TDT4501 - COMPUTER SCIENCE,
SPECIALIZATION PROJECT

FALL 2014

Particle-In-Cell Codes in CUDA

Author: Olav Emil Eiksund

Supervisor: Anne Cathrine Elster

January 3, 2015

Abstract

General-purpose computing on graphics processing units, GPGPU, is a recent trend in parallel computing. With hundreds or thousands of threads on a single device, it offers massive parallelism at a relatively low cost.

Nvidia's parallel computing environment CUDA is one of the leading GPGPU technologies. This project has implemented a Particle-In-Cell code using CUDA and the cuFFT library. The PIC method is a technique for solving partial differential equations, and are used to simulate the behavior of charged particles.

The different steps of the simulation algorithm have been timed for different combinations of simuilation grid resolution and particle numerosity. Based on the results from these benchmarks we determine bottlenecks and issues with the implementation, and give suggestions for optimizations.

Contents

List of Figures

Chapter 1

Introduction

This work has examined the application of GPU acceleration of *Particle-In-Cell codes*, a particle simulation scheme. Based on previous work using Pthreads, MPI and OpenMP, this work has used Nvidia's CUDA framework to parallelize the simulation.

The particle-in-cell method consists of modeling charged particles in an electric field, with the solving of Poisson's equation being an important step. With applications such as simulating the behavior of plasma or modeling particle acceleration, there is an interest in performing these simulations with high accuracy, and with a good performance so as to enable longer simulations etc.

Because of the limited resources available for this project, the scope has been more on issues encountered when developing the simulator, rather than optimizing it for industrial or scientific use. Some questions serving as a goal for the project are:

a. Can Particle-In-Cell simulations be implemented in CUDA? Easily?

b. Which issues are there when mapping the problem to CUDA? Which solutions or alternatives are available?

c. Which parts of the simulation turn out to be critical paths? Can these parts of the code be improved further?

1.1 Motivation

1.1.1 Why parallelize PIC codes?

Particle-in-cell codes lends themselves reasonably well to parallelization, and the algorithm itself is quite simple, with few steps. With few dependencies between elements, the sequential part of computation need not be to large, allowing us to parallelize the algorithm without getting dominated by sequential code (see 2.3).

1.1.2 Why parallelize with GPGPU?

GPGPU is a technology that makes use of a large number of slower cores to enable massive computational throughput, and can be very cost effective compared to scaling with CPUs. Since many steps of the PIC algorithm consist of running the same computation on all elements in a large two or three dimensional grid, GPGPU seems to be a great fit.

1.2 Related Work

A brief introduction to previous work that serves as a base for this project.

The overarching goal for the project has been to re-implement the algorithm from my advisor Anne Cathrine Elster's Ph.d. Parallelization Issues and Particle-In-Cell Codes (Cornell, 1994) [1] in CUDA. This work has therefore served as an important theoretic background during development. Elster implemented a parallel PIC simulation with Pthreads that was aimed at the distributed memory computer KSR1. Due to hardware limitations the implementation was in 2D. The work explains the physics behind the simulation, and goes into some detail on how to optimize for performance on a distributed shared memory machine. Elster also demonstrated how the simulation could reproduce physical phenomena such as plasma wave oscillation when sufficient simulation parameters were chosen. While both SOR and FFT solvers 2.2 are described and explained, the implementation made use of the FFT solver.

Jan C. Meyer [2] implemented PIC codes using MPI, as part of a project to compare cluster computer applicability to supercomputer problems. The simulation is based on Elster's previous work, but uses the SOR solver instead, and rewrote code from scratch due to problems in converting the existing work. Issues with the implementation are discussed, and it is suggested that the implementation can be optimized quite a bit.

This was the initial goal of the masters project of Nils M. Larsgård [3], but focus was switched to optimizing Elster's original code. The main topic of the work is the performance of hybrid OpenMP/MPI code, and like Meyer's code uses MPI for communication between nodes. While the FFTW library was used to improve the FFT solver performance, OpenMP was used to accelerate the SOR solver within nodes.

1.3 Thesis Outline

Introduction This section, an introduction to the work, and the motivation behind it.

Background Some background on the physics behind Particle-In-Cell codes, along with some insight into the two solvers used. In addition of the recent history in parallel computing is given, with a special focus on GPGPU and CUDA.

Implementation Details about the implementation. Some known problems and issues are mentioned, as well as what can be done to improve upon the current implementation.

Testing Various aspects of the implementation are benchmarked. Details about the testing methodology, and suggestions for further tests to be done in the future.

Results Performance resuslts from benchmarks.

Discussion Interpretation of results and evaluation of the implementation. Existing flaws and potential for improvement is discussed.

Conclusion A summary of findings, and ideas for future work.

Chapter 2

Background

This section will provide some details on the physics being modeled and the math behind the solvers used. In addition an introduction to modern parallel programming and GPGPU will be given, with special focus on CUDA.

2.1 Particle-In-Cell Codes

Particle simulations attempt to model the behavior and interactions of particles and their forces, with applications in for example fluid dynamics or studies of plasma, they tend to be very compute intensive. One type of simulations is called Particle-in-cell codes.

Particle-in-cell simulations consist of finding the electrical charge distribution resulting from the particle charges, solving the field to find the electrical potential, and move the particles based on the resulting electrical force and their current velocity. The simulation is discrete in both time and space and sufficient resolution in either is necessary to avoid aliasing, and properly model physical phenomena such as plasma wave oscillation.

2.1.1 Subject of simulation

The simulation models the behavior of charged particles in an electric field. In some PIC simulations a magnetic field is present as well, but in this project the magnetic field is assumed to have no effect. Particles are modeled with charge, mass, position and velocity.

2.1.2 Discrete Model Equations

The following equations form the basis of the simulation. The simulation procedure can be derived from these equations, as seen below. For further detail on these equations see chapter 3 of [1]. The only major difference is that this explanation is given with three dimensional space.

$$\nabla^2\Phi = -\frac{\rho}{\epsilon_0} \tag{2.1}$$

$$\mathbf{E} = -\nabla\Phi \tag{2.2}$$

$$\frac{dv}{dt} = \frac{q\mathbf{E}}{m} \tag{2.3}$$

$$\frac{dx}{dt} = v \tag{2.4}$$

2.1.2.1 Poisson's equation

Poisson's equation is a *partial differential equation* (PDE) with uses in various disciplines, and specifically electrostatics. The general form of the equation is $\Delta\phi = f$, which in euclidean space is equal to $\nabla^2\phi = f$. In 3D Cartesian coordinates this becomes $\frac{\partial^2\phi}{\partial^2 x} + \frac{\partial^2\phi}{\partial^2 y} + \frac{\partial^2\phi}{\partial^2 z} = f$.

The form found in equation 2.1 stems from Maxwell's equations, specifically Gauss's law for electricity. In differential form written as

$$\nabla \cdot \mathbf{D} = \rho_f$$

the equation states that the divergence of the electric displacement field $\mathbf{D}$ is equal to the free charge density ρ_f. With the assumption of no polarization or bound charges we have that

$$\mathbf{D} = \epsilon_0 \mathbf{E}$$

Combining these leads us to

$$\nabla \cdot \mathbf{E} = \frac{\rho_f}{\epsilon_0} \tag{2.5}$$

We assume there is no changing magnetic field present, in which the Maxwell-Faraday Equation gives us

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} = 0$$

and in turn equation 2.2:

$$\mathbf{E} = -\nabla\Phi$$

Inserting this into equation 2.5 we get:

$$\nabla \cdot \nabla\Phi = -\frac{\rho_f}{\epsilon_0}$$

or

$$\nabla^2\Phi = -\frac{\rho_f}{\epsilon_0}$$

and in 3D form

$$\frac{\partial^2\Phi}{\partial^2 x} + \frac{\partial^2\Phi}{\partial^2 y} + \frac{\partial^2\Phi}{\partial^2 z} = -\frac{\rho_f}{\epsilon_0}$$

By solving this equation using a 3D PDE solver we can find the electric potential Φ given the distribution of electric charge ρ_f. This form of Poisson's equation is also known as Poisson's equation for electrostatics.

2.1.2.2 Electrical force from Potential

Given the electric potential field Φ we want to determine the electrical force on a particle at position p. Using equation 2.2 we can find the electric field $\mathbf{E}$

$$\mathbf{E} = -\nabla\Phi$$

which is related to the electric force by

$$\mathbf{F}_e = q\mathbf{E}$$

Discretized We can determine the field strength at each grid vertex using first order finite differences differences in each direction:

$$\mathbf{E}_x(i,j,k) = -\frac{\Phi(i+1,j,k)-\Phi(i-1,j,k)}{h_x} = \frac{\Phi(i-1,j,k)-\Phi(i+1,j,k)}{h_x}$$

$$\mathbf{E}_y(i,j,k) = \frac{\Phi(i,j-1,k)-\Phi(i,j+1,k)}{h_y}$$

$$\mathbf{E}_z(i,j,k) = \frac{\Phi(i,j,k-1)-\Phi(i,j,k+1)}{h_z}$$

To find the electrical force on a particle we then need to interpolate the field strength at its position from its cell vertices (see figure 3.1), and divide by the particle's charge.

2.1.2.3 Acceleration and velocity from Electrical force

Given the force exerted on a particle we can use Newton's laws of motion to determine the effect on it's velocity and position. We know that the particle's acceleration is

$$\mathbf{a} = \frac{\mathbf{F}}{m}$$

$$\mathbf{a} = \frac{d\mathbf{v}(t)}{dt}$$

$$\mathbf{v} = \frac{d\mathbf{x}(t)}{dt}$$

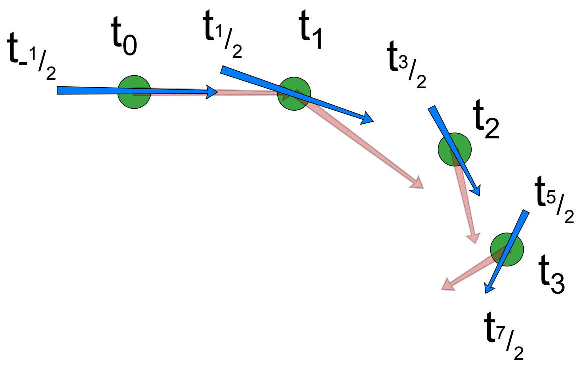
The leap frog method The particle's position and velocity is updated by integrating these equations, and in order to ensure a stable simulation a leap frog integration method is used. By letting the position and velocity updates occur with half a time-step's delay between, they "leap" over each other, resulting in a method more stable than the Euler method for oscillations. The resulting discrete equations are:

$$v_{n+{}^1/_2} = v_{n-{}^1/_2} + \frac{\mathbf{F}(x,y,z)}{m} \cdot \Delta t$$

$$p_{n+1} = p_n + v_{n-{}^1/_2} \cdot \Delta t$$

2.1.2.4 Finding the charge distribution ρ_f

In order to find the charge density on the grid we interpolate a particle's contribution to charge on it's neighboring vertices, similarly to how the electrical field strength is interpolated. A particle's contribution to a vertex is inversely proportional to their distance. The trilinear interpolation used is illustrated in figure 3.1



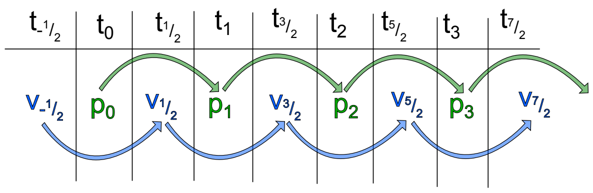


Figure 2.1: An illustration showing how velocity and position are updated in a way causing them to "leap frog" over each other. Shown in transparent red are arrows illustrating v_n.

