EXPLORATION INTO THE PERFORMANCE OF ASYMMETRIC D-ARY HEAP-BASED ALGORITHMS FOR THE HSA ARCHITECTURE

A Thesis
presented in partial fulfillment of requirements
for the degree of Master's of Science
in the Department of Computer and Information Science
The University of Mississippi

by

Stephen Blake Adams

May 2014

ABSTRACT

Heterogeneous computing is a fairly recent trend in both hardware and software design; based around identifying the opportunities presented by utilizing all available hardware components in a computing system to perform a computationally intensive task in the most efficient way possible. One incredibly interesting field of the heterogeneous computing paradigm is general purpose computing on the graphic processing unit. General purpose computing on the graphic processing unit consists of utilizing the hardware capabilities of the graphic processing unit to perform computationally intensive tasks which exhibit many opportunities for parallel execution. While many vector or matrix-based data structures and algorithms showcase the performance benefits through this computing paradigm, many graph/tree-based data structures and algorithms are understood to be unsuitable for the nature of the GPGPU computing paradigm.

The d-heap, a tree-based data structure, has undergone many design changes to take advantage of different trends in computer technology. The introduction of the memory hierarchy and the popularity of varying levels of data caches presented the development of the implicit d-heap which ensured that child nodes would not span across cache blocks. Based upon the general structural design of the implicit d-heap, is the asymmetric d-heap, introduced by Brian Vinter and Weifeng Liu. The asymmetric d-heap seeks a heterogeneous solution to the common heap data structure that utilizes both the throughput oriented processing cores of the graphic processing unit and the latency oriented processing cores of the central processing unit. We explore both the limitations of current GPGPU computing solutions and the possible performance benefit opportunities of a truly heterogeneous system in understanding the nature of the ad-heap data structure which is designed specifically for the tightly coupled THC(Truly Heterogeneous Computing) architectural concept promoted

by the HSA(Heterogeneous Systems Architecture) Foundation. Using the batch k-selection algorithm, the behavior behind the design of the ad-heap presents a great deal of interesting information which can be utilized in the design of future heterogeneous solutions for existing data structures and associated algorithms which would normally not benefit from current GPGPU technology. Using both a loosely coupled discrete-GPU based experimental platform and a more tightly-coupled accelerated processing unit-based platform; we explore many of the limitations concerning the asymmetric d-heap design by understanding the performance and behavior of the design on both platforms. We do so by presenting a more accurate and practical implementation of the ad-Heap design for both experimental platforms and addressing the performance metrics and limitations uncovered by the series of experiments. By understanding these limitations and analyzing the different aspects of the general design; we begin to understand many of the design decisions and other general details that have to be considered when distributing the computational workload between both devices on a HSA-based architecture.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

A BRIEF INTRODUCTION TO HETEROGENEOUS COMPUTING

By taking the very basic meaning of Heterogeneous(heter·o·ge·ne·ous), defined in the Merriam-Webster dictionary as "made up of parts that are different", and Computing(com·put·ing) defined by the same source as "to determine especially by mathematical means"(Merriam-Webster Online (2009)); we can very easily understand the primary motivation behind the popular computing paradigm described by Heterogeneous Computing. The typical computing platform is a fascinating myriad of hardware components working closely together to produce a solution to a given problem. Even on such a minuscule level, each individual logical gate has a different purpose, a different methodology behind their execution mechanism. Given these differences, each logical gate within an integrated circuit execute to produce a common synergistic solution. Synergy is very simply described as the instance where the combined production of all parts working together is greater than the sum weight of those same parts. Though an important trait in successful business management environments, synergy is also extremely important on a fundamental level of computer architecture. Synergy is a term that perfectly characterizes the purpose of heterogeneous computing.

1.1 The Computing Paradigm of Heterogeneous Computing

The computing device is comprised of many electronic hardware components. A heterogeneous computing solution seeks to utilize all of these components in an efficient manner which best meets the goal of the application or the programmer of the application. The programmer must seek a solution to his application which best utilizes these components in a

synergistic manner. Application performance goals may vary from platform to platform depending on both the user-base and the hardware requirements. Some solutions may demand the best execution performance for a specific algorithm or method while other solutions may utilize an optimal scheduling heuristic to ensure that the overall hardware platform is fully utilized and all tasks are receiving enough processing power to balance their computations at an optimal level. With the recent popularity in mobile computing electronics, a primary goal of many heterogeneous computing solutions have been to effectively perform at an optimal level while efficiently throttling the overall power consumption of the device. Given the limited power resources of modern mobile electronics, the ability to produce the application results while maintaining a suitable power consumption level has been crucial in both software and hardware design. Heterogeneous computing is both a software and hardware-based paradigm with much ongoing research in both areas.

1.2 Recent Development Trends in Heterogeneous Computing

Recent research and development has resulted in a interesting concept titled as Heterogeneous System Architecture (HSA), a topic which will ultimately be discussed in a later chapter. Many hardware manufacturers such as AMD, ARM, and Samsung have invested a large amount of development effort in heterogeneous hardware platforms which locate both the graphic accelerator compute units (i.e., GPUs) and the low latency central processing cores (i.e., CPUs) on the same silicon die. The two groups of processing cores share either the same system memory directly or the same last-level cache within the hardware design. By situating both hardware components on the same silicon die, the PCI-E bus which is used for communication between the devices is removed, thus removing a possible performance bottleneck (memory transfers)(Daga et al. (2011)). AMD's recent APU(Accelerated Processing Unit) is based around this design concept(Branover et al. (2012))((AMD APU Fusion), 2010). The more tightly coupled heterogeneous platform allows for performance and power consumption benefits while only sacrificing the raw computational peak perfor-

mance of a discrete graphic processing unit(D'Alberto (2012)). Intel's HD Graphics and Ivy Bridge architecture was Intel's architectural debut into this realm of heterogeneous computing(Damaraju et al. (2012)). AMD's "Kaveri" platform is the most recent development in accomplishing the design concept described as a truly heterogeneous computing system by AMD.

The hardware capabilities of mobile devices such as phones and tablets have accelerated in recent years. With this acceleration of technology, application demands have also seen a large amount of growth and focus. Computer vision and augmented reality-based applications were once mainly a application area limited to powerful stationary computers with large and powerful graphic processing units capable of handling the large load of performing the computational intensive feature detection and graphics rendering. With innovations in both hardware design and heterogeneous computing technology, these applications have been introduced to the realm of mobile electronics; allowing anything from 'smartphones' to 'smartTVs' to demonstrate exciting augmented reality and graphically-intensive applications. There has been a great deal of developmental research in implementing existing efficient computer vision applications using newly introduced heterogeneous computing API features in popular open source computer vision libraries such as OpenCV. OpenCV is an interesting example of a popular development library which now includes several heterogeneous computing aspects to better improve the performance of its computer vision features(Pulli et al. (2012)). Computer vision application development frameworks such PTAM(or Parallel Tracking and Mapping) can utilize many of the GPGPU libraries within the OpenCV computer vision library to accelerate the performance of their FAST corner feature detection computation(Klein and Murray (2007)).

Topics such as AMD's "Kaveri" technology and OpenCV's additional GPGPU computing libraries are excellent examples of both the exploration and interest presented by software and hardware companies in heterogeneous computing(Bradski). From investigating the performance differences between discrete and integrated graphic processing units in

general purpose computations to exploring the potential performance benefits of incorporating OpenCV's OpenCL libraries in the parallel tracking and mapping augmented reality application; heterogeneous computing has presented a volatile and interesting research area with many aspects to explore and many areas that I have yet to fully understand. In this large area of potential research, general purpose programming for graphic processing units present one of the largest areas of innovation involving heterogeneous computing.

CHAPTER 2

THE BASICS OF GENERAL PURPOSE PROGRAMMING WITH THE GPU

To best understand the concepts discussed and theorized in the overall research surrounding this thesis requires only a very basic understanding of both computer architecture, graphic processing unit design, and general purpose computing with graphic processing units (a.k.a., GPGPU). General purpose computing, an incredibly broad area of programming, is generally described as computational tasks without a specific or more-so simply a general purpose. Whereas the typical programming tasks for a graphic processing unit can be described as graphics rendering or the typical programming tasks for a sound processing unit can described as processing and producing audio signals to and from the computer's applications; general purpose computing tasks are often handled by the central processing unit of the computer. As you can imagine, general purpose computation with the graphic processing unit involves designating a subset of these general purpose tasks to be handled by the graphic processing unit of the computer. Revisiting the principals described by heterogeneous computing, all tasks within this subset of general purpose computational tasks must share some characteristic or detail which allows them to be processed more efficiently on the graphic processing units. To better understand how to distinguish between these programming tasks requires a small understanding of the architectural designs of both the central processing unit and the graphic processing unit(s).

2.1 Understanding the Hardware Characteristics of the CPU

The central processing unit can be considered as the primary component of the modern computer. Self-described by its name, as a processing unit, the central processing unit

governs all programming tasks and processes central to the overall computer. These tasks include everything from basic arithmetic or logical calculations to the general input and output operations that are typically encountered with your standard computer program. In a sense, the central processing unit can be understood as the governing unit of all other hardware components that the computing device is comprised of. The central processing unit has undergone an extensive amount of changes over the last century with the introduction of transistors, integrated circuits and eventually microprocessors and multi-core processors. The popular Moore's Law, named after Intel co-founder Gordon E. Moore, characterizes this growth in CPU technology by accurately predicting that the number of transistors on integrated circuits would double approximately every two years. As such, the central processing unit is a very unique and exciting component of computer hardware that could warrant an entire paper describing the evolution of its design in itself.

The central processing unit, given its role in the line-up of individual hardware components is to effectively, efficiently, and accurately execute the computer program(or computer programs). The central processing unit is built with extensive instruction pipelines consisting of complicated integrated circuits and designed around an efficient memory hierarchy which begins with the processing core's registers and continuing through a series of memory and instruction caches before reaching the system memory. All of these aspects of the central processing unit's design is simply to achieve one central goal; to effectively, efficiently, and accurately execute the computer program. In general, the central processing unit is optimized for sequential processing and low memory latency. Given the introduction of multi-core central processing units, more opportunity for parallelism exists amongst the physical cores of the central processing units, though this still does not amount to the level of parallelism available in modern graphic processing units. Hence, we begin to draw the metaphorical line between the programming tasks which are better suited for which hardware component(Hennessy and Patterson (2003)).

2.2 Leveraging the Hardware Characteristics of the GPU for General Purpose Programming

Differing from the central processing unit, the graphic processing unit is designed with an entirely different purpose. The primary motivation behind graphic processing units were, and still are, to efficiently render the graphical components desired by the computer application and related software. To achieve this, the graphic processing unit must have the ability to perform thousands of simultaneous calculations to produce the accurate image frame according to the demands of the software (e.g., graphics pipeline). This design requirement eventually lead to the large dedicated and discrete graphic processing units which are evident on the modern computer hardware market today. The computer architectural reflection of this requirement is a fairly large electronic circuit composed of thousands of processing engines or cores. This design trait allows for a large level of data parallelism and throughput while sacrificing the memory latency presented in modern central processing units. The graphic processing unit does offer its own memory hierarchy with its own associated memory spaces very similar to modern central processing unit design(Hennessy and Patterson (2003)).

Figure 2.1 gives a very basic representation of a traditional loosely coupled heterogeneous computing platform consisting of a discrete GPU and multi-core CPU platform. The graphic processing unit device hardware consists of many compute units, each compute unit with its own large amount of SIMD processing engines. In the loosely coupled discrete GPU-based heterogeneous platform, communication and memory transfer between devices take place on the PCI-E bus. This connection scheme often presents a potential bottleneck in computationally intensive applications that may require frequent large data transfers between both devices.

General purpose computation on the graphic processing unit is to effectively leverage the resources of the device to increase the performance on many computationally intensive programs which offer many opportunities of high-level data parallelism. The graphic pro-

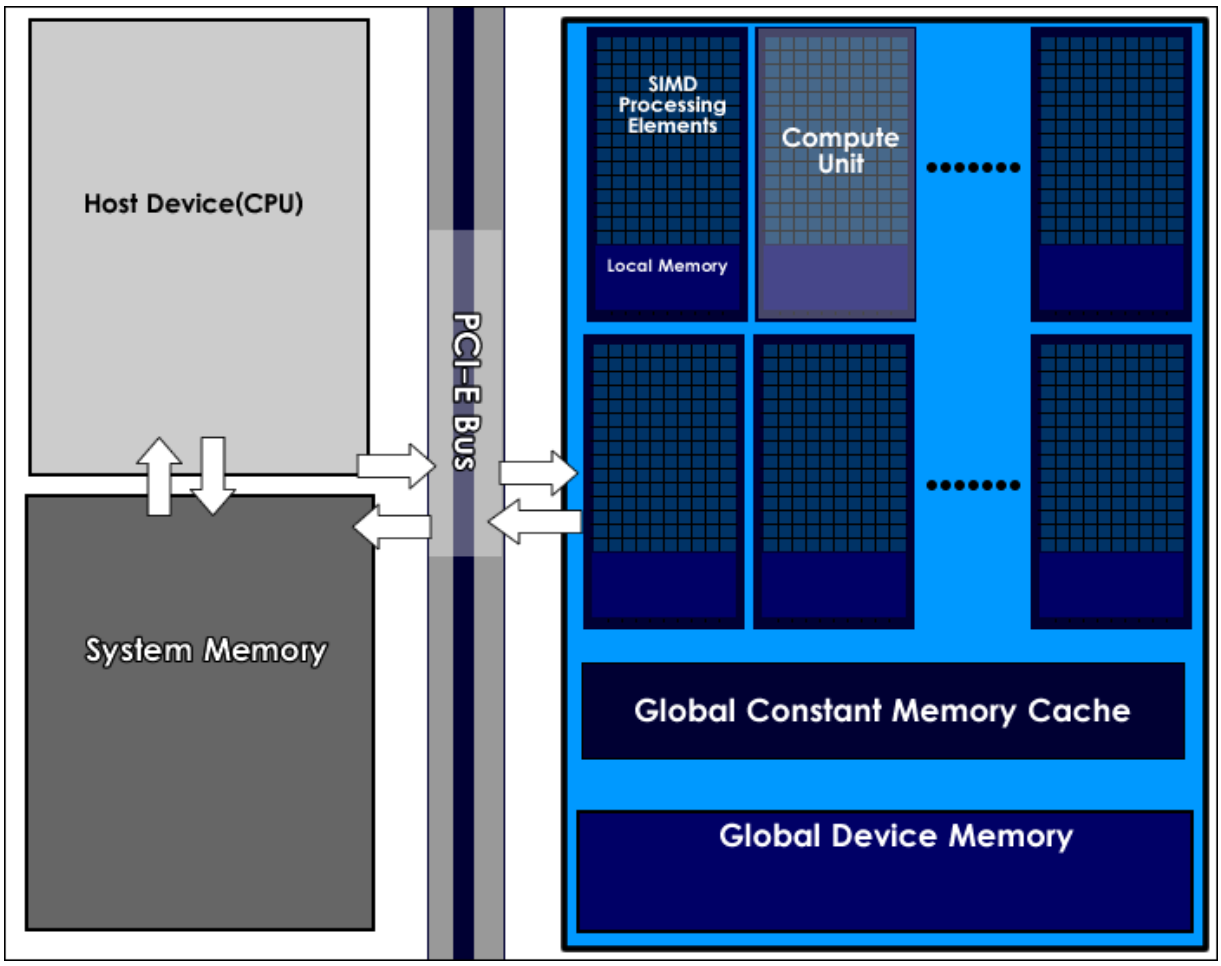


Figure 2.1. A simplified diagram of discrete GPU system and a very high level representation of the architectural components of the GPU.

cessing unit is comprised of thousands of processing cores described as shader engines in relationship to the device for graphic rendering tasks. Within the graphic processing unit, these shader engines could be separated into different groups corresponding to their specific task or purpose within the graphics pipeline in the production of the frame. These different groups consist of the commonly known pixel and vertex shaders and the newer geometry and tessellation shaders. The introduction of the "Unified Shader Model" which introduced a consistent set of instructions across all shader types essentially constructed the developmental bridge into utilizing the graphic processing unit for general purpose programming tasks.

The consistent instruction set introduced in the unified shader model eventually lead to the development of general purpose compute programs known as kernels. To understand the intricacies and mechanics of kernel programs requires a general understanding of both the architectural philosophy of the graphic processing unit and the programming platform or API used to construct the kernel program. To understand the nature of kernel programs is to understand the nature of the graphic processing unit itself and more importantly the shader engines(or stream processors) which the graphic processing unit uses to achieve the high-level of throughput to produce better performance in computationally intensive parallel applications. It is also helpful to remember that the graphic processing unit is merely supplemental to the central processing unit, thus staying true to the very synergistic nature of heterogeneous computing.

2.3 Modern Parallel Programming Platforms for GPGPU Programming

In the area of general purpose computing with graphic processing units, there exists many programming platforms or APIs whose primary purpose is allowing the programmer to create applications which utilize the graphic processing technology for their own general purpose programs. Two of the most popular programming platforms typically discussed in this area of research is OpenCL and CUDA. CUDA, the product of a collaborated effort of NVIDIA and the University Of Toronto, is a C-based(and later extended to C++/FORTRAN) proprietary programming platform and GPGPU SDK for NVIDIA graphic processing units only(NVIDIA Corporation (2011)). OpenCL, developed by the Khronos Group(originally developed by Apple Inc.), is a cross-platform heterogeneous parallel programming platform and framework. OpenCL is C-based but also offers C++ extensions for programmers who prefer the C++ programming language variety. OpenCL is an interesting programming platform in comparison to CUDA, distinctly because OpenCL does not limit itself to only graphic processing units. OpenCL describes itself as both a cross-platform heterogeneous programming platform, allowing kernel programs to be executed on

any OpenCL compatible parallel processing device which can also include multi-core microprocessors or digital signal processors.

Between these two programming platforms, you may find that synonymous objects and concepts are titled differently which can lead to a lot of general confusion. Information provided by both hardware and software manufacturers also suffer from this confusion; where hardware and software components may switch titles depending on the context of the sentence that they are presented in. Examples of this synonymous terminology include stream processors which are often described as shader engines, SIMD cores, or SIMD engines. The SIMD acronym means "Single Instruction Multiple Data" and characterizes the execution model of the kernel program. SIMD originates from a characterization known as Flynn's taxonomy which consists of SISD(Single Instruction Single Data), SIMD, MISD(Multiple Instruction Single Data), and MIMD(Multiple Instruction Multiple Data). In later revisions, the above definitions have been extended to programs rather than just instructions. For the nature of general purpose computing on graphic processing units, we only concern ourselves with the SIMD model.

As we explore the programmability of the graphic processing unit and the programming platforms which unlock this capability, we encounter synonymous terms such as work-items(OpenCL) and threads(CUDA). Similar to the standard computer science definition of a thread, the work-item or thread can be viewed as the smallest unit of computation which essentially executes the sequence of instructions described in the kernel program. In addition to work-items and threads, we have work-groups or thread-blocks. As you can assume from their given labels, work-groups and thread-blocks can be one/two/or three dimensional structures comprised of work-items or threads, respectively. Continuing with the case of the OpenCL programming platform, you begin to see a hierarchy of computation. This hierarchy begins with a overall grid consisting of work-groups of work-items. These work-items are further grouped together to form corresponding wavefronts within the work-groups. The wavefront consists of 64 work-items which are executed simultaneously.

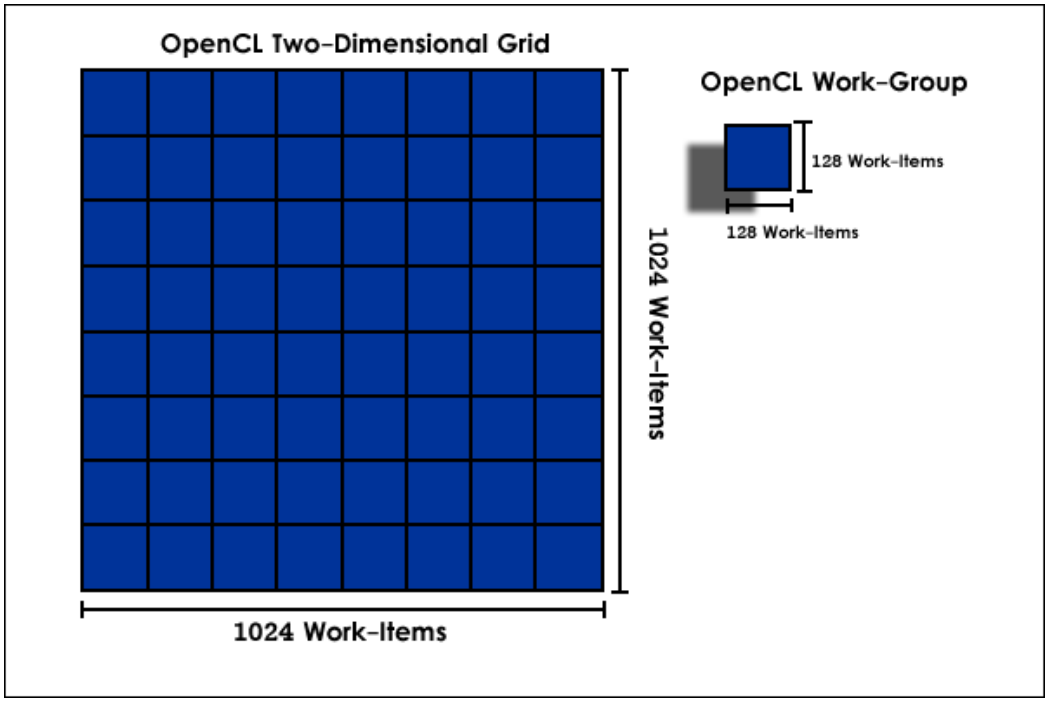


Figure 2.2. Understanding the thread configuration and how it applies to your problem space and algorithm allows you to understand how to best utilize the hardware.

