Leopold–Franzens–University
Innsbruck

Institute of Computer Science
Distributed and Parallel Systems

GPGPU on Apache Hadoop

Master Seminar 2 (SS 2012)

Supervisor: Prof. Dr. Thomas Fahringer

Martin Illecker

Innsbruck, June 12, 2012

GPGPU on Apache Hadoop

Martin Illecker

martin.illecker@student.uibk.ac.at

Abstract This seminar thesis gives a short introduction into *Apache Hadoop* and its *MapReduce* framework. The main part will show the different possibilities to use *Apache Hadoop* not only on *Central Processing Units, CPUs* also on graphics processing units, so called *General-Purpose computing on Graphics Processing Units, GPGPU*.
Apache Hadoop is a free cluster system which supports data-intensive distributed applications. Beside its cluster management functions *Apache Hadoop* provides also the *MapReduce* framework. The *MapReduce* framework is a programming model which helps the user to program jobs for huge data sets.
This paper will show how *Apache Hadoop* can be used on graphics processing units and therefor two technologies are presented. *Open Computing Language (OpenCL)* is the first technology which was developed by Apple and later by the Khronos Group. The second technology is *Compute Unified Device Architecture (CUDA)* which is hosted by NVIDIA.
At the end the paper will summarize these approaches and give a short conclusion.

Contents

1 Introduction

Since the turn of the millennium the clock rate, which always has been an indicator for computing power of processors, remains nearly static. One of the biggest problems for the more and more shrinking structures was the thermal power. Operating at a higher clock rate always requires more power. But the need of increasing computing power especially in economy and research will never end. After that critical point a new way to improve the performance was to develop multiple processors or multiple cores on a chip. Another possible solution to improve computing power is to combine computers to a cluster. But the high energy and space consumption of computer clusters leads to a quite new technology called *General-Purpose computing on Graphics Processing Units, GPGPU. GPGPU* uses the high potential of parallelism available in graphics processing units with more than hundred cores, so called stream processors. By default GPUs are constructed in high grade of parallelism to calculate a lot of instructions in parallel, for example matrix operations or other mathematical problems. *GPGPU* gives the possibility to execute also common calculations on these stream processors. The goal would be to substitute the *Central Processing Unit, CPU* with a *Graphics Processing Unit, GPU* or to combine their computing power.

1.1 Overview of the document

The introduction at the beginning gives a short overview about cluster computing. The available technologies of GPGPU, the programming and memory model are introduced in chapter 2. In the next chapter Apache Hadoop and its components are presented. Chapter 3 explains the programming model MapReduce which is used by Hadoop. After that the two main chapters 4 and 5 will show how Apache Hadoop can be used on GPUs. Therefor the following two technologies are used: OpenCL which is used in [1] and CUDA which is based on the development of [2] and [3]
Finally this seminar paper ends up with a conclusion.

1.2 Cluster computing

Cluster computing means solving a problem on a cluster of workstations. Clusters consist of a set of connected computers, which work together and act like one single system. This principle of parallel transparency is called *Single System Image (SSI)*. Normally a user does not know about all that computers in the background and is only communicating with the front-end.
The computers in such a cluster are called nodes and these nodes can be separated in master nodes and slave nodes. Master nodes are managing and controlling the cluster. Slave nodes or also called "workers" or "compute nodes" provide the resources like memory or computational power and do the real job. Clusters can be differentiated in the following categories: [4]

1. **Homogen Cluster**
   Clusters are homogeneous, if all nodes in the cluster run the same operation system and are built of the same hardware architecture.

2. **Heterogen Cluster**
   Contrary to homogeny clusters, heterogen clusters use different operation systems and different hardware architectures.

3. **Central Cluster or "Glass-House Cluster"**
   A Glass-House Cluster means that all nodes are located in one central location, for example one server room. Nodes are located closed to each other. Centralized clusters are often also homogeneous and use high speed networks to connect each node.

4. **Decentral Cluster or "Campus-Wide Cluster"**
   Decentralized clusters are distributed over a large area and are very homogeneous in serval ways. Nodes of such a cluster can change at any time and their connection to each other is normally not a high speed network. One big benefit of Campus-Wide Clusters is their distribution which minimizes the vulnerability against loss of power, fire and so on.

1.3 General-Purpose computing on Graphics Processing Units

A *Graphics Processing Unit, GPU* typically handles only computations for computer graphics. But in the quite new field of *General-Purpose computing on Graphics Processing Units, GPGPU* the GPU performs also computations which are normally processed by a CPU. Since 2000 when Microsoft introduced Direct3D 8.0 with its Shader concept, the programming model capabilities increased more and more. The parallel nature of GPUs caused by vector and geometry calculations leads us now to about 3072 Shader processors of the new Nvidia GeForce GTX 690. This graphic card in picture 1 has two Kepler GK104 GPUs with each 1536 Shaders and 4 GB of GDDR5 memory (2GB per GPU) on it and reaches about 5,6 teraflops (floating-point operations per second). Additionally these cards can go quad-SLI (four cards in one computer system). In contrast to GPUs the newest CPU of Intel codenamed Knights Corner presented in November 2011 at a supercomputing conference in Seattle hosts 50 x86 cores on one chip and reaches only one teraflop. [5]