2.1.3 Procedure

1. Find charge density by accumulating particle charges on the grid.
2. Solve for electrical potential using equation 2.1.
3. Derive electric field strength at all grid vertices from potential, using finite differences (2.2).
4. Update particles:
   (a) Determine electrical field strength at particle position by trilinear interpolation of current cell vertex values.
   (b) Using relation between acceleration, electrical force and electrical field strength, update particle velocity (2.3).
   (c) Update particle position using updated velocity (2.4).

2.2 Solvers

In order to handle different boundary conditions two different solvers have been implemented, one using the *Fast Fourier Transform*, the other based on *Successive Over-Relaxation*. These are the same described by Elster in section 3.3 of [1].

2.2.1 Fast Fourier Transform

The fast Fourier transform is an efficient implementation of a discrete Fourier transform, which calculates a spectral transform of a discrete signal. With extremely broad application, the FFT is known as one of the most important algorithms in computer science today. A brief explanation of the FFT and it's background will be given.

Fourier Transform

From a mathematical viewpoint the Fourier transform is an expression of a function of time as a function of frequency. Related to the Fourier series, where a function can be approximated by a sum of sine and cosine terms, the Fourier transform is similar to an infinite series, and the sum is replaced by an integral:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x)e^{-2\pi i x\xi}dx \tag{2.6}$$

Although equation 2.6 shows a one dimensional transform this generalizes to any number of dimensions.

DFT

A DFT shares most properties with the continuous Fourier transform, apart from being discrete in both input and output. While the continuous transform ia a mathematical operation, the DFT has practical applications in ex. signal processing. Compare equation 2.6 for the Fourier transform with 2.7 for the DFT:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi i k n/N}, k \in \mathbb{Z} \tag{2.7}$$

X is a length N array of complex values, where each element X_k represents one frequency bucket. A one dimensional DFT can be seen as finding the frequency distribution of the input signal, and as such each output element depends on the entire input array. The number of operations required for a full transformation is therefore $O(N^2)$.

Cooley-Tukey FFT algorithm

The Cooley-Tukey FFT is of the most frequently used fast Fourier transforms. The FFT is an improvement over the DFT, dividing a transform into a composite of smaller transforms recursively. The following example will divide by 2 (radix-2 FFT), but any small prime can be used (mixed-radix FFT), letting the algorithm handle any length $n = 2^a \cdot 3^b \cdot 5^c$... The sum in equation 2.7 is split into even and odd sums:

$$\begin{aligned}
X_k &= \sum_{n=0}^{N/2-1} x_{2n} \cdot e^{\frac{-2\pi ik(2n)}{N}} + \sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{\frac{-2\pi ik(2n+1)}{N}} \\
X_k &= \sum_{n=0}^{N/2-1} x_{2n} \cdot e^{\frac{-2\pi ik(2n)}{N}} + e^{\frac{-2\pi ik}{N}} \cdot \sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{\frac{-2\pi ik(2n)}{N}} \\
X_k &= \sum_{n=0}^{N/2-1} x_{2n} \cdot e^{\frac{-2\pi ikn}{N/2}} + e^{\frac{-2\pi ik}{N}} \cdot \sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{\frac{-2\pi ikn}{N/2}}
\end{aligned}$$

We can see that each sum corresponds to the transform of the odd and even parts of the input, with a common twiddle factor $e^{\frac{-2\pi ikn}{N/2}}$. Since the input is assumed to be periodic either sum has the same value for $k = m$ and $k = m + \frac{N}{2}$. Expressing the even sum as C_k and the odd as D_k we get:

$$\begin{aligned}
X_k &= C_k + e^{\frac{-2\pi ik}{N}} \cdot D_k \\
X_{k+\frac{N}{2}} &= C_{k+\frac{N}{2}} + e^{\frac{-2\pi i}{N}(k+\frac{N}{2})} \cdot D_{k+\frac{N}{2}} \\
&= C_k + e^{\frac{-2\pi i}{N}k} \cdot e^{\frac{-2\pi i}{N} \cdot \frac{N}{2}} \cdot D_k \\
&= C_k + e^{\frac{-2\pi i}{N}k} \cdot e^{-\pi i} \cdot D_k \\
&= C_k + e^{\frac{-2\pi i}{N}k} \cdot (-1) \cdot D_k \\
&= C_k - e^{\frac{-2\pi i}{N}k} \cdot D_k
\end{aligned}$$

By reusing the values the number of computations is drastically reduced compared to the plain DFT.

Performance

A main reason that the FFT has seen such widespread use is it's performance, sporting an $O\left(n \cdot log(n)\right)$ complexity, compared to the DFT at $O(n^2)$. As long as the dimensions of the input are multiples of small primes, ideally a power of 2, the FFT is very efficient. For a number of reasons, ex. data alignment, this is often the case.

Properties/Issues

An issue with the Fourier transform is aliasing, which may occur when there are signals in the input with frequencies higher than the resolution of the transform. The result is that the transform interprets the samples from the signal at a lower frequency. See figure 2.2 for an illustration. The *Nyquist theorem* states that to avoid aliasing a sampling frequency at least twice the highest input frequency is needed. In cases where the highest input frequency is unknown this can be hard to ensure, but aliasing will often be clearly visible in the output, for example as procedural noise. Avoiding aliasing is one reason we want to maximize grid resolution in the implementation, since the sampling frequency is the inverse of grid length in each direction, $f_x = n_x^{-1}$.

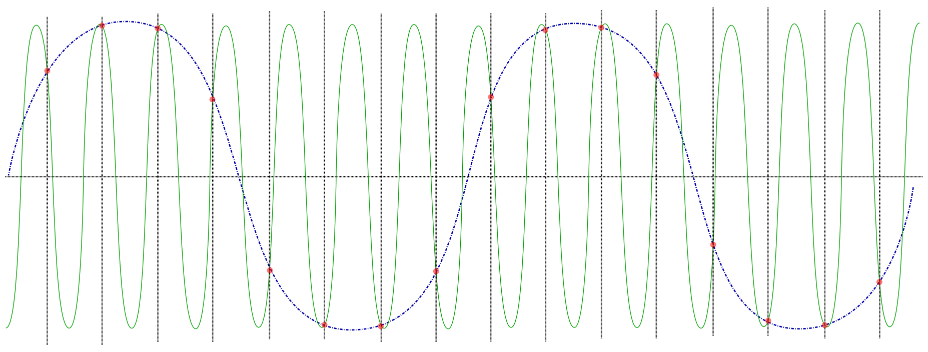


Figure 2.2: An illustration of how a high frequency signal (green) can be misrepresented as a low frequency one (blue) due to insufficient sampling frequency. Samples shown as red circles.

2.2.2 SOR solver

Successive over-relaxation is a scheme for iterative solvers, using a relaxation factor as a weight to ensure either faster convergence. One assumes that the solution "lies further ahead", by weighting the value of neighboring elements higher at the cost of an element's previous value.

System of linear equations

A system of linear equations is the problem of finding n unknowns $\mathbf{x}$, given m linear equations on the form $a_{i1}x_1 + a_{i2}x_2 + a_{i3}x_3 + ... + a_{in}x_n = b_i$. In practice we are interested in solving systems where $n \equiv m$ meaning there exists one unique solution. The simplest such system is $ax = b$, which is trivial in nature. Practical applications can have systems with several thousand variables. Our problem can be represented as a two dimensional system, with a relation between values in either direction.

The Jacobi method

Among the most famous iterative linear solvers for systems of linear equations are the Jacobi and Gauss-Seidel methods. Brief explanations of both follow. Assume a systen of linear equations:

$$A\mathbf{x} = \mathbf{b}$$

A is split into a diagonal and remainder matrix.

$$\begin{aligned} A &= D + R \\ A\mathbf{x} &= (D + R)\mathbf{x} = \mathbf{b} \\ D\mathbf{x} &= \mathbf{b} - R\mathbf{x} \\ \mathbf{x} &= D^{-1}(\mathbf{b} - R\mathbf{x}) \end{aligned}$$

Iteration n updates an element using the equation:

$$\mathbf{x}^{(n+1)} = D^{-1}(\mathbf{b} - R\mathbf{x}^{(n)})$$

Pseudocode using a 5 point stencil on a 2D matrix:

```
while (err > threshold):
  for all i, j:
    updated[i, j] =
        (old[i-1, j] + old[i+1, j] + old[i, j-1] + old[i, j+1])/4

    err_ij = abs(updated[i,j] - old[i,j])
    if (err_ij > err):
      err = err_ij
```

Gauss-Seidel

Because a value depends on all neighboring old values, updates have to be written to a new matrix, resulting in a $2n$ storage requirement. The Gauss-Seidel method improves upon this by replacing reads to old values with updated ones, thus allowing in-place updates. In addition this is shown to converge faster. Using the same matrix A,

$$A = L + D + U$$

A is divided into lower triangle, diagonal, and upper triangle matrices.

$$\begin{aligned} A\mathbf{x} = (L + D + U)\mathbf{x} &= \mathbf{b} \\ (L + D)\mathbf{x} &= \mathbf{b} - U\mathbf{x} \\ \mathbf{x} &= (L + D)^{-1}(\mathbf{b} - U\mathbf{x}) \end{aligned}$$

And as for Jacobi an element is updated as

$$\mathbf{x}^{(n+1)} = (L + D)^{-1}(\mathbf{b} - U\mathbf{x}^{(n)})$$

and pseudo-code for the same problem:

```
while (err > threshold):
  for all i, j:
    updt[i, j] =
        (updt[i-1, j] + old[i+1, j] + updt[i, j-1] + old[i, j+1] )/4

    err_ij = abs(updated[i,j] - old[i,j])
    if (err_ij > err):
      err = err_ij
```

This way no subsequent calculations use *old*$[i, j]$ and updated *updt* values can be written to the *old* matrix. Gauss-Seidel is therefore a more storage efficient method, important for large problems, and memory bound systems.

Over-relaxation

A variant of Gauss-Seidel, successive over-relaxation has a minor modification which tends to result in faster convergence. By using a weight ω (the *relaxation factor*), the old value is used to either speed up convergence, or prevent divergence:

$$\begin{aligned} A &= L + D + U \\ (L + D + U)\mathbf{x} &= \mathbf{b} \\ \omega(L + D + U)\mathbf{x} &= \omega\mathbf{b} \\ \big[\omega L + (\omega + 1 - 1)D\big]\mathbf{x} &= \omega\mathbf{b} - \omega U\mathbf{x} \\ (\omega L + D)\mathbf{x} &= \omega\mathbf{b} - \big[\omega U + (\omega - 1)D\big]\mathbf{x} \\ \mathbf{x} &= (\omega L + D)^{-1}(\omega\mathbf{b} - \big[\omega U + (\omega - 1)D\big]\mathbf{x}) \\ \mathbf{x}^{(n+1)} &= (\omega L + D)^{-1}(\omega\mathbf{b} - \big[\omega U + (\omega - 1)D\big]\mathbf{x}^{(n)}) \end{aligned}$$

```
while (err > threshold):
  for all i, j:
    temp = (updt[i-1, j] + old[i+1, j] + updt[i, j-1] + old[i, j+1] )/4
    updt[i, j] = (omega-1) * old[i, j] + omega * temp

    err_ij = abs(updated[i,j] - old[i,j])
    if (err_ij > err):
      err = err_ij
```

The value of ω is used to control the influence the old value has on the new. $\omega = 1$ is the same as Gauss-Seidel, $\omega = {}^1/_2$ is to take the average of the old and updated values, while $\omega = 2$ would result in $new = 2 \cdot updated - old$. For $\omega < 1$ change between iterations is diminished, and gives a lower chance of divergence. $\omega > 1$ places more focus on neighboring values, and may speed up convergence. We want to speed up convergence, and good choice for ω has been shown to lie around 1.78.