Understanding these programming concepts allows the programmer to visualize the parallelism of his application and how different aspects of his application can translate to a kernel program. Understanding the underlying hardware concepts can help the programmer understand the execution model which his program relies upon. Revisiting the hardware components of the graphic processing unit described previously, these thousands of stream processing cores are distributed to a group of compute units(also often called execution units). Each compute unit has its own local (or scratchpad) memory while all compute units share a common constant/global device memory area. Relating to this architectural model, the previously mentioned work-groups are processed by the compute unit. All compute units execute their work-groups simultaneously. Though a compute unit can process several work-groups (depending on the hardware and memory limitations of the device), the compute unit does so in a sequential manner. Each work-item has its own private memory

while every work-item in a given work-group share a memory area titled as local memory(or scratchpad memory). Every work-item in the global space share the same central device memory area appropriately named global memory. As you can imagine, given the purpose and motivation behind the architectural design of the graphic processing unit, the memory bandwidth between these memory spaces differ greatly and provide a primary focal point in the implementation details of this research.

Above is a very basic description of the relationship between the software and hardware components which are described by the OpenCL programming platform for graphic processing units. As you can imagine, given the volatile and dynamic nature of the graphic processing technology and ongoing development and innovation in this area, there are many different real-world varieties of this technology ranging from the basic and heavily marketed discrete graphic processing units to the interesting system-on-chip heterogeneous variety which is currently rising in popularity amongst mobile devices. Given these varying representations of the same fundamental hardware device, the discussed programming platform aspects remain consistent between each of these representations.

2.4 Adapting a Serial Executing Application for GPU Hardware

From a software or program-level, the implementation or incorporation of OpenCL within the application consists of separating the application into two essential parts. The portion of the application that is primarily executed on the central processing unit is called the "host" code and serves as the primary program which sets up the OpenCL programming environment, manages and creates the OpenCL data structures, launches the OpenCL kernel(s), and operates on the return values of the kernel programs. The standard OpenCL setup process consists of identifying the OpenCL platform on the computer and recognizing the array of OpenCL compatible devices of this platform. This array of OpenCL compatible devices can consist of both multi-core central processing units as well as graphic processing

units or any general purpose parallel processing unit. Once the OpenCL compatible device(s) is chosen, the OpenCL context is created for the use of managing other OpenCL objects such as command queues, OpenCL memory buffers, and also OpenCL kernel objects amongst this OpenCL device or collection of OpenCL devices described by the OpenCL context. There is a large amount of setup code involved in implementing OpenCL into the application but much of it can simply be repeated from application to application in most cases. The large amount of general setup code does allow for a large amount of control over the OpenCL components of your application(Munshi et al. (2011)).

Aside from setting up the OpenCL environment within the application, the programmer must also seek to understand which areas of his application can benefit the most from the graphic processing unit technology. This approach can begin with profiling and characterizing which portions of the application are most computationally intensive. Once these portions are discovered and profiled, the programmer must determine the parallelism opportunities of these computationally intensive sequences. Essentially polarizing the computationally intensive highly parallel portions of the program and the standard sequential portions is the key to determine which areas of the application can benefit most from general purpose programming on the graphic processing unit. Often an easy way to visualize this strategy is to seek for portions of the program which perform a small amount of calculations over an extremely large data set. A common example of an algorithm which exhibit these characteristics is the basic vector addition example which performs the addition of two elements in two separate vectors while storing the result in another vector at the index which corresponds to the index of the two elements of the original vectors. Another common example is matrix multiplication which performs the multiplication between two matrices(fairly self-explanatory)(Matsumoto et al. (2012)). Both of these examples are often cited as the beginner's introduction to the realm of general purpose programming with the graphic processing unit; the essential "Hello World" of the heterogeneous computing world.

Adapting the application to utilize the OpenCL programming platform requires both

a great understanding of the algorithm's problem domain and also the characteristics of the OpenCL programming platform which will exploit the potential parallelism of the algorithm. The kernel program, which can be viewed as a function itself, is a unique program based in a somewhat limited version of the C99 C standard. Once the kernel program has been created, the programmer acquires the responsibility of designating the problem domain that kernel program shall execute within. This domain is characterized by two primary aspects, the global work-items specification and the local work-items specification. These two details can be expressed in three dimensions, depending on which representation maps best to the algorithm that it is effectively attempting to optimize. In a naive explanation, if the problem domain consists of a single one-dimensional data set then the most optimal specification would be one-dimensional; this includes problem domains that require a computation on a large vector or array of data. Similarly, such as the matrix multiplication example discussed earlier, if the problem domain consists of a two-dimensional data set, these specifications can be expressed as two dimensional; and so forth with a three-dimensional data set. Acknowledging these details allows the programmer to specify the amount of work-items and the subsequent grouping of work-items in a identifiable model that the kernel program will effectively execute on.

While this may seem overcomplicated to a beginner programmer in OpenCL, with ongoing experience in utilizing the OpenCL programming platform in their programs, the programmer begins to see the optimization possibilities and the resulting parallel execution model more clearly in their subsequent applications. Returning to the concept of work-items and work-groups, once the programmer has specified the dimensions and amount of work-items in the global problem space and the dimensions and amount of work-items in the local problem space(within each work-group), OpenCL processes this information and executes the kernel program within this specified problem space. The behavior of the work-items within this N-dimensional problem space must be understood by the programmer when translating portions of his algorithms(or his entire algorithm) to a suitable kernel program. Work-items

within the global problem space share the same global memory space while work-items within the same local work-group share the local memory space. Synchronization methods such as work item barriers and memory fences are familiar instruments to most parallel programmers and they are also required within the kernel program to protect the integrity of the model and prevent any discrepancy between the OpenCL execution model and the kernel program itself. Within the 64 work-item wave front, there is also an opportunity for thread-divergence which can also affect the overall hardware utilization and present opportunities for less-than-optimal performance.

2.5 Understanding the OpenCL Memory Model

Revisiting the OpenCL memory model, we acknowledge that there are four primary memory spaces that the kernel programmer must be aware of when optimizing his algorithm based upon OpenCL's specifications. The first and foremost is the system memory or host memory which is shared between the "host" device and the OpenCL device. In discrete graphic processing platforms, memory transfers between the host device's system memory and OpenCL device's global memory is performed over a PCI-Express bus. This transfer protocol presents a performance bottleneck where the OpenCL kernel's performance is limited by the slower transfer bandwidth of the PCI-Express bus. There has been a large amount of research and focus in hardware solutions to relieve this bottleneck. One common approach is evident in a modern hardware component described as an APU or accelerated processing unit. In an accelerated processing unit, the graphic processing unit and the central processing unit are much more closely coupled on a single silicon die. While this removes the memory transfer bottleneck and improves the power consumption of the device; there are some performance limitations inherent to the graphic processing unit design when coupled with the central processing unit on the same silicon die. These limitations can be described as a lower number of dedicated transistors to the graphic processing unit resulting in a lower number of stream processing SIMD cores and a lower number of general compute(or

execution) units(Munshi et al. (2011)).

The next memory area of interest within the OpenCL memory model is titled as global memory. The name, rather self-explanatory, implies the definition of the memory space; global/constant memory is the memory space shared by all work-groups within all compute units on the OpenCL device. This memory area is not synchronized between work-items or work-groups, so careful detail and focus must be exercised in the memory handling of the kernel program when working within this memory space to prevent any discrepancy between the global work-items which read and write from this memory space. Within each work-group is a memory space titled as local(or scratchpad) memory which is synchronized between all work-items within the work-group. Each work-item within the work-group shares this memory space and acquires the same view of this memory space. Predictably, the memory latency between the local memory space and global/constant memory space differs greatly, therefore it is suggested to perform the majority of memory accesses in the local memory space area and limiting the amount of memory access to the global memory space to increase the overall performance and lower the overall memory latency.

Since all work-items within the work-group share this same local memory, there is also a large amount of careful detail that must be exercised in programming the kernel program to execute accurately and consistently within this local memory scope. Each work-item has its own private memory which is used by that work-item for storing and processing information specific to that work-item such as common variables within the kernel program. Data within this private memory can not be accessed or viewed by other work-items. Generally, designating the memory space that information must be stored in is done either within the kernel function's argument field by an associated prefix before each argument or within the kernel program itself by the same prefix. OpenCL kernels may include auxiliary functions but given the nature of the memory addressing scheme within the OpenCL memory model, these functions cannot take arguments such as pointers to memory spaces; therefore these functions can be easily understood or viewed as inline functions within the main kernel pro-

gram. Understanding the OpenCL memory model allows the programmer to understand how to structure the memory accessing nature of his resulting kernel program.

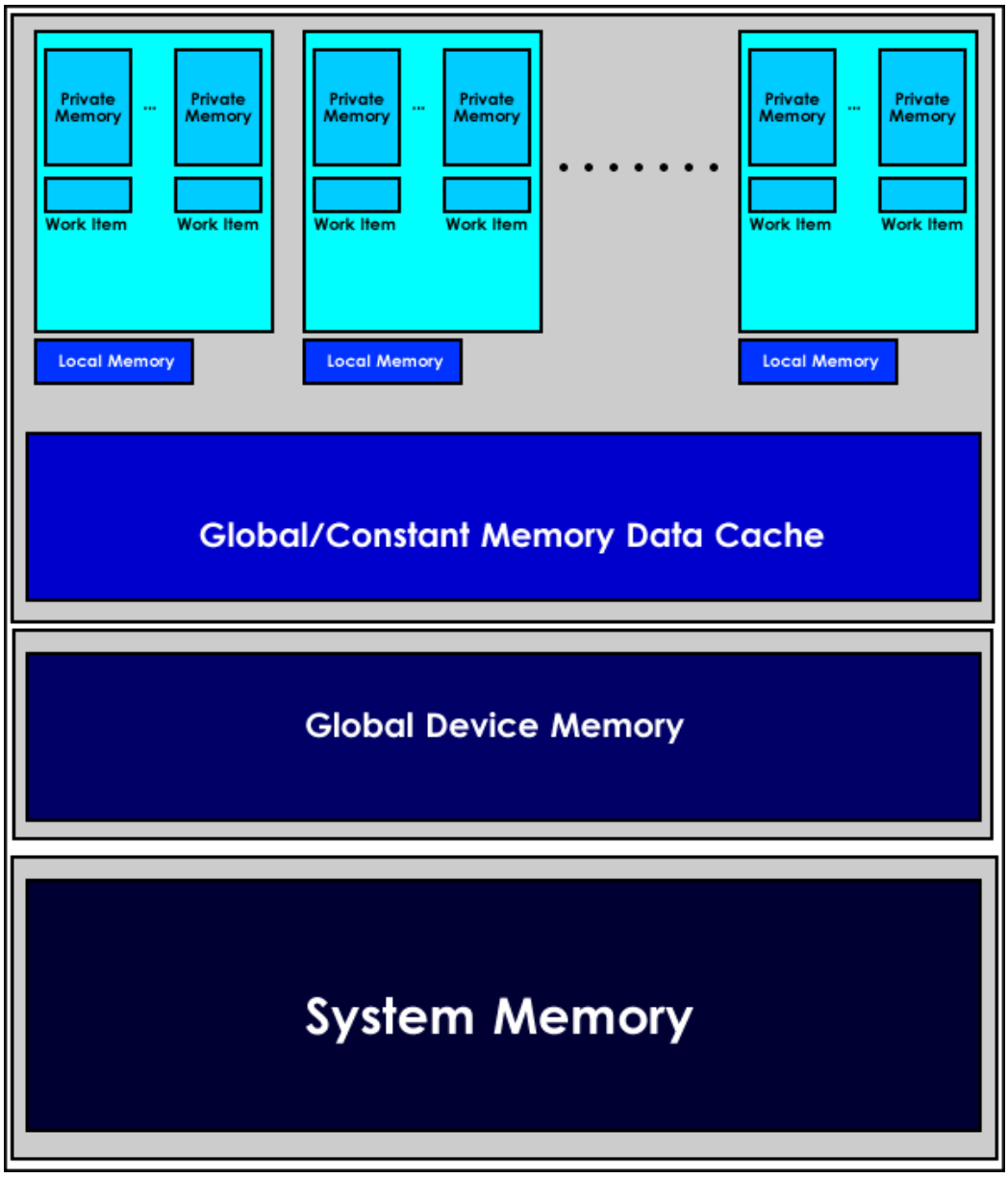


Figure 2.3. The layout of the very basic graphical representation of the OpenCL memory model.

To summarize the entire OpenCL programming model in a very basic manner, we begin with the OpenCL components present within the "host" program which consists of the basic setup OpenCL objects such as the OpenCL platform(s) comprised of OpenCL

devices. The OpenCL context consists of the management of the OpenCL objects such as OpenCL memory buffers, OpenCL execution command queues, and OpenCL programs which consist of OpenCL kernel objects, across the collection of OpenCL devices within the OpenCL platform of interest. You begin to note a hierarchy present within this realm of the OpenCL programming model. While this may seems like a very large and tedious amount of boiler-plate code; this hierarchy allows for an high level of control and design freedom to optimize the application for the performance benefits of high-level data and task parallelism. Understanding this aspect of the OpenCL programming model and adapting it your application involves understanding the relationship between your application and the OpenCL device(s).

2.6 Summmarizing the Basics of GPGPU Programming with OpenCL

To further summarize the OpenCL programming model that we have discussed in earlier pages. We explore the OpenCL execution model which closely mirrors the hardware architecture details of the OpenCL device with concepts such work-items and work-groups with each work-item processing the sequence of code within the kernel object. The work-items are handled by the OpenCL device's SIMD processing cores while the compute units of the OpenCL device effectively handles each local work-group. Understanding the nature of the OpenCL device specific to the platform executing your application helps you determine which OpenCL device is best suited for your algorithm and which execution model is best suited for that OpenCL device. In addition to understanding the very basic nature of the OpenCL execution model, the OpenCL memory model describes the resource management detail that must be considered to promote the best performance out of the kernel program. Understanding the role and characteristic of each individual memory area allows the programmer to effectively utilize the entire hardware in an incredibly efficient manner exhibiting both high level parallel performance and efficient memory handling for low memory

latency. Understanding the hardware and software-related characteristics of each hardware components allows the programmer to develop an optimal workload distribution scheduler to determine which workload is best for which device; this approach has resulted in many interesting research topics such as the dynamic scheduling of the breadth-first search algorithm over real-world graph instances(Hong et al. (2011)).

It may seem that only specifying details of the OpenCL programming platform for general purpose computing on the graphic processing unit may limit the understanding of the concept to only the details of the OpenCL programming platform; given the fact that there are many other programming platforms available. But the concepts discussed in the previous pages involving the different characteristics of OpenCL programming are easily translated to other existing GPGPU programming platforms as well. Given the design of modern graphic processing units, each programming platform follows a similar programming model design to exploit the performance benefits based upon the central architectural design. OpenCL is the most widely used and available implementation of this programming paradigm due to its open source and cross-platform nature which are the basic design goals by the Khronos Group. For this thesis, OpenCL has been the platform of choice.

Understanding the very nature of heterogeneous computing and general purpose programming on graphic processing units will provide insight behind the motivation of my research and the implementation details described in my experiments. Understanding the potential benefits of incorporating these concepts in modern algorithms to fully utilize modern technology designed around the concept of heterogeneous computing will allow you to see the nature and behavior. Described in this chapter is only the very basic details of general purpose computing with graphic processing units which are required for further understanding of this document and its associated research. Programming-specific details and API characteristics were not discussed and can provide material for yet another paper; but their details aren't effectively useful for understanding this document.

CHAPTER 3

ADAPTING THE HEAP FOR HETEROGENEOUS COMPUTING

Algorithm(Al·go·rithm), is a term originating from the transliteration of the surname of Arabian mathematician al-Khwarizmi, famous for his introduction of mathematical concepts such as algebra to Western civilization. An algorithm is best described as the sequence of steps to achieve a solution to a specific task or problem. In the realm of computer science, an algorithm can be understood as the mechanics of the computer program which execute to perform a certain task. Every computer science student is extremely familiar with the concept of an algorithm; perhaps from a formal definition perspective or simply the inherent nature of their approach to programming solutions. With further investigation of algorithms within the area of computer science, we begin to see their most primitive components, the data structures(Levitin (2002)).

Re-examining the phrase "*Computer and Information Science*", we note the inclusion of the term "*Information*". A large component of the study of Computer Science involves understanding the representation and handling of information, often referred to as data. The purpose of the computing device as a mechanical device is to effectively process, represent, and handle data for the purpose of accomplishing a solution at the discretion of both the programmer and the user. To accomplish this goal, information or data must be represented in a manner which allows the device to easily perform the calculations and computations demanded by the algorithm. These resulting structures are appropriately called "Data Structures" and they exist in many different varieties with different purposes. Combining the science behind these data structures and the mechanics of the algorithms that utilize them presents an ever-changing area of focus, research, and analysis to develop algorithmic solutions that use the computing device hardware more effectively.

Many of these information representation structures have a counterpart within the area of mathematics. For example, many varieties of the graph structure are utilized in computer science, these varieties can also be characterized in their mechanics and efficiency by graph theory, a vast and interesting area of research and study within mathematics. Often these complicated and intricate structures are representative of a much larger set of information being processed by the computer program; this allows for a level of abstraction of the problem domain being handled and processed by the computer program. Internally, or within the programming language itself, exists a concrete representation that can be modified or structured to implement a much more complicated data structure. An example of this concrete data representation within the C programming language is the basic array.

3.1 Addressing the Nature and Design of the Heap Data Structure

Referring back to graph-based data structures, a great example of this structure type is the tree-based data structure, the heap. Tree-based implies an acyclic graph hierarchical representation of the information which begins with a "root" node branching into subsequent nodes with each subsequent node branching into more nodes. The heap is an unusual tree-based data structure in comparison to other common tree-based data structures such as Binary Search Trees, Red-Black Trees, and also B-Trees. Each heap implementation or representation share a common property that distinguishes them as a heap; this property is the ordering of the nodes which the Heap data structure consists of. The heap data structure comes in two primary varieties, the Max Heap and the Min Heap. As the names imply, the Max Heap is characterized by the root node having the maximum value or key of all nodes within the Heap data structure; the Min Heap is the opposite. The ordering property of the heap data structure reflects these varieties by ensuring that the keys of the children of a specific node is either greater-than or less-than the keys of that specific parent node. With this ordering specification exhibited throughout the entire heap data structure; the root value

becomes whichever node that has the maximum or minimum key or value(Levitin (2002)).

The heap data structure is a complete tree structure exhibiting the smallest possible height for a Tree-based structure based upon the number of nodes within the heap. As nodes are inserted into the heap, the Tree-representation essentially grows from left to right on the fringe level of leaf nodes until a new level within Tree structure is required. The mechanics of the insert operations are different from typical tree insert operations to ensure that the integrity of the heap ordering property is retained. The heap structure provides an efficient way to extract the maximum or minimum values amongst a collection of values or keys in constant time. While the root node's value or key is the maximum or minimum order statistic of all keys or values within the tree, the Heap isn't completely sorted in any definite manner given the lack of relationship between all keys or values on any given level within the heap tree.

