Democratizing General Purpose GPU Programming through OpenCL and Scala

A Dat 5 report by:

Raider Beck
Helge Willum Larsen
Tommy Jensen

Supervised by:

Bent Thomsen

JANUARY 10, 2011

AALBORG UNIVERSITET

Student report – *not* an official document

Aalborg University
Department of Computer Science
Selma Lagerlofs Vej 300
9220 Aalborg
Telephone:(45)96358080
http://www.cs.aau.dk

Title:

Democratizing General Purpose GPU Programming through OpenCL and Scala

Theme:

GPGPU, Productivity

Project timeframe:

Dat5, September 1 2010 - January 10, 2011

Project group:

d505a

Group members:

Helge Willum Larsen

Reidar Beck

Tommy Jensen

Supervisor:

Bent Thomsen

Abstract:

General Purpose GPU programming has the potential to increase the speed with which many computations can be done.
We show a number of examples of such improvements, investigate how one can benchmark different implementations of GPGPU programming paradigms and how one can measure the productivity of programmers.
Finally we implement and describe a simple toolkit to enable ordinary programmers to benefit from the use of a GPU.

Copies: 5

Total pages: 93

Appendices: 0

Paper finished: 10$^{\text{th}}$ of January 2011

Preface

This report was written during the Dat5 project period by group d505a.

The intended audience of the report are people who have at least the same general level of knowledge in computer science as the authors, although little knowledge of GPU programming is assumed.

A note on numbering In this report figures, tables and equations have a number of the form $x.y$ where x is the page on which they were inserted and y is a unique number for that page. This should make it easier for the reader to locate the reference in question, as it will be found on either that or the next page. We realize this breaks with the tradition of using section numbers and may produce some confusion, but we believe that the benefits makes up for these disadvantages once the reader has read this introduction.

Finally, we would like to thank:

Bent Thomsen for supervising throughout this project.

Contents

For, usually and fitly, the presence of an introduction is held to imply that there is something of consequence and importance to be introduced.

Arthur Machen

1 Introduction

In this report we document and develop an implementation of a toolkit that enables programmers who have little experience with GPU programming to take advantage of the fast parallel computations that todays graphics cards are capable of.

The product is aimed at programmers who do not necessarily have a degree in Computer Science, but the intended reader of the report is somebody who has at least the equivalent of bachelor degree in Computer Science and has had some experience with compilers and functional languages.

This report is structured as follows: in chapter 2 we introduce the problem statement as well as the motivation for our project along with a short history of GPU programming. In the analysis part we present various things people have used the GPU to gain speedup, then we describe how OpenCL is structured and how it is used.

In chapter 6 we presents a number of considerations that must be taken into account in order to improve the performance of the code running on the GPU. In chapter 7 we look at some ways to measure and improve programmer productivity, chapter 8 is devoted to paradigms which have little in common with OpenCL but which may result in improved programmer productivity. Chapter 9 looks at ways one might benchmark a GPU programming environment.

Chapters 10, 11 and 12 an devoted to documenting how we implemented a small simple proof of concept for a toolkit to make it easier to program the GPU.

In the final part of the report we evaluate the project and present several ideas for extensions and continued development of the toolkit.

If we can really understand the problem, the answer will come out of it, because the answer is not separate from the problem.

Jiddu Krishnamurti

2 Problem statement

A GPU is a very efficient chip for concurrent floating point calculations, but at present it is programmed in a rather low-level C-like language. We want to examine whether it is possible to make a better high-level abstraction, and look at how its performance might be benchmarked against alternatives, both aimed at the GPU and the CPU.

2.1 Motivation

Before we begin programming the GPU, we have to answer the obvious question: why?

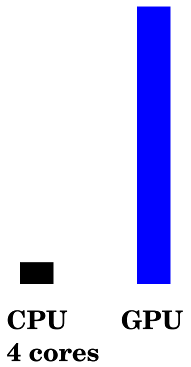


Figure 11.1: GFLOPS on a Xeon E5420 CPU versus a Tesla C870 GPU

The answer is equally simple: because the GPU, when utilized correctly, can do floating point operations much faster than the CPU. Since the GPU architecture is parallel by nature, it will become even more powerful as transistors continue to shrink, giving room for more cores while the clock-frequency remains relatively unchanged.

Our test equipment contains a quad-core Intel Xeon E5420 processor and two NVIDIA Tesla C870 GPUs. Even if all four CPU cores are fully utilized, the Xeon can deliver no more than 40 GFLOPS[Int], while each Tesla card is able to achieve up to 350 GFLOPS performance (512 GFLOPS peak)[Nvib].

2.2 Differences between the CPU and GPU

It would be wrong to simply compare the theoretic speed of the CPU and the GPU, because they are two fundamentally different architectures – a GPU is a SIMD[1] architecture, whereas the CPU is a SISD[2] architecture; there are things for which one architecture is better suited than the other.

Because the CPU operates on one piece of data at a time[3], it is not able to take advantage of situations where the same instruction should be carried out on different pieces of data; it is certainly possible for it to do so but it will not happen as fast. On the other hand, it is good at applying different operations to different data.

Contrary to the CPU, the GPU typically has a larger number of threads that all run the same instructions on different data, such as computing the sine value of a large set of input data. The GPU is less efficient in situations where the operations on the data vary, such as computing the sine value for some of the input but applying another operation to the rest. For concurrent operations it can however be very fast[4].

[1] Single Instruction, Multiple Data

[2] Single Instruction, Single Data

[3] Multicore processors can operate on more than one thing at a time, but most CPUs today have very few cores compared to even the cheapest graphic cards so it does not fundamentally change the way things work.

[4] 350 GFLOPS on the Tesla C870[Nvib] versus 40 GFLOPS on the Xeon

Another reason to focus on the GPU model of development is the increasing number of CPU cores. As a result of the continuing shrinking of the transistors, extra space on the CPU die is no longer used to make it faster, but to give it more and more cores. If Moore's law stays true and the number of transistors, and thereby cores, double every 18 months, in 15 year we will see 2048 core processors as common as dual-cores are today. This means the CPU of 2025 may look more like the GPU than the CPU of 2010.

2.2.1 A Short Overview of the Architecture of a GPU

This section provides a short overview of the architecture of the GPU; a more detailed overview will be given in 6.

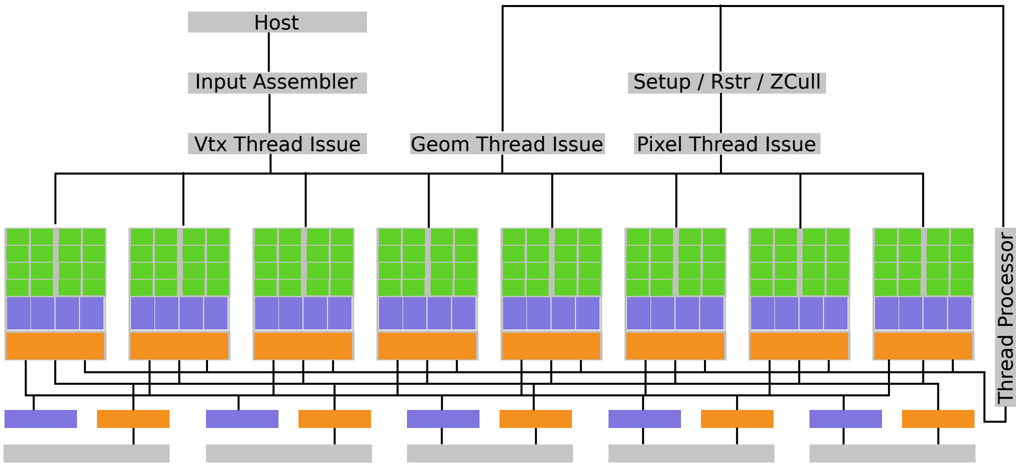


Figure 13.1: Block diagram of a Geforce 8800 graphic card [Nvia].

Figure 13.1 shows a block diagram of the architecture of the Geforce 8800, on which our Tesla cards are based. As seen in the figure the GPU has a number of blocks, each of which contains a large number of threads per block. Per specification[Nvi10a], the threads, in groups of 32, must execute the same instruction. This gives some idea about the cost of running code that treats each piece of data differently.

E5420[Int].

2.3 A short history of GPU programming

While many of the technologies in general purpose GPU programming are new, it does have a relatively long history dating back to at least 1978[DOLG$^+$, p. 36]. However, it was not until the beginning of the 2000's that the OpenGL API and DirectX added programmable shading to their capabilities[Wikc], exposing GPU programming to the mass market.

Until CUDA and CTM emerged in 2006[Nvi10a][AMDb], programmable shaders were practically the only way to program the graphics card in consumer computers. Shaders were not designed for general purpose computing and so put limitations on what could be done.

Along with the NVIDIA's CUDA platform and ATI's competing CTM (Close to Metal) programming interface[1] came the widespread exploitation of GPUs to accelerate non-graphics related computation. The concept of using GPGPUs (General Purpose Graphics Processing Units) as a modified form of stream processor has become one of the hot topics in parallelism, and is considered to be one of the next *Big Things* in computing.

[1] CTM was short-lived and evolved to ATI CAL (Compute Abstraction Layer) as part of the ATI Stream SDK in December 2007[AMDb]

Part I

Analysis

This is a world of action, and not for moping and droning in.

Charles Dickens

3 Various uses of the GPU

This part of the report presents an variety of different things various people have used the GPU to do. Some of these examples predates the CUDA framework (they can be found it the survey paper[DOLG$^+$]), and instead uses programmable shaders to do the GPU calculations with – a method that is both more difficult (since shader programming is much more restrictive than CUDA or OpenCL, but works on almost all GPUs) and unlikely to be as effective; nonetheless some impressive speed improvements can be found in these examples.

Since there are so many different things one could use GPU programming for, this part does not contain a complete survey. Instead of presenting a complete survey, we aim to give some idea of what GPU programming has been used for so that we might better understand its benefits and disadvantages.

3.1 Cryptography

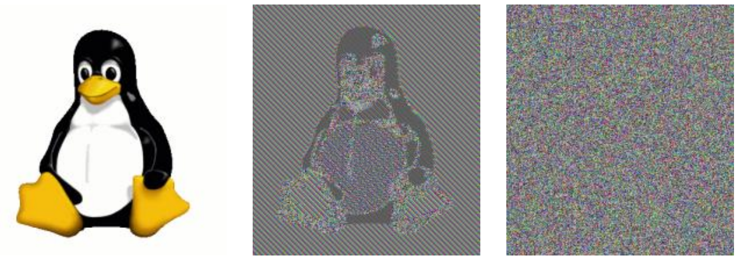
One of the areas where more computer power is always desirable is the area of cryptography: if we have enough computational power to encrypt something, others will desire more computer power to break the encryption. If they have enough computer power to break the encryption, we desire more power to use a better encryption method.

In addition to the aforementioned speed requirements, cryptography is also well suited for GPU acceleration since many modern encryption standards, such as AES, encrypt the plain-text as a series of blocks. Each of these blocks can be encrypted separately from the rest, so a GPU implementation can take advantage of this to encrypt the input in parallel.

We will not describe AES in detail here but all the details, as well as source code, can be found in[Sch95] under the name Rijndael.

[GRV10] describe an implementation of AES which takes advantage of the GPU to achieve speed-ups of about 5 times, compared to a straight CPU implementation.

Unfortunately one cannot just use his method directly, because although the blocks can be encrypted separately doing so means that the blocks cannot be chained – and an attacker can change the order of a blocks as well as delete any number of blocks. In addition pattern in the plain-text may leak into the blocks (see Figure 18.1(a)), alternatively each encrypted block can be xor'ed against the plain-text on the next block. Doing so prevents the patterns from showing through the encryption and means that no part of the file can be changed without the resulting file being gibberish. This is what has been done to Figure 18.1(c).



(a) The original image[tux]

(b) Encrypted without chained blocks

(c) Encrypted with chained blocks

Figure 18.1: An image encrypted with and without chained blocks[Wika]

This, however, does not mean that the research is not useful: one can easily decrypt AES with chained blocks in parallel and a server with more than one client can achieve the speed-up by encrypting more than one stream at once.

The real benefit of GPU programming would be for those who are interested in breaking encryption. Due to any encryption scheme being far more difficult to break than to use, the code breaker will need much more computational power than the user of the code.

Indeed that has already been achieved[Rot] using Amazons new GPU cluster instance to bruteforce the value of 16 SHA1 hashes in 49 minutes (at a cost of only $2.10).

