University of Edinburgh
School of Informatics

Cloud computing on Heterogeneous cores: MapReduce on GPUs

Undergraduate Masters Dissertation
Informatics

Karol Tomasz Pogonowski s0838671

April 3, 2013

Abstract: Cerberus, a MapReduce framework written in OpenCL and targeting GPGPUs is designed and implemented. MapReduce is a distributed programming framework that allows highly parallel processing on massive scale using commodity CPU hardware. Its success led to development of many different implementations with different targets, including a multi-CPU one called Phoenix. The transition to GPGPU processing for MapReduce tasks is discussed and after research on related work a novel library facilitating efficient MapReduce for GPGPUs is created on the basis of Phoenix. Unlike other solutions, it features a combined approach for memory access, supporting both dynamic atomic access, as well as a faster static, atomic-free approach without additional processing cost. The library outperforms or matches other CPU and GPU-based state-of-the-art implementations in four of six tasks, while underperforming in the rest two string-based applications.

Acknowledgements

I would like to thank my supervisor, Professor Michael O'Boyle, for guiding me throughout this project. Moreover, I would like to thank Ghoultown band for providing long-lasting inspiration with their music.

Contents

1. Introduction

High Performance Computing (HPC) is a specialist area that deals with analysis of massive data. In order to process such enormous sets and perform complex calculations computation clusters are used, coupled with fairly standard development techniques borrowed from other computer science methods. For a long time, the development of such applications was handled using imperative programming languages, such as C. They were mainly used because of wide adoption in other programming areas - and hence a wide support and recruitment base. However, there were some problems included in implementing HPC systems in C - mainly concerning synchronization. Imperative languages are not pure, eg. their functions can - and normally will - produce side-effects (that is, beside returning a value, the function either modifies some state or has some interaction outside its local scope), which alter the global state of the computation device [1]. Those side-effects are hard to emulate in a distributed and massively parallel environment and lead to either coherence problems or high-cost hardware remedies, which often have latency costs.

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions, which leads to lack of global or mutable state. It has been largely abandoned since the downfall of Lisp, however recent advancements in common languages like Haskell or Scala, as well as specialized ones like Mathematica or XSLT provide new insight into benefits of using this paradigm. In a pure functional language, methods cannot have any global state nor any side-effects. This promotes a different kind of reasoning about programming, but more importantly it allows flexible and scalable abstraction for massively parallel computations [2]. Since every computation is free of state and idempotent, an infinitely large array of such computation can be executed without synchronization problems. This makes it a robust candidate for parallel computing [3].

The MapReduce framework, originally conceived by Google in mid-2000s [4], is an important tool in HPC operating on computing clusters. It combines the idea of side-effect free functions with architecture built to support massively parallel applications. The framework operates on key-value pairs and builds upon two common functions of functional programming: map, which applies a single function to all elements in the collection independently; and reduce (also called fold), which combines all values with the same key and performs some accumulation operation on it (for instance, sum). This abstraction provides a robust, flexible, scalable and very efficient workflow, where user has to worry only about the implementation of those two functions (and, if needed, other optional ones such as sort or merge), whereas the MapReduce runtime will handle

the rest. The open-source Java implementation of MapReduce paradigm called Hadoop was created by Apache Foundation and is currently broadly used for Big Data computation [4]. Meantime, several local (running on a single machine) implementations of MapReduce were created, including Phoenix, Mars, MapCG and StreamMR.

In this dissertation we propose Cerberus, a new MapReduce framework aimed at running on GPGPU and written in OpenCL. It will be built upon the Phoenix MapReduce library and borrow ideas from other implementation to produce a robust and flexible heterogeneous computing platform.

1.1 MapReduce architecture

The MapReduce architecture takes its name from the main two components it comprises - map function and reduce function. Those two were originally developed for functional programming, however their lack of side-effects perfectly suited them for parallel, large scale computations and Google engineers appropriately exploited that potential. The map function operates on a single input unit and produces a key-value pair for each of them. The reduce function, also called fold, allows accumulation of results with same keys produced by map function. Together, those two simple functions can represent a large percentage of high-speed computation. Since there no side-effects, as many instances as needed of each function can be executed simultaneously. The basic workflow of a standard MapReduce task is as follows:

- The input is split to several buffers using some logical rules, eg. uniform split
- The map function processes each input unit and produces key-value pairs for it
- The partition function divides the map output as input for the reduce phase, normally sorting by the key, eg. each reduce tasks processes only pairs with unique key
- The reduce function folds results from multiple keys and produces a single key-value pair from that
- The merge function combines the results produced by reduce for final output

1.2 Phoenix MapReduce library

Phoenix MapReduce system was created at the end of 2000s, with the first whitepaper describing it published in February 2007 [5]. The initial goal was to develop a MapReduce library operating on a single machine in C language; then compare it to other paradigms, namely an ordinary sequential implementation, as well as one based on pthreads' thread-level parallelism. The original version didn't boast much performance gain over sequential code and was slower than pthreads. However, an update to the library brought performance on par with pthreads and severely overpowered the sequential code. The authors also released another, updated version of the library, written in C++ in 2011. This version was more modular and flexible, however due to the complexity of C++ it was decided to use the second version of the C library instead. Phoenix includes a number of interesting features for a CPU MapReduce, including own, custom scheduler, designed to work well with MapReduce, L2 cache optimization routines or memory access locator [5].

1.3 OpenCL

The OpenCL programming language was developed by the Khronos Group to achieve highly portable language for computations on heterogeneous cores, such as combination of CPU and GPU [7]. The standard was developed in full cooperation with the hardware manufacturers to ensure low-level, powerful and fast interface that would work on a variety of hardware. This is achieved using special OpenCL drivers, which translate the ordinary OpenCL code into hardware-specific assembly. The OpenCL code is compiled on the host device on-the-fly and the drivers are responsible for compatibility with the standard. OpenCL is heavily based on C99 and has a rudimentary, optimized function library. The obvious difference between OpenCL and standard C99 code is that there are no pointers in OpenCL - it is the only logical conclusion for that fact that most GPU units lack uniform address space (an exception is the new and future hardware, this is further detailed in the last chapter). What is more, there are certain features of OpenCL that make the development harder and much different than ordinary CPU code, or even GPU languages such as CUDA. For one, dynamic memory allocation is absent and all the memory used in the OpenCL kernels must be statically allocated by the calling application [7]. Secondly, only coarse-grained scheduling options are available (based on task parallelism), whereas the OpenCL runtime manages the resources dynamically. Thirdly, the provided library is very scarce and lacks fundamental C functions such as memset, strcpy or printf. Finally, debugging process is much harder, as the automated tools are still in early development and the error codes returned by the runtime are often

cryptic - moreover, sometimes instead of displaying an error code, they produce a hard-to-track segmentation fault at a seemingly random location. As we will see, despite those problems OpenCL can still be used to provide robust programming environment. In this dissertation we use the current version of OpenCL, which is 1.2 [8].

1.4 MapReduce on GPU

The architecture of GPUs makes them much better suited for Big Data MapReduce computations than CPUs. The main reason for that is that GPUs have always been designed to maximize the throughput of computations - in order to produce high frame rates for 3D graphics, a massively parallel system of texture and polygon memory, computing functions and intermediate buffers has always been required. Hence, the GPU design has three distinct characteristics:

- It possesses high number of independent computing units, each with dedicated, fast local memory
- It has a big and fast main memory bank, however access to it is normally not cached and in some cases slower than in CPUs
- The workflow is never divergent - all the data gets processed uniformly in a Single instruction multiple data (SIMD) manner

Those features are the key to efficient MapReduce computations [9]. First of all, the CPUs cannot execute many threads in parallel, while GPUs are able to a SIMD approach with a big number of computation units - hence, a large number of independent, parallel tasks. Secondly, the main memory of GPUs is much better suited for continuous, streaming operations on large buffers. The transfer rates for GPU memory are often few orders of magnitude faster than conventional CPU memory. It is especially true for coalesced reads and writes, which benefit from the addressing system of GPUs, allowing all the needed data to be fetched in one go. Thirdly, since GPUs lacks branch prediction hardware, divergent flow structures, such as if statements, severely degrade performance (for instance, with a simple if with a binary condition, the GPU runtime runs all the kernels and simply discards the ones, which fail the if check). This makes porting ordinary CPU applications very challenging - and the performance gains miniscule, especially if not enough care is taken to optimize memory operations. However, the MapReduce architecture explicitly guarantees absence of divergent flows in the kernels, sequential streaming access to data and large number of independent computations - and thus is perfectly suited for this platform. The progress in design of GPUs in recent years allows us to predict that GPUs will slowly integrate much of the general-purpose features of CPUs, making the development even

easier. To name few examples, the recent NVIDIA Fermi architecture introduced L2 caches, global unified memory address space and efficient atomic operations to all their computing products. This trend will only increase the relevance of GPGPUs in regards to HPC and MapReduce in the coming years. We will return to this topic in the final chapter.

2. Related work

Pertinent academic work is examined in this chapter. Firstly, the base for this dissertation and one of the first multi-processor implementations called Phoenix is described, followed by a newer GPU implementation Mars written in CUDA. In the end, a combined approach of CPU and GPU MapReduce presented by MapCG is debated, as well as AMD GPU-exclusive implementation called StreamMR.