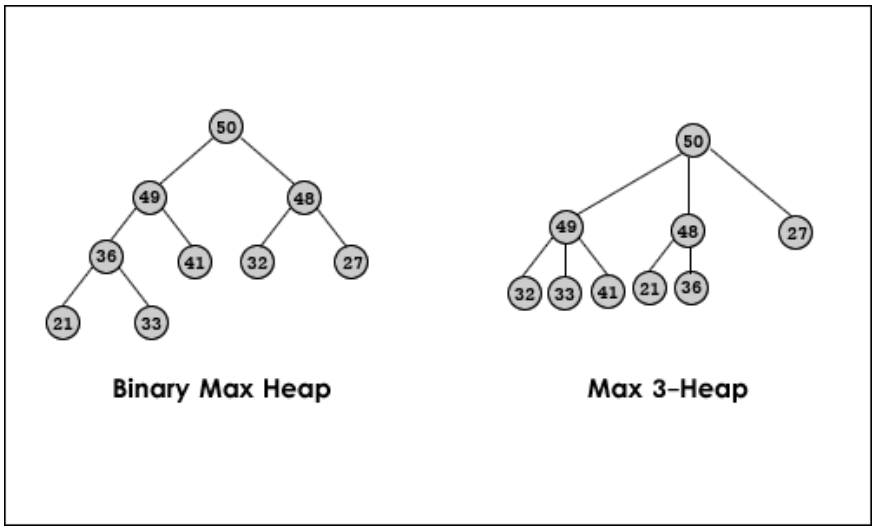


Figure 3.1. As with most data structures and information representations, it is much easier to understand with a visual reference.

In its most simplistic form, the Binary Heap, the heap consists of nodes where each node, aside from the nodes on the last full level, have two children nodes. The nodes within the last full level of the heap, following the description stated earlier in this chapter, can have as little as no children depending on the population of the last fringe level of the

heap data structure. This is the most basic implementation of the heap data structure. A representation that would exhibit a smaller number of children yet retain the connectivity of the graph structure would result in a sorted degenerate tree. The heap structure can be further extended to include cases where the heap nodes are allowed a greater number of children. This is known as the d-heap or d-ary Heap where d denotes the maximum number of children each node is allowed. d-heaps are very common within many graph-based algorithms in computer science.

The basic operations of the heap data structure are similar to the basic operations surrounding most common data structures. There is the *Insert-Node* operation, *Update-Key* operation, and *Delete-Min/Max* operations and associated observer operations for extracting information from the heap structure such as the value of the heap's root node. The *Insert-Node* operation is fairly straightforward, simply inserting a new node and associated key-value into the heap structure. The *Update-Key* operation essentially changes the key value of a given node within the heap structure. The *Delete-Min/Max* operation removes the root node from the heap and oftentimes returns this value to the calling function for further use. The mechanics of these operations become a unique focal point, since any operation that modifies the structure of the heap structure or the values of the nodes within the heap structure might require additional work to ensure that the heap ordering property remains intact. This additional work can be described by two different methods known as bottom-up reconstruction or top-down reconstruction.

In the case of inserting a new node or updating the value of a node to a new value which is greater than(or less than in the case of a Min Heap) the original's parent node's value, the bottom-up reconstruction is utilized. In the bottom-up reconstruction, the new node or node of interest is propagated up the structure of the heap until the heap ordering property is satisfied. Similarly, the top-down reconstruction method is used when the new node or value is propagated down the structure of the heap until the heap ordering property is satisfied. When a new node is inserted into a heap structure, the node is placed as the

last possible child node within the heap given the heap's structure constraints. This node is then propagated up through the heap through a series of comparison and swaps with node and its subsequent parent(s). When the root node of the heap is deleted, the last node in the heap replaces the root and comparison and swaps between the node of interest and its maximum child node are performed until the ordering property is satisfied. When a node's key is updated or changed to a new value, either approach can be taken; depending on the relationship between the node and its parent or associated children node. Understanding the mechanics of these operations allows for an understanding of how to optimize the these operations in heterogeneous solutions discussed in the previous chapters.

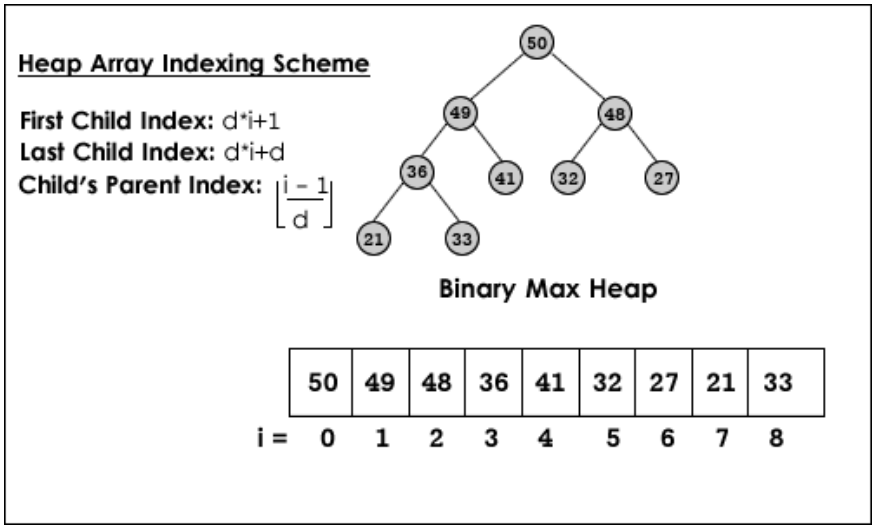


Figure 3.2. The array indexing scheme demonstrated on a binary heap structure.

The concrete representation of the Heap structure is often implemented as an array where each element within the array corresponds to a node within the heap structure. This representation presents an efficient indexing methodology of accessing heap node details such as the parent of a specific node or the corresponding children of a specific node. These constant-time indexing operations provide an efficient method of execution for performing the basic operations that are natural to the heap data structure.

3.2 Adapting the Heap Data Structure for Latest Trends in Technology

The heap is an incredible tool in implementing many efficient and common computer science algorithms such as finding the kth order statistic from a collection of values(an algorithm we'll be inspecting further), the essential Heap Sort, and many graph-based algorithms. Research is continuously driven by the possibility of further optimizing both algorithms and data structures to effectively take advantage of modern technology and modern programming platforms to improve their general performance. One interesting modification in the concrete representation of the Heap structure, for better performance on symmetric multicore processors, is the introduction of the implicit d-heap.

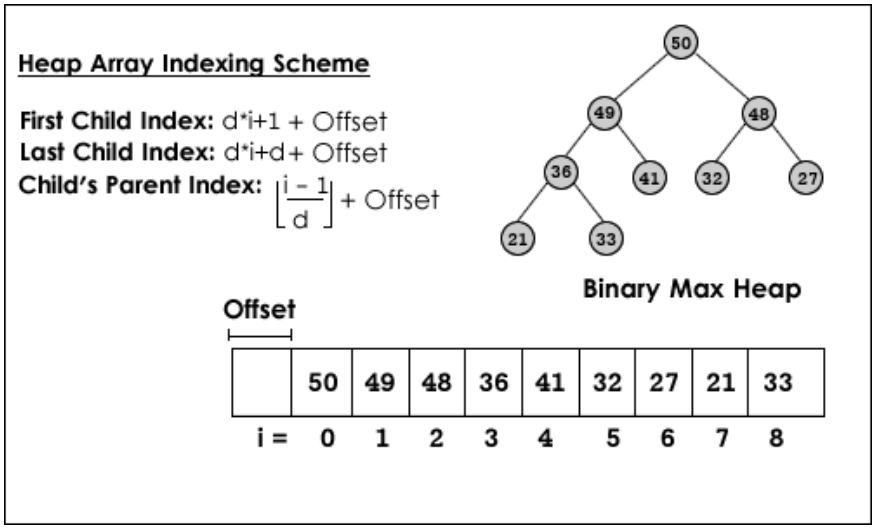


Figure 3.3. The Implicit d-ary heap.

The implicit d-heap was an intellectual response to the growing popularity of the memory hierarchy and the use of data caches on modern processors in the late 20th century. As the performance of the modern processor continued to improve with higher clock frequencies, the memory latency between the processing core and the physical system memory became a performance bottleneck]. The introduction of a memory hierarchy brought about an interest in adapting different algorithms and associated data structures to take advantage

of the benefits of the cache performance. Ladner and LaMarca proposed the implicit d-heap structural design to better align the components of the heap structure with the processing hardware's cache blocks(LaMarca and Ladner (1996)). In this newer design, siblings of a node would not span over cache blocks. They achieved this design goal by adding an offset of (d-1), where d is the maximum number of child nodes, to the head of the Heap's array representation. This additional offset required minor changes to the array addressing scheme as well.

Similar to the rise in popularity of the associated memory hierarchy, many researchers and developers are investigating the performance optimization opportunities presented by translating portions of their existing algorithms to high throughput-oriented devices such as the graphic processing unit. Revisiting the concepts covered in the previous chapters on heterogeneous computing with the graphic processing unit; many algorithms have exhibited a large performance increase in utilizing the high-throughput oriented hardware of the graphic processing unit. Unfortunately, many highly divergent graph-based algorithms and data structures have had a complicated time adjusting to this current trend and remains a heavily researched topic. For example, the heap structure which had undergone a structural change for better performance on modern symmetric multi-core processors, became the focal point of Weifeng Liu and Brian Vinter(Liu and Vinter (2014)).

3.3 The *ad*-Heap, Designed for Truly Heterogeneous Systems

Weifeng Liu and Brian Vinter investigated the associated operations of the traditional d-heap data structure looking for opportunities to exploit any potential parallelism. They discovered that the only opportunity for parallelism within these operations was the Top-Down Heap reconstruction sequence which required the maximum child of the specific node of interest to be found on each level. In this case, the opportunity for parallelism was to find the maximum child from a large group of children. The main issue of this discovery

was that the level of data parallelism of this process is limited by the number of children or the value of d. Therefore in cases that the value of d is quite small, the opportunity to see any improvement from the graphic processing unit was also very small. In this case, the performance of the utilizing the graphic processing unit would be even worse than the typical mult-core implementation since the hardware of the graphic processing unit was not being fully utilized.

Modern hardware manufacturers, recognizing the potential for more powerful and power efficient platforms, began to research and develop solutions which would eliminate much of the overhead and bandwidth issues that typically bottle-necked the traditional loosely coupled heterogeneous platforms. This technology, a product of the HSA Foundation, is described as a truly heterogeneous platform that would allow the host device and the graphic processing unit to work more closely together within a unified memory space eliminating much of the memory cost and address mapping encountered in traditional loosely coupled heterogeneous platforms. On a hardware level, the central processing unit and graphic processing unit would be connected by either a shared data cache or the system memory; but located on the same silicon die thus reducing the slow memory bandwidth of the PCI-E bus. Given these essential design changes, the application load could distributed more easily by both the programmer and the operating system to which device would be able to provide the best performance. Additionally, this technology would combine the central processing unit and graphic processing unit in a closely manner to reduce the context differential typically encountered with traditional heterogeneous platforms and their associated kernel launching and execution protocol.

Though hardware platforms such as AMD's APU or accelerated processing unit and Intel's merged CPU and GPU have somewhat implemented much of these features on a hardware level by locating both hardware components on the same silicon die; the associated heterogeneous programming platforms have yet to completely implement the necessary synchronization and communication features to remove the context switching overhead. This

is the essence of heterogeneous computing and Weifeng Liu and Brian Vinter theorized and implemented what they described as the ad-heap or the asymmetric d-heap based around this trend of modern technology. The asymmetric d-heap, as the name implies, is an implicit d-heap structured for performance on asymmetric multi-core processors. An asymmetric multi-core processor is synonymous to a heterogeneous computing platform; essentially combining two types of processing cores into one computational unit.

These compute units are separated into two categories, the throughput-oriented unit(such as the typical graphic processing compute unit) and the latency-oriented unit(such as the typical central processing unit's processing core). The design motivation behind the asymmetric d-heap was to effectively separate the execution mechanism of the d-heap to allow the portions that presented opportunity for parallelism to be executed on the throughput-oriented cores and the portions that required sequential execution to be executed on the latency-oriented cores. This is very similar to adapting most algorithms to loosely coupled heterogeneous platforms; but given the nature of the d-heap structure and the relatively small amount of opportunity for parallelism, this provided an opportunity to explore the possible benefits of more closely coupled heterogeneous platforms.

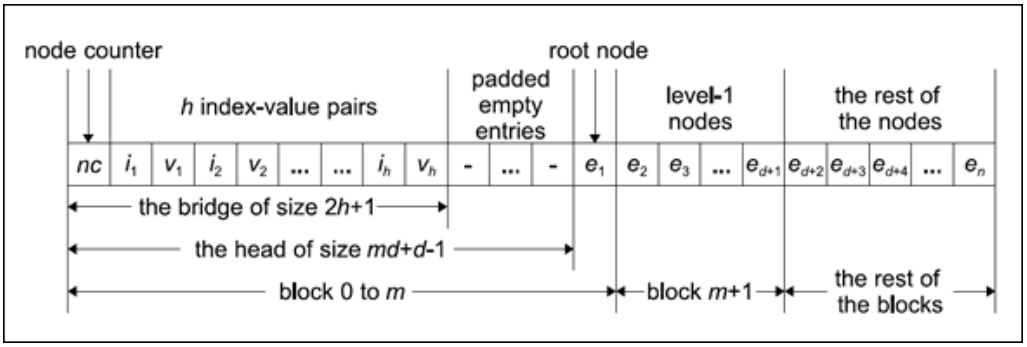


Figure 3.4. The ad-Heap array structure proposed by Weifeng Liu and Brian Vinter(Liu and Vinter (2014)).

To accomplish this, the original d-heap structure was only modified in a modest sense. The original empty head of the implicit d-heap became a storage container for information

that would be later used by the latency-oriented cores. If an operation performed on the *ad*-Heap structure required the heap to be reconstructed from the top down; the graphic processing unit handled the downward propagation of the value. On each level of the heap, the children of the node of interest would be analyzed in parallel to determine the maximum child. Once the maximum child was found, it would be compared to the current node of interest. In a traditional execution of this operation, if the maximum child was greater than the node of interest then the two nodes would effectively be swapped and the node of interest would continue to propagate down the heap. In the case where this computation is performed on the graphic processing unit, a space of memory is allocated in the local memory of the device which is described as the implicit bridge. When a swap is needed between two nodes, the index of the node and the new value are stored as an index-value pair on this implicit bridge rather than the assignment being directly handled within the device code.

This index-value pair placement and implicit bridge construction is to avoid the associated bandwidth and thread divergency penalty incurred by allowing only one thread within the device's wavefront to perform the assignment to global memory. Once the value has been propagated down the heap to its proper location, the workload of the throughput-oriented cores are finished and the wavefront offloads the implicit bridge to global memory in the location of the implicit d-heap's empty head section where the latency-oriented cores can perform the proper assignments based on the information provided in the implicit bridge. The implicit bridge is composed of a node counter which keeps an associated count of the index-value pairs on the implicit bridge.

To perform this effectively, a complicated level of synchronization and communication must be implemented to ensure that both the throughput-oriented compute units and latency-oriented compute units are able to communicate their results and workload with as little overhead as possible. Unfortunately this programming model is only theoretically presented by the HSA Foundation and presented in a theoretical sense in the associated *ad*-Heap paper by Weifeng Liu and Brian Vinter. Despite this, and the current lack of

truly heterogeneous HSA platforms available, Weifeng Liu and Brian Vinter simulated the modern technology with their proposed design by counting the number *find-maxchild* and *compare-and-swap* operations in the d-heap on the CPU and executing the same amount of work with their *ad*-heap implementation on the CPU and the GPU. In their performance statistics of this simulation, they also included the approximate cost of the synchronization between the throughput-oriented and latency-oriented processing cores. They found a significant improvement in this design over similar experimental platforms that executed the operations either strictly on the standalone CPU, loosely coupled CPU and GPU platform.

CHAPTER 4

ADDRESSING THE LIMITATIONS OF GPU COMPUTING WITH HSA ARCHITECTURE

Weifeng Liu and Brian Vinter addressed some of the common issues surrounding general programming on the graphic processing unit in their unique design of the *ad*-heap structure. The design philosophy promoted the use of the theoretical AMP(Asymmetric Multi-core Processors) technology to generate the performance benefits of general purpose programming on the graphic processing unit while negating the possible issues which would normally limit these performance benefits(Vuduc et al. (2010)). Normally these limitations can be categorized as either an under-utilization of the hardware or memory/computational thread-related execution which does not translate well to modern graphic processing unit technology(Owens et al. (2005)).

4.1 Limitations of Current GPU Hardware/Software Solutions

Modern discrete graphic processing units typically perform active data transfers across the PCI-E bus to the system memory which is accessed by the central processing unit. In applications where there is frequent communication and data transfer between the central processing unit throughout the course of the application's execution; there is an associated penalty which corresponds to both the memory bandwidth and the memory paging/handling overhead which can limit the performance of the application. Similarly, there is an estimated associated overhead from the context-switch associated with each kernel launch in an application that may require a large number of kernel launches. Therefore, in an application whose execution is more closely coupled between the central processing unit and

graphic processing unit, the associated overhead of the memory handling and kernel launches can dramatically decrease the overall performance thus rendering the graphic processing unit solution to be ineffective compared to a strict-central processing unit solution.

Algorithms which require a large amount of irregular memory access patterns are typically unable to reap the performance benefits of highly parallel hardware devices such as the graphic processing unit. The GPU hardware is designed and optimized for highly parallel tasks and computation. When individual computational threads issue memory requests in a manner which is not uniform to the group of parallel threads as a whole, the underlying memory subsystem of the graphic processing unit is unable to effectively process the requests in a low-latency efficient manner as the central processing unit typically would. Generally, programmers must adapt their existing algorithms to properly utilize the graphic processing unit's memory subsystem to fully benefit from the computational power of the hardware.

Thread divergence can often affect the performance of modern graphic processing units. If the application's amount of parallel work decreases over the course of the kernel execution, the amount of utilized SIMD threads will also decrease resulting in an under-utilization and inefficient use of the graphic processing unit technology. Oftentimes, this is encountered when thread-id based conditional blocks within the kernel code result in the threads either diverging in their overall work pattern or a large amount of threads becoming idle within much of the kernel program's execution. The resulting low hardware utilization reflects the performance of the kernel program and the overall performance of the application.

When translating sections of the application for execution on the graphics rendering device, the programmer typically investigates the large loop-based computation. Loop-based computations which exhibit iteration-based data dependencies often do not translate well to the parallel nature of the graphic processing unit. In similar nature, algorithms which depend on the manipulation of shared data between each iteration of the loop can promote in-deterministic results from multiple threads attempting to modify or access the shared data in a parallel fashion. To resolve this issue on modern graphic processing units, the use

of synchronization methods and/or atomic operations provide a solution but often hinders the parallel nature of the program's execution and reduces the overall performance.

4.2 The Hardware and Software Design of the HSA Solution

As discussed previously, the *ad*-heap is a very interesting case where the current limitations of the hardware and software solutions of general purpose programming on the graphic processing unit would otherwise prevent any benefits from translating the data structure and its operations from a traditional multi-core central processing unit implementation. Despite these limitations, by adapting the structure design and general execution concept of its operations, the *ad*-heap should inherit an increased performance on systems which follow the True Heterogeneous Computing philosophy. Many hardware manufacturers such as ARM Holdings, AMD, and Qualcomm have combined their development and research efforts with academic research groups at institutions such as Northeastern, University of Illinois, and the University of Mississippi to form the non-profit consortium known as the HSA Foundation. The HSA(Heterogeneous System Architecture) foundation seeks to advance the topic of truly heterogeneous computational systems as innovative technology solutions to the modern heterogeneous computing paradigm((HSA Foundation) (2013)).

The design of the HSA architecture addresses many of the limitations of current discrete graphic processing unit heterogeneous solutions. By tightly-coupling the central processing unit and graphic processing unit, the PCI-E memory transfer penalty is effectively eliminated. Most modern APU or accelerated processing units resolve this potential bottleneck by allowing the graphic processing unit and central processing unit to share the same system memory while residing on the same silicon die. The HSA solution seeks to allow the central processing unit and graphics accelerator unit to share the same last-level cache of the primary memory hierarchy of the system rather than the system memory which further reducing the bandwidth penalty((AMD Developer Central) (2013)).

Further addressing the limitations of modern GPGPU heterogeneous platforms' mem-

ory handling scheme; the HSA software design incorporates the use of a unified virtual memory space. In current modern GPGPU heterogeneous solutions, there is a large amount of memory setup and transfer handling overhead. Aside from the underlying behavior of pinned memory within the OpenCL programming platform and its associated memory-handling commands, the programmer must be able to adequately structure and manage his memory and its associated transferring schemes. With a unified virtual memory space, the addressing scheme between CPU and GPU devices becomes more natural to the programmer. This allows pointers to be freely passed between both devices but also more physical addressable memory can be paged to and from the disk.

Applications which typically execute over increasingly large data sets may not be applicable to current graphic processing units and their limited physical memory capacity. The unified physical and virtual memory requirement of HSA-based hardware and software solutions will resolve this issue. By removing the typical off-chip memory access traffic of modern heterogeneous GPGPU solutions, the truly heterogeneous solution proposed by the HSA Foundation will allow fine-grained memory handling between the CPU and GPU devices requiring both devices to access the same coherent block of memory.