3.1.1 Pyrit

Pyrit is a tool designed to speed up password guessing for wireless access points protected by WPA-PSK (i.e. access points where each user authenticates with the same shared key), using both GPUs and CPUs. According to their own benchmarks, they are able to compute 89.000 keys a second using four GeForce 295GTX cards[pro10], while a Core2Duo dual-core 2.5ghz with SSE2 can compute only 1.300 keys a second[pro10].

According to [Ano], on the Amazon GPU cluster instance, Pyrit is capable of computing about 48088.8 keys per second, utilizing two Tesla M2050 GPUs and two quad-core CPUs.

3.2 Simulations

Any simulation that can be run on a CPU can in principle be run on a GPU. The degree to which their speed can be improved, however, depends on the characteristics of the particular simulation: some are interactive, e.g. the heart-surgery simulator described in [MHr], some simulations enable the user to test a large number of possibilities without having to incur the cost of creating them in reality, like the Protein Structure simulation in [Pha05]. Finally, some of the simulations are reconstructions of reality, such as the medical image reconstructor in [Pha05].

As computational power increases, more and more simulations which would previously not have been possible to do in real-time, suddenly become possible. One such example is described in [LLW04] where GPUs are used to accelerate fluid computational dynamics to create a real-time, interactive simulation of realistic flowing water.

Many simulations rely heavily on floating point calculations, and as a result should be improvable by a great deal if written to take advantage of the GPU. Indeed the Molecular Dynamics Simulation in [SFEV$^+$09] claims a speed improvement of 700 times over a single core solution, when run on a GPU.

3.3 Artificial Intelligence

Artificial intelligence (AI) is a very broad subject and the meaning of the term is not precisely defined, but informally it is the subject of making computers do what was previously considered as requiring human thought.

The field of AI makes use of a large number of different techniques. Two of these being used in many different contexts to solve given problems are Genetic Algorithms, which are inspired by natural evolution, and Neural Networks, which are inspired by the networks of neurons in the human brain.

3.3.1 Genetic Algorithms

Genetic algorithms use ideas inspired by natural selection to evolve a solution to a particular problem. They where first proposed by John Holland (see the review in [HR75]).

The process of the Genetic Algorithm can be illustrated as follows [WW09]:

1. Randomly generate a number of possible solutions to the problem we are to find a solution for.
2. Evaluate the fitness of each member, and if the result is good

enough or there are no more computational resources left, terminate.

3. Select the elements which have a high enough fitness that they should be used for the next generation, create the next generation by mixing and mutating these individuals randomly based on the calculated fitness.

4. Start over with step 2.

Since step 2 often involves a large number of individuals whose fitness need to be determined, and the determination of the fitness of each element can be computed independently of the others, genetic algorithms would be an excellent candidate for GPU optimization (i.e. it is embarrasingly parallel[1]). One paper[WW09] describes such optimizations, with speed-ups between 1.16 and 5.30 on the GPU. Further improvement could properly be achieved since the implementation was done without CUDA or OpenCL, and the random numbers were generated on the CPU. Another paper[HB07] shows speed-ups between 0.21 and 34.63 times, but it depends on which benchmarks are used and on the choice of GPU and CPU. Both papers show higher speed-ups as the size of the population increases.

3.3.2 K Nearest Neighbor

The K Nearest Neighbor is a technique useful in solving classification problems. It works by organizing all the identified items into an N-dimensional grid and then, for each of the unknown items located, the k closest identified items, and assign the same classification to the unknown item as the majority of the neighbors have. The K Nearest Neighbor technique is widely used [GDB08], but it is an $O(i * u)$ algorithm[2]. Because of this relatively high amount of computations relative to the amount of data, it is a good candidate for GPU acceleration. This has indeed been done a number of times: [GDB08] got a 120 times

[1] An embarrasingly parallel workload is one for which little or no effort is required to seperate the problem into a number of parallel tasks, usually in cases where there exists no dependency between those tasks.

[2] Where i is the number of $i$dentified and u is the number of $u$nknown items.

increase and [QMN09] got an 88 times increase in performance; others such as [LLWJ09] have also implemented a GPU version of this algorithm.

4
CUDA

NVIDIA's parallel computing architecture CUDA (Compute Unified Device Architecture) was introduced in November 2006.

With the release of the ATI Radeon 9700 in October 2002, pixel and vertex shaders were expanded to implement looping and lengthy floating point math[Wikc]. This indeed made programmable shaders much more flexible than previously, but nonetheless not nearly as geared towards *General Purpose Computing* on the GPU as with the use of CUDA. While each new version of the Shader Model enables even more programmability and features through graphics APIs[Wikb], CUDA and ATI CTM expose the instruction set and memory of GPU hardware so developers can fully leverage this parallel computation engine.

4.1 Platform & Code Portability

The CUDA platform is designed to support various APIs, namely CUDA C, OpenCL, Microsoft DirectCompute and CUDA Fortran[Nvi10a]. Depending on the chosen API, developed GPU code may be compatible with other platforms. The two CUDA-specific APIs, C and Fortran, only support newer NVIDIA GPUs, just like Brook+ from the AMD Stream SDK[1] runs exclusively on AMD hardware.

CUDA C currently dominates the GPGPU programming market de-

[1] In 2006, the ATI company was acquired by AMD

spite being a proprietary and vendor-locked API. While there are obvious reasons to favor an open and cross-vendor solution like OpenCL, NVIDIA continuously promote their API as a better alternative, effectively tying users to their platform and defending market share. By keeping CUDA C ahead of OpenCL, offering the most complete set of libraries and providing developers with free articles and courses, it continues to be the preferred choice for GPGPU programming amongst the majority of developers.[Strb][Mel]

4.2 CUDA C

CUDA C, commonly referred to as just CUDA, is a collection of NVIDIA extensions for the C programming language. It allows the programmer to define kernels: special C functions that are executed a certain number of times in parallel on the GPU. The contents of these kernels are restricted to using a subset of C, as not all operations are supported on the GPU. All memory handling on the GPU device is done manually by the programmer and data is transfered between GPU and host memory using special buffers.[Nvi10a]

4.3 Further Details

OpenCL, which is described in detail in the following chapter, looks and behaves very similar to the CUDA platform.

The most important difference, and also the main reason to choose OpenCL, is that it is a heterogeneous platform. CUDA, on the other hand, is homogeneous, targeting only NVIDIA GPUs.

There are a few differences in terminology as well as some lacking features in OpenCL, but otherwise CUDA and OpenCL roughly look the same.[Stra]

5 OpenCL

OpenCL (Open Computing Language) is an open industry standard for making programs that can be executed on heterogeneous platforms consisting of GPUs, CPUs, and other processors. It is a framework for making parallel programming and includes a programming language, API, libraries and runtime system. Unlike NVIDA's CUDA and AMD's CTM, OpenCL is not bound to any specific hardware.

OpenCL was initially developed at Apple but later a proposal was made in collaboration with NVIDIA, AMD, IBM and Intel. This proposal was sent to the Khronos Group which is a non-profit member founded industry consortium that focuses on creation of open standards of CPU/GPU acceleration. On December 2008 OpenCL 1.0 was released and on June 14 2010 a version 1.1 was made. Recently various hardware manufacturers including NVIDIA and AMD have added OpenCL to their GPU acceleration frameworks.

5.1 Platform Model

As OpenCL is uniform across different hardware and OpenCL provides an abstract platform model. Figure 26.1 shows the OpenCL platform model that consists of a host processor with some local main memory, that is connected to one or more compute devices (CD's). A CD is made up of many Compute Units (CU's) and memory regions as will be explained in subsection 5.3. These CU's are again made up of many Processing Elements (PE's) that are scalar processors running in lock-step.

When relating the platform model to hardware, a host is typically a CPU and a compute device can typically be a GPU, multi-core CPU or other processors like the Cell/B.E. A CU is comparable to a multiprocessor on a GPU or a core in a multiprocessor CPU. PE's are comparable to the smallest core on a GPU that can execute a single thread only On a CPU a PE is comparable to a Arithmetic Logic Unit (ALU). [Coo10]

The host interacts with a device by submitting commands. These commands include execution tasks that are to be executed on the PEs. PEs tasks are executes as Single Instruction Multiple Data (SIMD) or Single Program Multiple Data (SPMD) units where each PE has its own program counter. SPMD are typically executed on general purpose devices such as CPUs while SIMD instructions require vector processors such as GPUs or vector units in a CPU. [amda]

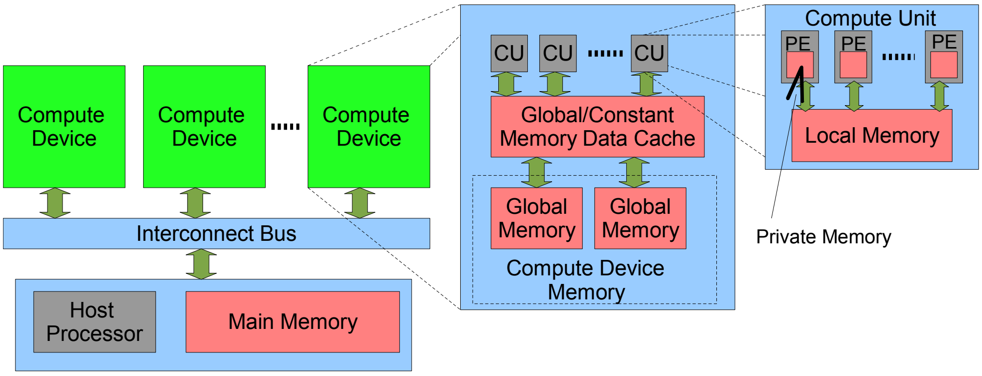


Figure 26.1: OpenCL Platform Model

5.2 Execution Model

An OpenCL application consists of a host program that runs on the host processor and kernels that execute on one or more devices. The host program defines and manages a context in which kernels execute. The host uses this context to contain and manage the state of the program.

An OpenCL kernel is a basic unit of executable code similar a function written in the OpenCL C programming language. As stated above the host interacts with the devices by sending commands to command-queues. These command-queues can contain kernel execution commands, memory commands (transfer or map memory object data) or synchronization commands that constrain the order of commands.

When kernels are sent for execution the host program creates an N-dimensional index space called NDRange where $1 \leq N \leq 3$. Each index point is associated with an execution instance of the kernel. An execution instance in OpenCL is called a work-item and is identified by it's unique global identifier. A OpenCL work-group contains one or more work-items and is identified by a work-group unique identifier. Work-items also have a unique local id within a work-group, therefore work-items can be identified by their global id or by a combination of their local id and work-group item. A work-item executes on a single PE and a work-group executes on a single CU. Relating this to the platform model a work-group a given work-group executes on a single CU while a single work-item executes on a single PE.

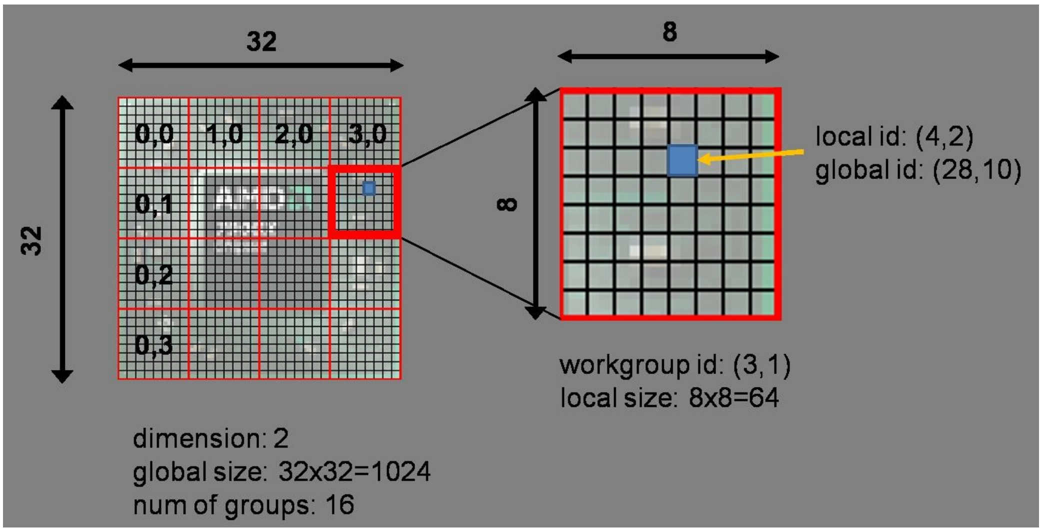


Figure 27.1: two-dimensional OpenCL work-group and work-items [amda]

Figure 27.1 shows a two-dimensional kernel with 16 indexed work-groups. Each of these work-groups includes 64 indexed work-items. The highlighted work-item has a local id of (4,2). It can also be addressed by it's global id by using the highlighted work-group offset of (3.1) by multiplying it with the work-group dimension length and adding the local id.