To use this huge power of GPUs the following technologies can be used:

1. **CUDA**
   Compute Unified Device Architecture (CUDA) was developed by NVIDIA and is a parallel computing architecture for GPUs. Unlike OpenCL, CUDA only works with NVIDIA GPUs. CUDA supports nearly all standard operating systems and its main programming language is C. But there exist also some third party wrappers for example for Java, Python, Perl, Fortran

Figure 1. Nvidia GeForce GTX 690 (28nm) with 5,6 teraflops from [6]

and so on. Since Hadoop is implemented entirely in Java, JCuda proves a solid foundation for bringing CUDA technology into Java applications on the Hadoop framework. [3] CUDA offers the availability to access the virtual instruction set and memory of the parallel computational elements. The features of CUDA increase with every new version of it and the current stable version is 4.2. The new Nvidia GeForce GTX 690 for example supports CUDA Compute Capability 3.0. [7]

2. **OpenCL**
   Open Computing Language (OpenCL) was initially developed by Apple and now by the Khronos Group. OpenCL is a free open standard. Unlike CUDA, OpenCL was not designed to support GPUs only. OpenCL is a framework which helps to write programs for heterogeneous platforms consisting of CPUs, GPUs or other processors. The main programming language for OpenCL is also C and there exist a lot of third party wrappers. The current version of OpenCL is 1.2 from November 2011 which is supported of every new CPU or GPU.[8]

3. **Direct Compute**
   Direct Compute is developed by Microsoft and the third important technology in GPGPU. Direct Compute is an application programming interface and belongs to Microsofts's DirectX. But Direct Compute is hardly used in the field of high performance computing.

Programming Model
Every GPGPU program consists of two components:

1. **Host Program**
   The host program is a system which manages resources like memory and hosts the user defined kernel application.

2. **Kernel**
   A kernel can be executed on one or more graphic cards and is always embedded in a host program.

In GPGPU normally data parallelism is used. Data parallelism means executing the same code on different independent data. Before executing a kernel the index space has to be defined, which sets the amount of maximum kernel instances. One kernel instance is called a Work-Item and more kernel instances can be grouped to Work-Groups.

Memory Model
The general memory model is presented in Figure 2. Compute Devices are e.g., graphic cards or CPUs. A Host can operate with one or more Compute Devices. A Work-Item has no direct access to the host memory. But the host system can copy data from host memory to global memory and constant memory. All work items are allowed to read and write data from the global memory but not to allocate. In the constant memory Work-Items are allowed to allocate and read data. All Work-Items of a Work-Group share a common local memory. [1]

2 Apache Hadoop

Apache Hadoop is a free Java based framework that supports data-intensive distributed applications. Hadoop is a high performance computing cluster framework which is developed by the Apache Software Foundation. In 2008 the Apache Group defined Hadoop as a top level project. The MapReduce programming model and Hadoop's Distributed File System (HDFS) are core elements of Hadoop, although the MapReduce open-source implementation and the HDFS were derived from Google's MapReduce and Google File System (GFS).
Hadoop was developed in Java which is also the main programming language. The latest stable release version 1.0.3 was published on May 16, 2012. [9] [10]

Hadoop also hides a lot of details e.g., automatic parallelization, load balancing, network and disk transfer optimization, handling of machine failures and so on. It offers three major components for a user: [1]

1. **Job Management System**
   The job management system manages all jobs and tasks which are executed on the cluster. If a job doesn't finish successfully the job management system

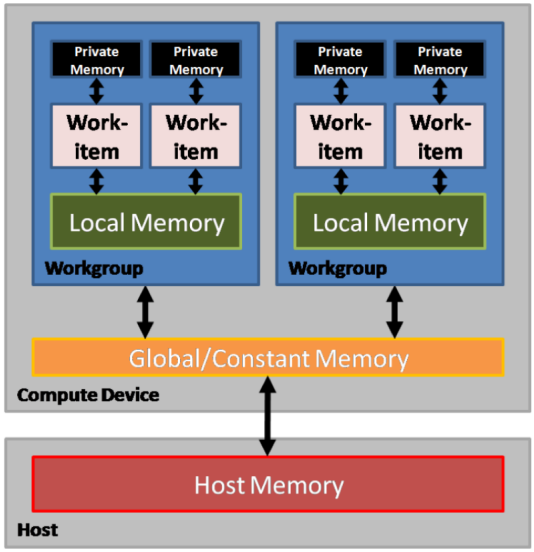

Figure 2. Memory Model from [1]

will execute this job on another node.

2. **Distributed Filesystem**
   All data which are processed by Hadoop are stored on a distributed filesystem. Hadoop supports different file systems but the default one is Hadoop Distributed File System (HDFS). HDFS is a filesystem designed for storing very large files on clusters of commodity hardware. Data will be stored redundant and distributed over several nodes. The block size and replication factor can be configured. HDFS also offers the great feature called data locality, which means executing a job on a specific node where the corresponding input data are already stored. There is no need to transfer large data set over the network.