Parallelized - Red-Black coloring

A problem with the Gauss-Seidel method is the dependency between an update and those preceding it. Because *updated*$[i]$ depends on *updated*$[i-1]$, these cannot be calculated in parallel. The Jacobi method however has no dependencies to elements from the same iteration, but cannot be done in-place. We can get the best of both worlds however, using a Red-Black Jacobi method. By updating every other element in parallel, we avoid read-write conflicts, and no dependencies between elements being updates, see figure 2.3.

2.3 Parallel Computing

Parallel computing may refer to any form of processing in which multiple operations are performed concurrently. This can range from vectorizing instructions to running ten thousands of threads on thousands of nodes on a supercomputer. A short introduction to the history and motivation behind parallel computing will be given, as well as a measure of performance for parallel computing.

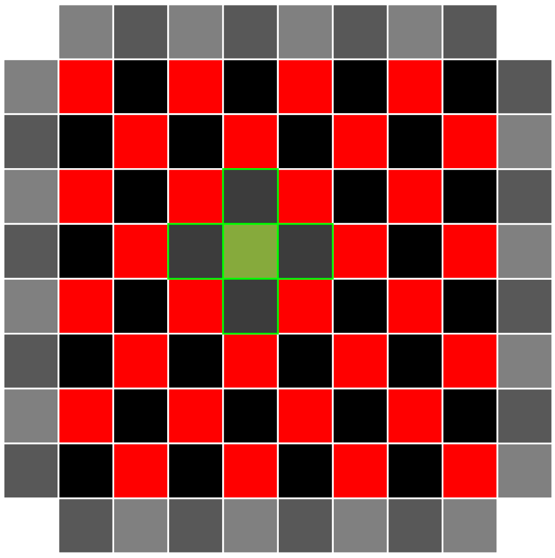


Figure 2.3: An example of 2D red-black coloring of a matrix. By first updating black, and then the red elements, calculations can be done in parallel, and in-place. As seen with the green stencil, the center element is only dependent on the its neighboring black elements, not on red ones.

Moore's law, limits of single core processing, and the rise of multi-core
Well known to every computer scientist, Moore's law described the consistent increase in processing power since the 1970s. Performance increased along with transistor density, and was observed to double approximately every two years. As long as performance could be increased by simply building a faster processor, interest in multi-core computing was limited. Three factors became readily visible during the early 2000s:

- The Memory Wall Processor speeds increased faster than memory speeds, making it harder to hide memory latency. The result was that further increases in CPU clock speed would be lost waiting for memory.
- The ILP Wall While instruction level parallelism (ILP) had been used to hide the growing disparity between CPU and memory clock speed, it became harder to extract further instruction level parallelism from a single stream of instructions.
- The Power Wall

  $$U_{CPUvoltage} = O(f_{CPUfrequency})$$

  $$P_{CPUpowerconsumption} = O(V^2_{CPUvoltage})$$

  Because of these relationships, and the fact that CPU clock frequency has been doubling yearly, and number of transistors steadily increasing, CPUs were consuming a lot of power, and becoming very hot dissipating this energy. While the increase in voltage has been mitigated somewhat by the shrinking die size, the small transistors and low voltage causes issues like subthreshold leaking and electromigration to become more prominent.

The result is that the cost of increasing performance while avoiding the power wall is too great, considering that memory latency is the bottleneck.

Because of these issues the industry has shifted to multi-core computing as a way of further increasing performance. By creating chips consisting of several smaller simpler processors rather than one large complex one, power consumption and manufacturing costs is decreased, at the cost of being harder to program.

Another solution has been to offload work to hardware accelerators, such as graphics cards. This method, known as GPGPU will be explained in more detail below.

Amdahl's law, Gustafson's law With the advent of parallel computing a new measure of performance is needed. while single core processing has a maximum performance equal to $IPS = f_{clock} \cdot IPC$, the number of instructions per second is equal to the number of instructions per clock cycle times the clock frequency. Assuming a fully parallelizable problem, the performance should scale with the number of course, $IPS = n_{cores} \cdot f_{clock} \cdot IPC$. In reality however, not all parts of a problem are easily parallelizable, often some setup or preprocessing has to be done sequentially. Because of this we need a measure dependent upon the number of processors utilized, and the portion or the problem that is parallelizable. In particular we want a measure of the speedup using n processors instead of one, based on the formula:

$$S_n = \frac{T_1^{execution\,time}}{T_n^{execution\,time}}$$

Amdahl's law gives an intuitive formula for the speedup of a program running on n processors where $P_{sequential}$ is the sequential fraction of the execution time:

$$T_n = T_1 \cdot (P_{sequential} + \frac{P_{parallel}}{n})$$

$$S_n = \frac{T_1}{T_n} = \frac{T_1}{T_1 \cdot (P_{sequential} + \frac{P_{parallel}}{n})} = \frac{1}{P_{sequential} + \frac{P_{parallel}}{n}}$$

This formula is useful in finding the potential speedup when parallelizing a sequential program, but as is evident, the sequential part will quickly dominate the parallel part. By setting $n = \infty$ the formula becomes $S_\infty = \frac{1}{P_{sequential}}$, and for a program with a 20% sequential part the maximum speedup is 5. With $n = 8$ this same problem would get a speedup of $S_8 = \frac{1}{0.2+\frac{0.8}{8}} = 1/0.3 = 3.33$, and with 16 cores the speedup would be $S_16 = 4$. Clearly this does not scale well, and GPU computing with thousands of cores makes little sense.

The answer given in *Gustafson's law* is that the approach to parallelizing the problem is wrong, and instead defines the speedup as

$$S_n = P_{sequential} + n \cdot P_{parallel} = n - P_{sequential} \cdot (n - 1)$$

The elementary difference here is an assumption that the size of the parallel part of the computation scales linearly with the number of processors. In other words, by increasing the problem size alojng with the number of processors, we can keep improving the speedup without getting dominated by the serial part.

One could also think of this as writing a parallel program instead of converting a serial one, allowing the workload to be optimized for parallel computing. This is also the clue GPGPU's potential.

2.4 GPGPU

General-purpose computing on graphics processing units (GPGPU) is a fairly recent trend in parallel computing, where programs are accelerated by running heavy computation on a graphics processing unit. Characterized by a large number of slower, cheaper cores, graphics cards today deliver massive parallelism to a standard desktop computer at low cost.

2.4.1 History

Traditional graphics pipelines allowed only data to pass from the CPU to the GPU, where it would be processed in several steps, rendered, and output to some display device. Around 2001 programmable shaders enabled researchers to experiment with GPU computing, by defining data in terms of graphics primitives. In order to alleviate some of the trouble of programming this way, GPU-specific languages and libraries were made, hiding the shader language from the programming. One such language was Sh [4].

Coinciding with the industry's shift towards multicore/parallel programming, GPUs became increasingly powerful and easier to develop for over the next few years. From 2006 GPUs shifted from graphics-specific pipelined devices towards generic stream processing, with increasingly powerful parallelism as well as general programmability. Today GPUs are contending with CPUs, offering a potentially much higher performance at the cost of some generality. For parallel applications, GPUs offer a significantly cheaper option than the CPUs required to achieve the same performance, both in terms of value and power budget. For this reason many current supercomputers are including GPUs to increase processing power per node, with three among the top ten having Nvidia Tesla GPUs installed[5]. Of the 500 75 use accelerator technology, with 53 using GPUs.

Take for instance the Titan supercomputer by Cray at Oak Ridge National Laboratory, which took the lead on the TOP500 list in November 2012, and uses Nvidia Tesla. Compared to their previous supercomputer Jaguar, performance increased from $2.3PF$ to $27PF$ while power increased from $7MW$ to $9MW$.[6] This means more than a tenfold increase in performance at the cost of only $1.3\times$ the power. Already ORN labs and Nvidia are cooperating to see similar scaling for their next supercomputer Summit, aiming for a 2017 installation[6].

2.4.2 Performance

The *Nvidia GeForce 8800 GTX* is a high end GPU from 2007 with a theoretical peak performance of 518.4 GFLOPS [1] and a power consumption of $155W$. A contemporary high end CPU, the *Intel® Core^{TM}2 Extreme Processor QX9650* had a peak performance of $96GFLOPS$ [2] with a power consumption of $130W$.

[1] http://www.techpowerup.com/gpudb/187/geforce-8800-gtx.html

[2] Assuming SSE: $4cores \cdot 8\frac{FLOPS}{cycle} \cdot 3.0GHz = 96GFLOPS$.

See http://ark.intel.com/products/33921/Intel-Core2-Extreme-Processor-QX9650-12M-Cache-3_00-GHz-1333-MHz-FSB.

A high end consumer GPU in 2014, the *GTX 980*, has a theoretical peak performance of $4612 GFLOPS$ [7], drawing $165W$ [3] and costs around 5000 NOK [4]. In comparison Intel's current high end CPU, the *Intel Core i7-5960X Extreme Edition*, has a theoretical peak performance of $768 GFLOPS$ [5], a power consumption of $140W$ [6], and a cost of ca 9500 NOK [7].

From these numbers it looks as if the GPU provides than five times the performance at half the cost, with only a slightly higher power consumption. In reality the performance of either will be lower, and for double precision operations the CPU performance will decrease a lot less than will the GPU's, but it is apparent that the GPU can provide relatively cheap performance, and that the performance continues to improve. Note that the above are only estimates based on specifications and not measured results, and that prices may vary between vendors and stores.

2.5 CUDA

The *Compute Unified Device Architecture*, or CUDA for short, is a GPGPU computing platform. Developed by Nvidia and introduced in 2006, CUDA is a relatively young technology, but has already seen employment in a number of applications and research papers.

To develop for the CUDA platform one can use CUDA libraries, languages including the C/C++ and Fortan extensions and the OpenCL framework, and compiler directives such as OpenACC/OpenMP [8]. In addition there exists third party wrappers for many other languages.

For this project the *CUDA C/C++* is used, and will be reflected in the brief introduction below. An introduction to what CUDA is by Nvidia can be found at http://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/.

2.5.1 Programming CUDA

The CUDA C/C++ has a few special features that sets it apart from standard C/C++. Most importantly, there is a distinction between *host* and *device* code. Host code runs on the CPU and is responsible for calling device code. Device code runs on the GPU, and can be recognized as functions with the `__global__` qualifier. These functions are known as kernels, and several instances are executed in parallel on the CUDA cores of the GPU.