5.3 Memory Model

OpenCL has four different memory regions that are available to work-items:

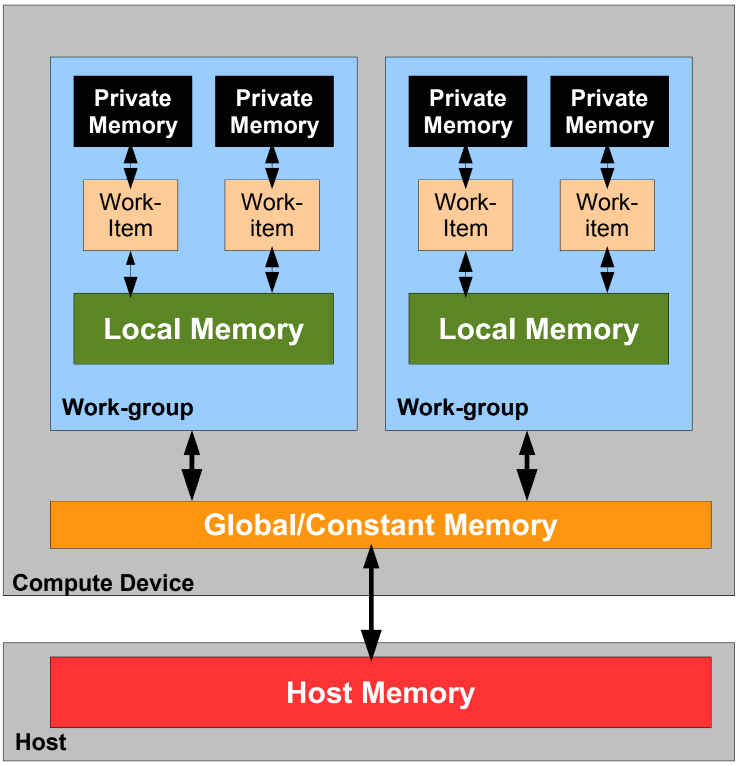


Figure 28.1: OpenCL Memory

- Global Memory: Allows read/write operations to all work-items in all work-groups.
- Constant Memory: Allocated by the host and is constant during the whole execution of a kernel.

- Local Memory: Local to a work-group and shared by work-items.
- Private Memory: Region private to a work-item.

The host memory and device memory are mostly independent of each other, this is because the host program is defined outside of OpenCL and programmed to run on the host CPU. The host program uses the OpenCL API to allocate memory objects to the device global memory. The host program then interacts with the device in two ways, by either explicitly copying data or by mapping and unmapping regions of a memory object.

To copy data explicitly the host needs to enqueue commands to transfer data between host memory and device memory. These transfers can be either blocking or non-blocking. In the case of a blocking transfer the function call returns when the used resources are safe to reuse. In the case of a non-blocking transfer the function call returns as soon as the command is enqueued on the device regardless if the host memory is safe to reuse.

For the mapping/unmapping method the host allows memory from the memory object to be mapped into its address space. The host can then do read/write operations on this region and unmap the region when it is done using it.

OpenCL defines a relaxed memory consistent model i.e. the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times [ope]. To ensure consistency OpenCL allows synchronization points to be defined in kernels. These synchronization points are called work-group barriers and when a work-item executes a barrier it halts and waits until all other work-items in the work-group have executed the barrier.

5.4 Program Example

Due to the fact that even a small OpenCL program is very large codewise we will show pieces of the five main steps that are needed to construct a OpenCL program. The five steps are as listed below:

- Initialization
- Allocate resources
- Creating programs/kernels
- Execution
- Tear Down

5.4.1 Initialization

```
 1  cl_int ciErr;
 2  cl_context cxGPUContext;
 3  cl_device_id cdDevice;
 4  cl_command_queue cqCommandQueue;
 5
 6  clErr = clGetPlatformIDs(1, &cpPlatform, NULL);
 7  cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL,
 8          NULL, ciErr);
 9  cqCommandQueue = clCreateCommandQueue(cxGPUContext,
10    cdDevice, 0, &ciErr1);
```

Listing 30.1: Initialization

First when making a OpenCL program you need to define and initialize the variables that are to be used. The first line in the listing defines an integer to hold potential error messages. In line six *clErr* gets assigned by *clGetPlatform* that is a query function that returns *CL_SUCCESS* if the function executed succesfully, otherwise it will output one of two error codes that are either *CL_INVALID_VALUE* or *CL_OUT_OF_HOST_MEMORY*. The first argument is the number of platform ids that can be added to *platforms*. The second argument returns a list of platforms found. The third argument returns the number of available OpenCL platforms that are available.

Line two defines a OpenCL context and on line seven-eight *cxGPUContext* gets assigned by *clCreateContext*. The first argument *clCreateContext* is a list of platforms and in this case zero to make it implementation defined. The next argument is the number of devices. Argument three is a pointer to a list of unique devices. Argument four is a callback function to report potential errors and in this case it is defined as *NULL* meaning that no function is defined. Argument five will be passed with argument four, but since argument four is *NULL* argument five is also *NULL*. Argument six is the error code that is defined on above. Line three defines a OpenCL device that is to be associated with a given context. Line Four defines a command-queue that needs to be associated with a specific context and device. On line nine-ten *cqCommandQueue* gets assigned by *clCreateCommandQueue*. Arguments one and two are described above. Argument three specifies a bit-field list of command-queue arguments that includes in-order or out-of-order executions of commands and if profiling is enabled. In this case zero stands for out-of order disabled and the second bitfield is unspecified and profile is by a minimum always enabled.

5.4.2 Allocate Resources

```
1 cl_mem ax_mem = clCreateBuffer(context,
2   CL_MEM_READ_ONLY, atom_buffer_size,
3   NULL, NULL)
4
5 ciErr1 = clEnqueueWriteBuffer(cqCommandQueue,
6             cmDevSrcA, CL_FALSE, 0,
7             sizeof(cl_float) * szGlobalWorkSize,
8             srcA, 0, NULL, NULL);
9 clFinish(cmd_queue);
```

Listing 31.1: Allocation

To be able to send information to your device you need to define a buffer that will be able to hold your elements. These elements can be of scalar types, vectors or user-defined structures.

In listing 31.1 a memory buffer object *ax_mem* being defined and initialized in line 1-3. This object is created using the *clCreateBuffer* function that returns a non-zero buffer object and a error code to indicate success or not. This function takes a context as its first argument and the second argument is a bit-field used to specify how this memory object can be used by kernels. Third argument specifies how much bytes of space to be allocated. Fourth argument can be used to point to a already allocated memory buffer. Fifth argument can be used to return a appropriate error code.

On line five the function *clEnqueueWriteBuffer* is used to enqueue commands to be written from the host memory to a buffer object on the device. *clEnqueueWriteBuffer* returns a int error code to indicate if it succeeded or not.

5.4.3 Creating programs/kernels

```
cl_program cpProgram;
cl_kernel ckkernel;
cpProgram = clCreateProgramWithSource(cxGPUContext,
              1, (const char **)&cSourceCL,
              &szKernelLength, &ciErr1);

ciErr1 = clBuildProgram(cpProgram, 0, NULL, NULL,
           NULL, NULL);
ckkernel = clCreateKernel(cpProgram, "VectorAdd",
             &ciErr1);
```

Listing 32.1: Kernel Creation

Listing 32.1 is the creation of the host program and Kernel from source. A program object *cpProgram* is created with a given context and loads the source code specified by *&cSourceCL*. After the loaded source-code gets compiled with the function *clCreateKernel*.

At last a kernel object is created with the *clCreateKernel* function that takes the compiled *cpProgram* kernelname and error code as argument.

5.4.4 Execution

```

size_t global_work_size, local_work_size;

```

```
 local_work_size = n;
 global_work_size = m;

err1 = clSetKernelArg(ckKernel, 0, sizeof(cl_mem),
        (void*)&cmDevSrcA);
err1 = clEnqueueNDRangeKernel(cqCommandQueue,
        ckKernel, 1, NULL, &szGlobalWorkSize,
        &szLocalWorkSize, 0, NULL, NULL);
```

Listing 33.0: Execution

In listing 32.2 dimensions of the work-items are defined where global work-size is the collection of all work-items and local work-items is their range in a work group.

Before a kernel can be executed it's arguments need to be set. Kernel arguments are set by using the API function *clSetKernelArg* that takes as input a kernel, index of argument, size of argument and a pointer to argument value. Now we are ready to enqueue a command to execute the kernel on a device, this is done with the *clEnqueueNDRangeKernel* API function.

5.4.5 Tear Down

```
1
2 ciErr1 = clEnqueueReadBuffer(cqCommandQueue, cmDevDst,
3           CL_TRUE, 0,sizeof(cl_float) *
4           szGlobalWorkSize, dst, 0, NULL, NULL);
5 clReleaseKernel(ckKernel);
6 clReleaseProgram(cpProgram);
7 clReleaseCommandQueue(cqCommandQueue);
8 clReleaseContext(cxGPUContext);
```

Listing 33.1: Tear Down

After doing the calculations on the device results need to be written to host memory from device memory, this is done with the **clEnqueueReadBuffer** function. Finally memory can be released by using the functions in lines four to seven.

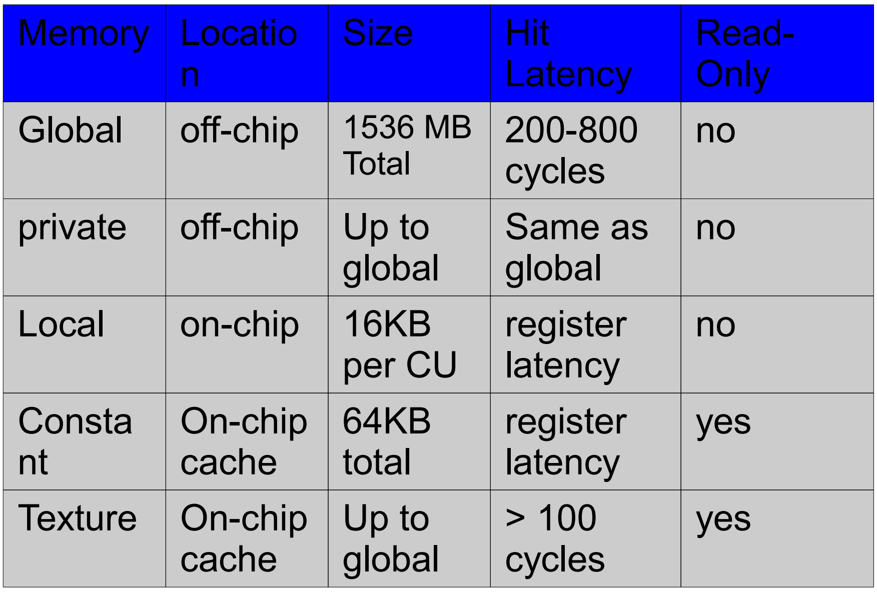
6
GPU and Utilization

Since GPGPU programming is fairly new and the programmer has to do most of the memory management himself we need to take a closer look at the actual GPU that we have available to help us in making code that utilizes the hardware the most. Since CUDA and OpenCL terminology is almost the same and we already have listed most of the OpenCL terminology in section 5 we are going to describe this chapter in OpenCL terminology unless explicitly mentioned.

We have in our possession two Nvidia Tesla C870 GPUs that are the first to be made in the Tesla series. Tesla C870 is build on the G80 line of GPUs from Nvidia and is similar to the well known GeForce 8800GTX, but with larger memory and no monitor connection-ports as the Tesla series are intended for GPGPU.

The C870 GPU has 128 thread processors (128 PEs in OpenCL terminology) where each of these operate at 1,35 GHz and combined are able to perform over 500 single-precision GFLOPS [Nvi08]. It has 1,5 GB dedicated GDDR3 memory running at 800 MHz, 64KB of constant memory and 8096 registers for holding instructions.

The C870 has a Compute Capability (CC) of 1.0 where devices with the same major revision number are of the same core architecture and the minor revision number corresponds to incremental improvement to the core architecture, possible introducing new features like double precision floating arithmetic. The most notable differences between the 1.x and the new 2.x is that each CU has 32 PEs instead of the 8 that our GPU has and two warp schedulers instead of one. [NVI10c].



