Algorithm Construction for GPGPU

SIMON HÖSSJER
and MATTIAS SVANSTRÖM

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2012

Algorithm Construction for GPGPU

SIMON HÖSSJER
and MATTIAS SVANSTRÖM

DD143X, Bachelor's Thesis in Computer Science (15 ECTS credits)
Degree Progr. in Computer Science and Engineering 300 credits
Royal Institute of Technology year 2012
Supervisor at CSC was Mårten Björkman
Examiner was Mårten Björkman

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2012/
hossjer_simon_OCH_svanstrom_mattias_K12038.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Abstract

Today every personal computer and almost every work-related computer has a GPU powerful enough to be used as a supplementary computational device. One framework which enables utilization of this is called OpenCL. We asked the question how one writes efficient algorithms on these GPGPU devices. We found that there are two major ways to run code concurrently, data parallel and task parallel. The runtime of an algorithm depends on the amount of the code that can be run in parallel rather than on the number of processing elements available on the device. We decided to test the theory by implementing parallel algorithms for matrix multiplications and integer sorting with radix sort. The results of the tests can be summarized as both good and bad. Even though there is a promising gain in performance there are also factors like memory accessing which is not always easy to control.

Sammanfattning

Idag har alla persondatorer och nästan alla arbetsrelaterade datorer en GPU kraftig nog att utnyttjas som en extra beräkningsenhet. Ett ramverk som möjliggör utnyttjandet av detta kallas OpenCL. Vi ställde oss frågan hur man skriver effektiva algoritmer till dessa GPGPU-enheter. Vi fann att det finns två sätt i huvudsak att köra kod parallellt på, data- samt uppgiftsparallellt. Körtiden av en algoritm beror snarare på andelen kod som kan köras parallellt än antalet processorelemet som finns tillgängliga på enheten. Vi bestämde oss för att testa teorin genom att implementera parallella algoritmer för matrismultiplikation och sortering av heltal med radix sort. Resultaten av testerna kan summeras till både bra och dåliga. Trots att det finns lovande vinst i prestanda så finns också faktorer som minnesåtkomster som inte alltid är lätt att kontrollera.

Statement of Collaboration

We divided the work of the implementations so that Mattias Svanström focused on the radix sort algorithm, while Simon Hössjer focused on the matrix multiplication algorithm. We had to research general knowledge of OpenCL and theory behind respective algorithms.

The analysis of the results, as well as writing of the report was mostly done together.

Document History

Version	Date	Edition
0.1	2012-03-06	NA
1.0	2012-04-12	First
1.1	2012-05-21	Final

Contents

Chapter 1

Introduction

In today's society many things are built using computers in one way or the other. The center most important part in all computers is the Central Processing Unit (CPU). Besides the CPU it is also common that there is a Graphics Processing Unit (GPU), though most often only in personal or work-related computers. These two processing units have been developed from two different sets of constraints. The purpose of the CPU is to handle tasks, only recently being able to handle several tasks simultaneously. The GPU has likewise been used to process graphics, though in a very parallel driven fashion. Because of its unique architecture, the GPU is being sought after as a supplementary computational unit.

In recent years frameworks for General Purpose computing on GPU (GPGPU) have found the market. As of writing there are two major implementations called CUDA and OpenCL. CUDA has been developed by NVIDIA and is thus only supported by their products[13], while OpenCL is built for cross-platform, being developed by Khronos Group. Since there is no significant difference in performance between these two [15] we have chosen to focus on OpenCL, because of its platform independence, as the main language in this paper.

When implementing an efficient algorithm, lowering runtime is often the main concern. Almost every personal or work-related computer today has a GPU that can be utilized for generic parallel tasks. The GPU has been proven to be able to lower runtime of certain algorithms, such as matrix multiplications. [14] Other related research includes various sorting algorithms, such as radix sort and bitonic sort, as well as pattern detection algorithms.

1.1 Purpose

Our aim is to describe the necessary steps to understand, construct and run concurrent algorithms on computers supporting parallel computing.

1.2 Problem statement

Some leading questions we try to answer are:

- How to use additional devices, as the GPU, for generic programming?
- How to write efficient parallel algorithms?
- How much faster does ones program get?

In other words, how to construct algorithms for GPGPU?

Chapter 2

Preliminaries

This chapter is about the necessary preliminaries in which to understand the rest of the report.

2.1 Technical background

In recent years, the central processing units in computers have gone through a change in the core of their design. Manufacturers used to focus on increasing the clock speed of the processor in order to improve the execution time of software. However, increasing the clock speed led to a higher power consumption which in turn led to an increased heat dissipation[5, 1.1 Why parallel]. This development ultimately led to inefficient processors, and manufacturers were forced to find new ways to increase the performance of their units.

Nowadays, instead of increasing the clock speed, manufacturers increase the number of cores within the processor. This enables a more economic power utilization while still being able to find the positive effects of faster software[3]. The consequence of this development leads to parallelization, i.e. software has to be designed in a parallel model in order to take full advantage of modern processors[5, 1.1 Why parallel].

The CPU and GPU follow two different paradigms of parellelism. Most modern CPUs are designed after Multiple Instruction Multiple Data paradigm, having multiple processing units each being able to process different data using different instructions[5, 1.2 Parallel Computing]. While most modern GPUs are designed after Single Instruction Multiple Data paradigm, having multiple processing units, where a single instruction is used across all units processing different data[5, 1.2 Parallel Computing].

