avoid race conditions. In the second algorithm, we used a single compute unit and had it calculate all items, meaning synchronization only occurred within a single block. Changing the algorithm though now means we have to sync 16 workgroups across each row (with a workgroup size of 256 and a row length of 4096). With these small problem sizes, we see this

extra synchronization manifest itself silently in the kernel's execution time and is where we must now focus our attention.

5.3.4 Algorithm 4: Caching

The final algorithm change I implemented was based on the third algorithm. Now, I also switched to using local memory in the kernel to cache data elements within a workgroup and reduce the overall number of accesses to global memory. Looking back to Chapter 4, you'll see that local memory has a peak bandwidth of 3.7 TB/s, compared to global memory's 264 GB/s (a best case for a dedicated GPU, our system memory is actually much slower). This has great potential to speed up our algorithm, and as we can see in Figures 5.9 and 5.10, execution time for the kernel was cut almost in half. As well, looking at the change in values of our hardware counters in Figure A.2, we see that while our FetchUnitBusy percentage hasn't increased, we did double our ALUBusy time which shows in our results where execution time dropped from 0.06 seconds for a 4096x4096 matrix, down to 0.04 seconds. Not shown here is the result for an 8192x8192 matrix which was 61 times faster than the sequential algorithm; not bad for an integrated GPU.

5.4 Summary

We began with a naïve algorithm using just a single compute unit to establish a working baseline of the 0-1 knapsack problem on an APU. Then, through profiling the code we found the first bottleneck, data transfer between the CPU and GPU at each iteration and were able to remove it, increasing our performance to 1.4 times that of

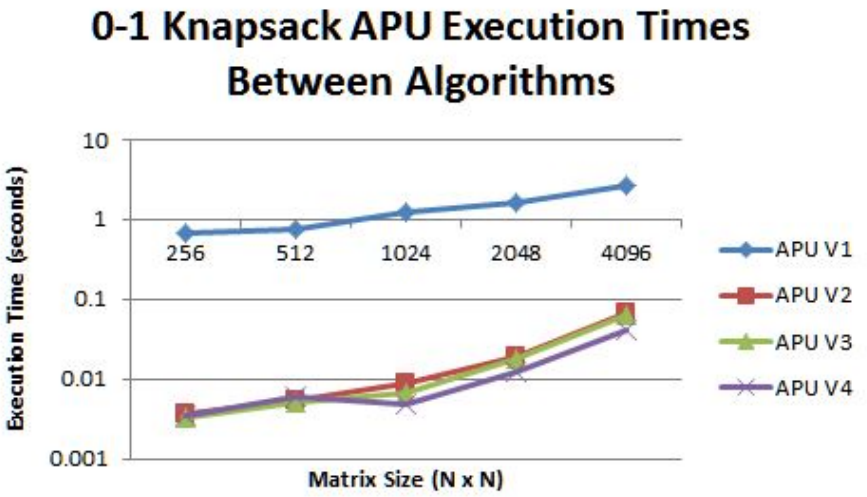


Figure 5.9: A comparison of the execution times of the algorithms as we've progressively optimized them.

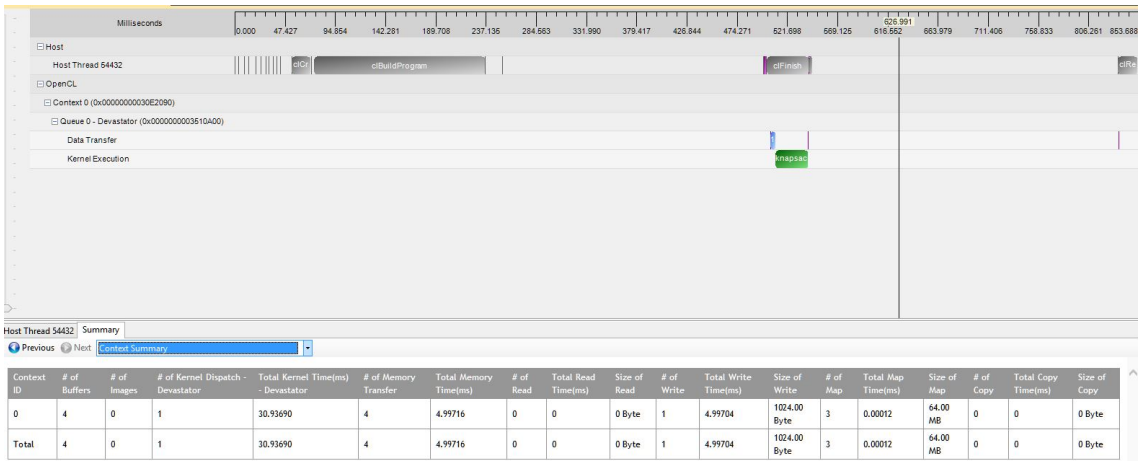


Figure 5.10: Version 4 of the dynamic programming algorithm for 0-1 knapsack which uses local memory to reduce the number of reads from global memory.

the sequential algorithm. We continued examining the algorithm to try and find more areas hurting performance and determined that, with the advanced APU architecture, removing memory transfer is possible and resulted in further performance gains. In

practice, we saw only a slight benefit of the APU's memory architecture come through as the extra synchronization between workgroups brought our execution time almost to where it was before, but it did provide us with an algorithm that could be scaled up to larger inputs relatively easily. Finally, we introduced local memory to reduce the amount of memory transfers to global memory which increased our performance by nearly 50% over our previous APU algorithm, and 61 times faster than the sequential algorithm for an 8192x8192 matrix.

5.5 Future Work

While we achieved excellent results, there are a few other points of interest. These include vectorization of the algorithm so that multiple work items can be grouped onto the hardware and computed at the same time (at an even finer grained level than workgroups). This algorithm change may potentially increase performance up to 16 times on current GPUs (this is the maximum vectorization level currently supported by OpenCL), though performance in practice would likely end up in the 2-8 times range based on the hardware setup (the A10-5800K APU uses a VLIW4 GPU) and increased resource utilization that vectorization would provide. As we see later in Chapter 7, during the implementation of the Monte Carlo algorithm, register pressure becomes a fundamental bottleneck for performance. With the 0-1 knapsack problem though, we can not use vectorization to hide this latency due to the amount of synchronization, which hurts our ability to scale the algorithm using this technique. Second, and not entirely unrelated is AMD's next generation of APUs which use the GCN architecture which has proven to be on average almost 40% faster

for compute work while at a 17% disadvantage in GFLOPS, a 50% disadvantage in shaders and a 78% disadvantage in memory bandwidth (Smith and S [2012]). Finally, while not completely solving the scaling issue as programs must still fit within GPU memory, the problem could be broken down for incredibly large problems (hundreds of millions) so that a subset of the problem could be worked on by the GPU. This would however reintroduce some of the data transfer issues of the first algorithm, but with the potential to use host memory on the APU (particularly on AMD's Kaveri APU which passes only memory pointers without any copying required), would likely end up scaling far better than passing data through buffers.

Chapter 6

Gaussian Elimination

6.1 Problem Definition

Gaussian elimination is a method for solving a system of linear equations. It reduces the set of equations such that we are able to find a value for each of the variables so as to satisfy the overall system. While not as easily pliable to applications compared to the 0-1 knapsack problem, matrices of equations define the world of mathematics around us and as such, make up components of the systems we see around us; an example of Gaussian Elimination can be seen in Figure 6.1.

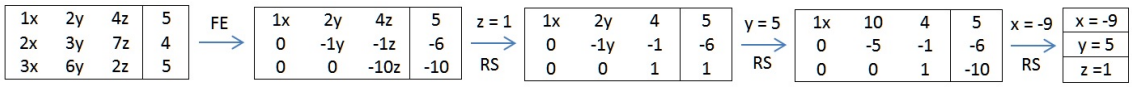


Figure 6.1: A simple example of Gaussian Elimination which uses reverse elimination to zero all values in one row except for one column. This value is then substituted into the other formulas and elimination is performed again. In this simple case, elimination isn't used to zero out the second row after the value of z is found since we can calculate the value of y as we know all other columns.

Algorithm 1: Pseudocode for the reverse elimination portion of the naïve algorithm.

```
for i ← n − 1 to 0 do
    // use the element at [i][n] as the pivot value
    // matrix[i][n] = matrix[i][n] / matrix[i][i]
    // matrix[i][i] = 1
    for j ← i − 1 to 0 do
        // zero all remaining columns leaving a single value
        // matrix[j][n] -= matrix[j][i] * matrix[i][n]
        // matrix[j][i] = 0
    end
end
```

6.2 Related Works

As eliminating variables is at the heart of solving a system of linear equations, the possible methods of solving a system of linear equations are quite limited and are instead focused on optimizing the system to reduce the number of calculations (but not reduce its time complexity unless certain conditions were met). One method, proposed by Gohberg et al. [1995] involved transforming matrices of the forms Toeplitz-like, Toeplitz-plus-Hankel-like and Vandermonde-like into Cauchy-like matrices for which they were able to create an $O(n^2)$ algorithm which used partial pivoting. The work of Demmel et al. [1999] however was able to produce a faster Gaussian Elimination algorithm, netting speedups averaging 2.6 times over multiple machines when run with four processors. Their algorithm was highly customized however (compared to the standard algorithm), using graph reduction and a custom scheduler to pipeline execution to remove dependencies. Lastly, we have the work of Che et al. [2008] which used a customized algorithm, and found that from a cumulative clock cycle perspective, the GPU was in fact able to outperform the sequential algorithm. Concluding,

while it is possible to speed up the Gaussian Elimination algorithm, it can't be done with the simple naïve algorithm just yet.

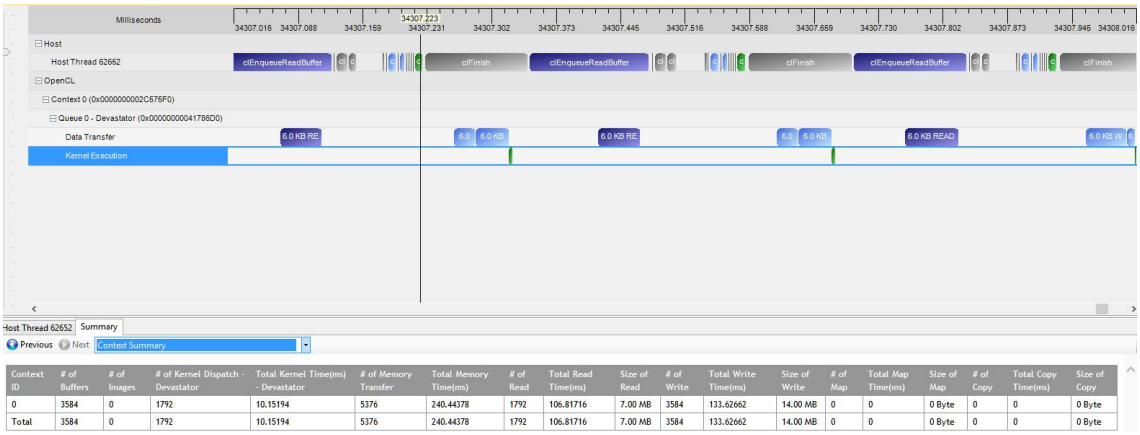
6.3 Results

In this section, I present several variants of the Gaussial Elimination algorithm. By using profiling information, I attempt to iteratively improve the algorithm's performance.

6.3.1 Algorithm 1: Naïve

As we did for the 0-1 knapsack problem, we start with a simple port of the CPU-based algorithm to create a baseline for algorithm performance. For this problem, we examined the reverse elimination portion of the algorithm since it runs in $O(n^2)$ time and would quickly show if our optimizations were successful. Our naïve approach however leads to less than optimal performance, as seen in Figures 6.2 and 6.3, because we return control to the CPU after each iteration for processing that must be done before the parallel work can be started. This leads to performance worse than CPU, which also happens with the parallel algorithm. The root cause for this performance loss is the number of forks/joins that must take place because the inner loop is the one that is parallelized.