Code is compiled using Nvidia's LLVM[8]-based compiler *NVCC*. NVCC in turn uses either the GNU compiler *gcc* or the Microsoft Visual Studio compiler *cl* to compile host code, while device code is compiled to the *ptx* assembler

[3] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980/specifications

[4] https://www.komplett.no/search?q=GTX980

[5] Assuming AVX and FMA: $8 cores \cdot 32 \frac{FLOPS}{cycle} \cdot 3.0 GHz = 192 GFLOPS$.
See also http://www.pugetsystems.com/blog/2014/08/29/Linpack-performance-Haswell-E-Core-i7-5960X-and-5930K-594/ for a lower estimate.

[6] http://ark.intel.com/products/82930/Intel-Core-i7-5960X-Processor-Extreme-Edition-20M-Cache-up-to-3_50-GHz

[7] https://www.komplett.no/intel-core-i7-5960x-extreme-edition/822376

[8] See http://llvm.org/

language [9]. When the program is executed on a device the Nvidia graphics driver compiles the ptx code into a device specific binary [10].

Below is an example of a CUDA kernel, and a call to this kernel:

```
__global__ kernel(float a, float* x, float *y, float *out, int max)
{
  // Find the index of the current thread:
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  // Stay within bounds of the arrays:
  if(idx >= max)
    return;
  // Perform computation:
  out[idx] = a * x[idx] + y[idx];
}

// Host code
int count = 20;
size_t size = count * sizeof(float)
float f = 0.25f;

float
  *h_x, *h_y, *h_out;
  *d_x, *d_y, *d_out;
cudaMalloc(&d_x, size);
cudaMalloc(&d_y, size);
cudaMalloc(&d_out, size);

// Copy input arrays to device:
cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice)
cudaMemcpy(d_y, h_y, size, cudaMemcpyHostToDevice)

kernel<<<1, 32>>>(a, d_x, d_y, d_out, count);

// Copy result back to host:
cudaMemcpy(h_out, d_out, size, cudaMemcpyDeviceToHost)
```

This is a typical CUDA kernel call, although stripped of error checks and the like. We see that the host has to allocate memory for device arrays, copy data to the device, launch the kernel and copy the result back.

In launching the kernel we see a deviation from standard C/C++ function calls, the <<<1, N>>> part. These are special parameters required by all kernels, specifying how the threads are structured, using the syntax<<<blocksInGrid, threadsInBlock>>>. Each of these accept a dim3 argument that specifies the length of blocks and grids in three dimensions. Above only one value is given, specifying the x dimension while y and z default to 1.

In the kernel body we find a thread's rank within this structure though threadIdx and blockIdx, while gridDim and blockDim correspond to the values given in <<<blocksInGrid, threadsInBlock>>>. Threads are launched in *warps*, groups of threads that execute concurrently on a core, and the number of threads we launch should always be a multiple of the warp size, which is usually 32. Because of this some threads we launch may not have a designated element to work on, and to avoid reading and writing out of bounds we check the thread index against the array size.

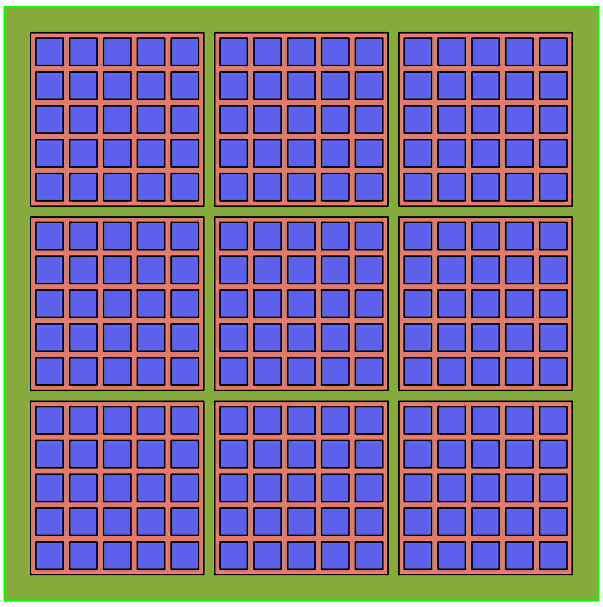


Figure 2.4: The thread hierarchy in CUDA is illustrated above, with a grid containing several blocks, which themselves contain several threads. This configuration would be calles as <<<(3, 3, 1), (5, 5, 1)>>>.

2.5.2 Memory

GPU-accelerated programs tend to be very sensitive to memory accesses, and so it is very important to be conscious of data locality when looking to optimize code. In CUDA there are three main levels of memory, global device memory, shared block memory, and thread local memory. Using figure 2.4 as an example, all threads (blue) have their own local memory, registers, and all threads in a block (red) have access to a shared memory, while all blocks may access global memory. The difference in memory latency between levels are in order of magnitude, so it is desirable to keep data as close to the thread level as possible. Local and shared memory lifetime is restricted to a kernel launch, so to keep results they have to be written to global memory.

In addition to these there is a constant memory which is read-only from device code, and texture memory which is optimized for stencil access.

2.5.3 cuFFT

cuFFT is Nvidia's FFT framework for CUDA. It consists of the cuFFT and cuFFTW libraries, with the former designed with GPUs in mind, and the latter serving as a porting tool for code using the well known FFTW library[9]. The cuFFT library is the one used in the implementation, and no further detail will be given on cuFFTW.

The library claims excellent performance for input sizes on the$n = 2^a \cdot 3^b \cdot 5^c \cdot 7^d$, and for prime factors up to 127 it uses the efficient Cooley-Tukey algorithm. For other sizes it falls back on the less accurate Bluestein's algorithm[13,

[9]http://www.fftw.org/

sec. 2.12] which handles large prime sizes better. Transforms of up to three dimensions are supported, with a maximum of 512 million elements for single precision and 256 million elements in double precision transforms, although limited by device memory and transform type (complex or real input etc.)[13, sec. 1].

Chapter 3

Implementation

Because the work this project is based on used double precision data types this is done here as well, to make sure errors are not due to insufficient accuracy. A switch to single precision data types was left as a question to be evaluated based on results from testing.

In contrast to the previous work however, this simulation is done in 3D, adding realism at the cost of accuracy. Adding an additional dimension increases the number of cells by resolution in that dimension times the number of cells in the plane. Because the memory usage and algorithm complexity increases, the resolution of each dimension will be limited. A 3D grid of 256^3 contains the same number of cells as a 2D grid at 4096^2.

3.1 Kernels

A description of the kernels follow.

3.1.1 determineChargesFromPotential

The kernel determining the charge density of the mesh from particle charges. The contribution of a particle to each vertex of its resident cell is found using trilinear interpolation, illustrated in fig. 3.1. The distribution of a particle

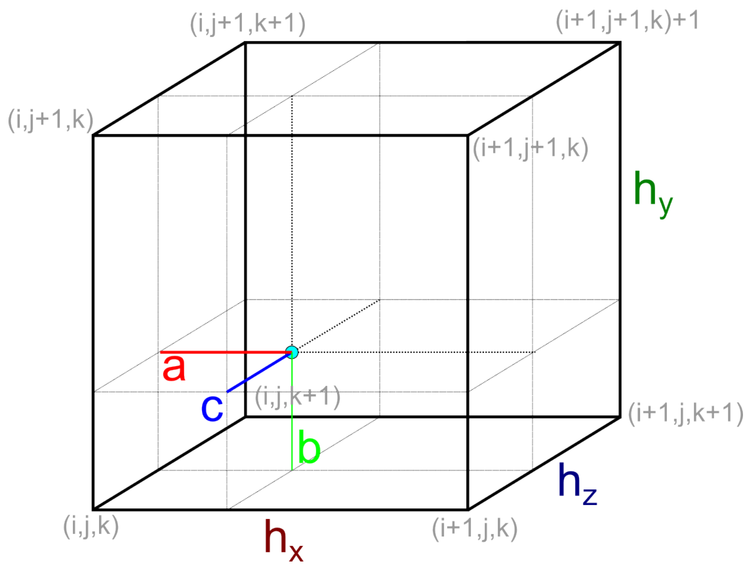


Figure 3.1: Trilinear interpolation of particles with mesh vertices.

charge ρ_p is as follows:

$$\begin{aligned}
\rho(i,j,k) \quad &+= \quad \frac{\rho_p}{(h_x \cdot h_y \cdot h_z)} \cdot (h_x - a) \cdot (h_y - b) \cdot (h_z - c) \\
\rho(i+1,j,k) \quad &+= \quad \frac{\rho_p}{(h_x \cdot h_y \cdot h_z)} \cdot (h_x - a) \cdot b \cdot (h_z - c) \\
\rho(i,j+1,k) \quad &+= \quad \frac{\rho_p}{(h_x \cdot h_y \cdot h_z)} \cdot a \cdot (h_y - b) \cdot (h_z - c) \\
\rho(i+1,j+1,k) \quad &+= \quad \frac{\rho_p}{(h_x \cdot h_y \cdot h_z)} \cdot a \cdot b \cdot (h_z - c) \\
\rho(i,j,k+1) \quad &+= \quad \frac{\rho_p}{(h_x \cdot h_y \cdot h_z)} \cdot (h_x - a) \cdot (h_y - b) \cdot c \\
\rho(i+1,j,k+1) \quad &+= \quad \frac{\rho_p}{(h_x \cdot h_y \cdot h_z)} \cdot (h_x - a) \cdot b \cdot c \\
\rho(i,j+1,k+1) \quad &+= \quad \frac{\rho_p}{(h_x \cdot h_y \cdot h_z)} \cdot a \cdot (h_y - b) \cdot c \\
\rho(i+1,j+1,k+1) \quad &+= \quad \frac{\rho_p}{(h_x \cdot h_y \cdot h_z)} \cdot a \cdot b \cdot c
\end{aligned}$$

As can be seen, the contribution for a vertex is proportional to its distance to the particle compared to the other vertices, the closest one receives the most charge and all contributions sums to ρ_p. Visualized in figure 3.1, dividing the cells along the interpolation lines, the contribution of a vertex is proportional to the volume of the opposite sub-prism. We can see that the variable terms of the equations above give the volume, for instance $\rho(i+1, j+1, k+1)$ has the term $a \cdot b \cdot c$ which is the volume of the prism in the opposite corner.

The kernel as is does not use shared memory to reduce memory latency, and it relies on slow atomic operations to avoid write conflicts.

3.1.2 electricFieldFromPotential

The electric field strength at a point is approximated as the difference in potential between it's neighbors, measured separately in each dimension.

$$E_x = \Phi_{i+1} - \Phi_{i-1}$$
$$E_y = \Phi_{j+1} - \Phi_{j-1}$$
$$E_z = \Phi_{k+1} - \Phi_{k-1}$$

The electric field is therefore stored as a vector at each point in the grid (array of structures).

electricFieldAtPoint is a device helper function for doing trilinear interpolation of the electric field strength at some floating point position. Similarly to the charge density calculations, field strength at a position is accumulated from cell vertices, with contribution from each one proportional to the distance to it.

