Analyzing the Benefits of Graphics Processing Units for Computation-Intensive Applications on Hadoop

by

Kevin Vasko

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
August 1, 2015

Keywords: CUDA, JCuda, GPGPU, Nvidia, Hadoop, YARN

Approved by

Weikuan Yu, Associate Professor of Computer Science and Software Engineering
David Umphress, Professor of Computer Science and Software Engineering
Dean Hendrix, Associate Professor of Computer Science and Software Engineering

Abstract

Due to the ever expanding amount of data that is being generated in the "Big Data era" there is an ever increasing challenge of processing this data. This work aims to tackle the challenge of improving the performance of processing unstructured data being generated by combining two different technologies of General Purpose Graphics Processor Units (GPGPUs) and Apache Hadoop. Many researchers have focused on improving either the GPGPU or Apache Hadoop; very little amount of evaluation data is available in combination of the two technologies, which we aim to study in our work. JCuda, JCublas and JCuFFT were used in conjunction with the CUDA library to incorporate GPGPU computation within the Apache Hadoop framework. We utilized the Nvidia Tesla M2050 and up to 16 compute nodes to evaluate the integration of the GPGPU and Apache Hadoop framework with three different use cases. Two synthetic benchmarks, matrix multiplication, fast Fourier transformations, and a real world application, image processing using the Gaussian blur filter. We were able to achieve up to, 6.91×, 2.48× and 1.49× of overall performance improvements for matrix multiplication, FFT and Gaussian blur respectively. We expect that our work will be useful to the reader to gauge the amount of performance that they would be able to achieve under certain workloads for their own work with the combination of GPGPU and Apache Hadoop.

Acknowledgments

First and foremost, I would like to thank my committee chair, Dr. Weikuan Yu for his willingness to take me on as a student. His support, leadership, helpfulness, thought provoking learning environment, generosity and constant encouragement was invaluable to my success as a graduate student here at Auburn University. Through his teaching and research methods he was able to provide an incalculable amount of theoretical and real world knowledge that will follow me through the rest of my career. I could not have asked for a better academic advisor and am honored to have studied under the leadership of Dr. Yu.

I would also like to express my gratitude to my other committee members, Dr. David Umphress and Dr. Dean Hendrix. Their helpfulness, advisement, and stimulating discussions that were provided in my tenure as a gradate student is greatly appreciated.

Also, a great big thank you goes to all members of the Parallel Architecture and System Laboratory (PASL). The members of the PASL proved to be an invaluable resource, that helped me immensely through my graduate studies. I am glad to have met and had the pleasure of working with all of the PASL members and am honored to call each and everyone of them a friend. In no particular order: Fang Zhou, Bin Wang, Michael Pritchard, Hai Pham, Zhuo Liu, Yandong Wang, Cong Xu, Teng Wang, Huansong Fu, Xinning Wang, Yue Zhu, Hao Zou, Dr. Hui Chen and Dr. Jianhui Yue. In addition, while not part of my committee or the PASL, I would like to thank Dr. Jeff Overbey for his time, help, and the discussions we had over my work.

In addition, I would like to thank my family members, Angela Vasko (mother), James Vasko (father), Paula Elgie (sister), Mark Elgie (brother-in-law), Millie Faircloth (grandmother), Grover Faircloth (grandfather) and Dr. Leigh Anne Conner (girlfriend) for their constant love, support and encouragement through my graduate study.

Last but not least, I would like to acknowledge the sponsors of this research and my graduate study at Auburn University. Particularly, I would like to acknowledge the Alabama Innovation Award and the National Science Foundation awards 1059376, 1320016, 1340947 and 1432892.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

CMP Chip Multi-Processing

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

GPGPU General Purpose Graphics Processing Unit

HDFS Hadoop Distributed File System

IaaS Infrastructure As A Service

NIST National Institute of Standards and Technology

PaaS Platform As A Service

SaaS Software As A Service

SIMD Same Instruction Multiple Data

SMT Simultaneous multithreading

Chapter 1

Introduction

We have now entered into an era that technologists have coined as, the "Big Data era[20]." Currently the amount of data that is being generated make it difficult or almost impossible to process the data reliably with traditional processing methods. To put the amount of data being generated into perspective, YouTube users upload 72 hours of new video, people send 204,000,000 emails, Google receives over 4,000,000 search queries, twitter gets 277,000 tweets, and Instagram users post 216,000 new photos every minute[27]. Being able to process this data in a timely manner would allow the owners of the data to make better business decisions, which in turn can provide various benefits, such as, greater profits, additional safety measures, etc. Unfortunately, there are massive challenges in processing this data. Due to the sheer volume of the data being generated, conventional systems are not capable of processing such colossal amounts of data in a timely manner. This leaves a void that needs to be filled with the latest technologies that are capable of handling these needs.

Over the past few decades we have seen a transition between the methods of how clients process data on computer systems. These transitions were intended to improve efficiency, speed, and reliability to the client, so their computational needs could be met. One of the classic computing techniques is called batch processing. With this method, clients were able to queue up many different programs to run in batches. This avoided the issue of the client having to manually enter each program after the previous job was completed. This allowed the computing resources to be used more effectively because the computing platforms were not left idle while the next job was input. Eventually, computing systems transitions into utilizing a method of processing known as time-sharing. This allowed many clients to utilize

the expensive computer systems more efficiently. More clients could be serviced such that if the system was waiting on service requests such as I/O, other jobs could be executing. As additional data was being generated, compute systems could adapt the configuration accordingly to manage and process the data.

One of the technologies that has emerged recently to cope with the "Big Data era" is cloud computing. The definition of cloud computing by the National Institute of Standards and Technology (NIST) is "a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort[30]." The cloud model has several essential characteristics such as, "broad network access, resource pooling and rapid elasticity." With these characteristics, the users of clouds can exploit varying amounts of computational power that would otherwise not be available due to cost constraints. A significant amount of capital for initial setup and continued expenses for maintenance would be required. With cloud computing it allows for no upfront investment for the hardware to process the immense amounts of data that would otherwise be out of reach because of cost. In addition, the cloud environments provided by large cloud vendors such as Amazon, Microsoft, IBM or Google can provide an extremely powerful computing facilities for clients[19].

In addition to cloud computing GPGPUs in recent years have exploded into the scene as highly capable programmable computational devices. In the early 1990's, GPGPUs were not designed with the intention of computation in mind, but to assist the CPU in accelerating 3D games. These early units were only capable of a fixed set of functionality. The next generation GPGPU devices were pixel and vertex shader units. These units enabled small amounts of programmability and by comparison were immensely more capable than their earlier counterparts. In 2006 NVidia introduced streaming processor units, which are generalized computing devices. Nvidia created the Compute Unified Device Architecture that provides a programming model for interacting with these GPGPU devices. With the introduction of the CUDA programming model, developers are able to easily utilize the processing power of GPGPUs effectively to solve a wide variety of computationally expensive problems. The work

in this area has resulted in an exceptionally parallelizable device that provides extraordinary throughput while providing an enormous amount of computational power.

The technologies mentioned above are individually capable of providing an immense amount of computational power on their own for engineers and scientists to exploit. However, as time goes on, the trends we discussed earlier with the "Big Data era" will become more demanding. Because of this impending challenge, combining these two technologies will allow us to leverage both for effective data processing. Utilizing both technologies would allow us to leverage advancements in each of the respective fields.

This work will focus on combining GPGPU with Apache Hadoop in a cloud computing environment to allow users to utilize the vast amounts of computation power to process unstructured data. Note that unstructured data can come in many forms, but is not limited to binary files, PDFs, Word documents, music, image data, and videos. For this work we will focus on improving the speed and capability of processing matrices, Fourier transforms and image data. Fast image processing frameworks in the cloud will be very useful for us to keep up with the growing demand and this work aims to provide those improvements in this area of research.

The rest of the paper will be organized as follows, Chapter 2 will provide background of the work, Chapter 3 will provide related works in this particular area of research, Chapter 4 will include the disgn and implementation and the issues that were solved, Chapter 5 will contain our evaluation, and in Chapter 6 we will provide our future work and conclusion.

Chapter 2

Background and Motivation

This chapter will be presenting more detailed information of the material that was introduced in chapter 1. Additional context will be discussed to inform the reader of information that might be ambiguous at this point. Along with a more detailed look of this work, the goals and motivations will be presented within this chapter. This chapter provides a conceptual overview for readers to understand Hadoop, cloud computing, its advantages in the respect to data processing, a general understanding of an approach to utilizing GPGPUs within a cloud environment and the way these two technologies can work together to achieve greater performance than each in their own respect. If you require additional resources on the topics that were discussed, please see the references at the end of this work.

In section 2.1 an introduction to the MapReduce programming model will be presented, along with a discussion on the MapReduce programming model and its usage for cloud computing. In section 2.2 the Apache Hadoop framework will be discussed in further detail. The information that is presented will include the frameworks overall design, how it is applied in the context of cloud computing, the typical use cases of the framework, and how the MapReduce framework will be utilized for this works purpose of improving matrix multiplication, fast Fourier transformations, and image processing within Hadoop. Section 2.3 will provide further detail of GPGPUs. A discussion of the applications of this technology and where it is being applied in today's society. This section will also include a brief architecture overview, an introduction to Nvidia's CUDA programming platform, a brief discussion of the impacts of tuning, the platforms parallelism capability, and the general strengths and weaknesses of the platform. In the conclusion of this chapter we will present goals and motivations of this work, which is to apply GPGPU computation within Hadoop that can be used within a

cloud computing setting to enhance the performance of processing unstructured information datasets.

2.1 MapReduce

MapReduce is a parallel distributed programming model that allows its users to utilize large amounts of data to solve highly complex computational tasks. It was first introduced in [22] and has become extremely popular since its introduction due to its highly capable parallel model. Its usefulness stems from the ability to allow the user to take large data sets that are not feasible to compute on standalone systems and process the given data efficiently on large clusters.

The MapReduce framework consists of four main sections, an input reader, a map function, a reduce function and an output writer. As shown in Algorithm 1, we utilize the word count example for illustration purposes. The input reader takes input and splits the provided information into usable segments. A book document passed to wordcount could be split by chapter (or any other meaningful way to the user). Each chapter is assigned a key and the values would be the words in that chapter. Each one of the splits that is created, is assigned to a Map function. The Map function takes in the Key and Value pairs and will perform some form of operation (sort, filtering, etc.) on the group of parameters. With the word count example the Map outputs the word with a trivial count of one. Once the Map function has completed, it will output its results as a Key and a list of Values *(K2,list(V2))* to the Reduce function. The Reduce function will then perform a summary operation and will output a list of values *list(V3)* with the result of the operation. The word count example would take the input values of words as the key, sum the occurrences and output the final sum. The output writer will take the information exported by the Reduce functions and write the information to storage.

With image datasets growing at a rapid pace in the Big Data era, the MapReduce programming model will greatly improve the processing capabilities due to its parallelizable

Algorithm 1 Word Count MapReduce Example[12]

```
1: function MAP(String chapter, List words)
2:     while words.hasNext() do
3:         word.set ← words.getNextElement()
4:     end while
5:     output(word, 1)
6: end function
7: function REDUCE(String key, Iterator values)
8:     sum ← 0
9:     while values.hasNext() do
10:        sum ← sum + values.getNext()
11:    end while
12:    output(key, sum)
13: end function
```

model. For this work, the datasets of images will be fed to an image processing framework that utilizes MapReduce as its underlying model. These image datasets are split into smaller segments and will be passed to the map function where any type of image algorithms can be performed on the image. The output of the algorithm upon the image will then be written out to storage for consumption. Ultimately, being able to perform work in parallel while taking advantage of larger pools of resources will produce the desired results in a shorter amount of time. Utilizing the MapReduce programming model will help in achieving this goal.

2.2 Apache Hadoop

Since MapReduce is simply a programming model, there needs to be a way to utilize this power in a more organized manner so we can take full advantage of MapReduce's performance capabilities. In this work, the Apache Hadoop[18] framework is used as it provides the capability to scale outwards while utilizing cheap commodity servers to provide massive compute clusters. These systems have successfully grown to hundreds or even thousands of nodes with petabyte scale clusters[21].