3. **Programming Model MapReduce**
   Hadoop offers a MapReduce framework which helps the developer to program a job very easily.

2.1 Components

Apache Hadoop has four core components which can be categorized in distributed storage and distributed computing. These components are based on a Master-Slave architecture. Master nodes are responsible for managing the cluster and Slave nodes do the real work. The four core components are: [9]

1. **JobTracker (Master)**
   The *JobTracker* coordinates all jobs and schedules parts of a job so called tasks to the *TaskTracker*. If a task fails the *JobTracker* will run in on another *TaskTracker* again. The *JobTracker* splits the job into tasks by using the block size of the distributed filesystem. The submitted jobs will also be added into an internal queue from where the job scheduler will pick it up.

2. **TaskTracker (Slave)**
   TaskTrackers are the slave nodes which run the tasks and do the work. It constantly sends progress reports to the *JobTracker* to show that the *TaskTracker* is still working and alive.

3. **NameNode (Master)**
   Like the *JobTracker* is master in sense of distributed computing, the *NameNode* is the master of the distributed storage. *NameNodes* store data on *DataNodes* and manage all the references. A *NameNode* is also responsible for redundancy and data consistency. If a *DataNode* fails the *NameNode* changes the reference to a redundant *NameNode*. Only the *NameNode* service includes the risk of a single point of failure. *NameNode* and *JobTracker*, which are master nodes can be started twice to minimize the risk of a single point of failure. The *Secondary NameNode* can be configured as an Active/-Passive cluster.

4. **DataNode (Slave)**
   DataNodes store the real data blocks on their local filesystem. All *DataNodes* together build the whole distributed file system (DFS). *DataNodes* communicate with each other to replicate the data and update changes. Every *DataNode* sends continuous heartbeats to the *NameNode* to show that it is alive.

Figure 3 shows the interaction of these components.
A Client submits a job to the JobTracker in process (a). The master node JobTracker splits the job in several tasks and then sends these tasks then to the TaskTracker (b). To execute the task, the TaskTracker needs the input data from the distributed file system (DFS). The optimal case would be to receive the input data from DataNode service on the same node caused by data locality (c). The input data has to be submitted from the client to the DFS (NameNode) before the job starts (d). The NameNode is the master of the distributed storage and will split and distribute the data over all DataNodes (e). Now the TaskTracker can execute the task with the input data which came directly from one NameNode. The results will be stored in the DFS.

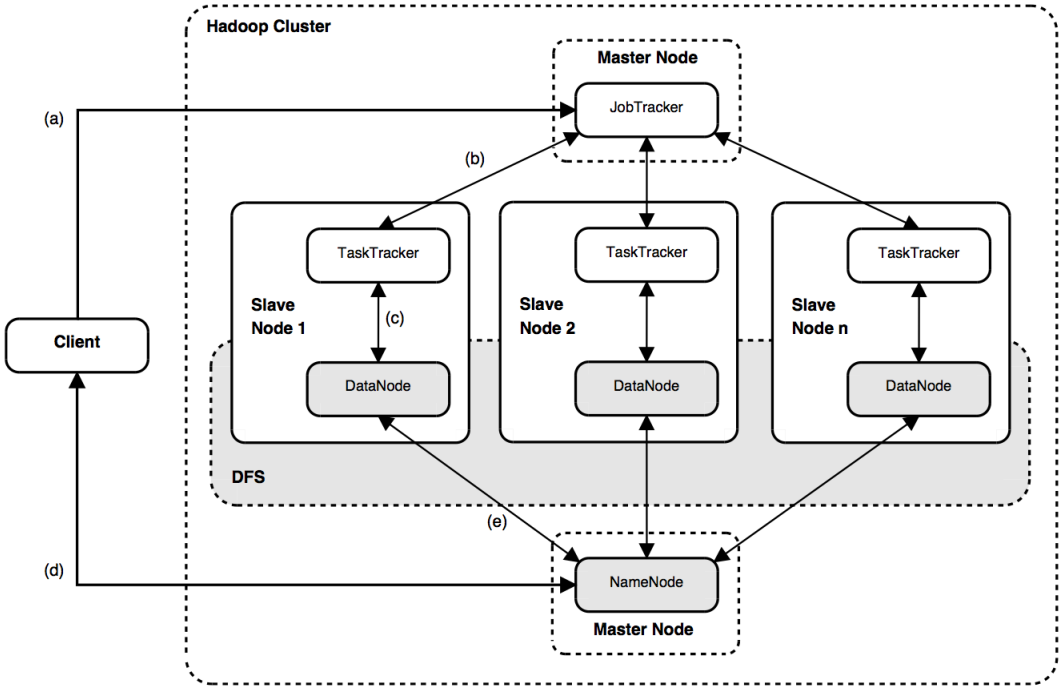

Figure 3. Components of Hadoop Cluster from [1]

3 MapReduce

MapReduce is a programming model which was initially developed by Google. The map and reduce functions come from one of the oldest functional programming language Lisp. Hadoop provides this MapReduce framework. A programmer only has to specify the map and reduce function. Each map invocation needs a key-value pair as input and outputs a sequence of independent key-value pairs. In the shuffle phase values are grouped together and passed to a reduce function, which processes them and outputs a new sequence of key-value pairs. This connection of input and output key-value pairs is shown in table 3.

