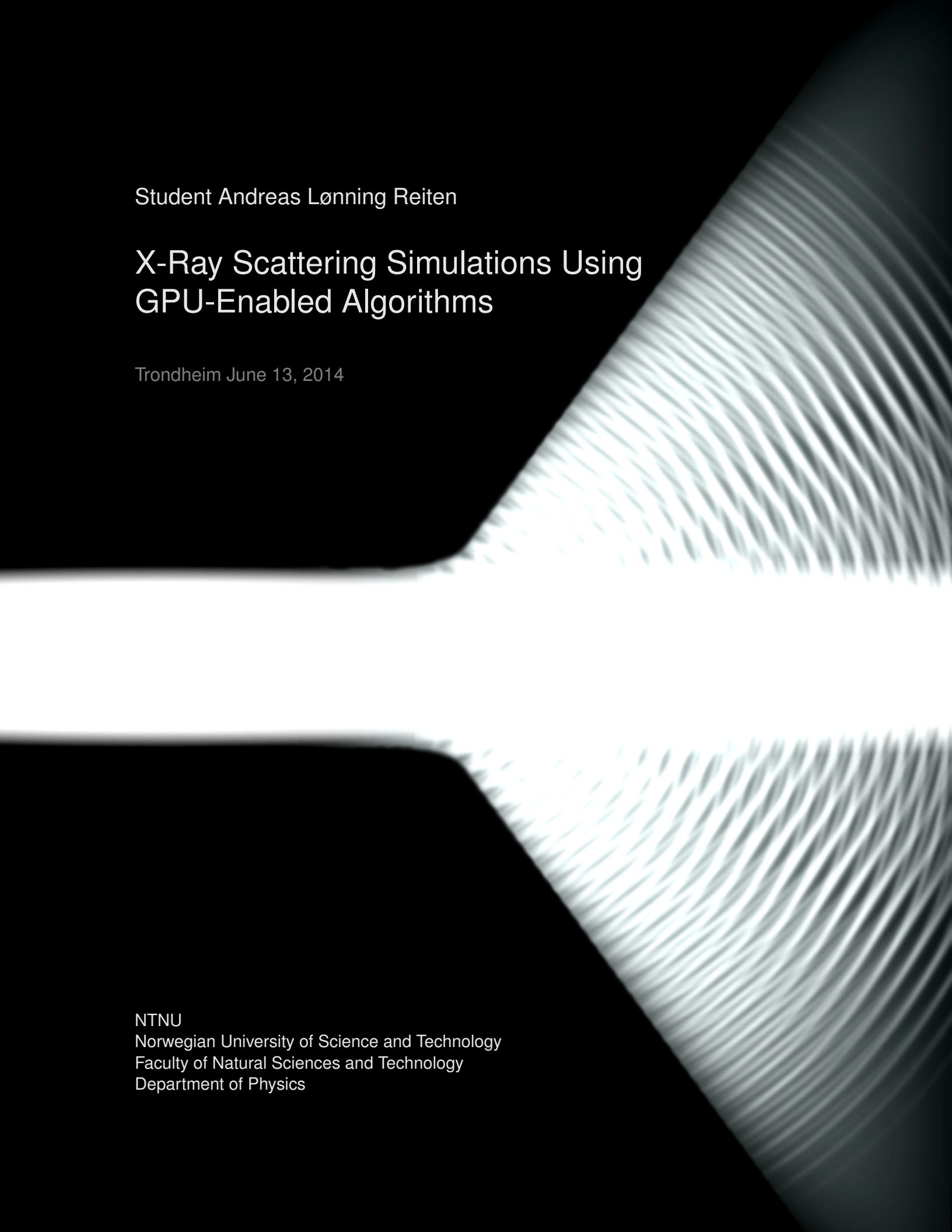
Student Andreas Lønning Reiten

X-Ray Scattering Simulations Using GPU-Enabled Algorithms

Trondheim June 13, 2014



NTNU
Norwegian University of Science and Technology
Faculty of Natural Sciences and Technology
Department of Physics

Preface

This report is the result of a specialization project in nanotechnology carried out at the Division of Condensed Matter Physics at the Norwegian University of Science and Technology (NTNU), under supervision of associate professor Dag W. Breiby and PhD candidate Jostein Bø Fløystad. The text targets readers with some experience from computational physics who wish to learn more about General Purpose computing on Graphics Processing Units (GPGPU) applied in a physics context.

I would like to thank Ingve Simonsen at NTNU for giving feedback and comments on the GPU algorithm from a computational perspective, and Kristin Høydalsvik at NTNU for providing X-ray diffraction data to compare with simulated results. I would also like to thank Kim A. G. Grimsrud for her highly relevant report on computer simulated X-ray powder diffraction [1], which served as a starting point for this project.

Finally, I would like to express special thanks to my supervisors, Dag W. Breiby and Jostein Bø Fløystad, for their support and guidance throughout the project. I would also like to tank them for the large amount of time they have put into reading multiple iterations of this report and for the feedback they have given.

Andreas Lønning Reiten
Trondheim, June 13, 2014

Abstract

A new program for calculating the powder diffraction pattern from nano-sized particles has been written to run on a graphics processing unit (GPU). The performance and precision of this software has been compared to a program running the same calculations on a central processing unit (CPU). The text also describes optimization techniques that were employed to better utilize the hardware resources present on the graphics card.

The GPU software showed a relatively large performance gain over the CPU algorithm, and there was also further improvement after optimization of the GPU code. The GPU algorithms exhibit precision losses compared to the CPU code, but the relative errors remain within acceptable bounds ($<< 1\%$).

Gelisio et al. [2] reported in 2009 that a similar program running on a GPU was limited in performance partly due to insufficient capacity of a specific type of fast memory cache. This report investigates, with the next generation of GPUs, whether performance is still limited by this factor.

The GPU accelerated program has been written using OpenCL [3], a portable open standard framework for writing programs that execute across platforms consisting of CPUs, GPUs, and other processors.

Contents

Chapter 1

Introduction

Driven by the computer gaming industry's hunger for more vivid and detailed graphics, General Processing Units (GPUs) are becoming increasingly powerful. Reaching out to the parallel computing market, GPU manufacturers are now producing more and more programmable devices. Currently, a decent desktop computer with a high end graphics card gives numerical performance in the tera FLOPS range, and with their relatively modest energy consumption GPUs are now building blocks in three of the world's top five supercomputers.

The Debye equation [4] gives the diffraction pattern of any powder consisting of equal and randomly oriented particles without assumptions of crystalline lattice and long range order. The number of arithmetic operations in the equation scales as N^2, where N is the number of atoms in one of the particles that constitute the powder. Even for nano-sized particles, there can be hundreds of thousands of atoms, making the simulation of the diffraction pattern a very time consuming process. In recent years, however, the equation has been increasingly used [5, 6, 7, 8, 2] not only due to the availability of increasingly powerful hardware, but also because small angle scattering from nano-sized particles cannot be modeled using regular bulk approximations. In addition, such particles can exhibit a variety of defects and variations from the ideal structure, and the Debye equation takes all of this into consideration. Furthermore, diffraction data from X-ray measurements can be matched against computer simulated results for characterization. This makes the Debye equation highly relevant for material studies in nanotechnology.

GPUs are parallel computing devices that boast impressive theoretical performance. This report investigates the performance and precision of this kind of hardware when used in single precision (SP) to calculate the Debye equation, and examines whether performance limiting factors discussed by Gelisio et al. [2] in 2009 are still predominant, now with the next generation of GPUs. There is also a comparison to a CPU based algorithm in double precision (DP), both in terms of performance and precision. The text also gives a description of the most common optimization techniques that should be applied to approach peak performance on GPUs.

All figures in this text are property of the author.

Chapter 2

GPU Theory and Programming

The CPU is perhaps what people primarily associate with computation hardware. The term CPU was introduced in the early 1960s by the industry, and the computing capabilities of this device have increased exponentially since then [9]. However, it should be noted that recent works forecast a decline of this trend due to technological barriers [10, 11].

The CPU carries out each instruction of a program sequentially on a single computing core performing arithmetical, logical, and input/output operations. A GPU, in contrast, executes each instruction of a program sequentially, but for many data elements at the same time, cf. fig. 2.1. To clarify, a CPU is a Single Instruction Single Data (SISD) computing architecture[1], and a GPU is a Single Instruction Multiple Data (SIMD) architecture, meaning that the GPU has several computing cores working on the same instruction but with different data. As this section will show, such parallelism can sometimes greatly improve the performance of a calculation. Later in this report, we show how a GPU optimized X-ray scattering simulation achieves a substantial speedup compared to its CPU equivalent. Note that GPUs rely on CPUs for instructions and thus cannot work by themselves.

As portrayed in fig. 2.1 there is a fundamental difference in how CPUs and GPUs process their workload. This has its explanation in the fundamental difference between the SISD and SIMD architectures which is also apparent in the number of cores these devices possess[2]; as of 2010 a typical consumer grade CPU has up to six cores while a GPU has several hundreds[3]. Furthermore, this translates to a significant difference in theoretical performance in terms of floating point operations per second (FLOPS), which is proportional to the number of cores and the core operating frequency. However, it should be noted that this

[1] Multi-core CPUs and CPU based clusters are in fact Multiple Instruction Multiple Data (MIMD) computing systems. However, this report views CPUs as single core devices for instructional purposes.

[2] The cores are all crammed onto a single die/chip. In the case of the CPU, this die is connected directly to the host (motherboard). For a GPU the die is connected to a graphics card, which in turn is connected to the host.

[3] The NVIDIA GeForce GTX 480 has 480 cores running at 1401 MHz, and the ATI Radeon HD 5870 has 1600 cores at 850MHz. The Intel Core i7-970 CPU has six cores at 3200MHz.

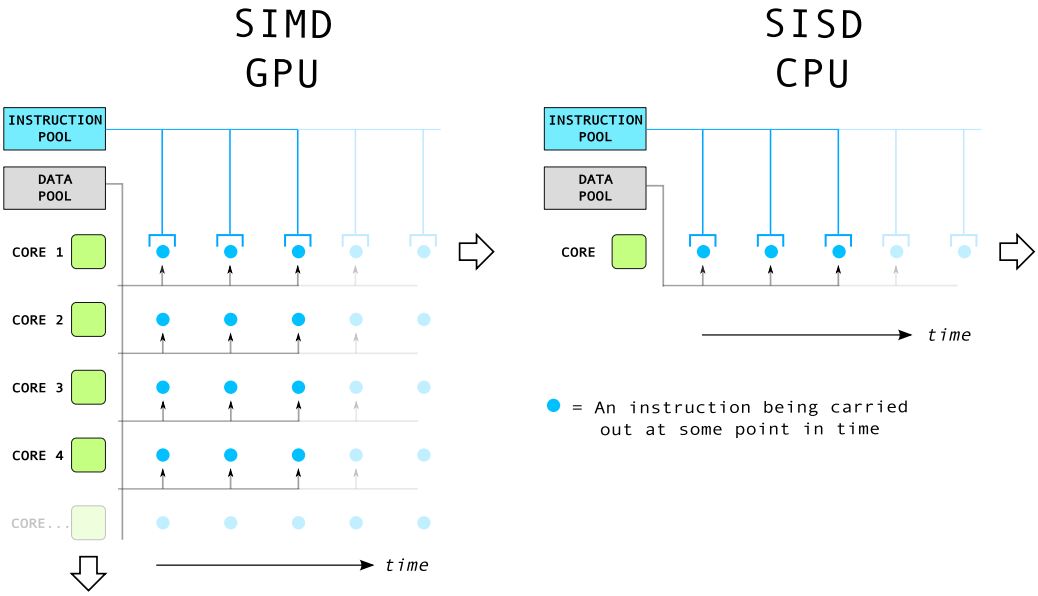


Figure 2.1: CPUs and GPUs differ greatly from one-another. A GPU is a SIMD computing architecture (left), and a CPU is a SISD architecture (right). The CPU has one computing core carrying out the instructions of a program sequentially. A GPU, in contrast, has several computing cores working on the same instruction but with different data for each core.

gage does not give a good measure of real-world performance, but it does give a hint. Refer to appendix A.1 for a more in-depth explanation of FLOPS.

FLOPS can be a performance limiter for computationally intensive algorithms, but in many cases memory bandwidth (MBW) is the bottleneck [12]. MBW is analogous to the speed at which data is transferred back and forth between the computing cores and device memory. An important part of GPU programming therefore revolves around constructing algorithms that minimize memory transfers and redundant use of stored data. Appendix A.2 gives a more thorough explanation of memory bandwidth. Table 2.1 shows some key attributes for a few mainstream graphics cards and specialized hardware, including theoretical FLOPS and MBW for single and double precision operations.

2.1 Architecture

In order to understand the ins and outs of GPU computing it is necessary first to have a basic understanding of the underlying hardware architecture. The architecture depends on manufacturer and model, but from a programming point of view it is the quantitative rather than the qualitative attributes that differ. Most newer graphics cards from both NVIDIA and ATI can run almost the exact same code due to qualitatively similar architectures. However, a code that is optimized for one architecture might not be optimal for a different one. Sections 2.1.1 and 2.1.2 go through some crucial concepts that follow as a consequence of hard-

Table 2.1: Comparison of computational hardware. The interpretation of FLOPS and MBW is discussed in appendices A.1 and A.2 respectively. Double precision (DP) throughput is considerably lower than single precision (SP) throughput for these devices. Note that FLOPS are generally not regarded as an accurate measure of actual performance.