2.1 Phoenix

The Phoenix MapReduce Library is a library implementing MapReduce model for multi-core and multi-processor systems [5]. It was originally developed in 2007 in C, however since an update was released, as well as a new version written in C++. The library allows for seamless invocation of the whole MapReduce cycle by simply filling the arguments struct and calling the main library function. It is possible to define various parameters for the library, such as declaration of functions - splitter, mapper, partitioner, reducer or merger, as well optimization-related ones such as L2 cache size, locator function to determine memory access pattern, number of map tasks, etc. The library itself is relatively lightweight; it is also statically linked with client application. Phoenix uses atomic operations to access the output queues and pthreads to spawn new tasks. This dissertation is based on Phoenix 2.0, the updated C version of the library.

2.2 Mars

Mars is a MapReduce library written in CUDA and C++ and runs on NVIDIA GPUs [9]. It is in many ways similar to Phoenix, however due to the change of architectural targets there are few major differences. Firstly, as a consequence of absence of dynamic memory allocation on GPUs Mars uses a two-stage system for each part of MapReduce tasks. The first part is called the counter and its purpose is to simulate the normal computation, but instead of writing to the output queue, increment a local output counter to indicate how much memory needs to be allocated for the second stage. After that, the intermediate buffers are created and the actual computation takes place. Secondly, Mars recognizes the performance degradation caused by using atomic operations, especially on GPUs, and proposes a different, static allocation system instead. It is linked to the aforementioned two-stage system - after the first stage, the library performs

a prefix sum over the local counter calculations of each thread. Since the writes to the output buffer can be statically analyzed by the data from prefix sum, no atomic counter is needed to govern the writes to the buffer. Also, this statically define memory accesses are easier to optimize in regards to coalesced memory reads on GPU, further increasing performance. Lastly, Mars allows tasks with just a map stage, while Phoenix performs identity reduce on such. Mars also boasts a CPU-emulated mode for performance measurement. It also features an extensive standard library, which includes optimized operations for strings, memory access, etc. According to the Mars paper [9], it boasts from 5 to 20 times speedup over Phoenix in certain tasks.

2.3 MapCG

MapCG is a recent development that aims to provide an abstraction layer on the level of MapReduce routines [10]. It features a custom made, C-like language that is later translated source-to-source either to an OpenMP implementation for CPU issue or a CUDA one for GPUs. Moreover, the authors created a custom-made memory allocator for both CPUs and GPUs, which rely on atomic operations, as well as a hash table to store the intermediate pairs. The hash table is a unique feature that allows skipping sort phase and increases the overall performance. Memory allocation for intermediate data is handled by creating a global pool of memory and then distributing writing blocks to each workgroup - this way, counting phase is unnecessary.

2.4 StreamMR

A relatively new and mysterious addition to MapReduce family, StreamMR is an OpenCL MapReduce library dedicated to AMD GPUs [11]. It's overall design is very similar to Mars, however the two-stage system is a bit different - in the first phase, the kernels write to user-allocated output. Once the output buffer is full, another kernel that is basically a counting kernel like Mars is executed on the rest of the input. Finally, new memory is allocated and another special kernel executes the unread input paths. The authors of the paper claim up to 30-fold performance improvement over Mars and Phoenix; however they have not released their source code and hence the claim cannot be properly evaluated. The released paper also compares an atomic implementation with the two-stage one and concludes that in every case the atomic-free implementation prevails.

3. Project goals

The main goal of this project is to prove GPGPU devices are well-suited for MapReduce tasks. The key to this goal is to create a robust OpenCL implementation of MapReduce library that is primarily targeted at GPUs. The first iteration of the project will be a proof-of-concept, not necessarily with top performance. Only when this initial implementation is proven to be viable, the next tasks of widening the target devices, optimizations, et cetera will be performed.

The secondary goal is to expand the supported hardware by APUs (such as Sony Cell Broadband or AMD Llano), DSPs and CPUs and hence allow heterogeneous computation using MapReduce framework and OpenCL. This will be a big challenge, once again mainly because of memory synchronization. As of now, there is no inbuilt feature in OpenCL that would allow for synchronization between two devices. It might be possible to use kernel events to build some of synchronization, much like proposed in StreamMR, yet it would surely be a very complicated routine, and probably wouldn't work as fast as theorized, creating a bottleneck.

A major aspect of project goals is to increase performance over vanilla Phoenix library, but also to properly compete with other GPU libraries such as Mars or StreamMR without sacrificing scalability. It is equally important to achieve good work balancing. This requires fair splitter and partitioner implementations that can dynamically assess whether workloads are balanced. It can also be beneficial to move the splitter and merge phases to GPU - although it is not clear if whether there would be any significant performance gain for either, as this is one of tasks poorly performed in GPUs.

Moreover, the library itself should be compact, compilable on variety of hardware and software, and the process of writing kernels as easy and straightforward as possible. The interface must be designed in a standardized manner and be robust.

Although performance is not a primary goal of this project, it is worth reasoning about hardware optimizations for specific devices should be deployed and the runtime should decide which optimization set is most suited for current hardware. As mentioned in the project proposal, this last part is an extensive study and could be a dissertation topic on its own. However, the estimation for completion time of mundane tasks implies there will be enough time to at least investigate some of the optimizations. For instance, some of GPU specific optimizations might be coalesced memory operations, which dramatically increases throughput of GPU memory. Another one could be exploiting the memory hierarchy of GPU, including using local and constant memories.

An important aspect of MapReduce, besides scalability, is its redundancy. Task

can fail often and the runtime must take care to ensure all the outputs are valid and well calculated. However, in a GPU setting the probability of single task failure is very low - and indeed would mean a physical damage to the card, such as memory corruption. This is so unlikely it will not be pursued as one of the systems goals.

The final goal, beyond the scope of this dissertation, is to release a working version of the library along with a test suite and publish it online. A whitepaper describing the system in depth would follow.

4. Implementation details

In this chapter, we will discuss the implementation strategy in detail. We will first start with the software engineering approach, followed by presentation of the general data flow in the library. Next, scheduling, memory and synchronization will be discussed, and finally an architectural comparison against other libraries will be shown and removed parts of the original library will be discussed.

4.1 The software engineering approach

The development of this application can be characterized as "agilefall", that is the broad planning component of the waterfall model (as in, specific times of milestone delivery, static development phase, etc.) are combined with short-term Agile methodology (incremental updates, small commits, short design-implement-test cycle). This approach proved successful so far in dealing with unforeseen circumstances, such as dynamic buffer sizes. Another feature of this system is that is starts with a working Phoenix library and incrementally builds on top of that. This way, it is simpler to reach a working stage for the OpenCL version, as well as it's easier to get rid of old code. Finally, a Mercurial DVCS is used to ease the development, keep online logs and backups and allow for easy and fast rollbacks in case of problems.

4.2 Data workflow

The workflow of the library is based on standard MapReduce one. The steps taken by the library can be summarized in those points:

- Library initialization (memory allocation, OpenCL initialization, setting of global parameters)
- Splitter is executed to determine the input data bounds for map tasks
- Map counter is executed to determine number of output key-value pairs (optional if using precomputed buffer size)
- Intermediate buffer is initialized with map count data
- Map tasks calculate their jobs and write to the intermediate memory
- Partitioner and optional sort is executed on the intermediate data

- Reduce counter is executed to determine number of output key-value pairs
- Intermediate reduce buffer is initialized with reduce count data
- Reduce takes place
- Merger combines all the tasks from different workgroups
- The data is read back to CPU memory and returned to the calling function

4.2.1 Kernels

There are two basic OpenCL kernels that are be specified by the user - map and reduce. For applications with non-static memory allocation schemes, another two are needed - map count and reduce count. If the reduce kernel is missing, the output buffers from map are simply copied over. The detailed function specifications are described below:

```
__kernel void map_count(__global const input_t* input,
        __global uint* output, uint data_size)
```

Simulates map task output on each input_t element of input. The result is a uint output number written to output. The data_size argument is used for determining the end of input data.

```
__kernel void map(__global const input_t* input,
        __global keyval_t* output, uint data_size)
```

Performs map task on each input_t element of input. Produces keyval_t tuples. The data_size argument is used for determining the end of input data.

```
__kernel void reduce_count(__global const keyval_t* input,
        __global uint* output, uint data_size)
```

Simulates reduce task output on each keyval_t tuple. The result is a uint output number written to output. The data_size argument is used for determining the end of input data.

```
__kernel void reduce(__global const keyval_t* input,
        __global keyval_t* output, uint data_size)
```

Performs reduce task on each keyval_t tuple. Produces accumulated keyval_t tuples. The data_size argument is used for determining the end of input data.

4.2.2 Splitter

The splitter function separates the input data into chunks of relevant size for each map input workgroup. Normally, it is uniform, ie. it divides the input data into equal parts using the number of workgroups and the data size. In some applications, it might be needed to ensure different split, for instance in word count the input is divided on whitespace. Such behaviour can be achieved using a simple splitter, which after parsing the input file will create a static array with each entry a char array of maximum word size length. The function signature of splitter is as follows (mr_env_t is a struct holding the internal state of the library):

```
void splitter (mr_env_t* env)
```

4.2.3 Partitioner

Partitioner is responsible for dividing the output data from map phase into suitable chunks and send it over to the reduce stage. There are different strategies available for unique applications. In order to ensure uniform task distribution, a simple fractional divide might suffice. However, with such partitioner it is not guaranteed that each reducer will only process one key, and as such additional care must be taken in writing reduce kernels, as well as writing a custom merger to combine the outputs. A traditional MapReduce strategy can also be taken, whereas partitioner divides the data in a matter that ensures each reduces processes only one unique key - yet this requires an expensive sort. The function signature for partitioner is shown below:

```
void partitioner(mr_env_t* env)
```

4.2.4 Merger

The merge function combines the output of all the reduce tasks in the CPU. It is basically a special version of the reduce function. In many applications, a merger is not needed. It depends on the way partitioner sends data to the reducers - if each reducer computes the output for a unique key, the merger only needs to copy all the results to the final buffer, without actually merging any key-value pairs. The function signature is as follows:

```
void merger(merger_dat_t* data)
```

4.3 Scheduling

As of now, no proper scheduling is taking place. This is because the OpenCL specification doesn't allow much freedom in this aspect - it only allows for task-level parallelism, using multiple NDRange kernel invocations. Possibly next iterations of OpenCL will add this functionality. On the other hand, it could be possible to emulate scheduling effects using available functions. For instance, we would like every map task to execute only single piece of data (instead of looping through all the data splitter gave to the particular map task) - to do that, we could use the global work size of clEnqueueNDRangeKernel to match the number of distinct map tasks and that would simply spawn that many kernel instances. However, it seems the global work size is platform-dependent and hence some data could not be easily enqueued this way (the problem with data bigger than device's memory is still not trivially solved).

