Summary of Master Thesis

This thesis explores the applicability of using GPUs in a machine learning and big data context. The motivation for this is the ever increasing problem of data created. In the last few years, the amount of data created has exploded, and a solution to handle this amount of data is needed. Another factor is the fact that GPUs can be very cost efficient compared to CPUs. Machine learning has already begun moving in the direction of utilizing big data frameworks, and some implementations of GPU frameworks for these has been implemented. A GPU implementation for the big data framework Flink has not yet been implemented, and based on this, and that Flink is developed for machine learning, Flink is the target big data framework for this project.

The thesis then explains the technical background needed to develop such a framework. This includes the theory behind big data frameworks, the MapReduce model, insight into two big data framework and a comparison between the CPU and GPU architecture. Another part of the theory is defining which class of machine learning problems is best fit to utilize the a GPU. After this, GPU frameworks are explored, and how these work explained in detail. A GPU framework is chosen for the project and initial tests done. These tests show that GPUs are indeed fit for a specific class of machine learning problems. Related work in the area of machine learning and big data with GPU support is then explored, and a recent implementation of a big data GPU framework, HeteroSpark, is explored in detail.

The thesis then provides an architectural overview of how a GPU framework would look like, and defines which part of the framework is provided by the framework itself, and which part that needs to be provided by the user of the framework. The architectural concept is then used to define how this would look like in a big data framework context, such that it can run on a cluster setup. If the user is to use the framework in a cluster context, it should automatically fit itself to the hardware on each of the workers in the cluster. How this is done and is explained and a model provided.

The two parts of the implementation is then explained in detail, first the User API and then the back end. What parts of the framework that the user interacts with and what is automatically done for the user is defined, and examples of this given. A brief introduction to the standard library of the framework is given. The implementation is then tested on a local setup and results given. The results show that there is a speed up if the data size is big enough, and that overhead has been introduced with the use of the framework. Unfortunately, due to time constraints, the framework has not been tested in a big data framework context. There is however given theoretical results of how tests on such a setup would be.

Finally the conclusion of the project is that there were two main contributions of the project, the first is a theoretical model of a GPU framework in a big data context, and the second being the framework implementation with user API and back end. The goal of the project was to implement and test a GPU machine learning framework in a big data context, and this has not been done. The theory behind it and a working machine learning GPU framework has however been developed and tested.

As a substitute for the full implementation of the framework in a big data context, the theoretical tests of such a setup has been given.

GPU Accelerated Machine Learning

Ulf Gaarde Simonsen, *Aalborg University - Selma Lagerlöfsvej 300 - 9000 Aalborg*

Abstract:
This article explores the possibility for scaling up machine learning problems using big data frameworks in conjunction with GPUs. In using GPUs to scale up these problems, an increase in speed and lower power consumption is seen. A short introduction to big data frameworks and CPU/GPU architecture is given, followed by a conceptual and technical model. A general purpose machine learning library for GPUs has been developed and will be explained in two parts, the User API and the back end. The implementation is tested and shows that there is a speed up to be gained by using GPUs in a machine learning context, unfortunately no tests on big data frameworks are performed. Finally the project is concluded and future work discussed.

I would like to thank my supervisor Thomas Dyhre Nielsen for his help and supervision during the development of this master thesis.

1. Introduction

As we are getting more and more data in the world, machine learning and big data are becoming more relevant than ever before. Currently the growth in data is so immense that new measures needs to be used to handle the increasing amount of data created daily. As of 2015, 90% of the data created in the world was between 2013-2015, and increasing in rate [24].

One of the ways to handle these huge amounts of data is machine learning[35], and with the increase in need for solutions to handle the growing amount of data, such techniques are interesting to look at.

Big data frameworks such as Hadoop, Spark and Flink have been used to support such scaling of big data by using multiple computers in a cluster to distribute the

workload[15][13][36]. Due to an increasing demand in the industry for alternatives to the huge frameworks such as Hadoop [2], and the need for more specific use of big data, new big data frameworks have been created. Spark is a new framework which is a general big data framework, marketed as a faster alternative to Hadoop. Flink[13] is another new big data framework, which utilizes a different model, and is more specialized in that it targets scaling up machine learning problems in big data.

When working on big data, scaling up problems to achieve a speed up is usually done by adding more hardware to solve the problem faster. Usually a cluster of commercial hardware, e.g. multiple computers with any form of CPU, is used to scale up performance on big data.

Another way to scale up is to use the GPUs(*Graphical Processing Units*) in these systems. This has been proposed as early as 2008, with the Mars framework for Hadoop[9]. More recently, newer GPU acceleration frameworks of big data algorithms has been explored and implemented. One such framework created for a big data is HeteroSpark[32], which enables GPU support for Spark. GPUs contain many cores, but each one less powerful than e.g. a CPU core. This makes GPUs suitable for simple data parallel tasks, and such could be great for scaling up some big data solutions.

As of writing this article Flink does not currently support the use of GPUs.

Apart from the potential of scaling up, GPUs are also more efficient when it comes to power consumption. The experiments of HeteroSpark shows this with great effect. Their results are as seen in Figure 8. The results only compare performance, but we can use the hardware used in the test as a benchmark for power consumption. The CPU used in their experiments, aa Intel Xeon Processor E5-2670, consumes around 115W while under full load [18], where as the GPU, a new NVidia Quadro graphics card, only consumes around 250W[30]. If we use the example as baseline for how much we can expect GPUs to perform on a big data problem, this reduces the consumption from 1840W(128 CPU cores) to 615W(8 CPU cores, 2 GPUs), where each CPU has 8 cores. This is only considering the power consumption of the CPUs and GPUs themselves, and not additional hardware that may be needed.

With this in mind, this article focuses on the applied use of GPU acceleration in the case of machine learning for scaling up problems.

The focus of this article will be to explore the possible solutions and create a framework that will enable scaling of existing machine learning problems using big data. Such a framework should be able to be used in conjunction with a big data framework such as Flink.

This article will first explain the MapReduce programming model and Flink framework followed by a comparison between the CPU and GPU. Related work will then be discussed, and based on this, an architectural model of a big data GPU framework is proposed. Implementation specific details will then be explored in depth, and experiments of the model will be conducted. Lastly the work will be concluded and future work explored.

The contribution of this project is a conceptual model of how a big data framework with GPU support would look like, and a general purpose machine learning GPU library.

2. Technical Background

In this section, existing big data solutions and GPU based scalable machine learning will be explained. In Section 2.1, the theory needed from Big Data frameworks will be explained. In Section 2.3 a comparison of the CPU and GPU will be made. In Section 2.3.1, how to scale algorithms and what class of machine learning algorithms that fit in a GPU context will be explained. In Section 2.3.2, existing GPU frameworks are explored. Lastly in Section 2.4, initial tests comparing CPU and GPU performance will be presented.

2.1. Big Data Frameworks

To understand the fundamentals of Big Data frameworks, the MapReduce programming model must be explained.

2.1.1. MapReduce

For big datasets, a proposed model from Google is MapReduce[7]. MapReduce is a programming model which is a combination of three steps, the map step, the shuffle step and the reduce step. This model is very useful when working with multiple workers, such as multiple CPUs or even just multiple cores. Each worker has a chunk of data, and performs computations only on this chunk. The user needs to provide a map and a reduce function. Key-value pairs are used to know what the map function will map to.

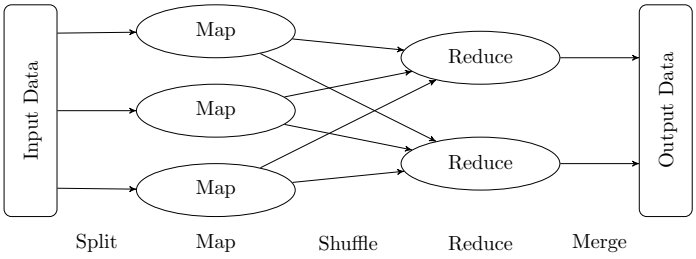

Figure 1: *The MapReduce programming model [23]*

Map reduce will do the three steps as follows:

- Map Step: The data is split up into even chunks and the map function is applied to each chunk of data, mapping the data to a set of key-value pairs.
- Shuffle Step: The data is shuffled between the workers such that only data mapped to the same key is on one given worker.
- Reduce Step: The reduce function is applied on the mapped data and an aggregated result is achieved.

This model allows for great scalability, as two simple functions will allow any amount of data to be split up and then combined again.

An simple example of how this would work can be explained using a non-standard deck of cards. Let us assume we have a deck of cards where there are no aces or face cards, such that we are left with only numbered cards, but it still has all four suits(spades, clubs, diamonds and hearts). The deck may contain multiple numbered cards of the same suit, e.g. three times the seven of clubs is allowed. Any number of cards in the deck is allowed, as long as they follow all the rules. The goal is to figure out the total value of the numbered cards in each suit.

- Key-value pair: The key is the suit of a given card and the value is the number on said card.