Moving to the execution, we see that only 4% of our total time is spent actually executing our kernel; the rest is spent waiting on memory operations. We also know from the algorithm that there are very few calculations done in each iteration of the loop, meaning overhead will have to be low in order to end up ahead. Examining the



Context ID	# of Buffers	# of Images	# of Kernel Dispatch - - Devastator	Total Kernel Time(ms) - Devastator	# of Memory Transfer	Total Memory Time(ms)	# of Read	Total Read Time(ms)	Size of Read	# of Write	Total Write Time(ms)	Size of Write	# of Map	Total Map Time(ms)	Size of Map	# of Copy	Total Copy Time(ms)	Size of Copy
0	3584	0	1792	10.15194	5376	240.44378	1792	106.81716	7.00 MB	3584	133.62662	14.00 MB	0	0	0 Byte	0	0	0 Byte
Total	3584	0	1792	10.15194	5376	240.44378	1792	106.81716	7.00 MB	3584	133.62662	14.00 MB	0	0	0 Byte	0	0	0 Byte

Figure 6.2: Profiler output for the naïve approach to parallelizing Gaussian Elimination. Note the amount of time spent waiting between iterations because the GPU cannot deal with small input sizes.

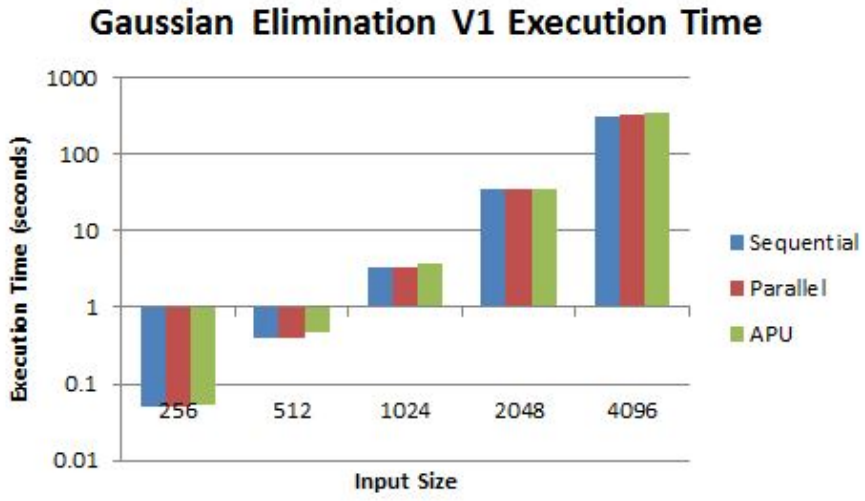


Figure 6.3: The execution time for our first algorithm, exhibiting performance following that stated by Amdahl's Law.

hardware counters in A.3 we can see the result of this, where we have an ALU/Fetch ratio of only 3.5. and the amount of time the hardware is busy running calculations is extremely low, averaging 3%.

6.3.2 Algorithm 2: Optimizing Memory Transfer

In an attempt to remove the limitation of memory and play the strengths of the APU, I used host memory in place of device buffers. Unfortunately, this didn't increase performance much, only 1%, but performance was still lagging behind the sequential version by over 10%, as can be seen in Figure 6.4. The profiler also showed that synchronization was not required after each iteration, and so removing it did increase performance, but to the degree of only 1%.

Figure 6.4: Examining the execution time we see minimal difference to the first algorithm due to the speed of sequential calculations, even with a reduced reliance on memory.

The problem with this algorithm is not actually the memory maps that are taking place, shown by Figure 6.5, but rather the fact that we must call out from our program to the OS to make the context switch to give control back to the CPU to perform the memory map. It is in this respect that AMD's future generations of APUs should perform extremely well, as they will be able to queue from the GPU to the CPU and

vice versa without any OS interaction, eliminating all time now spent doing memory map operations. Now, the only other way to speed up the algorithm would be to manually remove these operations.

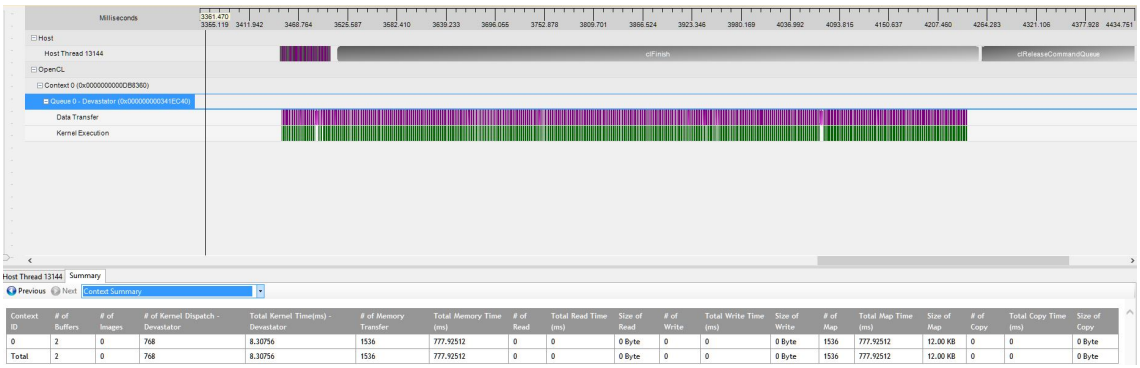


Figure 6.5: The second algorithm exhibits the same performance characteristics as the first algorithm since we are not actually limited by memory.

6.3.3 Algorithm 3: Non-traditional Approach

In the final algorithm, I decided to go with a very non-traditional GPU programming method, using the APU for all calculations including the sequential portion. I chose this approach as Kaveri was not available at the time, which would benefit from passing only memory pointers, leading to less context switching between the two devices. To build the algorithm this way involves making the threads synchronize before and after the sequential work and have only a single thread run the sequential calculations. In this manner, the tradeoff is using the APU's slower GPU resources (single-threaded) in exchange for removing the OS context switches.

Examining the results of Figure 6.6 we see something astonishing. For an input of 1024, we match that of the sequential algorithm, and for 2048 we actually beat it

Figure 6.6: The third algorithm's execution time which shows we were actually faster for smaller input sizes, meaning that with a more advanced APU like Kaveri we might actually obtain a speedup with large inputs.

(albeit minimally). Unfortunately, as we move to 4096 items, the combination of the extra synchronization (since we're using even more workgroups) and more sequential items to calculate, the program execution does not complete within a reasonable amount of time. On the upside, this does show promise for our previous hypothesis of using Kaveri's heterogeneous CPU-to-GPU queues to remove context switching and its potential for accelerating this difficult problem.

6.4 Summary

Gaussian Elimination has proven to be a very difficult algorithm to optimize for performance. Even using host memory (playing to the APU's strengths) we were not able to overcome the inefficiencies of interacting with the OS every iteration to han-

dle device memory operations. However, the attempt at non-traditional processing methods shows promising results for small input sizes where we were able to beat the sequential result, proving the concept of single device computing (even on a GPU). With next generation APUs, this bottleneck will be removed as OpenCL CPU and GPU devices will be able to queue work directly to each other, without interaction from the OS, and while remaining in userspace. This should finally make this difficult algorithm possible to parallelize, paving the way for optimizing similar problems.

6.5 Future Work

In the future, I would like to re-examine this problem with either Kaveri or AMD's next generation APU which would include the ability to heterogeneously queue from the CPU to the GPU and vice versa. This would enable us to remove the overhead resulting from context switching between the devices that is present in the current systems. As well, I would like to investigate the use of vectorization to accelerate the problem since it should provide better GPU performance once we are able to seamlessly switch between computing devices.

Chapter 7

Monte Carlo Simulation

7.1 Problem Definition

A Monte Carlo algorithm works by using randomization to build a solution. For American option pricing it calculates potential stock prices according to probability using a function which takes many variables including: strike price, expiration date of the option, interest rate, stock volatility and growth rate of the stock. This function is run many times and averages the results to obtain what should be a good solution, with a given probability of the answer being incorrect. The difficulty in the Monte Carlo method is that it is very compute intensive, often requiring hundreds of thousands or millions of iterations to come to an answer that can be used. However, it performs very well for systems with many coupled variables like fluid modeling, artificial intelligence and economics and as such, any way to speed up this technique would provide real world results.

7.2 Related Works

As Monte Carlo simulations have been around for so many years after the introduction of the first computer and they are found in so many areas, I will only cover the techniques used to accelerate option pricing, namely parallel processing with CPUs, GPUs and dedicated hardware.

The first method is to introduce parallel programming, where one (or many) systems calculate a portion of the result set and combine them to determine a final value. Dockner and Moritsch [1999] took the approach of using a distributed memory system with SPARC computing nodes to accelerate their algorithm. As the Monte Carlo algorithm only incurs communication time at the very beginning (to tell each worker what to do) and very end of its work (to combine the results), the execution time drops almost linearly with the number of cores. As such, they were able to drop the execution time from 0.9 seconds to less than 0.1 seconds. Based on what we know about the Monte Carlo algorithm, it will scale almost linearly as you add more hardware, meaning these results fall in line with expectations but will likely soon run into the minimum time to compute wall due to running out of processing power.

Next is the use of the graphics processing unit which has hundreds of cores that can work independently, much like a multi-core system. Abbas-Turki and Lapeyre [2009] took this approach and modified the Monte Carlo algorithm so that it could run on the GPU. With the parallel processing capabilities of the GPU, the authors were able to achieve a 1.4 and 64.5 times speedup (for the slowest and fastest portions of the algorithm) over the sequential CPU-based algorithm. This type of speedup shows that the GPU is well suited to Monte Carlo simulations and there may only be one

device that can do better, dedicated hardware.

The last major option to speed up Monte Carlo option pricing is to use hardware tailored for the specific purpose of calculating option prices. This was implemented by Sanchez-Roman et al. [2013] whose approach was to design an algorithm for Asian options to be executed in a field-programmable gate array. By implementing the algorithm in hardware, the inefficiencies of an OS and generic hardware are removed, resulting in better performance with the tradeoff of a non-portable algorithm. Taking this approach, they were able to achieve a speedup of over 500 times that of the sequential CPU algorithm when run with 3650 time steps and 10 million simulations. These results are quite impressive, showing what dedicated hardware can do to speed up a problem. The only downside to this approach is that it's not easily accessible to most people as it requires specific hardware and must be programmed with only this function.

7.3 Results

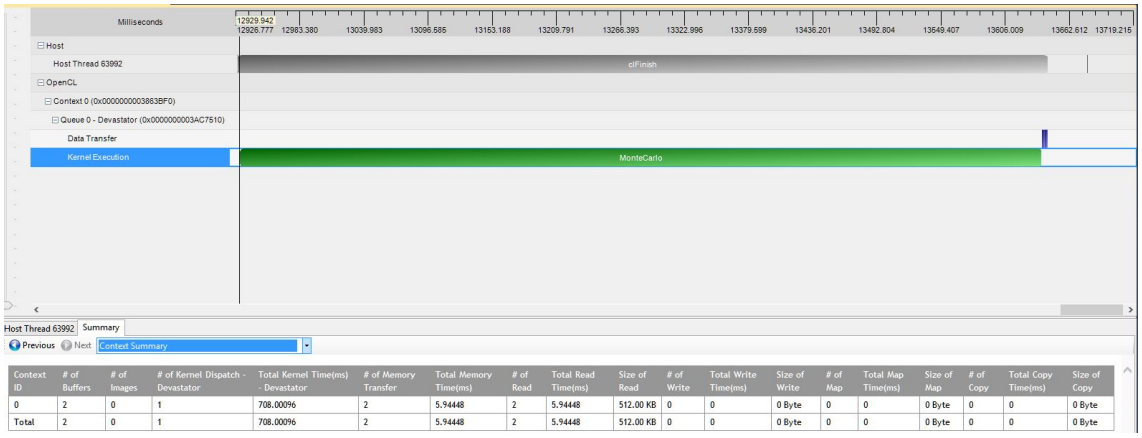
In this section, I provide several implemantions of the Monte-Carlo method for the APU.

7.3.1 Algorithm 1: Naïve

I began by creating a simple version which used all available hardware resources on the GPU but communicated using global memory (Doerksen et al. [2012b]). I profiled the execution of this algorithm and noticed a few key things: namely the memory accesses, the kernel occupancy, ALUBusy percentage and ALUPacking percentage.