Model	**Year**	**Release price** [USD]	**Cores**	**Frequency** [MHz]	**FLOPS (SP)** [TFLOPS]	**FLOPS (DP)** [TFLOPS]	**Bandwidth** [GB/s]
NVIDIA GTX 480	March 26, 2010	$499	480	1401	1.345	0.168	177.4
NVIDIA GTX 470	March 26, 2010	$349	448	1215	1.089	0.136	133.9
ATI/AMD Radeon HD 5870	September 23, 2009	>$300	1600	850	2.720	n/a	153.6
ATI/AMD Radeon HD 5850	September 30, 2009	>$300	1440	725	2.088	n/a	128
NVIDIA M2070	Q2, 2010	>$4000	448	1150	1.288	1.030	150.3

ware architecture.

2.1.1 GPU layout

The GPU die is divided in several small subsections as shown in fig. 2.2. Each of these regions constitute a *compute unit*. Each compute unit consists of many[4] smaller cores, or *processing elements*, that carry out most of the arithmetic operations. The processing elements in a compute unit can talk to each other and share data using a small cache[5] of *local memory*, which is very fast. The compute units can not communicate with each other.

A number of caches and control units such as registers, memory controllers, special function units, load/store units, and instruction schedulers occupy the rest of the die space. However, their functions generally fall outside the scope of this discussion. Moving off the chip there is a huge memory reservoir called *global memory*. Global memory is both much larger and much slower than local memory. It is used mainly to store input data before a program is invoked and to store results before they are read by the host.

[4]The NVIDIA GTX 480 and 470 have 32 single precision processing elements per compute unit. The ATI HD 5870 has 80. There are in addition special processing elements for some mathematical functions.

[5]A memory reservoir used to temporarily store data that will be used a lot is often referred to as a *cache*. Caches traditionally trade size for speed.

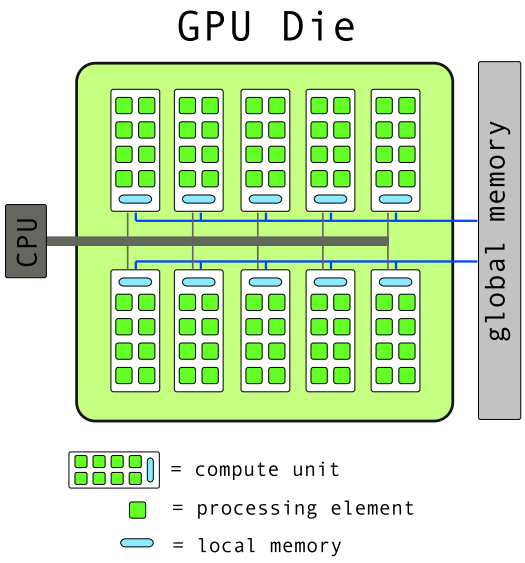


Figure 2.2: A boiled-down representation of a GPU die showing some of the more essential parts. The CPU sends instructions to the compute units consisting of i.a. processing elements and local memory. Global memory is accessible both by the host and the compute units. A CPU sends instructions to the compute units.

2.1.2 The work group hierarchy

The code that is sent to a GPU is called a *kernel* and consists of a sequence of instructions that is forwarded in segments to the compute units. The compute unit then dispatches the instructions to the processing elements.

The kernel running on a processing element is called a *work item*, and the programmer can make a group of them map to the same compute unit. These work items form a *work group* and can share data through the compute unit's local memory cache as illustrated in fig. 2.3.

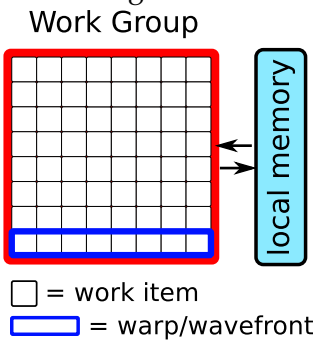


Figure 2.3: The work group consists of multiple work items assigned to one compute unit that can share data through local memory. Work items are serviced by the processing elements warp by warp.

Work items within a work group are processed in clusters corresponding to the number of processing elements on the compute unit. NVIDIA has named these clusters *warps*, while ATI calls them *wavefronts*. In the remainder of this

text these clusters will be referred to as warps. Work items are always serviced together in batches like this. Furthermore, warps can be left to idle while waiting for memory transfers to complete and thus allowing for other warps to be serviced in the meantime. The size of a work group should be a multiple of the size of a warp to keep all processing elements active[6].

Each work item can be distinguished by unique *indices* both within the work group and the grid. For example, the index can be used to access specified addresses in local or global memory.

The programmer decides how much work is done by each work item when writing the kernel, and naturally the work should be distributed so that the number of work groups is greater than the number of compute units to prevent any of them from idling. It is also possible for a compute unit to work on several work groups at once to hide memory latencies. This will be examined in section 2.3.5.

Together the work groups make up a grid as shown in fig. 2.4. The dimensions of the work groups and grid have no real significance other than being a convenience when working with multidimensional problems. Still, the total number of work items in a work group, and of work groups in a grid, has important consequences as described in section 2.3.5.

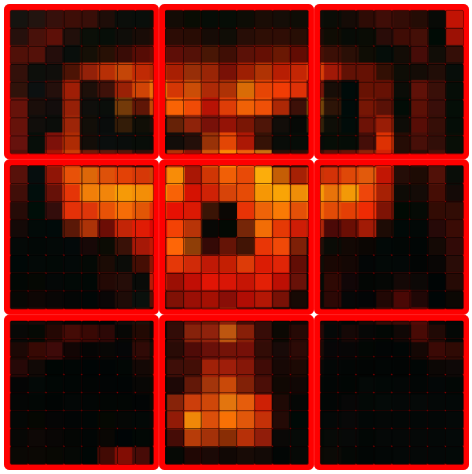


Figure 2.4: The work groups that are needed to solve a problem form a grid. The dimensions of the grid can easily be mapped to a multidimensional problem for the sake of convenience. One example is to let the grid span over an image and have each work item represent a pixel as shown here.

2.2 OpenCL

OpenCL[7] is a framework for writing programs that execute across platforms consisting of CPUs, GPUs, and other processors. OpenCL includes a language for

[6] As an example, the NVIDIA GTX 480 has a warp size of 32. The work groups should therefore be of size $32n$ where $n = 1, 2, 3..., 48$. The upper limit of $n = 48$ arises due to a hardware limit of 1536 resident work items on a compute unit.

[7] Abbreviation for Open Computing Language.

writing kernels and application programming interfaces[8] (APIs) that are used to define and then control the platforms [3].

OpenCL is developed and maintained by the member-funded Khronos Group that works to create open standards for graphics, media and parallel computation. Many well known hardware manufacturers are member of the group, and NVIDIA, AMD/ATI, and Intel are among them. The main argument for using OpenCL instead of other APIs is the open standard that makes it much more portable than for example the CUDA API from NVIDIA or the Stream API from AMD/ATI, which are limited t.

2.2.1 PyOpenCL

PyOpenCL [13] accesses the OpenCL parallel computation API from Python [14], a high level open source programming language. Python is especially convenient as an upper level code layer due to its clean syntax and broad functionality.

2.3 Performance Optimization

This section highlights why optimization is essential to achieve appreciable performance in general purpose GPU computing and gives a few examples of how it can be realized. The following paragraphs first describes high priority optimization factors and then proceeds to briefly discuss some that are normally less important. The content is largely based on guides from NVIDIA [15, 16] and ATI [17].

2.3.1 Structuring an Algorithm

Evidently, the algorithm has to be written for parallel execution: There are several processing elements working at the same time on a work group, and the kernel should be written in such a way that the work is equally distributed among the work items. For example, if a work group needs to load some data and has 256 work items, then all those should fetch $1/256$ of that data each, and, depending on the problem, either do the necessary operations on that data or store it in local memory to be shared with the other work items. This is an example of parallelization. Figure 2.5 shows how a simple code changes from a sequential to a parallel implementation.

Generally, during the optimization process it is important to be aware of significant performance limiters. For example, optimizing instruction usage of a kernel that is mostly limited by memory accesses will not result in any significant performance gain. NVIDIA and ATI offer free proprietary software that can be used to monitor these performance limiters. Also, for larger kernels, it is recommended to optimize heavier parts of the code first, since this is where the largest gain in performance can be obtained.

[8] An application programming interface is an interface implemented by a software program that enables it to interact with other software.

```
for(int i = 0; i < N; i++){
     out_data[i] = in_data[i];
}
```

```
int i = get_global_id(1); //get unique work item index
out_data[i] = in_data[i];
```

Figure 2.5: Top: A sequential implementation of an algorithm that copies an array of length N. Bottom: The corresponding parallel implementation. Each of the N work items copies one element. Note that a for-loop is unnecessary since the programmer has already specified the number of work items that should carry out the task.

Memory transfer involving global memory is a costly operation in terms of clock cycles compared to transfers within local memory and most arithmetic operations. The algorithm should therefore be constructed in a way that minimizes redundant memory access, or even better, reduces memory access altogether. Global memory latency can be hidden if sufficiently many arithmetic instructions can be issued while waiting for the transfer to complete. However, this is only possible for certain types of problems. The ratio of clock cycles spent for the arithmetic operations to the number of clock cycles spent for load/store operations will be referred to as arithmetic density in this text.

2.3.2 Global Memory

Input and output data has to be stored in global memory before and after a kernel is invoked, respectively. Assuming the algorithm has already been designed to minimize global memory transfer, substantial performance losses can still occur if the data is not accessed in a way that agrees on a hardware level.

Global memory is divided in several *memory banks* that are accessed from each compute unit, and if two memory access requests go to the same memory bank, hardware serializes the access. This is commonly referred to as a *bank conflict*. When data is written to global memory, consecutive elements are stored in consecutive memory banks. It is therefore preferred to access this data in the same order it was stored to minimize serialization. This is called *coalesced access*. As an example, if only every second element in an array of data was read, only half of the memory banks would contribute to transfer the data that was actually requested. A common term for this event is *strided access*, and in the former example the stride was two. Figure 2.6 shows a kernel with strided access, and fig. 2.7 displays the resulting effect on global memory bandwidth.

```
__kernel void strided_access(
__global const float* in_data,
__global float* out_data,
int stride)
{
     int i = get_global_id(1);
     out_data[i] = in_data[i*stride];
}
```

Figure 2.6: A kernel that copies elements stored in global memory. Strided access can be achieved by changing *stride* to a desired value as shown in fig. 2.7.

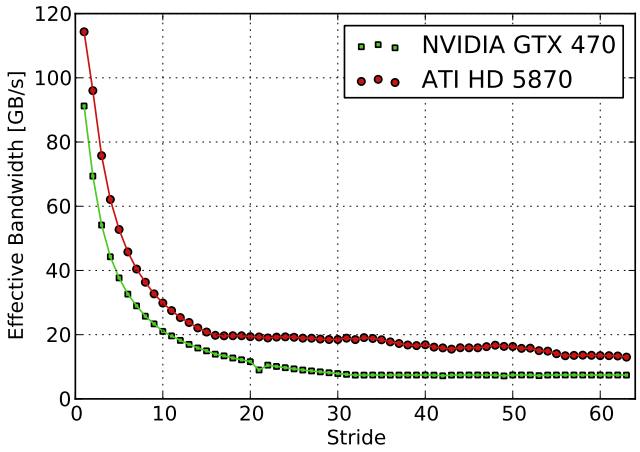


Figure 2.7: Strided access severely impaired the memory bandwidth of the graphics cards included in this test.

When data is stored in global memory, successive words[9] are stored in successive memory banks, but there is only a limited number of memory banks. Work items in a warp ideally read from unique banks without any bank conflicts. However, data is always read in segments of successive words that span over several memory banks, the first part of an array being the first segment etc.. A warp that accesses words stored in two or more segments might have to read both segments in their entirety, spending twice as much time. For example, a warp that reads elements 1 to 32 in an array of length 64 stored in two segments (as opposed to elements 0 to 31) might[10] have to read them both. This is called *misaligned access*. Figure 2.8 gives a graphical overview of both strided and misaligned access patterns.

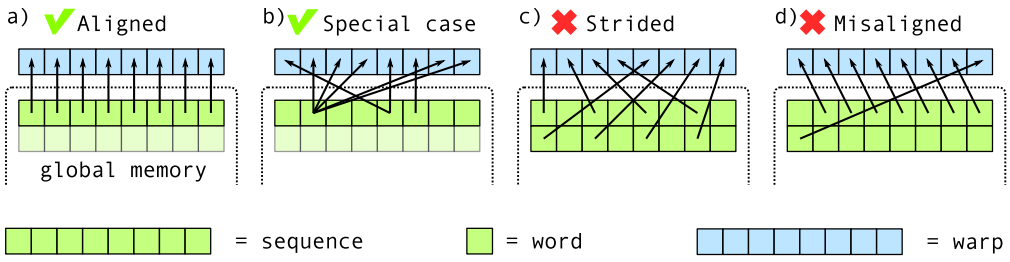


Figure 2.8: **a)** Aligned and coalesced access of global memory. **b)** There is no penalty in the case where the same memory address is accessed by multiple elements of the same warp. **c)** Strided access does not take full advantage of the hardware parallelism due to bank conflicts. The stride in this example is two. **d)** Misaligned access forces the processing elements to read the first sequence in addition to a great part of the last one.