2.2 Theoretical background

Software contains a set of instructions which can be executed. It is up to the programmer to decide in what order the instructions are executed. Either the order of execution can be sequential, where only one instruction is active at a time. Or it can be in a parallel order, where multiple operations are active at the same time, utilizing more computional resource[9, p. 7].

The order leads to two different types of programming models which are fundamentally different. It is typically not easy to change from a sequential model to a parallel model. Directly exporting a set of sequential operations and importing them in a parallel environment gives no automatic performance improvement[5, 1.3 Parallel Computing].

Furthermore, having instructions executed in parallel introduce problems which a sequential execution do not have to worry about. Problems arising from parallel execution include managing memory conflicts, scheduling individual instructions and handling data dependencies[9, p. 8]. A higher level language is necessary for parallel programming to be viable. OpenCL is one such language.

2.3 OpenCL

OpenCL is a framework which enables parallel programming of heterogeneous systems [5, 2.1 What is OpenCL?]. When different types of devices work together in a system the system is called heterogeneous. An example of a heterogeneous system is when the CPU and the GPU is used. Meanwhile, a homogeneous system is when it is not heterogeneous, that is when the devices are alike. For example the processors in a CPU can be used as a homogeneous system. OpenCL is not bound to be used in heterogeneous systems and can thus also be used for homogeneous systems [5, 2.1 What is OpenCL?].

Since OpenCL is built with heterogeneous systems in mind it is up to the host process to find and setup up the connections to the devices.

2.3.1 Platform model

OpenCL defines an abstract layer of a heterogeneous platform [9, p. 12]. An OpenCL platform always includes a host that is able to interact with the external environment outside of the OpenCL platform. The host also communicates with one or more OpenCL devices. Because of OpenCL's abstraction layer a device can be a CPU, GPU or any other device supporting OpenCL. Furthermore the devices are composed by compute units which themselves are made up of processing elements. See figure 2.1 for an overview of the platform model. The computations on a device are made in the processing elements[7, p. 19].

A set of devices in a platform can form a context which becomes the environment that executes kernels.

2.3.2 Memory model

In every OpenCL device there are four types of memory which can be used. The types are global, constant, local and private memory. The host is only able to access

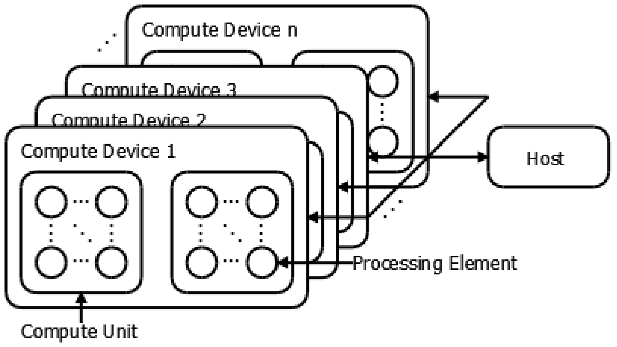

Figure 2.1: The platform model of any OpenCL application [2, OpenCL Platform Model].

the global and constant memory on a device while the local memory is used by each computing unit and the private memory is used per processing element. Figure 2.2 illustrates the OpenCL memory model on a device.

2.3.3 Execution model

OpenCL uses so called command queues to queue tasks to be executed in a context. Kernels can be executed as a single task or over an index space, which can have 1, 2 or 3 dimensions [2, Kernels]. For every element in the index space a work item is created. All work items execute the same program. It should be noted, though, that the code may branch.

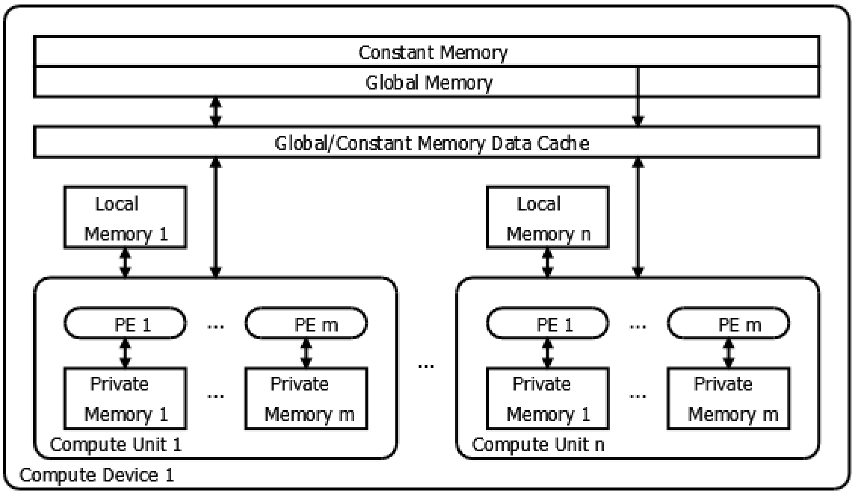

Figure 2.2: The memory model as can be seen in a device [2, OpenCL Memory Model]. PE is the processing elements.

2.3.4 Performance

Even though OpenCL is built as a cross-platform framework the inventors claims that the performance is still top priority[5, 2.3 An Overview of OpenCL: Performance]. Apparently, speed is such an important aspect of OpenCL that it is possible for developers to use platform-specific instructions, throwing away platform independence[7, p. 361].

Chapter 3

Process

This chapter will explain the methods used for this project. Section 3.1 will define and explain the theory needed for parallel algorithms. Section 3.2 will describe the algorithms, and how the methods were used to implement them.