4.3.1 Compute units workload

The GPGPU architecture is logically divided into several independent pieces called Compute Units (CU). Each of those is basically a processor core and is capable of executing one or more workgroups, where a workgroup consists of a fixed number of workitems. In order to exploit all the hardware resources of GPGPU, a specific, optimal number of workitems and workgroups must be set. In general, there should be more workgroups than there are CUs [12] and there should be maximum number of workitems in each of them. The task of determining most optimal combination is a not an easy one, as we will see in the experimental section, and it varies greatly between different kernels. Sometimes, it's required to increase the number of effective tasks (that is, number of workitems times number of workgroups) in order to ensure valid kernel behaviour - in some cases, too many memory accesses per kernel can produce unexpected results, such as segmentation fault or out of resources error. Since the number of maximum workitems is constant and usually already maxed out, the only way to do that is to increase the number of workgroups. Yet, this often causes decrease in performance, as more memory objects must be handled. This is especially pertinent to reduce tasks. This issue is further discussed in the results section.

4.4 Synchronization

Inherent in the design of MapReduce system is the ability to emit intermediate pairs. However, this task is not as easy as on CPU, where dynamic memory allocation allows easy implementation and high performance. In this section we

will discuss the steps taken in conceiving a robust solution that would fit both MapReduce's expectation and the reality of GPU hardware.

Firstly, a crude idea of buffers with a fixed size that should roughly accommodate the output data was created. It was based on the assumption that we cannot estimate the output buffer sizes without actually running the kernels (which is often true, as we will later see). However, there are at least two reasons against such implementation - the buffer might actually not hold all of the data, leading either to unexpected output or data lost; moreover even if it does, there is a possibility that the limited GPU memory space is wasted. A modification of algorithm that would alleviate the first problem would be to a kernel event once the memory object allocated was full, whereas the CPU code would listen for the event, and once it is received, save the data from the old buffer, then run a kernel calculating how much more space is needed (a count kernel discussed in more detail below), using this information create a new buffer and resume operations where stopped (this is the approach taken in StreamMR). After reasoning about possible implementation of such algorithm it became apparent that the code would be extremely complex and possibly inefficient, furthermore the problem of possible wasted space would not be fixed.

Thus, this idea was abandoned and another one followed, inspired by memory barriers and fences featured in OpenCL. The barrier would simply make all the kernels stop at the same point (writing to the output buffer), synchronize and then continue. This is quite straightforward approach that shouldn't reduce parallelism much, provided the flow of kernel is uniform, and it is a standard and well understood language feature. However, it soon became apparent that the OpenCL specification states that memory barriers and fences can only synchronize within a work-group. This had major implications - one could envision running a single work-group (a work-group can be run in parallel on several compute units or concurrently within one [7]), yet, modern GPUs often contain dozens of work-groups and hence using just one would severely reduce performance - and in the case of running the tasks across multiple GPUs, synchronization would be impossible. On the other hand, it would possible to divide the input into several tasks executed by different work-groups - given the side-effect free of MapReduce architecture. The only disadvantage could be additional merging required to combine the output of different reduce kernels, yet even this could be minimized by using specialized partitioner.

Furthermore, an architecture based on Mars could be used - namely, using a count kernel executing before the actual computation to calculate the size of output buffer. A count kernel executes the same instructions a normal one would, however instead of writing the results back, it increments a per-thread output counter. Afterwards, a prefix sum is computed on the resulting array and this data is then used to specify the place in output buffer for each thread stati-

cally. The big advantage of this approach is the lack of atomics, which can be a performance bottleneck, especially if they are located in the global memory. On the other hand, this approach requires a two-stage system, with redundant computations, which can often double the computation time.

At some point it became apparent that some applications have static memory access, ones that could be precomputed to calculate most optimal access pattern for both reads and writes. Firstly, it is often possible to determine the exact number of map output pairs based on number of tasks. For instance, in histogram each pixel of the input produces three key-value pairs, one for each RGB component. Hence, there are three times as map output key-value pairs as there are pixels in the input. Thus, if each map kernel must process four map tasks, there will be a total of twelve key-values. As for reads, simply using the number of tasks per map and the kernel index is needed to effectively identify, which part of input to read. This approach is significantly simpler and faster than any other presented, however it can only be used in some applications. It also allows for optimizations of reads and writes into coalesced accesses, which dramatically improves the kernel performance. There is also no synchronization step and hence this design does scale well.

In the end, it was decided that a different approach will be taken - a hybrid one combining static memory access and precomputed output buffer size where possible, with multiple workgroup dynamic buffers using OpenCL atomic operations on local counters in all other cases.

The input data is divided between the workgroups, each having a distinct input and output memory objects located in the global memory. In static cases, the buffer position is determined using precomputed data. For dynamic cases, each workgroup has a unique output counter, stored in local memory, which stores the next free part of the buffer and is updated using atomics. The memory writes to the global memory within a work-group are always combined - hence it is impossible to use global memory buffers for all the work-groups, as the output counters would deadlock and a distributed approach is required. The 32-bit atomic operations were an extension in OpenCL 1.1, but they made it to the main release in latest OpenCL 1.2 [8] (though 64-bit ones are still an extension). Thus, it can be safely assumed that future hardware will also support those operations. At the end of execution, the memory buffers are merged using a specialized merger function.

4.5 Memory

This part will discuss problems related to memory. There are several shortcomings in OpenCL specification, which require workarounds for efficient or reliable use. Firstly, accessing scalar data types is supposedly slower than accessing a vector4, as all memory accesses are addressed in a vector manner. However, in many applications this approach is either infeasible or increases complexity of kernel code - in histogram, the bitmap is 24-bit, represented as a unsigned char. In order to embrace vector data types, each histogram task would need to process a multiple of four tasks (least common multiple of three and four is twelve), retrieving all the data at once. It was decided that this issue is not as significant as presented (as some other papers proved [9]) and it would not be pursued in first iteration to keep the code complexity low. Secondly, OpenCL data types explicitly request 16-byte alignment. While this is not an issue with simple data types like ints, custom structs might get excessively padded, leading to wasted memory space. This is supposedly a technical issue related to the graphics pipeline architecture, which is optimized for vector4 texture memory operations. Thirdly, there is a limited number of memory operations a kernel can execute - if it exceeds it, an out of resources error is produced by the runtime. There is also a kernel-specific register boundary, which limits the maximum workgroup size. Thus, applications must reasonably assess the amount of resources they use.

Another issue exists for input that would normally be passed by pointers, as in Phoenix library. For example, the const char array that is passed for word count is not divided into different chunks of memory. Instead, a pointer and length is passed and the map function uses that data to find relevant chunk. However, OpenCL doesn't allow passing pointers - in fact, it is explicitly stated in the specification that even if it did, there is no guarantee that different kernels see the same part of memory as equal pointer. Hence, a statically allocated array must be used for handling such dynamic input - yet, such buffers must accommodate all the strings and also remain as saturated as possible. This leads not only to increased computation for splitter, which must first divide the data and then copy it to the static buffers, but also reduces performance as some of the buffers are not fully used and both space and access time is wasted. Moreover, due to the limit on number of memory operations, the applications operating on strings are very prone to returning errors, and to remedy the problem the input must be split among more map kernels, effectively increasing the workgroup count as the maximum workgroup size is also limited by hardware.

Finally, there will be some datasets that would not fully fit into device's memory. In order to calculate such amounts of information, the data and its execution would need to be divided. Since the design decisions led to isolated computation distributed between many workgroups, it is possible to easily extend this feature

to provide support for such enormous datasets. Moreover, the approach is also compatible with multi-device computation, which could be next step not only in increasing the raw computing power, but also to enable truly heterogeneous computation that would use different types of devices. This part is deemed not essential in the first stage of development and hence will be discussed as future work.

4.6 Computation heterogeneity

In order to produce a robust library, we must ensure each task is handled by the hardware most appropriate for it. By using such heterogeneous approach, we can achieve the highest resource utilization. In regards to that, Cerberus has been designed to use both CPU and GPU in its MapReduce task. For CPUs, the splitter, partitioner and merger functions are used. All those tasks are not well-suited for GPUs, as they are explicitly single-threaded, require access to internal library state and might combine some dynamically-allocated memory objects. On the other hand, the map and reduce functions are better run on GPU, as they are massively parallel, have few memory accesses and mostly computation. The sort function is an exception - currently, word count uses a standard single-threaded CPU qsort implementation to sort the intermediate data. There are fast and efficient parallel sort implementations that could be used in an OpenCL setting, however as only one of the test applications makes use of this feature, it was decided to postpone it until subsequent library versions.