Let us assume we have two workers, each having some of the non-standard deck of cards divided between them. With this setup the use of the MapReduce model would go through the following steps:

- Map Step: Each worker sorts the deck of cards into piles, each pile consisting only of cards with the same key. In this example, four piles, each one with a different suit.
- Shuffle Step: The data is now shuffled around, such that the first worker receives the piles with the hearts and diamonds, and the second worker receives the piles with spades and clubs.
- Reduce Step: The data is reduced to the aggregated results we wanted to achieve, which is the total value on each of the piles with a given key. In this case it is the total value on each card with a given suit.

2.1.2. Apache Spark

Apache Spark is a new big data framework made for big data analytics. The framework requires a cluster manager, e.g. Hadoop Yarn [34]. Spark also requires a distributed file system such as Hadoop Distributed File System(*HDFS*). Spark has become popular because of its speed up compared to Hadoop, as Spark is up to 100 times faster than Hadoop MapReduce in processing data in memory, and up to 10 times faster on the disk. This is the case for algorithms such as logistic regression [36].

The reason Spark is faster is due to its batch processing framework. It uses small parts of the data for each iterations instead of the whole data at once. This is called *micro batching*. This however comes at at cost of latency, as Spark needs to schedule a new set of tasks after each iteration on the data. Data used in Spark is stored in Sparks own form of dataset, called Resilient Distributed Dataset(*RDD*). RDDs are resilient due to the way they are created. Each time an operation is done on a RDD a new RDD is created, and this improves safety of the system, as Spark can easily convert back to a prior version of an RDD if there is an error in the task or hardware failure [6].

RDDs come with a few weaknesses. As a new RDD has to be created each time an operation on an RDD is performed, some functions will not perform well on Spark. An example of this is sorting. As a new RDD has to be created before a new task can be

made, the dataset has to be read entirely before it can be partitioning can be done. As seen in Figure 2, this makes it so the execution of the terasort is very linear.

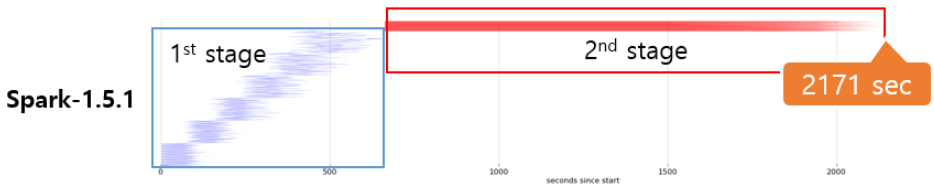

Figure 2: *Overview of the terasorting process with Spark[27] [21]*

2.2. Apache Flink

Apache Flink is another new big data framework from Apache, and is lesser known and used than Apache Spark. Flinks main application is in big data analytics and machine learning. Flink introduces a new model by using pure streaming in addition to working in batches. Flink only schedules tasks once, and this causes Flink to not have the latency problems that Spark has. Flink has some additional functionality in that it allows for each iteration of an iterative process to be run on a single machine instead of the entire cluster if this functionality is wished for in an application[33].

Comparing Spark and Flink we can take a look at the sorting example from Section 2.1.2. Since flink does not utilize RDDs, it does not have the same problem with the limit of scheduling a new task on the data before a task is complete. This means that flink can partition the data at the same time as it is read in and makes it so the task runs more concurrently than linear as seen in Figure 3.

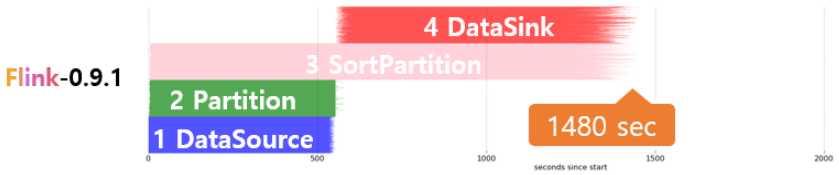

Figure 3: *Overview of the TeraSorting process with Flink[27] [21]*

Based on Flink being a big data framework that already has the focus on machine learning, and it supports both streaming and batch programming, it has been chosen as the big data framework for this project.

2.3. GPU/CPU Comparison

To understand why one would use a GPU instead of a CPU for machine learning problems, we have to take a look at the physical architecture of the CPU and GPU. Figure 2 is an overview of how much space is utilized by the different hardware components in a CPU

and GPU. In a CPU, a lot of space is used on the cache, and an equal amount is used on control and ALUs (*Arithmetic Computational Units*). This means that a CPU has a lot of hardware to control and compute special tasks, as well as a big cache to utilize. GPUs on the other hand uses almost all of their physical space on ALUs, and very little on control and cache. This means that a GPU does not have the ability to perform many hardware specific tasks and use the cache to quickly switch out data. A GPU however, does have the ability to make simple computations on many ALUs in parallel. A simple comparison of the physical usage in hardware is as seen in Figure 4.

When programming a CPU, you can explicitly manage and schedule threads and usually run a few heavyweight threads to handle a given process. Usually each of these threads are individually programmed. When programming a GPU you use single instruction multiple data[17], *SIMD* for short, which performs the same operation on multiple points of data in parallel. This allows many threads to be programmed for the same task on different data points. Threads are more light weight and are programmed in batches rather than individually. A set of SIMD instructions and what data it is to be used on is called a *kernel*[28].

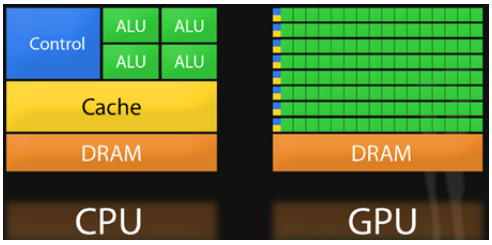

Figure 4: *CPU & GPU architecture simplified[8]*

2.3.1. Scalable Algorithms

The GPU architecture is not suited for all kinds of machine learning problems, but it is great at a specific class of problems. Sutter and Larus [37] points out that systems with multiple cores benefit from applications that run concurrently. This means that there should be little to no communication between the cores. Chang-Tao Chu et. al.[10] has looked at how this is best achieved and come across Kearns' Statistical Query Model[1].

Many algorithms that calculate sufficient statistics gradients fit this model, because their calculations can be batched up into sub calculations. This means that we can easily distribute the calculations over multiple cores and do partial computations where the results in the end are aggregated. Chan-Tao Chu et. al. calls this for of algorithms the *"summation form"* One way to show how the "summation form" can be applied in practice is to make a simple modification to the well-known gradient descent algorithm, as seen in Appendix A.1.

$$w := w - \alpha \nabla Q(w) = w - \alpha \sum_{j=1}^{c} \sum_{i=1}^{n_j} \nabla Q_i(w) \tag{1}$$

This extension splits the sum of entries into smaller sums. Using partial sums rather than a sum over the whole dataset, the data is split up into c different sub datasets, and the results is the sum over all c sub datasets. This algorithm can now scale with the number of cores on a given piece of hardware, e.g. a GPU with 1000 cores will do the computation on 1000 smaller sub datasets rather than on one big dataset.

This can be applied to machine learning models such as logistic regression, as shown by Chan-Tao Chu et al[10]. The definition of logistic regression can be seen in Appendix A.2. To fit the parameters of logistic regression, it must be combined with a model such as gradient descent. If using gradient descent, logistic regression would fit the summation form and be able to be run concurrently. This would be done by calculating the update of the parameters concurrently and summing them afterwards.

Clustering algorithms can also benefit be scaled up even though it is not obvious. An example of a clustering algorithm could be K-means (Appendix A.3). K-means require a lot of communication each iteration, as centroids are continuously calculated based on the new members of its cluster. This requires an update of what centroid each data point is a part of each iteration. As this requires communication between each core, it is not obvious the it fits the summation form. However, it is clear that the computation of e.g. euclidean distance can be sampled into sub computations and done concurrently, and when done the results aggregated such that one can compute the position of the centroids[1].

2.3.2. Existing GPU Frameworks

As for GPU programming frameworks, two major ones exist, Nvidia's CUDA[29] and Khronos Group's OpenCL[14]. They both work in a similar fashion, and uses a high level API such as Java or C# to write an application using the framework, and a low level language such as CUDA c or OpenCL can be used to write the program for the GPU, called a *kernel*. To use these frameworks, the user must handle all the interaction between the CPU and the GPU. This includes compilation of the kernel, allocation of memory on the GPU and copy of data to and from the GPU from the CPU.

Both frameworks work when used solely for GPU computation, but OpenCL has the upside that it also runs on CPUs. This might be relevant if one is to use a mix of CPU and GPU workers, since the same openCL implementation can be used for both CPU and GPU. Due to the similarities between CUDA and OpenCL, only OpenCL will be used as an example for this section.

If used for the GPU, the user needs to implement a few steps. This includes *serializing the data*, which is the process of transforming the data into an easy to work with format for the GPU. This is usually done by transforming ones data into a simple data structure such as a single or multidimensional array. Furthermore, the user has to specify and allocate memory for the data that needs to be copied to the GPU, as well as allocate and specify where the answers of the computation is to be put. Lastly, which user written GPU program, the *kernel*, that is to be run on the GPU must be specified, and arguments set.