Algorithm 2: Source code for the Monte Carlo naïve algorithm.

```
// initialize parameters
// K: strike price
// T: time step
// S: stock price
// δ: continuous dividend yield
// σ: volatility
// N: number of steps
// M: number of simulations

dt = T/N;

// r is the interest rate
nudt = (r − δ − (pow(σ, 2)/2))dt;
sigsdt = σ ∗ sqrt(dt);
sumCT = 0;
sumCT2 = 0;

for j ← 0 to M do
    lnSt = log(S);

    for i ← 0 to N do
        lnSt = lnSt + nudt + sigsdt*rand();
    end
    // stock price at time T
    ST = exp(lnSt);
    CT = max(0, ST - K);

    // sum of payoffs at maturity date
    sumCT = sumCT + CT;
    sumCT2 = sumCT2 + CT*CT;
end
// final option value, averaged sumCT and discounting current date to T
c0 = sumCT/(M*exp(-r*T));
// std deviation of the option values
SD = sqrt((sumCT2 - sumCT*sumCT/M)*exp(-2*r*T)/(M-1));
// standard error
SE = SD/sqrt((float)M);
```

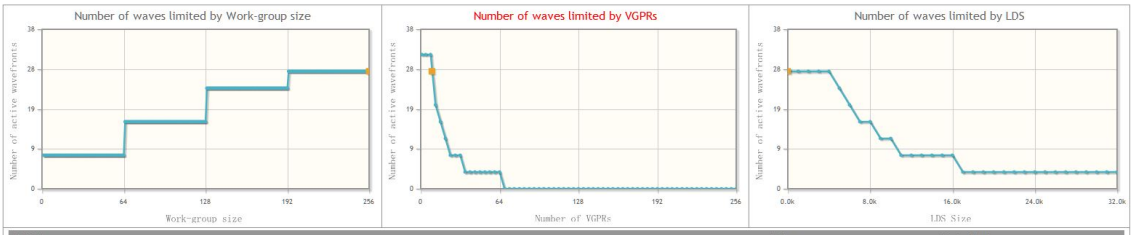


Context ID	# of Buffers	# of Images	# of Kernel Dispatch - Devastator	Total Kernel Time(ms) - Devastator	# of Memory Transfer	Total Memory Time(ms)	# of Read	Total Read Time(ms)	Size of Read	# of Write	Total Write Time(ms)	Size of Write	# of Map	Total Map Time(ms)	Size of Map	# of Copy	Total Copy Time(ms)	Size of Copy
0	2	0	1	708.00096	2	5.94448	2	5.94448	512.00 KB	0	0	0 Byte	0	0	0 Byte	0	0	0 Byte
Total	2	0	1	708.00096	2	5.94448	2	5.94448	512.00 KB	0	0	0 Byte	0	0	0 Byte	0	0	0 Byte

Figure 7.1: The context summary for the Monte Carlo algorithm shows almost no memory accesses being performed meaning there is little to optimize in this respect.

Starting with the memory accesses as shown by Figure 7.1, we see that very little time is spent doing memory operations, meaning there is no need to optimize for this. At the same time however, due to this behavior, we can also say that the APU's strengths will not be shown with this algorithm since it's entirely compute-bound. As well, from Figure A.4 we see that kernel occupancy is not 100%, but what does this mean? For that we turn to Figure 7.2 which shows high level information about our kernel running on our hardware and how it is limited. In this case, we see that our program is limited by Vector General Purpose Registers (VGPRs), or the number of registers used to store vectors. Due to this we can only run 28 workgroups instead of 32 on our hardware, losing 12.5% performance immediately. Resulting from this are the other two noteworthy variables, ALUBusy percentage and ALUPacking percentage which are reduced because we have run into hardware limitations due to using too many registers.

The result of this simple Monte Carlo option pricing algorithm actually turns out



Variable	Value	Device Limit
Device Info		
Device name	Devastator	
Number of compute units	6	
Max number of waves per compute unit	32	
Max number of work-groups per compute unit	8	
Wavefront size	64	
Kernel Info		
Kernel name	MonteCarlo	
Vector GPR usage per work-item	9	256
LDS usage per work-group	0	32768
Flattened work-group size	256	256
Flattened global work size	32768	16777216
Number of waves per work-group	4	4
Kernel Occupancy		
Number of waves limited by Vector GPR and Work-group size	28	32
Number of waves limited by LDS and Work-group size	32	32
Number of waves limited by Work-group size	32	32
Limiting factor(s)	VGPR	
Estimated occupancy	87.5%	

Figure 7.2: Kernel occupancy graph which examines your hardware and kernel and advises you on what is bottlenecking your program.

to be very good, even if we aren't running the hardware to its maximum capabilities. From Figure 7.3, the APU's execution time is only 0.7 seconds for 32,768 loops using 131,072 random data points while the sequential algorithm is over 10 seconds with the multi-core OpenMP algorithm sliding in between at 2.6 seconds. As this algorithm was the most basic form we could create (minimal effort is required to implement the algorithm), these results are very good, showing off what the GPU can do for these trivially parallel programs. However, we can't stop here since more performance could be achieved, which is why I looked to use local memory instead of registers to store variables.

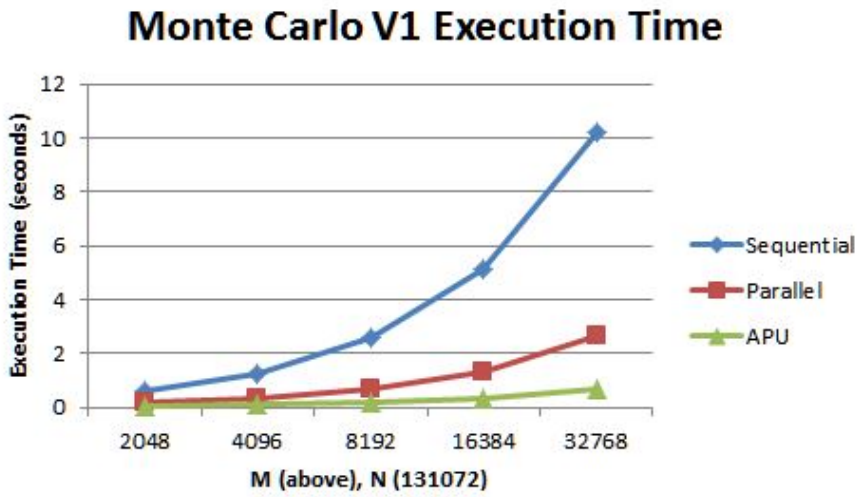


Figure 7.3: The APU, with its limited computational resources still manages to outperform the CPU quite handily, even with a basic, ported algorithm.

7.3.2 Algorithm 2: Loop Unrolling

In an attempt to reduce the amount of registers used by the algorithm, I placed all possible variables in local memory instead of creating private variables. Unfortunately, this had no effect on reducing the number of VGPRs used by the algorithm. Instead, as seen in Figure 7.4, execution time actually increased which I attribute to the slower speed of local memory compared to registers since the hardware built caching mechanism could not make up for the slower speed. Coupled with this, we weren't able to run more waves on the hardware, leading to no performance benefit. Since altering the memory scheme didn't help with freeing up the number of registers I examined another optimization technique, vectorization, to see if it would be possible to "power through" this disadvantage and get more use from the hardware we do have since some of the registers should be possible to re-use.

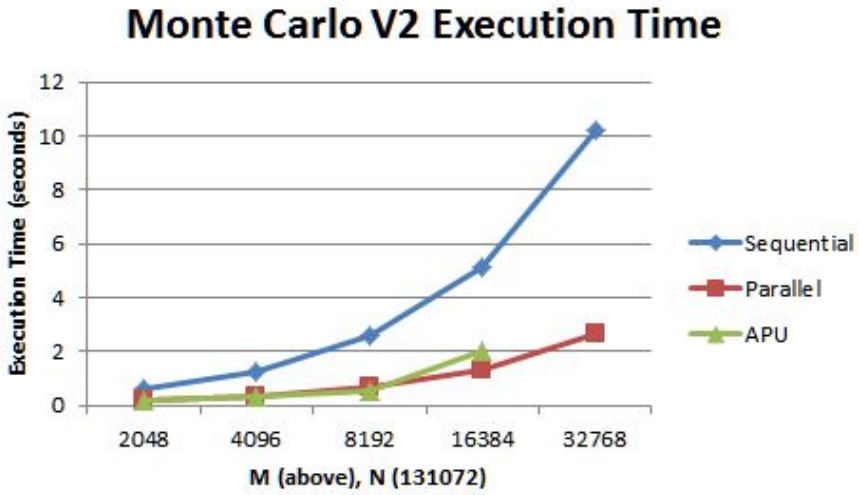


Figure 7.4: Using local memory to attempt to reduce register pressure has actually decreased performance to less than that of our parallel CPU algorithm, forcing us to look in other areas for improvement.

7.3.3 Algorithm 3: Vectorization

This final algorithm reverted back to using global memory and vectorization to take advantage of the APU's VLIW4 architecture. While testing these changes, I had to answer the question, would 8 and 16 element vectorization actually improve performance or hurt it since the hardware itself is only built to handle 4 elements? The short answer is yes, we do see a performance improvement even after passing 4 element vectorization. I attribute the gains to better scheduling (or instruction queueing) which keeps the ALUs busy. The net gain can be seen in both Figures 7.5 and 7.6 where we now execute the largest input size in only 0.09 seconds, or a speedup of 7.7 compared to our first algorithm, and 113 times faster than the sequential algorithm. Looking one step further, we were able to execute 131,072 items for 131,072 time steps in only 0.25 seconds using 16 element vectorization,

nearly three times as fast as our first algorithm while computing 4 times the amount of data points.

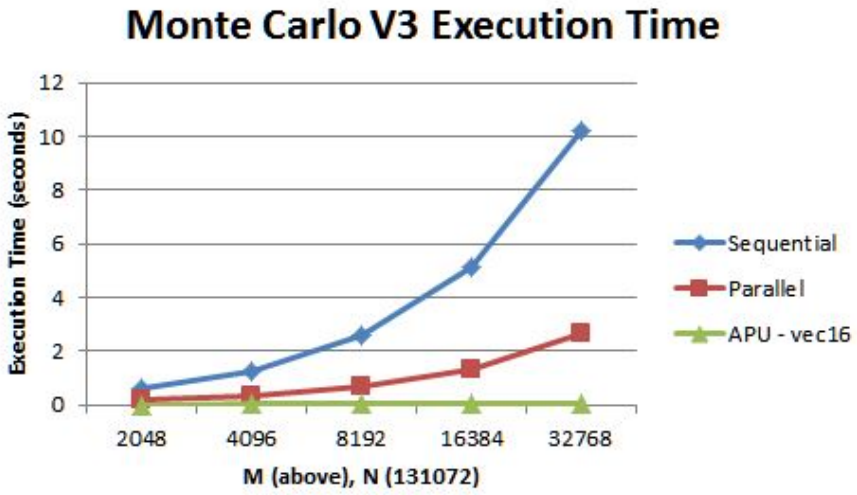


Figure 7.5: The final algorithm uses vectorization to compute multiple items simultaneously on the hardware (at a finer-grained level than SIMD engines) leading to a drastic performance improvement.

Moving to the scaling of vectorization, we see a near linear decrease in execution time by continually doubling the number of elements to vectorize. At the same time however, looking at Figure 7.7 we see that this has had a major effect on the number of waves that the hardware can schedule, reducing overall performance since we can now only run the hardware at 12.5% of its capacity. This is unfortunate since we've now hit a hard limitation unless there is a way to reduce the register pressure that occurs with vectorization. Alternatively, using a high-end GPU would likely reduce this limitation as there would be more registers, improving performance dramatically.