4.7 Optimizations

There are few optimization patterns that were used in development of this library - both relating to hardware and software. The details of those are described below.

First of all, exploiting GPU memory architecture is very beneficial for optimal transfers. This includes not using too much private memory, as it would spill to global buffer and reduce performance significantly, as well as using local memory for storing data relevant to the whole workgroup. This is well visible in the histogram reduce kernel. The kernel needs a buffer with 768 values to store the intermediate colour count. However, storing such data structure independently for each reducer would be inefficient and probably spill to global memory. To combat this, a local memory structure, which holds the counts for all the workitems combined is used and is accessed using local atomic memory operations.

More importantly, in order to take advantage of the high-bandwidth streaming

GPU memory, the accesses must be coalesced - that is, consecutive kernels must access consecutive addresses of memory. This allows the scheduler to combine the multiple transactions into one and reduce the transfer latency. This method is used in almost all applications - for ones dealing with strings, the read accesses a lot of bytes for a single task already and hence the coalescing doesn't improve the performance. By extension, the GPUs are optimized for vector4 accesses in memory (as the operations are aligned on 16 bytes), due to the fact that textures are stored primarily in this manner. Hence, access using built-in vector4 types is faster than normal one. This optimization is often hard to implement effectively and hence it is prominently used only in one application, Similarity Score, which deals with operations on long vectors and hence can easily benefit from such addressing. Finally, some MapReduce-specific optimizations were used. For instance, the linear regression map kernel performs partial reduction of its output to reduce memory signature and computation stress on reduce phase. What is more, the default partitioner that is used in most applications doesn't divide the map output using unique keys - instead, it splits the buffer uniformly, ensuring equal computational load on all workgroups. This approach requires a merger function, yet as one is already compulsory for our design, this is not a problem.

4.8 Removed parts of the original library

Throughout the development of this library, parts of Phoenix were consecutively removed and rewritten to fit the new design structure. Although it seemed like most of the original code would be fit in the new setting, in the end the vast majority of original code was removed. The general function structure was retained, with a single function calling the entire library. The timing code is largely untouched, as are the initialization functions for the sample application, dealing with loading and preprocessing MapReduce data. The argument and environmental structs were reduced in size and altered to adapt to the new model.

4.9 Comparison with other libraries

In this part we will discuss the relative differences and similarities between our library and three others mentioned in the relevant work - Mars, MapCG and StreamMR.

The main division that can be made between those four is the method of accessing the output buffers. In Mars, a counting phase and prefix sum is used to statically find the boundaries for each thread. In MapCG, a custom memory allocator

is used in combination with atomics to ensure on-the-fly and reliable access. StreamMR, on the other hand, uses a user-allocated buffer size for the first phase of calculations, where the kernels write to the buffer in an atomic-free manner until it's full. Once that happens, a count kernel is executed on the rest of the data, new space allocated and the computation resumed from that point. Cerberus uses a system similar to Mars, however it realizes that many buffer sizes can be statically determined and that if the tasks are spread between kernels uniformly, no prefix sum is needed to determine the exact output location. For all the other data, Cerberus uses a dynamic approach based on output counters updated by local atomics.

Another distinctive feature is the programming language used in the libraries and hence the hardware supported. Mars is written using CUDA and C++, thus only being able to run on NVIDIA GPUs - and using an emulator, on CPUs. MapCG uses an intermediate language that is later translated either to OpenMP for CPU issue, or to CUDA for GPU one. This increases target hardware, however source-to-source translation is a difficult problem and cannot work as efficiently as intended, particularly for obscure kernels. StreamMR is written using OpenCL and C++, however it was targeted exclusively at AMD GPUs and hence can perform unexpectedly on other hardware. Cerberus is written in C and OpenCL and can be executed on any OpenCL-capable device, including CPUs, GPUs, ALUs and DSPs.

5. Experimental tests

Herein, we present the results of the experimental tests carried out to assess performance of Cerberus. The test suite is detailed in the beginning, followed by methodology and test setup, eventually reaching presentation of the experimental results.

5.1 Test suite

The application used in the test suite will be detailed in this part. There are six applications in total, five of which (histogram, matrix multiplication, linear regression, string match and word count) will be used for comparison with Phoenix, while other four (matrix multiplication, similarity score, string match and word count) will be examined versus Mars.

5.1.1 Histogram

Histogram is used to count frequency of specific pixel colours in an image. This technique is used in various applications, for instance in visual recognition. The map function reads one pixel, represented as three chars, from the 24-bit bitmap input file and outputs a (int, uint) keyval with colour index as key and count as value. The reduce function accumulates all the values from the input buffer in local memory and eventually emits all non-zero key-values. The merger function adds up the results from different workgroups to form a single 768-entry buffer. This application has a static access pattern. Although the output key has a maximum of 768 entries and thus would fit in a short, due to optimization reasons it is kept at int to help align the memory accesses.

5.1.2 Matrix Multiplication

Matrix multiplication is a binary operation that takes a pair of matrices, and produces another matrix. This rudimentary mathematical operation is used extremely widely, mostly for linear algebra calculations in fields like engineering, physics and computer science. It is also one of the exemplary applications for GPGPU computing, as it is highly parallelizable and has a low communication-computation ratio, making it perfect candidate for GPU computing. This application has only map phase, which operates on (int, int) pair as an input to

indicate where in the matrix the map kernel should start. The actual matrix is kept in auxiliary memory, which is a single buffer distributed across all the workgroups. This setting allows improving performance and simplifying splitter. After obtaining the coordinates of its chunk, the map kernel performs component-wise matrix multiplication and emits a single (uint, int) pair, where key is the index in the output matrix and value is the calculated number for that cell. The merger application extracts the index data and forms a uniform matrix as the output. This one also features a static access pattern.

5.1.3 Linear Regression

Linear Regression is a mathematical method for modeling the relationship between a scalar dependent variable y and one or more explanatory variables X. It is used in a wide variety of applications, for instance as a predictive model or to quantify the strength of relationship between two variables. The map function processes two chars from the input, x and y, and calculates the five output components. In order to improve performance and decrease memory traffic, the map function also does a partial reduction - instead of emitting key-values for every input, the function accumulates the data from all processed input and outputs a single set of five keyval pairs - each (int, long) pair has the parameter index as key and the sum as the value. The merger accumulates all the data into single five-pair output. It also has a static access data, and as above the key type is extended to help memory alignment.

5.1.4 Similarity Score

Similarity score is a mathematical method used to calculate the similarity of two documents, which are represented as a feature vector and thus is based on linear algebra. The test uses dot product and other methods to multiply the vectors of each of the documents. The input consists of a int2 vector that indicates which of the documents should be analyzed for current input. The actual matrix holding feature vectors is supplied as auxiliary data in order to improve performance. Only map phase is needed, which after calculating the score between two documents, emits a (float, int2) keyval, where the key is the result of the comparison and the value is an int2 holding the indices of the documents. This application features an optimization idea borrowed from Mars, namely using vector4 types to read the document vector. This supposedly increases memory bandwidth. The merger function simply copies the data back. As above, it features static memory accesses. This application is very resource-intensitive and running with small number of workgroups results in out of resources error. Hence,

this test suite uses 128 workgroups for small and medium tasks and 512 for large one.

5.1.5 String Match

String match is a popular performance tests, as well as being one of the most ubiquitous operation that is used on large datasets, most notably word text. It is also widely used in bioinformatics for genetic code comparison. The method of operation is straightforward - given some input text and control strings calculate how many times each of the control strings appears in the input text. Firstly, we need to discuss the problems associated with this application. In Phoenix, the string input data was kept as a single, contiguous memory block and the input tasks obtained pointers to specific chunks of memory they would need to process. However, such pointer passing is not supported by OpenCL and hence the input needed to be divided into static memory chunks, each one having a 128 char length buffer that was the maximum supported length of a single line. The splitter function cut the input into chunks at most 128 chars long, while keeping whole words intact. The map input takes a single 128 char line, and by iterating through it isolates the words and compares them with hardcoded match words. Note that these keywords used to be kept in either global constant store, or local memory one, however it became apparent that these accesses are necessarily much slower than access to hardcoded pre-processor value in each kernel. Once the comparison is done, a (int, int) pair is emitted, with key indicating which string was matched and value indicating how many times. The reduce function accumulates all the results in local memory and outputs final key-values (in the test case there were 4 control strings and hence 4 output pairs per workgroup). The merger combines all the pairs into unified output. This application cannot exploit static access pattern, hence it uses atomic counters to control the output writes.

5.1.6 Word Count

Word count is a popular application, often used in speech processing and statistical linguistics to obtain information about the language used, such as word frequency, number of unique words, etc. It's also one of the simplest applications, often used in rudimentary benchmarks. Word count isolates all the words in the input texts and counts each number of appearances. As in string match, the input data must be divided into 128 char length input lines by the specialized splitter. Afterwards, the map input takes a single 128 char line with individual words intact and by iterating over it, it isolates the words, and emits a single (word_t, uint) pair for each of the words, where the key is as 16 char type used

for storing words and the value is the appearance count. The reduce function proved to be very slowed, due to many string comparisons made and hence it was decided that reduce phase will be forfeit and the merger will handle the accumulation of the counts. As above, the word count kernels use dynamic memory access and hence are handled by atomic output counters.