Memory	Location	Size	Hit Latency	Read-Only
Global	off-chip	1536 MB Total	200-800 cycles	no
private	off-chip	Up to global	Same as global	no
Local	on-chip	16KB per CU	register latency	no
Constant	On-chip cache	64KB total	register latency	yes
Texture	On-chip cache	Up to global	> 100 cycles	yes

Table 35.1: C870 Memory Properties

6.1 Utilization

CPUs and GPUs are architectural different as explained in 2.2. GPUs from NVIDIA and AMD have different architectures and furthermore even GPUs from the same vendor are different. OpenCL is heterogeneous and works across different architectures but the performance will wary depending on which device it is run on. To be able to get maximum throughput you may need to optimize your code for the individual architectures. Since our GPU has 1.0 compute capability as mentioned in 6 this section will be targeted at this particular architecture.

To utilize a GPU [NVI10c] lists 3 different performance strategies

- Maximize parallel execution to achieve maximum utilization;
- Optimize memory usage to achieve maximum memory throughput;
- Optimize instruction usage to achieve maximum instruction throughput.

On what performance strategy to use depends on what part of your code doesn't perform optimal, therefore optimizing instruction usage of a kernel when it most likely is memory accesses that are the problem won't result in significant performance gains.

6.1.1 Maximum Parallel Execution

To maximize utilization the program should be structured so that it exposed as much parallelism as possible and efficiently maps this parallelism to the various components to keep them busy at all times. At the application level it should maximize parallel execution between the host, the device and the bus connecting the host to the device. Serial workloads should be run on the host while parallel should be run on the device.

At device level the application should maximize parallel execution between the CUs of a device. For CC.x only one kernel can execute on a device at a time, therefor there should be at least as many work-groups as there are CUs. Devices that are CC 2.x can have multiple kernels executing on the same device, so to get maximum utilization multiple kernels can be queued and run on the same device at the same time.

At the lowest level parallel execution between PEs should be maximized. This means having enough work-items running to keep as many as possible PEs occupied at all times, this should be achieved by having enough warps ready to execute with zero latency.

A warp is not explicitly defined in OpenCL or CUDA but it is useful to know about them as they are used in memory transfers. When a CU is given a work-group they are partitioned into warps of 32 work-items that are aligned by their local id. A warp gets scheduled to a warp-queue by a warp-scheduler as seen in fig 39.1.

According to [NVI10b] and [RRB$^+$08] the the number of work-items in a work-group should be minimum 32 or of a multitude of 32. According to [Kan10] the work-group size should be a multiple of 64 instead of 32 if you are using a AMD GPU, therefor trying a multiple of 64 should be a best practice, as using 64 is optimal for both types of GPUs while the opposite is not.

An SM can perform zero-overhead scheduling to interleave warps and hide the latency of global memory accesses and arithmetic operations. When one warp stalls, the CU can quickly switch to a ready warp resident in the SM. Up to 8 work-groups can run on per CU at any one time [RRB$^+$08]. The common reason that a warp is not ready to execute its next instruction is when its input operands are not yet available and needs to wait some cycles to get access to them.

Execution time varies depending on the instructions but with device compability 1.x, all work-items in a warp the scheduler needs to issue the instruction over 4 cycles for a single-precision floating-point operation [NVI10c]. If the operands are off-chip, memory latency is around 400-800 cycles.

The number of warps required to keep the warp scheduler busy during such hight latency periods depends on the kernel code, but in general more warps are required if the ratio of off-chip memory instructions is high. As an example: if the ratio is 10 then to hide latency of 600 cycles, around 16 warps are required for a CC 1.x device.

6.1.2 Memory Throughput

The first step to maximize memory throughput is to minimize data transfers with low bandwidth, this means minimizing transfers between host and device and furthermore minimize transfers between global memory and the device by maximizing use of on-chip memory and local memory.

Minimizing transfers between host and device is important because of the 86,4 GBps on-chip memory compared to 8 GBps on a PCI Express $*16$ Gen2 [NVI10b] one way of doing this is to move code from the host to the device to execute, even though that means running a kernel with low parallel computations. This should only be done if your code has problems hiding memory latency as it helps keeping the PEs busy.

Batching many small memory accesses into one large transfer is better for throughput than doing many small memory accesses. Therefore accessing global memory should be done with one or more coalesced accesses. When more than one work-items in a running warp executes an instruction that accesses global memory, it coalesces all the global memory accesses from these work-items into one or more memory accesses.

Global memory supports words of size equal to 1,2,4,8,16 bytes. If accesses to global memory are not equal to these words or not aligned (starting address is a multiple of that size) the access does not get coalesced correctly and instead gets fragmented into many accesses degrading performance. Luckily OpenCL built in types already have this requirement fulfilled.

The memory model is very similar to OpenCLs 28.1 but added texture memory.35.1.

Private memory scope is local to a single work-item but it is off-chip so it has the same latency and low bandwidth as global memory. Private memory is used only to hold automatic variables, in case of insufficient register space [NVI10b].

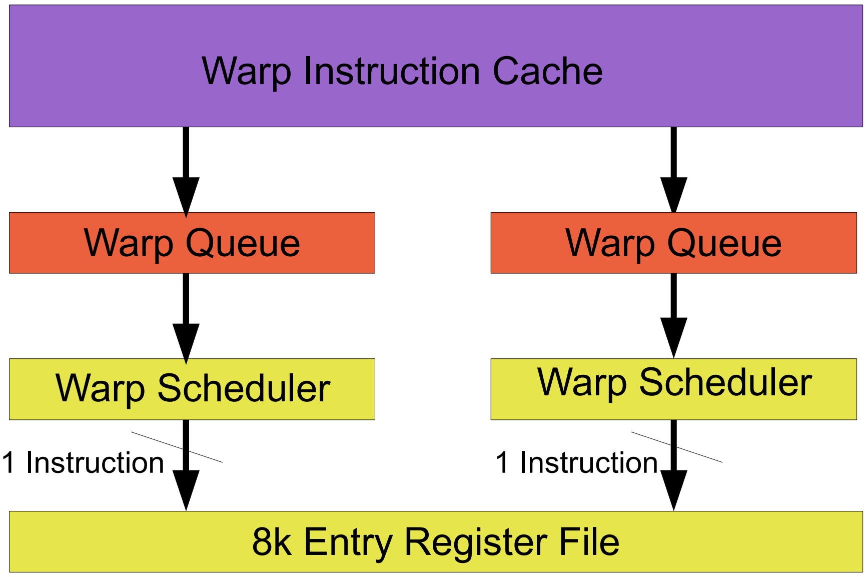


Figure 39.1: Warp Scheduler in Interleaving Mode

Local memory is on-chip and therefore is much faster than global and private memory. Local memory is organized into equally sized memory modules, called banks that can be accessed simultaneously. Therefore any access to local memory that spans over n distinct banks results in n times the bandwidth of a single bank access. However, if multiple addresses request access to the same memory bank a conflict occurs and they have to be serialized. Exception to a bank conflict is if all threads in a half-warp (16 work-items) adress the same memory bank, resulting in a broadcast. Ways to make sure that your code minimizes bank conflicts is by using 32-bit Strided Access [NVI10c] and 32-bit Broadcast Access[NVI10c].

Constant memory is on-chip, is cached and has a total of 64KB. Reading from constant memory only requires off-chip read if there is a cache miss. If for 1.x devices, all work-items in a half-warp want to access the same memory address it only takes one memory read.However if constant memory address requests are different for some work-items they have to be serialized resulting in a reduced throughput.

Texture memory is like constant memory also cached on-chip but optimized for 2D spatial locality, so that work-items that have texture addresses close to each other will gain throughput. Texture memory is usually preferred over global memory if the memory reads do not follow the access patterns for coalesced memory reads.

6.1.3 Instruction Throughput

To get maximized instruction throughput a minimize use of instructions with low throughput and minimize diverging warps should be used.

If it doesn't affect the end result arithmetic instructions can trade precision for speed by using *native_*∗ [NVI10c] instead of regular functions and trading single-precision over double-precision.

Branching or control instruction (if, switch, do, for, while) can really lower instruction throughput as all PEs in a CU run in lock-step one instruction at a time. This means that whenever work-items in a warp diverge, every diverging path needs to be serialized requiring more instructions for the same warp.

When the control flow depends on the work-item ID, the code should be written to minimize diverging warps, this can be controlled as the programmer knows which work-items are withing what warp.

Theirs not to make reply,
Theirs not to reason why,
Theirs but to do and die:
Into the valley of Death
Rode the six hundred.

Alfred, Lord Tennyson. Charge of the Light Brigade

7 Programmer Productivity

The speed at which a program runs is only one of the things that must be considered to evaluate different technologies. Another very important consideration is the productivity at which the technology can be used. For example the virtual machines that is used to run programs written in languages like Ruby[1] or Python[2] involve an overhead (either a memory overhead if a JIT compiler is used, or an overhead to interpret bytecode if it is not) that is not present in a version of the program that is written in C. Ruby and Python are used because of the, in all likely-hood correct, assumption that the same program can be written much faster in Ruby or Python than in C.

While we can quantify the difference between the time it takes two different programs to execute, measure how much memory they each use, and so on, it is much more difficult to measure how much effort it took to write them.

Various techniques have been used as a proxy for the complexity of writing a computer program:

1. Lines of Code (sometimes Source Lines of Code, not counting blank lines and comments)
2. Function points
3. Time spend writing it

[1] `http://ruby-lang.org`
[2] `http://www.python.org`

There is nothing wrong with these measures per-see, a program with n lines of code is likely to have been written faster than a program with $20 \times n$ lines of code, but this assumes that the developers writing the code has about the same skill, understanding of the problems involved, the techniques and technology used to solve it, etc.

Function points[Jon94, JA] are a different way to estimate the complexity of a software program: here one takes into account what the program must do, which external data it works with, what it must do with this data, etc and uses this to sum up how many points each of these things take. It does not take into account the skill or familiarity a particular programmer may have with the system, nor does it account for how much code is necessary to express a function point. An additional problem is that two people, even if both have the same training and experience in the same organization, may estimate the function points for a given specification with as much as 30% difference[Kem93], but even if the program source code is available, the estimates of its complexity may differ with as much as 10%[Kem93]. This indicates that function points, even if it takes the same time to program the code necessary to create two different function points, is not an accurate or reliable means of computing the complexity of a program nor is it a good measure to calculate the costs of developing a program, despite what [JA] has done.

Item number 3 is properly the simplest way to estimate the complexity of the program, but it suffers from the same problem as the line of code metric, in that it depends a lot on the developer who writes the code and it can only be used after the program has been written.

This means that these ways to measure programmer productivity are only able to give a very rough suggestion as to the actual complexity of the task, and the numbers are not useful to quantify the difference.

One of the problems is that we do not have a good model for judging how programmers develop software.[FRGMS07] has done some empirical research on how to model programmer productivity, and while his data-set is very small and his model for programmer productivity simplified, it is still useful as a good starting point for an actual model for how programmers work. His model can be found in Figure 44.1.

[FRGMS07] uses timed Markov models to model the process of devel-

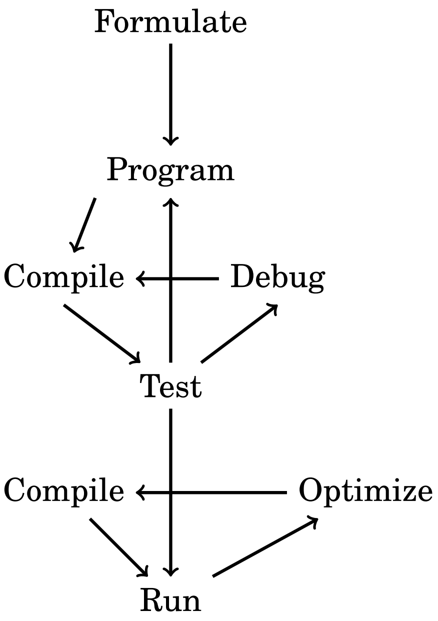


Figure 44.1: Basic overview of the observed process of developing a program for high performance computing[FRGMS07], expressed as a (Timed) Markov Model, without the probabilities. These can be seen in Table 44.1.

oping software for high performance computing, but there are several problems with the model:

1. The data has been collected based on very few samples.
2. Those were students, not professionals who had extended experience with these tools.
3. As presented, we cannot know how many times the developers where in each state, nor do we have access to data to answer questions such as "what is the probability that a program works after the programmer has debugged it 5 times?" which is almost certainly not going to be the same as the probability that the program works after the programmer has debugged it once; Instead we only have the average probability that the program worked.

Of these the most important is Item 1 which means that we cannot use

Current state	Next State	Chance of moving
Test	Program	0.237
Test	Debug	0.713
Test	Run	0.048
Run	Optimize	0.699
Run	*Done*	0.035

Table 44.1: The various probabilities for the transit between different states as seen in Figure 44.1. These numbers can be seen in [FRGMS07, page 4]

the actual numbers to compare with anything, as there simply is not enough data points to have any statistical confidence in the results.