By allowing the programmer's application process to directly dispatch the computational work of the graphic processing unit to a per-application queue eliminates the typical overhead penalty of depending on the operation system's kernel services to dispatch the workload. HSA allows for per-application command queues which can handle computational dispatch requests while in User Mode rather than relying on the underlying hardware device driver to handle the overall workload queuing scheme. There are also complications regarding current GPGPU solutions related to situations where a process may essentially hog the graphic processing unit's hardware for an extended period of time disallowing other processes to utilize the hardware for their computation. This issue is addressed by preemptive GPU context switching in truly heterogeneous computing solutions.

4.3 Truly Heterogeneous Computing Solution and its Importance in Benefiting Modern Algorithms

As I will explore in future chapters with the basic *ad*-heap design philosophy, many existing algorithms and data structures may effectively benefit from a more tightly-coupled heterogeneous platform with the features presented by the HSA Foundation for a truly heterogeneous computing system. Issues which ultimately plague algorithms and operations which require frequent kernel launch and memory transfers when translated to graphic processing units will be alleviated with these new hardware and software solutions to ensure that only the performance benefits remain. Following the goal of heterogeneous computing to obtain a synergistic solution through optimally using all hardware components to the best of their abilities, HSA Foundation seeks to remove much of the limitations inherited by modern GPU technology to allow this goal to be both more easily obtained but also more natural to the application programmer((AMD Developer Central) (2013)).

CHAPTER 5

EXPLORING THE AD-HEAP-BASED BATCH K-SELECTION BEHAVIOR

Weifeng Liu and Brian Vinter's *ad*-heap data structure design is incredibly interesting because it presents a perfect case study for a data structure whose operations would typically not produce any performance benefit from being translated to the graphic processing unit. The only parallel portion of the top-down reconstruction of the heap is to find the maximum child on each level of children. Given this, fully utilizing or saturating the SIMD processing cores of the graphic processing unit is limited to the branch size(or d) of the heap being processed. In a truly heterogeneous platform, the heap data structure can be re-designed to effectively take advantage of the different processing components of the profile over the course of its operation. Therefore, the *ad*-heap structure is particularly interesting for exploring the potential benefits that a truly heterogeneous system could present for data structures and algorithms which require a much closely coupling and interaction between both the central processing unit and graphic processing unit.

5.1 Characteristics of the Batch *k*-Selection Algorithm

Following a similar example as presented in the *ad*-heap conference paper; I implemented the heap-based batch k-Selection algorithm. The batch k-Selection algorithm consists of processing a list set of sub-lists by constructing and assigning a heap for each sub-list within the set. The heaps that are constructed for the each sub-list are the size of k and constructed by inserting the first k elements within each sub-list into the Heap data structure. Once the heap is constructed from the first k items in the sub-list, each subsequent item within the sub-list is compared to the current root of the heap structure. If the

subsequent item's value is less than the current root node of the heap, than the root node of the heap is updated with that value and the next subsequent sub-list item is examined. Once all sub-lists have been processed, the root node of each heap is the kth order statistic of that corresponding sublist.

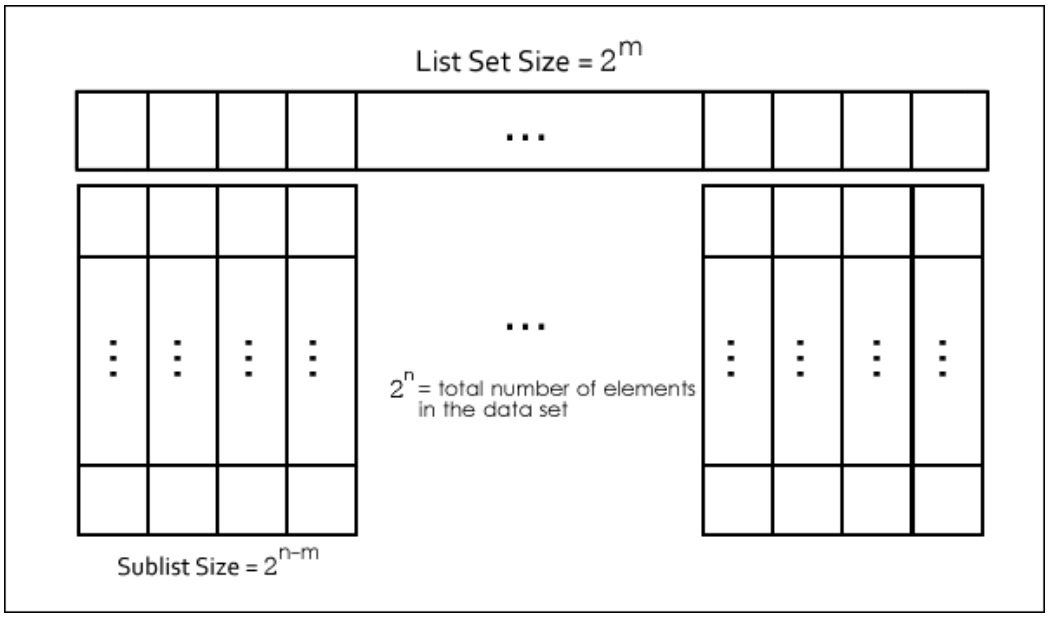


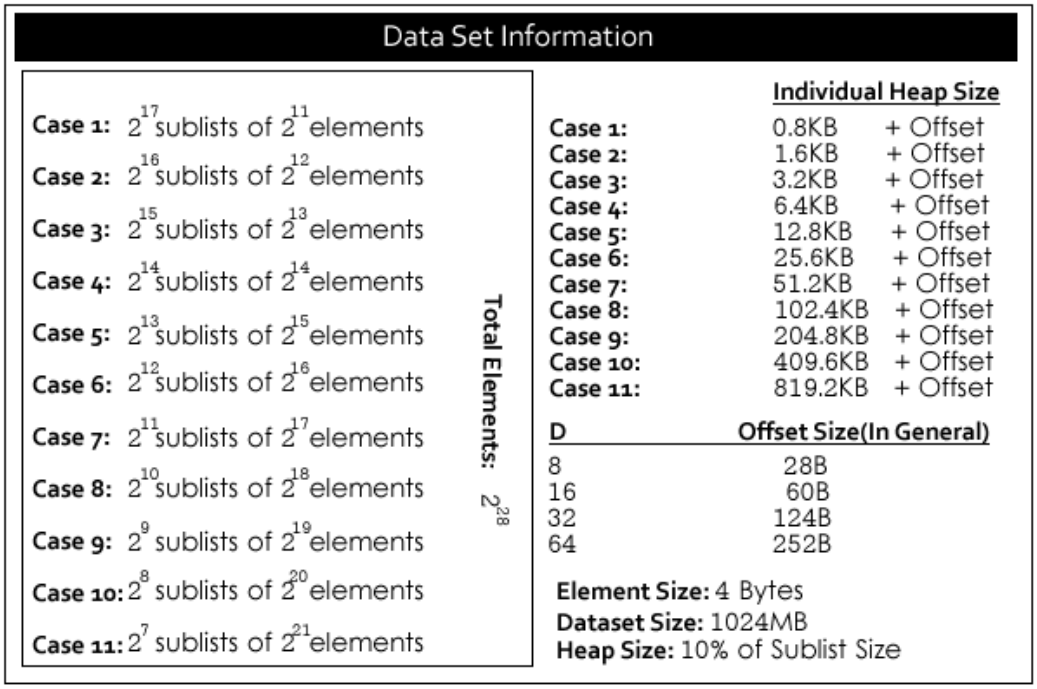
Figure 5.1. The Batch k-Selection algorithm finds the kth order statistic in each sub-list of the sub.

The batch k-Selection algorithm presented an interesting algorithm to examine the Heap data structure's performance. The mechanics of the algorithm ensures that only two basic operations of the Heap data structure is performed, but these operations are performed in high volume. The implementation of the fairly large data set for computational purposes was an array of unsigned integers segmented by the size of each sub-list. Given an array of size 2^n unsigned integers, this array would be further segmented into m sub-lists of size 2^{n-m}.

The versatility of this segmentation method allows the data set to be partitioned into different test cases by presenting test cases where the sub-list size is fairly small yet the number of sub-lists are much larger to test cases where the sub-list size is fairly large but

the number of sub-lists are much smaller without changing the overall data-set size. For my implementation, I set the global data set size to 2^{28} and the smallest sub-set size consisting of 2^{11} unsigned integers. The test cases ranged from 2^{17} sub-lists of size 2^{11} to 2^{7} of size 2^{21}. To keep the computational work fairly consistent through each case, I allow the heap size k to be $0.1l$ where l is the length of sub-list in that case. This methodology is the same as the experimental methodology presented in Weifeng Liu and Brian Vinter's *ad*-heap conference paper to test their simulation. The resulting data set information is shown in Figure 5.2.

Figure 5.2. Total data set size and the corresponding test cases and data size.



As described earlier, the batch k-selection algorithm is an extremely interesting algorithm to explore in relevance to the heap data structure given the large volume of similar operations performed when processing each sub-list. Given a sub-list of size l and $k = 0.1l$, you are guaranteed to perform k *Insert-Node* operations to construct the heap to process the sub-list and for each *Insert-Node* operation, the heap would be reconstructed from the bottom-up to restore the ordering property. The true volume of work is found in the numer-

ous *Update-Key* operations performed when fully processing the sub-list. Given a sub-list of size l and a heap of size k, you have an l-k elements remaining to be processed. This introduces an upper bound of l-k on *Update-Key* operations to fully process the sub-list with each operation performing a top-down reconstruction of the heap to restore the heap ordering property.

While processing the sub-list, the *Update-Key* operation would only be called to update the root node in cases where the current element in the sub-list being processed is smaller than the current root node of the heap. Therefore, it can be speculated that the upper bound previously mentioned would be extremely generous as the rate of *Update-Key* operations would essentially slow down significantly as the root of the heap structure progressively got smaller. The theoretical ad-heap top-down reconstruction computation would be performed with every *Update-Key* operation that replaces the root node with a smaller value; therefore understanding how many times the *Update-Key* operation is performed would promote a better understanding of the overall work of the ad-heap structure within the k-selection algorithm. Using the implicit d-heap implementation of the batch k-selection algorithm which recursively performs the heap reconstructions, I seek to find a consistent pattern behind these operations in each of the individual test cases of the data set by counting the number of skips(when the new value in the sub-list is greater than the root) and updates(where the new value is less).

Regardless of the size of the sub-list being processed, the percentage of Update-Key operations performed on the remaining elements of the sub-list remain the same. Only 25-26% of the elements within the l-k remainder of the sub-list will require an Update-Key operation while 75-74% will effectively be skipped. Despite this small percentage, as the sub-list size grows considerably, the number of Update-Key operations also grows considerably as well. In cases where the sub-list size is as large as 1048576 elements, this can amount to as many as 245,000 Update-Key operations. The above execution statistics were the average statistics of all sub-lists of the same size processed sequentially on a symmetric

multi-core central processor. This information is shown in Table 5.1.

Figure 5.3. As the sub-list is being processed, the number of updates diminish.

Batch K-Selection Algorithm Statistics			
Average Number Of Updates Per Sublists	Average Number Of Skips Per Sublists	K-Value	Remaining Elements
469(25.4%)	1374(74.6%)	204	1843
941(25.6%)	2745(74.4%)	409	3686
1885(25.6%)	5847(74.4%)	819	7372
3771(25.6%)	10974(74.4%)	1638	14745
7542(25.6%)	21949(74.4%)	3276	29491
15086(25.6%)	43896(74.4%)	6553	58982
30179(25.6%)	87785(74.4%)	13107	117964
60365(25.6%)	175564(74.4%)	26214	235929
120732(25.6%)	351127(74.4%)	52428	471859
241464(25.6%)	702254(74.4%)	104857	943718
482855(25.6%)	1404581(74.4%)	209715	1887436

Understanding the characteristics of the Batch k-Selection algorithm, we can already identify some common issues which would result in a problematic scenario with the current limitations of discrete graphic processing unit-based heterogeneous platforms.With nearly 500,000 Update-Key operations performed per sub-list resulting in nearly 500,000 iterations of the same memory transfer process and the combined overhead of 500,000 kernel launches, the algorithm would suffer some obvious issues on any modern GPGPU platform. To investigate this further, I implemented two CPU-based variations of the algorithms to be used for comparison metrics.

5.2 Introducing the Physical Experimental Platforms

Two available experimental platforms were utilized for both the performance comparison of the standard CPU d-heap implementation of the batch k-selection algorithm and also

GPU *ad*-heap implementation to understand the execution characteristics of each method in comparison to one another. Each experimental platform offers its own unique hardware-based characteristic which provided some interesting information behind each platform's individual execution model. The details of the two experimental platforms are provided in Table 5.3.

Figure 5.4. Physical Experimental Platform Specifications

Machine 1	Machine 2
CPU: AMD FX-8350 Black Edition	**CPU:** AMD A10-7850K APU
CPU Cores/Clock Rate/Architecture: 8 / 4.0GHZ / Vishera	**CPU Cores/Clock Rate/Architecture:** 4 / 3.7GHZ / Kaveri
CPU Peak Single Precision Throughput:	**CPU Peak Single Precision Throughput:**
CPU Thermal Design Power: 125w	**CPU Thermal Design Power:** 95w
System Memory/Channels/Bandwidth: 32GB / Dual / 1866MHZ	**System Memory/Channels/Bandwidth:** 16GB / Single / 1333MHZ
GPU: AMD HD-7970	**GPU:** AMD Radeon R7 Series
GPU Execution Units/Architecture: 32 / Tahiti	**GPU Execution Units/Architecture:** 8 / Spectre
GPU SIMD Cores/Clock Rate: 2048 / 925MHZ	**GPU SIMD Cores/Clock Rate:** 512 / 720 MHZ
GPU Peak Single Precision Throughput:	**GPU Peak Single Precision Throughput:**
GPU Local Memory: 32KB	**GPU Local Memory:** 32KB
GPU Memory/Bandwidth: 3072MB / 1.375GHZ – 264GB/sec	**GPU Memory/Bandwidth:** 1.82GB / 2.4GHZ
GPU Thermal Design Power: 300w	**GPU Thermal Design Power:** 45W/65W/95W
GPU Driver Version: 1348.5(VM)	**GPU Driver Version:** 1359.4(VM)
Operating System: CentOS 6.5	**Operating System:** Ubuntu 12.04
Compiler/Library: gcc version 4.4.7	**Compiler/Library:** gcc version 4.6.3
Implementation: C++ / OpenCL	**Implementation:** C++ / OpenCL

Machine 1 provides the basic discrete GPGPU experimental platform with the exceptionally powerful AMD HD-7970 discrete graphic processing unit while Machine 2 offers the latest in HSA-influenced hardware design while lacking the functional programming platform when provides many of the other HSA-influenced features such as the unified virtual memory space and addressing scheme and the pre-emptive context switching and user-mode dispatch command queues. Both machines feature powerful multi-core central processing units and enough global memory available on their respective devices to effectively store the entire large list set of sub-lists if needed.

Baseline statistics were provided by the serial execution of the batch k-selection algorithm where each sub-list within the list set was executed in a sequential manner. The sequential execution simply performed the *Insert-Key* operation for the first k elements within each sub-list while calling the *Update-Key* operation for every subsequent element where the subsequent value is less than the current root node's key value. The *Update-Key* operation would replace the current root node's key value with the subsequent element of lesser value and recursively call the top-down reconstruction function which sequentially searched for the maximum child amongst each level of the top-down propagation process. In addition to the test cases shown above, for each heap size, different d values were used which would effectively increase the computational load of both finding maximum child and traversing the height of the overall heap. When d is much smaller, less children must be examined to extract the maximum child but the height of the heap is much larger.

5.3 Basic CPU Implementation of Heap Operations

Algorithm 1 The control process of processing the sub-list of interest.

```
1: function HEAP_BUILD (main list index)
2:       heap = malloc(size of the heap)
3:       sublist index = main list index*size of the sublist
4:
5:       for i = 0, i goes to k do
6:            sublist value = main list set[sublist index+i]
7:            HEAP INSERT NODE(heap, sublist value)
8:       endfor
9:
10:      for i = k, i goes to the size of the sublist do
11:         sublist value = main list set[sublist index+i]
12:
13:         if(sublist value < root of heap)
14:              HEAP UPDATE KEY(heap, root, sublist value)
15:           endif
16:         endfor
17:
18:      return root of heap
19: endfunction
```

The *HEAP_BUILD* function is the central control function which effectively constructs the heap data structure and processes the sublist against the heap.

Algorithm 2 Execution details of the *Insert-Node* operation.

```
1: function HEAP INSERT NODE (*heap, sublist value)
2:
3:       heap[end of heap] = sublist value
4:
5:       if this is not the first node inserted
6:            HEAP BOTTOM UP(heap, end of heap index)
7:       endif
8:
9: endfunction
10: function HEAP BOTTOM UP  (*heap, heap index)
11:
12:      if heap index is not the root
13:           parent index = GET PARENT INDEX( )
14:
15:             if heap[parent index] < = heap[heap index]
16:                  temp node = heap[parent index]
17:                  heap[parent index] = heap[heap index]
18:                  heap[heap index] = temp node
19:                  HEAP BOTTOM UP(heap, parent index)
20:             endif
21:        endif
22:
23: endfunction
```

The *HEAP_BUILD* operation can be distinguished by two parts. The first part is the initial construction of the d/ad-heap data structure which requires the use of the Insert-Node

operation to insert the first k nodes into the allocated heap data structure. Once the heap has been fully constructed, the remaining sub-list is processed by calling the *Update-Key* operation sequentially while exhausting the sub-list until the kth smallest value is extracted.

The *Update Key* operation takes the heap data structure, the index of the node being updated, and the new key value which the node is being updated to and performs whichever heap property reconstruction process that is necessary given the respective relationship between the new key value and the original parent and children in relationship to that node. If the node being updated is not the root node then there is an equal opportunity for both the bottom-up and top-down reconstruction process. If the node is the root node, then we can assume that if the heap ordering property is not satisfied by the updated node's new key value then the node will propagate down the tree until the ordering property is restored. In the case of the batch k-selection algorithm, the root node is the only node being updated and each update will require the top-down propagation of the new key value.

As previously mentioned, the computational complexity of the *Update Key* operation is heavily dependent on the value of d. If the d value is small, than the sequential selection of the maximum child at each level during the downward propagation is small, yet there are more levels of the heap to effectively traverse in the duration of the propagation. The opposite is true for when the d value is large, but when the d value is large there is more opportunity for data parallelism. Much like the bottom up reconstruction process, the propagation process is called in a recursive manner until either the new node value has reached a suitable location within the heap or the new node has either approach the root(in the case of the bottom-up reconstruction) or approached the last level of the heap(in the case of the top-down reconstruction). Unlike the top-down reconstruction, in the bottom-up reconstruction there is no opportunity for parallel optimization since the process only consists of serial *compare-and-swap* operations to compare and swap the current node with its parent node if the current node is greater than the parent node.

Algorithm 3 Execution details of the *Update-Key* operation on the CPU.

```
1: function HEAP UPDATE KEY (*heap, index, sublist value)
2:
3:       heap[index] = sublist value
4:       first child index = GET FIRST CHILD INDEX( )
5:
6:       if index != 0
7:           parent index = GET PARENT INDEX ( )
8:
9:           if sublist value > heap[parent index]
10:                HEAP BOTTOM UP(heap, index)
11:          else
12:                if first child index < end of the heap
13:                     HEAP TOP DOWN(heap, index):
14:                endif
15:          endif
16:      else
17:           if first child index < end of the heap
18:                HEAP TOP DOWN(heap, index):
19:           endif
20:      endif
21:
22: endfunction
23: function HEAP TOP DOWN  (*heap, heap index)
24:
25:     first chld index = GET FIRST CHILD INDEX ( )
26:     max child = heap[first child index]
27:     max child index = first child index
28:
29:     for i = 2, i goes to d
30:          next child index = GET NEXT CHILD INDEX ( )
31:
32           if next child index < end of heap
33:
34:               if heap[next child index] > max child
35:                  max child = heap[next child index]
36:                  max child array index = next child index
37:               endif
38:
39:          endif
40:     endfor
41:
42:     if heap[heap index] < max_child
43:          heap[max child array index] = heap[heap index]
44:          heap[heap index] = max child
45:     endif
46:
47:     if first child index of the new node < end of heap
48:          HEAP TOP DOWN(max child array index)
49:     endif
50:
51:
52: endfunction
```

In the above described algorithms, much of the detailed arithmetic to ensure proper array indexing is abstracted away.

The methods described as *GET NEXT CHILD INDEX*, *GET FIRST CHILD INDEX*, and *GET PARENT INDEX* are simple equations based upon program execution-specific

details that are used to properly access the concrete representation of the heap data structure.