3.1 Algorithm Construction

3.1.1 Sequential vs. parallel

Traditionally, software has been designed after a sequential model. Instructions are executed one after another having only a single instruction active at a single point in time. However, modern hardware is designed in a parallel model. The main source of performance comes from parallel execution. In order to take advantage of that, it is necessary to design programs which fit a such an architecture. Ideally, the program should be able to handle multiple operations executing at the same time with no dependencies between data. On modern computers, sequential software will lead to unutilized hardware and inefficient programs.

3.1.2 Runtime of parallel algorithms

Many people write parallel algorithms for the sole purpose of increasing performance. This begs the question of how much faster the program as a whole gets.

One law that tries to shed light on performance gains is Amdahl's Law. The law works under the assumption that parallel code scales with the number of working units[6, p. 94]. If we let S denote the time the program spends in sequential code and P the time to spend in code which can be run in parallel, then it is easy to understand that the runtime of the program, as a whole, will be that of $S + P$. However, since the parallel part of the program can be optimized by running it concurrently the following formula appears:

$$\text{Runtime} = S + \frac{P}{N} \tag{3.1}$$

where N is the total number of work units available. We see from equation (3.1) that when the number of work units increases the runtime gets dominated by the sequential part of the program. We also see that the gained speedup drastically decreases as the sequential part of the program increases. Consult table 3.1 for an illustration of this phenomena. Amdahl's Law tells us the maximum speedup possible. But practically it is unrealistic as, among other things, the division and distribution of the parallel part of the program to the work units can be unbalanced. None the less, it gives us an approximation of the maximum imaginable speedup.

Parallel percentage (%)	Runtime (%)	Speedup
100	$0 + 100/8 = 12.5$	$100/12.5 = 8$
90	$10 + 90/8 = 21.25$	$100/21.25 \simeq 4.7$
50	$50 + 50/8 = 56.25$	$100/56.25 \simeq 1.8$

Table 3.1: Examples of gained speedup with 8 work units available. The speedup decreases exponentially with the amount spent in sequential code.

3.1.3 Implementation

Certain steps must be taken to design parallel software. The programmer has to find which instructions, if any, can be executed in parallel. For example, the problem being solved could be shaped in a way where certain instructions depend on data from a previous instruction. Those instructions would not be possible to parallelize. Furthermore, having instructions operating in parallel causes overhead. Both from managing the concurrency and from reading and writing data to memory. The programmer must consider if the benefits of parallelization outweigh those negative effects. Essentially, the problem has to be analyzed carefully.

3.1.4 Parallel models

There exist many parallel programming models. The two most general models are data-parallel and task-parallel. They are suited for different types of problems and hardware.

In a data-parallel model, the problem can be split in to a set of individual data elements. The parallel work lies in performing the same operations on partitions of the data set, where each partition can be updated concurrently. An overview of data-parallelism is shown in figure 3.1. The data set here consists of a single array of length n. Each element in the array is assigned to a processor element. Each processor element concurrently applies a function on their partition which renders the resulting array.

Task-parallel on the other hand split the problem in a set of individual operations which can be run concurrently. Here, the parallel work lies in executing the operations in the set concurrently. An overview of task-parallelism is shown

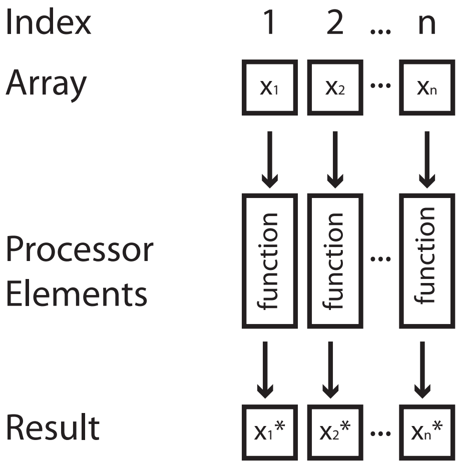

Figure 3.1: The data-parallel model shown on an n long array. The topmost row is the input array, the following row show the processor elements executing the function, and finally and the resulting array.

in figure 3.2. The function set here consists of n operations. Each processor element is assigned one or more functions, where the gray area depicts placeholders for functions between x_3 to $x_n - 1$. Since each function will need a varied amount of computational resources, each function will finish at different times. A problem here is to schedule all functions in an optimized manner leaving no processor element idle.

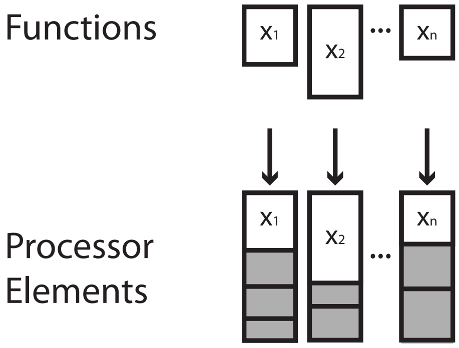

Figure 3.2: The task-parallel model shown on a set of n functions. The topmost row is the functions and following row show the processor elements.

3.1.5 Sequential algorithms

Some algorithms have neither data nor task independence. One such example is the Fibonacci series. The Fibonacci series is defined as

$$a_i = a_{i-2} + a_{i-1}$$

where a_i is the ith element of the series for $i \geq 2$ with the definitions of $a_0 = 0$ and $a_1 = 1$. The Fibonacci series is a recursive series. To calculate the ith element one also needs the two preceding elements, which in turn needs their preceding elements and so forth. This is an algorithm that will not benefit from GPU computing.