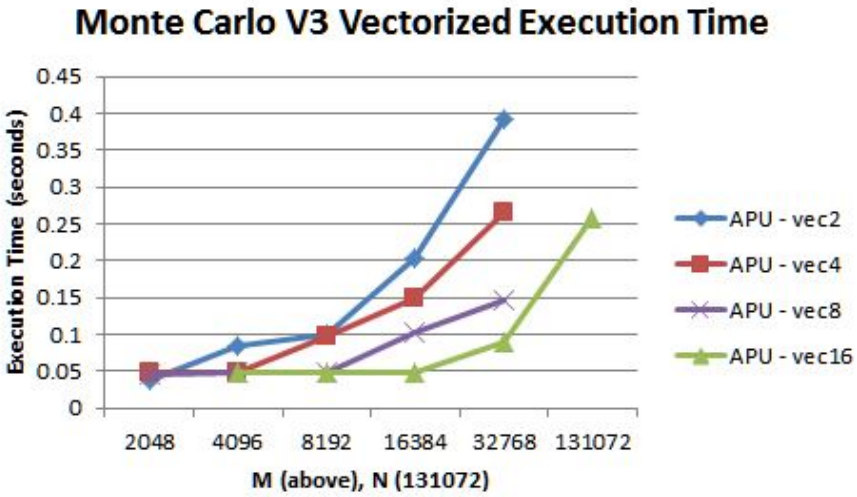
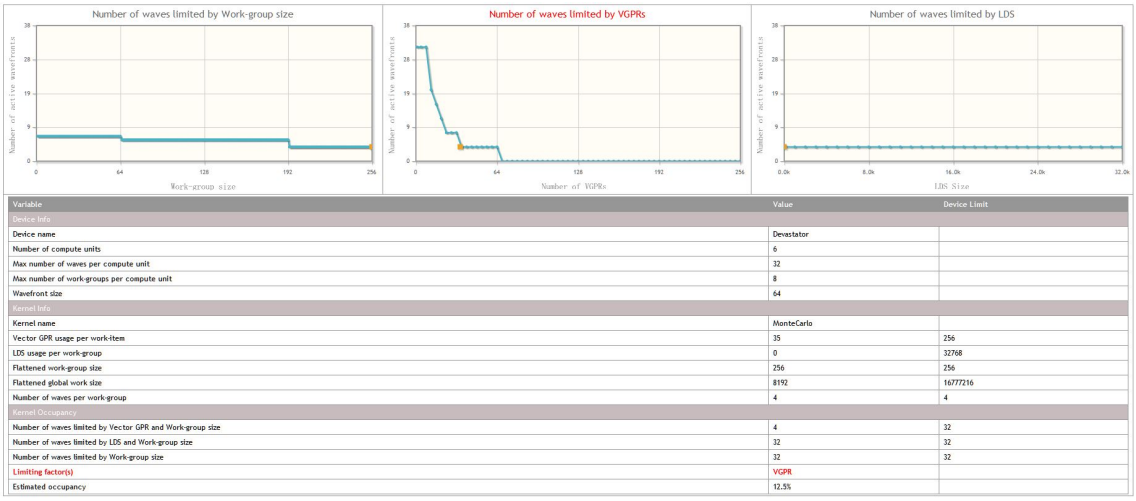


Figure 7.6: Comparing the performance of vectorized algorithms we achieve almost linear scaling all the way up to 16 elements. For problems like Monte Carlo simulations, extremely wide parallelism cannot be beaten.

7.4 Summary

We began with a simple Monte Carlo algorithm for option pricing and ported it directly onto the APU. From this, we were able to achieve a speedup of 14 times that of the sequential algorithm with minimal coding required. Next, since we were limited by the number of registers in the first algorithm, I attempted to use local memory to store private variables. Unfortunately this did not work and actually led to performance degredation. Lastly, to try and extract more performance out of the hardware I turned to a technique called vectorization which groups data elements together for smarter memory accesses and, in the case of AMD's hardware in particular, better scheduling and packing of instructions onto the hardware for better utilization. This increased performance dramatically, now over 110 times faster than the sequential algorithm which is quite a feat considering this is just an APU which has nowhere



Variable	Value	Device Limit
Device Info		
Device name	Devastator	
Number of compute units	6	
Max number of waves per compute unit	32	
Max number of work-groups per compute unit	8	
Wavefront size	64	
Kernel Info		
Kernel name	MonteCarlo	
Vector GPR usage per work-item	35	256
LDS usage per work-group	0	32768
Flattened work-group size	256	256
Flattened global work size	8192	16777216
Number of waves per work-group	4	4
Kernel Occupancy		
Number of waves limited by Vector GPR and Work-group size	4	32
Number of waves limited by LDS and Work-group size	32	32
Number of waves limited by Work-group size	32	32
Limiting factor(s)	VGPR	
Estimated occupancy	12.5%	

Figure 7.7: The kernel occupancy graph for the 16-element vectorized algorithm. Note how we can now run very few waves on the hardware because we have run out of registers.

near the amount of computational resources compared to a dedicated GPU.

7.5 Future Work

Given that this is a trivially parallel algorithm, the amount of further optimizations are very minimal. At this time, the only work that remains to be done is to search for a method of reducing the register pressure that currently exists in the algorithm, particularly when vectorization is used.

Chapter 8

Binomial Lattice

8.1 Problem Definition

The binomial lattice for option pricing is one method to approximate the price of an option. Using the binomial lattice method, we essentially create a tree, as shown in Figure 8.1. Moving from left to right we have the number of time steps, which could in theory be increased to infinity (ignoring the limitations of computational feasibility). At the very right, or final time step, we have the maturity date of the option, or when it must be exercised. The points in between represent the price at a given time step, and, as we're using a binomial method, the price can in each time step, either increase or decrease. There are other, more advanced models such as the trinomial method where the price could increase, remain unchanged, or decrease at each step, as well as n-ary tree methods to enable more fine grained pricing (limited by computational power).

The option variation we study here is the American Lookback option which tries

to find the minimum or maximum value (based on buy or sell) of an option at a given point in time between the purchase date and maturity date. The American Lookback option is categorized as an exotic option because it has features not found in standard options, such as the ability to exercise the option any time before the expiration date (as opposed to a European option which can only be acted upon at the time of expiration). Based on the large tree structure of the problem after many time steps, and simplistic synchronization that occurs at every time step, we theorize that the APU should perform very well for this problem.

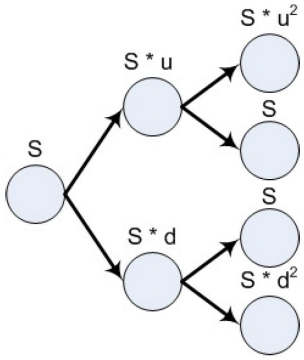


Figure 8.1: A binomial tree structure constructed for option pricing for 3 time steps (but could be sub-divided until we run out of computational power). Image taken from Solomon et al. [2010].

8.2 Related Works

Option pricing is not a new field and there are many ways to try and calculate an option's price, like Monte Carlo (covered in Chapter 6), binomial/trinomial lattice and finite-difference methods. As mentioned earlier, most work with Monte Carlo methods in this area has not been towards new algorithms but rather the optimization

of existing ones. These optimizations can be summarized to using new types of hardware to accelerate the problem better than that of a normal CPU.

One approach, taken by Solomon et al. [2010] was to use the GPU because of its massively parallel nature and high-speed memory. Through optimizations like using shared memory and memory coalescing, they were able to dramatically improve performance over their naïve GPU algorithm, obtaining a speedup of over 3.5 times. Additionally, the authors also implemented a hybrid version of their algorithm that would use the CPU simultaneously with the GPU. The result of this however was minimal to zero performance gain simply because the GPU dominates the overall execution time given how many work items it is calculating. Compared to the sequential CPU algorithm, they were able to achieve a speedup of over 100 times, demonstrating that special purpose hardware is the best method to calculate option prices.

Furthering the idea of special purpose hardware for computing is the work by Tse et al. [2009] who used a FPGA to accelerate quadrature methods used for option pricing. By removing the inefficiencies of "general purpose" hardware such as the CPU or GPU (though the GPU is better suited to this task than the CPU) they were able to create a system that would only be able to calculate option prices, but do it very fast and for little power. In their results, they managed to achieve a 32.8 times speed up over the CPU for single precision work while using only 4.4 watts (maximum power). Compared to the GPU their FPGA was approximately half as fast for single precision, but achieved those results at 1/13 the clock speed and using 45 times less power. These are very impressive results, but they do require specialized hardware only suited to calculate one thing meaning they can not be used by the layman; this

is where our research using APUs attempts to bridge the gap.

Algorithm 3: Source code for the Binomial Lattice naïve algorithm.

```
if globalID == 0 then
    yValue = pow(u, 0);
    tempOptionValues[0] = max(yValue - 1.0f, ((pu * optionValues[1] * d) +
    (pd * optionValues[0] * u)) * disc);
end
if globalID <= i then
    yValue = pow(u, globalID);
    tempOptionValues[globalID] = max(yValue - 1, ((pu *
    optionValues[globalID + 1] * d) + (pd * optionValues[globalID - 1] * u)) *
    disc);
end
// synchronize workgroups before writing to global memory

if globalID <= i then
    optionValues[globalID] = tempOptionValues[globalID];
end
```

8.3 Results

I implemented three different versions of the binomial lattice algorithm on the APU. This section highlights these implementations and provides an analysis on information retrieved by the profiler.

8.3.1 Algorithm 1: Naïve

The first algorithm I created was naïve, and a first attempt to get a working algorithm running on the APU (Doerksen et al. [2012a]). Given that this is a very computationally intensive algorithm, we see that in Figure 8.2 we still manage to

achieve better results than both the sequential and parallel algorithms without using any optimizations. For inputs smaller than 4096 time steps the APU has no benefit over the sequential algorithm, but once we reach 32,768 time steps, we managed to obtain a speedup of 6.2 times faster.

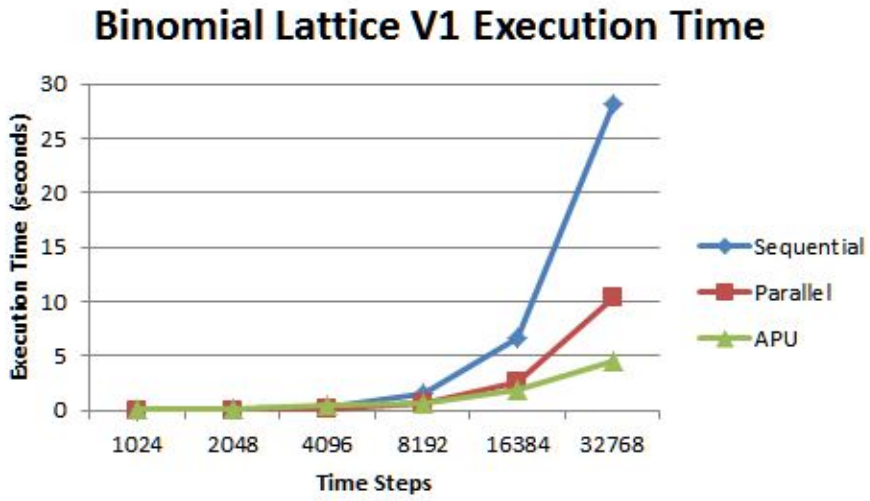


Figure 8.2: Even with a basic, first attempt algorithm, the APU is able to outperform even our parallel CPU algorithm.

Next I profiled the algorithm (see Figure 8.3) to see where it could be improved and, just like the 0-1 knapsack and Gaussian Elimination problems, we are handing control to/from the device after every iteration reducing performance. One good thing to note is that we're not performing any data transfer between iterations which means we are at least not losing much performance. This also means that there is little/no benefit by optimizing memory. Unfortunately as well, the APU's strengths will not be fully utilized with this problem.