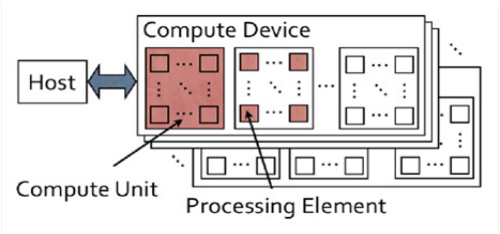

Figure 5: *Host of process in OpenCL[12]*

To get a better understanding of how OpenCL handles a kernel, we need to look at how OpenCL handles a kernel on an architectural basis. As seen in Figure 5, the host CPU communicates and sends out work to the different compute devices. A compute device is in this case either may consist of a CPU core or a collection of GPU cores.

As seen in Figure 6, each compute unit is a part of a block of compute unites which share some local memory. This needs to be taken into account when programming the kernel, as each worker might or might not have the ability to share local variables and data, depending on if they are in the same block or not. The only way for multiple workers to access the same information is either through the global memory. It is good practice to make sure that only compute units in the same block share information.

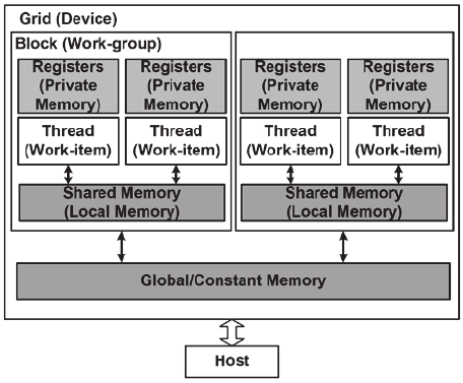

Figure 6: *Architecture of OpenCL[12]*

As of now CUDA only supports Nvidia graphics cards, while OpenCL supports both Nvidia and AMD graphics cards, as well as CPUs. Based on this, OpenCL will be used for this project.

2.4. Initial Test

A simple way of checking if the GPU is better at handling parallel problems than the CPU is to conduct a basic test. A test was set up to run a logistic regression model, using the extended gradient descent from Equation 1, on the criteo [4] dataset with 100.000 samples over 100 iterations. The test was conducted on an i7-2600k CPU, only using 1-4 cores, and on a Nvidia GTX 770 running 8 kernels. The average of the test results can be seen in Figure 7. It is clear that the GPU performs the task much faster, even when the CPU runs all of its 4 hardware threads, the GPU is almost 6 times faster.

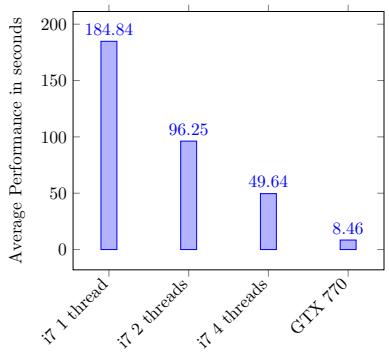

Figure 7: *Logistic Regression CPU/GPU, 100.000 samples, 100 iterations*

3. Related Work

In this section related work to the article will be explored, in Section 3.1 the general work of using GPUs in machine learning will be discussed. In Section 3.2, an implementation of a big data GPU framework, HeteroSpark, is explored.

3.1. GPU and Big Data Machine Learning

One of the main uses of GPUs at the time of writing this article is in deep machine learning [19]. Deep learning focuses on extracting features from data using computational models instead of using a human to define them by hand. A way to do this is to use Convolutional Neural Networks(*CNNs*)[11], which use several layers to learn features. CNNs are typically used in image recognition where they can learn features from images e.g. if you feed some images of faces to a CNN it will find recognizable features such as eyes, nose and a mouth. The reason GPUs are used for deep learning is that it is very computationally expensive to do this, but the problem of e.g. image recognition is very easy to split up and divide into many similar sub-problems. The architecture of GPUs fit perfectly, because it utilizes the SIMD format as explained in Section 2.3.

In the area of big data GPUs have been used since 2008 where a framework called MARS was developed[9]. The focus of MARS is to combine the MapReduce model with GPUs. It does this by generating the required code for the GPU to run both the Map and Reduce steps on the GPU itself. For its time it showed a great speed up compared to cluster setups such as Hadoop [16], but MARS is less flexible when it comes to the dataset size selection. The gap in performance between Hadoop and Mars becomes smaller as the size of the data increases as shown by prior research [16].

3.2. HeteroSpark

In recent times a newer implementation of a GPU big data framework has been made, in the form of HeteroSpark[32]. A look at their test results, as seen in Figure 8 reveals that a single computer with 8 CPU cores and 2 GPUs performs on par with a cluster

of 16 computers with 8 CPU cores, when working on machine learning problems such as logistic regression, K-Means and Word2Vec. As seen in Figure 8, GPUs can solve problems much faster than CPUs.

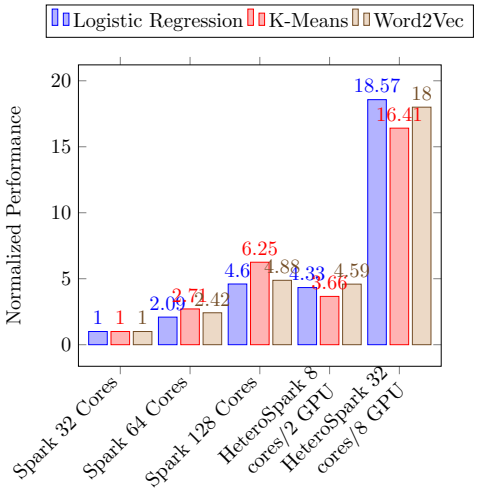

Figure 8: *Performance of HeteroSpark v.s. Spark[32]*

As seen in Figure 9, HeteroSparks worker consists of two parts, the CPU worker and the GPU worker. These two workers are connected through a Java Remote Method Invocation(*RMI*)[31] layer that serializes and de-serializes the data. The CPU is treated as a RMI client, while the GPU is treated as a RMI server. This is done such that the GPU can use existing libraries through its dynamically linked library (.so), and is linked through the Java Native Interface(*JNI*). This ensures easy access to existing libraries at the cost of a more complex model, and more overhead with the client-server communication.

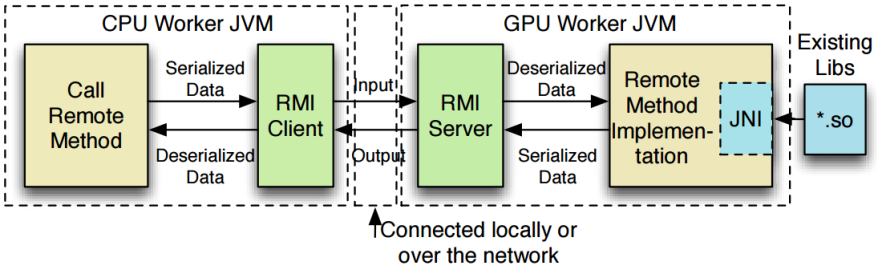

Figure 9: *HeteroSpark CPU-GPU Communication[32]*

Based on this RMI client-server model, HeteroSpark proposes a model on how to use GPUs in conjunction with Spark. The proposed model is as seen in Figure 10.

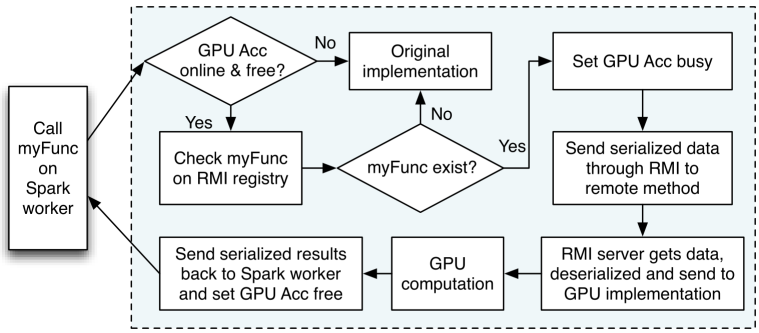

Figure 10: *Glue logic of HeteroSpark [32]*

When calling a function that needs to use the GPU on a given Spark worker, the worker then checks if it actually has a GPU and it is currently not in use. It then checks if the function the users tries to invoke exists in the RMI register, a simple check to make sure the GPU can use the function. If this function exists in the RMI registry, the GPU is set as busy and data is serialized through the RMI layer. Once the data is on the GPU, the RMI server component deserializes it and runs the GPU implementation of the function the user wants to execute. The results are then serialized and sent back to the Spark worker, as if the problem had been solved on a regular Spark worker

4. Framework: Architectural Overview

The idea is to create a general framework to speed up machine learning algorithms by using the GPU. Such a model needs to be simple for the user to apply, yet flexible enough to allow custom solutions. An architectural model of such as framework can be see in Figure 11.