3.1.6 Using OpenCL

The choice between which parallel model to use is largely dependant on the problem being solved. OpenCL support both data- and task-parallel models on any compatible hardware. However, the model and hardware must be chosen wisely, because the hardware is designed to fit a specific model. For example, the data-parallel model is best suited on the GPU, because it is designed with a large amount of cores which can execute the same task. While the task-parallel model is better suited on the CPU since it is designed with fewer cores having tasks executing independently of each other.

When setting up the application for a heterogeneous system the host ought to do the following steps [9, p. 11].

1. Discover the components that make up the heterogeneous system.
2. Make sure that the software suits the appropriate hardware.
3. Load the kernels which are to be executed.
4. Set up the needed memory for the application.
5. Execute the kernels in the right order and on the right devices.
6. Collect the results of the computations.

It should be noted that while using the GPU as a computational device the CPU does not have be idle, waiting for the calculations to complete. In some applications it is wise to divide the work between the GPU and the CPU to increase performance further [14]. For instance, the CPU can be used to help calculate a portion of a matrix multiplication together with the GPU.

3.2 Practical testing

3.2.1 Matrix multiplication

Computers are very good at crunching large amounts of data. Multiplying large matrices is a task very well suited for computers.

How it works

Let A and B be matrices of size $m \times p$ and $p \times n$ respectively. Then the two matrices can be written as

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,p} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,p} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{p,1} & b_{p,2} & \cdots & b_{p,n} \end{bmatrix}.$$

The resulting matrix $C = A \cdot B$ could then be written as

$$C = \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \cdots & c_{m,n} \end{bmatrix}$$

where the element $c_{i,j}$ $(1 \leq i \leq m, 1 \leq j \leq n)$ is calculated by

$$c_{i,j} = a_{i,1} \cdot b_{1,j} + a_{i,2} \cdot b_{2,j} + \ldots + a_{i,p} \cdot b_{p,j},$$

that is the scalar product of row i from A with column j from B.

Analyzing dependencies

The attentive reader notices that the scalar products for every element in the resulting matrix are all independent of one another. Thus they can all be run concurrently.

However, is it possible to optimize more? Yes, somewhat. The scalar product can be improved by splitting it into parts that each computes a lesser part of the scalar product. When the parts have completed, their results need to be summarized to get the final result of the scalar product. But the more we split the less work every part does and the more we need to summarize. We get an optimization problem because too many parts, as with too few, would lead to a loss of performance.

Implementation

We decided to focus on squared matrices being power of two to keep the implementation as simple as possible.

Two algorithms, beside the sequential algorithm, was implemented for OpenCL. The first one was optimized to calculate the scalar products of the resulting elements concurrently. The second version tries to optimize that very scalar product by splitting it into a number of equally sized parts.

Code for our matrix multiplication algorithms can be found at A.1.

3.2.2 Radix sort

Sorting is a fundamental problem in computer science [4, p. 197]. Applications commonly need sorted data for internal calculations or external display. As an effect, applications rely on efficient algorithms to maximize their own performance.

How it works

Radix sort is one of the most efficient sorting algorithms[10, p. 3]. The algorithm assumes that the keys in the array are n-bits long and process the keys one bit at time starting at the least significant bit moving to the most significant bit.

Within each of the n passes three different tasks are performed. First the keys are scanned and placed in buckets depending on if the bit at the current pass is 0 or 1. Secondly, relocation offsets are computed by counting the number of keys with the same bit occurring earlier in the array. This is the offset index at which the element should be written. Having computed the relocation offset the pass is completed by scattering all of the elements at the correct offset in the output array. Figure 3.3 illustrates this process.

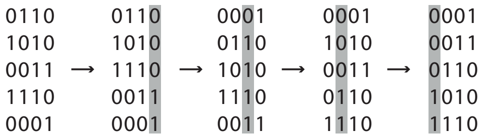

Figure 3.3: The radix sort algorithm on an input array containing elements of 4 bits. The leftmost column is the input and the following columns show the resulting array after each pass.

Analyzing dependencies

The different tasks done in each pass depend on data from the previous task. For example, the relocation offsets can only be computed after the scan is complete, and the scattering of elements can only be done once the offsets have been calculated. Therefore, it is not possible to execute the tasks in each pass in parallel.

After analyzing the individual tasks, data independence is found in the task of scanning and scattering the keys. These tasks execute instructions involving reading and placing elements in an array. Thus, they can execute in parallel individually, especially suited for the data-parallel model. Data dependence is however found in the task of calculating the relocation indices, where every index depend on the index of the previous element. It can only run sequentially.

Implementation

We decided to focus on sorting arrays sized in the power of two, containing integer data of 32 bit size. The reason being a more simple implementation.

Beside the sequential algorithm, a parallel algorithm was implemented in the OpenCL framework. It is using the properties mentioned in the previous section, where two of the tasks in each pass execute in a data-parallel model, and the third task executing sequentially on a single core in the parallel system.

Code for our radix sort algorithms can be found at A.2.

Chapter 4

Results

This chapter presents the results of the implemented algorithms described in the previous chapter.

4.1 Hardware

For our testing environment we chose to use personal computers rather than a more efficient distributed computing system, e.g. cluster. This was done for two reasons. Unlike a few years ago, parallel processing is now common in modern computers, most computational hardware is designed with a parallel architecture. It was an interesting task to take advantage of that in an easily accessible way. Another reason for not using a cluster was to get more relevant results. Normally, people do not have access to such systems.