The Apache Hadoop framework is an open-source implementation in Java of two components, MapReduce and HDFS. MapReduce provides the distributed data processing engine and HDFS supports a data store that is similar to Google Big Table[22]. The HDFS provides a persistent store that is capable of dealing with massive datasets, that is typically broken up into blocks, usually 128MB in size. This data is then distributed across the cluster via socket layer communication. The information within HDFS can also be replicated as needed to provide additional redundancy, while also taking advantage of data locality during processing. Since the data is distributed across the cluster, a MapReduce job and its tasks can benefit from the distributed nature during processing.

In an Apache Hadoop deployment, there are several services in a multi-node cluster. On the MapReduce layer there is the TaskTracker and JobTracker. The two that are required for the HDFS, are the NameNode and DataNode services. For simple implementations there is typically one master node hosting the NameNode and JobTracker services. The other services (DataNode and TaskTracker) are ran on the slave nodes. They can be run on the master node, but is not recommended. More complex installations would have additional services for fail-over or high availability.

The NameNode service provides centralized meta-data information of the data stored within the HDFS. It maintains a list of the files that are stored within HDFS and DataNode holds the data. In addition, it also keeps track of files that are replicated within the DataNodes. The NameNode does not hold the data itself, only the meta-data to tell the client where to retrieve the required data. The DataNode services in HDFS perform the functionality of hosting the blocks of data while serving the clients requests for data as needed. When a client asks for information from the HDFS, the NameNode returns the location of the DataNode that holds the information. The client can then request the information from the DataNode directly to perform the operation.

The processing layer consists of the JobTracker and the TaskTracker. The JobTracker provides the functionality of coordinating with the NameNode service to determine which

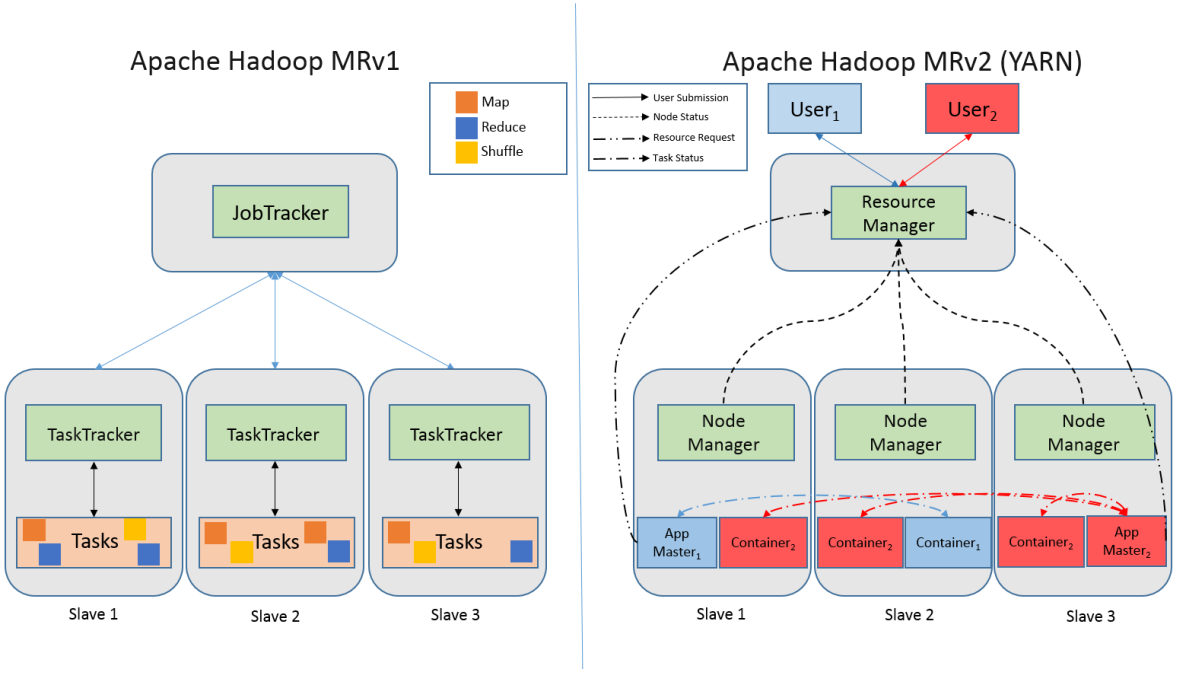

Figure 2.1: Apache Hadoop YARN[9][7]

nodes have the data, ascertains which TaskTracker nodes have available resources, and then designates the MapReduce job to the chosen TaskTracker node[24]. In typical installations of Apache Hadoop there is a single JobTracker service on a master node for the entire cluster which handles the scheduling of jobs from client requests. After the JobTracker assigns the jobs to the TaskTracker nodes (slaves), they are monitored for their progress. Any job failures will be relaunched by the JobTracker. The TaskTracker service has a certain number of slots (ideally based on resources available on the node) which can be occupied by a Map, Reduce or Shuffle job. The JobTracker makes a best effort to take advantage of data locality by scheduling the Map, Reduce, or Shuffle jobs on the slave nodes that hold the data needing to be processed.

Apache Hadoop YARN[17] (Yet Another Resource Negotiator) is the second generation of the Apache Hadoop project, sometimes referred to as Apache Hadoop MapReduce Version 2. YARN shifts the way the system provisions resources for the jobs executing on the cluster. In Figure 2.1 it can be seen that the JobTracker and the TaskManager's functionality were taken over by the ResourceManager and NodeManager in YARN, however, they differ

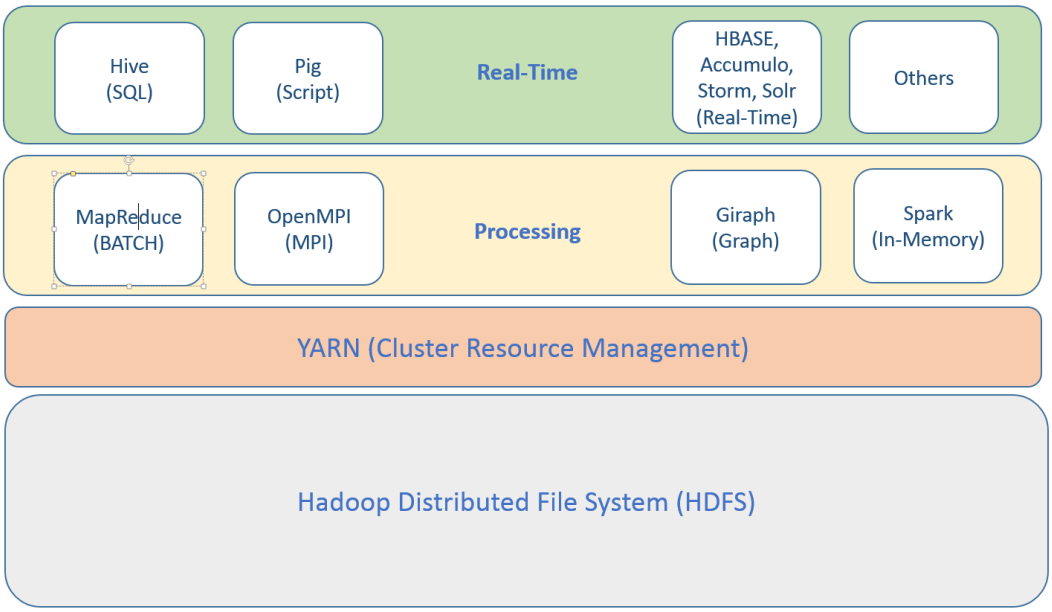

Figure 2.2: Apache Hadoop YARN Ecosystem[8]

slightly. YARN's ResourceManager functions solely for allocating resources to the applications requesting resources on the cluster. It does not manage any of the tasks on the cluster and does perform any actions on the tasks running on the cluster, which is the responsibility of the NodeManager. The NodeManager is per-node and manages the nodes resources (CPU, memory, disk) accordingly and reports back to the ResourceManager of this utilization. The ApplicationMaster is accountable for communicating with the ResourceManager to request appropriate resources for the containers and to manage the state of the containers, such as their progress.

The shift to this new platform allows for greater flexibility, scalability, cluster utilization and availability for users. In addition, since more companies are storing their data in HDFS, YARN provides the functionality to use different programming models other than the MapReduce programming model to utilize the data stored within HDFS as we can see in 2.2. Instead of just having a single programming model (MapReduce) to take advantage of the data within HDFS, other methods such as the real-time processing engines can allow the users to utilize the data in a more effective manner. Even though this work only focuses

on utilizing MapReduce, you can see that there are many other areas within this ecosystem that could benefit from the utilization of GPGPUs. Also, with Hadoop being distributed in nature it is ideal to be used within a cloud environment to process the vast amount of data being generated in this "Big Data era."

2.3 General Purpose Graphic Processor Units

Even though the frequency of CPUs have not increased in recent years, the performance trend of processors has continued [36]. Because of this, it has required designers to seek other methods to improve the performance of computer systems. Much of the improvement is achieved through advancements in caching systems, instruction pipelining optimizations, instruction sets, out-of-order execution, register renaming, and multiprocessing. SMT and CMP are two technique types that have been employed to enable multi-processing capabilities of today's processors. SMT enables this capability by allowing CPUs to execute instructions from different processors in the same computational cycle. CMP or more commonly referred to as a "Multi-core processor" [39] is a single die with multiple processing units on the chip. The processing units can be coupled as tightly or loosely as the designer chooses, an example of this is the sharing of cache.

With more processing cores available on a system, additional processes can be executed, allowing the system to handle more tasks, which result in substantial improvements for certain workloads. While CPU designers are slowly adding additional cores as manufacturing processes and architectural advancements improve, more resources have to be focused on handling other computational needs besides processing (e.g. branching logic). On the other hand, GPGPU designers do not have to worry about complex branching logic or difficult instruction sets, and can focus on computational logic. This has resulted in GPGPUs having hundreds or even thousands of compute cores to perform computation.

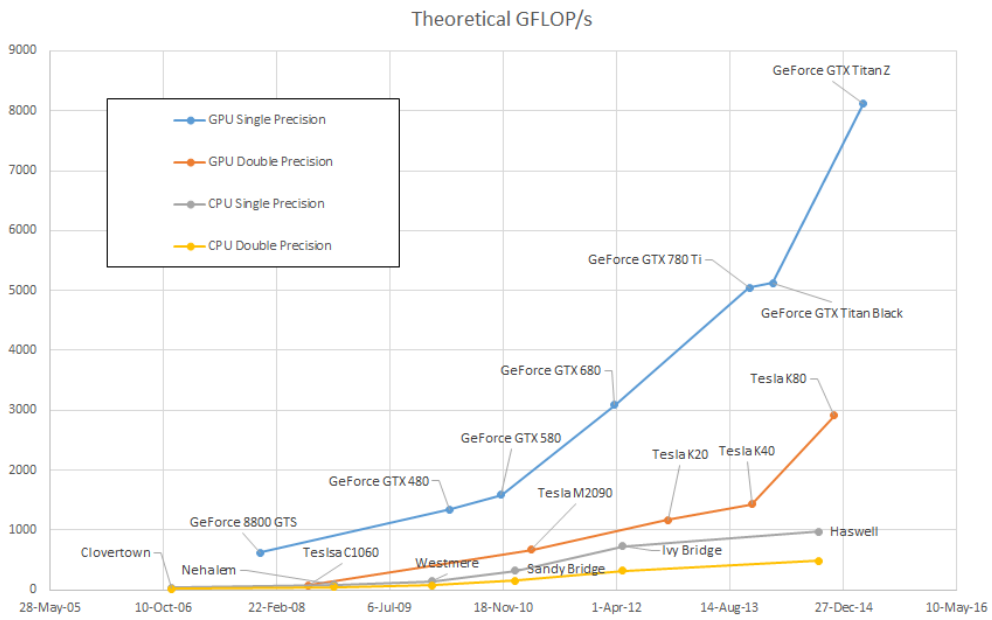

Figure 2.3: GPU Theoretical GFLOP/s [4]

2.3.1 Motivation of utilizing the performance of GPGPUs compared to CPUs

GPGPUs are highly efficient, in that they enable a high amount of parallel processing capabilities and throughput, typically several orders of magnitude of floating point computational power and far greater memory bandwidth over their CPU counterparts[32], even though actual frequency of GPGPUs are typically slower than traditional CPUs. This is a result of designers being able to prioritize the available die space to data processing logic, instead of flow control and caching as we discussed earlier. Ideally, a GPGPU will act as a coprocessor to aid the CPU in performing the complex computational work of parallelizable and computationally expensive code. In Figures 2.3 and 2.4 we can easily see a strong motivation to take advantage of the tremendous computational power and memory bandwidth capacity the GPGPU has over traditional CPUs.