There are two fundamental CPU solutions to the batch k-selection algorithm using the central processing unit as the primary execution device. The first solution features the sequential processing of each sub-list in every possible test case. The second solution features a concurrent processing of groups of sub-lists by utilizing the POSIX Multi-threading API. Given a number of POSIX threads, the list set of sub-lists is divided by this number into smaller groups of sub-lists. Each sub-list in the group is processed in a sequential manner with each group processed by a separate POSIX thread concurrently. This is a larger scale data parallel solution to the batch k-selection algorithm but since the ISA is the same for each processing core of the multi-core processor, this does not qualify as uniquely heterogeneous. Aside from testing each of cases presented in Table 5.1, the value of d ranges from 8 to 64. The multi-threaded POSIX solution uses the same number of POSIX threads as there are physical cores on the processor. Therefore, Machine 1 dispatches eight POSIX threads and Machine 2 dispatches only four POSIX threads.

Before showcasing the results batch k-selection algorithm, I decided to test the total execution time for the Update-Key operation on the CPU using the recursive algorithm described in Algorithm 3. Each possible heap size was tested with each corresponding d-value ranging from 8 to 64. The test heap was generated by simply inserting nodes whose value reflected the current count of nodes inserted into the heap incremented by one. In the case where the heap size is 204, the root node(i.e., the maximum value within the heap) would be 204. This ensures that there is only non-zero distinct values within the heap; therefore the Update-Key would simply replace the root node to 0 ensuring that the new node of 0 would be propagated completely down the tree to the last level since it would be the smallest value within the heap. This testing mechanism ensures that the entire heap is traversed during the top-down reconstruction consistently across all test-cases, providing consistent results.

5.4 CPU Update-Key Operation Results

Figure 5.5. 8-Heap CPU Update Key Performance

CPU Update-Key Operation Performance			
D	Heap Size	Machine 1 CPU Wall Clock Execution Time(ms)	Machine 2 CPU Wall Clock Execution Time(ms)
8	204 Elements	0.000976562	0.000976562
8	409 Elements	0.000976562	0.000976562
8	819 Elements	0.000976562	0.000976562
8	1638 Elements	0.000976562	0.000976562
8	3276 Elements	0.000976562	0.000976562
8	6553 Elements	0.000976562	0.000976562
8	13107 Elements	0.000976562	0.000976562
8	26214 Elements	0.000976562	0.000976562
8	52428 Elements	0.000976562	0.000976562
8	104857 Elements	0.000976562	0.000976562
8	209715 Elements	0.000976562	0.000976562

Figure 5.6. 16-Heap CPU Update Key Performance

CPU Update-Key Operation Performance			
D	Heap Size	Machine 1 CPU Wall Clock Execution Time(ms)	Machine 2 CPU Wall Clock Execution Time(ms)
16	204 Elements	0.000976562	0.000976562
16	409 Elements	0.000976562	0.000976562
16	819 Elements	0.000976562	0.000976562
16	1638 Elements	0.000976562	0.000976562
16	3276 Elements	0.00119209	0.000976562
16	6553 Elements	0.000976562	0.000976562
16	13107 Elements	0.000976562	0.000976562
16	26214 Elements	0.000976562	0.00119209
16	52428 Elements	0.000976562	00.000976562
16	104857 Elements	0.000976562	0.00119209
16	209715 Elements	0.000976562	0.000976562

Figure 5.7. 32-Heap CPU Update Key Performance

CPU Update-Key Operation Performance			
D	Heap Size	Machine 1 CPU Wall Clock Execution Time(ms)	Machine 2 CPU Wall Clock Execution Time(ms)
32	204 Elements	0.000976562	0.000976562
32	409 Elements	0.000976562	0.000976562
32	819 Elements	0.000976562	0.000976562
32	1638 Elements	0.000976562	0.000976562
32	3276 Elements	0.000976562	0.00119209
32	6553 Elements	0.000976562	0.000976562
32	13107 Elements	0.000976562	0.000976562
32	26214 Elements	0.000976562	0.000976562
32	52428 Elements	0.000976562	0.000976562
32	104857 Elements	0.000976562	0.000976562
32	209715 Elements	0.000976562	0.000976562

Figure 5.8. 64-Heap CPU Update Key Performance

CPU Update-Key Operation Performance			
D	Heap Size	Machine 1 CPU Wall Clock Execution Time(ms)	Machine 2 CPU Wall Clock Execution Time(ms)
64	204 Elements	0.00119209	0.000976562
64	409 Elements	0.000976562	0.000976562
64	819 Elements	0.000976562	0.000976562
64	1638 Elements	0.000976562	0.000976562
64	3276 Elements	0.000976562	0.00119209
64	6553 Elements	0.000976562	0.00190735
64	13107 Elements	0.000976562	0.00119209
64	26214 Elements	0.000976562	0.00119209
64	52428 Elements	0.000976562	0.000976562
64	104857 Elements	0.000976562	0.000976562
64	209715 Elements	0.000976562	0.000976562

Without delving too deep into the individual statistics of each possible test case. You can see that the *Update-Key* operation is almost instantaneous for every possible test case of heap sizes and each possible of d. The d-heap CPU implementation of the *Update-Key* operation is extremely fast for performing the operation on one single heap. It isn't until the complete execution of the k-selection algorithm that the performance trend can be accurately observed.

5.5 CPU Batch k-Selection Algorithm Results

Next we test the serial CPU implementation of d-heap using all possible test cases and d values ranging from 8 to 64.

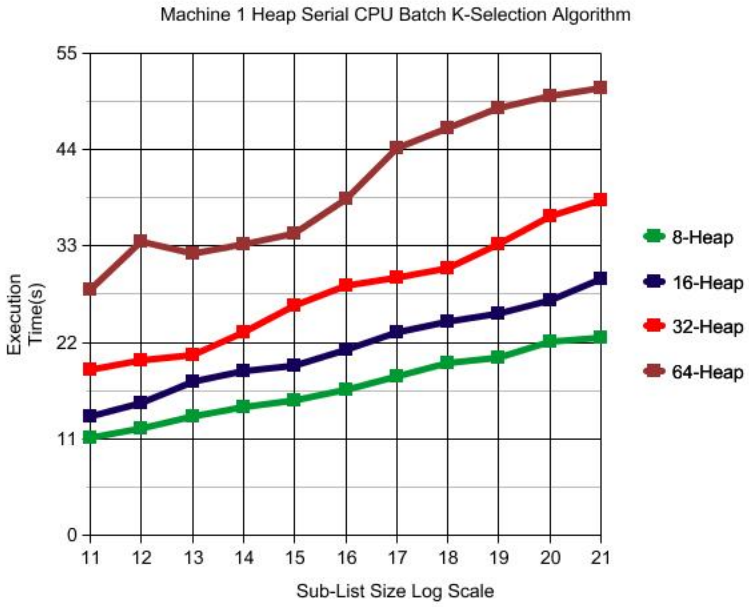


Figure 5.9. Machine 1 k-selection algorithm serially executed over all possible test cases.

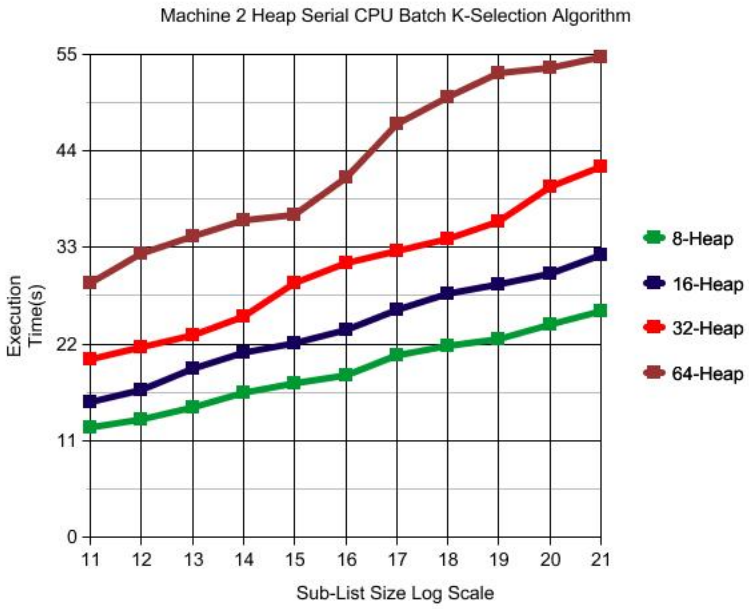


Figure 5.10. Machine 2 k-selection algorithm serially executed over all possible test cases.

The above graphs show that the larger values of d have a predictably slower execution time regardless of heap size. Even more-so, the performance gap between the 64-heap and the 32-heap is as large as over 50 % slower. This visual and empirical trend can be observed in the execution of both experimental platforms.

Next, I effectively tested and compared the multi-threaded batch k-selection algorithm CPU implementation and compared the execution wall-clock time to the serial execution. As noted before, the multi-threaded implementation utilizes the POSIX thread API by dispatching as many POSIX threads as there are physical cores on the device. The list set of sub-list is then segmented into groups of sub-lists based on the number of POSIX threads. Therefore four POSIX threads will relate to four sub-list groups and so forth. Each POSIX thread will process its group of sublists serially but all groups will be processing concurrently. Machine 1 dispatched 8 POSIX threads while Machine 2 dispatched 4 POSIX threads(Butenhof (1997)).

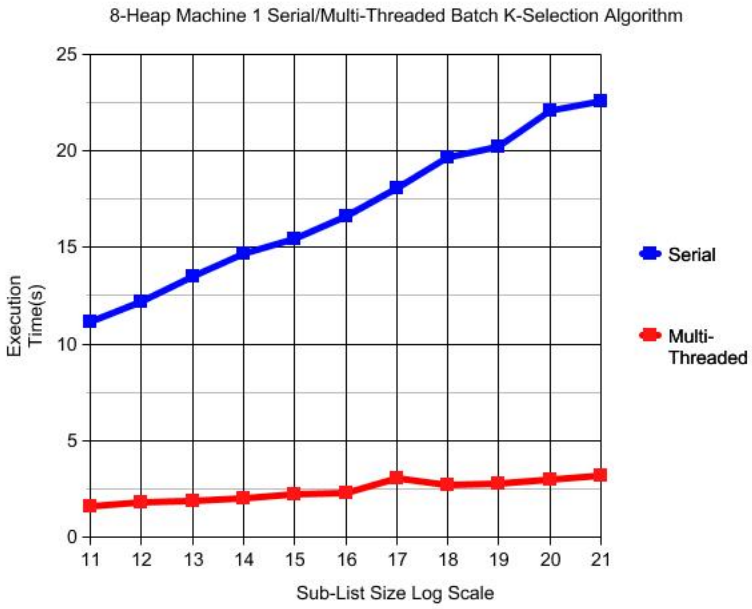


Figure 5.11. Machine 1 multi-threaded versus serially executed k-selection algorithm on the 8-heap.

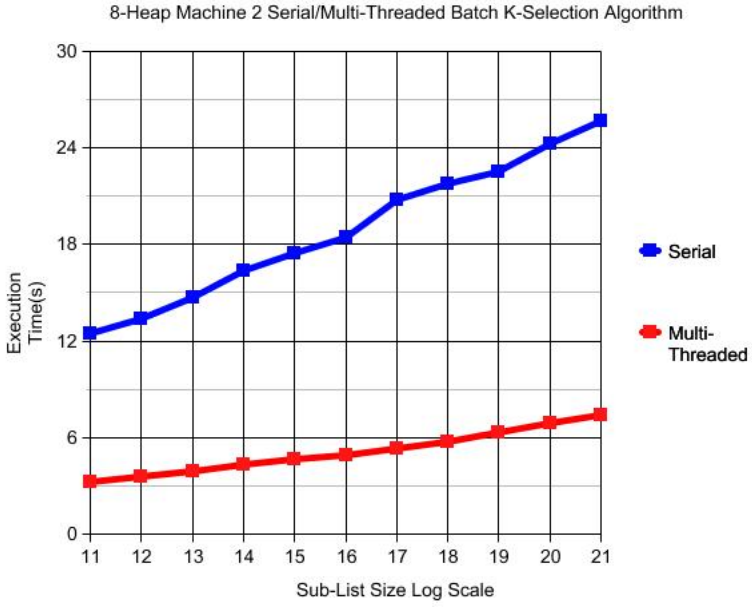


Figure 5.12. Machine 2 multi-threaded versus serially executed k-selection algorithm on the 8-heap.

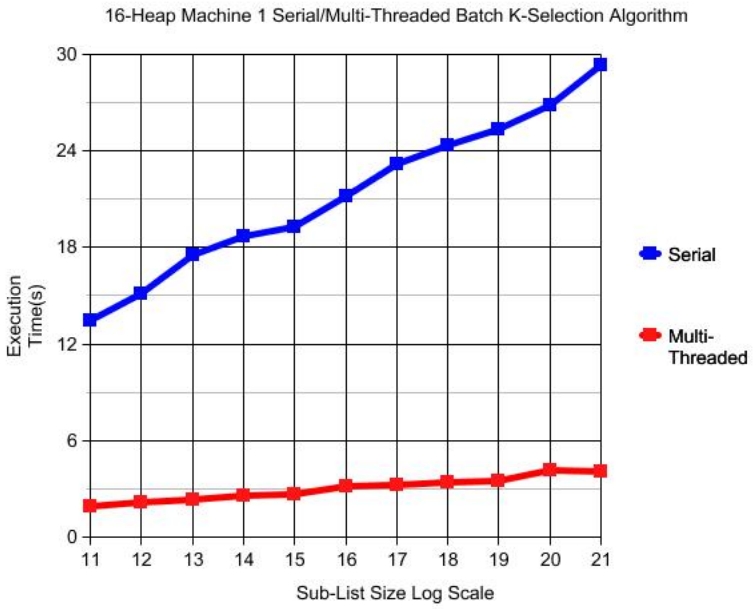


Figure 5.13. Machine 1 multi-threaded versus serially executed k-selection algorithm on the 16-heap.

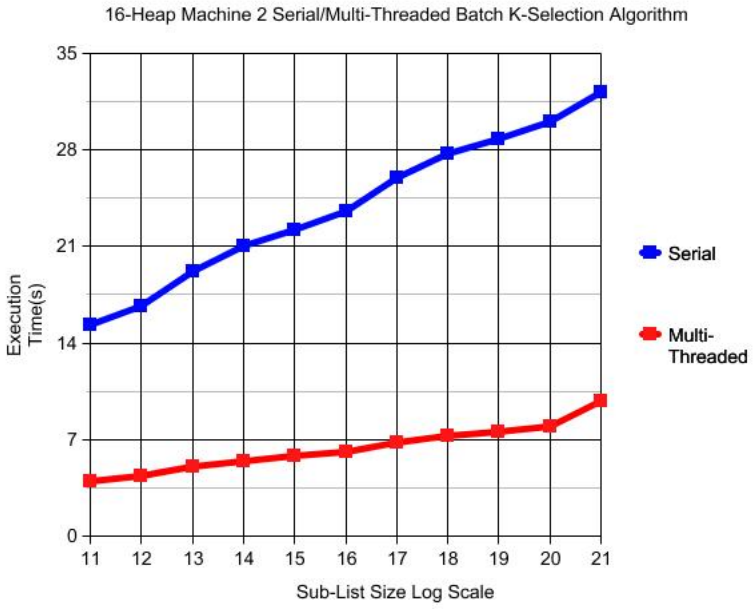


Figure 5.14. Machine 2 multi-threaded versus serially executed k-selection algorithm on the 16-heap.

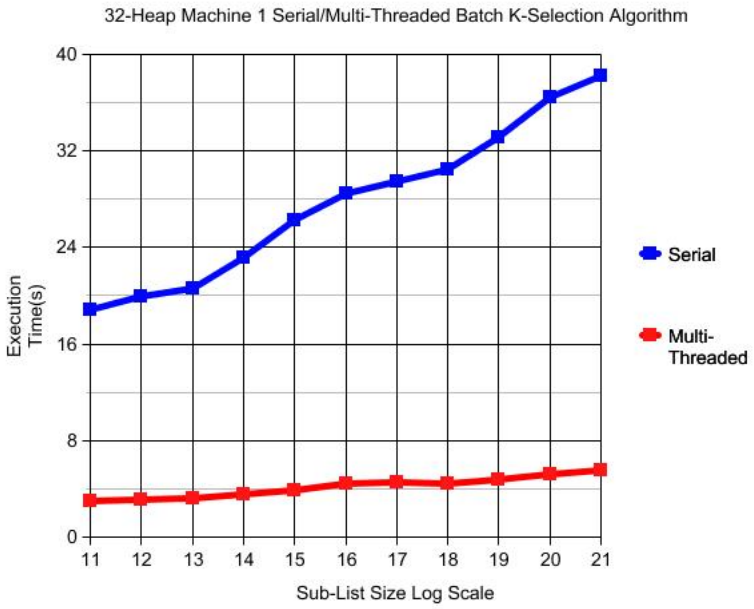


Figure 5.15. Machine 1 multi-threaded versus serially executed k-selection algorithm on the 32-heap.

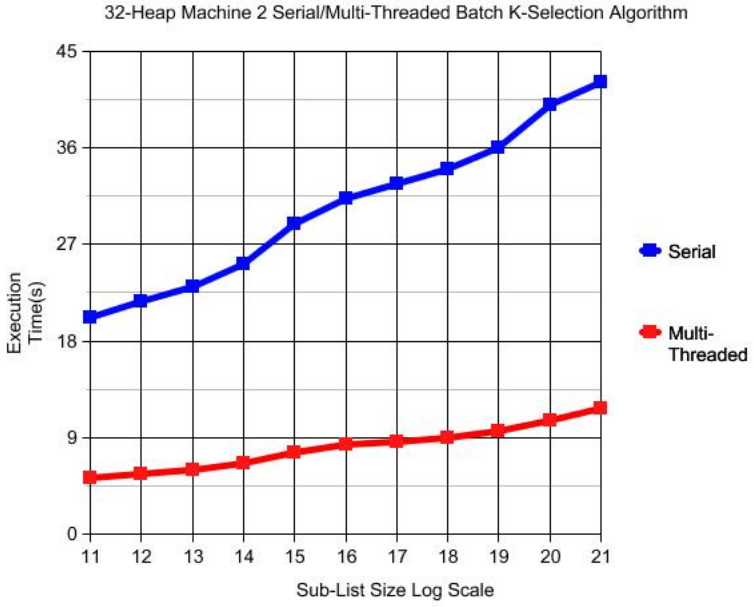


Figure 5.16. Machine 2 multi-threaded versus serially executed k-selection algorithm on the 32-heap.

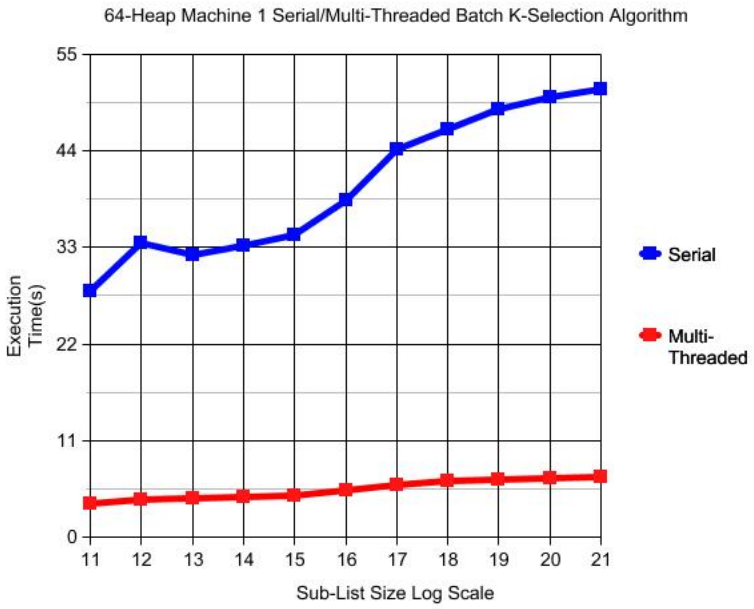


Figure 5.17. Machine 1 multi-threaded versus serially executed k-selection algorithm on the 64-heap.

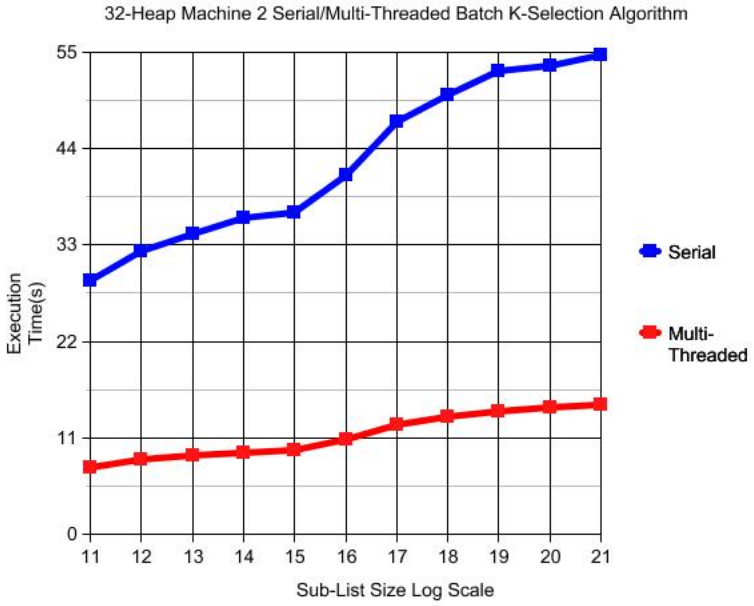


Figure 5.18. Machine 2 multi-threaded versus serially executed k-selection algorithm on the 64-heap.