All tests have been run on several different systems. This was done in order to find discrepancies between the systems, ultimately yielding a more credible result. The system specifications used in our testing environment can be found in table 4.1.

Table 4.1: Hardware specifications used.

Identifier	Alpha	Beta	Gamma
CPU	Intel Core 2 Quad (2.83 GHz 12MB L2 Cache)	Intel Core i5-450M processor (2.4 GHz, 3MB L3 Cache)	Intel Core i5-750 processor (2.67 GHz, 8MB L3 Cache)
GPU	GeForce 8400 GS 512MB, 16 cores[11]	ATI Mobility Radeon HD 5650 Graphics 1GB, 400 Stream Processing Units[1]	GeForce GTX 560 Ti 1GB, 384 CUDA cores[12]
OS	Linux 2.6.32 UBUNTU	Windows 7 Home Premium 64-bit	Linux 3.3.1 ARCH

4.2 Implementations

All computers have run a sequential algorithm and the GPU implementations. However, we found that both the computers Beta and Gamma additionally had the ability to use the CPU as a computing device, apart from the GPU. Therefore, we decided to test our implementations on those devices as well.

It's always a good practice to place results in a context. Since multiplying matrices is a common application, many programs have already optimized it to its fullest. One such program is Matlab, but seeing as only the Gamma computer had access to it, we had to find another reference program which all computers supported. We decided to use uBLAS[16], a C++ class template library being a part of the boost library, as another matrix multiplication reference, because of its platform independence and ease of use.

4.2.1 Matrix multiplication

The matrix multiplication algorithms uses a fair amount of memory and because of this, testing matrices of sizes over 1024 × 1024 was not viable. By consulting equation (3.1) we expected that the runtime of the first OpenCL implementation of the matrix multiplication algorithm to be roughly 16, 400 and 384 times faster, for the respective computers, than the sequential algorithm. For the second algorithm the maximum speed gain is still the same as the first implementation, as the number of processing units have not changed. Of course factors such as clock frequency and memory accessing affect the expected performance gains.

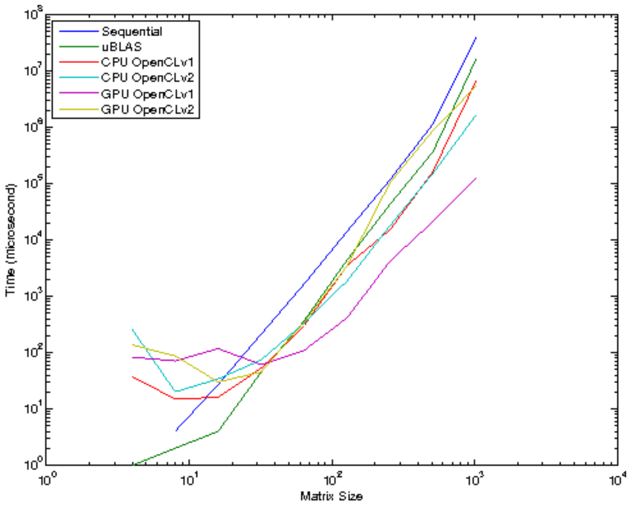

Figure 4.1: Graph showing the results from the Beta computer. Running the first OpenCL algorithm on the GPU gave the fastest runtime of about 300× faster.

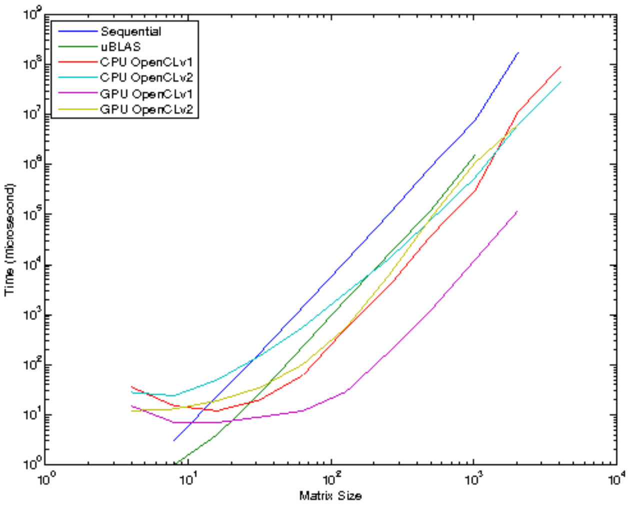

Figure 4.2: Graph showing the results from the Gamma computer. As with Beta the first OpenCL algorithm yielded the fastest runtime, 1000× faster than its sequential counterpart.

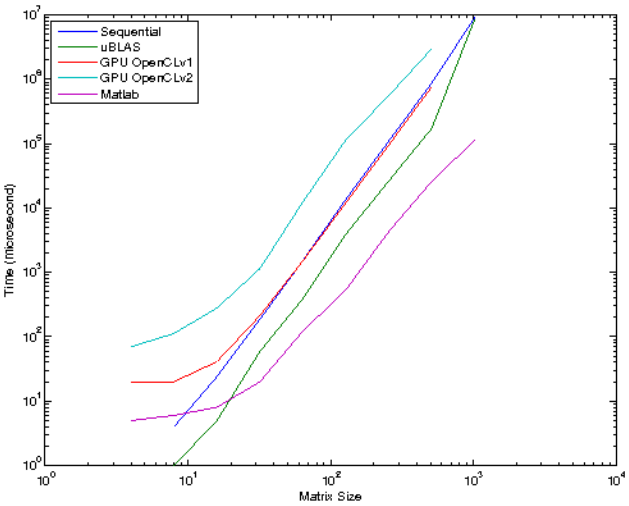

Figure 4.3: Graph showing the results from the Alpha computer. No runtime of the OpenCL implementations appears to be that far from the sequential algorithm. No noticeable speedup gain can be seen.