On a final note, some of the kernel's input data may be of a predictable nature, and it should be considered to reconstruct it within the kernel instead of reading it from global memory.

2.3.3 Local Memory

Recall from section 2.1.2 that each compute unit administers a small cache of on-chip local memory. Local memory is substantially faster than global memory[11], and it is a good idea to use local memory for caching data that will be used more than once within a work group or to store some intermediate results. Doing so helps reduce redundant transfers involving global memory and can greatly

[9] *Word* is the term used for the natural unit of data for a given architecture. For example, on a 32-bit system, each word has a size of 32-bits, and memory transfers are therefore done in in segments of 32 bits. A single precision value is 32 bits and therefore has the size of exactly one word on a 32-bit system. A double precision value is 64 bits and therefore has the size of exactly two words on a 32-bit system, but only one word on a 64-bit system. Note that a compute unit loads or stores several words at once.

[10] Some recent graphics cards tend to mitigate or circumvent some of the issues with misaligned access by caching memory transactions [16].

[11] For a 2009 GPU from NVIDIA, the throughput of memory operations was 8 operations per clock cycle, but there was in addition a latency of 400 to 600 clock cycles when accessing global memory. The latency for accessing local memory was approximately a factor of 100 lower. [15].

improve performance. The lifespan of data stored in local memory is the same as the lifespan of the work group that used it.

Data sent to local memory is stored in multiple memory banks and is therefore vulnerable to strided access. Therefore, if multiple addresses of a memory request map to the same memory bank, the accesses are serialized. However, there is no bank conflict between work items accessing the same element in a bank.

When one instruction of a kernel invocation writes a word to local memory, and the successive instruction for some other work item involves reading the same data, there is no guarantee that the given word is actually stored before it is read. However, the work items can be forced to synchronize by raising a *memory fence* in-between the two events. Figure 2.9 displays code and possible output for two such scenarios. Memory fences cost additional clock cycles and force processing elements to idle while waiting for each other. They should therefore be used sparsely.

2.3.4 Execution Paths

Any flow control instruction, such as **if**, **while**, **for**, **switch**, or **do**, can potentially force work items within a warp to take different execution paths. If this happens, the different executions paths will be serialized and executed sequentially, increasing the total number of instructions executed for the warp and leaving a number of processing elements idle. When all the different execution paths have completed, the work items converge back to the same execution path. Substantial performance improvements can be realized by taking care that the code seldom requires work items in a warp to diverge. In other words, a work group can execute disjoint code paths at no extra cost as long as its warps do not.

2.3.5 NDRange and Occupancy

Registers are fast on-chip memory units that store variables declared by the kernel. Each compute unit has a fixed number of registers to distribute among its resident work groups, and as long as there are enough free registers a compute unit will continue to launch concurrent work groups. Local memory is used on a per-work-group basis, meaning there also has to be sufficient local memory in order for several work groups to spawn. When the processing elements on a compute unit invoke a memory transfer for a warp they can put the warp aside and execute instructions for other warps while out-waiting the memory latency. This allows for completely hiding latency from registers and local memory (and to some extent global memory). The more active warps the more latency can be hidden away.

There are hardware limits to the maximum number of active work groups, warps, and work items on a compute unit. The ratio of resident warps to the maximum number of resident warps given by the hardware limit is called *occupancy*, and it is used as a performance gage since GPU devices typically need a certain occupancy to completely hide register latency. Choosing execution pa-

```
in_data = [ 0.  1.  2.  3.  ...  63.  64.  65.  66.  ...
          125.  126.  127.]
```

```
__kernel void flip(
__global const float* in_data,
__global float* out_data,
__local float* cache)
{
     int i = get_local_id(1);
     cache[i] = in_data[i];
     out_data[i] = cache[127-i];
}
```

```
out_data = [ 127.  126.  125.  124.  ...  64.  0.  0.  0.
           ...  2.  1.  0.]
```

```
__kernel void barrier_flip(
__global const float* in_data,
__global float* out_data,
__local float* cache)
{
     int i = get_local_id(1);
     cache[i] = in_data[i];
     // Raise memory fence
     barrier(CLK_LOCAL_MEM_FENCE);
     out_data[i] = cache[127-i];
}
```

```
out_data = [ 127.  126.  125.  124.  ...  64.  63.  62.
           61.  ...  2.  1.  0.]
```

Figure 2.9: Two kernels attempt to flip an array of 128 elements, but the former is not guaranteed to give the correct result due to the lack of a synchronization point. A synchronization point is introduced by adding the command `barrier(CLK_LOCAL_MEM_FENCE)`.

rameters such as the work group size, register usage, and local memory usage is a matter of striking a balance between latency hiding and resource utilization. For example, while having a work group use only part of the available local memory can help hide latency, it often means more redundant memory transfers.

NDRange is a term used in OpenCL to describe the dimensions of the work groups and grid for a specific kernel invocation. While these dimensions can be practical in order to more conveniently project a multidimensional problem, cf. fig. 2.4, they have no impact on performance. The size, however, does: The number of work groups in a grid should be greater than, and preferably a multiple of, the number of compute units so that none of them can idle. In addition, there should be multiple active work groups per compute unit so that work groups that are not waiting for a memory fence can keep the hardware busy. To scale to future devices, the number of work groups per kernel launch should be in the hundreds, as kernels with thousands of work groups will scale across multiple future GPU generations [15]. For kernels where each work group take approximately the same amount of time to execute, the number of work groups should be a multiple of the number of compute units to avoid the last batch of work groups to occupy only a fraction of the compute units.

2.3.6 Loop Unrolling

As this report will show later, loop unrolling can sometimes help to greatly improve performance. Unrolling a loop means increasing the quantity of instructions per iteration by reducing the number of iterations. Doing so not only reduces conditional checks of the loop overhead; it also helps the compiler to coalesce fetch and store instructions if these are placed in adjacent code lines as shown in fig. 2.11. However, doing so also taxes the registers since registers are not freed up until the variables they hold are no longer needed. Figure 2.10 shows the performance differences between the two kernels in fig. 2.11. The two kernels do the exact same job, but one of them a loop has been unrolled by a factor of four. The optimal unrolling factor for a kernel generally is considered to be a

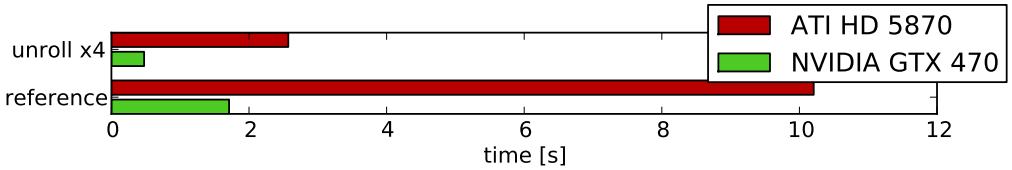


Figure 2.10: Performance for the kernels shown in fig. 2.11 for an array of $N = 256,000$ randomly generated numbers and $loopCount = N$. Unrolling by a factor of four improved performance substantially on the graphics cards included in this test.

```
__kernel void normal_loop(
__global const float* in_data,
__global float* out_data,
uint loopCount)
{
     uint i = get_global_id(1);
     for(uint j = 0; j < loopCount; j++){
          out_data[i] += in_data[j];
     }
}
```

```
__kernel void unrolled_loop(
__global const float* in_data,
__global float* out_data,
uint loopCount)
{
     uint i = get_global_id(1);
     for(uint j = 0; j < loopCount; j+=4){
          float X0 = in_data[j];
          float X1 = in_data[j+1];
          float X2 = in_data[j+2];
          float X3 = in_data[j+3];

          out_data[i] += X0;
          out_data[i] += X1;
          out_data[i] += X2;
          out_data[i] += X3;
     }
}
```

Figure 2.11: The first kernel shows a regular for-loop. The next shows the same loop unrolled by a factor of four (the kernels add some elements of an array to each element of that same array and have no use other than being instructive). Performance comparisons are shown in fig. 2.10.

2.3.7 Native Math Library And Compilation Flags

Some mathematical functions, like `sin`, `cos` and `sqrt` to name a few, can be replaced by native device instructions by appending the `native_` prefix. Native functions typically have better performance compared to their standard versions, but at the cost of precision. The accuracy and behavior of such functions are implementation defined. Figure 2.12 gives an impression of how the `sin` function varies from the `native_sin` function in terms of performance, for example. Figure 2.13 shows the standard deviation of the same functions using C++ generated double precision CPU calculations as a reference.

On a final note, OpenCL offers a number of optional compilation flags that trade precision for speed. Discretion is advised.

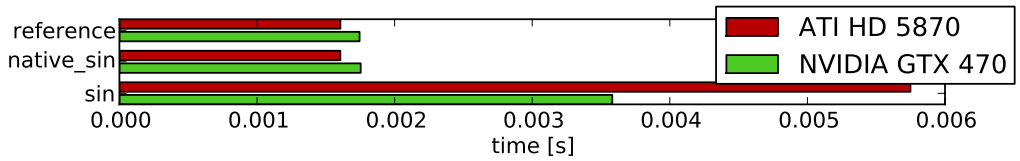


Figure 2.12: Performance for an implementation defined native function on two graphics cards. The reference kernel copies an array of $25,600,000$ randomly generated single precision numbers. The two other kernels calculate `sin` and `native_sin`, respectively, for each element of the same array before copying it. The kernel that calculates `native_sin` is limited by global memory bandwidth.

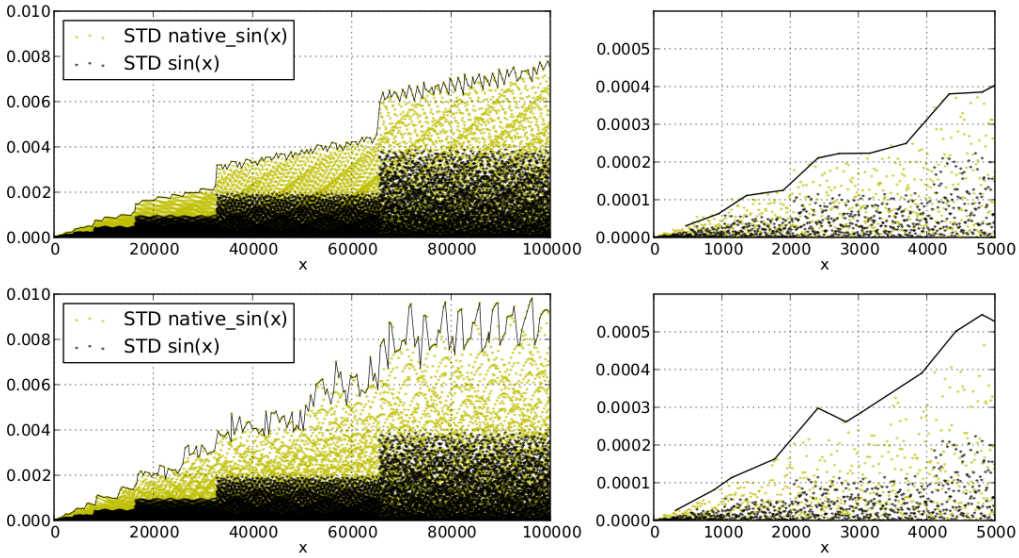


Figure 2.13: The standard deviation for `sin(x)` and `native_sin(x)` in single precision as a function of x using C++ generated double precision CPU calculations as a reference. The continuous black lines trace the maximum values of the standard deviation for `native_sin(x)`. Top: NVIDIA GTX 470. Bottom: ATI HD 5870.

Chapter 3

X-ray Theory

This chapter gives an introduction to the physics of the Debye equation without going into excessive detail. The chapter begins with an explanation of X-ray scattering and continues to explain how this phenomenon can be used to obtain structural information. The chapter proceeds with the derivation of the Debye formula, which can be used to simulate wide angle and small angle X-ray scattering from orientationally averaged systems, such as powders of identical and randomly oriented particles or objects in solution. The last section describes a slightly rewritten and computationally less demanding version of this equation.

3.1 Scattering

When X-rays interact with matter, any given photon has a certain chance to be scattered by the electron cloud that surrounds the atoms. Many materials exhibit periodic structure at the atomic scale and consequently a periodic electron distribution. Constructive and destructive interference occurs when the periodic structures of such materials scatter electromagnetic radiation in or out of phase, respectively. For example, if two initially coherent photons of the same energy are scattered in the same direction, they will experience a relative phase shift ϕ, and if $\phi = 2n\pi$ their amplitudes will interfere constructively ($n \in 0, \pm 1, \pm 2...$). $\phi = (2n+1)\pi$, on the other hand, corresponds to destructive interference. The phase shifts occur because photons are deflected by different atoms when they scatter, introducing spatial variations between their paths. This phenomenon, which falls under the collective term *diffraction*, is fundamental in X-ray scattering.

X-ray diffraction basically involves a source, a monochromator, a collimator, a sample, and a detector. A monochromatic beam of nearly parallel X-rays is incident on a sample and the scattered radiation is measured by a detector at different angles and sample orientations. The resulting plot of intensity vs. angle holds a lot of information about the sample. Moreover, it has peaks for angles of constructive interference, but tends to diminish for angles of low or destructive interference. However, in order to interpret the results it is much more convenient to treat scattering in reciprocal space, or $\mathbf{Q}$-space (as opposed to real-space).