3.1.3 updateParticles

```
ax = p.electricfield.x * cfg.charge_by_mass;
prev = (1 - cfg.drag * cfg.ts);
p.velocity.x = p.velocity.x * prev + ax * cfg.ts;
p.position.x += p.velocity.x * cfg.ts;
```

The particle update kernel first finds the field electrical field strength at the particle's position using the helper function electricFieldAtPoint, and then the acceleration using $a_p = \frac{F_e}{m_p} = \frac{E \cdot q_p}{m_p}$. Velocity and position is updated through a leap frog method, where there is a ${}^{\Delta t}/_2$ delay between velocity and position updates.

$$v^{n+1/2} = v^{n+1/2} + a^n \cdot \Delta t$$

$$p^{n+1} = p^n + v^n \cdot \Delta t$$

The result is that a position update uses a velocity value that lies between the two points in time, an "average" value, and similarly for velocity updates.

3.2 cuFFT

The FFT-based solver uses the cuFFT library, and this section will therefore focus on usage rather than implementation.

3.2.1 FFT setup

To run an FFT a plan has to be set up first, and stored using a cufftHandle. For single transforms with simple data layouts the cufftPlan#d() functions provide a simple interface for 1D, 2D and 3D transforms, while more complex setups may need to use cufftPlanMany(). This function allows input and output to be batched and strided data. All plans specify the data types of the transform with options of *real-to-complex*, *complex-to-real*(implicitly an inverse transform), or *complex-to-complex*, and single or double precision. R stands for real, C for complex, while D and Z are the same only for double precision.

```
//cufftPlanMany signature
cufftResult cufftPlanMany(
  cufftHandle *plan,// Pointer to the plan
  int rank,          // dimensionality, (1, 2 or 3)
  int *n,            // n[i] = size of dimension i
  int *inembed,      // Size of dimensions in storage
  int istride,       // Distance between elements in inner dimension
  int idist,         // Distance between first elements in a batch
  int *onembed,      // Same as the above but for output
  int ostride,       // ...
  int odist,         // ...
  cufftType type,    // real or complex, single or double precision
  int batch          // multiple transforms with one call
);
```

3.2.2 FFT call

After the plan has been created, the transform can be executed using cufftExecX2X(). X2X may be either R2C, C2R, C2C, D2Z, Z2D, Z2Z and must match the type parameter of the plan. In addition to the plan generated using cufftPlanXX(), pointers to input and output data must ge supplied. For C2C and Z2Z an additional direction parameter must be set. R2C is implicitly forward, Z2D is implicitly inverse and so on.

```
//cufftExecC2C signature
cufftResult cufftExecC2C(
  cufftHandle plan,      // Plan generated as above
  cufftComplex *idata,   // Input data pointer
  cufftComplex *odata,   // output data pointer
  int direction          // Direction (forward/inverse)
);
```

```
// Example cufft procedure
cufftHandle plan, iplan;
cufftCreate(&plan);
cufftCreate(&iplan);
cufftPlanMany(plan, ...); // Paln the transforms
cufftPlanMany(iplan, ...);
cufftExecR2C(plan, ...); // Forward transform
// ...
// Do something in spectral domain.
// ...
cufftExecC2R(iplan, ...); // Inverse transform
```

solve After the data has been transformed using cufft, solving the for the field is done by multiplying each value by $^1/_{k^2}$, and then scaling by $^1/_{\epsilon_0}$ to get Φ.

$$\frac{1}{\epsilon_0 \cdot \left(\left(\frac{(2\pi \cdot \mathrm{i}}{L_x}\right)^2 + \left(\frac{2\pi \cdot \mathrm{j}}{L_y}\right)^2 + \left(\frac{2\pi \cdot \mathrm{k}}{L_z}\right)^2\right)}$$

In addition, we need to normalize the transformation, scaling the elements by the size of the data set, which is $N_x \cdot N_y \cdot N_z$. The resulting computation is then to multiply the value at (i, j, k) by

$$\frac{1}{4\pi^2 \cdot \epsilon_0 \cdot N_x \cdot N_y \cdot N_z \cdot \left({}^{\mathrm{i}^2}/_{L_x^2} + {}^{\mathrm{j}^2}/_{L_y^2} + {}^{\mathrm{k}^2}/_{L_z^2} \right)}$$

```
scale_factor = 1/(eps_0 * 4*pi*pi * n.x * n.y * n.z);

double scale = scale_factor /
      (i*i/(l.x*l.x) + j*j/(l.y*l.y) + k*k/(l.z*l.z));

//Complex number:
row[i].x *= scale;
row[i].y *= scale;
```

3.3 SOR

From Gauss's law we have

$$\nabla^2\Phi = \frac{\partial^2\Phi}{\partial x^2} + \frac{\partial^2\Phi}{\partial y^2} + \frac{\partial^2\Phi}{\partial z^2} = -\frac{\rho_f}{\epsilon_0}$$

This is approximated using second order finite differences.

$$\nabla^2\Phi \approx \frac{\Phi_{i-1} + \Phi_{i+1} + \Phi_{j-1} + \Phi_{j+1} + \Phi_{k-1} + \Phi_{k+1} - 6\Phi_{i,j,k}}{h_x \cdot h_y \cdot h_z} = -\frac{\rho_{i,j,k}}{\epsilon_0}$$

We solve for $\Phi_{i,j,k}$

$$\Phi_{i,j,k} = \frac{\rho_{i,j,k} \cdot \frac{h_x h_y h_z}{\epsilon_0} + \Phi_{i-1} + \Phi_{i+1} + \Phi_{j-1} + \Phi_{j+1} + \Phi_{k-1} + \Phi_{k+1}}{6}$$

Over-relaxed updates use

$$\begin{aligned} \Phi_{i,j,k}^{(n+1)} &= (\omega - 1)\Phi_{i,j,k}^{(n)} + \omega \cdot \frac{\Phi_{i-1}^{(n)} + \Phi_{i+1}^{(n)} + \Phi_{j-1}^{(n)} + \Phi_{j+1}^{(n)} + \Phi_{k-1}^{(n)} + \Phi_{k+1}^{(n)}}{6} \\ \Phi_{i,j,k}^{(n+1)} &= \Phi_{i,j,k}^{(n)} + \\ & \omega \cdot \left(\frac{\Phi_{i-1}^{(n)} + \Phi_{i+1}^{(n)} + \Phi_{j-1^{(n)}} + \Phi_{j+1^{(n)}} + \Phi_{k-1}^{(n)} + \Phi_{k+1}^{(n)}}{6} - \Phi_{i,j,k}^{(n)} \right) \end{aligned} \tag{3.1}$$

The SOR solver builds on the red-black Jacobi method described in 2.2.2, using over relaxation to speed up convergence. While previous work[1, sec. 3.3.3][3, sec. 2.7.2] used five-point stencils in a 2D solver, a seven-point stencil is used to account for the z dimension, this being a 3D solver. For this solver boundary values are set equal to the center value ($\Phi_{i,j,k}$).

As is, the kernel is run a fixed number of times, rather than checking the error. Every iteration the kernel is run twice, once each for red and black colored tiles, allowing in-place updates in parallel.

SOR kernels An initSOR kernel is run first, saturating the *Phi* array with the appropriate values,

$$\Phi_{i,j,k} = \frac{\rho_{i,j,k} \cdot h_x h_y h_z}{6 \cdot \epsilon_0}$$

When the SOR kernel itself is called, the k index is calculated as follows, in order to implement red-black coloring:

$$k = 2 \cdot Idx.z + (i + j + flag)\%2$$

where flag is 0 or 1 depending on whether red or black tiles are updated. The update itself uses equation 3.1:

```
tmp = (left + right + down + up + front + back)/6;
Phi[i][j][k] = center + cfg.omega * (tmp - center);
```

3.4 Setup

The implementation currently uses parameters set in a getConfig() function, rather than with preprocessor macros. While this prevents the compiler from optimizing certain calculations it allows parameters to be read in at runtime, functionality that existed in Elster's original implementation, which this one it intended to be a CUDA version of. Parameters that can be set and their default values for testing are shown in section 4.3.

All device memory allocations except that for the particle array use pitched pointers to ensure data alignment, helping ensure coalesced memory accesses when possible. While the arrays probably could be reused to save space, such an optimization is left for further improvements. For the current version the focus has been on ease of debugging and understandability, for which separate arrays are a better fit.

Kernel grid and block settings (specified as kernel<<<grid, block>>>(...)) are chosen to achieve 256 threads per block, with a (256, 1, 1) configuration for particle-indexing kernels, and (16, 4, 4) for field-indexing kernels. Since the particle array is one dimensional its setting is trivial, but the choice for the three dimensional ones need some consideration. For the sake of coalesced memory access one would want threads to access values that are sequential in the x-dimension, so it makes sense to have a wider x dimension. But if shared memory is used to speed up computation and we want to do a border exchange, a square grid offers the fewest $^{neighbor}/_{element}$ ratio, reducing the number of transfers relative to the number of in-block computations. In the implementation a value of 16 is chosen to ensure 128-byte alignment ($16 \cdot sizeof(double) = 128$) while maximizing the block volume.

3.5 Particle tracing

Particle tracing is here implemented by copying the particle array from device to host every iteration. The host has a $(N_{iterations}+1) \cdot N_{particles} \cdot sizeof(Particle)$ array, where each particle's data is stored for each iteration. After the simulation loop has executed this array is used to output an xml file structured as follows:

```
<root n_iterations="..." n_particles="..." particle_interval="...">
  <iteration time="...">
    <particle id="...">
      <position x="..." y="..." z="..."/>
      <velocity x="..." y="..." z="..."/>
      <electric x="..." y="..." z="..."/>
    <particle>
  </iteration>
</root>
```

This file is read by a python script using matplotlib to animate the movement of the particles. As of yet this script is sensitive to the data volume, and works best with a reduced number of updates. Because of this, and in order to reduce the performance hit associated with transferring the particle array every iteration, an interval between particle transfers is used, *particle_interval*. The exact interval used depends on the resolution needed to properly trace the particles' movements, which is a function of the testing parameters and typical particle movement speed.

Another option to reduce memory transfer latency is to instead store the array on the device, and then transfer the whole array after execution. For small values of $N_{particles}$, where the cost of initiating a transfer (API call etc.) is significant compared to the actual data transfer, this would likely lead to some speedup since

$$T_{init} + T_{trans}(N_{trans} \cdot N_{particles}) < N_{trans} \cdot (T_{init} + T_{trans}(N_{particles}))$$

. For large values of $N_{particles}$ however this would be less pronounced, since $T_{init} << T_{trans}(N_{particles})$. In addition, keeping a large array stored on the device consumes memory otherwise available to increase the problem size, thus limiting the grid resolution n_x, n_y, n_z and $N_{particles}$, based on the number of iterations. To get the best of both worlds on could store a certain number of iterations' worth of data on the device, and then transfer them, making room for further updates on the device. By selecting a transfer frequency so that

Chapter 4

Testing

4.1 Testing methodology and motivation

In testing the implementation the goal is to see how performance is affected by various factors. This includes testing for different problem configurations to see which parameters have the biggest impact, and testing runtime for the different kernels to determine which ones among them are the most demanding. The parameters that are tested for are *number of iterations* $N_{iterations}$, *grid resolution in all dimensions*n_x, n_yandn_z, and *number of particles* $N_{particles}$.