Figure A.5 details the hardware counters which show we are using lots of VGPRs which is reducing the number of wavefronts that can be scheduled onto the hardware

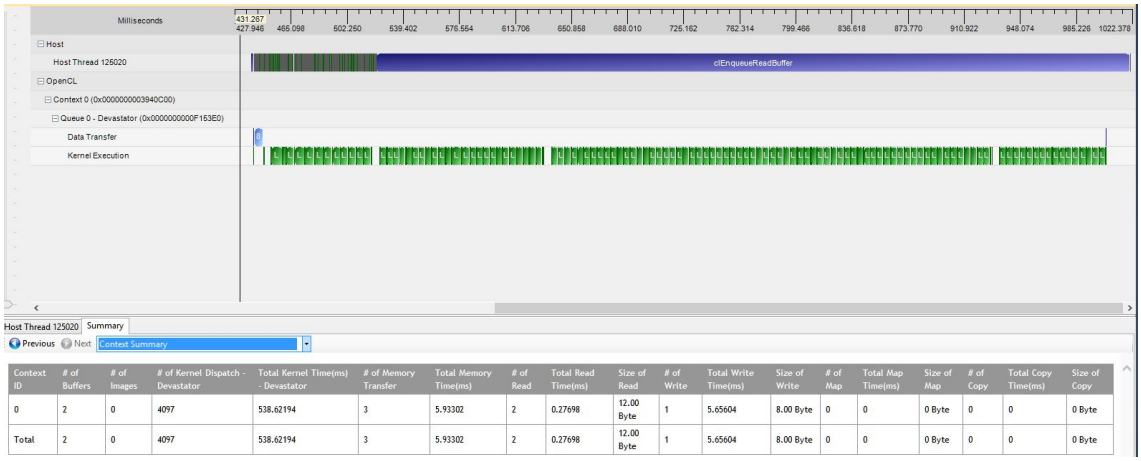


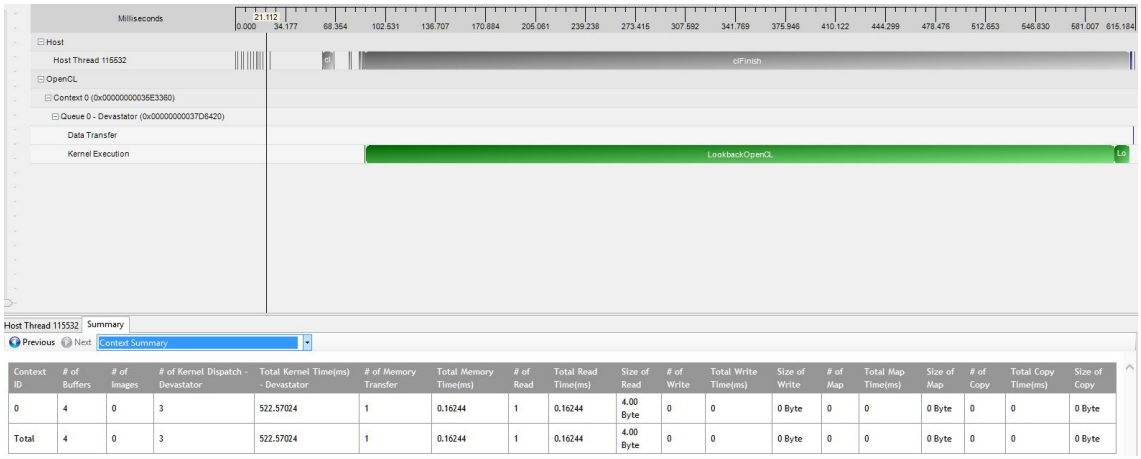
Figure 8.3: The profiling results for the first algorithm which shows control is being handed back to the CPU after each iteration, decreasing performance.

by half. On the upside, we are keeping the hardware relatively busy (40% according to the ALUBusy percentage variable) and removing the overhead of returning to the CPU each iteration was the next optimization undertaken.

8.3.2 Algorithm 2: Loop Unrolling

To begin, I used the basis of the first algorithm and added unrolling of the loops (like we did with the 0-1 knapsack and Monte Carlo algorithms) to avoid passing control back to the CPU which was reducing performance. I also added in local memory to try and reduce the pressure we saw with the VGPRs, similar to that seen in the Monte Carlo algorithm in Chapter 6. To measure this, I again profiled the execution (Figure 8.4) where we can see that no longer is execution broken up (this particular run unrolled all 32,768 iterations, if it was smaller we would see the execution broken up into blocks).

From the hardware counters (see Figure A.6), we can see that using local mem-



Context ID	# of Buffers	# of Images	# of Kernel Dispatch - Devastator	Total Kernel Time(ms) - Devastator	# of Memory Transfer	Total Memory Time(ms)	# of Read	Total Read Time(ms)	Size of Read	# of Write	Total Write Time(ms)	Size of Write	# of Map	Total Map Time(ms)	Size of Map	# of Copy	Total Copy Time(ms)	Size of Copy
0	4	0	3	522.57024	1	0.16244	1	0.16244	4.00 Byte	0	0	0 Byte	0	0	0 Byte	0	0	0 Byte
Total	4	0	3	522.57024	1	0.16244	1	0.16244	4.00 Byte	0	0	0 Byte	0	0	0 Byte	0	0	0 Byte

Figure 8.4: A screenshot of the profiler for the algorithm which has been unrolled for groups of 32,768 iterations to stay on the device before returning control to the CPU. Notice how the LookbackOpenCL kernel isn't split up like it was earlier meaning the hardware is being kept busy for longer.

ory has decreased the pressure on VGPRs, but not enough to increase our kernel occupancy which remains at 50%. So just like in the Monte Carlo algorithm, using local memory shows no real benefit to performance since we are so drastically limited by compute. Looking at the other effects of unrolling though we see that now the ALUBusy percentage has actually decreased by almost 15%, ALUPacking has decreased nearly 50% and we've now run into some (very) minor stalls on both the fetch and write units.

Regardless of these small changes in the hardware counters, looking at Figure 8.5 we achieve a very good speedup, nearly nine times faster than our first GPU algorithm. There is still work to be done however, as seen in the Monte Carlo algorithm, vectorization provided a substantial benefit for performance, though it did come with increased algorithm complexity.

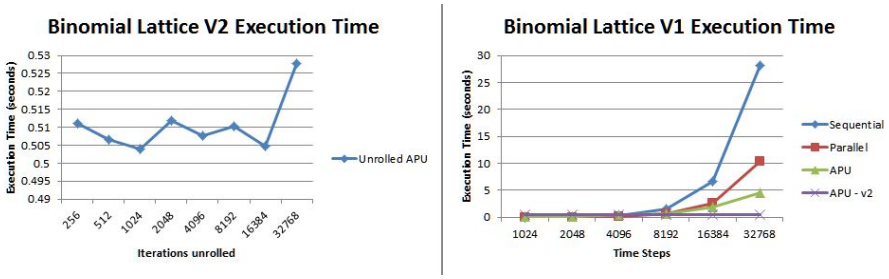


Figure 8.5: The execution time for the unrolled algorithm. Notice how unrolling anywhere from 256-16,384 iterations doesn't seem to change performance but having unrolling drastically increases overall performance compared to the CPU algorithms.

8.3.3 Algorithm 3: Vectorization

The final algorithm uses vectorization to attempt to increase performance. The decision to try vectorization was a result of the performance increase that we saw in the Monte Carlo problem. While very parallel like the Monte Carlo algorithm, the binomial lattice problem has a much more spread-out memory access pattern, and we're removing half of the available work items through each iteration, making the problem much more difficult to optimize. Looking at Figure 8.6, we see that due to the unrolling, the program is split into blocks of execution, but there is no time lost between iterations as the scheduling works out very well. We can also see that there is very little time spent doing memory operations (as Algorithm 2 was the basis for this algorithm) meaning there is little to no performance to be gained through those types of optimizations.

Looking at the execution time of the algorithm, we see it has not changed, even though we have "doubled" our computing power. This is a stark contrast to the Monte Carlo algorithm where performance doubled moving to 2-element vectorization, doubling again for each power of 2, all the way up to 16-element vectorization.

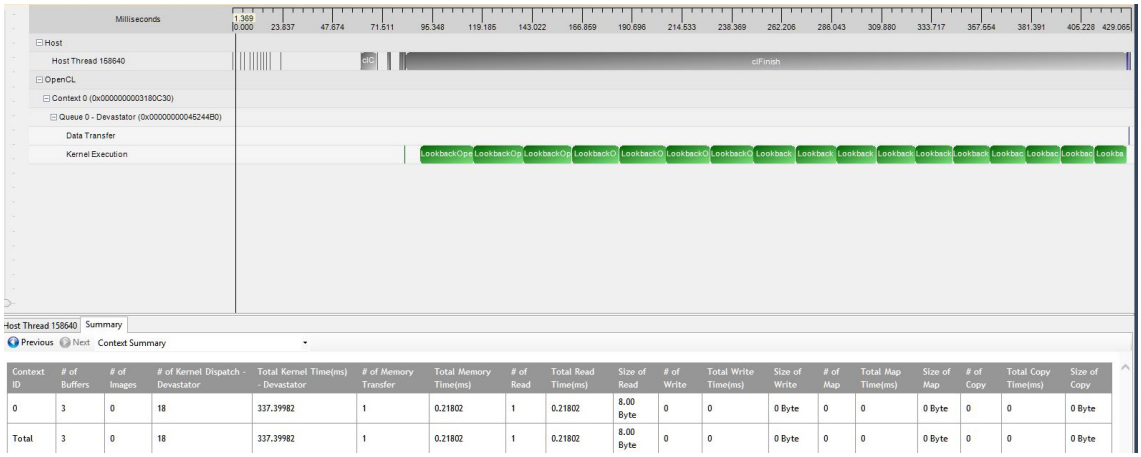


Figure 8.6: A screenshot of the profiler for the algorithm which has been unrolled for groups of 2048 iterations creating back-to-back execution blocks on the device which are scheduled well enough to have minimal time without computations being performed.

This then forces us to ask the question of why we don't see the same behavior, when the algorithm is also very parallel? To find the cause of no performance increase, we turn to the hardware counters collected during execution.

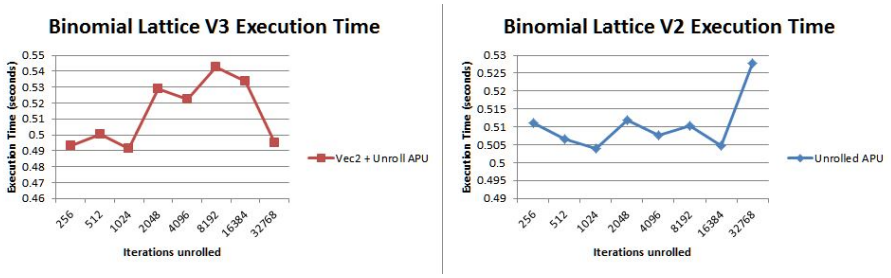


Figure 8.7: The execution time for the unrolled and 2-element vectorized algorithm. It's interesting to see that performance has not increased through the use of vectorization.

Examining the hardware counters of Figure A.7 we see a few points of interest. First is that the VGPRs have not increased dramatically like they did with the Monte

Carlo algorithm, meaning that whatever registers are being used, are being properly re-used (this should in theory help performance). Second is the ALUFetchRatio has actually increased to, on average, three times that of our second algorithm, meaning we are doing more work per memory access. Last we have the FetchUnitBusy percentage and Fetch/WriteUnitStalled percentages, all of which have gotten better. This then returns us to the question of why didn't performance increase?

Looking back to what we said earlier about the algorithm, at each iteration, we are throwing away half of the work that is being done (due to the reduction), which causes very inefficient use of resources. The second contributing factor is the memory accesses themselves. Since we are combining elements into a single variable (with multiple components), when we remove half of them (by moving to the next iteration) we cut the usable memory in half as well, meaning we are no longer calculating x items per compute unit, but instead we're now computing x/2. We also run into the issue of the memory addresses themselves which are now much more spread out and difficult to optimize at the hardware level since we continue to remove work items. It is from these combinations of factors that we don't see any noticeable performance benefit from vectorization in the binomial lattice algorithm compared to the Monte Carlo algorithm.

8.4 Summary

Through optimizing the binomial lattice problem we managed to achieve performance 50 times faster than the sequential algorithm. Useful techniques we saw here involved the use of loop unrolling to keep as many iterations on the device as pos-

sible to reduce the amount of time the device isn't computing. We also looked at vectorization based on the results we obtained in the Monte Carlo algorithm. Unfortunately, this time we did not achieve any measurable increase in performance due to the removal of half of the work items every iteration and with less than optimal use of the vectorized data elements (half of which were "discarded" every iteration).

8.5 Future Work

Given what we know now about the difficulties on optimizing the binomial lattice problem there are a few other areas I would like to study. First is the use of AMD's next generation APUs which should provide better memory management and handling of the removal of work items at each iteration with heterogeneous queueing. Second would be to deeply optimize the algorithm to try and reduce the number of VPGRs which resulted in being able to schedule only half of the wavefronts on the hardware during each execution. Lastly, I would like to look into the use of GCN GPUs to see if their architecture, with their hardware-based scheduling, would better handle the situation where items are being removed during execution.

Chapter 9

Discussion of Technologies

Like we talked about in Chapter 3, AMD has completed phases one and two of their heterogeneous systems architecture with Llano and Trinity. Now, their third APU, Kaveri, unifies the memory space of the CPU and GPU, making memory passing obsolete. This is done through the use of pointers which are passed between devices instead of copying memory and eliminates the overhead relating to retrieving data from off the chip.