2.3.2 CUDA Architecture

As presented in the introduction, CUDA is a parallel computing platform and programming model that provided a way for developers to programmatically access the computational

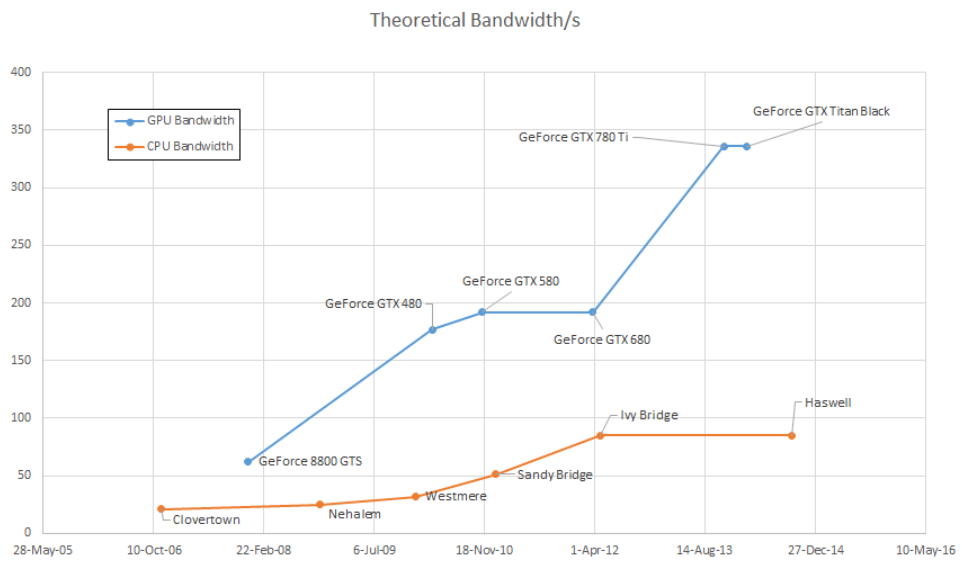

Figure 2.4: GPU Theoretical Bandwidth [4]

power of Nvidia GPGPUs. The first iteration of the CUDA architecture was released[33] by Nvidia in 2006. It consists of the general principles of the device architectures, CUDA C/C++, along with several libraries. As we have discussed above the GPU architecture accentuates the idea of processing several items in parallel at the cost of a slower rate instead of processing a single item at a faster rate. With CUDA, programmers can utilize high level programming languages such as C, C++ and Fortan instead of having to resort to assembly language development to write their applications. The CUDA applications written in CUDA C/C++ and are compiled with the nvcc compiler and while hold close to the C/C++ standards CUDA C/C++ includes extensions to those standards.

GPGPUs are capable of producing several orders of magnitude[37][28] of processing performance improvements over the same CPU implementation due to their highly parallel nature. Their parallel nature stems from the ability to launch hundreds or even thousands of threads simultaneously to have work computed in parallel. At the lowest level in the CUDA architecture there exists a Streaming Processors (SP) also known as a CUDA core, these devices are capable of executing a single thread of instructions. The multiprocessor, also known as Streaming Multiprocessors (SMs) contains an array of SPs. The instructions

processed by the SM of each SP is executed in lock-step, meaning all SPs will execute the same instructions. If a SP thread is required to execute a different set of instructions from the other SPs, the other SPs will go into a NOOP state waiting until all SPs are executing the same instruction again. This causes a reduction in parallelism and can greatly impact performance. Each SM has its own set of registers and shared memory available to the SPs. Each one of the SMs can communicate with the global memory on the GPGPU device. The host can only interact with the global memory, constant and texture memory devices as you can see in Figure 2.5. When a programmer wants to execute code upon the GPGPU they are required to copy the data from the host memory to global memory, constant memory, or texture memory, call the appropriate function for the kernel to get executed upon the CUDA device and then copy the information back to the host.

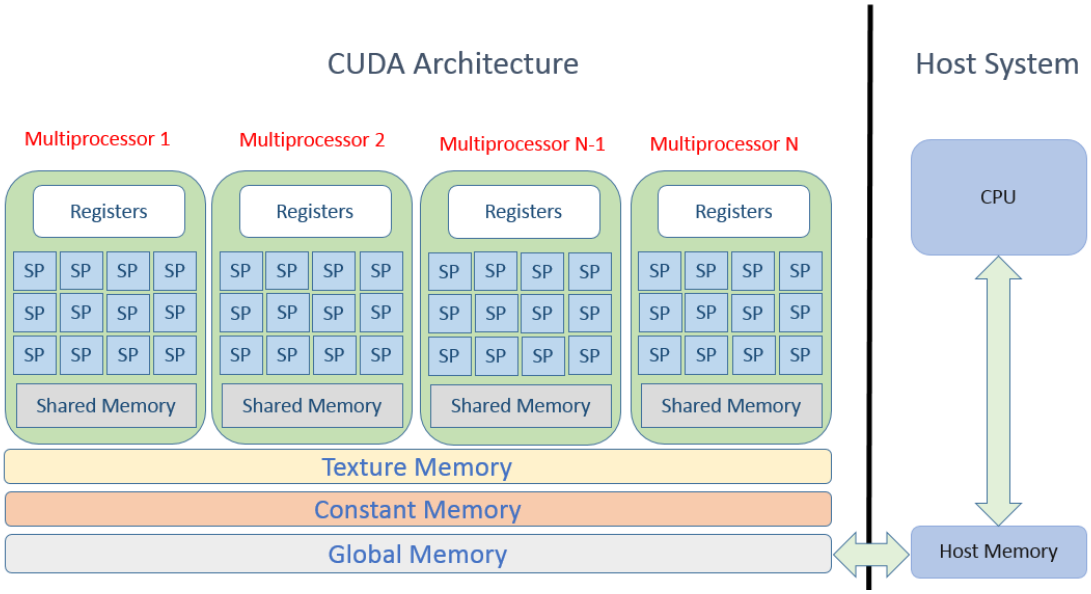

Figure 2.5: CUDA Architecture [2][4]

GPGPUs are excellent in data parallelism where large amounts of data is treated the same way. For example in this work with matrices and image processing, the GPGPU is ideal as each piece of data is treated in similar ways. However, two things one should be aware of when utilizing the GPGPU, which can significantly impact the usefulness of the GPGPU: 1) Since the GPGPU architecture focuses on parallelism and the SMs execute the same

instructions on all SPs in lock-step fashion if there is any form of control divergence (e.g. computation is done based on a certain core) the cores that are executing the branch will execute while the other cores that did not meet the branch condition will suspend execution until all cores are executing the instruction again; 2) There exists a large bottleneck between the host memory and the global memory of the GPGPU device, the PCIExpress bus. The developer needs to be aware of this particular issue as this bottleneck could render any performance improvement gained by the parallel capability of the GPGPU moot.

For our work, we plan to investigate the advantage of the highly parallel nature of the GPGPU and MapReduce and leverage them to tackle the growing problem of processing matrix and image data at a pace that the industry demands. In combination with cloud computing, we aim to provide significant performance improvements with this work, which will enable the capability of processing larger data sets that will help engineers and scientists tackle larger and more difficult issues.

Chapter 3

Related Works

In this chapter we will be discussing the various works that are related to what we are presenting in this paper. These research topics include utilization of GPGPUs for image processing, utilizing MapReduce, utilization optimizations of GPGPUs with MapReduce, and the topics of combining Hadoop and GPGPU computation together. This section provides a review of past contributions in this field and show related methods being utilized to tackle these challenges today.

Currently, processors have not seen as much improvement in performance from the result of clock speed increases as they have in recent history. This is due to physical limitations such as heat and power constraints. However, performance of processors continue to increase due to other optimizations, such as pipelining, architectural, parallel processing and manufacturing processes. With parallel processing techniques, the amount of performance certain applications over non-paralleled computation can be substantial for certain workloads. Recently the computing industry started to utilize GPGPUs as they provide extremely high parallelism and are ideal to provide additional performance[34].

Matrix multiplication, FFT computation and image processing are types of workloads that are highly suited for GPGPUs. As discussed earlier in this work, GPGPUs sacrifice raw single threaded performance in place of multithreaded throughput. When dealing with pixel information of an image there is a great amount of parallelism that can be exploited to obtain a high amount of performance from a GPGPU than a CPU. This is because most image processing algorithms are performing the same computation work on different pixels to obtain the result, which is exactly the type of workload that GPGPUs excel at. To provide some context, Nvidia has successfully shown a significant improvement on the performance of

certain image processing algorithms of several orders of magnitude. The results of NVidia's latest evaluation of NPP 6.5 on the NVidia K40m compared against the Intel Ivy Bridge single socket 12-core E5-2697 v2 @ 2.70GHz shows an average speed of up to 4.1× on filtering and up to 109× on color processing functions. Other algorithms such as, color conversion, alpha composition and geometry transforms showed a speed up of up to 11×, 15× and 19× respectively.

The OpenCV project has recently started to support the CUDA architecture within the library[14]. With their recent work they have seen up to 30× performance gains with select primitive image processing algorithms and between 7× and 12× performance gains in certain Stereo Vision and SURF keypoints algorithms respectively. With performance seen in improvement by Harvey et al. in [25] with computer vision algorithms, it gives good reason for us to investigate into other avenues to attempt to improve the performance of this type of work in a larger scale.

Many researchers have started to integrate GPGPU computation into the MapReduce[22] programming paradigm to further improve parallel data processing. One work describes their framework they called "MARS" [26], which intends to provide two goals, 1) ease of programming and 2) performance. This work is one of the first approaches to integrate both ideas of GPGPU computation and the MapReduce programming paradigm. While this work does describe a good starting point for a generic MapReduce framework that includes GPGPU computation within the MapReduce paradigm, it only presents the integration of the MapReduce programming paradigm and GPGPU computation within their own "custom" API and MapReduce framework, furthermore no evaluation was performed in regards to image processing.

Abbasi et al. in [16] describe their method of improving the utilization of the GPGPU within Hadoop. They describe that the major drawback of simply utilizing the JNI to integrate GPGPU and Hadoop computation is that it causes the programmer excessive work to

develop additional code within the map and reduce functions to parse the intermediate results that are generated from the GPGPU. They also explain that some optimizations might be avoided as the MapReduce runtime is not aware of the GPGPU. They give an example that many MapReduce runtimes might be competing over a single GPGPU at the same time as they are unaware of each other. Fortunately as development on the CUDA architecture has progressed, this is now only partially true as additional changes to the CUDA framework does assist with this issue to resolve multiple job conflicts. The paper introduces what they call the Surena framework which stitches an existing GPGPU MapReduce implementation into the CPU MapReduce runtime. The authors claim that with this approach, map or reduce tasks are offloaded to the GPU as a whole by the MapReduce runtime given that an implementation is available. Thus, allowing the MapReduce runtime to be aware of the GPGPU and make smart decisions. Unfortunately, a deficiency with their framework is that it seems to be only applicable to the Map phase of the MapReduce model. This causes problems as it is not fully encompassing the entire MapReduce programming model. In addition, they do not perform any evaluations with their work in the areas of image processing or at a larger scale.

Grossman et al. in [23] present their work of HadoopCL. The main goals of this paper are to provide GPGPU integration within Hadoop with the characteristics of "1) an easy-to-learn and flexible application programming interface in a high level and popular programming language, 2) the reliability guarantees and distributed filesystem of Hadoop, and 3) the low power consumption and performance acceleration of heterogeneous processors." The authors claim that HadoopCL is an extension to Hadoop which supports execution of user-written Java kernels on heterogeneous devices. The authors used an open source tool developed by AMD called APARAPI which is a JIT compiler that with their work can achieve "up to a 3x overall speed up and better than 55x speedup of the computational section for MapReduce applications." According to the paper the APARAPI framework takes suitable Java code and translates it to code that can be run on GPGPUs via the OpenCL framework. While

this work would be easy to use and more efficient than writing pure OpenCL code it has a disadvantage of having to translate (APARAPI) Java to OpenCL. This can reduce the amount of efficiency that can be obtained and also limit the complexity of what applications a developer can utilize using the HadoopCL method.