Given the above figures, we can easily observe that the multi-threaded implementation outperforms the serial execution easily. Given the overall parallel nature of the k-selection algorithm which requires multiple heaps to be processed where no single heap is dependent

on the results of another heap, this high-level task parallelism approach to the multi-threaded implementation is very beneficial.

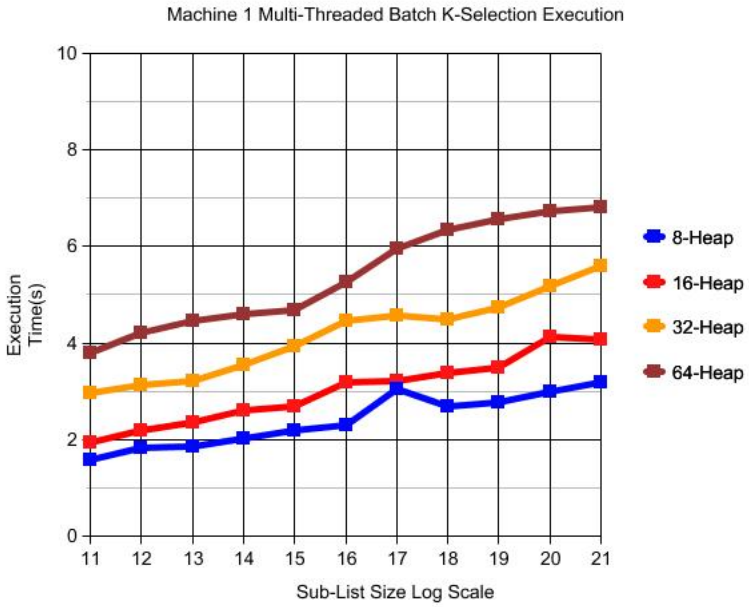


Figure 5.19. Comparing the execution time of different values of d using the multi-threaded implementation on Machine 1.

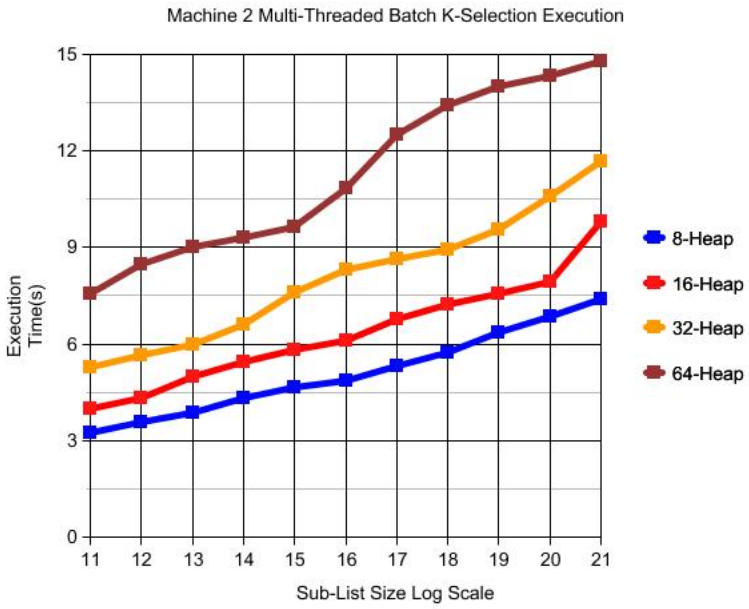


Figure 5.20. Comparing the execution time of different values of d using the multi-threaded implementation on Machine 2.

Comparing the execution time of each value of d over all test cases showcases the exact same trend that could be observed in the serial CPU experiments. For larger values of d, the performance of the algorithm ultimately suffers. Once again, this is predictable as the CPU computation serially performs the *FIND-MAXCHILD* computation regardless of the high-level task parallelism of groups of sub-lists being processed concurrently. Given these distinct observations regarding the limitations of the CPU implementations, we identified the opportunities for potential benefit of the GPGPU implementation of the ad-heap.

5.6 Exploring the *ad*-Heap Execution Characteristics on the GPU

Traditionally, the d-heap data structure did not reap same performance benefits that many other data structures enjoyed with the introduction of GPGPU computing. The d-heap displayed a very low level of data-parallelism in its operation. The ad-heap, designed for more tightly coupled asymmetric multi-processing systems, was designed to take advantage of both processing core designs within the asymmetric multi-core system by exploiting the very little potential for data-parallelism in the *FIND-MAXCHILD* operation and the serial

computation of the *COMPARE-AND-SWAP* operations during the top-down reconstruction.

The *ad*-heap *Update-Key* operation focuses on utilizing the head section of the implicit d-heap structure. The implicit bridge array is allocated in local memory for the given work-group. The implicit bridge structure, as shown in Chapter 3, stores the index-value pairs which the CPU will use for re-assignment. One thread is responsible for storing this information on the implicit bridge. On each iteration, as the new value is propagated down the heap structure, with a wavefront size of n and the value of d as the maximum count of child nodes; the child nodes are loaded into an allocated local memory area of size d in d/n wavefront transactions. A simple commutative max reduction scheme is performed to find the maximum child value in parallel. Once the maximum child is found, d threads find its corresponding index within the heap data structure. If the maximum value is greater than the new value, than the index-value pair is stored on the bridge. This process continues until the new value is relocated to a location within the heap that satisfies the heap ordering property or if all levels of the heap have been traversed.

Algorithm 4 ad-Heap Kernel Design for processing only one heap.

```
1: kernel function TCU_workload (__global *heap, d, shift_d, n, h, newv, i, head_offset, __local *scratch, __local*bridge)
2:
3:      unsigned int v = newv
4:      unsigned int index = i
5:      unsigned int end_of_heap = n+head_offset
6:      int global_index = GET-GLOBAL-THREAD-ID()
7:
8:
9:      if global_index == 0
10:          bridge[0] = 0
11:     endif
12:
13:     int first_child_array_index = ((index << shift_d)+1)+head_offset
14:
15:
16:     while first_child_array_index < end_of_heap
17:
18:         unsigned int maxi
19:         unsigned int maxv
20:
21:         unsigned int child_array_index = first_child_array_index+global_index
22:
23:         int local_index = GET-LOCAL-THREAD-ID()
24:
25:         if local_index < d
26:
27:             if child_array_index < end_of_heap
28:                 scratch[local_index] = heap[child_array_index]
29:             else
30:                 scratch[local_index] = -INFINITY
31:         endif
32
33:         LOCAL-MEM-THREAD-BARRIER
34:
35:         for offset = GET-WORKGROUP-SIZE >> 1, offset goes to 0, offset >>= 1
36:
37:             if local_index < offset
38:                 unsigned int other = scratch[local_index + offset]
39:                 unsigned int mine = scratch[local_index]
40:             endif
41:             LOCAL-MEM-THREAD-BARRIER
42:         endfor
43:
44:         maxv = scratch[0]
45:
46:         if local_index < d
47:             if heap[child_array_index] == maxv
48:                 scratch[0] = child_array_index-head_offset
49:             endif
50:         endif
51:
52:         LOCAL-MEM-THREAD-BARRIER
53:         maxi = scratch[0]
54:
55:         if maxv > v
56:
57:             if global_index == 0
58:                 bridge[(bridge[0] << 1)+1] = index
59:                 bridge[(bridge[0] << 1)+2] = maxv
60:                 bridge[0] = bridge[0] + 1
61:             endif
62:             index = maxi
63:         else
64:             break;
65:         endif
66:
67:         first_child_array_index = ((index << shift-d)+1)+head_offset
68:
69:     endwhile
70:
71:     if global_index == 0
72:
73:         bridge[(bridge[0] << 1)+1] = index
74:         bridge[(bridge[0] << 1)+2] = v
75:         bridge[0] = bridge[0]+1
76:     endif
77:
78:     if global_index < (h << 1)+1
79:         heap[global_index] = bridge[global_index]
80:     endif
81: endfunction
```

Once the propagation process is finished, the new value (which has resided in the private data memory space of the thread throughout this process) is added as the last index-value pair on the implicit bridge. The implicit bridge is offloaded to the heap data structure which resides in the global memory space. Once the heap data structure has been read back to the Host CPU device, the CPU performs the necessary re-assignments from the implicit bridge information. To avoid any unnecessary memory transactions from global memory which would result in an increased memory bandwidth, the majority of the work is done within the local memory space. Memory transactions from global memory generally handled in a regular accessing pattern, such as reading the corresponding children nodes from the heap data structure and writing the implicit bridge information back to the heap data structure.

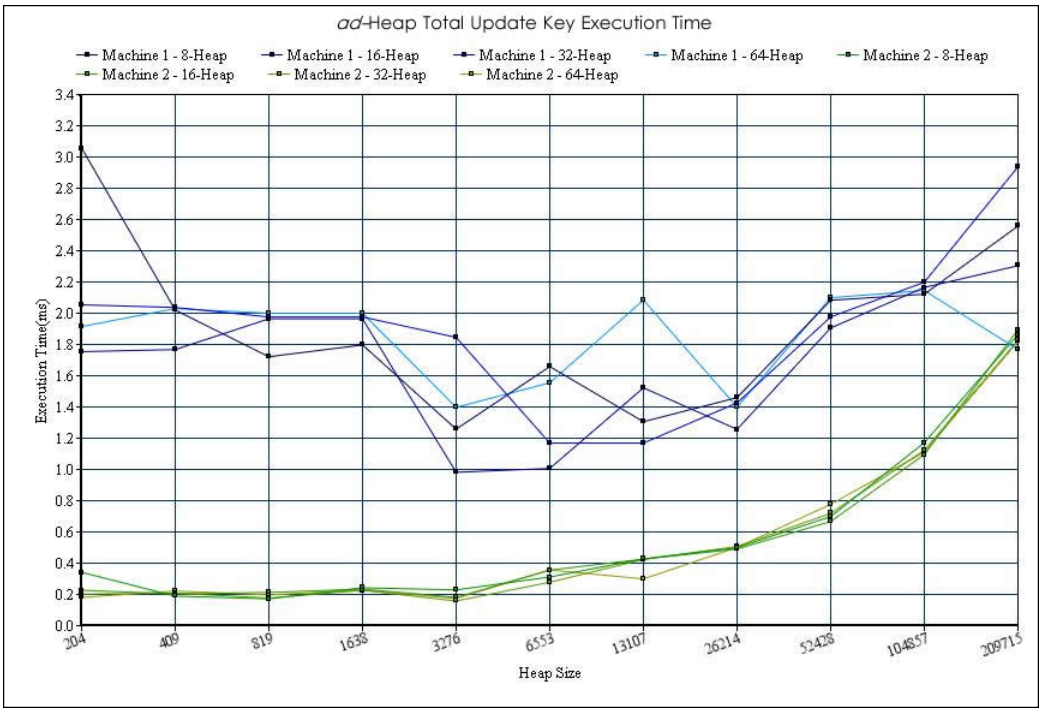


Figure 5.21. Comparing the total update execution time between both experimental platforms for all values of d

Figure 5.22. 8-Heap *ad*-Heap *Update-Key* Operation Performance

Single Heap TCU-Workload Update Performance Statistics - Total Update Performance(ms)			
D	Heap Size	Machine 1	Machine 2
8	204 Elements	3.05762	0.341634
8	409 Elements	2.02018	0.189046
8	819 Elements	1.72135	0.171468
8	1638 Elements	1.79696	0.242676
8	3276 Elements	1.25798	0.229004
8	6553 Elements	1.65869	0.311279
8	13107 Elements	1.30558	0.427734
8	26214 Elements	1.45972	0.499349
8	52428 Elements	2.08301	0.698568
8	104857 Elements	2.12158	1.17163
8	209715 Elements	2.55794	1.86239

Figure 5.23. 16-Heap *ad*-Heap *Update-Key* Operation Performance

Single Heap TCU-Workload Update Performance Statistics - Total Update Performance(ms)			
D	Heap Size	Machine 1	Machine 2
16	204 Elements	1.75407	0.226318
16	409 Elements	1.76742	0.203369
16	819 Elements	1.96427	0.212728
16	1638 Elements	1.2793	0.233887
16	3276 Elements	0.981689	0.181315
16	6553 Elements	1.00643	0.355632
16	13107 Elements	1.52254	0.426025
16	26214 Elements	1.25692	0.40641
16	52428 Elements	1.90568	0.666748
16	104857 Elements	2.16504	1.0953
16	209715 Elements	2.30404	1.89559

Figure 5.24. 32-Heap *ad*-Heap *Update-Key* Operation Performance

Single Heap TCU-Workload Update Performance Statistics - Total Update Performance(ms)			
D	Heap Size	Machine 1	Machine 2
32	204 Elements	2.05371	0.202067
32	409 Elements	2.03695	0.208008
32	819 Elements	1.97591	0.176921
32	1638 Elements	2.02197	0.226074
32	3276 Elements	1.84538	0.157959
32	6553 Elements	1.16895	0.278076
32	13107 Elements	1.16895	0.423665
32	26214 Elements	1.42594	0.506999
32	52428 Elements	1.97656	0.71932
32	104857 Elements	2.19857	1.12305
32	209715 Elements	2.9375	1.82715

Figure 5.25. 64-Heap *ad*-Heap *Update-Key* Operation Performance

Single Heap TCU-Workload Update Performance Statistics - Total Update Performance(ms)			
D	Heap Size	Machine 1	Machine 2
64	204 Elements	1.91471	0.18099
64	409 Elements	2.03003	0.22168
64	819 Elements	1.99967	0.197021
64	1638 Elements	1.82568	0.22762
64	3276 Elements	1.39762	0.174561
64	6553 Elements	1.55461	0.356771
64	13107 Elements	2.08431	0.429932
64	26214 Elements	1.39893	0.501058
64	52428 Elements	2.09937	0.776611
64	104857 Elements	2.14526	1.11108
64	209715 Elements	1.76807	1.81925

As can be observed in Tables 5.8 - 5.11, Machine 2 (the APU experimental platforms) performs the *Update-Key* operation much quicker than Machine 1 (the discrete GPU platform). Both platforms generally do not perform the *Update-Key* operation very efficiently in comparison to the CPU recursive implementation. Machine 2's performance begins to degrade as the heap size becomes increasingly large. As we explore the different characteristics of the execution of the *ad*-heap *Update-Key* operation we begin to see the limitations of this implementation.

First, we observe the kernel execution time between the two experimental platforms. The wall-clock performance metric is captured by two methods; the traditional Linux *gettimeofday()* API and OpenCL's own event profiling API.

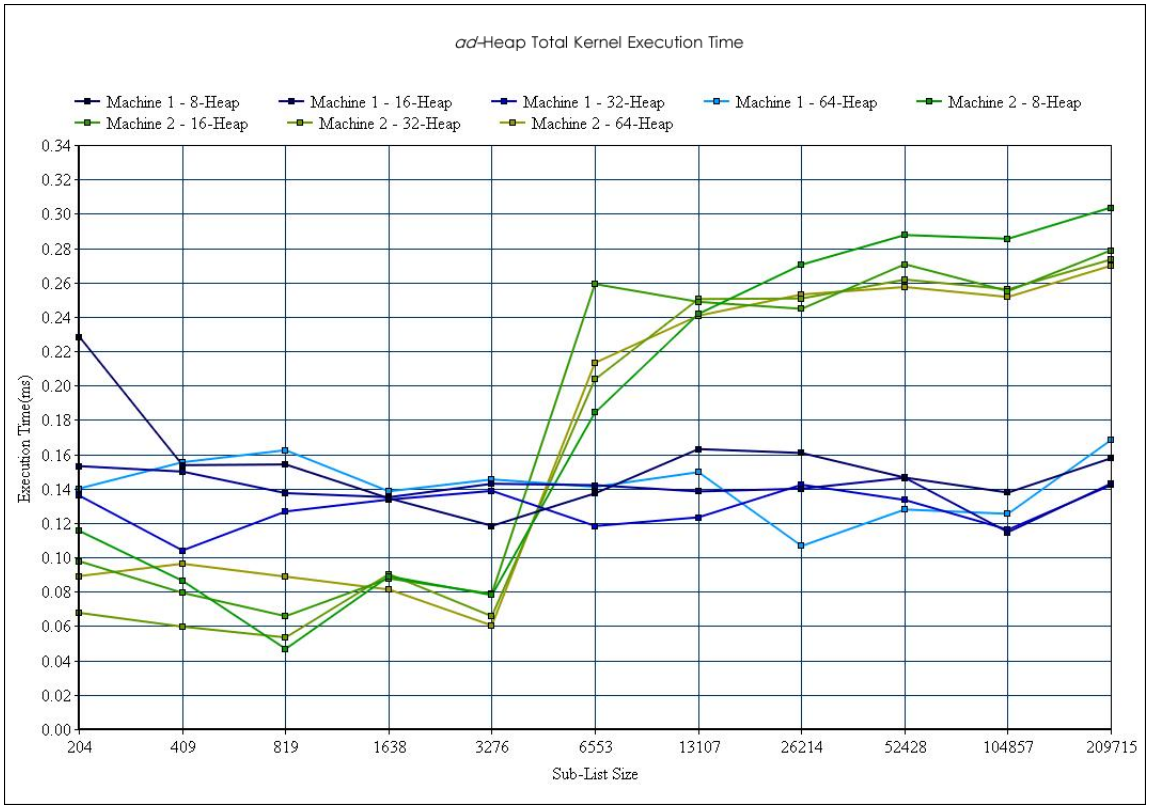


Figure 5.26. Comparing the total kernel execution time between both experimental platforms for all values of d

We see that the wall-clock execution time of the kernel program is relatively comparable between both platforms. While Machine 1's kernel execution time is fairly steady throughout all test cases, there is a fairly drastic performance loss for Machine 2 for increasing sizes of the heap starting at a heap size of 6553.

Figure 5.27. 8-Heap *ad*-Heap Kernel Execution Times

Single Heap TCU-Workload Update Performance Statistics - Kernel Execution Time Profile | Wall-Clock(ms)			
D	**Heap Size**	**Machine 1**	**Machine 2**
8	204 Elements	0.0264687 | 0.228353	0.03588 | 0.115397
8	409 Elements	0.0238023 | 0.153646	0.0377967 | 0.0863444
8	819 Elements	0.0280493 | 0.154053	0.0198403 | 0.0467122
8	1638 Elements	0.0275553 | 0.134359	0.0413347 | 0.0889486
8	3276 Elements	0.027407 | 0.118327	0.040009 | 0.0780436
8	6553 Elements	0.0323457 | 0.137288	0.130089 | 0.184326
8	13107 Elements	0.032394 | 0.163005	0.164996 | 0.241699
8	26214 Elements	0.0334817 | 0.160726	0.187823 | 0.270345
8	52428 Elements	0.0378763 | 0.146403	0.199464 | 0.287598
8	104857 Elements	0.0378763 | 0.137695	0.19919 | 0.2854
8	209715 Elements	0.0389137 | 0.157715	0.207559 | 0.30363

Figure 5.28. 16-Heap *ad*-Heap Kernel Execution Times

Single Heap TCU-Workload Update Performance Statistics - Kernel Execution Time Profile | Wall-Clock(ms)			
D	**Heap Size**	**Machine 1**	**Machine 2**
16	204 Elements	0.020395 | 0.153076	0.0381577 | 0.0977376
16	409 Elements	0.0261233 | 0.149902	0.031016 | 0.0793457
16	819 Elements	0.0261727 | 0.13737	0.0238387 | 0.0657552
16	1638 Elements	0.024938 | 0.13501	0.410797 | 0.0877279
16	3276 Elements	0.025926 | 0.142822	0.0474657 | 0.0786947
16	6553 Elements	0.030815 | 0.142008	0.196068 | 0.259033
16	13107 Elements	0.0312593 | 0.138346	0.172981 | 0.248698
16	26214 Elements	0.0309137 | 0.139974	0.175361 | 0.244629
16	52428 Elements	0.0318513 | 0.146322	0.183702 | 0.270589
16	104857 Elements	0.03758 | 0.114665	0.1854 | 0.254964
16	209715 Elements	0.0363953 | 0.142985	0.186836 | 0.278727

Figure 5.29. 32-Heap *ad*-Heap Kernel Execution Times

Single Heap TCU-Workload Update Performance Statistics - Kernel Execution Time Profile | Wall-Clock(ms)			
D	**Heap Size**	**Machine 1**	**Machine 2**
32	204 Elements	0.0210863 | 0.135905	0.0205733 | 0.0677083
32	409 Elements	0.021037 | 0.104004	0.0183357 | 0.0596517
32	819 Elements	0.022469 | 0.126709	0.0155227 | 0.0533854
32	1638 Elements	0.027802 | 0.133626	0.041528 | 0.0902507
32	3276 Elements	0.026321 | 0.138672	0.0389993 | 0.0656738
32	6553 Elements	0.0278027 | 0.118245	0.145553 | 0.203695
32	13107 Elements	0.0284447 | 0.123291	0.174023 | 0.250326
32	26214 Elements	0.0288893 | 0.142253	0.169761 | 0.250651
32	52428 Elements	0.0349133 | 0.133382	0.174106 | 0.261637
32	104857 Elements	0.0346173 | 0.11613	0.174415 | 0.256022
32	209715 Elements	0.0351107 | 0.14209	0.181859 | 0.273356

Figure 5.30. 64-Heap *ad*-Heap Kernel Execution Times

Single Heap TCU-Workload Update Performance Statistics - Kernel Execution Time Profile | Wall-Clock(ms)			
D	**Heap Size**	**Machine 1**	**Machine 2**
64	204 Elements	0.023901 | 0.140055	0.0385893 | 0.0891113
64	409 Elements	0.0248397| 0.155355	0.031475 | 0.0962728
64	819 Elements	0.023753 | 0.162354	0.0379923 | 0.0887044
64	1638 Elements	0.023062 | 0.138428	0.034892 | 0.0812988
64	3276 Elements	0.024938 | 0.145345	0.0319807 | 0.0603027
64	6553 Elements	0.0296297 | 0.141032	0.133398 | 0.213298
64	13107 Elements	0.030074 | 0.149577	0.162556 | 0.240641
64	26214 Elements	0.0305183 | 0.106608	0.169645 | 0.253011
64	52428 Elements	0.0321483 | 0.127848	0.173246 | 0.257324
64	104857 Elements	0.032 | 0.125326	0.172687 | 0.251546
64	209715 Elements	0.0320493 | 0.168376	0.176323 | 0.269775

With fairly comparable kernel execution times, we investigate another general component of the traditional OpenCL execution process; the memory handling aspect. With each *Update-Key* operation, the kernel takes a OpenCL memory object as an argument which stores the heap data structure being processed. This information is either written as a buffer object to the device and effectively read back(as is the case with Machine 1) or the buffer is mapped/unmapped to device memory(as is the case with Machine 2). Machine 1, being a discrete GPU based experimental platform handles all data transfers along the PCI-E bus while Machine 2, being an accelerated processing unit bypasses this transfer protocol by sharing the same system memory with the CPU.