The Kaveri (and future Carrizo) APUs should also be extremely powerful for problems in which the workload changes at runtime, such as Gaussian Elimination. Reusing the hardware in this manner enables the APU to also take advantage of the hardware level cache which the GPU is now free to access (with coherency being enforced). A second benefit of the combined same address space is that the GPU now "has" the same amount of memory as the host machine (since pages can be swapped in automatically), making very large problem sizes now workable on the APU (Kyriazis [2012]), just as they would be on a CPU.

This is only AMD's third step in their plan of bringing the APU to the mainstream consumer as their final APU, Carrizo, is scheduled to be released during 2015. This advanced APU will be the "final product" that AMD started developing almost a decade ago and extends the capabilities to include:

- Context switching between applications accessing the GPU
- Pre-emption of GPU processes to enable low latency for time sensitive programs
- Quality of Service via enabling resource and user prioritization
- Queueing from any HSA device within an application to any other HSA device's queue

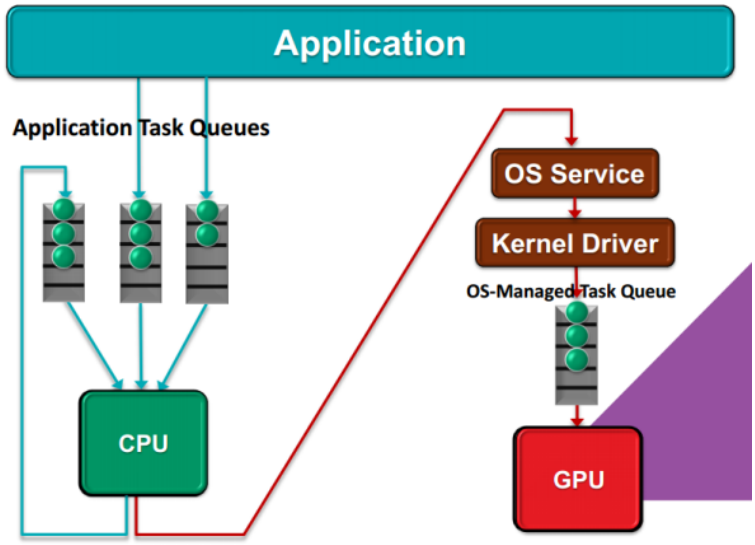


Figure 9.1: The current queueing model which uses the CPU to control all other devices which requires OS intervention to move data and schedule instructions. Image from Hruska [2013].

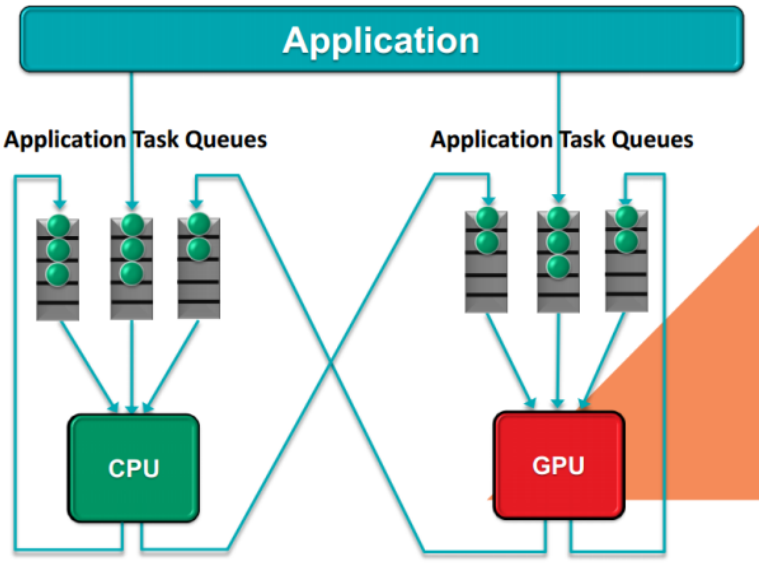


Figure 9.2: The future queueing model which moves all operations into user space. This removes the OS, reducing latency and enables any HSA device to enqueue to any other HSA device's queue. Image taken from Hruska [2013].

These advanced hardware-level capabilities will make programming the APU easier than ever before and will remove many of the bottlenecks that exist in current implementations. If you look back to the chapter on Gaussian Elimination, our performance was limited by OS interaction as it was required to do the queueing/mapping of device buffers (or the GPU's sequential thread performance for the final algorithm). Now, programmers will be able to compute with the best device present in a system without having to spend as much time examining the hardware in an attempt to find at which input size each device becomes more efficient; they can simply choose the best device for a given period in time since context switching and no data transfer is now possible. What this means is that for difficult programs such as Gaussian Elimination, we could easily switch to the GPU for just a single iteration and then transition back to the CPU since there is now minimal overhead.

Other exciting advancements are Mantle and AMD's clean sweep of the consoles, which are related to a degree. First, Mantle is an API, similar to DirectX or OpenGL, but lower level, and optimized for AMD's GCN architectures. The reason for Mantle was to remove the overhead that has accumulated throughout the years in both DirectX and OpenGL enabling better performance by reducing the dependency on the CPU. This enables game developers to push the hardware to new limits, achieving performance not before seen AMD [2013b,a]. First results were released in February of 2014 and Mantle shows a performance advantage averaging 25% over DirectX 11 with a game engine built to support it Kean [2014]. Other results from the popular video game Battlefield 4 show promising performance improvements of over 10%, which is impressive given Mantle is a brand new technology and was added to the game as an afterthought. Not everyone though is happy about a new player in this market.

Some might say another API in the graphics field is doomed to fail, just like Glide (which ran only on 3dfx hardware in the 1990s), but they are missing a few key things. One is that Mantle is not a proprietary API like Glide was, meaning anyone can use it in their software and/or hardware Schiesser [2013]. A second drawback commonly mentioned is that this will enforce the need for another code branch to support this API. While technically true, most games now are first developed on and for consoles which don't support the same branch of DirectX and OpenGL as PCs do (meaning developers are already splitting the code base into console and PC). This leads into the next advantage that AMD has, consoles.

AMD produced a clean sweep of the next generation consoles, the PlayStation 4,

XBox One and Wii U. These consoles use an AMD CPU and GPU (aside from the Wii U which uses an IBM CPU), glued together in an almost APU-like fashion. What this means is that the major gaming consoles for approximately the next ten years will be based around APUs (given the hardware life of the current PS3 and 360). Returning to my previous point on Mantle and why it should succeed, developers currently have to create three different versions of a game, one for the PS3, 360 and PC as they all use different hardware. Now, because AMD has the console market, developers can have a single code branch for both consoles and AMD PCs (with minor changes to support features not provided by the PS4 or XBox One and for PCs using GPUs from other manufacturers) which is much easier than the previous generation because the hardware was different. Looking at this from a developers perspective, they now can now re-use most of their code between consoles (reducing development time and effort), and, if the hardware is the same on the PC, receive the same performance benefit that the consoles receive (because of the low level access the API provides).

Chapter 10

Conclusion and Future Work

In this thesis I studied the Llano APU architecture in depth and studied four problems with different characteristics. For each of these problems I provided multiple variants and implementations in OpenCL and studied the performance of the programs in detail using the AMD's OpenCL profiler.

For the 0-1 knapsack problem, I provide four variants or implementations. The first algorithm indicates the bottleneck of the memory transfer over the PCI Express bus and the context switching that occurs when transferring control from the CPU and the GPU and vice versa. Using the profiler, I computed the number of reads and writes and the percentage of communication and computation. Using this profiling information, I was able to perform guided optimizations (examining both software and hardware counters) to iteratively improve the execution time of my algorithms.

To remove the PCI Express bottleneck, I used loop unrolling in the second variant. This modification required some amount of synchronization to avoid race conditions but resulted in significantly improved performance with the execution time being half

of the sequential algorithm.

The third variant used host memory and scaled across all available compute cores (only a single compute core was used in the first and second variants). Using the profiler, I realized the hardware was now less utilized, 27% for the ALU, 22% for the FetchUnit and the write unit became stalled. This is the result of using more compute units which resulted in more synchronization being required (between compute units), but also enables us to scale the algorithm to larger input sizes.

The final change I implemented was the use of local memory in the kernel to cache data elements within a workgroup and reduce the overall number of accesses to global memory. This variant doubled the ALUBusy time, decreasing execution time from 0.06 seconds for for a 4096x4096 matrix to 0.04 seconds. For a larger 8192x8192 matrix, the fourth algorithm was 61 times faster than the sequential algorithm.

I provided three variant implementations of the Gaussian Elimination algorithm. In the first variant, profiling information showed that only 4% of the total run time was spent executing the kernel, and the remainder was spent waiting on memory operations. This can be seen in an ALU/Fetch ratio of only 3.5 and the amount of time the hardware is busy running calculations is only 3%.

In the second variant, to remove the limitation of memory and play the strengths of the APU, I used host memory in place of device buffers. The performance improvement was only 1% without synchronization and lagged behind the sequential algorithm by over 10%. The profiler output indicated the significant performance decrease was due to the interaction of the OS when context switching between the CPU and GPU. On the Llano APU, this interaction can not be avoided, but the removal

of OS involvement is a planned feature in the future Carrizo APU.

The third and final variant removed the interaction of the OS and used only the GPU cores. There was a great improvement in terms of execution time for a certain number of data points, but beyond that, synchronization of threads on the GPU took effect and brought down performance.

Using the profiler, I found that the Monte Carlo algorithm spends very little time on memory operations but was limited by the number of registers used to store vectors, reducing performance. This result in reduced ALUBusy and ALUPacking percentages because we ran into hardware limitations.

To reduce register usage in the second variant, the data was stored in local memory. Unfortunately, since local memory is slower than registers, this did not have a significant benefit to overall performance (and did not reduce register pressure in the algorithm).

The third variant made use of global memory and vectorization. Although packing several elements into a single variable decreased execution time, it also decreased the number of waves that the hardware can schedule simultaneously. This resulted in a utilization of only 12.5% of the device due to register pressure.

For the Binomial Lattice algrorithm, the first variant indicated that for large time steps we were able to see significant speedups. Since there was minimal data transfer between iterations, memory optimization was not a necessity. The profiler also indicated the the ALUBusy percentage was about 40%, but a large number of VGPRs were used.

The second variant used loop unrolling to avoid passing control back to the CPU,

which was reducing performance in the first variant. To decrease register pressure I also chose to use local memory. However, after optimizations, our kernel occupancy was only 50% and the ALUBusy percentage decreased by almost 15% and ALUPacking decreased nearly 50% while local memory had relatively no benefit.

The final variant used vectorization to use the capabilities of the APU's VLIW architecture. However, this technique had no tangible benefit to overall performance as the tree data structure significantly reduces the amount of work at each step, causing an irregular workload. In vectorization, we combine data elements into a single variable. In the Binomial Lattice algorithm, this causes irregular memory accesses because the data themselves are spread out and global coalescing becomes difficult.

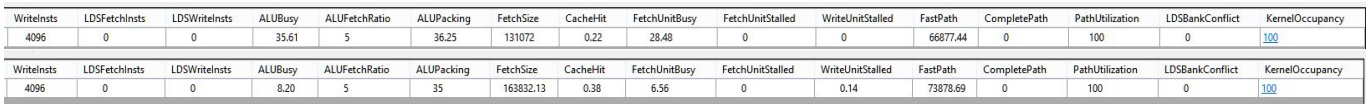
In summary, compared to a traditional GPU the APU is of course much weaker as it must share die space and thermal design power with a CPU. However, the APU allows us to handle memory-bound problems due to its on-chip memory. There are some weaknesses due to hardware limitations, but those are beyond our control (although many will be rectified with the Carrizo APU).

My conclusion is that it requires a lot of effort in developing efficient algorithms on these new heterogeneous architectures, even for these simple problems. The programmer requires a base understanding of the hardware in order to map the available hardware resources to the algorithm. Profiling is also very important to understand the performance of the program and provide hints for optimization to the programmer. Current versions of the APU schedule statically, but future architectures will provide more hardware scheduling, taking some of that burden away from the programmer.