Phase	Input	Output
Map	$< Key_1, Value_1 >$	$List(< Key_2, Value_2 >)$
Reduce	$< Key_2, List(Value_2) >$	$List(< Key_3, Value_3 >)$

Table 1. Input and Output of Map and Reduce

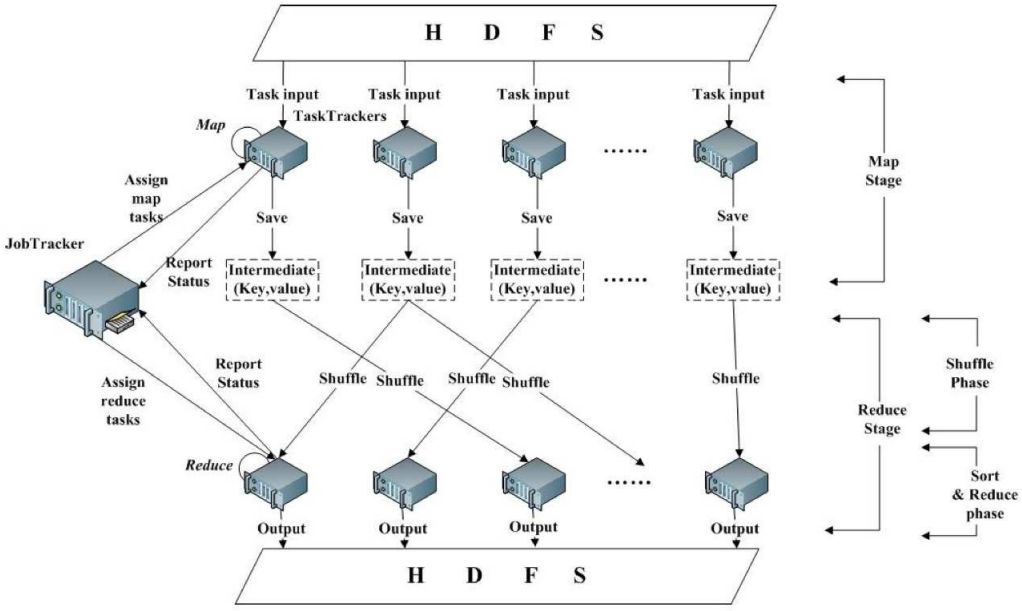

Figure 4. MapReduce Framework in Hadoop from [3]

Figure 4 illustrates the structure of the MapReduce Framework in Hadoop. At the beginning the client submits a job to the master node JobTracker. The JobTracker splits the job's input date on HDSF into tasks. Then the job is scheduled in a queue and depending on submission time, priority, user group or other parameters the tasks will be assigned to TaskTrackers. Every TaskTracker sends status reports to the JobTracker to update the current state.
At the map stage a task will issue a new JVM to run this map task. The TaskTracker receives the input data from the HDFS and employs the map function. The output of the map function are intermediate key-value pairs, which are stored in TaskTracker's local memory or local disk. The size of local memory for intermediate results can be configured.
The reduce stage will start when the first map task has finished. Hadoop's MapReduce framework will create one reduce task for each key of the intermediate results in default. But programmers can define the amount of reduce tasks to get the best performance. The reduce task consists of three phases: shuffle, sort and reduce. The shuffle phase will group all the intermediate key-values pairs from the map task by the key. The sort phase sorts the key-value pairs according to their value. At the end the reduce function is executed and the final key-value pairs are stored in the DFS. [3]

4 GPGPU using OpenCL

This chapter will show how OpenCL can be integrated into Hadoop in [1].
OpenCL specifies the following definitions: Compute Devices are devices that support OpenCL e.g., graphic cards or CPUs. A Host consists of one or more Compute Devices. A Compute Device can also have one or more Compute Units e.g., Shaders in graphic cards or cores in a CPU. A Compute Unit consists of one or more Processing Elements.

4.1 Parallelism Strategies

There exist two parallelism strategies for involving the GPU. The first option would be to integrate the GPU in task scheduling. This would be very complex and is explained in [2]. Therefore the TaskTracker of Hadoop has to be extended with a scheduling for CPU and GPU. Each task has to be implemented for CPU and GPU.
[1] chooses the second parallelism strategy, which is task parallelism. A task can be executed either on CPU or GPU but not on both of them. Therefore a scheduling or dynamic selection of the task implementation is not necessary. A task has to be explicit implemented for a graphic card.
Figure 5 illustrates the two-stage task parallelism strategy. In the first stage the JobTracker splits the job in tasks. The second stage starts the task on the CPU which can later invoke the GPU. Function or data parallelism is required to make the second stage possible. On a graphic card only one kernel can be executed.