Item 3 is the most important, however. As the Google Tech talk by one of the authors of [FRGMS07] points out[1], if we can create a tool that speeds up the time it takes to fix each bug, but which still requires us to go through the same number of debug-run-debug cycles, at most we can expect a linear improvement in programmer productivity, whereas if we can cut down on the number of times the developer has to go through the cycle, we can obtain a super-linear speedup. Since programmers spend most of their time debugging[FRGMS07] optimizing this step is extra important.

Though we do not have the actual numbers, we can write the equation for the total time taken up by debugging as a recursive equation (where P_i is the probability that debug attempt i is necessary and Time_i is the time taken by debug attempt i):

$$T_0 = \mathrm{Time}_o + \mathrm{P}_1 \times T_1 \tag{46.1}$$

$$T_i = \mathrm{Time}_i + \mathrm{P}_{i+1} \times T_{i+1} \tag{46.2}$$

With this knowledge, we can now design our system such that the risk of having to debug it is lessened.

[1] Available at `http://www.youtube.com/watch?v=Zbmd85p981o`

To raise new questions, new possibilities, to regard old problems from a new angle, requires creative imagination and marks real advance in science.

Albert Einstein

8 Alternate paradigms

Because the OpenCL Kernel language is a variant of C, it is possible to write any program that can be written in it, but doing so is not necessarily the most efficient way to program a GPU, because C is not originally designed to express parallel computations[1].

We will therefore look at a number of different paradigms that may be better suited to express the parallelism in the problems which we wish to compute on the GPU.

The first of these paradigms we will look at is the MapReduce paradigm.

8.1 MapReduce

MapReduce[DG04] is a concept and the name of the related software invented by Google as a way to abstract computation of terabytes of data across large clusters of unreliable commodity computers, such that the person who writes the software does not have to worry about reliability, redundancy or how to distribute the tasks between the different computers[DG04].

MapReduce is proprietary software and it relies on other software, in-

[1] C was designed as a language to implement Unix in, without tying it down to a particular instruction set, though it can be, and has been, used for many other applications as well

ternal to Google, such as GFS[DG04], but the idea behind MapReduce has been implemented in the open source software Hadoop[had], available from the Apache foundation. As such, the examples we provide here are based on api of the open source implementation, not the proprietary implementation.

MapReduce is based on the idea from functional programming[DG04] of the `map` and `reduce` functions. Map is a function that takes as argument a function and a list of values to apply this function to, then returns a new list where the ith element is the result of applying the function to the i input element. Assuming that the function does not have any hidden state (which is often the case in functional programming), it does not matter in what order the function is actually run, nor does it matter if it is run more than once on some of the input, as long as there is at least one result for each of the input elements.

An example of a (trivial) use of the map function in Haskell:

```
double = 2 *
map double [1 2 3 4 5 6 7]
```

Listing 48.1: A simple use of the map function: Creating a new list where the i element is $2 * i$.

`Reduce` is similarly a function which takes a function as an argument, and a list of arguments to run this function on. Unlike the `map` function, `reduce` typically returns fewer arguments than it is given (in at least one of the examples in [DG04], the reduce function is simply replaced with the identity function and in the MapReduce system in [HFL+08], the reduce step is optional).

```
foldr (+) 0 [1 2 3 4 5 6 7]
```

Listing 48.2: A simple use of the foldr function (Haskells name for reduce): Summing the numbers from 0 to 7.

One of the simplest examples of MapReduce is the word counting problem [DG04, HFL+08, yah], where the objective is to count the number of times a word appears in some, usually very large, text.

```
public static class Map
    extends Mapper<LongWritable, Text, Text,
                   IntWritable> {
```

```
      private final static IntWritable one =
          new IntWritable(1);
      private Text word = new Text();
      public void map(LongWritable key, Text value,
                      Context context)
        throws IOException, InterruptedException {
          String line = value.toString();
          StringTokenizer tokenizer =
              new StringTokenizer(line);
          while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
          }
      }
}
```

Listing 49.0: A mapreduce example: Word counting [yah].

```
public static class Reduce extends
    Reducer<Text, IntWritable, Text, IntWritable> {
  public void reduce(Text key,
                     Iterable<IntWritable> values,
                     Context context)throws
    IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      context.write(key, new IntWritable(sum));
    }
}
```

Listing 49.1: A mapreduce example: Word counting (reduce part)[yah].

Other examples are Distributed Grep, where the objective is to find the lines in some text that contains a given word, counting requests from log files, computing inverse Web-Link graphs, inverted indexes or the distributed sort[DG04]. The distributed sort is one example of a situation where the function for the reduce step is the identity function.

While the MapReduce algorithm was originally meant to be used to divide work between large clusters of commodity machines, it has also been used to schedule work on a single multi-core computer[RRP+07]; finally there have been a number of implementations of MapReduce for the GPU, both Mars[HFL+08] and Merge[LCWM08]. The latter is of particular interest because it uses an entirely new language, developed for this purpose, and aims at making it easier to program GPUs (as well as other architectures, such as the Cell processor) for programmers who are not accustomed to traditional GPU programming. MapCG[HCC+10] deserve a special mention as well, because it aims to bridge the gap between CPU and GPU programming by allowing the programmer to use the exact same source code both on the GPU and CPU and still make use of the MapReduce concept.

As useful as MapReduce is as an abstraction, there are many situations in which it is not appropriate and would be of very little use. One of the most important of these is solving equations by iteration – these are well suited the be computed by the GPU, since the same data is used to compute many times, which means that the price paid for sending the data from main memory to the memory on the GPU have to be paid only once, but the benefit of faster computation can be reaped many times.

8.2 Aparapi

Aparapi is a completely new technology, announced by AMD on October 13 and currently available only as an alpha release, which allows programmers to write code that can be run on the GPU entirely in Java code, and which relieves them from having to manually allocate memory on the GPU and schedule which work items should be run when. Aparapi is meant to run on top of OpenCL and keep the programmer from having to worry about the details of the underlying software.

On the face of it, Aparapi is a promising new technology, but at least currently it has a lot of limitations regarding what the programmer can do in the kernels, and therefore doesn't live up to the promises of freeing the programmer from having to learn the differences between the style of development that they are used to and the way to program GPUs, since the code they write in Java that is supposed to run on the GPU is very limited. In particular the following can't be done[Amd10]:

1. More than one dimensional arrays.
2. The .length field of an array can't be used, and as a consequence an array can't be used in a foreach loop
3. Static methods are not supported
4. Overloaded methods with different arguments are not supported
5. No support for exceptions, throw, catch, etc
6. No new objects can be created
7. Neither the switch, continue or break statements can be used
8. No methods from the Java class libraries can be used, except for java.lang.Math – which are reimplemented to use the GPU instructions.
9. In order to get good performance, the programmer still have to take the underlying limitations of the hardware (no protection against division by zero to name one example).
10. The `double` and `char` data types can't be used.

Should the programmer do any of this (or violate any of the other restrictions, the list is not complete) Aparapi will run the kernel on the CPUs instead, which means that the programmer will not reap the benefit of the faster floating point operations, but some benefit may still be achieved, assuming that the program is executed on a computer with more than one processor.

While it is very early to say, this is properly the biggest issue with Aparapi — one does theoretically write in Java, but almost everything a programmer would do in Java can't be done in Aparapi, but could be done if one used OpenCL directly; at level of abstraction which is offered by Aparapi, the programmer is properly better of learning enough C to write the kernel directly.

Aparapi is very early alpha software, and some of these restrictions seems very likely to be removed in the future. In particular, there is no reason that the .length field of the arrays couldn't be sent as extra arguments to the code that the Java kernel is compiled down to, the compiler need then only replace the reads from the length field with reads from this field. OpenCL already have 2- and 3-dimensional arrays, so again there is no reason not to allow the developer to use these in Aparapi.

8.2.1 Merge Framework

Merge[LCWM08] is a framework for the development of software to run on the GPU. Like Mars it uses the MapReduce paradigm, but unlike Mars it is able to automatically schedule different work items to be run on the available hardware, including different CPU cores, GPUs as well as any Cell processors that may be available. Unlike both Mars and Aparapi, it doesn't attempt to use an already existing language, but instead uses a language specifically designed to be used for this purpose.

Designing a new language is both a big advantage – since it can be, and has been, tuned to the specific requirements of the environment under which it is supposed to run – and a big disadvantage, since it requires that every programmer who wishes to make use of this system have to learn a new language (this is the exact situation which Aparapi wants to prevent by making it possible for the programmer to write his code in a language he is already familiar with).

Nevertheless, there is a lot to be gained from writing a specialized programming language. A big advantage is that it makes it possible steer the programmer towards writing programmers in such a way that they can take advantage of the specific advantages offered by the OpenCL and steer the programmer away from those features which makes the program slower. The creators of Aparapi, having chosen Java as their implementation language, didn't have this opportunity and as a result anybody who uses it runs the risk of writing a program which seems to be fine, but can only run on the CPU (an as such isn't able to take advantage of the extra computational power of the GPU).

Listing 52.1: An example of the language used in the Merge framework

```
bundle km0 {
void kmdp(Array2D<float> dp,
Array2D<float> cc,
Array1D<int> assgn,
Array2D<float> ccnew,
Array1D<int> hist) {
// Assign observations and compute new cluster
// centers for next iteration in parallel
mapreduce(int i=0; i < dp.rows; i++)
kmdp(dp[i],cc,assgn[i],
red<sum>(ccnew),red<sum>(hist));
}};
```

One of the benefits of OpenCL is that the code that is written can be run both on the CPU and on a GPU, but the Merge framework language isn't designed to do that, rather the idea behind it is that one can write a program gives the same results, but which can be run on different architectures and can be written in a way that takes advantage of benefits of a particular architecture.

Finally, Merge Framework is able to dynamically (that is, at run time) schedule which tasks should run on the available compute devices. This is especially useful for OpenCL, because it can use both the CPU and GPU to run tasks on, as well as any Cell-processors that may be attached to the host machine (CUDA in contrasts only works with a GPU) – as a result, a host machine nearly always have more than one OpenCL enabled device (the only machine that doesn't would be a machine with only one CPU with only one core, which is increasingly rare, and either no GPU or a GPU that does not support OpenCL).

8.3 hiCUDA

[HA09] is an entirely different approach to efficient GPU programming than the various MapReduce inspired frameworks. hiCUDA is inspired by OpenMPI[1]. This is a smart because researchers working with high performance computing are used to working with clusters and they quite often program these with MPI and OpenMPI. [CR10] evaluated

[1] General introduction to MPI can be found in [GGKK03], OpenMPI is an extension of and implementation of MPI with some special directives which aims to make it easier to program in.

several different approaches and found that OpenMPI was both faster to program and more efficient than alternatives that are better known outside the high performance computing area, such as pthreads[1].

An example of hiCUDA code used to calculate the matrix resulting from the multiplication of a 64×128 with a 128×32 matrix can be seen in Listing 54.1. Compare this with the same code for a sequential matrix multiplication (Listing 54.2).

```
float A[64][128];
float B[128][32];
float C[64][32];

//Randomly init A and B.
randomInitArr((float*)A, 64*128);
randomInitArr((float*)B, 128*32);
//C = A * B
for(i=0;i< 64; ++i){
  for(j=0;j<32;++j){
    float sum = 0;
    for(k=0; k< 128; ++k){
      sum += A[i][k]* B[k][j];
    }
    C[i][j] = sum;
  }
}
printMatrix((float*)C, 64, 32);
```

Listing 54.2: Sequential matrix multiplication code from [HA09].

This illustrates that surprisingly little code has to be added to make it possible to GPU accelerate the program, and while the changes does require some knowledge of GPU programming (it is necessary, for example, to know how much memory each multiprocessor has and consequentially how much to copy from global memory at a time) but even so hiCUDA has been used in at least one case where it took only 4 week to develop the solution, whereas the same solution implemented in CUDA took 3 months[HA09]. At the same time benchmarks show that the speed of a hiCUDA program is within 2% of the speed of the

[1] Phreads gives a much larger degree of freedom for the programmer, but this also means that it takes more effort[CR10].

```
  float A[64][128];
  float B[128][32];
  float C[64][32];

  //Randomly init A and B.
  randomInitArr((float*)A, 64*128);
  randomInitArr((float*)B, 128*32);
#pragma hicuda global alloc A[*][*] copyin
#pragma hicuda global alloc B[*][*] copyin
#pragma hicuda global alloc C[*][*]
#pragma hicuda kernel matrixMul tblock(4,2) thread (16,16)
  //C = A * B
#pragma hicuda loop_partition over_tblock over_thread
  for(i=0;i< 64; ++i){
#pragma hicuda loop_partition over_tblock over_thread
    for(j=0;j<32;++j){
      float sum = 0;
      for(kk= 0; kk <128; kk += 32){
#pragma hicuda shared alloc A[i][kk:kk+31] copyin
#pragma hicuda shared alloc B[kk:kk+32] copyin
#pragma hicuda barrier
        for(k=0; k< 32; ++k){
          sum += A[i][kk+k]* B[kk+k][j];
        }
#pragma hicuda barrier
#pragma hicuda shared remove A B
      }
      C[i][j] = sum;
    }
  }
#pragma hicuda kernel_end
#pragma hicuda global copyout C[*][*]
#pragma hicuda global free A B C
  printMatrix((float*)C, 64, 32);
```

Listing 54.1: hiCUDA matrix multiplication program from [HA09]

same program, written directly in CUDA.