Using $\mathbf{Q}$-space is a convenient way of representing information, and one

which is ideal for studying periodic structures. **Q**-space is the same as Fourier space and has dimensions of inverse length. Both electromagnetic radiation and periodic electron distributions can be described by Fourier theory as superpositions of sines and cosines [18], both of which are described by their respective wave vectors. X-rays are described by a wave vector

$$\mathbf{k}_{Xray} = \frac{2\pi}{\lambda}\widehat{\mathbf{k}}_{Xray} \tag{3.1}$$

where λ is the X-ray wavelength and $\widehat{\mathbf{k}}_{Xray}$ is a unit vector in the direction of the ray. Periodic electron distributions, that is, crystal lattices, are described by reciprocal lattice vectors

$$\mathbf{G} = \frac{2\pi}{\|\mathbf{d}\|}\widehat{\mathbf{d}}, \tag{3.2}$$

where $\mathbf{d}$ is an inter-atomic spacing vector in real space. Figure 3.1 shows a periodic lattice in **Q**-space with an incoming X-ray $\mathbf{k}_i$. Note that the reciprocal lattice is actually the **Q**-space representation of the electron distribution of the real space lattice. The scattered X-ray, denoted by $\mathbf{k}_f$, can take any direction, but the kinetic energy remains constant in the scattering process. The Laue condition [19] states that a necessary condition for constructive interference is

$$\mathbf{Q} \equiv \mathbf{k}_f - \mathbf{k}_i = \mathbf{G}, \tag{3.3}$$

which is true only for certain scattering angles.

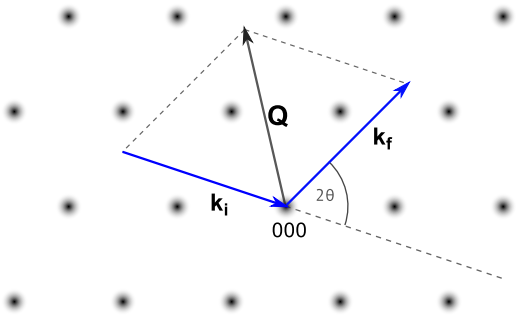


Figure 3.1: An incoming X-ray $\mathbf{k}_i$ is scattered by an atomic lattice plane, which is represented in **Q**-space by a reciprocal lattice. The scattered X-ray is denoted by $\mathbf{k}_f$. If **Q** coincides with a reciprocal lattice vector, there will be constructive interference and a bright, observable reflection in the direction of the corresponding $\mathbf{k}_f$.

The sphere spanned by **Q** in fig. 3.2 is titled the Ewald sphere and has the property that $\mathbf{Q} = \mathbf{G}$ in the places where it intersects with the reciprocal lattice. In other words, X-rays that exit with $\mathbf{k}_f$ such that $\mathbf{Q} = \mathbf{G}$ yield bright reflections.

In basic powder diffraction experiments with a stationary source, the sample and detector can be rotated about collinear axes at angular speeds ω and 2ω, respectively. In such setups the angle 2θ between $\mathbf{k}_i$ and $\mathbf{k}_f$ changes, thus scanning the intensity $I(\mathbf{Q})$ for a specific range of **Q**. The sample is rotated such that **Q** keeps its direction with respect to the reciprocal lattice, so it is only the magnitude of **Q** that changes during the scan. By simple geometry the conversion from

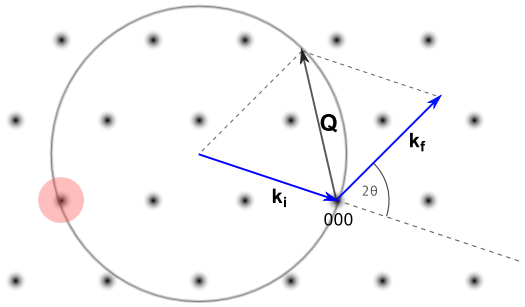


Figure 3.2: The Ewald sphere shows which $\mathbf{k}_f$ can be expected to produce constructive interference by fulfilling the Laue condition $\mathbf{Q} = \mathbf{G}$. In the illustrated case the Ewald sphere intersects with the reciprocal lattice in one single place (marked by a red spot), and the corresponding $\mathbf{k}_f$ may therefore yield a bright reflection.

the angle between $\mathbf{k}_i$ and $\mathbf{k}_f$, namely 2θ, and the corresponding magnitude of $\mathbf{Q}$, is given by

$$Q = 2\|\mathbf{k}\| \sin\theta = \frac{4\pi}{\lambda} \sin\theta \tag{3.4}$$

If the sample is a powder consisting of randomly oriented particles, the reciprocal lattice will take a corresponding number of different orientations around its origin, smearing out discrete lattice points to spherical shells. The resulting intersections with the Ewald sphere are rings that form diffraction patterns like the one shown in fig. 3.3. The same figure shows a cross section of a similar pattern, displaying the intensity as a function of Q. The patterns can be used to find inter-atomic spacings through eqs. (3.1) and (3.2).

The relative phase of a scattered X-ray is generally given by

$$e^{i(\mathbf{k}_f - \mathbf{k}_i)\cdot\mathbf{r}_j} = e^{i\mathbf{Q}\cdot\mathbf{r}_j}, \tag{3.5}$$

where $\mathbf{r}_j$ is the position of the jth atom. For a group of atoms[1] the scattering amplitude is proportional to the structure factor given by

$$F(\mathbf{Q}) = \sum_i f_j(Q) e^{i\mathbf{Q}\cdot\mathbf{r}_j}, \tag{3.6}$$

where $f_j(Q)$ is the presumably spherically symmetric atomic form factor of the jth atom, given by the integral

$$f(\mathbf{Q}) = \int \rho(\mathbf{r}) e^{i\mathbf{Q}\cdot\mathbf{r}} \mathrm{d}^3\mathbf{r}, \tag{3.7}$$

where $\rho(\mathbf{r})$ denotes the electron density. The atomic form factor characterizes the specific atom's electron distribution and thus its ability to scatter X-rays. Furthermore, the scattering intensity $I(\mathbf{Q})$ from a group of N atoms, when disregarding

[1] This is not restricted to a periodic lattice.

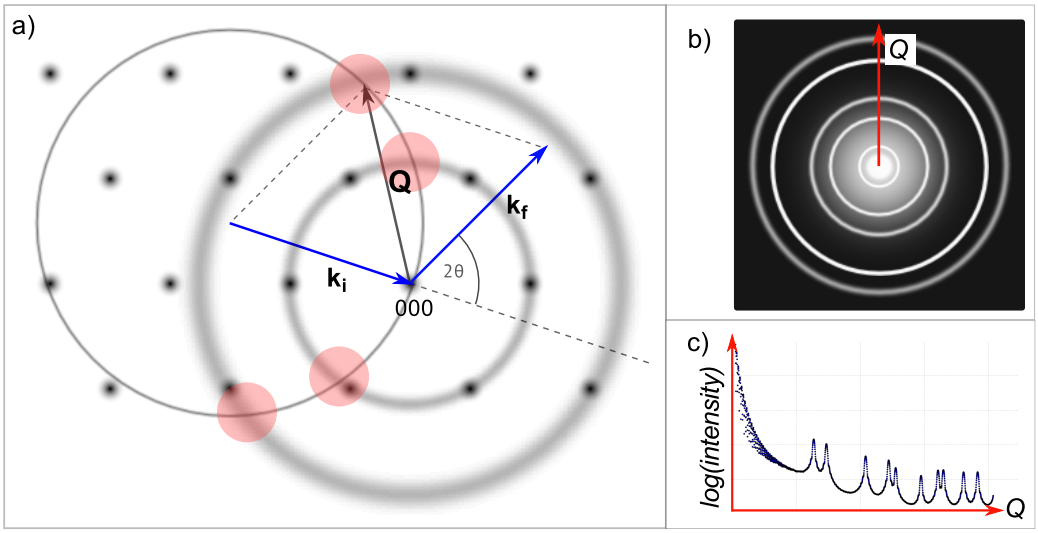


Figure 3.3: In powder diffraction the particles can take any orientation resulting in a reciprocal space filled with spherical shells rather than points (**a**). As a result the intersections between the reciprocal lattice and the Ewald sphere form rings that allow cone-shaped shells of X-rays to escape the powder and give bright reflections. The resulting image is a ring-shaped pattern (**b**). A radial average of a similar pattern (**c**), called a diffractogram, displays the intensity as a function of Q.

correction factors, is equal to the structure factor squared,

$$I(\mathbf{Q}) = |F(\mathbf{Q})|^2 = \sum_{i=1}^{N}\sum_{j=1}^{N} f_i(\mathbf{Q}) f_j(\mathbf{Q}) e^{i\mathbf{Q}\cdot\mathbf{r}_{ij}}, \tag{3.8}$$

where $\mathbf{r}_{ij} \equiv \mathbf{r}_i - \mathbf{r}_j$.

3.2 Debye Formula

Equation (3.8) describes diffraction from one particle made up from N atoms. The Debye formula [19], in contrast, describes diffraction from an aggregate of such particles taking all possible rotations in space. Noting that the orientational average of $e^{i\mathbf{Q}\cdot\mathbf{r}}$ is

$$\left\langle e^{i\mathbf{Q}\cdot\mathbf{r}} \right\rangle = \frac{\sin Qr}{Qr}, \tag{3.9}$$

the Debye formula is readily obtained:

$$I(Q) = \sum_{i=1}^{N}\sum_{j=1}^{N} f_i(Q) f_j(Q) \frac{\sin Qr_{ij}}{Qr_{ij}} \tag{3.10}$$

Note that all vectors have changed to scalar values. The output of the Debye formula is a diffractogram as shown in fig. 3.3, or in other words, a plot of $I(Q)$. Equation (3.9) is based on the following assumptions:

- Incoming X-rays scatter only once (Kinematical approximation [19]).
- X-rays scatter elastically.
- The particles are perfectly randomly oriented.

The last assumption is sometimes problematic in paractice, but the first two are generally considered to be excellent [19].

Exploiting the symmetry properties $r_{ij} = r_{ji}$ and[2] $r_{ii} = r_{jj} = 0$, the Debye formula simplifies to

$$\boxed{I(Q) = \sum_{k=1}^{N} f_k^2(Q) + 2\sum_{i=1}^{N-1}\sum_{j=i+1}^{N} f_i(Q) f_j(Q) \frac{\sin Qr_{ij}}{Qr_{ij}}} \tag{3.11}$$

which is still a computationally heavy function for large N with the number of calculations scaling as $N(N-1)/2 \approx N^2/2 \propto N^2$.

In order for the synthetic results obtained from eq. (3.11) to be comparable to real world measurements, it is necessary to introduce some correction factors [10, 19]. However, such comparisons are not in the scope of this report.

3.3 Specific Formulation

The Debye eq. (3.11) allows for all N atoms of a particle to take different atomic form factors. However, from a computational point of view this is not a desirable situation because each of the $N(N-1)/2$ contributions to $I(Q)$ then depend on two variables, $f_i(Q)$ and $f_j(Q)$, that have to be loaded from memory for all $N(N-1)/2$ computations.

It is both convenient and logical to limit the problem to a small number of atom types since in any case there cannot be more types of atoms in a particle than there are elements, or strictly, ions. In order to benefit from this limitation the atoms can be arranged by their respective form factors as shown in fig. 3.4. The atomic form factors can then be factored out of eqs. (3.9) and (3.11) for the various combinations of atoms.

It is apparent from fig. 3.4 that the combinations of atoms generate both triangular and rectangular regions of computations. Denoting the respective atom types of a particle by m and n, the contribution $I_{mn}(Q)$ to $I(Q)$ from the rectangular regions where $m \neq n$ is given by

$$I_{mn}(Q) = f_m(Q) f_n(Q) V(Q), \tag{3.12}$$

where

$$V(Q) = \sum_{i,j} \frac{\sin Qr_{ij}}{Qr_{ij}}, \tag{3.13}$$

[2] Note that $\lim_{x\to 0} \sin(x)/x = 1$.

and i and j run over all atoms of type m and n, respectively. For the triangular regions where $m = n$, eq. (3.12) reduces to

$$I_{mm}(Q) = N_m f_m^2(Q) + 2 f_m^2(Q) W(Q), \tag{3.14}$$

where

$$W(Q) = \sum_{i=1}^{N_m - 1} \sum_{j=i+1}^{N_m} \frac{\sin Q r_{ij}}{Q r_{ij}} \tag{3.15}$$

and N_m denotes the number of atoms of type m.