4.2 Hadoop and OpenCL

Apache Hadoop offers three possibilities for using other programming languages than Java. **Hadoop Streaming** is provided by the framework and allows users to write their map and reduce function in any languages. It uses Unix standard streams as the interface between Hadoop and the program. The second option to connect to Hadoop are pipes, which is a C++ interface. Unlike Streaming, **Hadoop Pipes** use sockets. TaskTrackers can communicate with other processes over these sockets. The third option is **JNI**, a native programming interface which allows Java to invoke applications and libraries written in C and C++.
[1] selected JavaCL for the best communication of OpenCL with Hadoop. JavaCL uses JNA to access OpenCL. The benefit of using Java as the core programming language is full access to all functions of the Hadoop API.

4.3 OpenCL Integration

First of all OpenCL needs a lot of initialization, e.g., kernel creation, platform and device requests, context and CommandQueue creation. But should these initialization work be implemented in each map or reduce phase? Of course not,

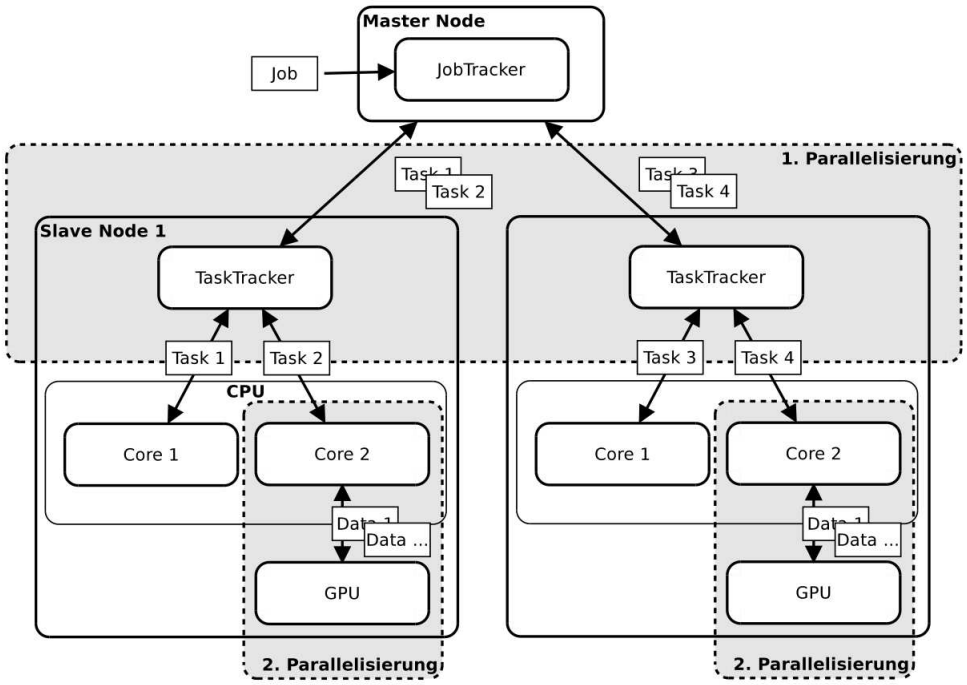

Figure 5. Task Parallelism from [1]

because the map and reduce function are executed for every key-value pair. But the Hadoop framework offers a setup and cleanup method. Setup will be executed once when a task starts. At the end of a task cleanup will be called.
Another important performance indicator is data management. All input data from the main memory have to be copied to the memory of the graphic card, before a job can be run. [1] figured out that the fastest way is to copy coherent data in a buffer from the main to the graphic card memory. To transfer data by an array it is necessary to know the size of the buffer. OpenCL API offers a function to get the size of the graphic card memory at runtime.
The following pseudocode from [1] illustrates a possible OpenCL integration including simple data management.

OpenCL Integration including Simple Data Management

```
1 Mapper:
2         setup():
3                 list = List()
4
5         map(key, value):
6                 list.add(key, value)
```

```
7
8           cleanup():
9                   input[] = List.toArray(list)
10                  openCL.copyTo(device , input)
11                  openCL.execute(kernel)
12                  results[] = openCL.copyFrom(device)
13                  foreach(r in results):
14                          mapPut(r.key, r.value)
15  Reducer:
16          reduce(key, iterator):
17                  list = List()
18                  foreach(value in iterator):
19                          list.add(key, value)
20                  input[] = List.toArray(list)
21                  openCL.copyTo(device , input)
22                  openCL.execute(kernel)
23                  results[] = openCL.copyFrom(device)
24                  foreach(r in results):
25                          reducePut(r.key, r.value)
```

The simple data management pseudocode saves all key-value pairs in the map phase to the main memory. This will cause a memory overflow if the input data exceeds the main memory size. After all key-value pairs have been executed in the map phase the cleanup method of the mapper is called. The cleanup method implements the real work. First it copies the data to the OpenCL device memory. After that it executes the kernel applications and transfers the results back to the main memory. The last step is to build key-value pairs from the results, which will be the input for the reducer.
The reducer doesn't need a setup or cleanup method because the reduce method will be called for a group of key-value pairs (grouped by the key). The second big drawback of this implementation is the complexity of memory management. Every reduce execution increases the memory usage of Java Virtual Machine (JVM). The Garbage Collector (GC) has to be executed very often which also will increase the execution time.