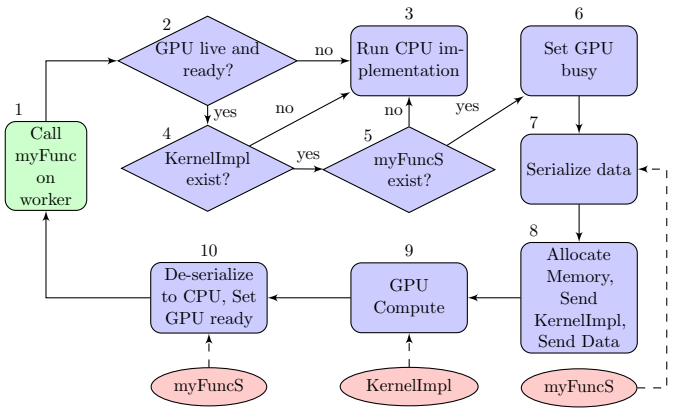

Figure 11: *GPU Framework*

To understand the model we need to define two functions first.

- Serialize Function: The serialize function(*myFuncS*) is a user defined function that takes the user data as input and outputs a single dimensional array that is needed for it to be used on the GPU. The user should define the transformation from the user data to the single dimensional array. The size of this array is then one of the determining factors for the allocation of the memory on the GPU.
- Kernel Implementation: The kernel implementation(*KernelImpl*) is a user defined kernel. This is the program for the GPU. This kernel handles all the computation on the GPU, and it has the serialized data as input and will output data in a standard or user defined format.

The framework start in step one by being called by the user in step 1. The user needs to import and declare that he or she intends to use the framework. When the user has declared the intent to use the framework, the framework goes to step 2 and checks whether or not there is a GPU, and if there is, if it is busy. This either results in the CPU or GPU implementation being used. If KernelImpl and myFuncS exist(step 4 and 5), there is a GPU present(step 2), and the GPU is ready(step 2), the framework will continue with the GPU implementation. If not, the framework will default to the CPU implementation(step 3). The next step is to set the GPU as busy(step 6), such that further calls to the same GPU will not result in the same GPU being used for multiple kernels. Once the GPU has been set as busy, the data will be serialized according the the user defined function myFuncS(step 7). When the data has been serialized, the memory on the GPU will be allocated(step 8). This is done based on the data given by the users myFuncS. The framework also needs some information about the underlying hardware to allocate the memory, e.g. how much memory there is on the target GPU. Once the memory has been allocated, the user defined KernelImpl and serialized data will be sent to the GPU(step 8). The KernelImpl will then run on the GPU and execute the user defined kernel(step 9). Once the kernel is finished, the kernel will return the data to the CPU and the data will be de-serialized based on the users myFuncS(step 10). Finally, the GPU is set as ready again (step 10).

On the frontend, the framework will act as an easy to use API for the user. The API enabled the user to easily implement a GPU implementation if the user thinks their problem would benefit form the use of a GPU. On the backend the framework will automate some functions for the user. Memory management is a big part of implementing a GPU into a system, and with this process being automated gives the user the ability to implement a GPU solution without the knowledge of how to handle or instantiate memory on a GPU. The framework also omits all the necessary information gathering that needs to be done to be able to use a GPU on a given system.

5. Proposed Framework

This section explores how a combination of Flink and OpenCL would look like, and explain the goals behind such a combination.

The main goal of the framework is to make the GPU easily available to the user, and such should be easy to use in a MapReduce context such as Flink. The user of the GPU in a MapReduce context could be beneficial in several steps of the process. In this project, the focus will be to enable the speed up of machine learning, and such will not focus on e.g. speed up of map and reduce functions by using the GPU.

When using a cluster setup such as the one seen in Figure 12, the user should easily be able to implement the same implementation of the GPU across all the different hardware there might be. The user will define the two functions myFuncS and KernelImpl on the master node and this will be sent out to each worker. The serialize function will then be run on the data on each individual worker and used as a basis for the *memory allocator* on that worker. The job of the memory allocator is to gather all the information needed to allocate memory on the GPU on that specific worker. This includes gathering some information about the hardware;

- The amount of main memory on the GPU on the given worker.
- The amount of compute units on the given worker.
- The size of a data instance of the users data.
- How many instances of data the user has given.

This is then used to calculate many data instances each compute unit on the given worker can handle and if needed, how many iterations the framework needs to make to handle all the data on the given worker.

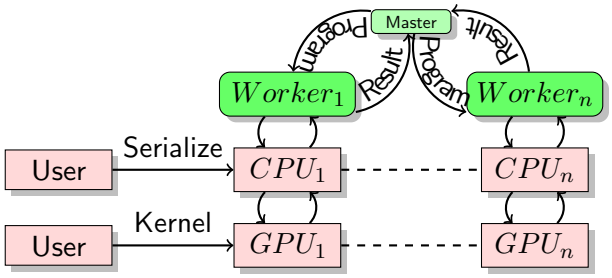

Figure 12: *Architecture overview*

- Master: The master sends the program used on each of the workers and is responsible for gathering all the results send back from each worker.
- User: The user contributes with two functions, the serialize function and the kernel function.
- Worker: The worker has a chunk of the data which needs to be worked on, a CPU and sometimes a GPU. It is responsible for getting the intermediate results, and sending it back to the master if needed.

- CPU: The CPU on a worker executes the program sent by the master. It checks if there is a GPU present and if, then generates the memory management based on the data and serialize function given by the user.
- GPU: The GPU runs the kernel function given by the user and returns the results to the CPU.

6. Framework: Implementation

This section explains the implementation and technical design decisions made during implementation. In Section 6.1, existing frameworks needed for the implementation will be explored. In Section 6.2, an overview of the user API part of the implementation is presented and design decisions explained. In Section 6.3, an overview of the back end of the framework is given, and the most essential functions presented. Finally in Section 6.4, a standard library for the framework will be presented.

6.1. Existing Frameworks

Since Flink only supports Java, Scala and Python, a solution to combine Flink and OpenCL must be found or created. There exists a handful of solutions for using OpenCL in a higher level language such as Java, mainly LWJGL [22] and jocl[20]. Using one of these libraries, one can combine Flink and OpenCL without the need for implementation of OpenCL in Flink. Since the only function of such a library in this project is binding OpenCL to Java, LWJGL has been chosen as it is the framework with the most extensive documentation.

6.2. Framework: User API

The implementation of the framework is done through using the Java implementation of Flink as well as the LWJGL library. It has to be noted that even though LWJGL is used for the OpenCL bindings to Java, each kernel must still be written, or generated, in native OpenCL.

The implementation is based on Figure 11, and implements each step either in Flink, LWJGL or OpenCL.

As the framework is used as an extension to a standard Flink program, the user will simply start the framework by importing it into an existing Flink project and specifying which part of the code is needed to run on the GPU. This is step 1 on Figure 11. As seen

```
OpenCLObject object = new OpenCLObject(dataArray,
        instances, weightSize);
object.runLogistic("cl/logistic.txt", "logistic",iterations);
```

Figure 13: *Creating an OpenCLObject*

in the code at Figure 13, to use the framework the user must create an OpenCLObject which takes three arguments. The first is the dataArray which is an array of doubles or floats containing the data. instances is the amount of floats or doubles a the user data consists of. Finally, weightSize is the amount of floats or doubles the users weights contains, given that the user wants to use an algorithm that requires this e.g. logistic regression. When a user defines a new OpenCLObject, the constructor fetches info about possible OpenCL compatible hardware in the system, this is step 2 and 3 on Figure 11. If the system has a GPU, it will use this as the target for the program, otherwise it will default to use the CPU. After the target hardware has been chosen, the constructor will call the memory allocator, which is one of the most essential parts of the framework. The memory allocator will be explained in detail in Section 6.3.

When the user has defined an OpenCLObject, they can then use the run method on it as seen in Figure 13. In this example the standard implementation of logistic regression is run. The run method takes three arguments. The first is a string with the name of the path to the kernel which is to be compiled. The second is a string with the name of the function that is to be called in the kernel on the GPU. Lastly, the third argument is the number of iterations the user wishes to run the kernel on the GPU.

6.3. Framework: Back end

Once the user has called the run method, nothing more needs to be done by the user. The back end of the framework will automatically calculate and allocate the memory needed and launch the kernel on the GPU. As said previously, the memory allocator is one of the most essential part of the framework, as it allows the framework to easily allocate memory on the GPU, regardless of what kind of hardware is used. The code for the memory allocator is as seen in Appendix B. LWJGL has built in functionality to check for the amount of memory and compute units the target hardware has. This information is used to compute how much memory there will be left for data, and how many data instances that fit with each compute unit. If there is more data than the target hardware can handle in a single iteration, the memory allocator will calculate how many iterations are needed to go through all the data. This is not to be confused with the user defined iterations, as the user defined iterations is the number of times the user expects the whole dataset to be used on the GPU.