In [40] Zhu et al. present their work of integrating Hadoop with GPGPU computation. The main attention of this paper is to present the approaches of exploiting both CPU and GPGPU resources. The paper presents four methods of incorporating GPGPU computational power within Hadoop. The four methods that are discussed in [40] are JCuda, JNI, Hadoop Streaming and Hadoop Pipes. The paper successfully presents the four different methods of integrating CUDA within the Hadoop framework. Their work concentrated on presenting the varying advantages and disadvantages of each method. Unfortunately for this work, it only focuses on the different methods of integrating Hadoop and GPGPU computation and does not consider scalability in their evaluation, which we plan to address in our work.

Chapter 4

Design and Implementation

The intention of this section is to present and discuss the specifics of our implementation to integrate GPGPU computation within Apache Hadoop. We will be introducing the JCuda library that we utilized to interface the GPGPU and Hadoop. A discussion of the advantages and disadvantages of using the library will be shown in detail. We do this to achieve our goal of demonstrating the benefits of utilizing GPGPU within a cloud based environment to improve the performance of our use cases of matrix multiplication, FFT compuation and image processing. In addition, we will discuss the matrix multiplication, FFT computation, and image processing algorithms we utilized to facility the testing of our work.

4.1 Introduction to JCuda

JCuda[10] is a set of Java libraries that provide bindings for the Nvidia CUDA architecture. Since we want to take advantage of Apache Hadoop which is the Java domain and CUDA is in the CUDA C/C++ domain, there needs to be some form of mediator between the two to facilitate interoperability between the two technologies. JCuda is an open source project that will facilitate this need, in addition to the CUDA bindings, it provides many other GPGPU libraries such as JCublas, JCufft, JCurand, JCusparse, and JCusolver that provide additional functionality for the developer. When using JCuda and Hadoop, the JCuda code written for the MapReduce applications can interact with the CUDA library, which can interact with the GPGPU without having to rely on an external CUDA C/C++ application to execute the work on the GPGPU device. JCuda does this by mirroring the CUDA API and making the CUDA library available within a Java environment for the programmer to access. The underlying JCuda API calls in turn call the associated CUDA

API calls appropriately. However, one item that needs to be made clear, JCuda is not a replacement for writing CUDA kernels (the code that executes on the GPGPU) but is a replacement for interacting with the CUDA device itself (e.g. cudaMemcpy(...)).

For example, if a developer is wanting to transfer data from their Java application to a GPGPU device, JCuda allows you to initialize the GPGPU device, allocate the memory, copy data to the GPGPU and execute a CUDA kernel in the same as you would if you were writing a CUDA C/C++ application. This enables us to reuse the data structures within Hadoop that we have already created directly without having to create an external CUDA C/C++ application to perform the work on the GPGPU.

4.1.1 Advantages of JCuda

Jcuda provides a great amount of flexibility in utilizing CUDA to improve applications with the computational advantages of GPGPUs. A significant advantage to using JCuda, is that there is a small learning curve to start using JCuda and when it is used there is minimal loss of functionality versus native CUDA C/C++ code. If the developer knows how to use CUDA C/C++, they will feel at home using JCuda. This is a result of JCuda being essentially a one-to-one API mirror of CUDA. In addition, the project is open source and utilizes the MIT license. This is the least restrictive open source license available, that allows reuse and modification of the library which we can adapt to our needs if required. It is also an extremely mature project in that it has been around since 2009 which is by the most part as long as the CUDA architecture itself has been available. JCuda is also fast, and does not impose significant overhead over native CUDA C/C++. While JCuda does provide some great advantages for our work, it does come with some disadvantages that we will discuss.

4.1.2 Disadvantages of JCuda

The first possible concern is that it is a small project with no backing from a big corporation. There is only one main contributor to this project which could pose problems, as he would be the bottleneck for updates or bug fixes. This could also lead to a stale project if for some reason the project owner no longer considers the project a priority. However, updates to JCuda are typically performed within a month of a major CUDA release by NVidia. Not all functionality is tested extensively, and it is up to the responsibility of the user of the library to validate correctness. This project has been active since 2009 and has been kept up to date with every release of CUDA to date, but there is a slight delay between when NVidia releases their SDK to when this project gets updated. While we have discussed a few of the short comings of the project itself we also want to point out a programmability deficiency which is related to the issue of Java not having support for pointers pointers. To overcome this, the project implements a Pointer class that the user can treat similar to "void*." While Java references are similar to pointers in C, they are not suitable for emulating native pointers as they don't allow pointer arithmetic and "references to references" are not possible. It is possible to create pointers to pointers to allocate 2D arrays, however there are limitations on how these pointers may be used. For example, pointers may not be written to. However, the project does note that future versions may support this functionality at a later date.

4.1.3 Integration of JCuda with Hadoop

The main JCuda package consists of two main APIs, the driver API and the runtime API. Additional libraries based on the native implementations are also included in separate packages, namely JCublas, JCufft, JCurand, JCusparse, and the JCusolver. All of the previously mentioned libraries have their own JAR files that were generated based on their respective native CUDA C/C++ implementation from NVidia. These libraries can be used

to enhance the functionality and capability of the programmers application with greater ease.

To get JCuda applications to run properly within Hadoop, changes to the Hadoop configuration is required. These changes are required to make Hadoop aware of the GPGPU and the CUDA library, as well as making the JCuda library accessible to the MapReduce application. If an external CUDA kernel is required, the MapReduce application will also need to be modified to obtain the PTX or CUBIN file (executable GPGPU code) from HDFS for the application to run appropriately.

4.2 Methods of Integration

For our work to make Hadoop aware of the GPGPU and the CUDA library we must make modifications to the mapred-site.xml file and change the property of "mapreduce.map.env" and "mapreduce.reduce.env" to the value $LD\_LIBRARY\_PATH = /usr/lib/JCuda/lib64 : /usr/local/cuda-6.5/lib64$. Obviously this has to be done on every slave and is dependent on the systems configuration. At this point for any Map or Reduce job the system will know where the CUDA library files and the JCuda library files are located. This is required, as JCuda utilizes the function calls in these libraries to interact with the GPGPU. The next requirement is on how to make the users application aware of JCuda. Several methods can be utilized, all with their own advantages and disadvantages.

4.2.1 Method: -Libjars

The first method to achieve this task is to use the libjars option on the command line interface. When executing a Hadoop job the developer can pass the JCuda Jcuda jar to the application easily by passing the "-libjars" parameter. The only requirement on this method is that the application has to implement the Tool interface[6]. The advantage is that it is easy to add any other JCuda libraries. In addition, the JCuda jar files will automatically be

distributed to the slave nodes as needed. The disadvantage is that it is only usable on the CLI.

4.2.2 Method: Hadoop Classpath method

The second method is to add the jar files to each machine and change the HADOOP_CLASSPATH environment variable on each slave to point to the JCuda jar files. While this is probably the simplest method, it is also the most cumbersome and will not scale to any degree. This method would be ideal for a single node setup or small development environment. It would not be able to scale to many nodes and would be difficult to maintain.

4.2.3 Method: Fat jar

The third method would be to use an external tool such as Maven to create what it calls a "fat jar." A "fat jar" is packaged in way that all classes required to run the application are included in a single jar file. This can be any external jar or class files, including Hadoop libraries, other 3rd party libraries and JCuda. There is no method of selectively adding only certain jar files to a "fat jar." Due to this, it creates an enormous jar file that would need to be distributed. Depending on the project, the file could be substantial in size. The advantage to this is that there wouldn't be any external configuration required and the entire project would be self contained in the "fat jar."

4.2.4 Method: Distributed Cache

The "Distributed Cache" method uses the Hadoop Distributed Cache to provide access to the JCuda Jar files for the slave nodes to access on HDFS. This functionality is provided by the Hadoop framework itself for making small files available to all slave nodes. The first step required to utilize this method requires the user to upload the JCuda Jar files to the HDFS. In addition, the MapReduce application needs to be modified to add a method that calls the addFileToClasspath within Hadoop to add the class file to Hadoops distributed cache[5].

The advantage is that it allows the developer to add the JCuda libraries programmatically which gives the most flexibility. The class files are managed automatically on the slave nodes within Hadoop's distributed cache. On the other hand, this requires the modification of the application and setup tasks for the management of the jar files (e.g. putting the jar files into HDFS).

4.2.5 Making PTX and CUBIN files available to Hadoop

In some instances there might be a need to write a custom CUDA kernel to perform a certain optimization for a particular piece of hardware, or due to a deficiency in a particular library. The custom kernel will need to be compiled with NVCC down to a PTX or CUBIN file which contain the custom kernel function in a format that the GPGPU can execute. The PTX or CUBIN file is required to be transfered to the GPGPU for the work to be completed on the GPGPU. This poses a challenge as each node would be required to have this code available. Without the PTX or CUBIN file on each Hadoop slave the application would not work correctly. This issue should not limit the developers ability to utilize GPGPU computation within Hadoop. This is a similar issue we described above when making the JCuda jar files available to Hadoop. To overcome this hurdle there are two methods that can be employed to resolve this issue: 1) Write the kernel code and bundle the code in a string within the Java application, and then compile the kernel on the fly within the MapReduce application when it is ran on Hadoop. 2) Write the kernel and compile the file down to a PTX or CUBIN file. Once you have the PTX or CUBIN file, store the file within HDFS and use the Hadoop distributed cache to make the MapReduce application aware of the GPGPU executable files. To do this the developer would need to call the "addCacheFile" method within the MapReduce job. At this point the ptx file is available to the application to utilize on all of the slaves. Both methods would require modification to the application but option two is more modular and orders of magnitude more manageable to maintain.

4.3 Use case methods utilized

For our work, we chose to utilize the "-Libjar" method for two of our use cases and "fat jar" method for the third way of making Hadoop aware of JCuda and the GPGPU. Only one of our use cases required the use of a PTX file, and for this case, we used the previously described second method to make the PTX file available to all of the slave nodes.

4.4 Applicable use cases

In the following subsections we present three different methods that exploit the capabilities of the GPGPU and Hadoop together. These methods utilize the methods we discussed above, such as the use of additional libraries and the ability to utilize custom kernel files (PTXs) within Hadoop. We provide the background and the tasks that are performed.

4.4.1 Single precision general matrix multiplication

Matrix multiplication is a great use case to demonstrate the computational power of the GPGPU. We want to demonstrate the ability of using the built in libraries provided by the JCublas library. The JCublas library provides the Java implementation of the CUDA cuBLAS (Basic Linear Algerbra Subprograms) library. We utilized a naive implementation of a matrix multiplication function and the CUDA cublasSgemm() function which computes the following, C = α * A * B + β * C [3]. For our work we performed the following: 1) Generated random single precision floating point data for matrix A, B and C for the input; 2) Passed the matrices to the naive implementation and stored the result. To compare against the GPGPU we saved how long this function took; 3) Transfered the input matrices to the GPGPU and called the cublasSgemm function providing the location of the input matrices and the location of where the output matrix should be stored; 4) transfered the GPGPU resulting matrix from the GPGPU; and 5) ran a validation against the resulting matrices of

the GPGPU and the CPU. If all elements of the matrices were within $10e^{-6}$ of each other, then the functions were of success and we compared the processing times of each.

Data was generated prior to processing and loaded into HDFS. A file consisted of each matrix on a single line seperated by a colon and each element delimited by a comma, each row was a new set of matrices. We utilized the NLineInputFormat class to create the input splits. A Hadoop job was created for the above process and then executed. To reduce complexity and improve clarity this is a Map only job.

4.4.2 Fast Fourier transformations

Like matrix multiplication, the computation of FFTs is another highly parallelize algorithm that can benefit greatly from the GPGPU architecture. With the addition of Hadoop we can gain even more performance. We utilized a similar process implementing the FFT use case as we did with the matrix multinational method above. However, instead of using a naive implementation of the CPU version, we used a third party library. JTransforms[11][38] is a mature, well tested and is touted as one of the fastest FFT libraries built fully in Java. We utilized a 1D complex to complex forward FFT for this use case. As before we generated random data, passed that data into the JTransforms and the JCufft 1D complex to complex methods respectively and compared the output data. To integrate into Hadoop we had the same implementation and process as the matrix multiplication except for the input data.

4.4.3 Image processing

As the other uses cases are only synthetic in practice we wanted to demonstrate the usefulness on a more practical application. Image processing can be utilized to provide great visual tools, in addition to providing insight into many other area of research. With the explosion of the "Big Data era", more and more images are being created that can be utilized in research, however it can pose a challenge to deal with this data.