5.2 Method overview

5.2.1 Setup

The test was executed on the GPGPU cluster on Edinburgh Compute and Data Facility (ECDF) servers. The server used was configured with these specifications:

- Intel Xeon Processor E5620 (2.4 GHz, 4 cores, 8 threads, 12M Cache)
- 24 GB RAM
- NVIDIA Tesla M2050 3GB VRAM
- Scientific Linux 6
- CUDA 4.2
- OpenCL 1.2

5.3 Results

The results of the experiments run on the test setup are shown below. Each column in the graphs shows a cumulative computation time for map (blue colour) and reduce (red colour) in milliseconds. The library initialization and finalization times were ignored as they were small and constant in every library. Each experiment used the best setting determined in first two tests. The data sizes for each experiment in small/medium/large order are respectively:

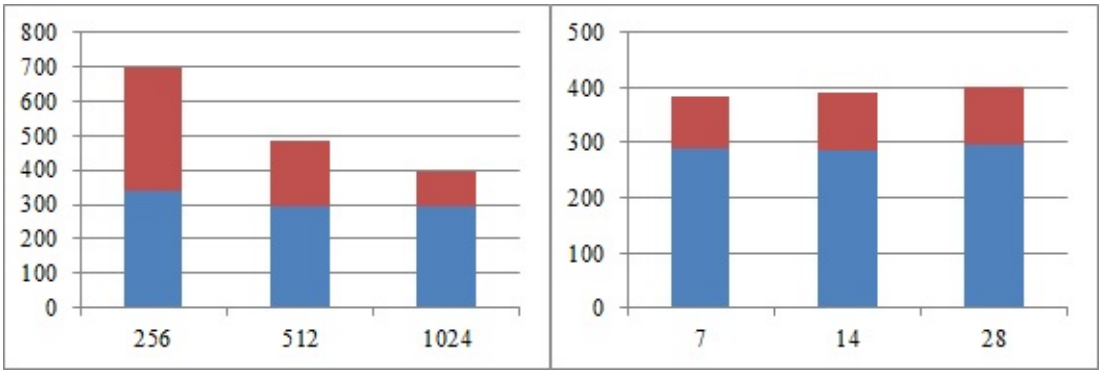
- Histogram - 50MB/100MB/200MB
- Linear Regression - 50MB/100MB/200MB
- Similarity Score - 512/1024/2048 documents, 128 vector size
- Matrix Multiplication - 512/1024/2048 side of the matrix
- String Match - 10MB/50MB/100MB
- Word Count - 10MB/50MB/100MB

5.3.1 Test 1: Number of workitems/workgroups

Firstly, we must find the optimal settings for number of workitems in a workgroup and number of workgroups. Those are mostly hardware specific settings; however some conjectures about the behaviour of the application can be deducted. For one, it is beneficial to use maximum number of workitems, as more threads help hide memory access latency. Moreover, number of workgroups should be at least the number of compute units a particular device has. This simply increases the occupancy of each of the compute units. However, some kernels are too resource-intensive and increasing number of workgroups could decrease performance in such setting.

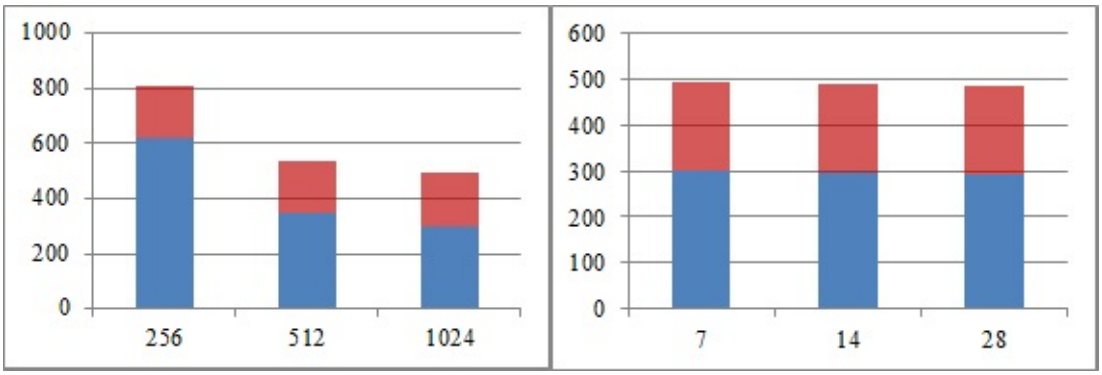
We use three tests to determine general optimum - Matrix Multiplication, Histogram and Word Count. There are three parameters tested for number of workitems - 256, 512 and the maximum supported by the device, 1024. As for number of workgroups, there are also three values - 7, 14 (the number of compute units of test card) and 28. All tests run on medium sized dataset. The results are shown below in Figure 5.1, 5.2 and 5.3 - the y axis shows computation time in milliseconds for map phase (blue colour) and reduce phase (red colour) combined. The x axis shows the particular setting used.

As shown in the tests, the number of workitems is consistent with hypotheses - all applications favour the largest setting. Curiously, for Word Count 512 setting comes ahead of 1024 by small margin. As for number of workgroups, the differences are small, yet in general seven seems to be the best setting.



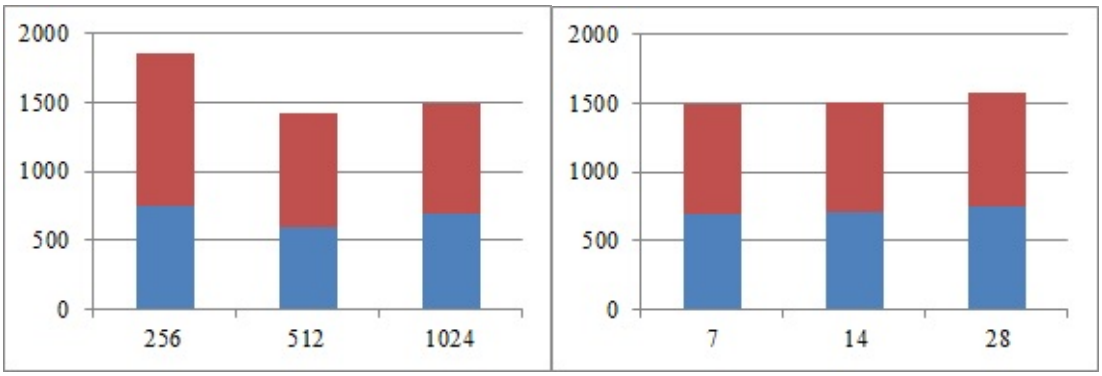
(a) Workitem variation (b) Workgroup variation

Figure 5.1: Test 1: Histogram



(a) Workitem variation (b) Workgroup variation

Figure 5.2: Test 1: Matrix Multiplication



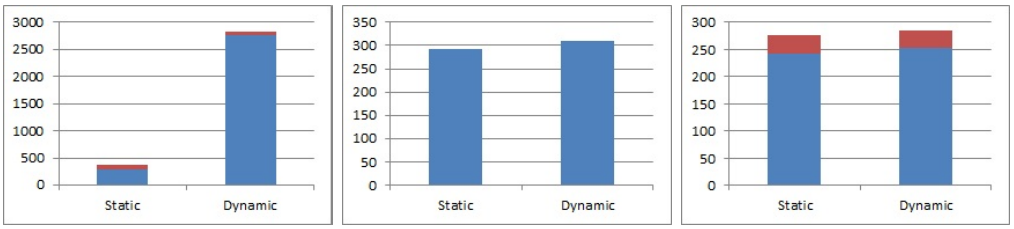
(a) Workitem variation (b) Workgroup variation

Figure 5.3: Test 1: Word Count

5.3.2 Test 2: Static and Dynamic memory access

The will now proceed to checking how much of a benefit static memory access is. We use three test suites - Histogram, Matrix Multiplication and Linear Regression running on medium data sets. One kernel version uses static writes, while the other uses dynamic, atomic-based ones. As before, the map is coloured blue, while reduce is red, the x axis shows the computation time in milliseconds and the y axis shows the setting used.

As seen in Figure 5.4 all application benefit from static access. The performance is almost 10 times increased for histogram, while Matrix Multiplication boasts a modest 10 percent increase, similarly to Linear Regression.



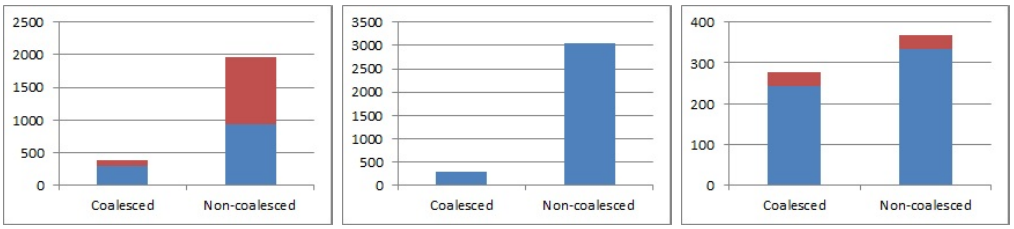
(a) Histogram (b) Matrix Multiplication (c) Linear Regression

Figure 5.4: Test 2: Static and Dynamic memory access

5.3.3 Test 3: Coalesced and Non-coalesced reads and writes

In the third test we check how much of an impact coalesced accesses have. Same tests as above are used here - Histogram, Matrix Multiplication and Linear Regression, running on medium data sets. The first kernel has both writes and reads coalesced, while the other lacks both. Again, maps are coloured blue, reduces red, the x axis shows computation time in milliseconds and the y axis the settings used.