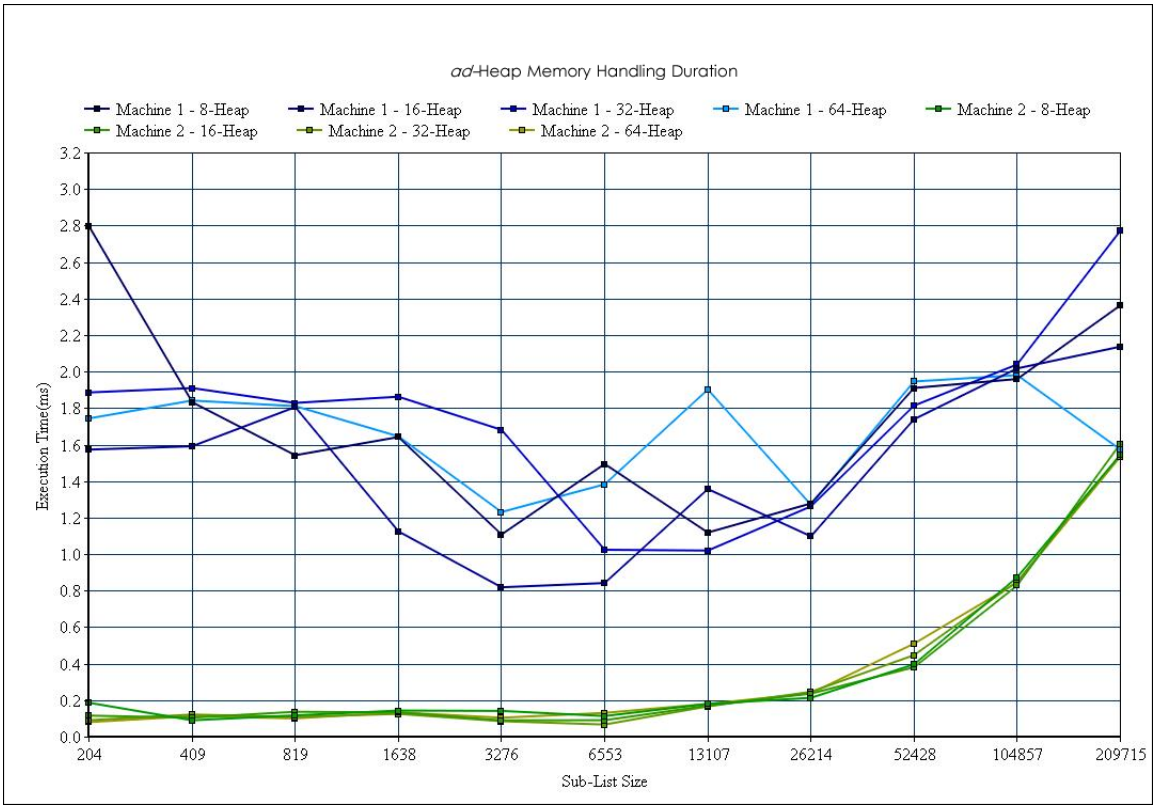


Figure 5.31. Comparing the total memory handling time between both experimental platforms for all values of d

Figure 5.32. 8-Heap ad-Heap Memory Handling Times

Single Heap TCU-Workload Update Performance Statistics - Memory Handling(Read/Write)			
D	**Heap Size**	**Machine 1**	**Machine 2**
8	204 Elements	2.79435	0.185628
8	409 Elements	1.83211	0.0895996
8	819 Elements	1.54142	0.116455
8	1638 Elements	1.64103	0.142985
8	3276 Elements	1.10742	0.140951
8	6553 Elements	1.49373	0.112956
8	13107 Elements	1.11808	0.179525
8	26214 Elements	1.27743	0.212646
8	52428 Elements	1.90999	0.39624
8	104857 Elements	1.95907	0.873617
8	209715 Elements	2.36271	1.54419

Figure 5.33. 16-Heap *ad*-Heap Memory Handling Times

Single Heap TCU-Workload Update Performance Statistics - Memory Handling(Read/Write)			
D	**Heap Size**	**Machine 1**	**Machine 2**
16	204 Elements	1.57235	0.115234
16	409 Elements	1.59058	0.104004
16	819 Elements	1.80509	0.136312
16	1638 Elements	1.12435	0.133057
16	3276 Elements	0.818197	0.0892741
16	6553 Elements	0.84139	0.0900065
16	13107 Elements	1.35628	0.169678
16	26214 Elements	1.09733	0.236003
16	52428 Elements	1.73869	0.38029
16	104857 Elements	2.01725	0.828288
16	209715 Elements	2.13631	1.60392

Figure 5.34. 32-Heap *ad*-Heap Memory Handling Times

Single Heap TCU-Workload Update Performance Statistics - Memory Handling(Read/Write)			
D	**Heap Size**	**Machine 1**	**Machine 2**
32	204 Elements	1.88542	0.0900065
32	409 Elements	1.90934	0.121012
32	819 Elements	1.82804	0.108073
32	1638 Elements	1.86206	0.124756
32	3276 Elements	1.6805	0.0846354
32	6553 Elements	1.02352	0.0664062
32	13107 Elements	1.01937	0.16569
32	26214 Elements	1.26131	0.245443
32	52428 Elements	1.8134	0.446696
32	104857 Elements	2.03923	0.850098
32	209715 Elements	2.77067	1.54289

Figure 5.35. 64-Heap *ad*-Heap Memory Handling Times

Single Heap TCU-Workload Update Performance Statistics - Memory Handling(Read/Write)			
D	**Heap Size**	**Machine 1**	**Machine 2**
64	204 Elements	1.74316	0.0795898
64	409 Elements	1.84106	0.112223
64	819 Elements	1.81038	0.100016
64	1638 Elements	1.64502	0.133382
64	3276 Elements	1.23039	0.103597
64	6553 Elements	1.38208	0.130371
64	13107 Elements	1.9012	0.179281
64	26214 Elements	1.26937	0.238118
64	52428 Elements	1.94539	0.509684
64	104857 Elements	2.9799	0.845215
64	209715 Elements	2.57227	1.53337

The absence of the PCI-E showcases obvious benefits on Machine 2's execution of the *Update-Key* operation. Despite both experimental platforms generally having comparable kernel execution times, the amount of time spent reading and writing the memory buffer objects on Machine 1 ultimately results in the decreased performance in comparison to Machine 2. In all cases on both platforms, the CPU portion of the *ad*-heap *Update-Key* is consistently instantaneous and does not affect the overall *Update-Key* performance.

Only when the heap size is at its greatest of all possible test cases, does the memory handling aspect of the *Update-Key* operation on both experimental platforms begin to share similar performance. A few distinct conclusions can be asserted from the above information that has resulted from the above experiments. The *ad*-heap implementation performance on one single heap does not compare to the basic *d*-heap implementation on both the serial CPU and multi-threaded CPU implementations. Though, there is an opportunity for parallelism when finding the maximum child nodes, this opportunity does not present enough parallelism to effectively saturate the compute units of the graphic processing unit.

When performing the *Update-Key* operation, the kernel function only requires enough work-items to perform the parallel reduction scheme on the branch of children and offload the implicit bridge to global memory. Therefore, in all cases, only (2**height of the heap*)+1 work-items are required. In a device with multiple compute units and hundreds to thousands of SIMD processing cores, this number of work-items completely under-utilizes the graphic processing unit's hardware.

Therefore, we can conclude that given a algorithm or application which utilizes only one single heap or a relatively small number of heap data structures; the *ad*-heap structure is not necessarily useful. The CPU recursive implementation of the *d*-heap will always outperform the *ad*-heap in this case, but this isn't strange or unusual. The *ad*-heap is better suited for cases where the algorithm or application exhibits a large amount of task-parallelism, where a large number of heaps need to be processed concurrently. The batch *k*-selection algorithm is a perfect example of such an algorithm.

5.7 Exploring the Batch *ad*-Heap Execution Characteristics on the GPU

We modify the implementation of the *ad*-heap based *Update-Key* operation such that multiple heaps can be offloaded to the device and processed in parallel. In this case, we allow each heap to handled by a separate work-group where each work-group consists of (2**height of the heap*)+1 work-items. Instead of a single heap data structure array passed to kernel function, a much larger array consisting of the batch of heaps is passed to the kernel function. The work-group ID is used to effectively index into this batch heap array. Along with the batch array of heap data structures, two additional arrays are also passed to the kernel function. These additional arrays correspond the new values(which are used in the update) and their corresponding index values(this is always zero since we're always replacing the root). These two additional arrays also correspond the batch size where a batch of n heaps will have two "new value" and "index of node to be updated" arrays of size n. Given a heap size of m, the batch heap array will be of size $m*n$.

This implementation is much different because it features both the data parallelism of finding the maximum child during the top-down reconstruction, which is absent from the CPU d-heap implementation, and it also features the higher level task-parallelism of concurrently processing the heap data structures. While, the multi-threaded CPU implementation is capable of processing groups of heaps concurrently; it is effectively limited by the number of physical cores on the CPU device. The GPU is limited by the number of compute units, but features a much larger number of compute units than the CPU has physical processing cores. The batch value corresponds to the number of sublists in the test case being processed. In implementing the batch k-selection algorithm with the batch *ad*-heap kernel, there were a few precautions that had to be taken to ensure that the algorithm performed accurately.

Each sub-list is processed at a different rate. For the *Update-Key* kernel to be launched, the "new value" array must have a valid new value for each sub-list being pro-

cessed. When a sub-list has been exhausted, it simply submits its current root as the new value in the "new value" array. The algorithm execution continues until all sub-lists have been exhausted, therefore the number of kernel launches or *Update-Key* operations is equal to whichever sub-list requires the most updates. This marching execution pattern ensures that only valid *Update-Key* kernel launches are called which enforces a minimum number of kernel launches equal to the number of which sublist requires the most *Update-Key* operations. By issuing more work-groups and more work-items, there is much more hardware utilization in the batch *ad*-heap implementation than the previous implementation which handled only one heap at a time. This implementation also provided insight on different hardware/software characteristics of each experimental platform.

Figure 17 shows some very interesting results for the *Update-Key* operation between both platforms. Machine 2 has significantly worse performance than Machine 1 for smaller heap sizes and a larger batch of heaps. Machine 1 has much better performance for test cases where there are 131072, 65536, and 32768 heaps in a batch. At the case of 16384 heaps of size 1638, both platforms perform roughly the same. From the test cases where there is 8192 heaps of size 3276, Machine 2 proceeds to showcase better performance.

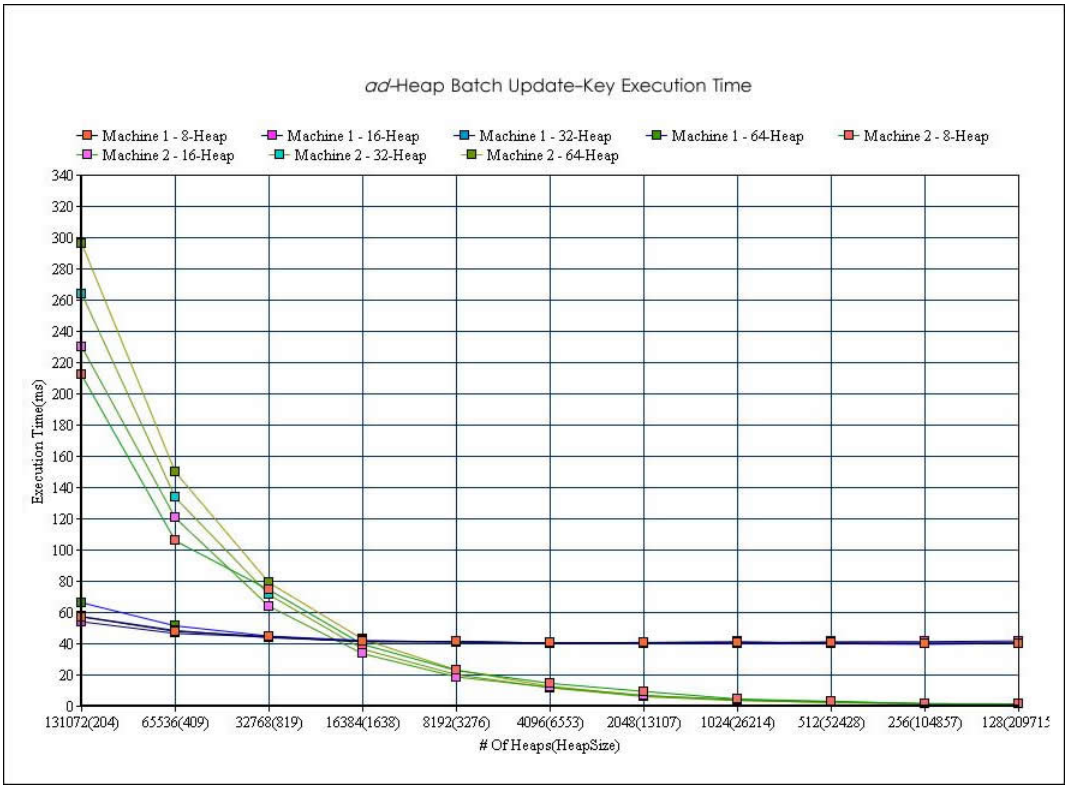


Figure 5.36. Comparing the total *Update-Key* operation execution times between both experimental platforms for all values of d

The observable trend of both platforms display the very characteristics of their hardware design. When given a large amount of work-groups, the computational power of Machine 2(the "Kaveri" accelerated processing unit) is more limited in comparison to Machine 1(the discrete GPU). Machine 2 features only eight compute units while Machine 1 features 32 compute units. Each compute unit concurrently processes the workload of one work-group. The Batch *Update-Key* *ad*-Heap issues as many work-groups as there are heaps in the current batch. Therefore, issuing p work-groups on a device of q compute units, would ensure that each compute unit is processing p/q work-groups sequentially.

Therefore Machine 2 lacks the sheer computational power to outperform Machine 1. This doesn't explain how Machine 2 begins to outperform Machine 1 after the number of heaps begins to decrease since both Machines' device hardware is being fully utilized and the compute units are fully saturated in all possible test cases. Therefore, we explore the kernel

execution time for both platforms to really understanding the computational performance between both platforms.

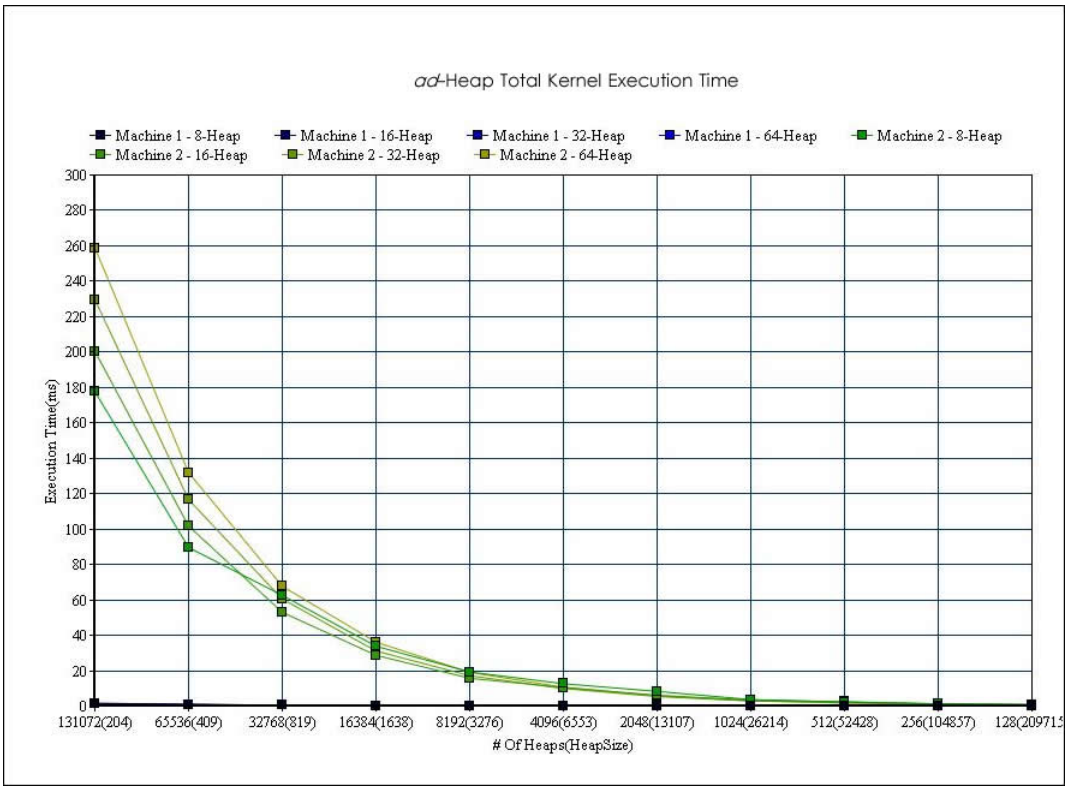


Figure 5.37. Comparing the kernel execution times between both experimental platforms for all values of d

Now we begin to see that the computational power of the discrete graphic processing unit outperforms the APU-based platform for almost all possible test cases. Kernel execution performance becomes similar in the last three test cases where we have a much smaller number of work-groups being issued(512,256,128). Therefore, we can conclude that the performance of the APU platform is effectively limited by its number of compute units in comparison to the Discrete-GPU based platform.