Our goals is to demonstrate the capabilities of the GPGPU and Hadoop to tackle these types of real world applications. For this work we chose to implement the Gaussian blur algorithm that we apply across our dataset. We chose to use the Gaussian blur algorithm due to its highly parallel nature, and its balance between the amount of computation required and data transfers. We feel this will effectively demonstrate the purpose of our work in an appropriate manner for real world applications. The Gaussian blur algorithm also known as a Gaussian smoothing function is extensively used in applications such as Photoshop to reduce noise and detail, computer vision, in addition to assisting with edge detection.

4.4.4 Image dataset

For our work we constructed image datasets consisting of satellite and astronomical images from NASA. The image data was retrieved from [13] and was modified to fit our needs. The base set of images was 2GB, and consisted of images that were greater than 8192x8192 pixels. For the tests that required additional data, the images were duplicated to generate the entire dataset. For the datasets that required specific dataset sizes, the image resolutions were modified to fit the constraints that were required for our analysis (e.g. 512x512, 1024x1024, 2048x2048, 4096x4096, 8192x8192).

4.4.5 Implementation of Gaussian blur

To perform a Gaussian blur on an image consists of two main tasks: 1) the creation of the Gaussian matrix and 2) performing a convolution of the Gaussian matrix on the image. Using the Gaussian function[31] $G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$, where x is the distance away from the original on the horizontal axis, y is the distance from the origin of the vertical axis and σ is the standard deviation of the Gaussian distribution we generated a single 5×5 Gaussian matrix with a standard deviation of 1 to apply to our datasets. We utilized the same Gaussian matrix for all of our datasets. The next step was to compute the convolution of the Gaussian matrix and our input image. The implementation of the convolution algorithm

can be seen in algorithm 2. This function is performed for each color channel. This same algorithm was translated into a custom CUDA kernel so that it could be executed on the GPGPU.

Algorithm 2 Convolution of Gaussian blur algorithm[1]

```
1: function GaussianBlur(gMatrix[], cChannel[], iH, iW)
2:     for i < iH − 2; i++ do
3:         for j < iW − 2; j++ do
4:             temp ← 0
5:             for ii = −2; ii ≤ 2; ii++ do
6:                 for jj = −2; ii ≤ 2; jj++ do
7:                     temp = temp + cChannel[(i + ii) ∗ iW + j + jj] ∗ gMatrix[jj + 2][ii + 2]
8:                 end for
9:             end for
10:            cChannel_new[i + iW + j] ← temp
11:        end for
12:    end for
13:    return cChannel_new
14: end function
```

The data flow for the Gaussian blur filter on Hadoop consisted of the input data, which was split into 128MB chunks, and sent to a Map function. The key to the Map function is the index location of the image and the value was the image data itself. Each Hadoop node would then process each image accordingly (CPU or GPGPU implementation) and then write the resulting image to HDFS. The GPGPU implementation differed slightly during processing in that, JCuda was utilized to transfer the gMatrix[], cChannel[], iH, and iW values to the GPGPU's global memory. At this point JCuda instructed the GPGPU to execute the custom GaussianBlur CUDA kernel. The data was then retrieved from the GPGPU global memory and then written to HDFS as it was done in the CPU only version. The next image would be passed to the Map function and the same process would be performed. There is no Reduce function as our output of the Map function is our desired result.

Chapter 5

Evaluation

In this chapter, we will demonstrate the effectiveness of our work on integrating the GPGPU with Apache Hadoop for two synthetic benchmarks (matrix multiplication and FFT calculation) and a real world application (image processing with utilizing the Gaussian blur function). First we will discuss our experimental environment and test platform, our testing methodologies and the results of the three different use cases that were presented above. In the graphs below we utilized the term "GPU" instead of GPGPU for readability purposes.

5.1 Experimental environment

5.1.1 Cluster setup

The cluster consisted of an Intel Xeon X5650 @ 2.67ghz, 24GB of system memory, Western Digital SATA 500GB 7200RPM disk drive and a Tesla M2050 with 3072MB of memory. The cluster interconnect was connected via 1Gbps Ethernet. All machines have identical configurations.

5.1.2 Hadoop configuration

Our evaluation setup used Hadoop-2.5.2 with Java JDK 1.7. In all of our tests a single node was dedicated as the ResourceManager and Namenode manager. In our evaluation tests the number of systems that are noted on the evaluation is the exact number of slaves for that particular test. For example, if a test used four nodes, four slaves were used with 1 additional node acting as the ResourceManager and Node manager, bringing the total

Parameter Name	Value
yarn.nodemanager.resource.memory-mb	16GB
yarn.scheduler.maximum-allocation-mb	15GB
yarn.scheduler.minimum-allocation-mb	1GB
mapreduce.map.memory.mb	13GB
mapreduce.map.java.opts	10GB
yarn.nodemanager.vmem-pmem-ratio	2.1
dfs.block.size	128MB
dfs.replication	3

Table 5.1: YARN parameters

server count to 5. Due to the memory requirements of some of the input data we used in our evaluation, certain YARN configuration settings were modified to accommodate this issue. The important configuration settings of our Hadoop cluster can be seen in Table 5.1.

5.2 Single precision general matrix multiplication

Matrix multiplication is a great use case to demonstrate the power of GPGPU computation due to its high amount of parallelism and requiring a high amount of throughput. For this section of the evaluation, we utilized the cublasSgemm() function in the JCuda, JCublas library for the GPU version and a naive implementation on the CPU for the of a single precision general matrix multiplication function for our workload. Each input matrix of $A[][]$, $B[][]$ and $C[][]$ were 500x500, 1000x1000, 2000x2000, 3000x3000 and 4000x4000 for each data point. The CPU and the GPGPU evaluations were ran on each dataset three times and the results were averaged. For the CPU version two different times were measured 1) the amount of time spent performing the computation (teal), 2) the amount of time required for all I/O (purple). For the GPGPU version, we measured three different times 1)the kernel computation time 2)the GPGPU bus time and 3) the same I/O latencies that are imposed on the CPU version. The kernel computation (red) is the amount of time the GPGPU spent performing actual computational work. The GPGPU bus time is the amount of time to transfer the data to and from the GPGPU.

In Figure 5.1 we show the substantial improvement that can be gained from utilizing GPGPUs. For illustration purposes we normalized the GPGPU times to the CPU times. We can see for the set of 500x500 matrices and even for the 1000x1000 matrices the amount of time spent moving the data over the GPGPU bus is a substantial portion of the total computation time for the GPGPU. As a result of the matrices being smaller, the amount of time for the CPU to perform the computational work isn't enough to offset the use of the GPGPU. However, as soon as we move up to larger matrices where the computational time becomes more substantial for the CPU we can see performance improvement from the GPGPU. In particular, by the 3000x3000 matrix size the vast majority of the total time is essentially I/O for the GPGPU version.

To make a direct comparison between the computation times of the GPGPU and CPU we plotted the performance increase for the GPGPU over the CPU on the orange trend line. For the 500x500, 1000x1000, 2000x2000, 3000x3000 and the 4000x4000 we were able to achieve a performance increase of .19×, .35×, 18.68×, 90.10×, and 256.63× respectively. The GPU Kernel time in each instance was $< 1\%$ of the total amount of time in each case.

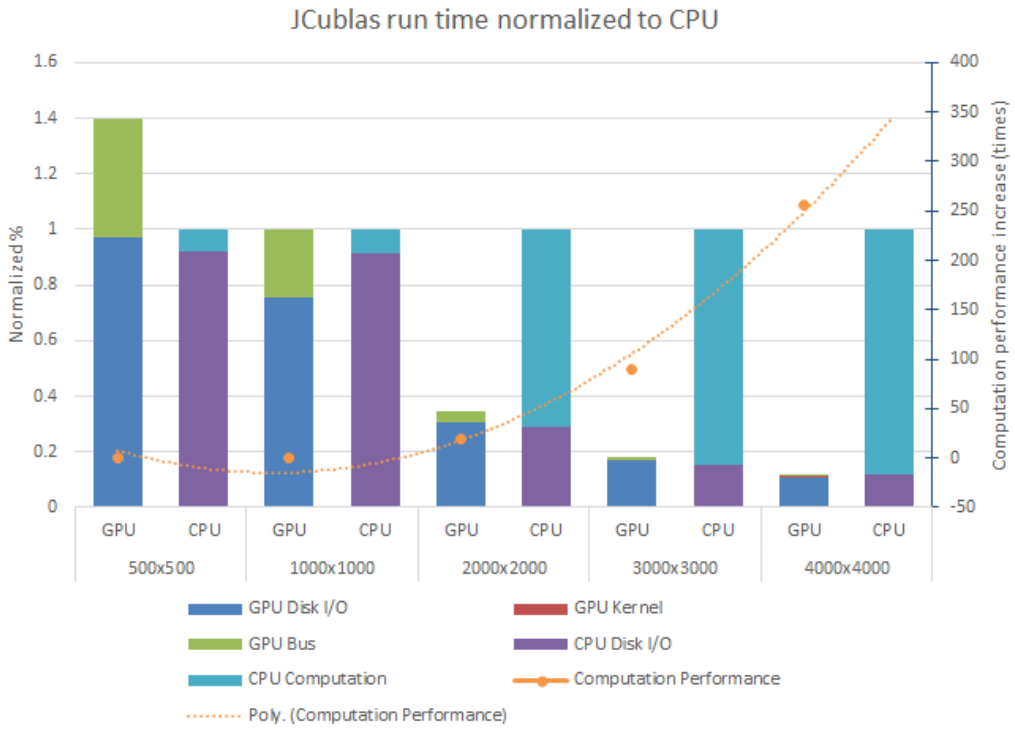

Figure 5.1: JCublas normalized to CPU

5.2.1 The Scalability of JCublas with Hadoop

To demonstrate the scalability of GPGPU computation with Hadoop we utilized a larger dataset of 64 matrices and scaled the cluster size from 4 nodes to 16 nodes. The 64 matrices were split evenly across the 4, 8 and 16 node tests. In figures 5.2, 5.3 and 5.4, on the left vertical axis we show the total run time of the entire MapReduce job to complete the workload on the GPGPU and CPU version respectively. On the right vertical axis we show the overall performance improvement we are able to obtain with the GPGPU over the CPU. We can see that for 4 nodes we were able to achieve a .75×, 1.14×, 2.56×, 3.03×, and 6.91× performance gain for 500x500 to 4000x4000 matrices respectively. For 8 nodes, we were able to achieve a .77×, 1.12×, 2.15×, 2.88×, and 6.16× increase. For 16 nodes a .54×, .74×, 1.77×, 3.34×, and 4.90×. From this we can see that the GPGPU provides a great deal of performance improvements while maintaining scalability. To be explicit, we were able to achieve overall performance gains of .75×, 1.13×, 2.56×, 3.03× for 4 nodes, .78×, 1.12×,

2.15×, 2.88× and 6.16× for 8 nodes, and .54×, .74×, 1.77×, 3.34× and 4.9× for 16 nodes for 500x500, 1000x1000, 2000x2000, 3000x300 and 4000x4000 sized matrices respectively. These improvements also include disk and any other I/O times.