When the run method is called, the framework tries to compile the kernel and launch it. This is done in a few steps. First, even before the kernel is compiled, any dynamic variables, such as the size of a data instance, needs to be pre-processed in the kernel. OpenCL does not allow for dynamically allocated arrays, so a way to work around this is to load the kernel into the framework as a string, and change any dynamic allocated arrays to their fixed size. The kernel is then compiled.

Allocating the memory of the GPU is the first part of step 8 in Figure 11. An example of this is as seen in figure 14. If you need to copy any data, LWJGL requires it to be of the type Buffer. In this case, the data, weight and answer arrays are converted to buffers. The type of memory wanted must then be specified. In this example, the data and weight memory are used only for computations and are not to be changed, so they

```
dataBuffer = UtilCL.toDoubleBuffer(dataArray);
weightBuffer = UtilCL.toDoubleBuffer(weightsArray);
answerBuffer = UtilCL.toDoubleBuffer(answerArray);

dataMemory = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, dataBuffer, null);
weightMemory = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, weightBuffer, null);
answerMemory = clCreateBuffer(context, CL_MEM_WRITE_ONLY |
CL_MEM_COPY_HOST_PTR, answerBuffer, null);
clEnqueueWriteBuffer(queue, dataMemory, 1, 0, dataBuffer, null, null);
clEnqueueWriteBuffer(queue, weightMemory, 1, 0, weightBuffer, null, null);
clFinish(queue);
```

Figure 14: *Memory allocation of a kernel using logistic regression*

are copied as read only. The answer memory is where we want the kernel to store the returned values, and such needs only to be write only. Once specified, the buffers are enqueued into the command queue and await execution. Enqueuing the buffers states that we want to copy the data contained in the buffer to the specified place. This is the reason the answerMemory is not enqueued yet, as we want to copy the buffer back to the CPU memory once the kernel has finished. after enqueuing the write buffers the copy is executed by calling the clFinish function on the command queue.

```
kernel1DGlobalWorkSize.put(0, kernelAmount);
kernel.setArg(0,dataMemory);
kernel.setArg(1,weightMemory);
kernel.setArg(2,answerMemory);
kernel.setArg(3,instancesPrKernel);
clEnqueueNDRangeKernel(queue, kernel, 1, null, kernel1DGlobalWorkSize,
 null, null, null);
clEnqueueReadBuffer(queue,answerMemory,1,0,answerBuffer,null,null);
clFinish(queue);
```

Figure 15: *Settings the arguments and launching kernels*

Now that the memory has been specified, the arguments of the kernel must be set and a kernel for each compute unit must be launched. This is the last two steps of step 8 in Figure 11. As seen in Figure 15, the arguments for the kernel are set, in this case the implementation of logistic regression. Once the arguments are set, the an amount of kernels are launched based on the data from the memory allocator. This is done by using the function clEnqueueNDRangeKernel. To use this we need to specify which queue and

kernel are used as well as how many kernels we want to launch. Other arguments can be set for special cases such as wanting the kernel to be finished before a specific event occurs.

After the arguments for the kernel has been set and it has been specified how many kernels are to be launched, the answerMemory is enqueued to copy the data back into the answerArray. Finally all the commands enqueued to the command queue are executed by calling clFinish on the queue.

Step 9 in Figure 11 is done on the target OpenCL hardware. Once the kernel is finished, the GPU is automatically set as ready and the framework can once again return to step 1.

Some steps of Figure 11 were skipped during the implementation. These are step 5, 7 and part of 10. As to use the framework, the user has to give the data as a single dimension array, the need for the serialize function to be inside of the framework has been omitted. This removes the need for a check of the existence of the function as well as the serialize and de-serialize function. It still remains as a core part of the conceptual model, as it is still needed for the framework to function.

6.4. Standard Library

The framework includes a standard library of pre-written OpenCL kernels. This includes a variety of widely used machine learning techniques, and can be used without the need of writing ones own kernel, these require no modification to be used. The standard library includes standard solutions to model training and clustering with the methods of logistic regression and K-Means currently.

7. Experiments

In this section, the framework will be tested and results given. In Section 7.1, the experimental setup will be given. In Section 7.2, the results of the experiments will be presented and evaluated. Finally in Section 7.3, an estimate of how the framework would work on a full Flink setup will be given.

7.1. Experimental Setup

The framework is evaluated with the two built-in machine learning functions logistic regression and K-means. The impact of the GPU is tested by benchmarking the performance of the framework implementation to a similar CPU implementation. Two datasets have been chosen for testing. The standard library implementation of Logistic regression is tested using the Criteo dataset[4]. The standard library implementation of K-Means is tested using the wine data set [3].

The framework is tested locally on two machines. The first machine has one CPU; i7-2600k 3.4 GHz with 8 gb of main memory and one GPU; Nvidia Geforce GTX 770 with 4gb memory. The second machine has one CPU; i5-6600k and one GPU; Nvidia Geforce GTX 1060 with 3gb memory.

Due to time constraints, a full implementation of the framework in Flink has not been made. Instead the framework will be tested to show how scalable and versatile it is by presenting results from different types of hardware.

7.2. Experimental Results

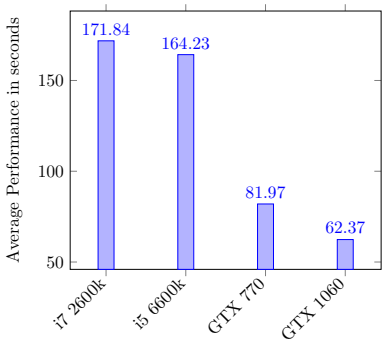

Figure 16: *Logistic Regression CPU/GPU, 1.000.000 samples, 100 iterations*

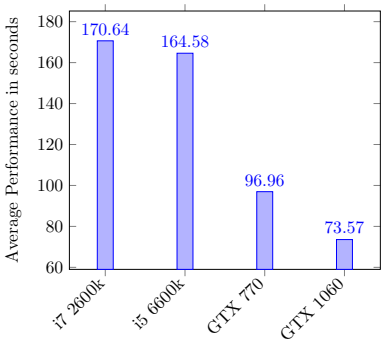

Figure 17: *Logistic Regression CPU/GPU, 10.000 samples, 10.000 iterations*

The framework has been tested with the two build in functions in the standard library. The results shown on Figure 16 and Figure 17, shows that using a GPU for solving problems such as logistic regression can be beneficial. The speedup is however not nearly as much as the initial tests showed, and this might be due to a few factors. With the introduction of the automated memory management, the framework has an increased overhead when using a GPU implementation. The framework first needs to calculate and allocate memory on the GPU. Another major factor is the need to copy the data to and from the GPU. To test this, another test with the same two setups has been run. Instead of 1 million samples run over 100 iterations, only 10.0000 were run, but over 10.000 iterations. This means that the same amount of data is processed, but due to the increase in iterations, the framework has to do a lot more of copying back and forth from the CPU to the GPU. This has a very noticeable increase on the performance of the GPU of about 18%.

The framework was also tested on the second standard library implementation. A kernel with K-Means was run with 50 centroids over 400 iterations and shows some interesting results. Figure 18 shows the results of the test with K-means, and in this case, the automated GPU implementation is slower than the CPU implementation. This might be caused by the fact that there are only 4968 samples of data, so the overhead of the data transfer causes the framework to be slower than a CPU implementation. This shows that it is not always right to use the framework, not even for problems that are easily run concurrently.

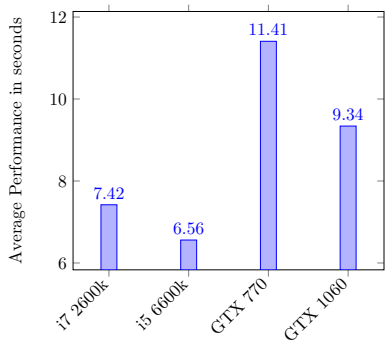

Figure 18: *K-Means CPU/GPU, 4968 samples, 400 iterations, 50 centroids*
A single test of the

7.3. Flink Test Approximation

As mentioned in the start of this section, due to time constraints, the framework was not tested on a fully integrated Flink cluster. An approximation on how a possible test on a such a cluster would have been can be made based on how tests on similar frameworks have been done. When Flink is integrated into to the framework, a few new causes of overhead is introduced. The first is the Flink framework itself. The second is the underlying file system used for the cluster. A file system such as Hadoop Distributed File System can be used in this case.

As for overhead, the Flink framework itself should only cause a minor impact on computational time. The underlying file system and communication between nodes on the cluster on the other hand causes some noticeable overhead. The question is how much this affects performance as we introduce more workers to the cluster.

If we look at the results from HeteroSpark, as seen in Figure 19, we can see that performance per worker slows down when new workers are introduced. If the two GPU results are used as an example, we go from a relative average performance of $2.096/GPU$ to $2.201/GPU$. If each new GPU is treated as a new worker, the spark loses about 0.83% performance for each new worker introduces. This is even more apparent with the CPU workers, where an increase from 4 to 16 workers introduces a loss of performance of 31.1% or 2.591% for each new CPU worker introduced.