The following pseudocode OpenCL integration optimized data management from [1] tries to solve the problems from the simple data management before. The only way to minimize the memory usage is to reuse an array. Only if an array is too small a new bigger array has to be allocated.

OpenCL Integration including Optimized Data Management Mapper

```
1 Mapper:
2           setup():
3                   buffer = Buffer()
4
5           map(key, value):
```

```
6                   if(!buffer.put(key, value)) //until buffer is full
7                           computeOpenCL(buffer)
8                           buffer.put(key, value) // add previous key-value pair
9
10          cleanup():
11                  if(buffer.hasData)
12                          computeOpenCL(buffer)
13                          buffer.reset() //setting the counter to zero
14
15          computeOpenCL(buffer):
16                  openCL.copyTo(device, buffer.data, 0, buffer.count)
17                  openCL.execute(kernel)
18                  results[] = openCL.copyFrom(device)
19                  foreach(r in results)
20                          mapPut(r.key, r.value)
```

The optimized data management pseudocode uses an implemented Buffer class. This buffer counts the current key-value pairs so that the array size can be bigger than the current key-value pairs amount. The map function adds key-value pairs in the buffer until the buffer is full or there is no input data left. When a buffer is full computeOpenCL sends data to the OpenCL device, executes the kernel and adds the resulting key-value pairs. Finally if the buffer still has data in it, computeOpenCL is called once more.

OpenCL Integration including Optimized Data Management Reducer

```
1 Reducer:
2           setup():
3                   buffer = Buffer()
4
5           reduce(key, iterator):
6                   buffer.reset()
7                   foreach(value in iterator):
8                           if(!buffer.put(key, value))
9                                   computeOpenCL(buffer)
10                                  buffer.put(key, value)
11                  if(buffer.hasData)
12                          computeOpenCL(buffer)
13
14          computeOpenCL(buffer):
15                  openCL.copyTo(device, buffer.data, 0, buffer.count)
16                  openCL.execute(kernel)
17                  results[] = openCL.copyFrom(device)
18                  foreach(r in results)
19                          reducePut(r.key, r.value)
```

The Reducer is very similar to the Mapper except that the previous cleanup is done at the end of the reduce function. The reduce function gets a key and a group of corresponding values. The first step is to reset the buffer, which sets the counter to zero. After that the buffer is filled with the key-value pairs. If the buffer is full or it still has data at the end, the computeOpenCL function will be called.

5 GPGPU using CUDA

This chapter will illustrate how Hadoop can interact with CUDA. CUDA has many benefits compared to other GPGPU solutions. It supports Scattered Reads, i.e., a program can access memory at arbitrary addresses on host and graphic device. Different hardware threads on the GPU are allowed to access a shared memory region. But CUDA also has some drawbacks e.g., CUDA-C disallows the usage of recursion and function pointers. [3]
The bandwidth of the bus between main memory and GPU memory is a bottleneck for CUDA as well as OpenCL.

5.1 CPU and GPU Architecture

Figure 6 shows the CPU and GPU architecture. Eight steps are needed to demonstrate a simple array summation example. First of all the colored squares in the GPU box are explained. The green ones are computation elements. GPU's global memory is indicated by a blue square. The smaller red squares are shared memories for a certain amount of computation elements.
The first step allocates three arrays in the main memory. After that also three arrays are created in the global memory of the GPU. Then two arrays, which should be summarized, are copied to GPU's global memory. Before adding the values in step five the values have to be loaded into the shared memory. Then the result is stored in the global memory of the GPU and copied back to the main memory. Finally all arrays will be deallocated.

5.2 Parallelism Strategies

In [3] they integrated GPU computation into the map stage. That is the same task parallelism strategy as shown in 5. The *TaskTracker* executes the map function on the CPU or in one core. The map function calls CUDA code. The second parallelism stage needs both CPU and GPU because GPU needs the CPU to process its input and output e.g., for accessing main memory.
[2] presented a hybrid map task scheduling which is very complex because the *TaskTracker* of Hadoop has to be extended and each job has to be implemented for CPU and GPU.