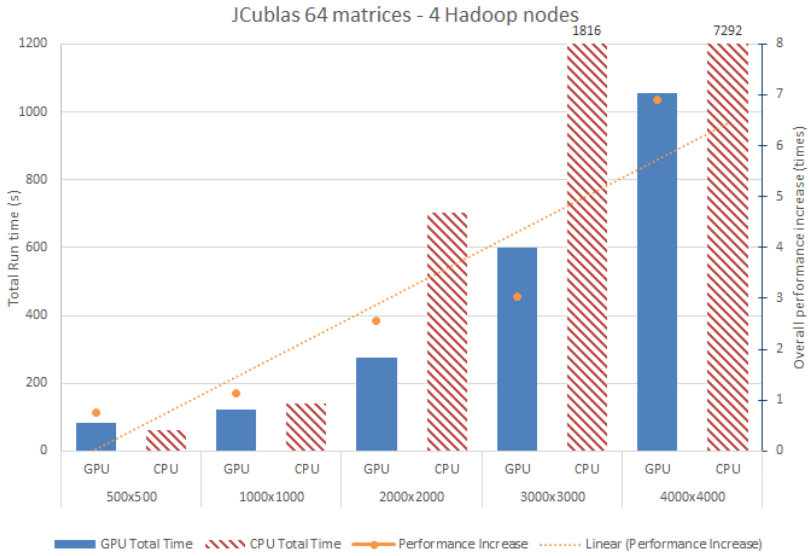

Figure 5.2: JCublas 4 Nodes

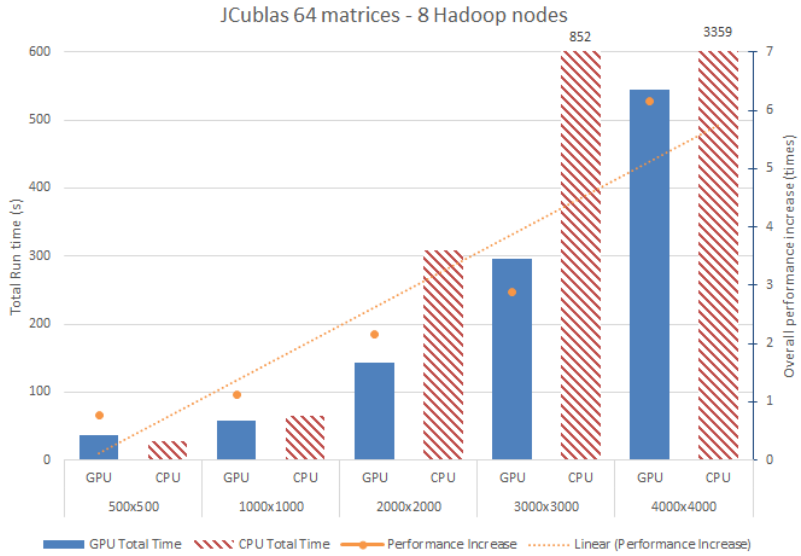

Figure 5.3: JCublas 8 Nodes

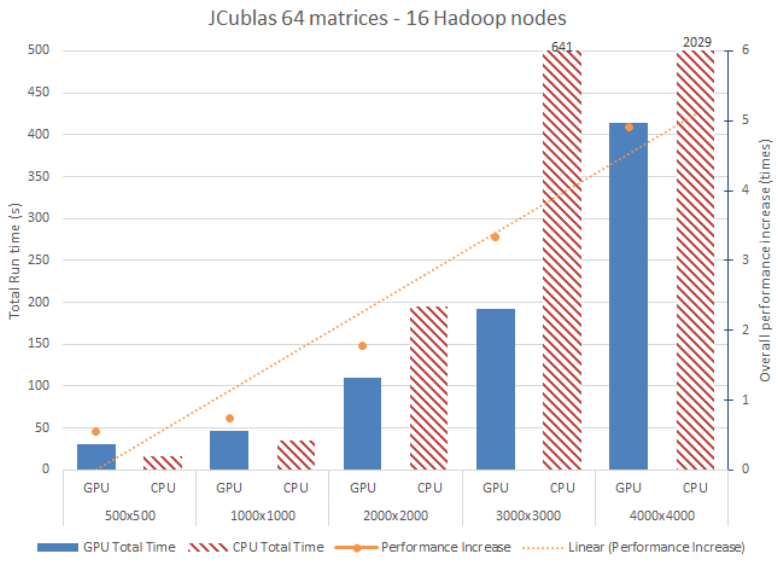

Figure 5.4: JCublas 16 Nodes

5.3 Fast Fourier transformations

In this section we present our evaluation of utilizing the JCuda library JCuFFT to compute the 1D complex to complex forward FFT. In Figure 5.5 we dissected the processing times in the same manner as we did in the matrix multiplication section for the CPU. The GPGPU dissection is slightly different in that the GPGPU kernel time includes both the computation time and the bus transfer time. This is due to how the JCuFFT library works in that the cufftExecC2C)() function handles the transfer internally. Test cases with 2, 4, 8, 16, 32 and 64 million elements were ran on the CPU and GPGPU. The FFT use case exhibits the same performance trend as the matrix multiplication, as the number of elements grows the CPU computation becomes the significant portion of the total processing time. As the number of elements increase for the GPGPU tests the percentage of the total processing time compared to the I/O becomes smaller. In addition, we can see that by the time we reach 64 million elements the GPGPU has essentially removed the computational expense compared to the CPU and made the entire process completely dependent on I/O. A direct comparison of computation-only speedup when utilizing the GPGPU can be seen on the

orange trend line. With 2, 4, 8, 16, 32, and 64 million elements we were able to achieve an improvement of 3.96×, 7.56×, 9.61×, 11.73×, 12.90× and a 13.73×, respectively.

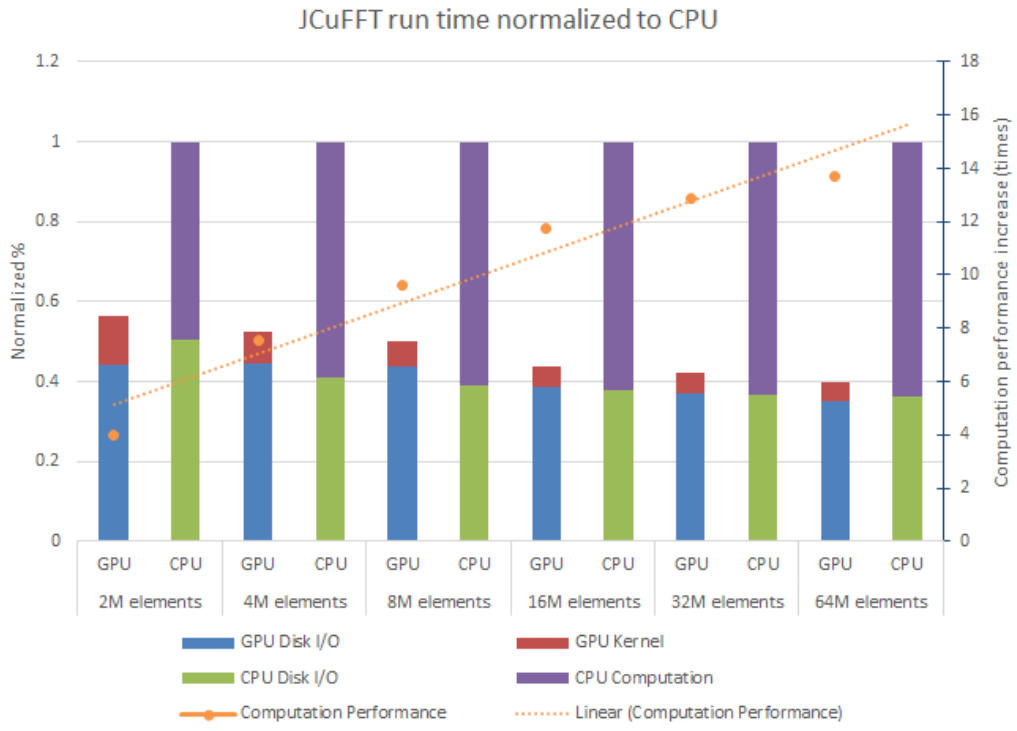

Figure 5.5: JCufft normalized to CPU

5.3.1 FFT variations in Hadoop inputsplit sizes

For this section, we want to demonstrate the impact of the input split sizes for Hadoop on performance. For this work we utilized the NLineInputFormat.class to create the input splits. Since each line contains the matrix that needs to be computed, each line could be a single input split. However, reducing the size of the input splits too small will cause a significant amount of overhead creating and destroying the jobs. In the opposite case, if the input split size is too large, jobs might take too long and you would not effectively utilize the cluster by having idle slaves.

We used four (three slaves and one master) nodes in the following figures we can see how the inputsplit size will affect overall performance. We used 1600 separate matrices in sizes of 2M, 4M, 8M, 16M, and 32M elements, broke them into sizes of 25, 50, 100, 200

and 400 records per each inputsplit. This test demonstrated that as the records inputsplits are small and the element sizes are small, the performance of the GPGPU can't overcome the amount of initialization process that is required for each split. More inputsplits result in more initialization time of the GPGPU. Since the GPGPU takes roughly 2 seconds to initialize for each split, there needs to be enough work to compensate for this time. In Figure 5.6 and 5.7 for 25 records per split we can see this issue. However, as the elements get larger the initialization time becomes less significant. When we reach 32M elements we can see that the variation isn't as significant as with less data. This is because the I/O has become the limiting factor. These test results show that much computation work needs to be applied against the GPGPU to overcome the initialization issue. In addition, it is best to avoid as much I/O as possible and select an optimal inputsplit size that can effectively use the cluster. We can see this result in all five figures where the 400 records per split doesn't effectively utilize all nodes and the effective performance drops.