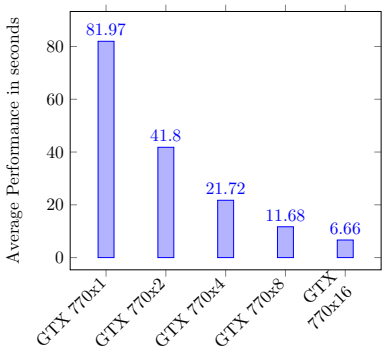

Figure 20: *Logistic Regression on Flink CPU/GPU, 1.000.000 samples, 100 iterations*

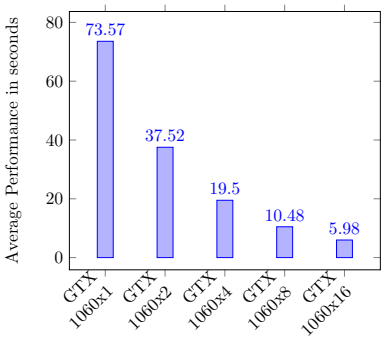

Figure 21: *Logistic Regression on Flink CPU/GPU, 1.000.000 samples, 100 iterations*

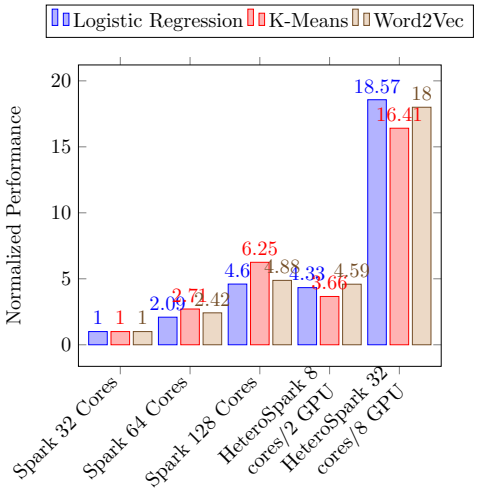

Figure 19: *Performance of HeteroSpark v.s. Spark[32]*

With this in mind, an expected overhead of 2% for each new worker seems reasonable, this gives us these new theoretical results. These are as seen in Figure 20 and Figure 21. This is however not a completely correct representation, as the increase in workers also increases the effect of the copy overhead mentioned before. With the introduction of more GPUs the overhead of copying from each CPU to each GPU increases and performance is lost.

8. Conclusion

In this project, the goal was to explore and propose a model to easily implement the support for GPUs in a big data framework context. The first contribution is the theoretical

model proposed in the article. In Section 4, this article has presented an architectural overview of how such a model could look like. The model clearly shows how a user can implement the usage of a GPU, with only having a few simple functions to define. In Section 5, a definition of how such a model would be incorporated into a big data context has been shown by proposing a model for the combination of the architectural model and the big data framework Flink.

The second contribution is the implemented framework. This framework consists of two parts, the user API and the back end. The user API fulfills the goal of an easy to use implementation of the framework. It enables a user to use a GPU in a machine learning context with minor effort. The back end automates a lot of work that otherwise would have been for the user to implement. This work includes the automation of memory allocation on the GPU and a standard library of machine learning techniques the user can use. The implemented framework can be used as an out of the box standard implementation if the user wishes by utilizing the standard library, or as a supporting framework for GPU implementation if the user wishes to use custom functions.

The main goal for the project was to implement a GPU framework to support machine learning on big data frameworks. This goal has not been met, as the framework has not been integrated into a big data framework. The project does however explain in theory how this could be done, and shows that there is a potential gain from implementing GPUs in a machine learning context. Test of a theoretical integration of the framework in a big data framework by the results presented in Section 7.3, and act as a substitute to a full implementation. The tests conducted in Section 7 does however show that the implemented framework works on different sets of hardware, and that using better hardware causes a speed up.

8.1. Future Work

As for future work, an integration of the framework into a big data framework such as Flink will be the first step. Integrating the framework into Flink opens up for the possibility to test the framework on more than one machine. By testing the machine in e.g. a cluster context, it might be easier to see how much of a benefit a single GPU gives in contrast to a CPU.

Another direction of future work would be to expand the look into the class of machine learning problems beyond the one used in this project. Other classes of machine learning problems might benefit from being run in a GPU context, and thus benefit from using the framework from this project.

References

[1] M. K. E. . Noise-tolerant learning from statistical queries. pages 392401, 1999.

[2] M. Asay. Hadoop demand falls as other big data tech rises. http://www.infoworld.com/article/2922720/big-data/hadoop-demand-falls-as-other-big-data-tech-rises.html, May 2015.

[3] P. Cortez. Wine quality data set. http://archive.ics.uci.edu/ml/datasets/Wine+Quality, 2009.

[4] CRITEO. Criteo dataset. https://www.kaggle.com/c/criteo-display-ad-challenge/data, June 2014.

[5] A. K. M. David L. Poole. Artificial intelligence - foundations of computational agents. Cambridge University Press, 2010.

[6] A. Dawar. Apache spark vs. mapreduce.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.

[8] D. Education. Massively parallel programming with gpus. https://people.duke.edu/ ccc14/sta-663/CUDAPython.html, Sept. 2016.

[9] B. H. et. al. Mars: A mapreduce framework on graphics processors. 2008.

[10] C.-T. C. et. al. Map-reduce for machine learning on multicore, 2007.

[11] T. L. et. al. Implementation of training convolutional neural networks. https://arxiv.org/ftp/arxiv/papers/1506/1506.01195.pdf, 2015.

[12] M. Fahmed. Opencl. https://mohamedfahmed.wordpress.com/2010/05/20/opencl-2/, Sept. 2016.

[13] Flink. Apache flink. https://flink.apache.org/, Mar. 2016.

[14] K. Group. Opencl. https://www.khronos.org/opencl/, Aug. 2016.

[15] Hadoop. Apache hadoop. http://hadoop.apache.org/, Mar. 2016.

[16] S. W. Heshan Li. Benchmark hadoop and mars: Mapreduce on cluster versus on gpu. http://www.cs.jhu.edu/ heshanli/liheshan/Projects_files/pp_final_project.pdf, 2009.

[17] S. C. E. Inc. Basics of simd programming. http://ftp.cvut.cz/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html, 2008.

[18] Intel. Intel xeon processor e5-2670 (20m cache, 2.60 ghz, 8.00 gt/s intel qpi). http://ark.intel.com/products/64595/IntelXeonProcessorE5-267020MCache-2_60GHz-8_00GTsIntelQPI.

[19] D. C. R. Itamar Arel and T. P. Karnowski. Deep machine learninga new frontier in artificial intelligence research. Research Frontier, 2010.

[20] JOCL. Java bindings for opencl. http://www.jocl.org/, Aug. 2016.

[21] D. Kim. Terasort for spark and flink with range partitioner, June 2015.

[22] LWJGL. Lightweight java game library 3. http://www.lwjgl.org, May 2016.

[23] e. a. Mikkel A. Madsen. Predicting the outcome of league of legends matches using machine learning on big data, 2015.

[24] V. . C. News. Every day big data statistics 2.5 quintillion bytes of data created daily. http://www.vcloudnews.com/every-day-big-data-statistics-2-5-quintillion-bytes-of-data-created-daily/, Apr. 2015.

[25] A. Ng. Logistic regression: Cost function. https://www.coursera.org/learn/machine-learning/lecture/1XG8G/costfunction, Oct. 2015.

[26] A. Ng. Logistic regression: Hypothesis representation. https://www.coursera.org/learn/machinelearning/lecture/RJXfB/hypothesis-representation, Oct. 2015.

[27] U. G. S. Nichlas Bo Nielsen. Optimizing item shopping for league of legends using machine learning. Technical report, Aalborg University, 2016.

[28] Nvidia. An easy introduction to cuda c and c++. https://devblogs.nvidia.com/parallelforall/easyintroductioncudacandc/, Oct. 2012.

[29] NVIDIA. Cuda. http://www.nvidia.com/object/cuda_home_new.html, Aug. 2016.

[30] Nvidia. Quadro quick specs. http://www.nvidia.com/object/quadro-desktop-gpus.html, Sept. 2016.

[31] Oracle. Trail: Rmi. https://docs.oracle.com/javase/tutorial/rmi/, Nov. 2016.

[32] N. Z. Y. C. Peilong Li, Yan Luo. Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms. 2015.

[33] I. Pointer. Apache flink: New hadoop contender squares off against spark. http://www.infoworld.com/article/2919602/hadoop/flinkhadoopsnew-contenderformapreducespark.html.

[34] M. Rouse. Apache hadoop yarn(yet another resource negotiator). http://searchdatamanagement.techtarget.com/definition/ApacheHadoopYARN-YetAnotherResourceNegotiator.