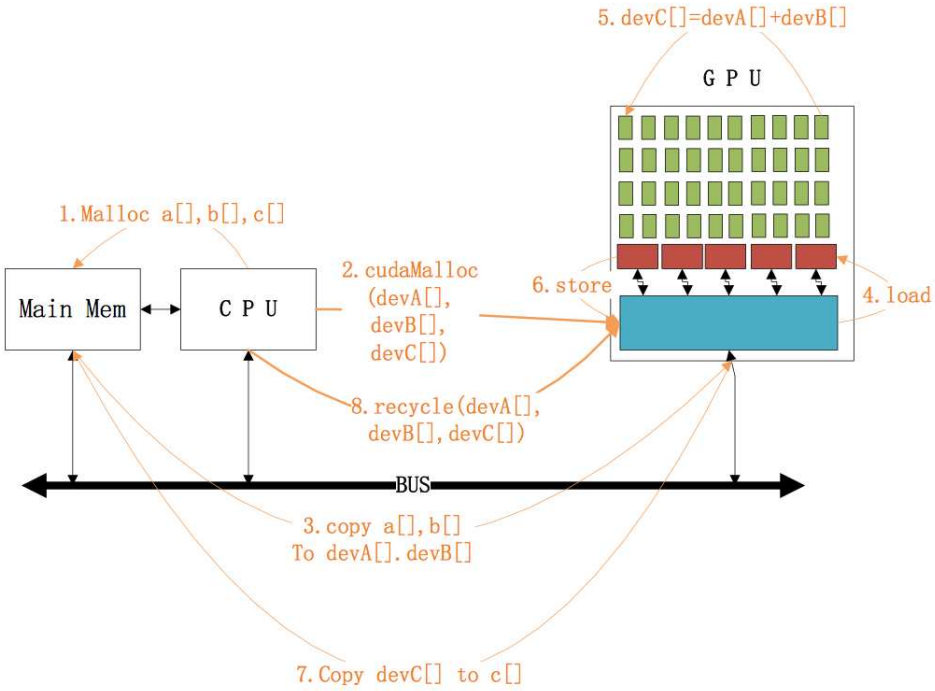

Figure 6. CPU and GPU Architecture from [3]

5.3 Hadoop and CUDA

CUDA-C++ can efficiently interact with Hadoop Pipes which is a C++ interface. But since Hadoop is implemented entirely in Java the better solution would be to use JCuda. JCuda is a CUDA binding for Java and is fully interoperable among different CUDA based libraries. [3] used Hadoop Pipes for connecting Hadoop and CUDA.

5.4 CUDA Integration

To implement CUDA on Hadoop CUDA the computation has to be formulated in terms of MapReduce mappers and reducers. At the beginning the typical structure of most CUDA programs have to be analyzed. A CUDA program consists of: [3]

1. **Kernel Configuration and Launching**
   Kernel configuration and launching code can be ported into a mapper function. The map function should perform the computation using a CUDA kernel.

2. **Control Computation (CPU)**
   This CPU-based control function is only for comparison purpose.
3. **Result Analysis and Reporting**
   Result analysis and reporting must not be ported to Hadoop.

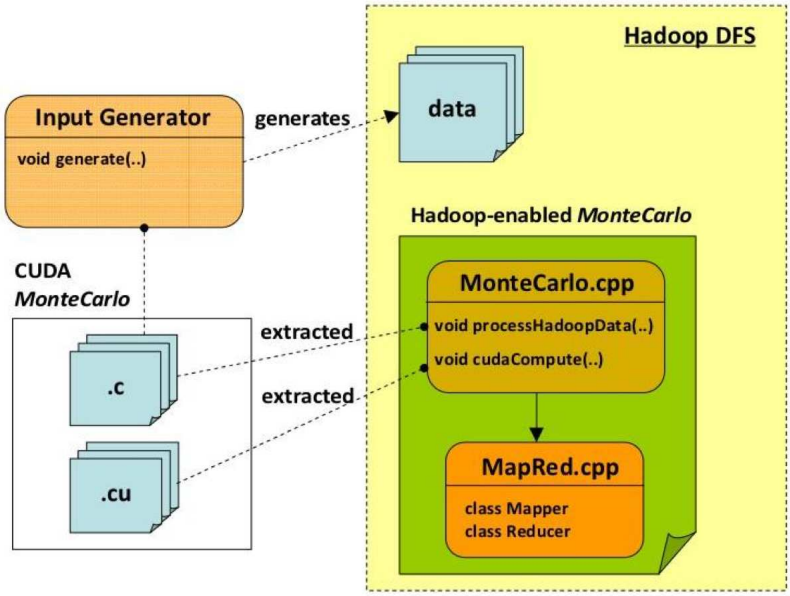

Figure 7. Porting CUDA to Hadoop from [3]

Figure 7 illustrates the porting of the MonteCarlo CUDA SDK example from CUDA to Hadoop. The data generation has been extracted in a new Input-Generator Class, which generates data and sends them to the DFS. A new Class MapRed.cpp has been developed in [3], which includes the following two abstract methods:

1. `processHadoopData()`
   The `processHadoopData()` has to be implemented, if the internal data structure of the CUDA program is different to the input data which come from the DFS.
2. `cudaCompute()`
   After executing `processHadoopData()` the MapRed class will invoke the `cudaCompute()` method. This will start the CUDA kernel. The results of the CUDA kernel application will be stored in the map object. These key-value pairs will be stored in DFS later.