Future work I would like to do includes examining both the Kaveri and Carrizo APUs (with an emphasis on Carrizo) for difficult problems such as Gaussian Elimination. Looking back to our results, we saw that much of the time was spent either waiting for buffers to transfer or waiting for OS interaction. With Carrizo, this time should be almost entirely eliminated due to not passing any memory (shared address space) and not having to leave user space to map memory (heterogeneous queueing). With these advantages, problems that change drastically during runtime (in terms of input size, connections or otherwise) should be much easier to implement and achieve a speedup using these next generation APUs. One other point of interest to return to would be to examine the effect of core and memory speed of the APU to see how they impact performance. As the APU relies heavily on system memory, it would be interesting to see how using faster memory would help increase performance, particularly for difficult problems that exhibit irregular workload.

Appendix A

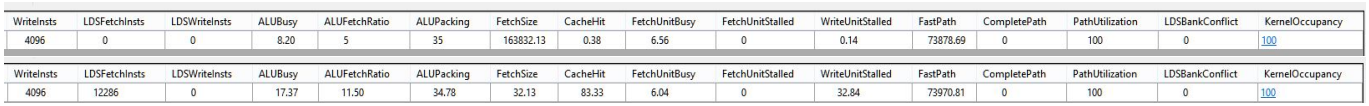
Supporting Data



WriteInsts	LDSFetchInsts	LDSWriteInsts	ALUBusy	ALUFetchRatio	ALUPacking	FetchSize	CacheHit	FetchUnitBusy	FetchUnitStalled	WriteUnitStalled	FastPath	CompletePath	PathUtilization	LDSBankConflict	KernelOccupancy
4096	0	0	35.61	5	36.25	131072	0.22	28.48	0	0	66877.44	0	100	0	100

WriteInsts	LDSFetchInsts	LDSWriteInsts	ALUBusy	ALUFetchRatio	ALUPacking	FetchSize	CacheHit	FetchUnitBusy	FetchUnitStalled	WriteUnitStalled	FastPath	CompletePath	PathUtilization	LDSBankConflict	KernelOccupancy
4096	0	0	8.20	5	35	163832.13	0.38	6.56	0	0.14	73878.69	0	100	0	100

Figure A.1: GPU performance counters for 0-1 knapsack algorithms v2 (top) and v3 (bottom). Upon closer examination we see that the hardware utilization counters have decreased going from v2 to v3.



WriteInsts	LDSFetchInsts	LDSWriteInsts	ALUBusy	ALUFetchRatio	ALUPacking	FetchSize	CacheHit	FetchUnitBusy	FetchUnitStalled	WriteUnitStalled	FastPath	CompletePath	PathUtilization	LDSBankConflict	KernelOccupancy
4096	0	0	8.20	5	35	163832.13	0.38	6.56	0	0.14	73878.69	0	100	0	100

WriteInsts	LDSFetchInsts	LDSWriteInsts	ALUBusy	ALUFetchRatio	ALUPacking	FetchSize	CacheHit	FetchUnitBusy	FetchUnitStalled	WriteUnitStalled	FastPath	CompletePath	PathUtilization	LDSBankConflict	KernelOccupancy
4096	12286	0	17.37	11.50	34.78	32.13	83.33	6.04	0	32.84	73970.81	0	100	0	100

Figure A.2: GPU performance counters for knapsack algorithms v3 (top) and v4 (bottom) which shows improved an improved ALUBusy time with the usage of local memory.

Method	ExecutionOrder	ThreadID	CallIndex	GlobalWorkSize	WorkGroupSize	Time	LocalMemSize	VGPRs	SGPRs	ScratchRegs	FCStacks	KernelOccupancy	Wavefronts	ALUInsts
reverse_elimination__k1_Devastator1	1	61516	19	{ 1792 1 1}	{ 256 1 1}	0.01692	0	3	NA	0	1	100	28	7
reverse_elimination__k1_Devastator1	2	61516	34	{ 1792 1 1}	{ 256 1 1}	0.01072	0	3	NA	0	1	100	28	7
reverse_elimination__k1_Devastator1	3	61516	49	{ 1792 1 1}	{ 256 1 1}	0.00580	0	3	NA	0	1	100	28	7
reverse_elimination__k1_Devastator1	4	61516	64	{ 1792 1 1}	{ 256 1 1}	0.01040	0	3	NA	0	1	100	28	7
reverse_elimination__k1_Devastator1	5	61516	79	{ 1792 1 1}	{ 256 1 1}	0.00534	0	3	NA	0	1	100	28	7
reverse_elimination__k1_Devastator1	6	61516	94	{ 1792 1 1}	{ 256 1 1}	0.01024	0	3	NA	0	1	100	28	7
reverse_elimination__k1_Devastator1	7	61516	109	{ 1792 1 1}	{ 256 1 1}	0.01044	0	3	NA	0	1	100	28	7

ALUInsts	FetchInsts	WriteInsts	LDSFetchInsts	LDSWriteInsts	ALUBusy	ALUFetchRatio	ALUPacking	FetchSize	CacheHit	FetchUnitBusy	FetchUnitStalled	WriteUnitStalled	FastPath	CompletePath	PathUtilization	LDSBankConflict
7	2	1	0	0	2.90	3.50	42.86	14	0	3.32	0	0	6.75	0	100	0
7	2	1	0	0	3.24	3.50	42.86	14	0	3.71	0	0	6.75	0	100	0
7	2	1	0	0	3.24	3.50	42.86	14	0	3.70	0	0	6.75	0	100	0
7	2	1	0	0	3.24	3.50	42.86	14	0	3.71	0	0	6.75	0	100	0
7	2	1	0	0	2.49	3.50	42.86	14	0	2.84	0	0	6.75	0	100	0
7	2	1	0	0	3.17	3.50	42.86	14	0	4.09	2.84	0	6.75	0	100	0
7	2	1	0	0	3.24	3.50	42.86	14	0	3.71	0	0	6.75	0	100	0

Figure A.3: GPU performance counters for the naïve Gaussian Elimination algorithm.

Method	ExecutionOrder	ThreadID	CallIndex	GlobalWorkSize	WorkGroupSize	Time	LocalMemSize	VGPRs	SGPRs	ScratchRegs	FCStacks	KernelOccupancy	Wavefronts	ALUInsts	FetchInsts	WriteInsts	LDSFetchInsts
MonteCarlo__k1_Devastator1	1	64936	34	{ 32768 1 1}	{ 256 1 1}	707.80992	0	9	NA	0	3	87.50	512	393265	3	4	0

WriteInsts	LDSFetchInsts	LDSWriteInsts	ALUBusy	ALUFetchRatio	ALUPacking	FetchSize	CacheHit	FetchUnitBusy	FetchUnitStalled	WriteUnitStalled	FastPath	CompletePath	PathUtilization	LDSBankConflict
4	0	0	24.04	131088.33	41.67	512.06	0.06	0	0	0	1048.69	0	100	0

Figure A.4: GPU performance counters for Monte Carlo algorithm v1.

Method	ExecutionOrder	ThreadID	CallIndex	GlobalWorkSize	WorkGroupSize	Time	LocalMemSize	VGPRs	SGPRs	ScratchRegs	FCStacks	KernelOccupancy	Wavefronts	ALUInsts	FetchInsts	WriteInsts	LDSFetchInsts
InitializeValues__k1_Devastator1	1	126604	35	{ 4096 1 1}	{ 256 1 1}	0.02912	0	10	NA	0	2	75	64	74.02	3	1.02	0
LookbackOpenCL__k2_Devastator1	2	126604	50	{ 4096 1 1}	{ 256 1 1}	0.03380	0	14	NA	0	2	50	64	87.11	6.03	2.02	0
LookbackOpenCL__k2_Devastator1	3	126604	53	{ 4096 1 1}	{ 256 1 1}	0.01200	0	14	NA	0	2	50	64	87.11	6.03	2.02	0
LookbackOpenCL__k2_Devastator1	4	126604	56	{ 4096 1 1}	{ 256 1 1}	0.02784	0	14	NA	0	2	50	64	87.11	6.03	2.02	0
LookbackOpenCL__k2_Devastator1	5	126604	59	{ 4096 1 1}	{ 256 1 1}	0.02748	0	14	NA	0	2	50	64	87.11	6.03	2.02	0
LookbackOpenCL__k2_Devastator1	6	126604	62	{ 4096 1 1}	{ 256 1 1}	0.02828	0	14	NA	0	2	50	64	87.11	6.03	2.02	0
LookbackOpenCL__k2_Devastator1	7	126604	65	{ 4096 1 1}	{ 256 1 1}	0.02764	0	14	NA	0	2	50	64	87.11	6.03	2.02	0

LDSFetchInsts	LDSWriteInsts	ALUBusy	ALUFetchRatio	ALUPacking	FetchSize	CacheHit	FetchUnitBusy	FetchUnitStalled	WriteUnitStalled	FastPath	CompletePath	PathUtilization	LDSBankConflict
0	0	40.38	24.67	71.61	0.63	83.33	6.55	0	0	15.75	0	100	0
0	0	40.26	14.44	66.05	32.69	13.27	11.11	0	0	32.13	0	100	0
0	0	40.68	14.44	66.05	32.63	13.29	11.22	0	0	32.25	0	100	0
0	0	41.08	14.44	66.05	32.63	13.29	11.33	0	0	32.06	0	100	0
0	0	40.68	14.44	66.05	32.63	13.29	11.22	0	0	32.13	0	100	0
0	0	41.09	14.44	66.05	32.63	13.29	11.33	0	0	32.13	0	100	0
0	0	41.08	14.44	66.05	32.63	13.29	11.33	0	0	32.25	0	100	0

Figure A.5: GPU performance counters for the naïve Binomial Lattice algorithm.

Method	ExecutionOrder	ThreadID	CallIndex	GlobalWorkSize	WorkGroupSize	Time	LocalMemSize	VGPRs	SGPRs	ScratchRegs	FCStacks	KernelOccupancy	Wavefronts	ALUInsts	FetchInsts	WriteInsts	LDSFetchInsts
InitializeValues__k1_Devastator1	1	98084	37	{ 32768 1 1}	{ 256 1 1}	0.03894	0	9	NA	0	2	87.50	512	74	3	1	0
LookbackOpenCL__k2_Devastator1	2	98084	52	{ 32768 1 1}	{ 256 1 1}	509.35962	8	13	NA	0	3	50	512	320384.97	32997	16481	0

LDSFetchInsts	LDSWriteInsts	ALUBusy	ALUFetchRatio	ALUPacking	FetchSize	CacheHit	FetchUnitBusy	FetchUnitStalled	WriteUnitStalled	FastPath	CompletePath	PathUtilization	LDSBankConflict
0	0	85.87	24.67	71.96	0.63	83.33	13.92	0	0	127.75	0	100	0
0	0	26.66	9.71	34	5213624.50	0	13.30	2.91	0.25	2163390.75	0	100	0

Figure A.6: GPU performance counters for the unrolled Binomial Lattice algorithm.

Method	ExecutionOrder	ThreadID	CallIndex	GlobalWorkSize	WorkGroupSize	Time	LocalMemSize	VGPRs	SGPRs	ScratchRegs	FCStacks	KernelOccupancy	Wavefronts	ALUInsts	FetchInsts	WriteInsts	LDSFetchInsts
InitializeValues__k1_Devastator1	1	156764	36	{ 2048 1 1}	{ 256 1 1}	0.01522	0	15	NA	0	2	50	32	111.97	4.06	1.03	0
LookbackOpenCL__k2_Devastator1	2	156764	51	{ 2304 1 1}	{ 256 1 1}	25.26870	8	14	NA	0	3	50	36	191755.42	7915.11	1935.17	0
LookbackOpenCL__k2_Devastator1	3	156764	56	{ 2048 1 1}	{ 256 1 1}	24.30354	8	14	NA	0	3	50	32	191597.47	7867.75	1921	0
LookbackOpenCL__k2_Devastator1	4	156764	61	{ 2048 1 1}	{ 256 1 1}	23.73920	8	14	NA	0	3	50	32	188184.59	6895.50	1680.94	0
LookbackOpenCL__k2_Devastator1	5	156764	66	{ 1792 1 1}	{ 256 1 1}	23.17322	8	14	NA	0	3	50	28	188005.96	6842	1665	0
LookbackOpenCL__k2_Devastator1	6	156764	71	{ 1792 1 1}	{ 256 1 1}	23.10816	8	14	NA	0	3	50	28	184617.54	5877.14	1427.21	0
LookbackOpenCL__k2_Devastator1	7	156764	76	{ 1536 1 1}	{ 256 1 1}	21.83462	8	14	NA	0	3	50	24	184411.96	5815.67	1409	0

LDSFetchInsts	LDSWriteInsts	ALUBusy	ALUFetchRatio	ALUPacking	FetchSize	CacheHit	FetchUnitBusy	FetchUnitStalled	WriteUnitStalled	FastPath	CompletePath	PathUtilization	LDSBankConflict
0	0	24.45	27.56	75.33	0.63	83.33	3.51	0	0	17.50	0	100	0
0	0	23.35	24.23	30.23	66724.81	0	3.73	0.14	0	32923.75	0	100	0
0	0	21.55	24.35	30.22	58264.56	0.01	3.39	0.07	0	28910.94	0	100	0
0	0	21.61	27.29	29.93	50380.31	0.01	3.02	0.02	0	25128.75	0	100	0
0	0	19.51	27.48	29.91	43072.06	0.01	2.69	0	0	21613.63	0	100	0
0	0	19.72	31.41	29.61	36339.81	0.01	2.37	0	0	18370.25	0	100	0

Figure A.7: GPU performance counters for the vectorized Binomial Lattice algorithm.