8.4 Accelerator

Accelerator is a program and API made by Microsoft Research in C# to allow programmers to use the GPU in their code, without specified knowledge of how the graphics pipeline works[TPO06].

8.4.1 An example of the use of Accelerator

```
  using Microsoft.Research.DataParallelArrays;
  static float[,] Blur(float[,] array, float[] kernel)
  {
    float[,] result;
    DFPA parallelArray = new DFPA(array);
    FPA resultX = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++) {
      int[] shiftDir = new int[] { 0, i};
      resultX += PA.Shift(parallelArray, shiftDir) * kernel[i];
    }

    FPA resultY = new FPA(0f, parallelArray.Shape);
    for (int i = 0; i < kernel.Length; i++) {
      int[] shiftDir = new int[] { i, 0 };
      resultY += PA.Shift(resultX, shiftDir) * kernel[i];
    }
    PA.ToArray(resultY, out result);
    parallelArray.Dispose();
    return result;
}
```

Listing 56.1: An Accelerator example of bluring from[TPO06].

Unlike programming languages such as C, Accelerator has the data-parrallel array, rather than a scalar value, as its basic datatype[TPO06].

The reason for this is, of course, that Accelerator is designed to make it easier to run code on the GPU, which is only an advantage if it is possible to exploit the parallel nature of the GPU.

If we compare the example in Listing 56.1 with the hiCUDA example (in Listing 54.1), a number of things are immediately obvious:

1. With Accelerator we tell the computer what to do, with hiCUDA we tell the computer how to do it.
2. As a consequence hereof, we expect that we will have to know a great deal more about the hardware which the code is supposed to run on to use hiCUDA, whereas one of the goals of Accelerator was to free us from this problem.
3. If we know enough about the underlying hardware, it is likely that the code created with Accelerator is not going to run as fast as if we wrote it with hiCUDA.

Of these 3 observations, observation 3 and observation 2 are the most interesting points, since they hint at the classic trade-of: languages which offer a higher abstraction is likely to be less efficient, if used by a programmer with deep knowledge of the system, than lower level languages but they are also likely to be faster to develop the program in.

The benchmarks in [TPO06] give some evidence of this trade-of: whereas the benchmarks in the hiCUDA paper[HA09] put them within 2% of the corresponding CYDA code, Accelerator only gives about 50% of the speed the hand-tuned code runs at; this number may have improved since [TPO06] as written, as it predates both CUDA and OpenCL and its code-generating capabilities is therefore limited to what can be expressed with shaders. In addition the benchmarks in the two papers similar but not equal, which makes direct comparison difficult.

Microsoft research is working on the next version of Accelerator[Res], which may improve these benchmarks and will be able to take advantage of CUDA.

I learned quickly, as I tell my graduate students now, there are no answers in the back of the book when the equipment doesn't work or the measurements look strange.

Martin Lewis Perl

9 Benchmarks

To be able to meaningfully talk about speed and productivity, we need a way to measure both the productivity of the programmers who does the work (we talk about this in Chapter 7 on page 43) as well as the trade of in terms of speed of the final application.

Unfortunately while benchmarks are easier to make than programmer productivity is to measure, they are still far from straight forward. Part of the reason for this is that there are so many different benchmarks to choose from and not everybody uses the same benchmarks and those benchmarks does not necessarily measure the same thing.

This does not mean that all benchmarks are equally important; one of the important benchmarks in high performance computing is the linpack benchmark, which is used to determine the TOP500 list of the worlds fastest supercomputers[Fat09]. Unfortunately the linpack benchmark is not very well suited for our purposes for three reasons:

1. It tests floating-point computations[Fat09], which we already know are fast on a GPU.
2. It uses double-precision floating point numbers, which the Tesla cards we have available cannot compute[1].
3. The benchmark consists of only one kind of computation, which consists of solving a number of linear-equations expressed as a

[1] The Tesla cards we have access to have compute capability 1.0, which, among other things, do not support double-precision floating point calculations

(dense) $n \times n$ matrix using LU-decomposition.

While we could do the benchmarks with the limited precision available, it is unlikely to tell us anything we do not already know (specifically that GPUs are fast at floating-point computations) and while we would be able to compare the results we would obtain against the different implementations, we would be unable to compare our results with the results of anybody else. Reason 3 is the most important though: it is simply a bad benchmark for what we want to test, since it does not do much to resemble the variety of work done in real life high performance computing.

The authors of [Fat09] did implement the linpack benchmark and was able to achieve almost 100% improvement (from about 40 Giga-flops to around 70) by moving all the computations from the CPU to the GPU and further improve this result to about 110 Giga-flops by using both the GPU and CPU in combination.

An alternative to this benchmarks are some benchmarks that are based on specific problems taken from program that are used in high performance computing. On such benchmark is the BioPerf benchmark, which contains problems and sample data-sets from computational biology such as sequence comparison, phylogenetic reconstruction[1] and protein structure prediction[BLLS06]. It currently contains 10 such problems[BLLS06]. We will not look at these problems any further, since it requires some understanding of computational biology, and that is not relevant for this project.

A similar example of a benchmark derived from the real world is the NAS benchmark, which is composed of problems taken from large-scale computational fluid dynamics[BBB$^+$91]. Unlike the problems in the BioPerf set, the problems in the NAS benchmark are surprisingly general[BBB$^+$91]:

1. Evaluate an integral based on pseudorandom trials.
2. A multigrid kernel, which tests communication between nodes.

[1] A technique used to compare genes of different animals to find the evolutionary development history between different animal species

3. A conjugate gradient to compute an eigenvalue for a sparse matrix.
4. A 3D partial differential solver using fast furrier transforms[1].
5. A kernel to sort large integers.
6. A system solver (LU) for a block triangular system[2].
7. A solution to compute independent systems of pentadiagonal scalar-equations[3].
8. A solution to compute independent systems of block tridiagonal equations[4].

These problems are specified in greater detail in [BBB$^+$91].

These benchmarks are somewhat more useful to us, although many of the problems are overlapping; this overlap is mostly a result of the indented implementation target of these benchmarks: large scale cluster computers. Since there is no long range communication on the GPU like one would find on a cluster grid, problems 2, 3, 7 and 8 are mainly meant to test how different requirements for long distance communication influence the system performance.

If we remove long distance communication from consideration (while still considering communication between the individual threads on the GPU), we can group the benchmarks together:

1. The integration based on pseudorandom values (this is based on problem 1).
2. A conjugate gradient to compute an eigenvalue for a matrix (this is based on problem 3).

[1] Fast furrier transforms is a computationally efficient method for converting a set of discrete input values to a corresponding frequency spectrum.

[2] A block triangular system is in this case a triangular matrix with sub-matrices as its elements.

[3] That is a diagonal matrix in which the diagonals above and below the diagonal may also have non-zero elements.

[4] A block tridiagonal matrix is a matrix in which the diagonal above and below the diagonal may also have non-zero elements; a block matrix is a matrix in which the elements are represented as sub-matrices (of equal size) rather than as scalar values.

3. The fast furrier transformation (this is based on problem 4).
4. A sorting kernel (this is based on problem 5).
5. A solver for linear-equations (this is based on problems 6, 7 and 8).

Finally there was a set of benchmarks in the paper on hiCUDA (see [HA09] and Section 8.3 on page 53), which the authors called “standard CUDA benchmarks”[HA09]. These are:

1. Matrix Multiplication.
2. Coulombic Potential[1].
3. Sum of Absolute Differences.
4. Two point Angular Correlation Function[2].
5. Rys Polynomial Equation Solver[3].

If we ignore the specifics of the equations use in item 2, 4 and 5, we have benchmarks for solving a simulation of interacting points, evaluating a function over an area and solving a polynomial equation.

[1] The Coulumb Potential is used to describe charges in water molecules, which effects how they interact with one another.

[2] This function is used to compute the probability of finding an astronomical body a specific (angular) distance from another astronomic body.

[3] Used to calculate 2-electron repulsion integrals between electrons in molecules.

Part II

Solution

How much easier it is to be critical than to be correct.

Benjamin Disraeli

10 Criteria for the Solution

As we mentioned in Chapter 7 on page 46, it is more important from a productivity point of view, to ensure that the solution we implement lowers the risk of having to debug the code as much as possible, rather than just speeding each debugging session up (such as by having it give out error messages that clearly identifies the place where there might be a problem).

A programmer who uses OpenCL today has to deal with a number of potentially problematic issues which have been removed for many many areas of software development, outside of high performance computing:

1. Pointers and pointer arithmetic.
2. Memory and memory handling.
3. Memory bandwidth and cache issues.
4. A limited type-system.

All of these issues except issue 3 comes as a result of the choice to use C as the language to write the kernels in. This was no doubt chosen for pragmatic reasons: it is at the same time a relatively low level language and a language with many programmers know, consistingly either the first or second programming language on the TIOBE index since 2001[TIO10].

The only language more popular than C, as of the 2010 TIOBE language index[TIO10], is Java. Unlike C Java does not allow the programmer to directly change pointers, which removes the complications given by issue 1, nor is the programmer expected to handle memory directly (Java has a garbage collector so the programmer does not have to manually free the memory he has allocated) which greatly helps with issue 2, since forgetting to deallocate memory is a common source of bugs.

Memory and pointer bugs are a class of bugs that can be very hard to track down, because they may not manifest themselves every time the code is run, making it much more difficult to figure out whether they are present or, if they have previously been found, have be solved.

The final issue (issue 4) is the type-system. C is able to catch some of the possible errors a computer programmer may make, such as assigning a float value when a pointer is expected, but C there are many things which the C type system cannot handle, such as polymorphism. This matters because a programmer may wish to avoid situations such as the addition of two floats which represents temperatures in different scales (such as Fahrenheit and Celsius) without first converting them to a common format, or to ensure that strings are properly escaped before they are saved in a database.

These issues would suggest that Java would be a better suggestion than C to write kernels in, but there is at least two big problems with that:

1. Many things that are often done in Java — such as creating objects or overriding functions in subclasses – are either very inefficient on the GPU or outright impossible (function pointers, which would be necessary to implement functions that can be overridden in subclasses, cannot be implemented on the GPU with current hardware). This is the fundamental problem with Aparapi (see Section 8.2 on page 50).

2. Java is designed to run on a variety of different machines, originally embedded consumer-electronic applications and later retarget towards applications for the internet[GJSB05, p. XXII], but those machines all run one or at most a few independent

threads, rather than the large number of SIMD threads that are the basis of the GPU architectures.

What we want to do then is to create a better way to interact with the GPU than OpenCL or the Java language but at the same time derive as much advantage as possible from the widespread use of Java.

10.1 Domain specific languages

Since development of entirely new languages is both a large undertaking and simultaneously faces large problems with mainstream adaption, we will instead create a simple embedded Domain Specific Language.

Traditional Domain specific languages like Logo[1] or SQL[2] are examples of DSL which aims to make it much easier to solve a specific set of related problems rather than making it somewhat easier to solve a large range of less related problems.

```
FD 75
RT 54
LT 21
BK 17
```

Listing 67.1: An example of a simple program in Logo[Ove].

Rather than write this as an entirely new language, we will embed this into an already existing language. This technique has become fairly popular in the Ruby world, and the web framework Ruby on Rails uses this technique in several different places: Listing 67.2 shows an example:

```
class CreatePosts < ActiveRecord::Migration
  def self.up
    create_table :posts do |t|
      t.string :name
```

[1] One implementation is available at http://www.fmslogo.org/index2.html.

[2] The language used in many databases management systems.

```
      t.string :title
      t.text :content
      t.timestamps
    end
  end

  def self.down
    drop_table :posts
  end
end
```

Listing 68.0: Setting up database table in Ruby[Gui11]

This leaves but one thing: the choice of language to embed the DSL in. The obvious choice would be Java, but while a lot of people know Java and can code in it, Java is rather strict in how much one can change the language which limits how expressive the DSL can be.