Figure 5.5 clearly shows that coalesced accesses provide huge benefit. For Histogram, the difference is five-fold, while for Matrix Multiplication it's as high as ten-fold. Linear Regression shows lower increase of about 30 percent.



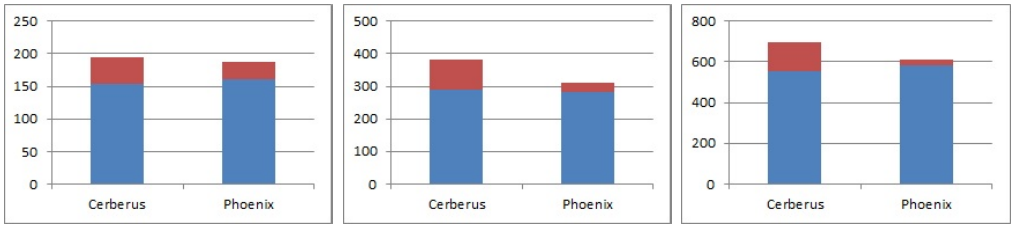
(a) Histogram (b) Matrix Multiplication (c) Linear Regression

Figure 5.5: Test 3: Coalesced and Non-coalesced reads and writes

5.3.4 Test 4: Phoenix

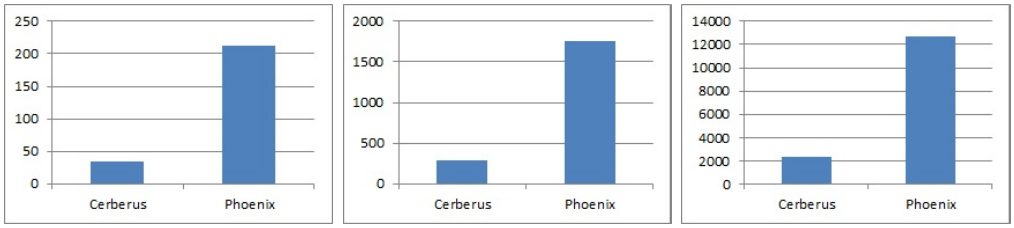
In this part we present results from comparing the library against Phoenix. Five tests are used for this comparison - Histogram, Matrix Multiplication, Linear Regression, String Match and Word Count. Each test runs three data sets of different size, detailed at the beginning of this chapter. Map phase is coloured blue, reduce phase is red, the x axis shows the computation time in milliseconds and the y axis shows the library used.

Figure 5.6 shows the Histogram results, which are very similar to each other. In Figure 5.7, Cerberus is in a clear lead in Matrix Multiplication - approximately 6-fold advantage over Phoenix. For Linear Regression, we can also notice lead of Cerberus, especially with bigger data sets and approaching 2-fold increase in Figure 5.8. Word Count and String Match are a clear win for Phoenix - in Figure 5.9 String Match Phoenix is 6 times faster, while in Word Count in Figure 5.10 it's as high as 10 times faster.



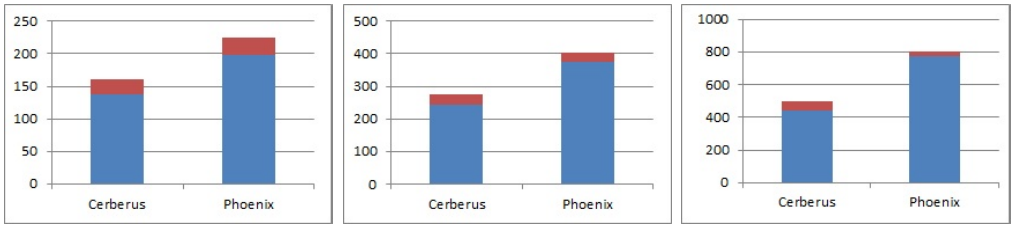
(a) Small (b) Medium (c) Large

Figure 5.6: Test 4: Histogram



(a) Small (b) Medium (c) Large

Figure 5.7: Test 4: Matrix Multiplication

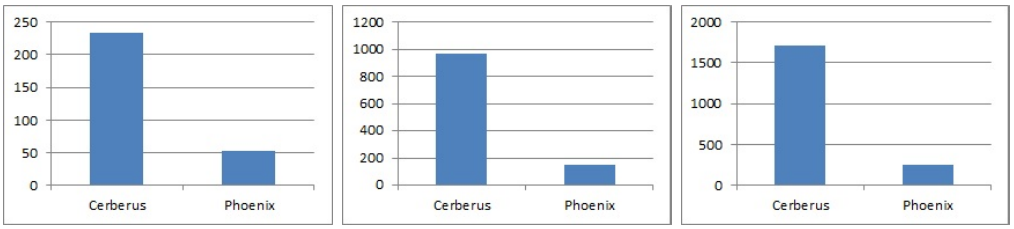


(a) Small (b) Medium (c) Large

Figure 5.8: Test 4: Linear regression

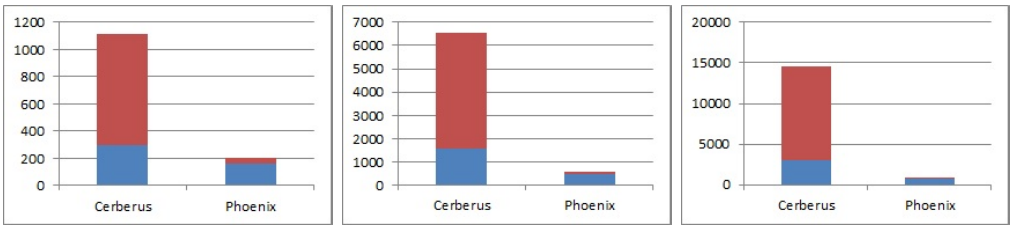
5.3.5 Test 5: Mars

Finally, we test the library versus Mars. There are four applications used - Matrix Multiplication, Similarity Score, String Match and Word Count. Each run three



(a) Small (b) Medium (c) Large

Figure 5.9: Test 4: String Match

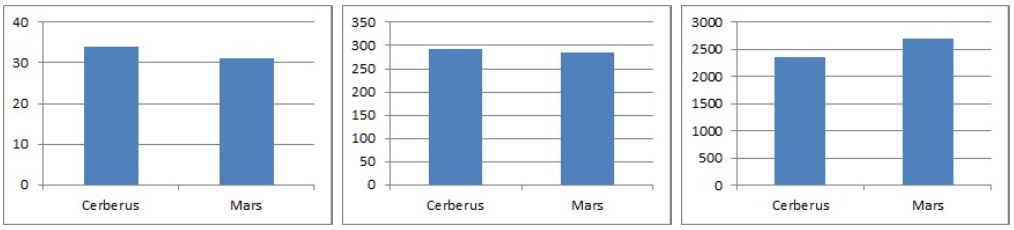


(a) Small (b) Medium (c) Large

Figure 5.10: Test 4: Word Count

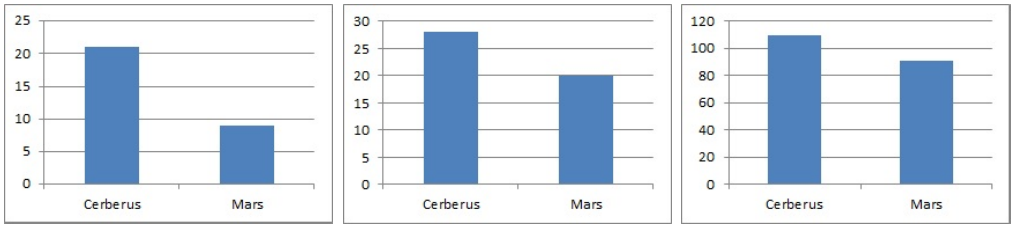
differently sized datasets. Since Mars doesn't support multiple keywords for String Match, a different dataset designed for single-word search is used instead. As before, maps are coloured blue, reduce is red, the x axis shows the computation time in milliseconds and the y axis displays the library used.

The calculation-intensive tasks perform similarly on Cerberus and Mars. Figure 5.11 highlights slight advantage of Mars in small and medium datasets, as well as a bigger increase in large datasets for Cerberus over Mars. In Figure 5.12 Similarity Score performs about 20 percent slower on Cerberus. For string-based tasks, Mars is much faster than Cerberus. String Match in Figure 5.13 is about 6 times faster, while Word Count in Figure 5.14 approximately 3 times slower.