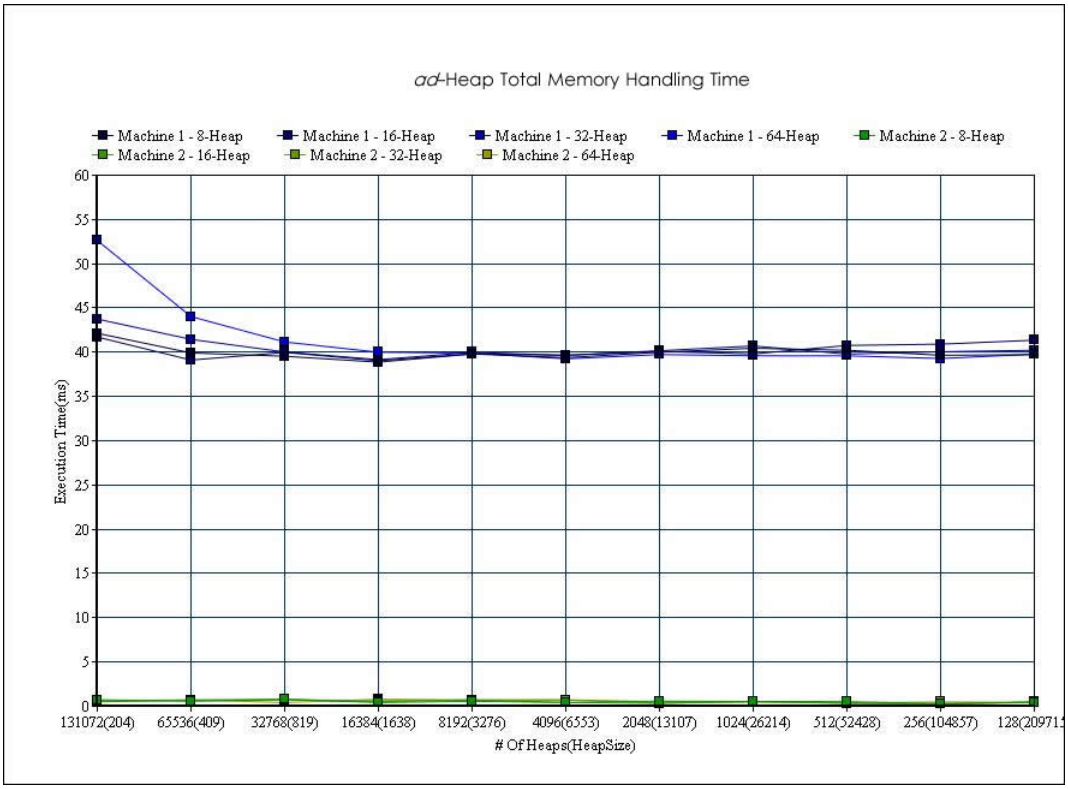


Figure 5.38. Comparing the memory handling times between both experimental platforms for all values of d

The point of divergence between both platforms, where Machine 2 begins outperforming Machine 1 can be attributed to the total memory handling time for both platforms. For each *Update-Key* operation in each test case, both platforms are handling writing and reading around the same amount of information; a batch array of heaps roughly the size of $0.1*2^{28}$ and two arrays whose size is the batch number. Therefore, the memory handling time is relatively the same across all test cases. Machine 2 features a significantly better memory handling time in comparison to Machine 1 similar to the Single Heap implementation.

Therefore, despite the computational performance difference between the two platforms; once the work-group size becomes smaller, the unified memory space of Machine 2 essentially presents the difference in performance between the two platforms.

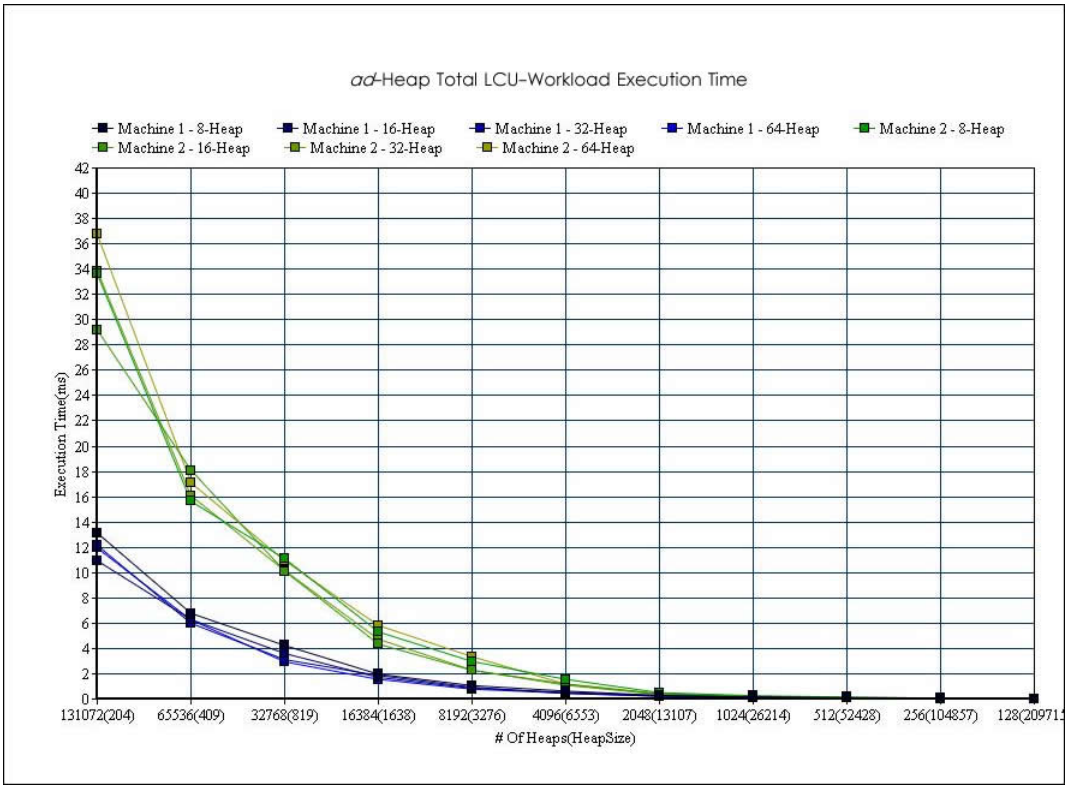


Figure 5.39. Comparing the LCU-Workload times between both experimental platforms for all values of d

Comparing the workload of the central processing unit to perform the re-assignment in all test cases show a similar trend to the one that can be observed in the total kernel execution graph.

In the batch *ad*-heap *Update-Key* operation, the POSIX thread API is used once again. In this case, there are as many POSIX threads dispatched as there are heaps in the batch. Each POSIX thread has a set of responsibilities in the execution the *Update-Key* operation. Each POSIX thread is responsible for building the *ad*-heap for its corresponding sub-list that it is processing and also loading the subsequent new values off the sub-list and performing the re-assignment for the heap data structure based on the implicit bridge information for that heap after the *Update-Key* kernel finishes. Only one POSIX thread is responsible for launching the kernel, but must wait until all other POSIX threads have finished their satisfying their responsibilities of populating the batch heap array and the two

corresponding "new value" and "index of nodes to be updated" arrays.

We have outlined many of the characteristics of the two experimental hardware platforms with both implementations of the *ad*-heap based batch k-selection algorithm. Both platforms have their hardware-specific trade-offs in terms of memory bandwidth and total computational performance. Though, neither experimental platforms perfectly model the HSA truly heterogeneous computing system which the *ad*-Heap data structure is designed to effectively execute upon. We are able to effectively simulate in a more practical manner, the batch k-Selection algorithm based upon the *ad*-Heap data structure enough to analyze the different aspects of the design of data structure itself and which problems are inherent to the data structure's general design and which problems we can resolve with current advancing technology such as the HSA-based Truly Heterogeneous Computing System. Therefore, the experimental cases do offer interesting insight into the execution behavior and other components of the *ad*-Heap structure and its corresponding operations.

CHAPTER 6

UNDERSTANDING AND COMPARING THE AD-HEAP-BASED BATCH K-SELECTION BEHAVIOR

The *ad*-heap data structure's general structural design and the design of the mechanics of its operations presents a very interesting case study for adapting familiar data structures and algorithms to take advantage of the modern technology concept described a truly heterogeneous computing system(THC). The *ad*-heap structure is completely dependent on the tightly coupled arrangement of both devices(the latency-oriented central processing unit and the throughput-oriented graphic processing unit). By implementing the *ad*-heap data structure on both of the previous experimental platforms, we are able to effectively see these active dependencies and how they may or may not limit the abilities and performance benefits of the *ad*-heap data structure.

The *ad*-heap data structure is reliant in its implementation on cases where the value of d is increasingly large, the larger value of d, the more opportunity for data-level parallelism in the top-down reconstruction process. Similarly, the value of d is limited by the implementation of the *ad*-heap structure. The most effective implementation of the *ad*-heap features both the lower data-level parallel opportunity implied by the value of d as well as the task-level parallel opportunity of multiple heaps being processed, concurrently. In this case, the value of d is limited by the maximum work-group size, given that each work-group processes a separate heap. The work-group size is also dependent on the size of the implicit bridge where each work-item in the work-group must offload the implicit bridge to global memory in parallel. The implicit bridge size is fairly dependent on the height of heap data structure which is dependent on both the total number of nodes within the heap and the value of d.

We begin to see the multitude of factors that must be considered in understanding the *ad*-heap data structure. In the results presented in Chapter 5, we see that given a single *Update-Key* operation, the implicit d-heap implementation on the central processing unit is extremely fast in top-down reconstruction computation for all values of d when measured in milliseconds. The true performance difference isn't apparent until thousands of *Update-Key* operations are effectively issued. The *ad*-heap as shown in the previous experiments will not be able to outperform the CPU implementation for reasonable values of d(generally 8,16,32,64) as a single heap-based implementation. It relies on a combination of both the task-level parallel opportunity of multiple heaps being processed concurrently and the lower level data-parallel opportunity of a large value of d.

This requirement is reflected in the multi-threaded POSIX API-based CPU implementation of the implicit d-heap. This implementation utilizes the multi-core central processing unit to concurrently process a group of implicit d-heaps. The performance benefit of this approach is easily observed over the sequential CPU implementation, this showcases the desire to follow a task-level parallel approach when the opportunity is available. The graphic processing unit further expands upon this with its greater number of compute units. Whereas a modern central processing unit typically has between 4-8 physical cores(as shown in our experimental platforms), the graphic processing unit has between 8-32 compute units(also as shown in our experimental platforms). In the multi-threaded CPU implementation, each processing core essentially handled a group of implicit d-heaps, the group of d-heaps are processed serially. Given c processors and a total group of h heaps, you would have each processor serially processing h/c heaps. This is also true for the graphic processing unit but with a much larger value of c and thus a smaller resulting group size.

Therefore, the graphic processing unit is theoretically better suited for this opportunity of parallel optimization. The *ad*-heap seeks to maximize this opportunity while still incorporating the multi-threaded low-latency abilities of the central processing unit to perform the serially executed assignments which the central processing unit is able to do so very

quickly. Also, as stated above, the central processing unit is able to perform the top-down reconstruction when the *Update-Key* operation is performed only once. So in addition to the task-level parallel benefits of the graphic processing unit; the graphic processing must be able perform the top-down reconstruction operation much quicker than the central processing unit. The *ad*-heap seeks to achieve this by minimizing any expensive or irregular memory accesses such as possible thread divergent accesses to the global memory space for assignment and limiting the reduction computation to be performed in the much closer local memory area; allowing the central processing unit to perform the necessary assignments off of the implicit bridge.

6.1 Practicality of the *ad*-heap data structure

As research and development continues to progress to seek heterogeneous solutions to effectively utilize all hardware components of the computing system in an efficient manner; interest will continue in exploring existing data structures and algorithms which normally would not benefit from the current loosely coupled GPGPU computing platforms. By exploring the *ad*-heap, we also explore the process of thought that goes into designing and adapting for this modern trend in computing technology. We identify the dependencies inherent to approaching such a solution and the different aspects of the structure and the mechanics of the associated operations which should be considered. By effectively simulating the practicality of the structure on two physical platforms, we are able to observe the effects of existing hardware on the effectiveness of the algorithm and the shortcomings which must be resolved to achieve the performance benefits described by the theoretical design of the data structure. By exploring each possible test case presented in Chapter 5, we are able to see the general effects of each data set size on the behavior of the *ad*-heap data structure.

In comparison to the CPU-based implicit d-heap implementation of the k-selection algorithm, the performance of the simulated *ad*-heap structure on the loosely coupled graphic

processing unit suffered from several obvious issues inherent to the experimental platform it was executed upon. In the case of Machine 1, given the reliance on the PCI-E bus, the majority of the *Update-Key* operation was generally spent in handling the memory transaction between the Host central processing unit and Device graphic processing unit. This performance bottleneck essentially provided the performance edge for the APU-based experimental platform in the test cases where the number of heaps being concurrently executed became increasingly smaller. As discussed in Chapter 4, in HSA Foundation's theoretical truly heterogeneous computing system, the typical problematic scenario presented by the extensive memory handling penalty essentially becomes a non-factor. From a hardware perspective, both devices would share the same physical memory in the last-level cache of the memory hierarchy, similar to the APU-based experimental platform where the effect of the memory handling was reduced significantly. From a software perspective, the unified virtual memory space would allow for a more natural interface of control over the memory management aspect of the heterogeneous solution. The current programming platform provided by OpenCL does not offer this unified virtual memory-based interaction.

Another observed shortcoming of the *ad*-heap in the previous experiments is the nature of the structure in the batch k-selection algorithm. The k-selection algorithm requires thousands of *Update-Key* operations to be performed for the algorithm to effectively produce the kth smallest element in each sub-list. Each *Update-Key* operation results in a associated launched kernel for the *ad*-heap structure. This is intentionally avoided in most current GPGPU computing solutions since the time to prepare the device for the kernel workload becomes incredibly penalizing if thousands of kernel launches are needed. As described in Chapter 5, the development of the tightly coupled truly heterogeneous HSA solution seeks to minimize the associated penalties of this interaction. By allowing the two hardware components to work more closely together, you present more opportunity to effectively distribute the workload of your application to whichever device can handle the workload more effectively.

The *ad*-heap data structure presents a uniquely interesting case study of the inspirations behind HSA Truly Heterogeneous Computing System. Much of the *ad*-heap design features are extremely dependent on the theoretical features of the HSA-based architecture. The *ad*-heap data structure, requires a asymmetric multi-core platform which allows each platform of processing cores to be able to share the workload in the more efficient manner possible. By eliminating the associated penalties suffered by this data structure on both the loosely-coupled discrete GPU-based implementation and the APU-based platform, the focal point of the design is drawn to the workload distribution. Each aspect of the operations of the structure must be distributed to whichever hardware component is best utilized for that workload. In this case, the top-down propagation of the Update-Key operation is offloaded to the throughput-oriented device since the parallel reduction scheme can find the maximum child theoretically faster than the sequential execution of the CPU while the assignment workload is offloaded to the central processing unit.

Theoretically, this is a reasonable approach, especially in the case of batch k-selection algorithm where task-level parallel opportunity also presents an interesting opportunity to fully saturate the compute units of the device. But as shown in experiment results, both the combination of the task-level parallelism and data-level parallelism does not showcase the performance benefits that you would expect from either experimental platform's throughput-oriented device. As observed, once the number of heaps becomes fairly large as in the first test cases in the previous experiments, the compute unit's serial execution queue of work-groups becomes increasingly large. Similarly, the multi-threaded CPU implementation, the serial execution queue of heaps becomes even-more so large. Whereas the multi-threaded CPU can only process 4-8 heaps concurrently, the GPU can process from 8(the APU-based device) to 32(the discrete GPU) heaps in parallel. Therefore, the performance benefit becomes dependent on the graphic processing unit's ability to quickly process each heap. This measurement becomes heavily dependent on the data-level parallelism available in the heap's structure.

We explored the effect of this data-level parallelism in our first experiments of the *ad*-heap data structure which essentially measured the performance metrics of the *Update-Key* operation on only one heap. By observing strictly kernel execution time, we can see that despite the parallel reduction scheme to find the maximum child, for values of d(8,16,32,64) as used in the original *ad*-heap, the data-level parallel opportunity is not necessarily enough to utilize the graphic processing unit's hardware to exhibit performance benefit over the recursive CPU implementation. Though, the graphic processing unit can issue enough work-items to find the maximum child in parallel, the CPU can sequentially perform the computation quick enough that the performance difference between each value of d can only be observed over thousands of instances of the *Update-Key* operation. While this doesn't necessarily discount the capabilities of the *ad*-heap data structure, it does place an emphasis on requiring a much larger value of d to possibly see a fundamental improvement.

6.2 The *ad*-heap as a design initiative.

In conclusion, even though the *ad*-heap may have been an overly optimistic design in its general premise, it does present an interesting exploration into the possible opportunities for optimizing existing algorithms and data structures based around the truly heterogeneous computing systems as proposed by the HSA Foundation. It is fairly difficult to gauge the actual abilities of the theoretical *ad*-heap data structure when the theoretical hardware/software platform which it is based around is currently unavailable; it is however easy to gauge the current technology limitations in relation to this design. By exploring an implementation that more closely simulates and resembled the original *ad*-heap design on current experimental platform which differ in hardware characteristics, we can observe many of the hardware and software obstacles to overcome to ensure that this design premise is translatable to other existing data structures and algorithms. The *ad*-heap as a design initiative presents a interesting perspective.

The initiative of heterogeneous computing revolves around the ability to fully utilize

all hardware components in the computing system to produce a synergistic solution. The initiative of the HSA Foundation is to minimize all possible hardware and software bottlenecks and obstacles that may hinder this relationship between all hardware components from both the programmer's perspective and also the hardware manufacturer's perspective. The *ad*-heap design presents a design initiative to seek out opportunities in every new or old existing algorithm or data structure to utilize this computing paradigm in its truest sense possible; a truly heterogeneous computing system. Because of these relatable concepts, the *ad*-heap presents a very interesting case to explore and understand.

BIBLIOGRAPHY

BIBLIOGRAPHY

(AMD APU Fusion) (2010).

(AMD Developer Central) (2013).

Bradski, G. (), *Dr. Dobb's Journal of Software Tools*.

Branover, A., D. Foley, and M. Steinman (2012), Amd fusion apu: Llano, *Micro, IEEE*, 32(2), 28–37, doi:10.1109/MM.2012.2.

Butenhof, D. R. (1997), *Programming with POSIX Threads*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Daga, M., A. Aji, and W. Feng (2011), On the efficacy of a fused cpu+ gpu processor (or apu) for parallel computing, in *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, p. 141149, IEEE.

D'Alberto, P. (2012), A heterogeneous accelerated matrix multiplication: Opencl + apu + gpu+ fast matrix multiply, *CoRR*, abs/1205.2927.

Damaraju, S., et al. (2012), A 22nm ia multi-cpu and gpu system-on-chip, in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pp. 56–57, doi:10.1109/ISSCC.2012.6176876.

Hennessy, J. L., and D. A. Patterson (2003), *Computer Architecture: A Quantitative Approach*, 3 ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Hong, S., T. Oguntebi, and K. Olukotun (2011), Efficient parallel graph exploration on multi-core cpu and gpu, in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 78–88, doi:10.1109/PACT.2011.14.

(HSA Foundation) (2013).

Klein, G., and D. Murray (2007), Parallel tracking and mapping for small AR workspaces, in *Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07)*, Nara, Japan.

LaMarca, A., and R. Ladner (1996), The influence of caches on the performance of heaps, *J. Exp. Algorithmics*, 1, doi:10.1145/235141.235145.

Levitin, A. V. (2002), *Introduction to the Design and Analysis of Algorithms*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Liu, W., and B. Vinter (2014), Ad-heap: An efficient heap data structure for asymmetric multicore processors, in *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pp. 54:54–54:63, ACM, New York, NY, USA, doi:10.1145/2576779.2576786.

Matsumoto, K., N. Nakasato, and S. G. Sedukhin (2012), Performance tuning of matrix multiplication in opencl on different gpus and cpus, in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pp. 396–405, IEEE Computer Society, Washington, DC, USA, doi:10.1109/SC.Companion.2012.59.

Merriam-Webster Online (2009), Merriam-Webster Online Dictionary.

Munshi, A., B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg (2011), *OpenCL Programming Guide*, 1 ed., Addison-Wesley Professional.

NVIDIA Corporation (2011), *NVIDIA CUDA C Programming Guide*.

Owens, J. D., S. Sengupta, and D. Horn (2005), Assessment of graphic processing units (gpus) for department of defense (dod) digital signal processing (dsp) applications, *Tech. rep.*

Pulli, K., A. Baksheev, K. Kornyakov, and V. Eruhimov (2012), Realtime computer vision with opencv, *Queue*, 10(4), 40:40–40:56, doi:10.1145/2181796.2206309.

Vuduc, R., A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure (2010), On the limits of gpu acceleration, in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pp. 13–13, USENIX Association, Berkeley, CA, USA.

VITA

STEPHEN BLAKE ADAMS

647 CR 2788 • Baldwyn, MS 38824 • (662)213-2571 • sbadams662@gmail.com

EDUCATION

M.S., Engineering Science, University of Mississippi, May 2014

Thesis: Exploration into the Performance of Asymmetric D-Ary Heap-Based Algorithms for the HSA Architecture

B.S., Computer and Information Science, University of Mississippi, May 2012

A.A., Northeast Mississippi Community College, May 2012

HONORS and FELLOWSHIPS

Upsilon Pi Epsilon Computer Science Honor Society, 2012
Department of Computer and Information Science, University of Mississippi
Gamma Beta Phi Honor Society, 2010
University Of Mississppi
Pi Mu Epsilon Mathematics Honor Society, 2012
University Of Mississppi