4.1.1 Procedure

Kernel timing is done using the following procedure based on the example by Mark Harris from Nvidia on devblogs.nvidia.com/....

```
cudaEvent_t beginning, end;
cudaEventCreate(&beginning);
cudaEventCreate(&end);

cudaMemcpy(...);

cudaEventRecord(beginning);
// Code that should be measured goes in here.
kernel<<<...>>>(...);
cudaEventRecord(end);

cudaMemcpy(...);

cudaEventSynchronize(end);
float timing = 0.0f;
cudaEventElapsedTime(&milliseconds, beginning, end);
```

The code above works in the following way: cudaEventRecord(beginning) records the time of the next event recorded, which would be the kernel call. The next cudaEventRecord(end) records when the next cuda event occurs, which would be the cudaMemcpy call. cudaEventSynchronize(end) blocks CPU execution until the event has been recorded, ensuring a correct measurement. By placing the code to be measured between cudaEventRecord(beginning) and cudaEventRecord(end) we can find $\Delta t = end - beginning$.

Other tests are timed using QueryPerformanceCounter functions provided by Windows:

```
_int64 t1;

QueryPerformanceCounter( (LARGE_INTEGER*)&t1 );
//
//code to be timed goes here
//
_int64  t2, ldFreq;
QueryPerformanceCounter( (LARGE_INTEGER*)&t2 );

QueryPerformanceFrequency( (LARGE_INTEGER*)&ldFreq );
double result = ((double)(t2 - t1) / (double)ldFreq) * 1000.0;

//or

_int64 t1;

StartTimer(&t1);
//
//code to be timed goes here
//
double result = StopTimer(t1);
```

Helper functions called StartTimer and StopTimer are used for simplicity, and to avoid code duplication.

4.2 System configuration

The test system configuration is given in the figures below.

4.2.1 Hardware

The relevant hardware of the test system is given in table 4.1.

CPU:	Core i7-3770K, Intel	3.50GHz × 4(8)
RAM:	Corsair Vengeance 16 GB	2 × 8192 MB DDR3 1600MHz, 667MHz max bandwidth
MB:	P8Z77-V PRO, ASUStek Computer Inc.	-
GPU:	Nvidia GeForce 660ti, Gigabyte OC	1344 Kepler CUDA Cores, 915Mhz, 2GB GDDR5
PSU:	Corsair HX650	650W

Table 4.1: Hardware used in testing.

4.2.2 Software

Relevant software is listed in table 4.2.

OS:	Windows 8.1
C/C++ compiler:	MS C/C++ Optimizing Compiler v17.0.60610.1
CUDA Toolkit version:	6.5
CUDA compiler:	nvcc 6.5.13
GeForce driver:	344.75

Table 4.2: Software used in testing.

4.3 Test parameter values

4.3.1 Constants

Most of the parameters remain more or less fixed across tests. This is mainly because while they affect the numerical accuracy of the simulation, they do not affect runtime performance. The values selected for these parameters are mostly the same as those used by Larsgård [3].

Physical constants
Value of π used in calculations:

$$\pi = 3.14159265359,$$

Value of permittivity of free space (electric constant):

$$\epsilon_0 = 8.854187817 \cdot 10^{-12} F/m,$$

Value of electron charge (unit charge?):

$$e_{charge} = -1.60217657 \cdot 10^{-19} C,$$

Value of electron mass:

$$e_{mass} = 9.10938291 \cdot 10^{-31} kg,$$

Simulation parameters
Simulation grid size:

$$L_x = L_y = L_z = 0.2m,$$

Time step between iterations:

$$\Delta t = 1 \cdot 10^{-6} s,$$

Drag term:

$$d_{drag} = 0,$$

SOR settings
SOR relaxation factor:

$$\omega = 1.78,$$

Error threshold for convergence (currently unused, fixed nr of iterations):

$$Err_{threshold} = 1 \cdot 10^{-9},$$

Number of SOR iterations run:

$$N_{SOR\_iterations} = 128,$$

Test specific
Grid resolution (nr of points in each direction):

$$n_x = n_y = n_z = \text{default: } 64,$$

Simulation iterations run:

$$N_{iterations} = \text{default: } 2048,$$

Number of particles simulated:

$$N_{particles} = \text{default: } 256$$

4.3.2 Variables

The testing parameters are $N_iterations$, $N_{particles}$ and n_x, y, z. Other than for tests involving the number of iterations only the simulation loop is timed, the duration of one iteration of the simulation. Default values are $N_{particles} = 256$ and $n_x, y, z = 64$. The FFT based solver is chosen as default.

Certain simulation settings are derived from these terms, in particular the threadPerBlock and blockPerGrid settings for each kernel. These depends on either the grid resolution or the particle count, as shown in section 3.4, and while they are not testing parameters, they will vary as part of the testing.

4.4 Description of tests

Descriptions of all tests done are given below. These include definitions of the parameters tested and their values, as well as a brief description of the motivation behind doing the specific test.

Performance will be measured as runtime of the code being tested. Time will be measured using the procedure given in section 4.1.1 (see also devblogs.nvidia.com/[1]).

4.4.1 Number of iterations

This test just shows the increase in runtime as a function of the number of iterations.This should scale linearly with $T(N_{iterations}) = N_{iterations} \cdot T(1)$, and a deviation from this would likely indicate something being wrong, or at least interesting.

[1]http://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/

Test id:	**1 - Iterations**
Measure:	$T_{simul}(N_{iterations})-$ Simulation time, time spent in the simulation loop.
Parameters:	$N_{iterations}-$ Number of iterations.
Values:	$N_{iterations} \in [0, 2048]$
Motivation:	This test serves only as a confirmation that runtime increases linearly with the number of iterations.

Test 1: Iterations

4.4.2 Grid resolution

The resolution of the simulation grid ais likely one of a few key factors in determining performance. While we would usually set $n_x = n_y = n_z$ and the effect of these on simulation accuracy should be equal in theory, the way the arrays are stored in memory is different. For instance any $n_x \in [1, 16]$ would result in the same memory footprint due to padding (here assuming elements 8 byte wide and a 128 byte alignment requirement).

Test id:	**2 - Grid - Isotropic scaling of** n
Measure:	$T(n_x = n_y = n_z)-$ Application runtime.
Parameters:	n_x, n_y, n_z- Resolution in each dimension.
Values:	$n_x = n_y = n_z \in [0, 256]$
Motivation:	Several of the kernels use one thread per grid element, and this metric is an important one. Two competing effects makes this an interesting one; a low resolution grid means less work to be done, potentially leaving multiprocessors on the card idle while reducing the number of memory accesses, but a high resolution one results in fewer particles per cell, thus potentially reducing write conflicts from particle-based kernels. See also test 4.8

Test 2: Grid - Isotropic scaling of n

Test id:	**3 - Grid - Odd and even valued** n
Measure:	$T(n_x = n_y = n_z)-$ Application runtime.
Parameters:	n_x, n_y, n_z- Resolution in each dimension.
Values:	$n_x = n_y = n_z = 2 \cdot k, k \in [0, 128]$ $n_x = n_y = n_z = 2 \cdot k + 1, k \in [0, 128]$
Motivation:	Simulation time using the FFT solver with grid sizes even and odd separated, to easier compare performance.

Test 3: Grid - Odd and even valued n

4.4.3 Particle count

First a measure of the impact the number of particles has on execution time, and then we see how different particle counts perform for different grid resolutions.

Test id:	**4 - Grid - Scaling of n along one dimension.**
Measure:	$T(n_x = n_y, n_z)-$ Application runtime.
Parameters:	n_z- Resolution in z dimension.
Values:	$n_x = n_y \in [0, 256]$
	$n_z \in [0, 256]$
Motivation:	Here we test the effects on performance of scaling n along a single dimension, for a different configuration of the other two dimensions.

Test 4: Grid - Scaling of n along one dimension.

Test id:	**5 - Particles1 - Performance impact of particle numerosity.**
Measure:	$T(N_{particles})-$ Application runtime.
Parameters:	$N_{particles}-$ Number of particles.
Values:	$N_{particles} \in [0, 65536]$
Motivation:	The number of particles affect device saturation; few particles mean less work to be done, fewer competing reads/writes, while a higher number of particles mean a larger number of multiprocessors will be kept busy.

Test 5: Particles1 - Performance impact of particle numerosity.

Test id:	**6 - Particles2 - Variation of both resolution and numerosity.**
Measure:	$T(N_{particles}, n)-$ Application runtime.
Parameters:	$N_{particles}-$ Number of particles.
	$n_x = n_y = n_z-$ Resolution in each dimension.
Values:	$N_{particles} =\in [0, 65536]$
	$n_x = n_y = n_z \in [0, 256]$
Motivation:	It can be interesting to see how particle number and grid resolution together affect performance, such as which parameter affects performance the most. Given that the complexity of several operations is $O(n_x \cdot n_y \cdot n_z \cdot N_{particles})$ makes it interesting to see how performance scales when both are increased.

Test 6: Particles2 - Variation of both resolution and numerosity.

4.4.4 Comparison

This pair of benchmarks time a single execution of each solver for varying grid sizes. The impact the grid size has on their performance will be interesting to see, as well as which has better performance scaling.

Test id:	**7 - Solvers - Comparing the runtime of the solvers.**
Measure:	$T_{solver}(Solver, n)-$ Solver execution time
Parameters:	The type of solver used.
	$n-$ Grid resolution.
Values:	Solver $-$ cuFFT-based solver, SOR-solver.
	$n \in [0, 256]$
Motivation:	By testing both solvers for various problem sizes we can get a measure of how well they scale, as well as which of them offers better performance.

Test 7: Solvers - Comparing the runtime of the solvers.

4.4.5 Kernel runtimes

These test involves measuring runtime for each kernel as functions of resolution and particle count. By timing each kernel and comparing these results with the solver results we can see where bottlenecks are, and may learn how the different kernels behave with variations of the simulation parameters. This is useful to identify what optimizations can be made and which kernels have the most potential.

Test id:	**8 - Kernels - Variation of resolution and particle numerosity.**
Measure:	$T_{kernel}(n, N_{particles})-$ Kernel execution time.
Parameters:	n_x, n_y, n_z- Grid resolution.
	$N_{particles}-$ Number of particles.
Values:	$n \in [0, 256]$
	$N_{particles} \in [0, 65536]$
Motivation:	Examining the performance for the different kernels for various configurations of particles and resolution, we may be able to identify potential bottlenecks. These results may assist in explaining global results above.

Test 8: Kernels - Variation of resolution and particle numerosity.

4.5 Suggestions for further testing

4.5.1 Solver

It would be useful to measure which of the two solvers offer better accuracy per performance. A test here could be to set the number of SOR iterations so both solver use the same amount of time and then compare the numerical accuracy of the results, but to do this we need to be able to tell which solver gives the best result, knowledge I do not have at the present.

Assuming that the FFT as an exact solver gives the correct answer, we could run both and then find a measure of the error by taking the average of the difference between the result for each element. By plotting this measure against grid resolution and SOR iterations we could study how these increase the accuracy of the SOR. Combining this with a plot of performance as a function of SOR iterations and grid resolution, we could see accuracy gain compared to performance loss, an interesting metric.

4.5.2 Single precision data type

If the implementation is extended to include an option of using single precision data, it would be useful to test the speedup this would gain us, as well as the inevitable loss of accuracy. The results of these tests could then help give an idea of the trade-off involved in choosing one or the other.

For the single precision implementation we would also want to run most of the tests described above. At least for the GeForce series of GPUs, throughput for single precision operations is significantly higher than for double precision operations, more so than the reduction in bandwidth congestion. For this reason memory latency could become even more of a factor, potentially reducing actual speedup, and test results might paint a different picture.