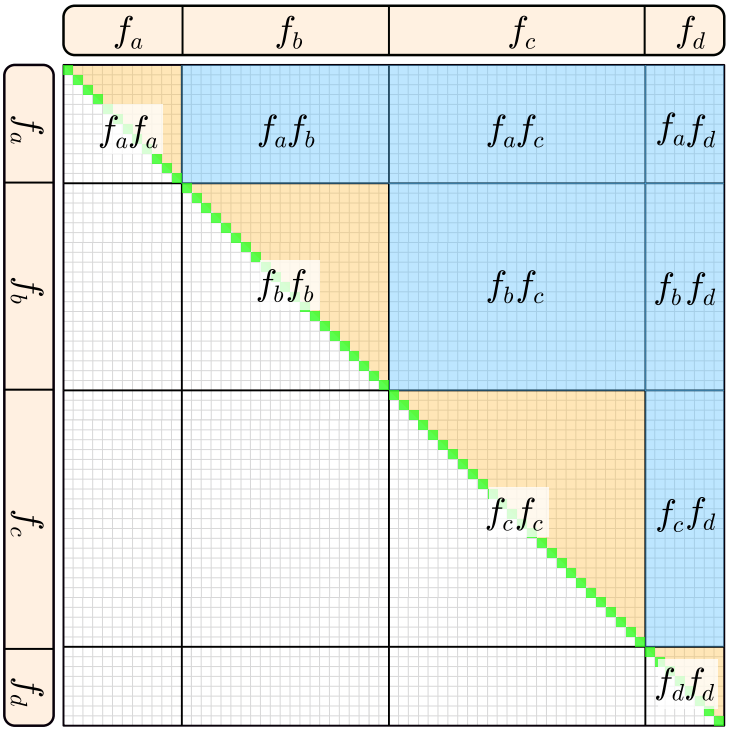


Figure 3.4: The problem space of the Debye equation for a lattice consisting of four different atom types (represented here by four atomic form factors). The total intensity $I(Q)$ is given by eq. (3.17). For the triangle-shaped instances where $f_m(Q) = f_n(Q)$ the intensity contributions can be obtained from eq. (3.14). The contributions from the rectangular tiles are given by eq. (3.12). Due to symmetry, no computations are necessary for the white area.

$I(Q)$ for a powder of the particles can then be expressed as

$$I(Q) = \sum_{m=1}^{M} \sum_{n=1}^{M} I_{mn}(Q) \tag{3.16}$$

where M is the total number of atom types. Using the symmetry property $I_{mn}(Q) =$

$I_{nm}(Q)$, eq. (3.16) reduces to

$$I(Q) = \sum_{m=1}^{M} I_{mm}(Q) + 2 \sum_{m=1}^{M-1} \sum_{n=m+1}^{M} I_{mn}(Q) \tag{3.17}$$

Equation (3.17) is the basis for an algorithm that can calculate $I(Q)$ for any powder of particles that satisfies the assumptions given in section 3.2. The particles can be of any composition and there is no requirement that their lattice is periodic, allowing the inclusion of strain, lattice defects, unusual shapes and other nanoscale features.

Chapter 4

X-ray Scattering Simulation

This chapter describes how a Debye algorithm was implemented in OpenCL to achieve appreciable performance on GPUs. The first section gives some perspective on the problem and outlines the implementation in some detail. There is also a short description of the Python framework used to generate an atomic lattice and administer communication between the host and the GPU.

4.1 Initial Considerations

The Debye equation gets increasingly laborious for large numbers of atoms N with the number of calculations scaling as $N(N-1)/2$. Furthermore, it might be desirable to have hundreds of data points (values of $I(Q)$), if not thousands. Denoting the number of data points by N_Q brings the total number of calculations to $N_Q N(N-1)/2$. For example, a spherical CoO particle with a diameter of 10 nm consists of close to 5.4×10^4 atoms, amounting to more than 1.45×10^9 calculations of the type

$$\operatorname{sinc} Qr_{ij} = \frac{\sin Qr_{ij}}{Qr_{ij}} \tag{4.1}$$

for each data point. A CoO particle with a diameter of 20 nm has more than 4.3×10^5 atoms ($8\times$ the previous particle) and needs a total of 9.3×10^{10} calculations ($8^2\times$ as many as for the previous particle). Furthermore, calculating eq. (4.1) involves several arithmetic and load/store operations. For two atoms i and j with coordinates (x_i, y_i, z_i) and (x_j, y_j, z_j), the distance between them is

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}, \tag{4.2}$$

meaning that each calculation involving two new atoms will load at least six separate values from memory in addition to storing a result. Since load/store operations generally take much longer than arithmetic operations, it is reasonable that memory latency could have some impact on performance. Also note that the Q-values can easily be reconstructed by the kernel as discussed in section 2.3.2. In order to reduce redundant access of memory it is therefore tempting to calculate eq. (4.1) for all Q once r_{ij} has been computed and the positions of two atoms have

been loaded. Furthermore, if each work item was told to compute eq. (4.1) for *one* pair of atoms, the work items could use local memory to share atom position data and to add up temporary results for different values of Q before storing the results in global memory. Continuing, the work items could replace one of their two atoms with a new one and calculate a new r_{ij} before repeating the former steps until all combinations ij had been cycled through. Even though this would help maximize the arithmetic density for the given problem, there is one issue: Several work items and work groups could potentially read from and write to the same locations in memory *at the same time*, which would cause erroneous results due to read/write conflicts. It can be argued that this problem is avoidable by careful loop handling and/or use of parallel reduction. However, it is still possible to aim for high arithmetic density, and thus performance, by using a tuned-down version of the previous example, adjusted to prevent read/write conflicts.

Equation (4.1) contains a sine, a square root and a division, all of which can be substituted by native functions as described in section 2.3.7. However, the accompanying precision losses must be considered. Loop unrolling should be implemented (section 2.3.6), and the grade of unrolling should be a variable parameter to allow for experimentation. It should also be possible to indirectly adjust the usage of local memory and registers to ease the search of a balance between occupancy and usage of these resources. The implementation should also show good scalability to future GPUs. One way to achieve this is to split the workload between at least several hundred work groups as explained in section 2.3.5.

4.2 Dealing With Single Precision

One important attribute of most consumer grade graphics cards is that they are optimized for single precision data (i.e. 32-bit words). Note that graphics cards have their origin in the gaming industry where they were only needed to render graphics without the special need for high precision. In numerics, however, precision is often crucial.

Precision can be defined as the smallest floating point number ϵ that, when added to the floating point number 1.0, produces a number different from 1.0. 32-bit single precision floating point numbers have $\epsilon \approx 1.19 \times 10^{-7}$, and double precision 64-bit floating point numbers have $\epsilon \approx 2.22 \times 10^{-16}$ [20]. While it is possible to do calculations in double precision on recent graphics cards, the theoretical FLOPS value is usually much lower than for single precision, cf. table 2.1. The implementation of the algorithm presented in this report is therefore written for single precision computations, and rather tries to reduce the likelihood of erroneous results that arise from adding values of very different magnitude[1]. Summarizing the last two sections, the algorithm should in particular focus on the following qualities:

- Performance

[1] Recall that the Debye equation involves a double sum, and therefore a significant number of additions.

- Precision
- Scalability to future GPUs.

4.2.1 Parallel Reduction

Parallel reduction is a way of calculating the sum of the elements of an array. It attempts to decrease the chance of adding values of substantially different magnitudes by adding adjacent values in several steps. The principle works best for arrays that do not show pronounced trends, such as increasing values along the array. Figure 4.1 illustrates the working principle of parallel reduction. Furthermore, parallel reduction is very well suited for parallel implementations.

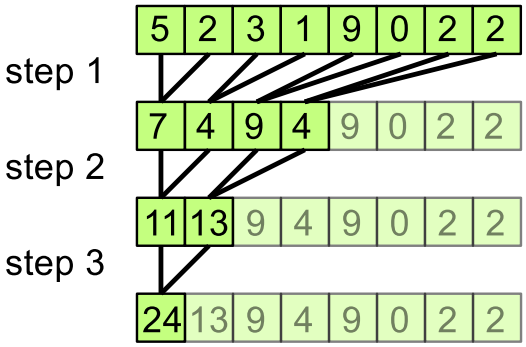


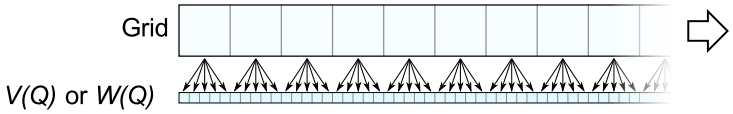
Figure 4.1: In parallel reduction adjacent values of an array are added together in several steps. The array of size eight above requires three steps to produce the result. The final step adds the values 13 and 11, whereas a more naive implementation might add up the elements sequentially and finally add 22 and 2, for which the difference is greater, thus leaving the addition more vulnerable to precision errors. The size of the array has to be a power of two. Note that the stride is one (section 2.3.2).

4.3 Algorithm

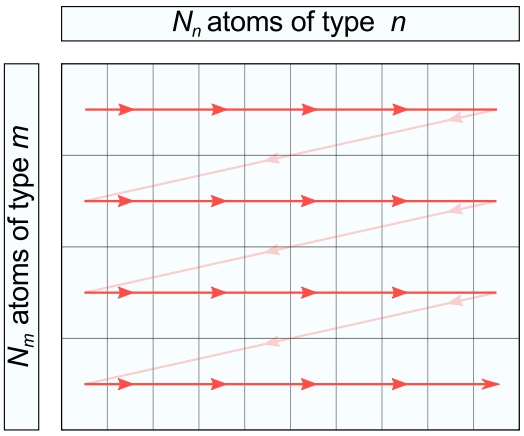
Two different kernels have been written in accordance with eq. (3.17). One of them, the *triangle algorithm*, computes $W(Q)$ in eq. (3.14). The second, called the *rectangle algorithm*, computes $V(Q)$ in eq. (3.12). The names originate from the two different shapes of the regions where $m = n$ or $m \neq n$ as shown in fig. 3.4. This section gives a description of the kernels.

The two kernels use the exact same principles except that the triangle algorithm exploits symmetry properties, effectively halving the time it takes to execute. Each work group is responsible for calculating $V(Q)$ or $W(Q)$ for $N_{Q,wg}$ values of Q, cf. fig. 4.2a. $N_{Q,wg}$ is one of the variable parameters of the algorithms, and by increasing it the number of memory transfers is reduced by a factor close to $1/N_{Q,wg}$, thereby increasing the arithmetic density.

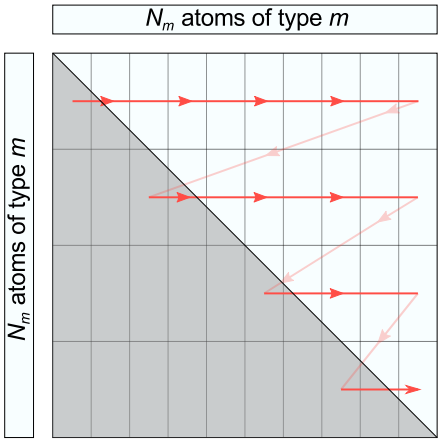
When a work group calculates $V(Q)$ or $W(Q)$ it divides the problem into multiple sub-regions, called *tiles*, as shown in figs. 4.2b and 4.2c. Each tile loads



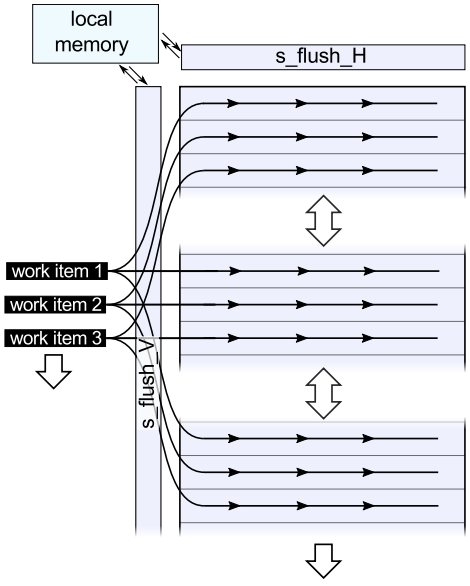
(a) The grid: Each work group computes $V(Q)$ or $W(Q)$ for $N_{Q,wg}$ successive values of Q.



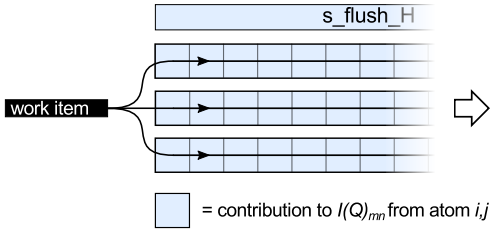
(b) The work groups of the *rectangle algorithm* cycle through all available tiles before sending any results to global memory.



(c) The work groups of the *triangle algorithm* avoid any tiles that can be omitted due to symmetry.



(d) The tile: All work items in a work group read atom positions from the same pool in local memory.



(e) Each of the N_{wi} work items has the ability to calculate $\operatorname{sinc} Qr_{ij}$ for several r_{ij} after loading an atom position from `s_flush_H`.

Figure 4.2: Two slightly different algorithms, namely the *rectangle* and the *triangle algorithm*, calculate eq. (3.15) or eq. (3.13) respectively.

a number of atom positions from global memory ($stride = 1$) and stores them in the local memory cache in the form of two arrays[2] `s_flush_V` and `s_flush_H`, cf. [21]. The number of atom positions stored in these two arrays defines the size of the tile. Their respective sizes are multiples of the number of work items per work group since they are accessed by all work items at once.

Throughout the lifetime of a tile the N_{wi} work items in the work group need to access the same atomic data several times, but since it is cached and freely accessible, it no longer has to be loaded from global memory for each request. The way this works is illustrated in fig. 4.2d: After transferring the data from global to local memory, each work item loads data corresponding to a few atom positions from `s_flush_V` such that the entire array is in use. Next, the work items all load a single atomic position from `s_flush_H` and calculate eq. (4.1) for $N_{Q,wg}$ different values of Q. This procedure is followed until each item in `s_flush_H` has been processed (cf. fig. 4.2e), at which time the data in `s_flush_H` is replaced and the next tile is initialized. Note that N_{wi} is one of the variable parameters of the algorithms, as are the number of atomic positions in `s_flush_V` and `s_flush_H`.