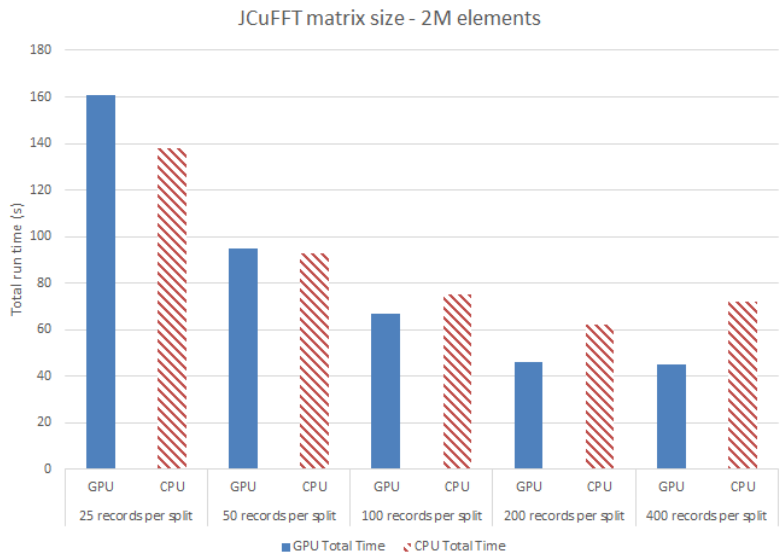

Figure 5.6: JCufft matrix size - 2M elements

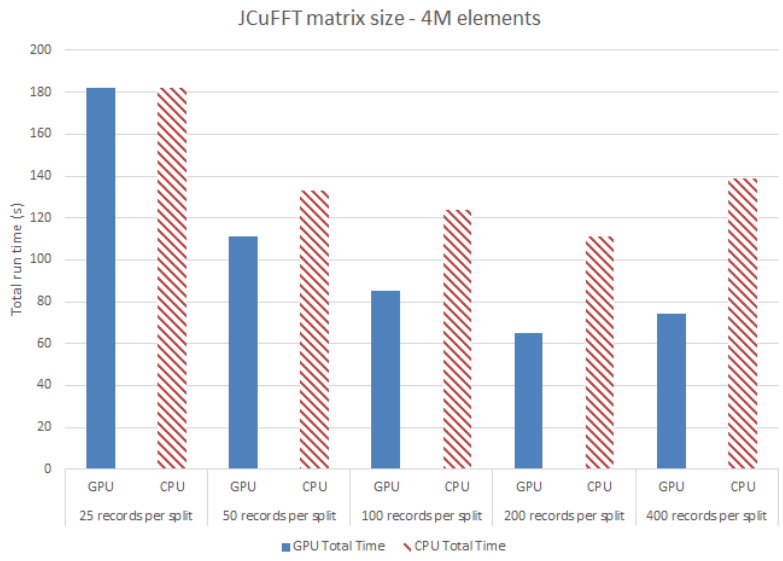

Figure 5.7: JCufft matrix size - 4M elements

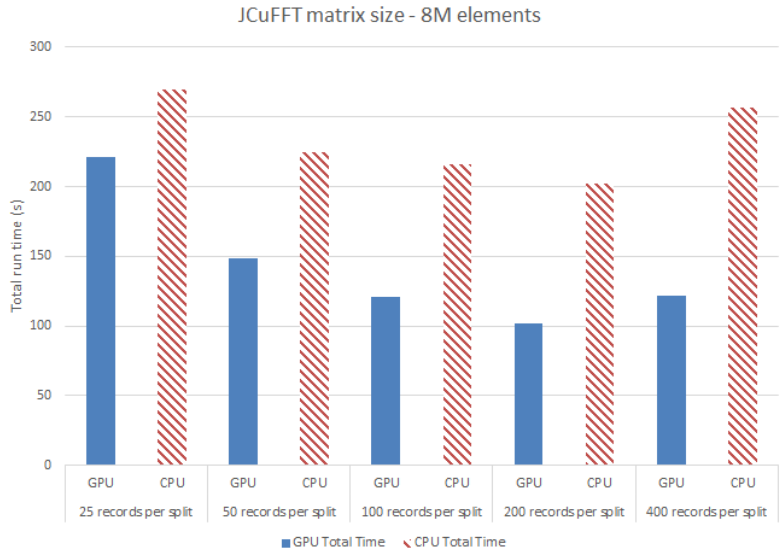

Figure 5.8: JCufft matrix size - 8M elements

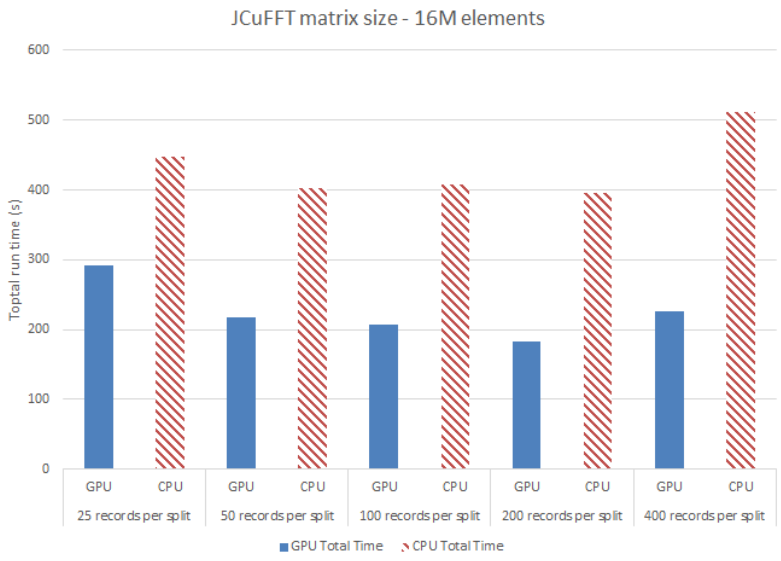

Figure 5.9: JCufft matrix size - 16M elements

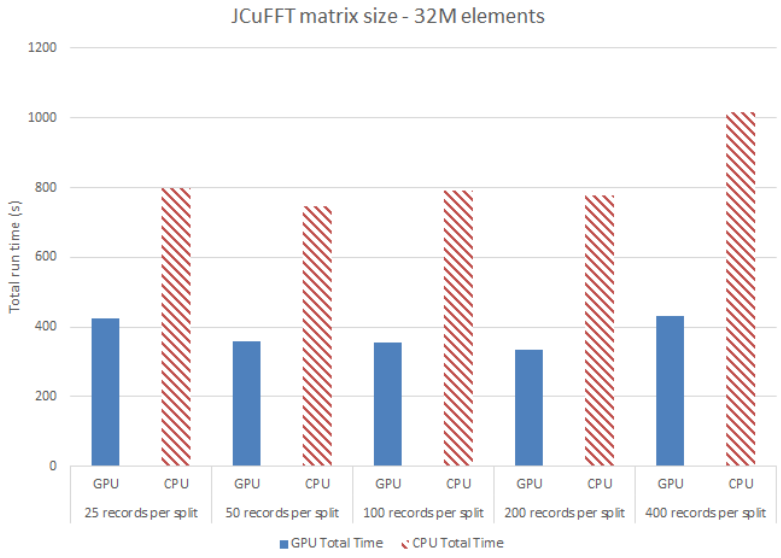

Figure 5.10: JCufft matrix size - 32M elements

5.3.2 FFT with Hadoop scalability

In this section we want to demonstrate the power of combining the powerful computational capability of the GPGPU we demonstrated in the previous section, alongside the benefits of Hadoop to even further improve the performance and scalability of the FFT computational tasks. For our workload, we utilized 1600 randomly generated floating point 1D

matrices in sizes of 2, 4, 8, 16, 32 and 64 million elements and computed the 1D complex to complex forward FFT of those arrays. The 1600 floating point arrays were split evenly among the 4, 8, and 16 nodes during each test. Figures 5.11, 5.12, and 5.13 show the total computation time of each test on the left vertical axis. The right vertical axis is associated with the orange points that show the amount of overall performance that was achieved (including disk and any other I/O required).

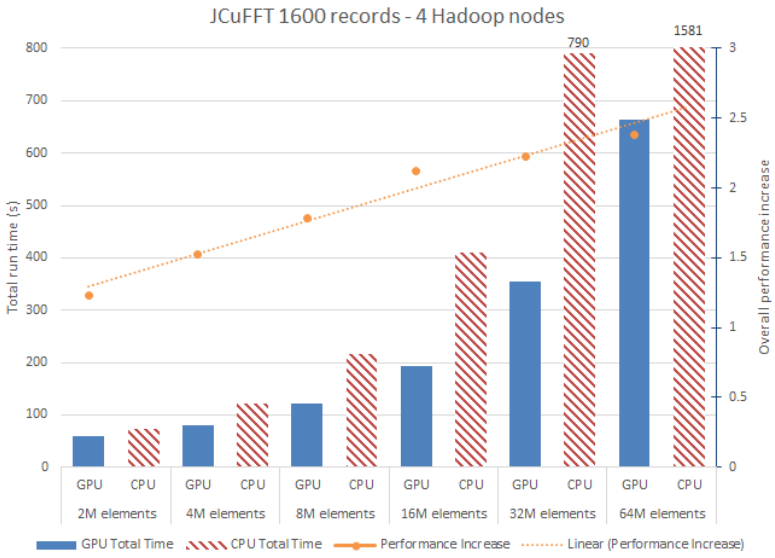

Figure 5.11: JCufft 4 Nodes

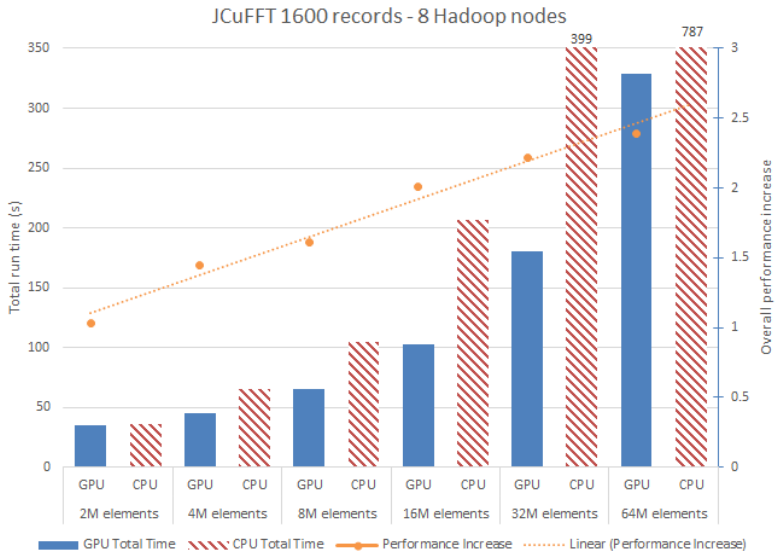

Figure 5.12: JCufft 8 Nodes

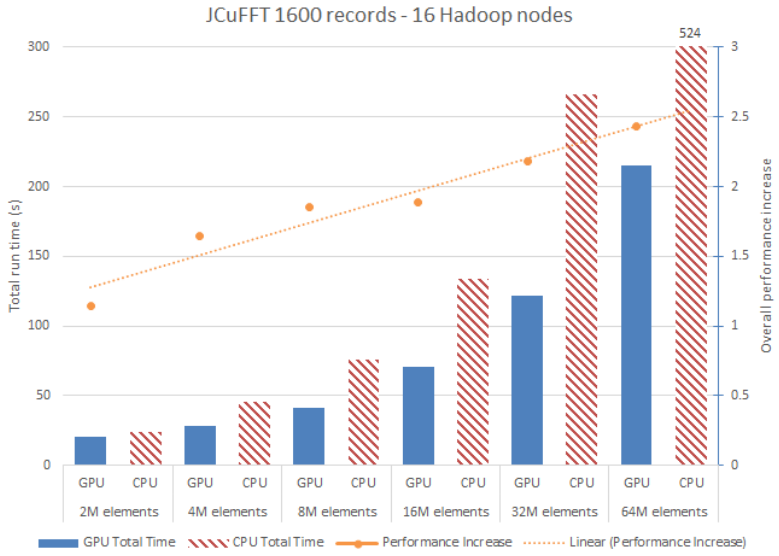

Figure 5.13: JCufft 16 Nodes

5.4 Image processing with Gaussian blur filter

In this section we want to demonstrate the effectiveness of our work being applied to a real world application. In Figure 5.14 we show the a breakdown of processing times for a 512x512, 1024x1024, 2048x2048, 4096x4096 and 8192x8192 set of images. As we have

done in the previous experiments, we normalized the GPGPU processing times to the CPU run time to show the effectiveness of the GPGPU compared to the CPU in a stacked bar chart. The bar chart is broken down into computation time (teal) and disk I/O (purple) for the CPU. The GPGPU bar chart stack shows the GPGPU kernel time (red), the GPU bus (green) time and the disk I/O (blue). We see again that the GPGPU kernel time and GPGPU bus time drastically improve the computation portion of every single data point, compared to the CPU computation time. The 8192x8192 dataset shows that the percentage of time the GPGPU spends processing the image is roughly 3% of the entire runtime and that the disk I/O and other latencies involved in the task are now the major bottleneck. This shows that the GPGPU essentially removed the computational wait time imposed by the CPU. The orange points on the right vertical axis of Figure 5.14 show the total improvement of the computational processing time (e.g. the amount of time the CPU or GPGPU spent performing calculations). The 16.68× performance improvement for an 8192x8192 image is consistent with what other researchers and software packages are able to achieve[14].

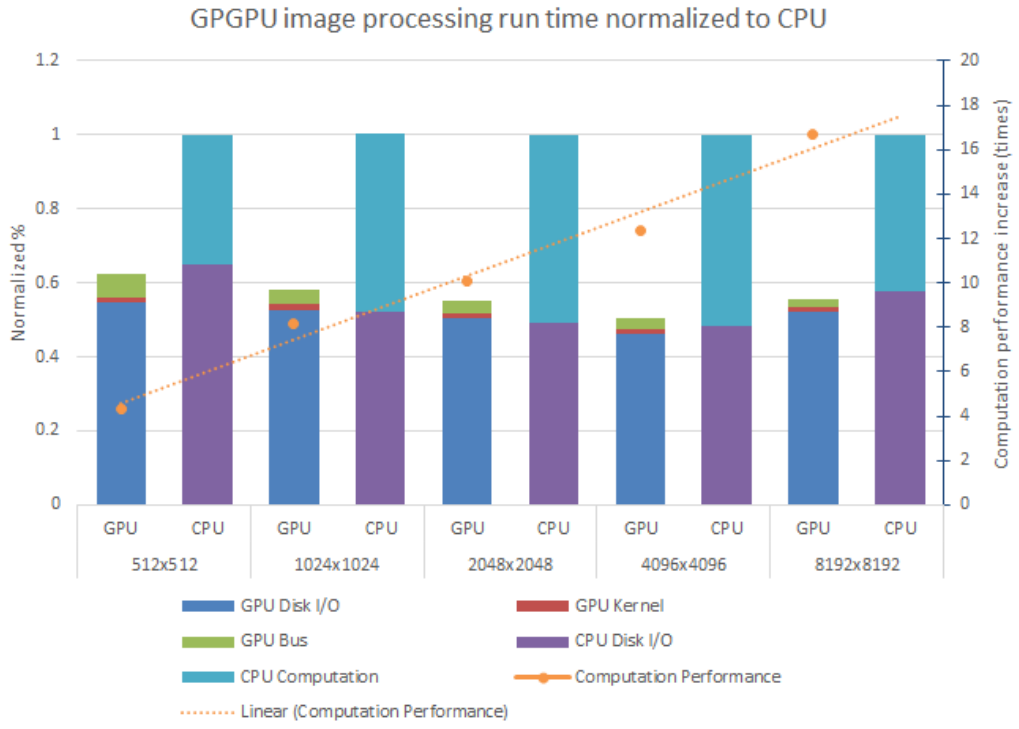

Figure 5.14: GPGPU Gaussian blur normalized to CPU

5.4.1 Scalability - Gaussian blur random dataset size

In this section, we discuss the overall improvements we were able to achieve when processing a mixture of a random dataset of images. This set of evaluations is intended to more closely represent a real world dataset. In the previous examples we only tested static image sizes to show the performance gains with specific image sizes. Unfortunately real world datasets are not typically of this nature, but are a mixture of image sizes. To perform this test we chose to process 10, 20, 40 and 80 GB of random image sizes. The images ranged from roughly 64x64 to 8192x8192 in size, but were not necessarily square. As shown by Figures 5.15, 5.16, and 5.17, we can see that we were able to maintain roughly a 1.5× performance improvement across all image dataset sizes from 10GB to 80GB. In addition, we were able to maintain this performance increase while scaling up the node count from 4 to 16 nodes. The left vertical bar shows the overall run time of the CPU and GPGPU version.