From what we can see in figure 4.1 the overhead of using OpenCL is greater than the speed of the sequential algorithm. For larger matrices, however, the overhead can be negligible. We also see that, for larger matrices, the sequential algorithm was slower than all the other implementations and that the first OpenCL implementation running on the GPU was the fastest. uBLAS, on the other hand, was faster than the sequential algorithm but almost always slower than our OpenCL implementations. By comparing the fastest runtime with the slowest at the largest tested matrix we approximate the speedup gain to be a factor of 300; well in the limit of our expected 400.

Looking at figure 4.2 we see much smoother curves for smaller matrices and straighter lines for larger than from the previous computer. Tough we again notices that the sequential algorithm was the slowest for larger matrices and the first OpenCL implementation running on the GPU was the fastest. Still, the uBLAS implementation was slower than our OpenCL implementations. However, by comparing the results at the largest tested matrix a most peculiar result appear. We see a speedup gain of well over 1000, a result we surly did not anticipate as the number of cores available are only 384.

Lastly we look at figure 4.3 and sees that the effect of GPU computing is almost non existing. For larger matrices the first OpenCL implementation is almost identical to the sequential algorithm. The second version, however, appears to be slower than both of them by a factor of 5. Though this time, uBLAS is faster than both the sequential algorithm and our OpenCL implementations. The last curve in the figure is that of matrix multiplication in Matlab. What we see is a major speedgain worthy of challangeing the fastest OpenCL implementation from both the previous computers. It begs the question of how fast Matlab would have multiplied the matrices on those computers.

4.2.2 Radix sort

Calculating an expected speed up from equation (3.1) becomes difficult seeing as the algorithm execute tasks both sequentially and in parallel. The total runtime for the algorithm is the sum of the runtime for each individual task. Dividing this value with the total runtime of the sequential implementation gives the expected speed up. The parallel implementation is expected to be roughly 2 times as fast for the Beta and Gamma computer, and 1.8 times as fast for the Alpha computer, due to fewer cores.

Figure 4.4 show the runtime for each implementation on the Beta computer. As can be seen from the graph, the results of the parallel implementations vary greatly depending on if it was run on the GPU or CPU. For smaller arrays, we can see that the sequential algorithm performs best. However, for arrays larger than 512 elements we note a speed up in the parallel CPU implementation of roughly 5 times. This factor is somewhat larger than what was expected. We note that the results for the parallel implementation on the GPU display a significant decrease in performance compared to the other implementations.

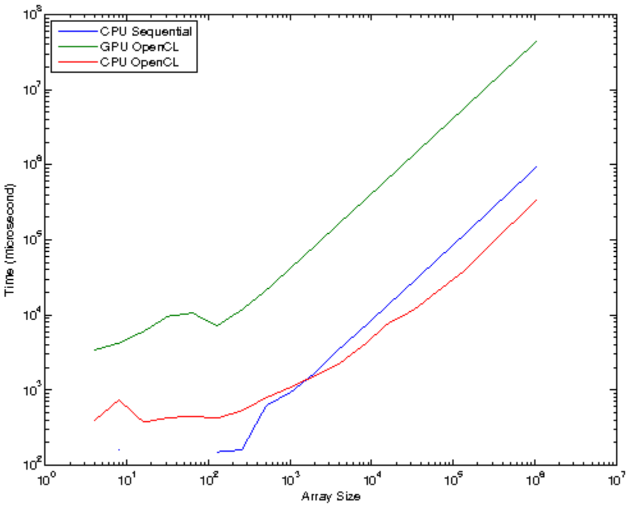

Figure 4.4: Graph showing the results from the Beta computer. Running the OpenCL algorithm on the CPU gave the fastest runtime. The GPU implementation performed worst.

Figure 4.5 show the runtime results on the Gamma computer. Similarly to the previous graph, we note that the parallel CPU implementation performs better on arrays larger than 512 elements. By comparing the fastest runtime to the sequential runtime we approximate the speed up gain to be a factor close to 5, slightly larger than expected. We again note that the parallel GPU implementation performs the worst of all implementations.

Figure 4.6 show the runtime for the two implementations on the Alpha computer. Looking at the graph we see the the parallel GPU implementation performing significantly worse than the sequential implementation. No increase in speed up was found in this result.

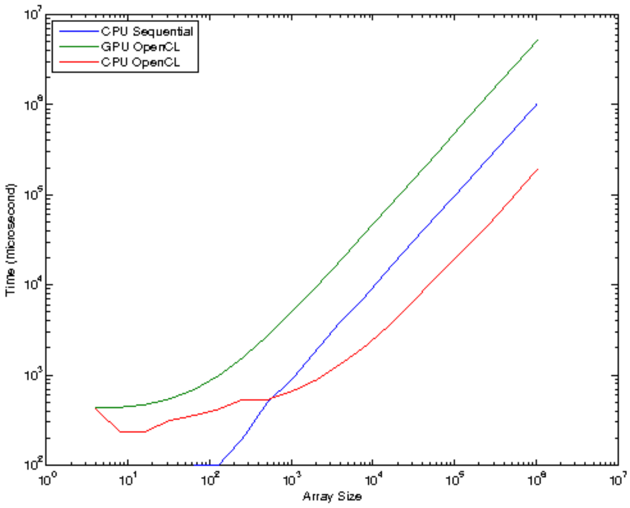

Figure 4.5: Graph showing the results from the Gamma computer. As with the Beta computer, the OpenCL algorithm on the CPU give the fastest runtime. The parallel GPU implementation performed worst.