Increasing the size of `s_flush_V` by a factor K makes each work item calculate eq. (4.1) for $N_{Q,wg}$ values of Q for K atom positions *for each atomic position* loaded from `s_flush_H`. This mechanism has been implemented as a fully unrolled for-loop that is unrolled by a factor $KN_{Q,wg}$. Note that K equals the length of `s_flush_V` divided by the number of work items per work group.

When several work items calculate $\operatorname{sinc} Qr_{ij}$ for the same value of Q (but different r_{ij}), the results are stored in an array in local memory of length N_{wi} to prevent the situation where several work items attempt to add their results to the same memory address, cf. section 4.1. Moreover, there has to be $N_{Q,wg}$ such arrays, meaning $N_{Q,wg}$ is capped since there is a finite and relatively small amount of local memory. The last action a the tile takes is to sum elements of each array using parallel reduction and add the results to a vector in global memory, cf. fig. 4.2a. Afterwards, the data stored in `s_flush_H` is replaced with another segment of atomic data, defining a new tile.

Note that the procedure of calculating $V(Q)$ or $W(Q)$ involves the sequential addition of K instances of eq. (4.1). These (small) sums are added to the vectors in local memory for each atomic position in `s_flush_H`, at which time the elements of each array are summed using parallel reduction. Each tile follows the same procedure and always finishes by adding $N_{Q,wg}$ contributions to $V(Q)$ or $W(Q)$ in global memory. The combined effects of parallel and segmented addition help mitigate the errors resulting from working in single precision.

The triangle algorithm avoids computations for tiles that can be omitted due to symmetry as shown in fig. 4.2c. For those tiles that are split between the excludable and non-excludable regions, only the relevant computations are carried out.

On a final note, any `sin` function or floating point division has been replaced by `native_sin` and `native_divide`, respectively (cf. section 2.3.7).

[2] The `V` and `H` in `s_flush_V` and `s_flush_H` stand for *vertical* and *horizontal*. `s_flush` comes from *store* and *flush*.

4.4 Limitations

The algorithms discussed in the prevision section have no limit on the number of atoms for which they can compute $V(Q)$ and $W(Q)$. The hardware, however, does. Figure 4.3 shows the relation between use of global memory and the number of atoms. In the event that a calculation requires more atom positions than global memory allows, the problem can simply be split into more manageable tasks, i.e. more instances of eqs. (3.13) and (3.15).

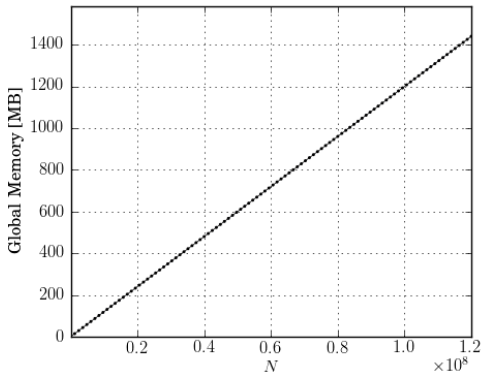


Figure 4.3: The algorithms work for relatively large numbers of atoms due to low usage of global memory. The NVIDIA GTX 470 has 1280 MB of global memory, and the ATI HD 5870 has 1024 MB.

Of greater consequence is the limited availability of local memory and registers even on recent graphics cards. The impact these limits have on performance is discussed in section 5.1.

4.5 Framework

A Python framework is used to produce results from the algorithms and administer any related tasks. Figure 4.4 portrays the various dependencies of the executable script that constitutes the framework.

The program starts out with the assembly of a particle. This is done by generating a lattice shaped like a rectangular prism of a particular length, width and height, cf. fig. 4.5a. Different atom types are stored in different arrays to keep them separate, which is necessary in order to use eqs. (3.12) and (3.14). Next, the lattice is cropped so as to form any desired particle shape, for example a sphere as shown in fig. 4.5b. The final step before the GPU algorithm is put to work involves padding each array with a few atomic positions such that the array lengths are multiples of `s_flush_V` and `s_flush_H`, respectively, and thereby fit the NDRange, cf. sections 2.3.5 and 4.3. The additional atoms are given positions that are so far away from the original particle and each-other that they will never give any noticeable contributions to the results, cf. fig. 4.5c. Note that eq. (4.1) de-

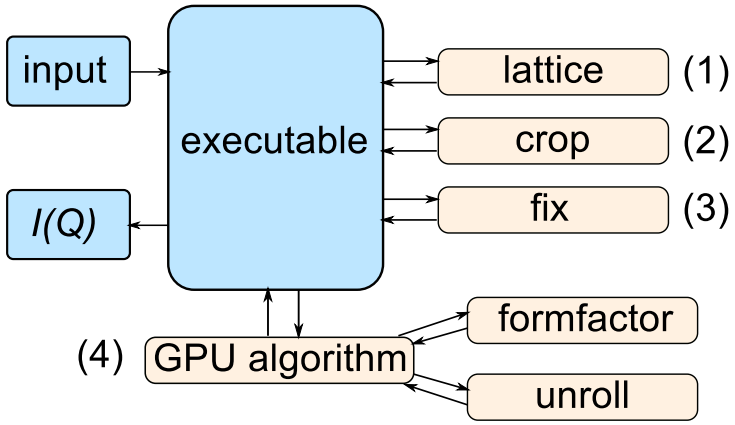
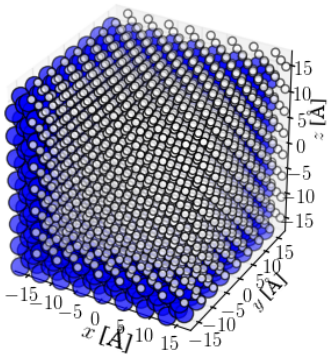


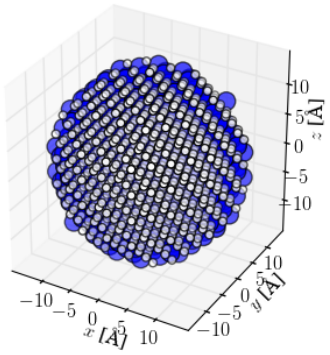
Figure 4.4: The various dependencies of the executable script that administers the subtasks. Based on the input variables, a Python script generates a lattice (1) and crops it to form a particle (2). Afterwards a few additional atoms are added to help the arrays match the specified NDRange parameters (3). Then the GPU calculates $I(Q)$ (4) with the help of a script that automatically unrolls the algorithm's most critical for-loop and returns a kernel to the GPU. The atomic form factors are loaded from a simple external script, written by Jostein B. Fløystad, and based on parameterizations by Waasmaier and Kirfel [22].

clines quickly for large inter-atomic distances, as shown in fig. 4.6. It appears that it is seldom required to add more than a few hundred atoms per array, regardless of the particle size.

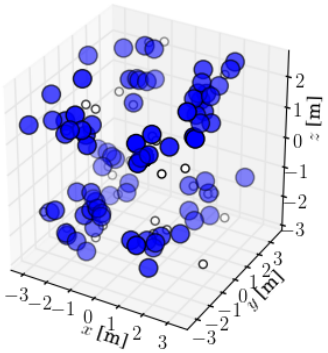
After the arrays containing particle position data have been finalized they are loaded into global memory of the GPU device. The problem is then divided into as many instances as required by eq. (3.17) to factor out the atomic form factors. Next, the GPU calculates the contributions to $I(Q)$ tile by tile until the problem has been solved, at which time the result can be stored to a file or plotted as shown in fig. 4.7.



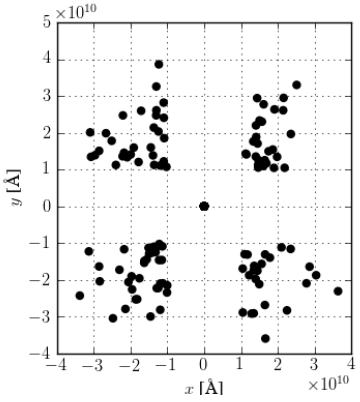
(a) A CoO lattice with the shape of a rectangular prism.



(b) A spherical CoO particle cut out from the lattice in a.



(c) A few additional atoms have been added to locations far away from the particle, which is located in the origin. Note the scale.



(d) Cross section view of the plot to the left. Note that none of the additional atoms are anywhere near the particle in the center, and that the inter-atomic distances are relatively large.

Figure 4.5: The program constructs the desired particle in two steps a and b. The third step adds a few atoms to make the array lengths multiples of `s_flush_V` and `s_flush_H`, respectively, to fit the NDRange: c, d.

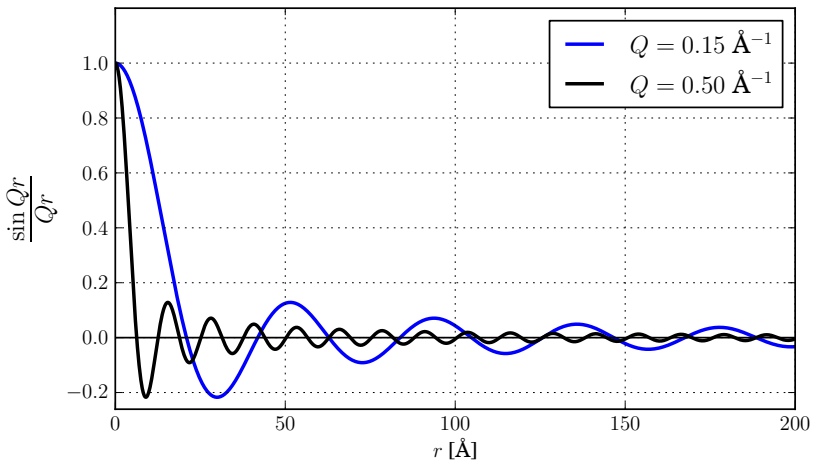


Figure 4.6: The trend of the sinc function in eq. (4.1) is that it has a declining amplitude for increasingly large r. As a consequence, the far-away atoms that are added to the atomic arrays to make them fit the NDRange will not give any significant errors.

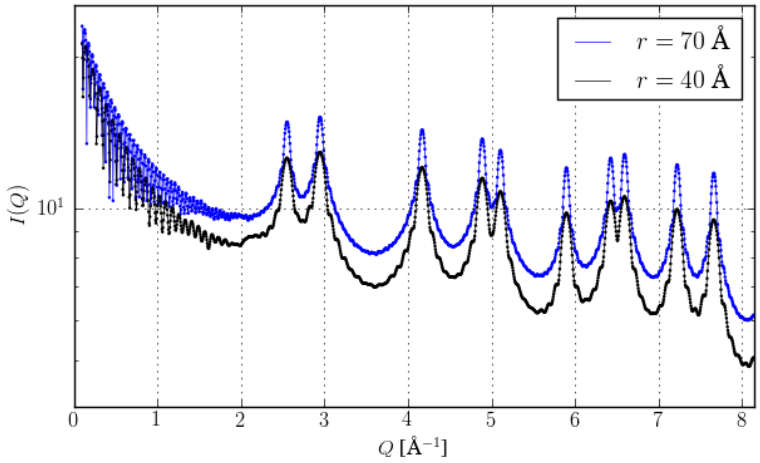


Figure 4.7: The result $I(Q)$ for spherical CoO particles with radius 40 and 70 Å (close to 1.4×10^4 and 1.5×10^5 atoms, respectively). The plots have 1456 Q-points. The calculations for the largest particle took 8 minutes and 31 seconds to compute, and the calculations for the smallest particle took 18 seconds. The section up to $Q = 2$ [Å] gives the Small Angle X-ray Scattering (SAXS), and the rest constitutes the Wide Angle X-ray Scattering (WAXS). The peaks in the WAXS region correspond to inter-atomic distances, while the SAXS region corresponds to larger distances such as particle size and separation. The largest particle has sharper and more intense reflections due to a higher degree of long-range order (although "long" in this context still is on the nano-scale).

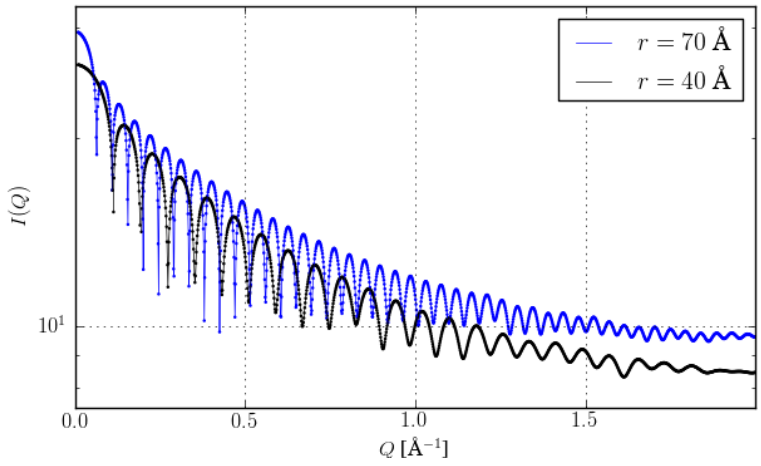


Figure 4.8: Small Angle X-ray Scattering (SAXS) for spherical CoO particles with radius 40 and 70 Å. The plots correspond to the form factor squared, $|f(\mathbf{Q})|^2$, of spheres with the same radii as the particles and where $\rho(\mathbf{r}) = \ constant$ (cf. eq. (3.7)).