On the other hand Scala, a language which we will present in much greater detail in the next chapter, has a much more malable syntax. Indeed BASIC has been embedded in Scala as a DSL:

```
object Lunar extends Baysick {
  def main(args:Array[String]) = {
    10 PRINT "Welcome to Baysick Lunar Lander v0.9"
    20 LET ('dist := 100)
    30 LET ('v := 1)
    40 LET ('fuel := 1000)
    50 LET ('mass := 1000)

    60 PRINT "You are drifting towards the moon."
    70 PRINT "You must decide how much fuel to burn."
    80 PRINT "To accelerate enter a positive number"
    90 PRINT "To decelerate a negative"

    100 PRINT "Distance " % 'dist % "km, " % "Velocity
      ↪     " % 'v % "km/s, " % "Fuel " % 'fuel
    110 INPUT 'burn
    120 IF ABS('burn) <= 'fuel THEN 150
    130 PRINT "You don't have that much fuel"
    140 GOTO 100
    150 LET ('v := 'v + 'burn * 10 / ('fuel + 'mass))
    160 LET ('fuel := 'fuel - ABS('burn))
    170 LET ('dist := 'dist - 'v)
    180 IF 'dist > 0 THEN 100
    190 PRINT "You have hit the surface"
    200 IF 'v < 3 THEN 240
    210 PRINT "Hit surface too fast (" % 'v % ")km/s"
    220 PRINT "You Crashed!"
    230 GOTO 250
    240 PRINT "Well done"

    250 END

    RUN
  }
}
```

Listing 69.0: Lunar Lander written as an embedded DSL in Scala[Fog09]

While there are good reason to choose Scala, we will not hide that part of the choice is due to our personal fondness for the language but that fondness is based on its expressiveness and compatibility with the Java platform: both of which are important properties for a language to embed DSLs in.

If you have tears, prepare to shed them now.
William Shakespeare

11 Scala

Scala is a modern computer programming language, which was developed by Martin Odersky. The Scala compiler – called `scalac` – compiles Scala code to jvm byte code (there is also a version which compiles to .NET bytecode) which means that a Scala program has access to all the libraries written for the jvm.

Scala, like Java, is an object-oriented language but unlike Java it is not exclusively so; it also has support for functional programming and actors. Scala can removes the Java limitation that exceptions which a method declares it can throw must be captured and it does not require that a class must be named the same as the file it is in – nor that there must be no more than one public class in each file.

```
public class Hello{
  public static void main(String[] args){
    System.out.println("hello,␣world");
  }
}
```

Listing 71.1: Hello world in Java

The difference between this and the corresponding Scala code is not that big, but it does show two different things: arrays of things are specified in the same way any other collection of things is and variable names are specified before the type, not after.

```
object Hello{
  def main(args: Array[String]) {
    println("hello, world");
  }
}
```

Listing 72.0: Hello world in Scala

However this example can be improved by inheriting from the `Application` trait:

```
object Hello extends Application{
  println "hello, world"
}
```

Listing 72.1: An improved version of Hello world in Scala

The benefits of using Scala is more apparent if we choose a more complicated example, so here is one that would require a lot more code to do in Java:

```
abstract case class Operation
case class Plus(val lhs: Operation,
                val rhs: Operation)
    extends Operation
case class Value(val number: int) extends Operation

object Calc{
  val calculation = Plus(Value(5),
                         Plus(Value(6),Value(7)))
  def calculate(cal: operation) : int = cal match{
    case Plus(lhs, rhs) => calculate(lhs) +
       calculate(rhs)
    case Value(number) => number
  }
  def main(args: Array[String]){
    println("The value is: " +
            calculate(calculation).toString);
  }
}
```

Listing 72.2: A simple calculator

There are a number of things to note about that example: case classes are classes which have special functions that allow us to create them without having to write new, and to extract their contents without having to specifically assign them one at a time.

Variables are specified like functions, except they use the keyword val (Akin to the final keyword in Java, but used more often) or var (if the value of the variable must be allowed to change), but observe that we do not specify the type of the variable – Scala is able to correctly infer that the type should be `Operation`.

Finally Scala – like many functional languages such as Erlang, Haskell and OCaml – supports pattern-matching, which is what we use to do the calculations. A more Java like example would have moved the calculation of the value, such that each class was responsible for computing its own value – but this requires that we have the ability to add more methods to a class (which may not be the case if we got the class from somebody else, or it is part of a public API), and means that if wanted to add the ability to also print it as an arithmetic expression we would have to add a new method to all the classes.

Extending the Object Calc to print the calculation is rather simple:

```
  def toRPNString(cal: operation): String =
    cal match{
      case Plus(lhs, rhs) => toRPNString(lhs)+
                             toRPNString(rhs)+"+"
      case Value(number) => number.toString
  }
  def operationToString(cal: operation): String =
    cal match{
      case Plus(lhs, rhs) => operationToString(lhs) +
                             "+" +
                             operationToString(rhs)
      case \_ => toRPNString(cal)
    }
```

There are many, many more features of Scala but we do not have the space here to show them all. A tour of Scala can be found at `http://www.scala-lang.org/node/104`.

Now that we have presented the language, we will have a look at the implementation of our software.

We are happy... we look forward to its implementation.

Justice Sabharwal

12 Implementation

In this chapter we describe the implementation of our toolkit, which we have decided to call it the Light GPGPU Bridge Toolkit, henceforth referred to simply as the toolkit. Our aim is to develop a proof of concept for a toolkit which bridge the disparity that exists between how programmers program ordinary computers and how they have to program the GPU.

The toolkit does not aim to do all that is required to make the transition from CPU to GPU programming smooth, so we called it the light toolkit.

The toolkit share a great deal in common with the Accelerator, but it does have a number of differences:

1. Accelerator is written in C#, whereas the toolkit is implemented in Scala. While C# is more flexible than Java, in particular it does allow a limited form of operator overriding, has support for anonymous functions and does not require that the name of the class is the same as the name of the source file but it is not as flexible as Scala. More importantly though it is possible to write the majority of the program in Java and write only the part that needs to use the GPU in Scala.
2. We only write the kernel, whereas Accelerator completely encapsulate everything necessary to program the GPU.
3. We make no assumptions about security, all we check for is that we are not past the length of the first float buffer.

4. We only support float buffers.

The central class in the compiler is the *GPUKernel* class, which is the class from which we construct the instruction tree (the root of this tree is stored in the variable *instruction*) and from which we compile the users instructions down to an OpenCL kernel.

The possible instructions in the language are each represented as a case class with a simple hierarchy where each class inherit the abstract *GPUInstruction* class. Since a few of the instructions take a type argument, the case classes that represents types all inherit from the abstract class *ArgumentType*, which in turn inherit from the *GPUInstruction* class as well.

```
class GPUKernel(input: GPUFloatArray){
  var instruction:GPUInstruction = input
  def +(rhs:GPUFloatArray)= {
    instruction = AddFloatArray(instruction, rhs)
    this
  }
  def -(rhs:GPUFloatArray) = {
    instruction = SubtractFloatArray(instruction,rhs)
    this
  }
  def +(rhs:Float) = {
    instruction = AddScalar(instruction,
      ↪   ScalarConstant(rhs))
    this
  }
  def -(rhs:Float) = {
    instruction = SubtractScalar(instruction,
      ↪   ScalarConstant(rhs))
    this
  }
  def sin() = {
    instruction = SineOfFloatArray(instruction)
    this
  }
  def cos() = {
    instruction = CosineOfFloatArray(instruction)
    this
  }
  def tan() = {
    instruction = TangentOfFloatArray(instruction)
    this
  }
```

Listing 77.0: The code which builds the tree as the user calls the functions

As the user calls each function, the corresponding object is made the root of the instruction tree. This enable us to compile the code by traversing the tree recursively.

To construct a kernel, the user can write code like

```
object Lgbt extends Application{
  val gpu = new GPUKernel(GPUFloatArray(Array(0f,1f,2f)))
  val gpu2 = gpu.sin + 4 - 2 + GPUFloatArray(Array(3f,4f,5f))
  println(gpu2 compileToKernel)
}
```

Because all the methods that the user can call to build the kernel return *this*, it is possible to chain them to the degree it is desired.

Since the kernel has to be expressed as a C function, we have to find out which arguments it takes, find a unique name for them and write the type. We do this by traversing the tree of instructions in pre-order at each step building a list of the arguments needed for that subtree.*GPUFloatArray* (which is a wrapper around an ordinary Scala float Array) is special because it requires us the add two items to the list of kernel arguments.

```
  private def kernelArguments(instruction : GPUInstruction) : List[GPUInstruction] = {
    instruction match {
      case AddFloatArray(lhs, rhs) => kernelArguments(lhs) ++ kernelArguments(rhs)
      case SubtractFloatArray(lhs, rhs) => kernelArguments(lhs) ++ kernelArguments(rhs)
    case AddScalar(lhs, rhs) if rhs.isInstanceOf[ScalarVariable] => List(FloatType()) ++ kernelArguments(lhs)
    case AddScalar(lhs, rhs) => kernelArguments(lhs)
    case SubtractScalar(lhs, rhs) if rhs.isInstanceOf[ScalarVariable] => List(FloatType()) ++ kernelArguments(lhs)
    case SubtractScalar(lhs, rhs) => kernelArguments(lhs)
    case SineOfFloatArray(lhs) => kernelArguments(lhs)
    case CosineOfFloatArray(lhs) => kernelArguments(lhs)
    case TangentOfFloatArray(lhs) => kernelArguments(lhs)
    case a:GPUFloatArray => List(a)
  }
}
```

Once we have a way to find the arguments we have to add to the header, we can write it:

```
private def compileHeader() : String = {
  var str = new StringBuilder
  var first = true
  str append "__kernel void kernel("
  for(val k <- kernelArguments(instruction)){
    if(first)
      first = false
    else
      str.append(", ")
    k match{
      case a:ArgumentType => str.append("__global ").append(a toType).append(" ").append(VariableName.getNextVariableName)
      case GPUFloatArray(_, name) => str.append("__global int ").append(name).append("Count, ").append("__global float* ").append(name)
    }
  }
```

```
    str.append(", __global ").append("float* ").append("out")
    str.append(")")
    str toString
  }
```

Listing 79.0: Header Compilation

Now that we know how to write the header, we can take a look at the body:

```
  private def compileBody() : String = {
    var str = new StringBuilder
    str append "{\n\tint id = get_global_id(0);\n"
    str.append("\tif(id < ").append(kernelArguments(instruction).first.asInstanceOf[GPUFloatArray].name).append("Count){\n")
    str append "\t\tout =" +compileInstructions(instruction) + ";\n"
    str append "\t}\n"
    str append "}"
    str toString
  }
```

Listing 79.1: The template for the body of the kernel

The most interesting thing here is that since a kernel can be though of as a loop where each iteration is executed in its own thread (with no guarantee as to the order), we can insert what would otherwise have been put into a loop directly into the kernel.

Finally we can write the code needed to compile the individual instructions. This is again done in preorder.

```
  private def compileInstructions(instruction : GPUInstruction) : String = {
    instruction match {
      case AddFloatArray(lhs, rhs) => compileInstructions(lhs) + "+" + compileInstructions(rhs)
      case SubtractFloatArray(lhs, rhs) => compileInstructions(lhs) + "-" +
          compileInstructions(rhs)
      case AddScalar(lhs,rhs) => compileInstructions(lhs)+ "+" + rhs.value
      case SubtractScalar(lhs, rhs) => compileInstructions(lhs) + "−"+ rhs.value
      case SineOfFloatArray(lhs) => "sin(" + compileInstructions(lhs) + ")"
      case CosineOfFloatArray(lhs) => "cos(" + compileInstructions(lhs) + ")"
      case TangentOfFloatArray(lhs) => "tan(" + compileInstructions(lhs) + ")"
      case GPUFloatArray(arr, name) => name+"[id]"
    }
  }
```

Listing 80.0: Instruction Compilation

All that is left to do is to put code to compile the header and the body of the function together:

```
  def compileToKernel() : String = {
    var str = new StringBuilder
    str append compileHeader()
    str append compileBody()
    str.toString
  }
}
```

The rest of the code can be perused on the accompanying cd.

Part III

Reflection

If you can look into the seeds of time, And say which grain will grow, and which will not, Speak.
Macbeth, Act I, scene iii

13 Future work

There are many directions from which one can extend the work we have presented in this report. The most interesting and useful thing would be to improve and extend the toolkit as well as the techniques for benchmarking and the different ways to measure programmer productivity.