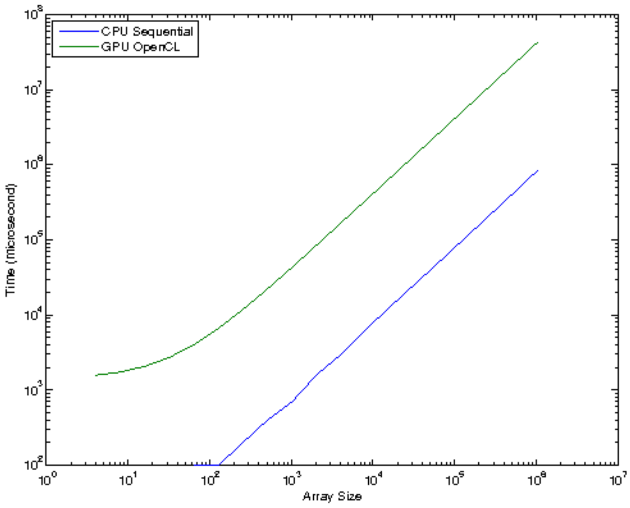

Figure 4.6: Graph showing the results from the Alpha computer. The sequential algorithm performed better than the parallel GPU implementation.

Chapter 5

Discussion

This chapter will explain the results gathered from this report. Section 5.1 will discuss the accuracy and fairness of the results. Section 5.2 will explain the conclusions drawn from this report, as well as the answers to the leading questions in section 1.2.

5.1 Analysis of results

Running tests of algorithm efficiency is difficult as the results are highly dependant on the hardware being run on. Still, we believe that we at least tried to avoid any hardware specifics by running tests on several computers. Of course, a few things are then again questionable. How come we got an increase of speed by a factor of 1000 on one computer multiplying matrices but not on the others? Why did the GPU perform worse than the sequential algorithm when sorting integers?

5.1.1 Measure of time

Time was measured for the sequential algorithms with the C standard library *time.h* using the *clock()* function. However, since *clock()* measures time badly, partly because different operative systems measures time differently, but also because of interference problems, the time was measured over several iterations of the algorithm and then divided by that number to get an average per run. That way we got more stable and trustworthy results.

For the OpenCL algorithms, however, the time was measured by the API itself using *cl_event* while enqueueing. Trying to measure time with *clock* will only result in failure as the enqueueing of tasks is asynchronous and will thus only display the time needed to place the task on the queue. The OpenCL *cl_event* is not entirely secure from interference either. It might have been a good idea to measure the time as an average here as well.

5.1.2 Matrix multiplication

For all implementations of the second OpenCL implementation it is believable that one can improve the results by choosing another number of sections. By doing small tests of the scalar product with varying sizes of arrays we found that the optimal number of sections varied by the size of the array and was thus not constant. Another number of sections will surly give another result for the matrix multiplication. We had trouble testing this theory as our computers ran out of memory as the algorithm takes up an extra factor of *number of sections* more memory than the first algorithm to be able to store all the partial scalar products.

Is there a logical reason to why the second OpenCL algorithm performed worse than the first? See it like this, if we look at a relative small 512×512 matrix, the number of elements to calculate would be a total of $512 \cdot 512 = 262144$. Now, if we have, say, 500 process elements, we would only be able to compute 500 elements at a time, all of which takes an equal amount of time to calculate. In other words, we would have to load and run $262144/500 \simeq 525$ sets of 500 elements each to calculate them all. Since all process elements are busy working in all sets but the last, is there any reason to divide the work any further? Will we gain any performance benefits? No, if anything we lose performance as more sets of elements gets loaded to run and more overhead needs to be stored.

5.1.3 Radix sort

When looking at the results of the radix sort implementations we see that the parallel GPU implementation performs worse than the parallel CPU implementation, in fact it even performs worse than the sequential implementation on all systems. We believe that the slow GPU results does not depend on our OpenCL implementation, since it performs as expected on the CPU. We speculate that one reason for these results is because of the way the GPU is designed. The multiple number of cores in the GPU will in this case have a negative effect on performance. Specifically, each core will need access to multiple memory objects on random indices, causing too large of a overhead in data movements for it to be effective. While the CPU has a design better suited for this algorithm. It is constructed with less cores, which means less overhead on data movement. Furthermore, the CPU can utilize its cache mechanisms to prevent slow memory operations on each instruction, something which the GPU does not have.

5.2 Conclusions

Using additional devices

In order to take advantage of additional devices for generic programming, it is neccessary to use a framework such as OpenCL. Specifically for OpenCL there are some important things to be aware of. There is a host application that probes which devices are available, and can be used as additional computational units.

An important property of the host application is that it manages all of the memory usage for the devices. The host application also handles the communication between the devices. In essence, it loads the programs on them, ensures that they are executed in the right order, and finally it interprets the computed results.

Efficient algorithms

Constructing efficient algorithms is not a trivial task. The ideal algorithm would connect the hardware and software in a symbiotic relationship. To be able to do that, the algorithm has to be analyzed carefully. If it contains a lot of dependencies between data, it might not be worth to implement a parallel version.

Speedup

Increasing the number of work units will not automatically lower the runtime of the program. The large speedups are gained by the amount of the program being able to run concurrently.