Chapter 5

Results and Discussion

The previous chapters lead to the description of two kernels, capable of calculating X-ray scattering for relatively complex particles. The following discussion focuses on the performance and precision of these algorithms, and how well they can be expected to scale for future graphics cards.

It is important to note that the tests performed in this chapter were done using a specific graphics card, and that any results presented here may differ from other GPUs and systems. The test system comprised a NVIDIA GTX 470 GPU combined with a 2.8 GHz Intel Xeon X3460 CPU, both of which are in the same price range. The GTX 470 has 48 KB of local memory and 32 K 32-bit registers for each of its 14 compute units. All performance measurements have been done using a grid containing $2 \times 14 N_{Q,wg}$ work groups, cf. section 2.3.5.

5.1 Performance

Getting appreciable performance on graphics cards is often more involved than just following the guidelines in section 2.3. There are some parameters that affect performance in ways that are hard to predict, such as the number of work items per work group, N_{wi}, and the amount of shared memory and registers allocated to each work group. These variables are closely tied to the algorithm parameters discussed in this text, and finding a good balance between them has proven to be crucial. Table 5.1 lists the significant input variables and the consequences of changing them.

As shown in fig. 5.1, increasing the size of `s_flush_H` results in performance losses. The most prominent effect of increasing the size of `s_flush_H` beyond N_{wi} is increased use of local memory, which it is better to allocate for other purposes. It also decreases the required number of tiles for each work group, and thus the number of parallel reductions[1] that take place. However, the parallel reductions are carried out relatively few times in comparison to other operations, and they are unlikely to limit the performance of these kernels. This leads to the conclusion that the size of `s_flush_H` should be set to its minimum value, i.e. N_{wi} to avoid excessive use of local memory.

[1] Recall that several arrays are summed using parallel reduction after processing each tile.

Table 5.1: Comparison of the parameters of the rectangle and triangle algorithms that affect performance, cf. section 4.3.

Input parameter	Resource	Effect of incrementing
N_{wi}	Registers, local memory	Sets the minimum size of `s_flush_V` and `s_flush_H`. May reduce the number of resident work groups on a compute unit.
Size of `s_flush_V`	Registers, local memory	Less redundant memory transfers but taxes local memory. Unrolls by a factor K at the cost of additional register usage, cf. sections 2.3.6 and 4.3. Increases arithmetic density.
Size of `s_flush_H`	Local memory	Increasing does not contribute to fewer redundant memory transfers. Consumes more local memory.
$N_{Q,wg}$	Registers, local memory	Unrolls by a factor $N_{Q,wg}$ at the cost of registers and local memory. Increases arithmetic density.

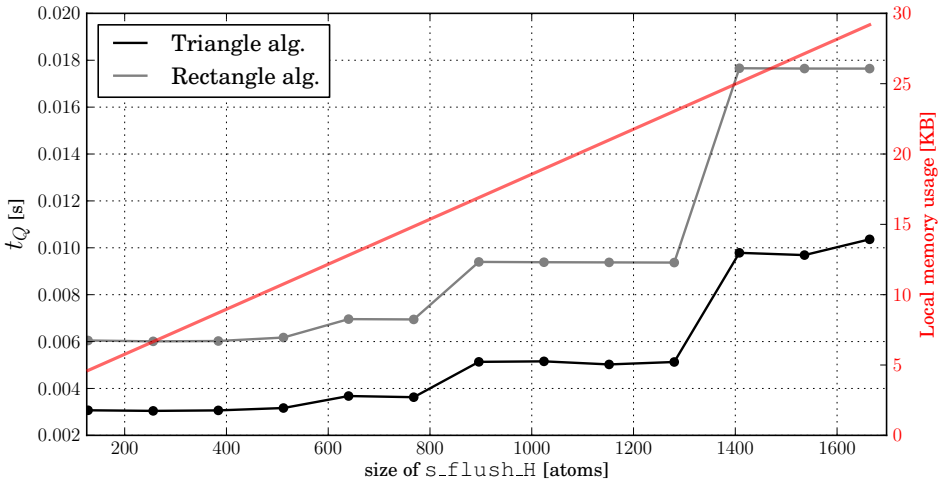


Figure 5.1: Increasing the size of `s_flush_H` eventually resulted in performance losses. The plot shows how much time it took to calculate eqs. (3.13) and (3.15) for *one* value of Q, denoted by t_Q, as a function of the size of `s_flush_H`. All other parameters were set to their default (i.e. minimal) values. The behavior can be explained by excessive use of local memory. Note the sudden decrease in performance (and increase in t_Q) where the usage of local memory exceeds half of the available amount, i.e. 24 of 48 KB. At this point it is only be possible for the compute units to host a single concurrent work group, meaning that other work groups cannot be serviced while waiting for memory fences (cf. section 2.3.5).

There are three remaining parameters: $N_{Q,wg}$ and the size of `s_flush_V` can be expected to have some positive effects on performance since they both contribute to higher arithmetic density and unrolling, cf. section 4.3. Besides, it is encouraged to experiment with different work group sizes, N_{wi} [16]. However, the size should be a power of two due to the parallel reduction (cf. section 4.2.1). Furthermore, the maximum number of work items per work group for the NVIDIA GTX 470 is 1024, and the warp size is 32. Given these constraints, the number of work items per work group should be either 32, 64, 128, 256, 512, or 1024. The other two parameters[2], $N_{Q,wg}$ and K, can take any values as long as there is enough local memory and available registers to launch at least one work group.

The performance for all possible combinations of the three parameters have been measured using a script that runs through any specified ranges of the input variables. A graphical representation of some of the results is shown in fig. 5.2. Table 5.2 shows the combinations of parameters that achieved best performance for each of the work group sizes.

Table 5.2: Comparison of the optimization parameters that resulted in the best performance. t_Q is the time it takes to calculate a single instance of eq. (3.15) or eq. (3.13) for one value of Q. The results have been measured for $N \approx 5 \times 10^4$ atoms and normalized to match that of a particle with exactly $N = 50.000$ atoms. R denotes the ratio between the time spent by the rectangle and the triangle algorithms. One combination produced an error and has been marked with N/A.

	Triangle algorithm				Rectangle algorithm				
Size of work group	t_Q [ms]	$N_{Q,wg}$	K	Local Mem. [KB]	t_Q [ms]	$N_{Q,wg}$	K	Local Mem. [KB]	R
1024			N/A		99.17	3	1	45.068	N/A
512	42.07	7	3	47.132	80.92	11	2	47.148	1.92
256	39.28	26	4	47.208	78.53	23	4	44.124	2.00
128	40.08	18	6	23.624	79.06	31	3	24.188	1.97
64	57.50	26	12	20.072	109.98	51	6	20.428	1.91
32	113.11	40	18	15.008	209.16	16	5	5.184	1.84921876661

Even though the local memory spent per Q-value is lower for small N_{wi}, the best times were achieved with $N_{wi} = 256$ and $K = 4$ for both algorithms, and with $N_{Q,wg} = 26$ and 23 for the triangle and the rectangle algorithms, respectively. Note that the triangle algorithm spends approximately half the time of the rectangle algorithm, which it should. Also, there is a tendency for the work groups to occupy either all or half of the local memory. Kernels that use only half can launch two concurrent work groups per compute unit, which is beneficial due to memory fences that are used in the algorithms (cf. section 2.3.5).

The optimization routines were carried out for $N \approx 5 \times 10^4$. A spot check for $N \approx 1 \times 10^5$ that was carried out for $N_{wi} = 256$ yielded the same ratios between the results as for $N \approx 5 \times 10^4$. Furthermore, in terms of performance, the only significant difference between varying N is the number of tiles that need to be calculated for each work group. Based on these findings the optimization routines for $N \approx 5 \times 10^4$ have been assumed to be valid for considerably greater

[2]The meanings of $N_{Q,wg}$ and K are clarified in table 5.1.

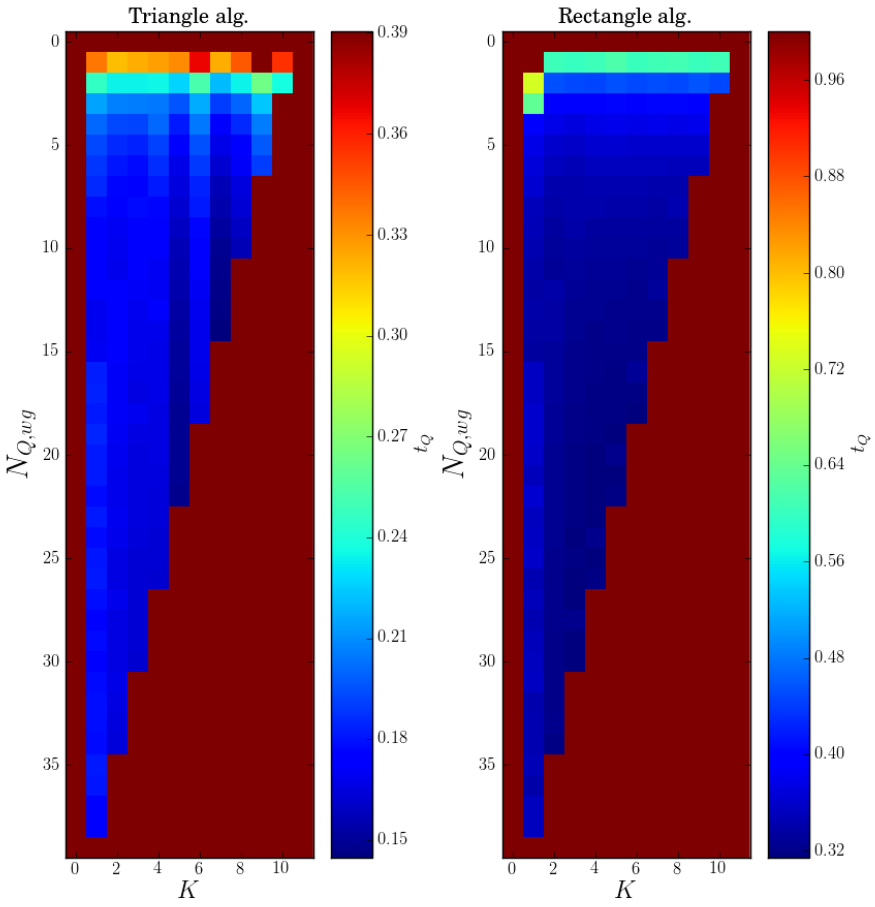


Figure 5.2: With a work group size of 256 the optimization parameters can take a multitude of different combinations. Recall from section 4.3 that K is an unrolling factor that is equal to the size of `s_flush_V` divided by N_{wi}. t_Q is the time it takes to calculate a single instance of eq. (3.13) or eq. (3.15) for one value of Q. t_Q has only been measured for those combinations of the parameters that cause a work group to use less than 48 KB of local memory, since exceeding the local memory limit results in an error.

values of N.

Figure 5.2 needs some further explanation. The dark red areas represent those combinations of K and $N_{Q,wg}$ that exceed the local memory limit of 48 KB or fall outside of the chosen parameter ranges. Furthermore, it is a trend that the performance increases for high $N_{Q,wg}K$, i.e. where the most of the local memory is utilized. It is therefore tempting to conclude that local memory is the major bottleneck. However, investigation of the kernel source code for the best-performing combinations of K and $N_{Q,wg}$ revealed register usage close to the maximum limit of 32 KB, with the complete unrolling (cf. section 4.3) requiring more than $N_{wi}N_{Q,wg}K$ registers per work group. Moreover, Gelisio et al. [2] report calculating $I(Q)$ in batches of 24 Q-steps on a NVIDIA GTX 285 graphics card with only 16 KB of local memory per compute unit, and go on to suggest that local memory is an important performance limiter for their calculations. Better control of register usage should therefore be implemented for the triangle and rectangle algorithms to investigate whether excessive register usage is a limiting factor[3] for high K and $N_{Q,wg}$. This could be realized by use of separate unrolling factors for these parameters to prevent potentially excessive register usage while still increasing the arithmetic density (cf. section 2.3.6). The resulting algorithm should initially be run with no unrolling at all to find the optimal values of K and $N_{Q,wg}$, and then optimized for the two unrolling factors.

Figure 5.3 shows performance comparisons of the initial and the optimized GPU algorithms. A plot for a CPU based double precision equivalent of the triangle algorithm written in C has also been included. This CPU code was programmed for performance, calculating N_Q Q-steps for each new r_{ij}, but has not been as carefully constructed as the GPU algorithms. The optimized triangle algorithm is three orders of magnitude faster than the CPU code, and it is uncertain whether this performance gap can be attributed to insufficient optimization of the CPU code alone. Moreover,the optimized GPU codes are about two to three times as fast as the initial, or default, algorithms where $K = N_{Q,wg} = 1$.

In order to further improve performance, the algorithm could be modified to ignore calculations involving padding atoms, thus eliminating the need for the framework routine that places these atoms around the lattice. The extra logic needed for this would influence the performance gain. There might also be a marginal benefit in precision.