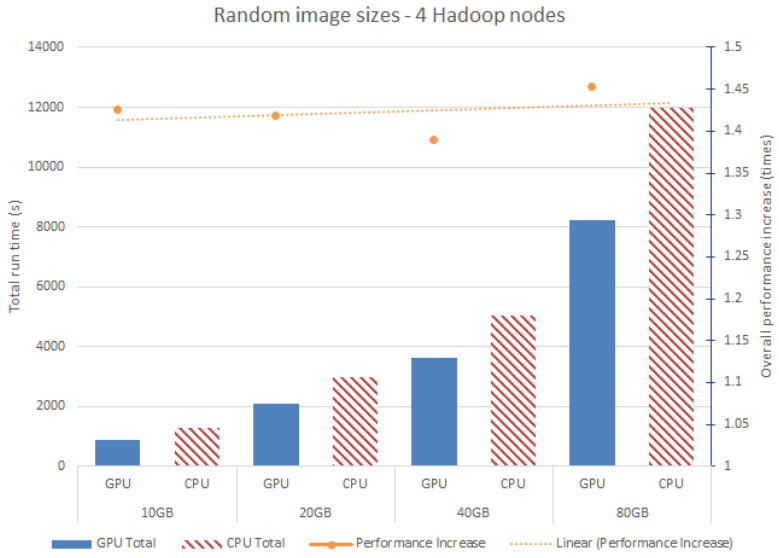

Figure 5.15: Gaussian blur random dataset - 4 Nodes

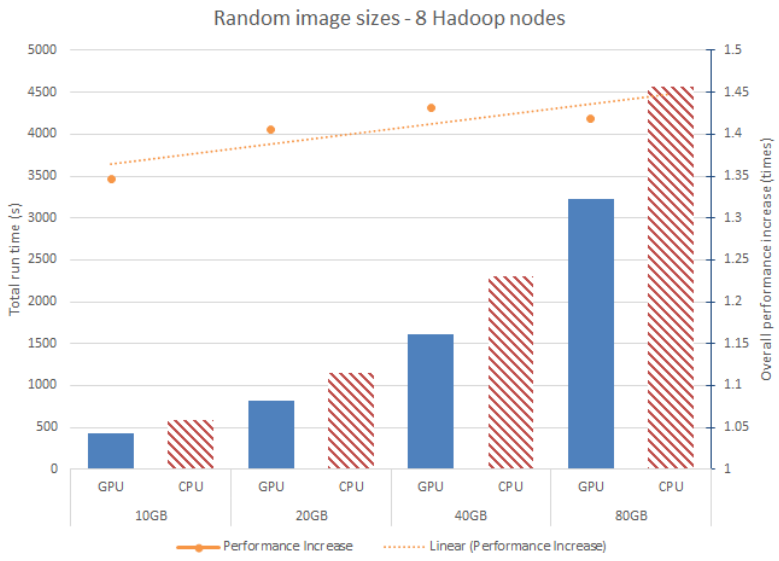

Figure 5.16: Gaussian blur random dataset - 8 Nodes

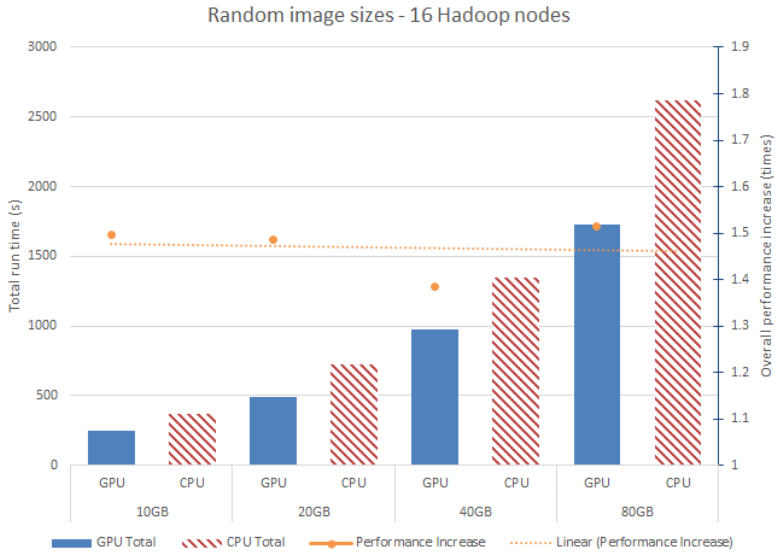

Figure 5.17: Gaussian blur random dataset - 16 Nodes

Chapter 6

Conclusion and future work

With the "Big Data era" producing information at an unprecedented pace, the requirements for improved computational methods and performance are becoming increasingly apparent. The work presented in this thesis can help make improvements in a very broad set of applications, be it in medical, scientific research, consumer products, or signal processing. This thesis provides a general coverage on the background of related technologies. This includes the understanding and the need for this work because of the "Big Data era", the early days of how data was processed and how it can be utilized today with GPGPU computational power, with the scalability of the Apache Hadoop framework.

We presented JCuda for our work, which was the library that was utilized to allow the interaction between the two technology domains of CUDA C/C++ and Java. The pros and cons of the JCuda library were presented to show the benefits that can be provided along with the drawbacks that might be of concern. We discussed the implementation methods of integrating the JCuda library within the Hadoop framework in addition to two different methods of adding external kernel dependencies for certain applications that have that requirement. We discussed three different use cases and presented their implementation that we utilized for our test evaluation. We feel that the three use cases are representative of the types of workload that would demonstrate the capability of the GPGPU and the scalability of Hadoop. We then demonstrated an extensive evaluation on the three chosen use cases. A dissection of the difference of computational time that was required to compute the respective tasks were completed for each use case. We also demonstrated the scalability of each use case using the GPGPU's total run time compared to the CPU's total run time for up to 16 compute nodes. In addition, for the image processing section we provided

evaluations on random data sizes and a ratio based approach to demonstrate where the user could obtain the most effective results.

While we were able to achieve a substantial amount of performance in certain cases, there is always room for improvement. For our future work we plan to take advantage of YARN's scheduling system to be aware of the GPGPU as a computational resource. This would allow the YARN schedule to make better decisions on where to send data. Improved usability would also be a greatly welcomed improvement, such as adding OpenCV to assist in providing a well known image processing library to aid in development. Thus, improving the usability along with taking advantage of any computational optimizations.

In conclusion, we hope to have effectively demonstrated the usefulness, practicality and the gains that can be realized from the combination of these two technologies and we hope that it provides a useful set of information that aids and further advances the usefulness for other areas of research research.

Bibliography

[1] *Fastest Gaussian blur*, Accessed June 2015. http://blog.ivank.net/fastest-gaussian-blur.html.

[2] *NVidia Architecture*, Accessed June 2015. https://incisiveradar.com/gpgpu-parallel-computing/.

[3] *Nvidia CUDA Cublas*, Accessed June 2015. http://docs.nvidia.com/cuda/cublas/#axzz3dErztsQo.

[4] *NVidia Programming Guide*, Accessed June 2015. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz3dErztsQo.

[5] *Apache Hadoop - Distributed Cache API*, Accessed March 2015. https://hadoop.apache.org/docs/current/api/org/apache/hadoop/filecache/DistributedCache.html.

[6] *Apache Hadoop - Interface Tool API*, Accessed March 2015. https://hadoop.apache.org/docs/current/api/org/apache/hadoop/util/Tool.html.

[7] *Apache Hadoop Ecosystem*, Accessed March 2015. http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.1.15/bk_using-apache-hadoop/content/yarn_overview.html.

[8] *Apache Hadoop Ecosystem*, Accessed March 2015. http://hortonworks.com/hadoop/yarn/.

[9] *Apache Hadoop YARN Architecture*, Accessed March 2015. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[10] *JCUDA - Java bindings for CUDA*, Accessed March 2015. http://www.jcuda.org/.

[11] *JTransforms*, Accessed March 2015. https://sites.google.com/site/piotrwendykier/software/jtransforms.

[12] *MapReduce Word Count Example*, Accessed March 2015. http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.

[13] *NASA Goddard Space Flight Center - Flickr*, Accessed March 2015. https://www.flickr.com/photos/24662369@N07/.

[14] *OpenCV utilizing CUDA*, Accessed March 2015. http://opencv.org/platforms/cuda.html.

[15] *The Top 20 Valuable Facebook Statistics Updated February 2015*, February 2015. https://zephoria.com/social-media/top-15-valuable-facebook-statistics/.

[16] Amin Abbasi, Farshad Khunjush, and Reza Azimi. A preliminary study of incorporating gpus in the hadoop framework. In *Computer Architecture and Digital Systems (CADS), 2012 16th CSI International Symposium on*, pages 178–185. IEEE, 2012.

[17] Apace. *Apache Hadoop YARN*, March 2015. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[18] Apache. *Apache Hadoop*, March 2015. https://hadoop.apache.org/.

[19] Jon Brodkin. *Amazon puts 26,496 CPU cores together, builds a Top 100 supercomputer*, November 2013. http://arstechnica.com/information-technology/2013/11/amazon-built-one-of-the-worlds-fastest-supercomputers-in-its-cloud/.

[20] Brad Brown, Michael Chui, and James Manyika. Are you ready for the era of big data. *McKinsey Quarterly*, 4:24–35, 2011.

[21] Dbms2. *Petabyte-scale Hadoop clusters (dozens of them)*, March 2015. http://www.dbms2.com/2011/07/06/petabyte-hadoop-clusters/.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[23] Max Grossman, Mauricio Breternitz, and Vivek Sarkar. Hadoopcl: Mapreduce on distributed heterogeneous platforms through seamless integration of hadoop and opencl. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1918–1927. IEEE, 2013.

[24] Apace Hadoop. *Apache Hadoop JobTracker*, March 2015. http://wiki.apache.org/hadoop/JobTracker.

[25] Jesse Patrick Harvey. Gpu acceleration of object classification algorithms using nvidia cuda. 2009.

[26] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[27] Josh James. *Data Never Sleeps 2.0*, April 2014. http://www.domo.com/blog/2014/04/data-never-sleeps-2-0/.

[28] Svetlin A Manavski and Giorgio Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC bioinformatics*, 9(Suppl 2):S10, 2008.

[29] Mary Meeker. Internet trends 2014-code conference. *Retrieved May*, 28:2014, 2014.

[30] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[31] Mark Nixon. *Feature extraction & image processing*. Academic Press, 2008.

[32] Nvidia. *CUDA C Programming Guide Introduction*, March 2015. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz3VPZFHZ6u`.

[33] Nvidia. *Nvidia Cuda 1.0 Release*, March 2015. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz3VPZFHZ6u`.

[34] Alexis Santos. *Cray's Jaguar supercomputer upgraded with NVIDIA Tesla GPUs, renamed Titan*, October 2012. `http://www.engadget.com/2012/10/29/cray-titan-supercomputer-nvidia-tesla-gpu-k20/`.

[35] Open Stack. *OpenStack Open Source Cloud Computing Software*, March 2015. `https://www.openstack.org/`.

[36] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal*, 30(3):202–210, 2005.

[37] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Recent Advances in Intrusion Detection*, pages 116–134. Springer, 2008.

[38] Piotr Wendykier and James G Nagy. Large-scale image deblurring in java. In *Computational Science–ICCS 2008*, pages 721–730. Springer, 2008.

[39] Wikipedia. Multi-core processor — wikipedia, the free encyclopedia, 2015. [Online; accessed 22-April-2015].

[40] Jie Zhu, Hai Jiang, Juanjuan Li, Erikson Hardesty, Kuan-Ching Li, and Zhongwen Li. Embedding gpu computations in hadoop. *International Journal of Networked and Distributed Computing*, 2(4):211–220, 2014.