There are a few specific directions in which the toolkit could be improved:

1. The toolkit should be improved such that it is better able to warn the user about possible errors.
2. The toolkit should be extended to include summing and multiplication of entire arrays as well as allow the user to use the full range of functions available in OpenCL. Additional improvements could include adding more advanced and specialized functions which are not available in OpenCL such as image recognition and statistical functions.
3. The toolkit is currently implemented in Scala which requires the programmer to be familiar with this language. To solve this problem a complete compiler could be made which allowed the user to implement the kernels in an extension to, or a subset of, Java.
4. The toolkit could be improved such that it can scan already existing programs and replace parts of them with GPU accelerated code where such changes could result in faster computation.

"what 's done is done".

Macbeth (Act III, Scene II)

14 Conclusion

The original goal of this report was to find out if it is possible to develop something which would allow programmers to develop for the GPU at a higher level of abstraction than the C-like language which is used in OpenCL and if so how to benchmark it against other solutions to the same problem.

We looked into programmer productivity and found that while there are no productivity studies with a large number of subjects available we were able to extract some knowledge about what we should do to minimize the time that the developer needs to spend on debugging specifically it is more important to lower the number of times a developer has to go back to debugging before the program is useful than it is to make each of these debug sessions shorter.

After this we looked at a number of benchmarks for high performance computing and found that they mostly used a common set of operations which included matrix multiplication, solving linear equations and sorting. We likewise found a few measurements of programmer productivity, however these were of no use.

As a way to improve programmer productivity, we researched a number of paradigms that are available as alternatives to CUDA and OpenCL. While these paradigms have their positive and negative sides we chose OpenCL as our target for making a toolkit because OpenCL is not tied to down to one hardware platform.

Using this research we implemented a toolkit which allowed users to write the OpenCL kernel at a higher abstraction that what they would otherwise have been able to. In the future works section we documented a number of ways in which the toolkit could be improved.

Bibliography

[amda] *Introduction to opencl programming*, http://developer.amd.com/zones/OpenCLZone/courses/Documents/Introduction_to_OpenCL_Programming%20Training_Guide%20(201005).pdf.

[AMDb] AMD, *A brief history of general purpose (gpgpu) computing*, http://www.amd.com/us/products/technologies/stream-technology/opencl/Pages/gpgpu-history.aspx.

[Amd10] Amd, *Aparapi readme*, http://developer.amd.com/zones/java/assets/README.html, 12 2010.

[Ano] Anonymous, *Amazon ec2 gpu computing*, http://groups.google.com/group/pyrit/browse_thread/thread/6fb00f6c41e6ee0c?pli=1.

[BBB$^+$91] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga, *The Nas Parallel Benchmarks*, International Journal of High Performance Computing Applications **5** (1991), no. 3, 63–73.

[BLLS06] David A. Bader, Yue Li, Tao Li, and Vipin Sachdeva, *Bioperf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications*, Tech. report, Georgia Institute of Technology, 2006, http://hdl.handle.net/1853/14386.

[Coo10] Andrew Cooke, *A practical introduction to opencl*, http://www.acooke.org/cute/APractical0.html, 12 2010.

[CR10] Srika Chowdary Ravela, *Comparison of shared memory based parallel programming models*, Master's thesis, Blekinge Tekniska Högskola, 2010.

[DG04] Jeffrey Dean and Sanjay Ghemawat, *Mapreduce: Simplified data processing on large clusters*, Tech. report, Google, Inc, 2004.

[DOLG$^+$] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timonthy J. Purcell, *A survey of general-purpose computation on graphics hardware*.

[Fat09] Massimiliano Fatica, *Accelerating linpack with cuda on heterogenous clusters*, Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (New York, NY, USA), GPGPU-2, ACM, 2009, pp. 46–51.

[Fog09] Fogus, *Baysick: A scala dsl implementing basic*, `http://blog.fogus.me/2009/03/26/baysick-a-scala-dsl-implementing-basic/`, 03 2009.

[FRGMS07] Andrew Funk, John R. Gilbert, David Mizell, and Viral Shah, *Modeling programmer workflows with timed markov models*, 2007.

[GDB08] Vincent Garcia, Eric Debreuve, and Michel Barlaud, *Fast k nearest neighbor search using gpu*, Computer Vision and Pattern Recognition Workshop **0** (2008), 1–6.

[GGKK03] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, *Introduction to parallel computing*, ch. 6, Pearson, Addison-Wesley, 2003.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The java language specification, third edition*, Addison Wesley, 2005.

[GRV10] Osvaldo Gervasi, Diego Russo, and Flavio Vella, *The AES Implantation Based on OpenCL for Multi/many Core Architecture*, Proceedings of the 2010 International Conference on Computational Science and Its Applications (Washington, DC, USA), ICCSA '10, IEEE Computer Society, 2010, pp. 129–134.

[Gui11] Rails Guides, *Ruby on rails guides: Getting started with rails*, `http://guides.rubyonrails.org/getting_started.html`, Jan 2011.

[HA09] Tianyi David Han and Tarek S. Abdelrahman, *hicuda: a high-level directive-based language for gpu programming*, Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (New York, NY, USA), GPGPU-2, ACM, 2009, pp. 52–61.

[had] *Apache hadoop project*, `http://hadoop.apache.org/`.

[HB07] S. Harding and W. Banzhaf, *Fast Genetic Programming and Artificial Developmental Systems on GPUs*, High Performance Computing Systems and Applications, 2007. HPCS 2007. 21st International Symposium on, May 2007, pp. 2 –2.

[HCC$^+$10] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin, *MapCG: writing parallel program portable between CPU and GPU*, Proceedings of the 19th international conference on Parallel architectures and compilation techniques (New York, NY, USA), PACT '10, ACM, 2010, pp. 217–226.

[HFL$^+$08] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang, *Mars: a mapreduce framework on graphics processors*, Proceedings of the 17th international conference on Parallel architectures and compilation techniques (New York, NY, USA), PACT '08, ACM, 2008, pp. 260–269.

[HR75] Frederick Hayes-Roth, *Review of "adaptation in natural and artificial systems by john h. holland", the u. of michigan press, 1975*, SIGART Bull. (1975), 15–15.

[Int] Intel, *Xeon processors*, `http://www.intel.com/support/processors/xeon/sb/CS-020863.htm`.

[JA] Alvin J. Alexander, *How to determine your software application size using function point analysis*, On web, acccessed 7 December 2010.

[Jon94] Capers Jones, *Function points*, Computer **27** (1994), 66–67.

[Kan10] David Kanter, *Amd's cayman gpu architecture*, http://realworldtech.com/page.cfm?ArticleID=RWT121410213827&p=1, 12 2010.

[Kem93] Chris F. Kemerer, *Reliability of function points measurement: a field experiment*, Commun. ACM **36** (1993), 85–97.

[LCWM08] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng, *Merge: a programming model for heterogeneous multi-core systems*, Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (New York, NY, USA), ASPLOS XIII, ACM, 2008, pp. 287–296.

[LLW04] Youquan Liu, Xuehui Liu, and Enhua Wu, *Real-Time 3D Fluid Simulation on GPU with Complex Obstacles*, Proceedings of the 12th Pacific Conference on Computer Graphics and Applications (PG'04), University of Macau, Macao, China, 2004.

[LLWJ09] Shenshen Liang, Ying Liu, Cheng Wang, and Liheng Jian, *A cuda-based parallel implementation of k-nearest neighbor algorithm*, Cyber-Enabled Distributed Computing and Knowledge Discovery, 2009. CyberC '09. International Conference on, 2009, pp. 291 –296.

[Mel] Melonakos, *GPU computing with MATHLAB*, http://blog.accelereyes.com/blog/2008/12/30/opencl/.

[MHr] Jesper Mosegaard, Peder Herborg, and Thomas Sangild Sørensen, *A GPU Accelerated Spring Mass System for Surgical Simulation*.

[Nvia] Nvidia, *Nvidia GeForce 8800 Architecture Technical Brief*.

[Nvib] ______, *Tesla computing solutions: Nvidia tesla c870*, http://www.nvidia.co.uk/page/tesla_gpu_processor.html.

[Nvi08] ______, *Nvidia Tesla C870 GPU Computing Processor Board*, http://www.nvidia.com/docs/IO/43395/C870-BoardSpec_BD-03399-001_v04.pdf, 2008.

[Nvi10a] ______, *Nvidia cuda c programming guide (version 3.2)*, http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf, 2010.

[NVI10b] NVIDIA, *NVIDIA opencl best practices guide v1.0*, http://www.nvidia.com/object/cuda_opencl_new.html, 12 2010.

[NVI10c] ______, *OpenCL programming guide for the cuda architecture v3.2*, http://www.nvidia.com/object/cuda_opencl_new.html, 12 2010.

[ope] *The opencl specification*, http://www.khronos.org/registry/cl/.

[Ove] Stack Overflow, *How do i move the turtle in logo*, http://stackoverflow.com/questions/1003841/how-do-i-move-the-turtle-in-logo.

[Pha05] Matt Pharr (ed.), *GPU Gems2: Programming techniques for high-performance graphics and general-purpose computation*, ch. 43-48, Addison-Wesley, 2005.

[pro10] Pyrit project, *Pyrit project*, http://code.google.com/p/pyrit/, 12 2010.

[QMN09] Deyuan Qiu, Stefan May, and Andreas Nüchter, *Gpu-accelerated nearest neighbor search for 3d registration*, Computer Vision Systems (Mario Fritz, Bernt Schiele, and Justus Piater, eds.), Lecture Notes in Computer Science, vol. 5815, Springer Berlin / Heidelberg, 2009, pp. 194–203.

[Res] Microsoft Research, *An introduction to microsoft accelerator v2*.

[Rot] Thomas Roth, *Cracking Passwords In The Cloud: Amazon's New EC2 GPU Instances*, http://stacksmashing.net/2010/11/15/cracking-in-the-cloud-amazons-new-ec2-gpu-instances/.

[RRB$^+$08] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W.

Hwu, *Optimization principles and application performance evaluation of a multithreaded gpu using cuda*, Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (New York, NY, USA), PPoPP '08, ACM, 2008, pp. 73–82.

[RRP$^+$07] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis, *Evaluating MapReduce for multi-core and multiprocessor systems*, In HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture, IEEE Computer Society, 2007, pp. 13–24.

[Sch95] Bruce Scheier, *Applied crypography, second edition*, John Wiley and Sons, 1995.

[SFEV$^+$09] Mark S. Friedrichs, Peter Eastman, Vishal Vaidyanathan, Mark Houston, Scott Legrand, Adam L. Beberg, Daniel L Ensign, Christopher M. Bruns, and Vijay S. Pande, *Accelerating molecular dynamic simulation on graphics processing units*, Wiley Periodicals, Wiley, 2009.

[Stra] StreamComputing.eu, *Difference between cuda and opencl 2010*, `http://www.streamcomputing.eu/blog/2010-04-22/difference-between-cuda-and-opencl`.

[Strb] ———, *nvidia's cuda vs opencl marketing*, `http://www.streamcomputing.eu/blog/2010-02-08/nvidia-cuda-and-opencl-marketing`.

[TIO10] TIOBE, *Tiobe index december 2010*, `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, December 2010.

[TPO06] David Tarditi, Sidd Puri, and Jose Oglesby, *Accelerator: using data parallelism to program gpus for general-purpose uses*, SIGOPS Oper. Syst. Rev. **40** (2006), 325–335.

[tux] Permission to use and/or modify this image is granted provided you acknowledge me lewing@isc.tamu.edu and The GIMP if someone asks.

[Wika] Wikipedia, *Block cipher modes of operation - electronic codebook*, `http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation#Electronic_codebook_.28ECB.29`.

[Wikb] ______, *GPGPU*, `http://en.wikipedia.org/wiki/GPGPU`.

[Wikc] ______, *Graphics processing unit*, `http://en.wikipedia.org/wiki/Graphics_processing_unit`.

[WW09] Man Wong and Tien Wong, *Implementation of Parallel Genetic Algorithms on Graphics Processing Units*, Intelligent and Evolutionary Systems (Mitsuo Gen, David Green, Osamu Katai, Bob McKay, Akira Namatame, Ruhul Sarker, and Byoung-Tak Zhang, eds.), Studies in Computational Intelligence, vol. 187, Springer Berlin / Heidelberg, 2009, 10.1007/978-3-540-95978-6_14, pp. 197–216.

[yah] *Yahoo hadoop tutorial*, `http://developer.yahoo.com/hadoop/tutorial/module3.html#running`.