On a final note, the efficiency of the Python framework has proven sufficient for the purpose of conducting performance tests, and generally takes only a small fraction of the time of the subsequent X-ray simulation.

5.2 Precision

In order to measure the accumulation of single precision errors in the triangle algorithm a plot of $I(Q)$ for $N = 13700$ atoms[4] was compared to a corresponding

[3] NVIDIA's Visual Profiler, a kernel benchmark program, does in fact say that registers are limiting the occupancy.

[4] The Co atoms of an CoO lattice were used.

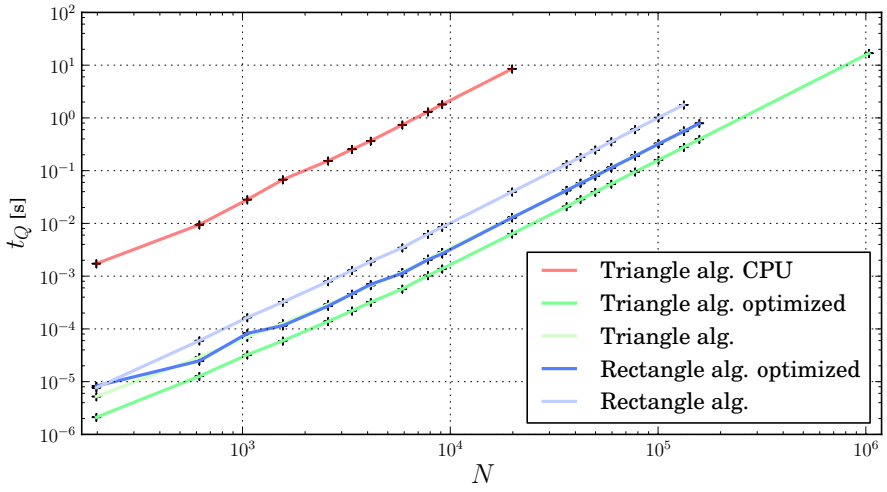


Figure 5.3: t_Q as a function of N for the rectangle and triangle algorithms before and after optimizing $N_{Q,wg}$, K, and N_{wi}. N excludes atoms that were added after the crop, cf. section 4.5. The optimized triangle algorithm achieves more than a $1000\times$ speedup over corresponding CPU algorithm. Furthermore, it is over twice as fast as the default algorithm with $K = N_{Q,wg} = 1$. The scaling factor is 2 for all alogrithms. The optimized rectangle algorithm overlaps the plot for the default version of the triangle algorithm to some extent.

plot generated by a double precision CPU code implemented in C. Figure 5.4 shows the standard deviation of the difference between the two data sets, and it is assumed that the double precision implementation has substantially higher accuracy. The double precision code has been optimized with respect to precision to make this assumption more valid. The standard deviation gives a relative error of 5×10^{-5} for the small feature at $Q \approx 5.8$ Å and a relative error of 5×10^{-6} for the largest peak at $Q \approx 2.5$ Å. Note that the CPU algorithm does not rely on padding atoms to meet NDRange requirements, and, moreover, that simulations for larger values of N did not increase the relative error.

Several factors limit the intensity resolution of real world diffraction experiments and introduce errors. Examples of such factors are instrument resolution and correction factors. Measurements where the error is below a few percent are often considered to be excellent. The error implied by fig. 5.4 is therefore relatively low in comparison. The purpose for generating simulated X-ray diffraction models is to compare the output data to real world results, and if there is already an error of a few percent, an additional relative error of 5×10^{-5} is not going to make much difference.

The GPU algorithms replace the `sin` function and the division operator in eq. (4.1) by native functions. The precision of `native_sin` is shown in fig. 2.13. According to the data, the simulations presented in this report do not introduce relative errors greater than 0.01% ($r_{ij} < 500$ Å). However, for large values of Qr_{ij} the native function should be substituted by the more accurate and much more

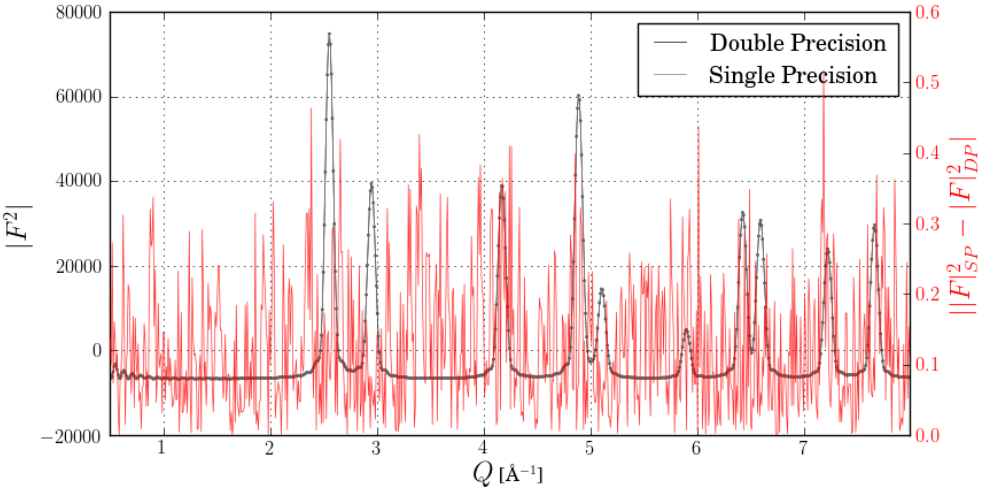


Figure 5.4: The standard deviation between the triangle algorithm and a precision optimized CPU code for a particle of $N = 1.3 \times 10^4$ atoms. The standard deviation gives a relative error of 5×10^{-5} for the small feature at $Q \approx 5.8$ Å and a relative error of 5×10^{-6} for the largest peak at $Q \approx 2.5$ Å

computationally heavy `sin` function. Finally, the GTX 470's `native_divide` function has the same accuracy as the normal implementation.

5.3 Scaling

Section 2.3.5 states that a kernel invocation should result in thousands of work groups in order to scale for multiple generations of future GPU devices. Given that that the number of work groups is $N_Q/N_{Q,wg}$, this is only the case for either high values of N_Q or low values of $N_{Q,wg}$. For the triangle and rectangle algorithms discussed in this report the most efficient combinations of parameters use $N_{Q,wg} = 26$ and 23, respectively. However, the GTX 470 is one of the first architectures that allows for multiple kernels to run simultaneously on one device. Equations (3.13) and (3.15) can therefore be split into as many sub-equations as necessary for a given system and invoked simultaneously as multiple kernels to get the desired amount of work groups. There are in addition other ways to split the workload over more work groups, such as splitting the work done by each work group *within* the kernel rather than over multiple kernels. In the latter approach each work group would process a given number of tiles to match any need.

Chapter 6

Conclusion and Further Work

A program that calculates the powder diffraction pattern of nano-sized particles has been written to run in single precision on GPUs using the OpenCL API. The performance and precision of this software has been compared to a CPU based program running in double precision. Additionally, several optimization techniques have been employed to better utilize the hardware resources present on the graphics card.

Findings include a significant speedup from the CPU based double precision algorithm to the optimized GPU based single precision algorithm at a tolerable precision loss. Moreover, there was a 2 - $3\times$ performance gain when optimizing the GPU algorithms with respect to usage of registers and fast, on-chip local memory. It was also argued that excessive register usage could be a key performance limiting factor, as well as a limited local memory cache as suggested by Gelisio et al.

The purpose of simulating powder diffraction patterns is of course to do comparative analyses with experimental results, and future work could therefore include writing of GPU-accelerated software for such purposes, for example by matching experimental data to previously generated diffraction patterns.

Future work could also include rewriting the GPU algorithms to see if even better performance can be achieved by increasing control over the loop unrolling and thereby use fewer registers. Additionally, the algorithm can be modified to ignore calculations involving padding atoms so as to slightly improve both performance and precision. Finally, the algorithm can be implemented to run on systems comprising multiple GPUs.

Appendix A

Calculating Theoretical Performance

A.1 FLOPS

Floating Point Operations Per Second (FLOPS) is a commonly used measure of the theoretical performance of a system or device. However, there are many other factors that influence actual instruction performance. This paragraph describes how to calculate FLOPS for a General processing Unit (GPU).

Modern GPUs are often measured by the number of Fused Multiply Add (FMA) instructions they can calculate per second times two. The FMA instruction calculates an expression of the type $ab + c$, and is therefore both a multiplication operation and an addition operation in one. The performance in FLOPS is then given by

$$2 \times (number\ of\ cores) \times (shader\ frequency), \tag{A.1}$$

where $shader\ frequency$ corresponds to the frequency of each core (i.e. processing element). For example, the NVIDIA GTX 470 has 14 compute units with 32 cores each running at a frequency of 1.215 GHz, resulting in a performance of $2 \times 14 \times 32 \times 1.215\,\text{GFLOPS} = 1088.64\,\text{GFLOPS}$.

A.2 Memory Bandwidth

Memory Bandwidth (MBW) is the rate at which data can be read from or stored into a semiconductor memory by a processor. For graphics cards, memory bandwidth is associated with the speed at which data is transferred between global memory and the cores (i.e. processing elements). The theoretical peak memory bandwidth for a graphics card is given by

$$(memory\ frequency) \times (memory\ bus\ width), \tag{A.2}$$

where $memory\ bus\ width$ denotes the amount of data transferred per global memory clock cycle. For example, the global memory of the GTX 470 operates at a frequency of 3.348 GHz and has a 320-bit $= 40$ bytes bus width, and therefore peaks at $3.348 \times 40\text{GB/s} = 133.92\text{GB/s}$.

Calculating the peak theoretical MBW and comparing it to the actual memory transfer rate for an algorithm (i.e. a kernel) is helpful to see if memory is limiting performance.

References

[1] K. A. G. Grimsrud, "X-ray diffraction studies of the atomic structure of cadmium sulfide and cadmium selenide nanoparticles." 2007. ii

[2] L. Gelisio, C. L. A. Ricardo, M. Leoni, and P. Scardi, "Real-space calculation of powder diffraction patterns on graphics processing units," *JOURNAL OF APPLIED CRYSTALLOGRAPHY*, vol. 43, pp. 647–653, JUN 2010. iii, 1, 37

[3] Khronos OpenCL Working Group, *The OpenCL Specification*, 1.1 ed., September 2010. iii, 7

[4] P. Debye, "Zerstreuung von röntgenstrahlen," *Annalen der Physik*, vol. 351, pp. 809–823, 1915. 1

[5] A. Cervellino, C. Giannini, and A. Guagliardi, "On the efficient evaluation of fourier patterns for nanoparticles and clusters," *Journal of Computational Chemistry*, 2006. 1

[6] A. Longo and A. Martorana, "Distorted f.c.c. arrangement of gold nanoclusters: a model of spherical particles with microstrains and stacking faults," *Journal of Applied Crystallography*, vol. 41, pp. 446–455, Apr 2008. 1

[7] A. Cervellino, C. Giannini, and A. Guagliardi, "Determination of nanoparticle structure type, size and strain distribution from X-ray data for monatomic f.c.c.-derived non-crystallographic nanoclusters," *Journal of Applied Crystallography*, vol. 36, pp. 1148–1158, Oct 2003. 1

[8] B. D. Hall and R. Monot, "Calculating the debye–scherrer diffraction pattern for large clusters," *Computers in Physics*, vol. 5, no. 4, pp. 414–417, 1991. 1

[9] Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, APRIL 1965. 2

[10] A. Vorokh and A. Rempel, "Atomic structure of cadmium sulfide nanoparticles," *Physics of the Solid State*, vol. 49, pp. 148–153, 2007. 10.1134/S1063783407010246. 2, 20

[11] J. Powell, "The quantum limit to moore's law," *Proceedings of the IEEE*, vol. 96, pp. 1247 –1248, aug. 2008. 2

[12] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, (New York, NY, USA), pp. 73–82, ACM, 2008. 3

[13] A. Klockner, "Pyopencl." http://mathema.tician.de/software/pyopencl, November 2010. 7

[14] T. E. Oliphant, "Python for scientific computing," *Computing in Science and Engineering*, vol. 9, pp. 10–20, 2007. 7

[15] NVIDIA® Corporation, *OpenCL Best Practices Guide*, May 2010. 7, 10, 13

[16] NVIDIA® Corporation, *OpenCL Programming Guide for the CUDA Architecture*, 3.2 ed., August 2010. 7, 10, 35

[17] Advanced Micro Devices®, *Programming Guide, ATI Stream Computing OpenCL*, 1.05 ed., August 2010. 7

[18] E. Kreyszig, *Advanced Engineering Mathematics*. John Wiley & Sons, 9th ed., 2006. 17

[19] J. Als-Nielsen, *Elements of Modern X-Ray Physics*. John Wiley & Sons, 2001. 17, 19, 20

[20] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes. The Art of Scientific Computing.* Cambridge University Press, 3rd ed., 2007. 24

[21] L. Nyland, M. Harris, and J. Prins, *GPU Gems 3, edited by H. Nguyen*. Addison Wesley Professional, 9th ed., 2007. 27

[22] D. Waasmaier and A. Kirfel, "New analytical scattering-factor functions for free atoms and ions," *Acta Crystallographica Section A*, vol. 51, pp. 416–431, May 1995. 29