4.5.3 System

Other tests that could be of interest are comparisons between running the same configuration on different systems. In particular this goes for the GPU involved. The Tesla series of GPUs have significantly better double precision capabilities compared to the GeForce series, relative to their single precision performance. A configuration that would fit on both systems should perform better on the Tesla because of the number of available double precision floating point units, even taking into account core, memory and bus clock speeds.

Chapter 5

Results

This section will present some results from testing, in the form of 2D and 3D plots as appropriate, along with a brief interpretation of the data.

All results are timed as described in section 4.1.1. Python with numpy and matplotlib is used to present the data in the diagrams below. Runtime is measured in milliseconds, and unless otherwise specified is the time taken for one iterations of the simulation, or one run through the tested code.

5.1 Number of iterations

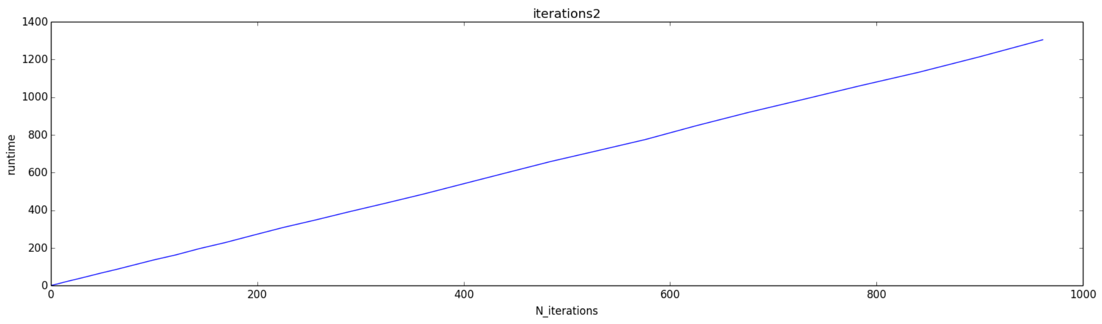


Figure 5.1: Simulation loop runtime as a function of $N_{iterations}$.

As be expected the simulation runtime scaled linearly with the number of iterations, as can be seen on figure 5.1. Runtime was approximately $1.35\frac{milliseconds}{iteration}$.

5.2 Grid resolution

The isotropic scaling of grid resolution is shown in figure 5.2. We see result very similar to that of the FFT-only test (fig 5.9). For n =221, 227, 231, 235, 239, 247, 251 and 255 the simulation fails, unable to allocate memory for the FFT work area.

Looking at figure 5.3 we see that runtime seems to scale linearly with n_z, and has quadratic growth for $n_x = n_y$. This makes sense since multiplying

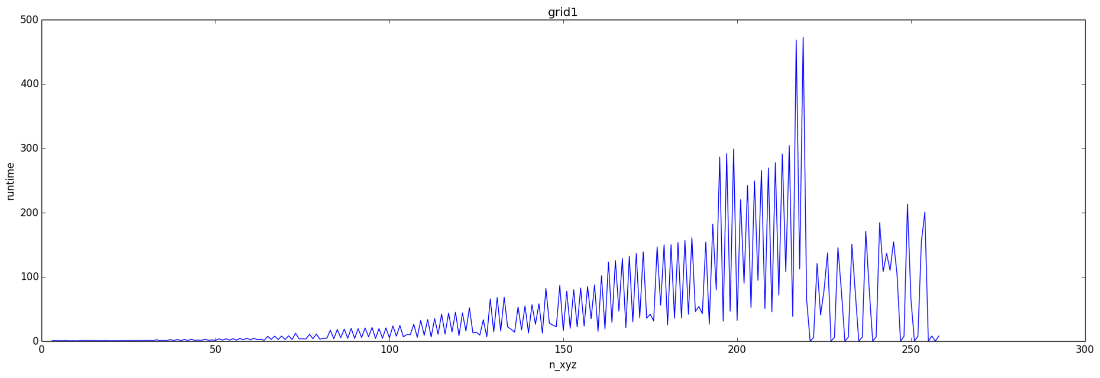


Figure 5.2: Simulation iteration runtime as a function of grid resolution. Isotropic.

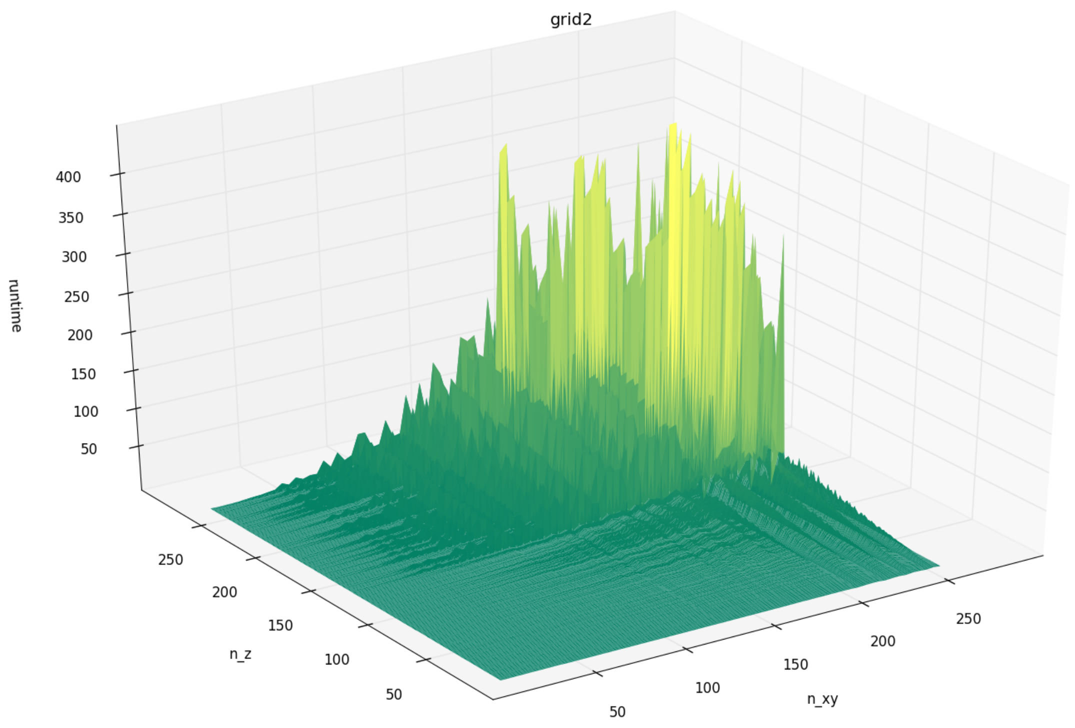


Figure 5.3: Runtime as a function of grid resolution, with scaling along z axis.

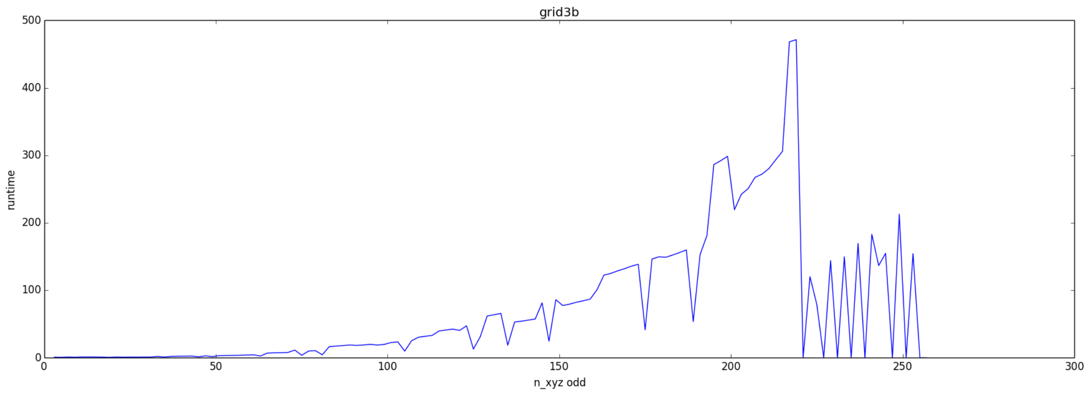


Figure 5.4: Runtime as a function of grid resolution, isotropic, odd valued size.

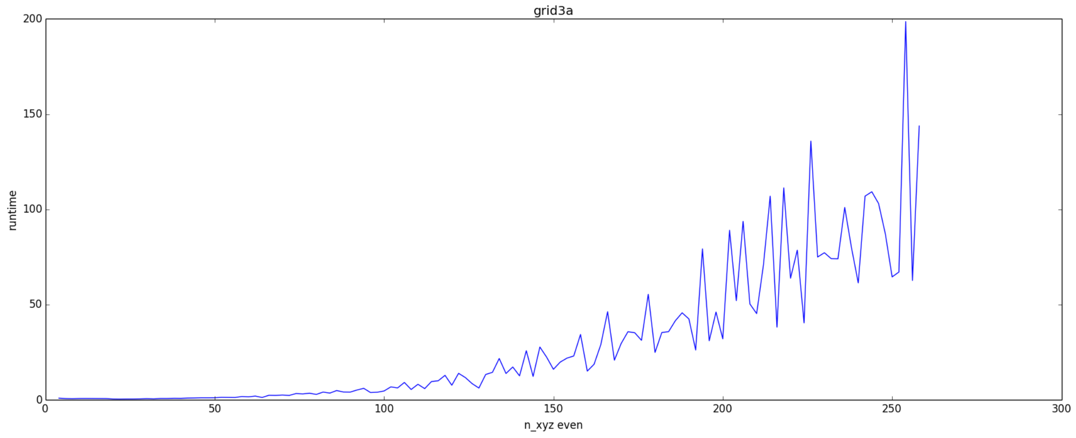


Figure 5.5: Runtime as a function of grid resolution, isotropic, even valued size.

resolution by a would result in $T(n_x \cdot n_y \cdot (a \cdot n_z)) = n \cdot T(n_x \cdot n_y \cdot n_z)$ while $T((a \cdot n_x) \cdot (a \cdot n_y) \cdot n_z) = n^2 \cdot T(n_x \cdot n_y \cdot n_z)$. For high values of both we see the same tendency as for isotropic scaling, where the difference between superior and inferior values becomes dominant, giving an irregular plot.

Figures 5.5 and 5.4 show the isotropic scaling decomposed into even and odd numbers, helping to isolate different effects. First off we see that while the shape of the plots are similar, the odd valued shows approximately twice the runtime of the even valued plot. We can also see that only odd valued resolutions crash the simulation. For the odd valued plot it is especially easy to isolate ideal resolutions: all the downward spikes (except failures) occur for values of $n = 3^b \cdot 5^c \cdot 7^d$, while the highest runtimes measured were for $219 = 3 \cdot 73$ and $217 = 3 \cdot 31$. Interestingly the simulation fails for 239, a prime, but appears to succeed for 241, another prime.

While the even plot shows a lot more spikes, the same holds true with regard to them being on the form $n = 2^a \cdot 3^b \cdot 5^c \cdot 7^d$. The highest measured runtime belongs to $254 = 2 \cdot 127$.