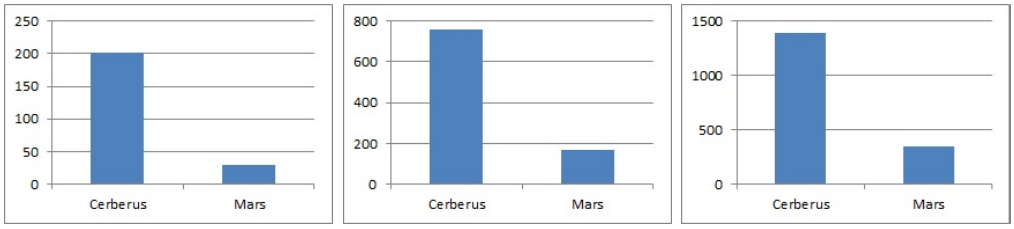
(a) Small (b) Medium (c) Large

Figure 5.11: Test 5: Matrix Multiplication



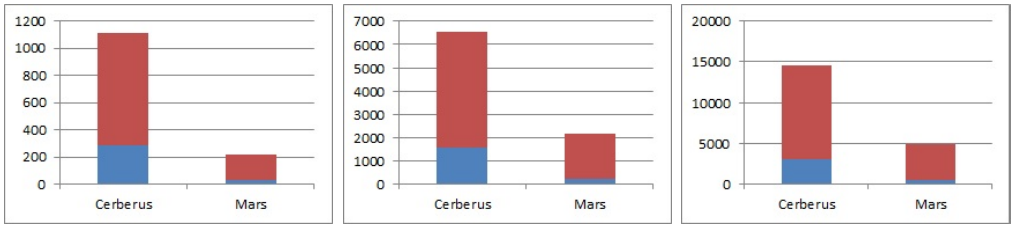
(a) Small (b) Medium (c) Large

Figure 5.12: Test 5: Similarity Score



(a) Small (b) Medium (c) Large

Figure 5.13: Test 5: String Match



(a) Small (b) Medium (c) Large

Figure 5.14: Test 5: Word Count

6. Discussion

Firstly, let's look at the results from Test 1. Figure 5.1, 5.2 and 5.3 show that maximum number of workitems within a workgroup is always the best choice, besides Word Count where the difference between optimal 512 and 1024 is very small. The difference is as much as two-fold between 256 and 1024 workitems. This is not surprising, considering the massively parallel architecture of GPUs, as more threads can utilize the resources of a Compute Unit more efficiently. Moreover, higher number of threads helps to mask blocking latency introduced by memory accesses. For instance, if a single thread accesses the main memory, it blocks the execution until the operand has been obtained, which often takes hundreds of cycles. Instead of waiting, the scheduler interleaves execution with another thread, effectively masking the stall. We also see that all applications favour lowest number of workgroups. In a typical case, the number of workgroups should be higher than number of compute units on a device. However, here we see that half of number of compute units is the optimal setting. This implies the resources of GPU are well-utilized with a smaller than usual number of workgroups. The minimal differences between the settings can be attributed to time needed to initialize buffers for additional workgroups. Thus, we can conclude with the optimal setting that is used throughout the rest of experiments - 1024 workitems and 7 workgroups.

We shall examine differences between static and dynamic memory accesses next. As seen in Figure 5.4, Histogram benefits greatly from static access. The differences is especially pronounced in the map kernel, as the reduce kernels have low amount of memory transfers. The map kernel, on the other hand, is completely memory-bound, as there is no particular computation done on the input data and three intermediate pairs are produced for each pixel. Figure 5.4 shows a modest increase of several percent for Matrix Multiplication. This is probably caused the small amount of output produced by each thread and the high amount of computations required. In other words, Matrix Multiplication is compute-bound and hence benefits from faster memory accesses only slightly. This is also true for Linear Regression, which uses a partial reduction technique in the map phase to reduce the number of intermediate pairs, and hence the amount of memory transfers (Figure 5.4). LR has a small 5 percent increase in map phase and no gains in reduce phase, as it is also not bound by number of memory transfers. Hence, the static accesses provide a massive advantage in memory-bound applications with high number of output per thread, while providing only slight benefit for others.

Thirdly, the coalescing of memory accesses needs to be examined. In Figure 5.5, we observe a three-fold increase in performance for Histogram map and over ten-fold for reduce stage. This finding is consistent with the hypothesis

concerning coalesced writes requiring less overall memory operations. The results are especially interesting for reduce stage, as it uses a local accumulate store for output and hence does not benefit from coalesced writes - the gain is purely from coalesced reads. It seems because of this lack of writes the kernel is able to utilized coalesced reads in a much better fashion than in standard read/write kernels and hence the gains. The effect is even more pronounced for Matrix Multiplication, with over 10-fold performance increase (Figure 5.5). It is clearly visible that using coalesced access the cache is utilized more efficiently. This is not a standard result, as the Matrix Multiplication example uses a different way of splitting the input - since all the tasks must have access to the whole source matrices, they cannot be split in a normal manner. Instead, the input is fed with coordinate pairs for multiplication, which are then used to find corresponding entries provided in an auxiliary matrix in a global memory object constant across the workgroups. Hence, this performance gain must be attributed to specific worksize of the task and device specification, which result in an optimal pattern access for this specific Fermi cache architecture. It's also possible that the multi-broadcast feature is used to convey the read-only matrix entries to all workgroups at once. As for Linear Regression, the gains are smaller but still significant - a 50 percent performance increase for map and no change for reduce (again, due to low amount of memory transfer). The gains shown in Figure 5.5 can also be explained by assuming the memory accesses were combined across the threads. In summary, memory coalescing provides a huge benefit for all applications.

The tests results with other libraries will now be evaluated. First of all, we will discuss comparison against Phoenix. Figure 5.6 shows the results are very similar for Histogram. Phoenix is approximately 15 percent faster across the data sizes. Reduce stage times are especially in favour of Phoenix, probably due to the fact that in Cerberus reduce involves a high number of operations on local memory to keep the counts of individual pixels. This application is also memory-bound and hence might perform better in a well-cached CPU environment. In Matrix Multiplication Cerberus shows its clear superiority - showing a consistent six-fold increase over Phoenix (Figure 5.7). This application is more computationally-intensive than Histogram and hence the higher ALU count of GPUs can prove its worth here.For Linear Regression Cerberus comes with a moderate lead. It's performance gains are especially pronounced at large data size, with over 60 percent increase, followed by 40 percent for medium and small ones (Figure 5.8). As in Matrix Multiplication, the memory accesses are scarce, while the computation is even more complex, resulting in a clear win of parallelized GPU architecture. It is clearly visible in the map stage, which performs additional calculations using partial reduction in order to reduce the number of intermediate pairs. The last two applications show much worse performance from Cerberus due to computation on strings, which prove to be inefficient in Cerberus. As for String Match, Figure 5.9 shows over five-fold gain for Phoenix. For Word Count, Phoenix is at least

twice as fast in map stage, and few orders of magnitude faster in reduce phase (Figure 5.10). Note that the standard approach with GPU-based reduce proved to be problematic for this application, as the reduce kernels needed to perform too many string comparison operations, which resulted in either horrible performance or crashes. Hence it was decided that a CPU merger will be more suitable for the task and used instead - the reduce time for Word Count then shows the CPU-based merger time. Additionally, it was determined what the performance bottleneck is and it proved to be string compare function. Those results point out to inefficient use of strings in Cerberus and need to provide a more robust solution, perhaps based on a similar string library optimized in Mars. As we can see, Cerberus proves generally faster than Phoenix for computational-intensitive tasks, sometimes boasting an order of magnitude performance increase. Yet, those based on string manipulation are severely underperforming and show clear benefit of CPU-based architectures.

Finally, let's discuss the results versus Mars. Figure 5.11 shows the results for Matrix Multiplication. As we can see, both application exhibit similar performance in this task. Mars is marginally faster for small and medium dataset, while with large one Cerberus takes a lead by about 15 percent. This implies both applications are well-written and make optimal use of GPU resources, hence the small difference. As for large dataset, it is possible that Cerberus' implementation is more cache friendly, allowing the pipeline to be more saturated than for Mars. In Similarity Score we also notice both libraries perform alike. Figure 5.12 shows the relatively high differences for small and medium datasets, yet those are probably skewed by the difference between OpenCL and CUDA runtime issue latency. For large dataset, the difference is much smaller, about 25 percent in favour of Mars. This also shows Cerberus is capable of optimally handling resources for computationally-intensive tasks. Once again, Cerberus is outperformed in string library, as Mars has a GPU-optimized library for standard string functions. Figure 5.14 shows a performance gain similar to Word Count, with Mars being approximately 5 times faster in String Match. Note that Mars is slower than Phoenix in this task, even though it matches against only one keyword, not four like Phoenix application. In Word Count Mars is on average three times faster than Cerberus (Figure 5.13; interestingly, Mars seems to be much slower in regards to Phoenix in this task). Again, bote how much slower the reduce phase is in regards to Phoenix - this is the consequence of architectural differences in GPU, which make handling strings less efficient. As shown, the performance of Cerberus is generally on par with Mars for computationally-intensive tasks (with a difference at most 20 percent, consistent with hypothesis shown in [13]), while it is easily outperformed in string-based ones.

7. Conclusion

To sum up, we started with description of the three technologies that form the basis of this dissertation - GPGPU computing, MapReduce framework and OpenCL programming language. We detailed each of them, as well as providing foundation for further developments. Finally, we presented Cerberus as combination of all three of them. Afterwards, the relevant work was discussed in detail, followed by statement of the project goals. In the next chapter we deliberate on the practical side of the project - the implementation. All the relevant information about the lengthy process of writing the actual library were displayed - including the software engineering approach, the data workflow, the kernels and helper functions. Moreover, we glanced over more specific problems associated with MapReduce on GPU - namely ones connected to synchronization, scheduling, memory access, as well as the heterogeneity of the library, the optimizations used and eventually a comparison against other libraries. Following, we described the experimental setup and methodology, as well as presenting the test suite in details. We also discussed the five tests used to determine both the prowess of the finished library against competitors, in addition to determining the best parameters and memory access models to be used in the experiments. In the end, we discussed the results in detail and formed this conclusion, followed by a chapter dealing with future work on Cerberus.

Overall, this research is considered successful, as not only it proved the GPGPU implementation possible, but also presented viable optimization and design methods, which allowed Cerberus to roughly match - and indeed sometimes even outmatch - the performance of state-of-the-art implementations, at least for most applications. For string based applications it became apparent that more development is needed, especially in terms of GPU-optimized standard libraries, like string compare or memory set. The significance of those results cannot be underestimated - in the digital world, where not only the datasets grows exponentially, but also hardware technology is routinely challenged by novel ideas, tapping the power of massively parallel GPGPUs is the only logical conclusion for dealing with Big Data. By displaying that Big Data computation is not only feasible, but also relatively straightforward and very efficient, we set the bright path for future generations of GPGPU developers.