5.5 CUDA Performance Analysis

Figure 8 shows the benefit of using GPGPU versus CPU. The introduction of GPU via CUDA into Hadoop delivers nearly a speedup of two in this example of a matrix multiplication with size of 1000. If the matrix size is increased to 1900 the speedup rises over three. This is due to the fact that when the input is small, the overhead from Hadoop dominates. [3]

	100x100	1000x1000	1900x1900
Execution Time CPU (sec)	120	161	520
Execution Time GPU (sec)	78	89	153

Table 2. Matrix Multiplication Execution Times

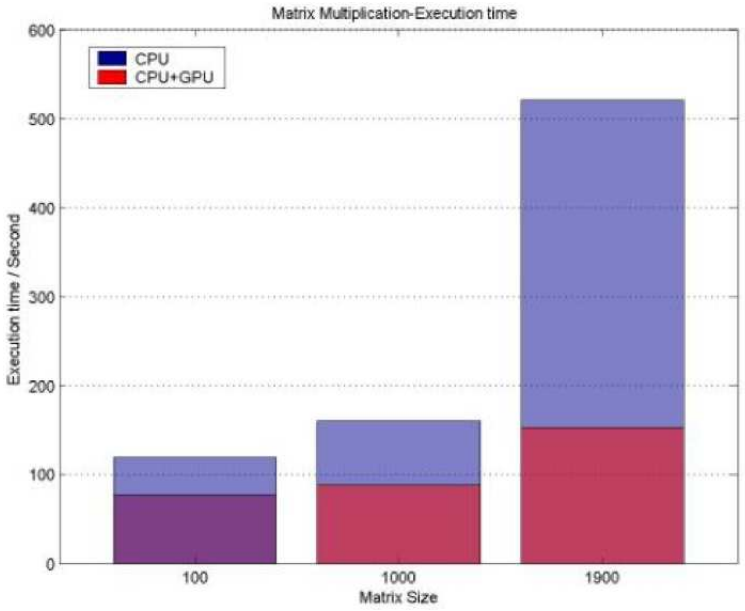

Figure 8. Matrix Multiplication Execution Times from [3]

6 Conclusion

This paper has presented the two most common options OpenCL and CUDA for using Hadoop with GPUs. The results of [1] and [3] have shown, that the usage of GPGPU in Apache Hadoop can improve the performance dramatically. In these two papers a simple task parallelism strategy was used, because it's the easiest way to implement CUDA or OpenCL without extending the Hadoop framework. If we would also use a more complicated hybrid task scheduling strategy as in [2] and a complex data management (HashTables) as in [11] and [12] the speedup would increase much more.
One big drawback of OpenCL is the missing datatype `string`. OpenCL only supports the datatype `char` and no additional string function. Every job which uses string input or output data in Hadoop e.g., evaluation of log files cannot be processed with OpenCL.
Another benefit of using GPGPU with Hadoop is the high cost-to-benefit ratio. It is cheaper to deploy GPUs on existing nodes than upgrading CPUs and adding more nodes to the cluster. Also the power consumption is lower when using GPU computing power. The substitution of the CPU with a GPU or the combination of both in Hadoop will lead to a better performance and efficiency.

References

1. Pieloth, C.: GPU-basierte Beschleunigung von MapReduce am Beispiel von OpenCL und Hadoop (2012)
2. Shirahata, K., Sato, H., Matsuoka, S.: Hybrid map task scheduling for gpu-based heterogeneous clusters. In: CloudCom, IEEE (2010) 733–740
3. He, C., Du, P.: CUDA Performance Study on Hadoop MapReduce Clusters (2010)
4. Tanenbaum, A.S.: Structured Computer Organization. 4th edn. Prentice Hall (1999)
5. Wikipedia: GeForce 600 Series — Wikipedia, The Free Encyclopedia, (2012) [Online; accessed 11-June-2012].
6. Guru3D: GeForce GTX 690 Review. http://www.guru3d.com/article/geforce-gtx-690-review/ (2012) [Online; accessed 11-June-2012].
7. NVIDIA: NVIDIA CUDA Programming Guide 4.2. (2012)
8. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Computing in Science and Engineering **12** (2010) 66–73
9. White, T.: Hadoop: The Definitive Guide. Second edn. O'Reilly (2009)
10. Wartala, R.: Hadoop: Zuverlässige, verteilte und skalierbare Big-Data-Anwendungen. professional reference. Open Source Press (2012)
11. Hong, C., Chen, D., Chen, W., Zheng, W., Lin, H.: MapCG: writing parallel program portable between CPU and GPU. In: Proceedings of the 19th international conference on Parallel architectures and compilation techniques. PACT '10, New York, NY, USA, ACM (2010) 217–226
12. Elteir, M., Lin, H., chun Feng, W., Scogland, T.: StreamMR: An Optimized MapReduce Framework for AMD GPUs. Parallel and Distributed Systems, International Conference on **0** (2011) 364–371