[35] SAS. Machine learning - what it is and why it matters. http://www.sas.com/en_us/insights/analytics/machine-learning.html, 2016.

[36] Spark. Apache hadoop. http://spark.apache.org/, Mar. 2016.

[37] H. Sutter and J. Larus. Software and the concurrency revolution. Queue, 3(7):5462,, 2005.

A. Machine Learning Techniques

A.1. Gradient Descent

- $Q(w)$ is the objective function where the parameter which minimizes w is to be estimated.
- n is the amount of entries in the dataset
- Q_i is the Q of the i-th iteration

$$Q(w) = \sum_{i=1}^{n} Q_i(w) \tag{2}$$

- α is the learning rate
- ∇ is the gradient of our function

$$w := w - \alpha \nabla Q(w) = w - \alpha \sum_{i=1}^{n} \nabla Q_i(w) \tag{3}$$

A.2. Logistic Regression

This is a section taken from a previous semester project[27].

Logistic regression classifies by forming a hypothesis, which can then be used to give a probability of a datapoint belonging to a given class. This hypothesis can be seen as a line that divides the data into two parts. Depending on the location of a new datapoint compared to the hypothesis, logistic regression will then determine the probability of that point belonging to one of the classes.

An example of how a hypothesis could look can be seen in Figure 22. In this figure there are two classes, the *red* and *blue* class and the hypothesis as the blue line. Logistic regression would guess that all points above the blue line belongs to the red class, and above, to the blue class.

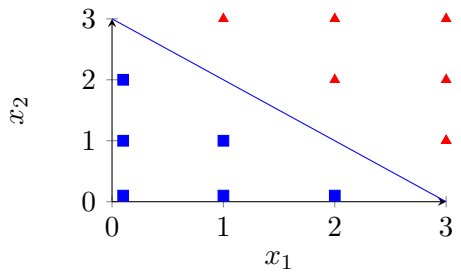

Figure 22: *Logistic regression example with the hypothesis being the line dividing the data*

Logistic regression finds a hypothesis $h_\theta(x)$, where x is the coordinates of the point that we want to find the probability for and θ is a parameter for defining the function of the hypothesis line. The hypothesis will then return the probability of the point belonging to one of the classes. For probabilities we would like them to be between 0 and 1:

$$0 \leq h_\theta(x) \leq 1 \tag{4}$$

When predicting y it is commonly done with a threshold as seen below:

$$\text{If } h_\theta(x) \geq 0.5, \text{ predict } y = 1$$

$$\text{If } h_\theta(x) < 0.5, \text{ predict } y = 0$$

The general hypothesis in logistic regression can be written as:

$$h_\theta(x) = g(\theta^T x)$$

The function g is defined as:

$$g(z) = \frac{1}{1 + e^{-z}}$$

This is called the *logistic function*. Figure 23 shows a plot of the basic logistic function. As the parameter z approaches ∞, the line $g(x)$ approaches 1 and 0 for $-z$ and $-\infty$. This ensures that the rules for the hypothesis seen in equation 4 will always be true, no matter how big x is.

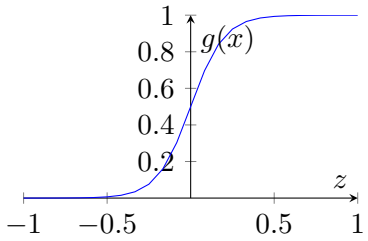

Figure 23: *Plot of the logistic function*

The hypothesis can now be used to predict which class a newly inserted data point has the highest probability of belonging to. Looking back at the basic logistic function shown on figure 23, the hypothesis would predict all variables with a z above 0 to be 1, as they have a probability higher than 0.5 [26].

The output of a hypothesis $h_\theta(x)$ can be interpreted as the probability that $y = 1$ given x, parameterized by θ. This can be written as:

$$h_\theta(x) = \mathrm{P}(y = 1|x, \theta)$$

This would only give the probability of $y = 1$, since y can only be either 1 or 0. We know how to find the probability of $y = 0$ as both probabilities must sum up to 1.

No hypothesis will fit all data distributions however, thus making it necessary to determine the optimal θ.

An example of why this is important could be if a point was added to the example in figure 22 resulting in figure 24, another hypothesis would fit the data better. In Figure 25 an example of a new hypothesis that captures the new point is given.

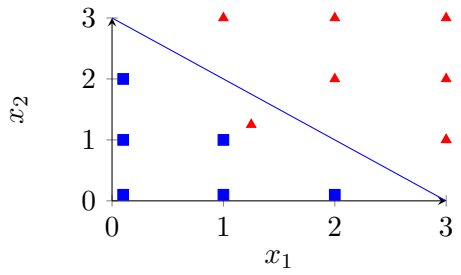

Figure 24: *Logistic regression example with a new added point*

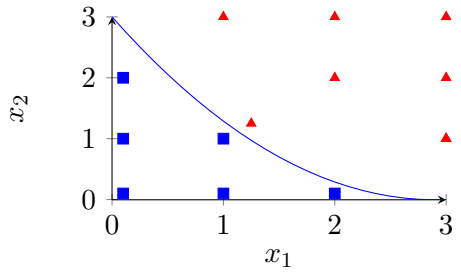

Figure 25: *Logistic regression example with a new hypothesis*

To fit the parameter θ one would use a cost function together with a learning algorithm that uses it. This learning algorithm could for example be one of the gradient descend algorithms, described in Section A.1

The cost function in general can be written as $Cost(h_\theta, y)$. This is read as the cost that the learning algorithm has to pay if it outputs $h_\theta(x)$ and the actual value is y. Given $y \in \{1, 0\}$, the cost function is as seen in Equation 5.

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases} \tag{5}$$

This cost function have some desirable properties. First of all it keeps the prediction value between 0 and 1, which important as we these are the only values that y can be predicted as. Secondly it gives an infinite cost if $h_\theta(x)$ returns 0 and the actual y value was 1, as this would be the same as it saying that outcome is impossible, which should never happen [25].

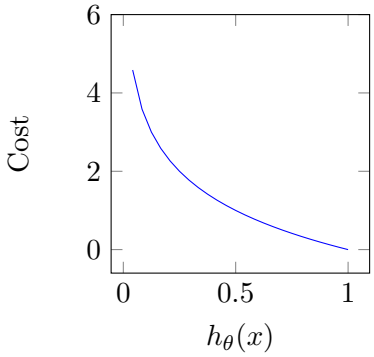

Figure 26: *Plot of* $-log(h_\theta(x))$

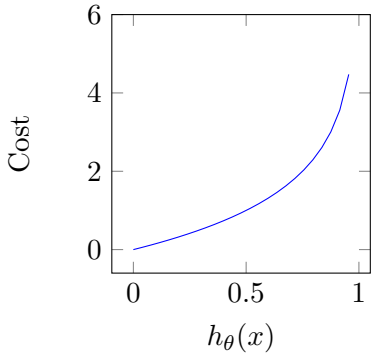

Figure 27: *Plot of* $-log(h_\theta(1-x))$

A.3. K-Means

This section introduces the concept of the well-known machine learning technique k-means, this section is based on the work of David L. Poole an Alan K. Mackworth[5]. k-means is an algorithm used for hard clustering, and utilizes the expectation-maximization(EM) model. Expectation maximization is a class of algorithms that work in two steps, the expectation(E)-step and maximization(M)-step. The E step classifies the data base on the current model and the M generates the best possible model based on the current classification of the data. The algorithm repeats the E and M step until the model converges.

k-means requires some training examples and a number of classes k as input. The output of the algorithm is a set of k classes, and a prediction of what the value is for each feature for each class, as well as an assignment of each training example to a class. The algorithm requires that each feature is given as a numerical, and thus limits the data it can be used on. K-means tries to find a set of classes that minimize the sum of squares errors for the predicted values for each example when derived from the class it belongs to.

Suppose E is the set of all training examples and the input features are the are $X_1, \ldots, X_n$. Let $val(e, X_j)$, be the value of input feature X_j for training example e. We assumed that ther are observed We will associate a class with each integer $i \in \{1, \ldots, k\}$. Given this the k-means algorithm outputs the following:

- A function $class : E \to \{1, \ldots, k\}$, meaning the $class(e)$ is the classification of a training example, e.g. if $class(e) = i$ e is in class i.
- a $pval$ function for each class $i \in \{1, \ldots, k\}$ and for each feature $X_J, pval(i, X_j)$ is the prediction for each training example in class i for feature X_j.

For at particular $class$ function and $pval$ the sum of squares error is as defined in Equation 6.

$$\sum_{e \in E} \sum_{j=1}^{n} (pval(class(e), X_j) - val(e, X_j))^2 \tag{6}$$

The goal of k-means is to find a $pval$ and $class$ function that minimizes the sum of squares error.