8. Future work

In this final chapter we explore ideas for future developments in this research area.

8.1 GPU merge and sort

It could be beneficial to run a GPU-based merger kernel instead of a current CPU design. It was decided that CPU merger would have be easier to develop and debug, moreover the advantage of GPUs in this particular task seem elusive. Firstly, merging requires operations on even bigger buffers, which might lead to out of resources errors in OpenCL, especially for computationally-intensive applications. Moreover, in merging tasks it would be difficult to effectively exploit GPU memory hierarchy. Thus, the sequential, cached and branch-predicted architecture of CPUs is more specialized for this task. On the other hand, with the improvements in integration of CPU and GPU code, such as support for cache and virtual memory discussed above, could increase the fitness of GPU processing for merging phase.

Moreover, an efficient parallel implementation of sort, such as parallel radix sort, running on GPU could be used to speed up processing. A sorter is normally part of the MapReduce implementation, however the test suites used for the experiments didn't require it (besides Word Count), and since it is not trivial to implement such algorithm it was left as future work.

8.2 Hash tables

We could use a hash table to store the intermediate data, much like the approach taken in MapCG [10]. Classically, hash functions allow average-case constant time access, at a cost of additional computation to find the hash key. Since GPUs are abundant in ALUs and the data sizes are big, this does not create a bottleneck. Such data structure combined with static memory access pattern could prove very fast and applicable. Moreover, it could be beneficial to hash strings used in the string-based tasks, such as Word Count. Using a good hashing function could significantly reduce the current biggest bottleneck, the string compare function.

8.3 Intermediate language for OpenCL kernels

OpenCL is a complex language, building upon C99, which is extensive in itself. This complexity increases the learning curve and results in fewer amounts of programmers, especially in the scientific community, being able to develop robust applications. In this setting it might be beneficial to create a standard set of templates or macros, such as ones pertaining to different memory access techniques discussed in previous sections that would ease the typical development process. Moreover, it could be even more flexible to allow using an intermediate language, possibly based on some dynamically-typed one such as Python, which would encapsulate all the required OpenCL capability in a clearer and simpler manner. There are already some open-source implementations that feature source-to-source translation from Python to OpenCL, even more there is also a multitude of wrappers that could be used for this task (such as PyOpenCL [16]).

8.4 Multiple devices

As discussed in previous sections, adding support for multiple devices, some of which might be heterogeneous, might be a pertinent extension of the library. Many HPC centres already use multi-GPU racks for complex computation, and by supporting such configuration the library would gain much scalability. This addition would be rather simple, as the biggest problem - split memory buffers - is already utilized.

8.5 Support for big data

Datasets that exceed the RAM size of the GPU will be increasingly common, especially among HPC users. To accommodate this problem, the task must be divided into several disjoint stages, with a function similar to merger that would consolidate the output. Such configuration could also be used for near real-time purposes, such as interactive speech recognition. Once again, the split memory buffer design allows for relatively easy implementation of this feature in the future.

8.6 Integration with Hadoop

One of the key advantages of using Hadoop is the support of Streaming, which allows users to run custom functions in any language, provided they can be executed through a shell script. Thus, one can specify their own methods in their

favourite programming language that handles IO through stdin/stdout channels, and have Hadoop handle all the problems associated with execution on massive scale. It is possible to further augment this feature - instead of running custom CPU-based functions, run it on Cerberus using a simple wrapper in a shell script. This way we combine the distributed computation framework running on CPUs, with specialized individual computing units running on GPUs, with truly minimal effort. Such integration could prove very beneficial to computation of enormous amounts of data over distributed, GPU-assisted computing clusters.

8.7 Future GPU architectures

Computer hardware is widely known for its fast pace of evolution and improvement, related to the famous Moore's Law. Today's state-of-the-art hardware is soon to be replaced by a new, more robust, efficient and integrated replacement. This process must be accounted for in the development of software, especially if performance is one of the key metrics. There are already rumours of future improvements in GPU architecture that would prove beneficial for GPGPU computation and MapReduce tasks in particular. We will discuss the advancements planned in roadmaps for two biggest GPU producers, NVIDIA and AMD.

For NVIDIA, the previous generation of GPU cards (Fermi) already contains L2 caches, which greatly increases performance of random memory accesses [14]. What is more, another extension, particularly important for scientific computing, is the inclusion of 64-bit floating point operations. Since such double-precision calculations are often the key of scientific calculations, such as gravitational simulations or finite element analysis, this move greatly expanded the user base of GPGPUs. The currently produced cards, Kepler, provide a feature not yet seen in GPGPU computing - dynamic parallelism, which allows the kernels run on GPU to spawn other kernels from within their execution scope. This finally allows recursive functions, as well as having control kernels, which handle other ones (similar to Power Processor Element found in IBM Cell CPUs). Future architectures, including Volta, will support virtual memory and allow for seamless data movement between CPUs and GPUs, as well as providing a single, unified address space that will allow pointers to any part of the memory - a truly heterogeneous design. The inclusion of branch prediction and new, more software-based instruction scheduling engines might improve performance of some of the kernels. Moreover, the plans for integration of small amount of very fast memory directly on the GPU die, in a manner similar to caches, would result in massive improvement for memory accesses on small blocks. Finally, development of bigger and faster video memory, such as stacked DRAM, will allow for arbitrary size datasets to be efficiently handled by the GPUs. Another curious idea, which more likely

will be revolutionary for HPC settings, is the support for virtualization of GPU resources in the upcoming NVIDIA Grid product line. Combined with Cerberus running on Hadoop, such setting might introduce a new era of GPGPU-powered massive scale cloud computing.

As for AMD, generally similar directions are taken in development of next-gen cards [15]. One big difference is that AMD also owns a successful CPU architecture and as such aims at providing an integrated heterogeneous solution, combining CPU, GPU and memory controller on single die. This technology, called Accelerated Processing Unit (APU), is already utilized on AMD Jaguar platform. In this platform, AMD has forfeit its long-lasting design principle of Very large instruction word (VLIW) Multiple instruction, multiple data (MIMD), instead replacing it with a Reduced instruction set computing (RISC) Single instruction, multiple data (SIMD) one, much more similar to those used in design of CPUs and NVIDIA GPUs. Moreover, the GPU and CPU use the same system bus, as well as sharing the random access memory. This allows for cheaper data transition between the two. Furthermore, the upcoming Heterogeneous System Architecture (HSA) will bring even more integration to the market. Firstly, it will feature a unified memory space for CPUs and GPUs that will provide virtual memory support. This is similar to NVIDIA's plans, however AMD takes it a step further - the chips will feature a single memory management unit (MMU), as well as mutually accessible caches. Those revolutionary ideas will surely dramatically improve performance of heterogeneous tasks. Moreover, the future chips are also likely to support pre-emptive context switching, as well as Quality-of-Service implementation that will be able to prioritize certain tasks. Virtualization technology will also be supported in the Radeon Sky.

Although many of these features are not directly usable in a MapReduce setting, the transition to a more unified model of heterogeneous computing will surely benefit the performance, as well as critically reducing the implementation gap between GPU and CPU. What is more, it will increase the user and support bases, allowing more relevant programs to be developed.

Bibliography

[1] D. Turner, *CSc 520 Principles of Programming Languages*. Department of Computer Science, University of Arizona 2005.

[2] C. Collberg, *Research Topics in Functional Programming*. Addison Wesley, Massachusetts, 1990.

[3] J. Hughes, *Why Functional Programming Matters*. University of Glasgow, 1990.

[4] J. Dean and J. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*. In the Proceedings of the 6th Symp. on Operating Systems Design and Implementation, Dec. 2004.

[5] Apache, *What Is Apache Hadoop?*. http://hadoop.apache.org, 2007. Retrieved 02-04-2013.

[6] C. Ranger et al., *Evaluating MapReduce for multi-core and multiprocessor systems*. In HPCA 07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, 2007.

[7] The Khronos Group., *OpenCL 1.0 Specification*. www.khronos.org/registry/cl/specs/opencl-1.0.pdf, December 2008. Retrieved 02-04-2013.

[8] The Khronos Group., *OpenCL 1.2 Specification*. www.khronos.org/registry/cl/specs/opencl-1.2.pdf, November 2011. Retrieved 02-04-2013.

[9] W. Fang et al., *Mars: Accelerating MapReduce with Graphics Processors*. IEEE Transactions on Parallel and Distributed Systems, 2010.

[10] C. Hong, et al., *MapCG: writing parallel program portable between CPU and GPU*. Proceedings of the 19th international conference on Parallel architectures and compilation techniques, 2010.

[11] M. Elteir et al., *StreamMR: An Optimized MapReduce Framework for AMD GPUs*. 17th IEEE International Conference on Parallel and Distributed Systems, 2011.

[12] B. Gaster et al., *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition*. 2nd edition, Morgan Kaufmann, Massachusetts, 2012.

[13] J. Fang et al., *A Comprehensive Performance Comparison of CUDA and OpenCL*. In proceeding of: International Conference on Parallel Processing, September 2011.

[14] J.H. Huang, *NVIDIA Presentation*. GPU Technology Conference 2013, San Jose, California.

[15] J. Sanders, *AMD Presentation*. GPU Technology Conference 2013, San Jose, California.

[16] A. Kloeckner, *PyOpenCL - Python OpenCL Wrapper*. http://mathema.tician.de/software/pyopencl. Retrieved 02-04-2013.