At least it is safe to say that finding data independences in algorithms and optimize them to data parallel algorithms can drastically decrease runtime. It is possible, however, that memory access and cache failures can neutralize the gained performance.

Bibliography

[1] AMD. *ATI Mobility Radeon HD 5650 GPU Specifications.* 2012 [cited 2012-04-11]. Available from `http://www.amd.com/us/products/notebook/graphics/ati-mobility-hd-5700/Pages/hd-5650-specs.aspx`

[2] AMD Staff. *OpenCL and the AMD APP SDK.* 2011 [cited 2012-04-11]. Available from `http://developer.amd.com/documentation/articles/pages/opencl-and-the-amd-app-sdk.aspx`.

[3] Chandrakasan AP, Potkonjak M, Mehra R, Rabaey J, Brodersen RW. *Optimizing Power Using Transformations.* 1995. 27 p.

[4] Cormen T, Leiserson C, Rivest R. *Introduction To Algorithms.* 3rd ed. MIT Press; 2009.

[5] Fixstars Corporation (Tsuchiyama R, Nakamura T, Iizuka T, Asahara A, Miki S). *OpenCL Programming Book.* Fixstars Corporation; 2010 Mars 31 [cited 2012-04-11]. Available from `http://www.fixstars.com/en/opencl/book/index.html`.

[6] Gove, D. *Multicore Application Programming For Windows, Linux, and Oracle Solaris.* Addison-Wesley Educational Publishers Inc; 2011.

[7] Khronos OpenCL Working Group. *The OpenCL Specification.* 2011. 377 p.

[8] Krüger F, Maitre O, Jiménez S, Baumes L, Collet P. *Speedups between x70 and x120 for a Generic Local Search (Memetic) Algorithm on a Single GPGPU Chip.* 2010. 501–511 p.

[9] Munshi A, Gaster B, Mattson T, Fung J, Ginsburg D. *OpenCL Programming Guide.* Addison-Wesley Professional; 2011.

[10] Nadathur S, Harris M, Garland M. *Designing Efficient Sorting Algorithms for Manycore GPUs.* 2009 May. 10 p.

[11] NVIDIA. *GeForce 8400 GS Specifications.* 2012 [cited 2012-04-11]. Available from `http://www.nvidia.com/objects/geforce_8400M.html`.

[12] NVIDIA. *GeForce GTX 560 Ti Specifications.* 2012 [cited 2012-04-11]. Available from `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-560ti`.

[13] NVIDIA. *What is CUDA.* 2012 [cited 2012-04-11]. Available from `http://developer.nvidia.com/what-cuda`.

[14] Ohshima S, Kise K, Katagiri T, Yuba T. *Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment.* 2005. 14 p.

[15] SiSoftware. *Q&A: Benchmarks: OpenCL GPGPU Performance (OpenCL vs. CUDA/STREAM).* 2009 [cited 2012-04-11]. Available from `http://www.sisoftware.net/?d=qa&f=gpgpu_gpu_perf&l=en&a=`.

[16] Walter J, Koch M, Winkler G, Bellot D. *Basic Linear Algebra Library.* 2010 [cited 2012-05-21]. Available from `http://www.boost.org/doc/libs/1_49_0/libs/numeric/ublas/doc/index.htm`.

Appendix A

Code

A.1 Matrix multiplication

```
__kernel void matrix_multiplication(const int size, __global const int* in1,
                __global const int* in2, __global int* out)
{
  int row = get_global_id(0);
  int col = get_global_id(1);

  int sum = 0;
  for (int i = 0; i < size; ++i)
        sum += in1[i + row * size] * in2[col + i * size];

  out[col + row * size] = sum;
}

__kernel void matrix_multiplication_by_section(__global const int* in1,
                __global const int* in2, __global int* out)
{
  const int size = get_global_size(0); // = get_global_size(1)
  const int sections = get_global_size(2);

  const int row = get_global_id(0);
  const int col = get_global_id(1);
  const int section = get_global_id(2);

  const int section_size = size / sections;

  const int from = section_size * section;
  const int to = from + section_size;

  int sum = 0;
  for (int i = from; i < to; ++i)
        sum += in1[i + row * size] * in2[col + i * size];
  out[section + (col + row * size) * sections] = sum;
}
__kernel void matrix_multiplication_sum(const int sections,
                __global const int* in, __global int* out)
{
  const int size = get_global_size(0); // = get_global_size(1)

  const int row = get_global_id(0);
  const int col = get_global_id(1);

  int sum = 0;
  for (int section = 0; section < sections; ++section)
        sum += in[section + (col + row * size) * sections];

  out[(col + row * size) * sections] = sum;
}
```

A.2 Radix sort

```
__kernel void scan(__global int* in, __global int* out, int phase)
{
        int i = get_global_id(0);

        out[i] = in[i] & phase;
        out[i] = out[i] != 0;
}

// in = hash
// out = relocation offsets
__kernel void index(__global int* in, __global int* out, int size)
{
        int offset = 0;

        for (int i = 0; i < size; i++) {
                out[i] = offset;
                offset += in[i] == 0;
        }

        for (int i = 0; i < size; i++) {
                out[size+i] = offset;
                offset += in[i];
        }
}

// in = alla våra tal
// out = out
__kernel void relocate(__global int* in, __global int* hash,
                __global int* out, __global int* offsets)
{
        int i = get_global_id(0);
        int size = get_global_size(0);

        out[offsets[i + hash[i] * size]] = in[i];
}
```

www.kth.se