At the start of the first iteration of the algorithm, each class is randomly assigned to a class. After this, it follows the EM model by doing the two following steps[5]:

- M-Step For each class i and feature X_j, assign to $pval(i, X_j$ the mean value of $val(e, X_j$ for each training example in class i:
- $$pval(i, X_j) \leftarrow \frac{\sum_{e:class(e)=i} val(e, X_j)}{|\{e : class(e) = i\}|} \tag{7}$$

  Where the denominator is the number of examples in class i.

In the E-step each training examples is assigned to a class. Each e is assigned to the class i that minimizes the following:

$$\sum_{j=1}^{n} (pval(i, X_j) - val(e, X))^2 \tag{8}$$

These two steps are done until convergence. k-means converges in when one of two conditions is met. First first is a defined iterative value, that simply sets how many iterations of the algorithm we wish to run. The second is when there both an E and M step has been completed without any change in any of the assignment of the classes.

B. Framework Code Examples

The memory allocator is one of the most essential functions of the framework. The memory allocator computes the amount of instances each kernel can handle based on the information it fetches about the hardware. The input to the memory allocator is the target OpenCL device, the data that needs to be copied and how many instances of it there is, and in this example the size of the weights used.

First the memory allocator utilizes two functions in the OpenCL framework, $CL\_DEVICE\_GLOBAL\_MEM\_SIZE$ and $CL\_DEVICE\_MAX\_COMPUTE\_UNITS$. These return the global memory size of the target OpenCL and how many compute units it has available. In this example it is assumed that the types of the data is Double. The total memory for each kernel is then calculated. The memory needed for the weights is subtracted. The size of a single data instance is then calculated based on the amount of data instances and total size of the data array. This is used to calculate how many kernel can handle at a maximum.

After all this, there is a check if the amount of data instances is less than the total amount the target OpenCL device can handle. If this is true, the instances are evenly divided between the kernels and the number of iterations needed go through the data is set to 1. This is then returned in a custom type memoryObject which contains the information about the iterations and instances per kernel.

If the check is false, the memory allocator calculates how many iterations that are needed to go through all the data. After this there is a check if the data allocated is a perfect fit for the kernels, and if this is not true, the amount of iterations is incremented by 1. This is due to the fact that the calculation of iterations is based on integer division. Finally a memoryObject with the information is returned.

```
//Input: Current OpenCL Device, data memory, amount of data
    instances, size of weights and results.
//Output: MemoryObject containing the number of iterations needed
 and the number of instances that should be used for each iteration
private MemoryObject calculateMemoryAllocation(CLDevice currentDevice,
    double[] dataMemory, int dataInstances, int weightSize)
{
    long globalMemSize = currentDevice.getInfoLong
        (CL_DEVICE_GLOBAL_MEM_SIZE);
    long computeUnits = currentDevice.getInfoInt
        (CL_DEVICE_MAX_COMPUTE_UNITS);

    int sizeOfDouble = Double.BYTES;

    long memPrKernel = globalMemSize/computeUnits;
    long memLeft = memPrKernel;
    memLeft -= (weightSize*2) * sizeOfDouble;

    int instanceSize = dataMemory.length/dataInstances;
    dataSize = instanceSize;
    int sizeOfInstance = instanceSize * sizeOfDouble;


    long maxInstancesPrKernel = memLeft / sizeOfInstance;
    long instancesPrKernel;
    long iterations;

    if(dataInstances < (maxInstancesPrKernel*computeUnits))
    {
        instancesPrKernel = dataInstances/computeUnits;
        iterations = 1;
        MemoryObject memoryObject = new MemoryObject(iterations
            ,computeUnits,instancesPrKernel);
```

```
            return memoryObject;
        }
        else
        {
            instancesPrKernel = maxInstancesPrKernel;
            iterations = dataInstances / instancesPrKernel;
        }

        int dataTotalBytes = dataInstances * sizeOfInstance;

        if((dataTotalBytes != 0) && !(dataTotalBytes % instancesPrKernel == 0))
            // Handle case of perfect memory fit
            iterations += 1;
        MemoryObject memoryObject = new MemoryObject(iterations,
            computeUnits,instancesPrKernel);
            //MemoryObject contains iterations and instancesPrKernel

        return memoryObject;
    }
```

C. Standard Library Kernels

C.1. Logistic Regression

A sample implementation of a kernel of logistic regression. The kernel consists of three functions, sigmoid, classify and logistic. The main function is the logistic function, which is the one of the standard functions in the framework, and the kernel called from the Java portion of the framework. The functions has a few inputs as follows:

- dataMem: A pointer to a read-only array of doubles, consisting of the data given by the user.
- weightsMem: A pointer to a read-only array of doubles, consisting of the weights given by the user.
- answerMem: A pointer to a write-only array of doubles, this is given by the automatically generated memory allocator.
- instances: An integer generated by the memory allocator which specifies how many data instances the kernel is to run.

Before the kernel is compiled, the preprocessor gets the program as a string and replaces any dynamic variables with their final value as generated by the memory allocator. An example of this is the variable WEIGHTLEGTHVALUE, which is a placeholder for the final value of the length of the array which is to store the temporary weights used in the logistic regression computation.

```
double sigmoid(double z)
{
    double e = M_E_F;

    double rNum = 1.0/(1.0 + pow(e,-z));
    return rNum;
}

double classify(double* x, global const double* weights)
{
    double logit = 0.0;

    for(int i = 0; i < 13; i++)
    {
        logit += x[i] * weights[i];
    }
    double temp = sigmoid(logit);
    return temp;
}

kernel void logistic(global const double* dataMem, global const
    double* weightsMem, global double* answerMem, const int instances) {

    unsigned int xid = get_global_id(0);
    double rate = 0.00001;
    int datLength = DATLENGTHVALUE;
    int range = instances;
    double label = 0.0;
    double tempWeights[WEIGHTLENGTHVALUE];
    double currentData[DATLENGTHVALUE];

    for(int i = 0; i < range; i++)
    {
        for(int j = 0; j < datLength; j++)
        {
            int a = sizeof(dataMem);
            if(j == 0)
            {
                int temp = (xid*range)+(i*datLength)+j;
                label = dataMem[(xid*range)+(i*datLength)+j];
            }
            else{
                currentData[j] = dataMem[(xid*range)+(i*datLength)+j];
```

```
            }

        }
        double predicted = classify(currentData,weightsMem);
        for(int j = 0; j < WEIGHTLENGTHVALUE; j++)
        {
            double tempValue = weightsMem[j] + rate *(label - predicted)
                * currentData[j];

            tempWeights[j] = tempWeights[j] + tempValue;
        }
    }
    for(int i = xid*WEIGHTLENGTHVALUE; i < xid*WEIGHTLENGTHVALUE
         + WEIGHTLENGTHVALUE; i++)
    {
        answerMem[i] = tempWeights[i];
    }
}
```

C.2. K-Means

A sample implementation of a kernel of K-Means. The kernel consists of two functions, calcDistance and kmeans. The main function is the kmeans function, which is one of the standard library functions in the framework. This is the function that is called by the Java portion of the program. The main function has a few inputs as follows:

- dataMem: A pointer to a read-only array of doubles, consisting of the data given by the user.
- centroidMem: A pointer to a read-only array of doubles, consisting of the centroids generated by the framework.
- answerMem: A pointer to a write-only array of doubles, this is given by the automatically generated memory allocator.
- instances: An integer generated by the memory allocator which specifies how many data instances the kernel is to run.

Before the kernel is compiled, the preprocessor gets the program as a string and replaces any dynamic variables with their final value as generated by the memory allocator. In this case it is only CENTROIDLENGTH that needs to be replaced by the preprocessor. It is replaced by the value the user has given for the amount of centroids the kernel is using.

```
double calcDistance(double* instance, double* centroid)
{
```

```
    double result = 0.0;
    for(int i = 0; i < CENTROIDLENGTH; i++)
    {
        double cFeature = centroid[i];
        double iFeature = instance[i];
        double toPow = iFeature - cFeature;
        double tempResult = pow(toPow,2);
        result += sqrt(tempResult);
    }
    return result;
}

kernel void kmeans(global const double* dataMem, global const double*
        centroidMem, global double* answerMem, const int instances) {
    unsigned int xid = get_global_id(0);
    int datLength = CENTROIDLENGTH;
    int range = instances;
    double kernelCentroid[CENTROIDLENGTH];
    int centroidStart = xid*CENTROIDLENGTH;
    double currentInstance[CENTROIDLENGTH];
    for(int i = 0; i < CENTROIDLENGTH; i++)
    {
        kernelCentroid[i] = centroidMem[centroidStart+i];
    }

    for(int i = 0; i < range; i++)
    {
        for(int j = 0; j < CENTROIDLENGTH; j++)
        {
            currentInstance[j] = dataMem[(xid*range)+(i*datLength)+j];
        }
        int updateLocation = xid*range + i;
        double temp = calcDistance(currentInstance,kernelCentroid);
        answerMem[updateLocation] = temp;
    }
}
```

C.3. Source Code

The full source code is available on: https://bitbucket.org/ulfsim/dat10 . This repository can freely be cloned in git or downloaded as a zip file.