Bibliography

L. A. Abbas-Turki and B. Lapeyre. American options pricing on multi-core graphic cards, 2009.

AMD. AMD Completes ATI Acquisition and Creates Processing Powerhouse, October 2006. http://www.amd.com/us/press-releases/Pages/Press_Release_113741.aspx.

AMD. AMD Demonstrates Worlds First X86 Dual-Core Processor. http://www.amd.com/us/press-releases/Pages/Press_Release_89872.aspx, August 2004.

AMD. BF4 + FROSTBITE + MANTLE, November 2013a. http://www.youtube.com/watch?v=KApdf4P2Iak#t=14.

AMD. Oxide Games AMD Mantle Presentation and Demo, December 2013b. http://www.youtube.com/watch?v=QIWyf8Hyjbg.

AMD. Heterogeneous Computing OpenCL™and the ATI Radeon™ HD 5870 ("Evergreen") Architecture, March 2010. http://developer.amd.com/gpu_assets/Heterogeneous_Computing_OpenCL_and_the_ATI_Radeon_HD_5870_Architecture_201003.pdf.

AMD. AMD Accelerated Parallel Processing OpenCL Programming Guide, July 2012. http://developer.amd.com/wordpress/media/2012/10/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf.

G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference (AFIPS)*, AFIPS '67 (Spring), pages 483–485, 1967.

I. T. Association. Infiniband roadmap. http://www.infinibandta.org/content/pages.php?pg=technology_overview, unknown 2013.

A. Boukedjar, M. Lalami, and D. El-Baz. Parallel Branch and Bound on a CPU-GPU System. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pages 392–398, 2012.

V. Boyer, D. E. Baz, and M. Elkihel. Solving knapsack problems on GPU. *Computers & Operations Research*, 39(1):42 – 47, 2012.

N. Brookwood. AMD Fusion™Family of APUs: Enabling a Superior, Immersive PC Experience, March 2010. http://www.amd.com/us/documents/48423_fusion_whitepaper_web.pdf.

S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach. Accelerating Compute-Intensive Applications with GPUs and FPGAs. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, June 2008.

P. Clarke. HSA close to setting hardware specs, March 2013. http://www.eetimes.com/document.asp?doc_id=1280621.

C. Demerjian. A llook at the Llano architecture, June 2011. http://semiaccurate.com/2011/06/20/a-llook-at-the-llano-architecture/.

C. Demerjian. Is AMD's Trinity much better than it appears?, May 2012. http://semiaccurate.com/2012/05/17/is-amds-trinity-much-better-than-it-appears/.

J. W. Demmel, J. R. Gilbert, and X. S. Li. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, 1999.

E. Dockner and H. Moritsch. Pricing Constant Maturity Floaters with Embedded Options Using Monte Carlo Simulation. 1999.

M. Doerksen, S. Solomon, and P. Thulasiraman. Designing APU Oriented Scientific Computing Applications in OpenCL. In *International Symposium on Advances of High Performance Computing and Networking in Conjunction with HPCC 2011, Banff, Alberta, Canada, Sept. 2-4, 2011*, pages 587–592, September 2011.

M. Doerksen, S. Solomon, P. Thulasiraman, and R. Thulasiram. Financial Option Pricing on APU. In *Contemporary Computing*, volume 306, pages 431–441. Springer Berlin Heidelberg, 2012a. URL http://dx.doi.org/10.1007/978-3-642-32129-0_43.

M. Doerksen, P. Thulasiraman, and R. Thulasiram. In *Optimizing Option Pricing Algorithms and Profiling Power Consumption on VLIW APU Architecture, The 10th*

International Symposium on Parallel and Distributed Processing with Applications, Madrid, Spain, July 10-13, 2012, pages 71–78, July 2012b.

J. Fung and S. Mann. OpenVIDIA: Parallel GPU Computer Vision. In *Proceedings of the 13th Annual ACM International Conference on Multimedia*, MULTIMEDIA '05, pages 849–852, 2005. URL `http://doi.acm.org/10.1145/1101149.1101334`.

B. R. Gaster and L. Howes. OpenCL™- Parallel computing for CPUs and GPUs, July 2010. `http://developer.amd.com/wordpress/media/2012/10/OpenCL_Parallel_Computing_for_CPUs_and_GPUs_201003.pdf`.

I. Gohberg, T. Kailath, and V. Olshevsky. Fast Gaussian elimination with partial pivoting for matrices with displacement structure. *Mathematics of computation*, 64 (212):1557–1576, 1995.

M. Gschwind. Chip Multiprocessing and the Cell Broadband Engine. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 1–8. ACM, 2006.

J. Hruska. How AMD's HSA Queuing Technology Simplifies GPU Acceleration, October 2013. `http://hothardware.com/News/How-AMDs-HSA-Queuing-Technology-Simplifies-GPU-Acceleration-/`.

Intel. Intel Core2 Extreme Processor QX9775 (12M Cache, 3.20 ghz, 1600 MHz FSB). `http://ark.intel.com/products/34692/Intel-Core2-Extreme-Processor-QX9775-12M-Cache-3_20-GHz-1600-MHz-FSB/`, unknown 2008.

Intel. Intel Core i7-4960X Processor Extreme Edition (15M Cache, up to 4.00

GHz, unknown 2013. http://ark.intel.com/products/77779/Intel-Core-i7-4960X-Processor-Extreme-Edition-15M-Cache-up-to-4_00-GHz.

D. Kanter. AMD Fusion Architecture and Llano, June 2011. http://www.realworldtech.com/fusion-llano/.

S. Kean. AMD Mantle API Real World BF4 Benchmark Performance On Catalyst 14.1, February 2014. http://www.legitreviews.com/amd-mantle-api-real-world-bf4-benchmark-performance-catalyst-141_134959/4.

G. Kyriazis. Heterogeneous System Architecture: A Technical Review, August 2012. http://developer.amd.com/wordpress/media/2012/10/hsa10.pdf.

A. Lal Shimpi. AMD's 2010 - 2011 Roadmaps: 1B Transistor Llano APU, Bobcat and Bulldozer, November 2009a. http://www.anandtech.com/show/2871.

A. Lal Shimpi. The Real Conroe Successor: Clarkdale & All You Need to Know about Westmere, September 2009b. http://www.anandtech.com/show/2846.

A. Lal Shimpi. The AMD A8-3850 Review: Llano on the Desktop, June 2011a. http://www.anandtech.com/show/4476/amd-a83850-review/1.

A. Lal Shimpi. AMD Core Counts and Bulldozer: Preparing for an APU World, November 2009c. http://www.anandtech.com/show/2881.

A. Lal Shimpi. Intel's Core i7 870 & i5 750, Lynnfield: Harder, Better, Faster Stronger, September 2009d. http://www.anandtech.com/show/2832/1.

A. Lal Shimpi. The Sandy Bridge Preview, August 2010. http://www.anandtech.com/show/3871/the-sandy-bridge-preview-three-wins-in-a-row/1.

A. Lal Shimpi. Intel and Micron Develop Hybrid Memory Cube, Stacked DRAM is Coming, September 2011b. http://www.anandtech.com/show/4819/intel-and-micron-develop-hybrid-memory-cube-stacked-dram-is-coming.

M. Lalami and D. El-Baz. GPU Implementation of the Branch and Bound Method for Knapsack Problems. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1769–1777, 2012.

V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *International Symposium of Computer Architecture*, pages 451–460, Saint-Malo, France, June 19-23 2010.

A. Lokhmotov. Mobile and embedded computing on Mali™ GPUs, December 2010. http://www.many-core.group.cam.ac.uk/ukgpucc2/talks/Lokhmotov.pdf.

M. Magee. Intel, AMD battle it out over APUs, April 2013. http://www.tgdaily.com/hardware-brief/70793-intel-amd-battle-it-out-over-apus.

Nvidia. What is CUDA. https://developer.nvidia.com/what-cuda, unknown 2006.

PCI-SIG. PCI Express®4.0 Frequently Asked Questions, January 2014. http://www.pcisig.com/news_room/faqs/FAQ_PCI_Express_4.0/.

PCI-SIG. PCI Express®3.0 Frequently Asked Questions, November 2010. http://www.pcisig.com/news_room/faqs/pcie3.0_faq/#EQ3.

D. Sanchez-Roman, V. Moreno, S. Lopez-Buedo, G. Sutter, I. Gonzalez, F. J. Gomez-Arribas, and J. Aracil. FPGA acceleration using high-level languages of a Monte-Carlo method for pricing complex options. *Journal of Systems Architecture*, 59(3): 135 – 143, 2013.

T. Schiesser. AMD unveils revolutionary 'Mantle' API to optimize GPU performance, September 2013. `http://www.techspot.com/news/54134-amd-unveils-revolutionary-mantle-api-to-optimize-gpu-performance.html`.

G. Shvets. Intel 8087 family, unknown unknowna. `http://www.cpu-world.com/CPUs/8087/`.

G. Shvets. AMD Athlon 64 X2 3800+ - ADA3800DAA5CD (ADA3800CDBOX). `http://cdn.cpu-world.com/CPUs/K8/S_AMD-ADA3800DAA5CD.jpg`, unknown unknownb.

M. Smith. NVIDIA'S TEGRA K1 MIGHT NOT MAKE SENSE FOR MOBILE, BUT WHAT ABOUT PCS?, January 2014. `http://www.digitaltrends.com/computing/tegra-k1-might-make-sense-mobile-pcs/#!zC8Xr`.

R. Smith and G. T. S. AMD Radeon HD 7750 & Radeon HD 7770 GHz Edition Review: Evading the Price/Performance Curve, February 2012. `http://www.anandtech.com/show/5541/amd-radeon-hd-7750-radeon-hd-7770-ghz-edition-review/21`.

S. Solomon, R. K. Thulasiram, and P. Thulasiraman. Option Pricing on the GPU.

In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pages 289–296, 2010.

B. Spiers and D. Wallez. High-performance computing on wall street. *Computer*, 43 (12):53–59, 2010.

S. Steele. ARM® GPUs Now and in the Future, June 2011. `http://www.arm.com/files/event/8_Steve_Steele_ARM_GPUs_Now_and_in_the_Future.pdf`.

A. H. Tse, D. B. Thomas, and W. Luk. Accelerating Quadrature Methods for Option Valuation. In *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*, pages 29–36, 2009.

W. Verduzco. Nvidia Unveils Kepler-Based Tegra K1 with 192 CUDA Cores and Optional 64-Bit Denver Processor!, January 2014. `http://www.xda-developers.com/android/nvidia-unveils-kepler-based-tegra-k1-with-192-cuda-cores-and-optional-64-bit-denver-processor/`.

R. Williams. Intel's Core i7-980X Extreme Edition - Ready for Sick Scores. `http://techgage.com/article/intels_core_i7-980x_extreme_edition_-_ready_for_sick_scores/8/`, March 2010.

D. Wollmann. AMD's Fusion A-Series chips official, June 2011. `http://www.engadget.com/2011/06/13/amds-fusion-a-series-for-mainstream-laptops-official-10-5-hour/`.

J. Zheng, L. Wang, R. Quhe, Q. Liu, H. Li, D. Yu, W.-N. Mei, J. Shi, Z. Gao, and

J. Lu. Sub-10 nm Gate Length Graphene Transistors: Operating at Terahertz Frequencies with Current Saturation. *Scientific reports*, 3, 2013.