5.3 Particle count

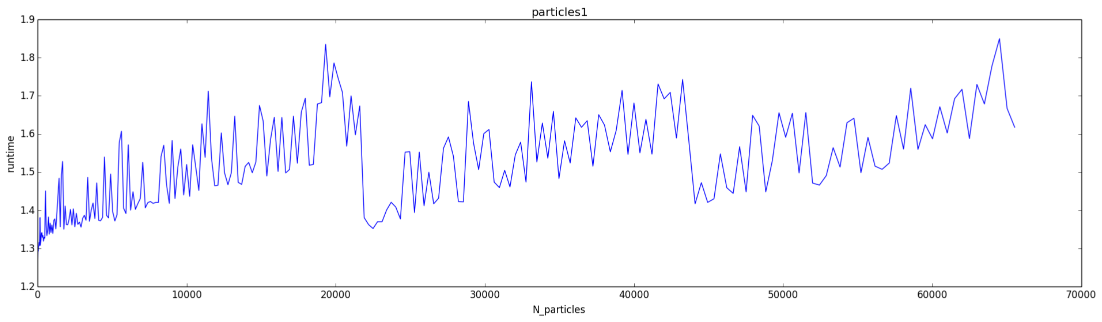


Figure 5.6: Runtime as a function of the number of particles.

Figure 5.6 shows how runtime varies with the number of particles, and while it appears to fluctuate a lot, it should be noted that the range of variation is $[1.3, 1.8]$. Compared to the grid resolution above it seems safe to say that the number of particles has a relatively low effect on the performance of the simulation. Worthy of note is that the plot seems to be repeating, showing the same trend for [0 - 22000], [22000 - 44000] and [44000 - 66000].

Figure 5.7 confirms that resolution has a much larger impact on performance than particle number, and it appears that the variations above may simply be noise. Indeed the only major variation along the particle axis is that the simulation fails for lower resolutions with increasing particle number, and for configurations with $N_{particles} > 5000$ and $n > 237$. Again the simulation fails because of insufficient memory, so while particle number may not impact runtime it has a clear impact on memory usage, and thus problem size.

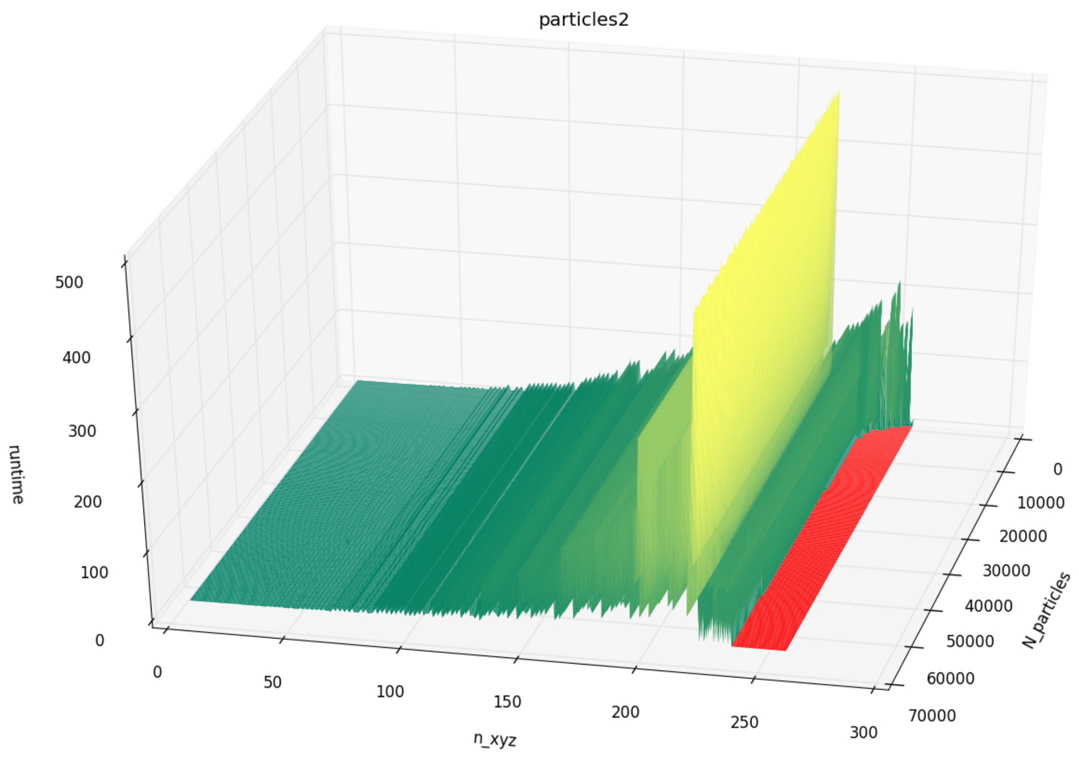


Figure 5.7: Runtime as a function of both the number of particles and grid resolution. Isotropic scaling of resolution.

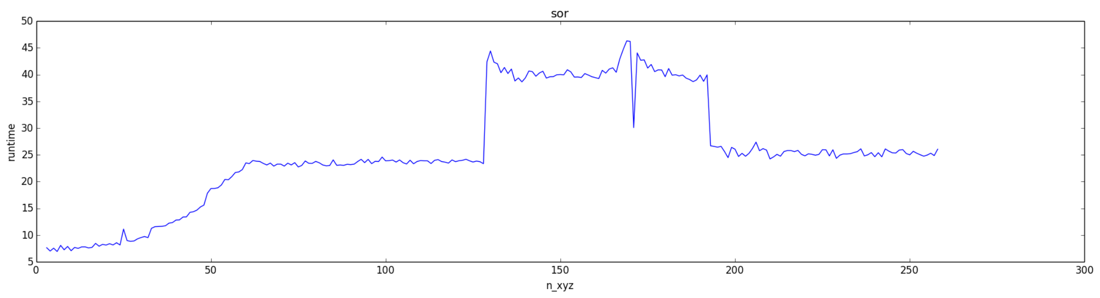


Figure 5.8: Comparison of solvers. Runtime of each solver as a function of grid resolution, isotropically scaled.

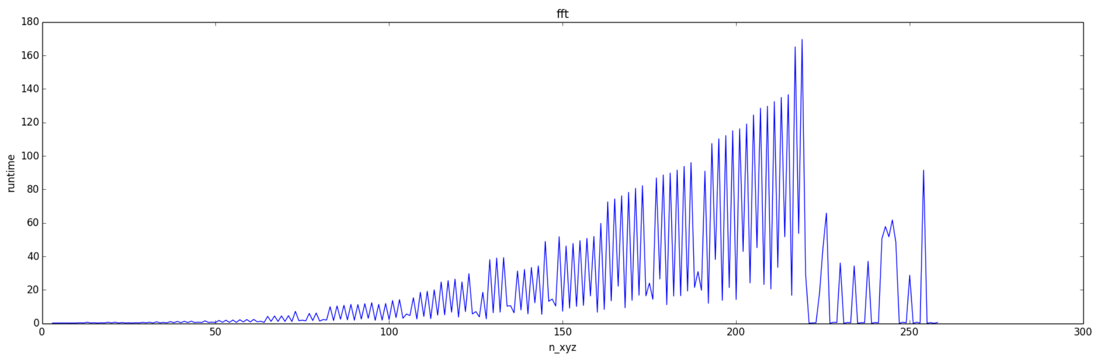


Figure 5.9: FFT solver runtime by grid resolution.

5.4 Solvers

In figures 5.9 and 5.8 we see solver runtime. We see that the runtime of either solver dominates that of the kernels (see below), explaining why $T_{simulation}(n_x, n_y, n_z) \approx T_{FFT}(n_x, n_y, n_z)$. Also of interest, while the FFT shows it's log-linear growth, the SOR is almost constant after $n = 64$.

5.5 Kernels

5.5.1 determineChargesFromParticles

This kernel shows an interesting performance metric, where runtime goes down as the grid resolution increases, and is more or less constant in the number of particles. Runtime evens out around $0.05ms$.

5.5.2 electricFieldFromPotential

Constant in the number of particles, runtime increases rapidly up to $0.10ms$ for $n = 62$, before easing off and fluctuating between 0.10 and $0.15ms$.

5.5.3 updateParticles

Apart from poor performance for low grid resolutions, this kernel has a runtime that seems to be more or less independent of both particle numerosity and grid resolution. Also has an average runtime around $0.5ms$

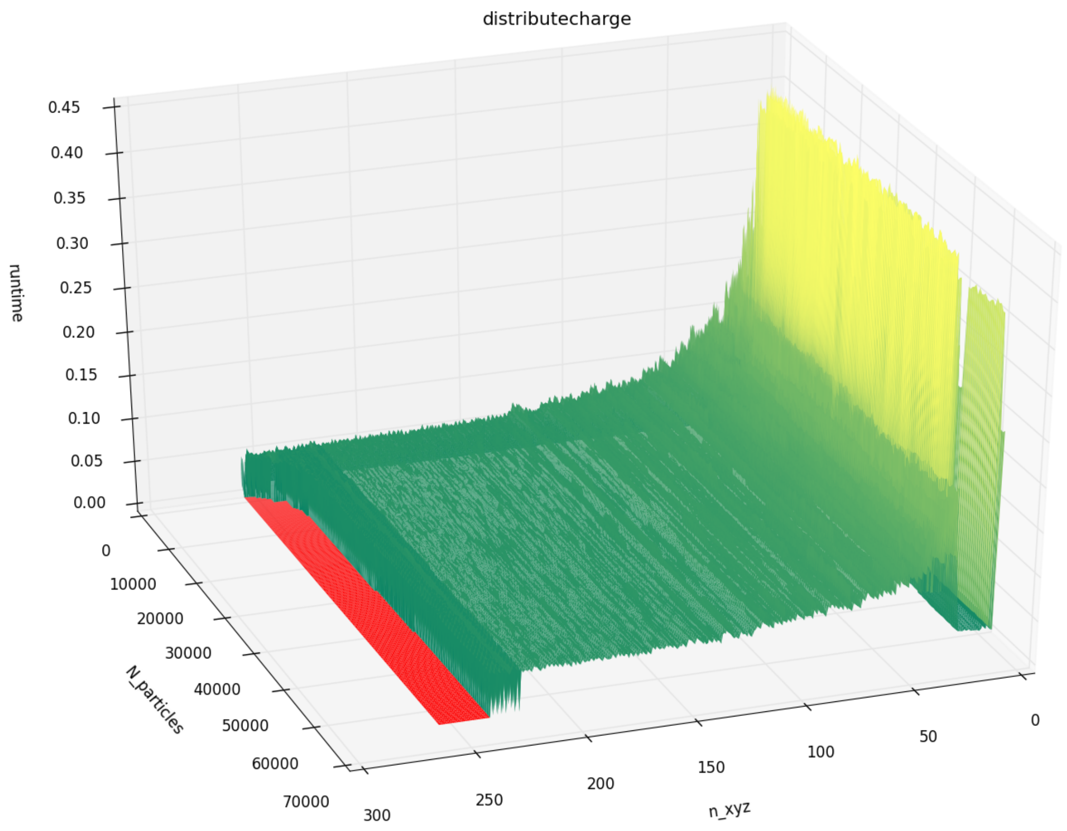


Figure 5.10: Runtime of the determineChargesFromParticles() kernel, as a function of both grid resolution and number of particles.

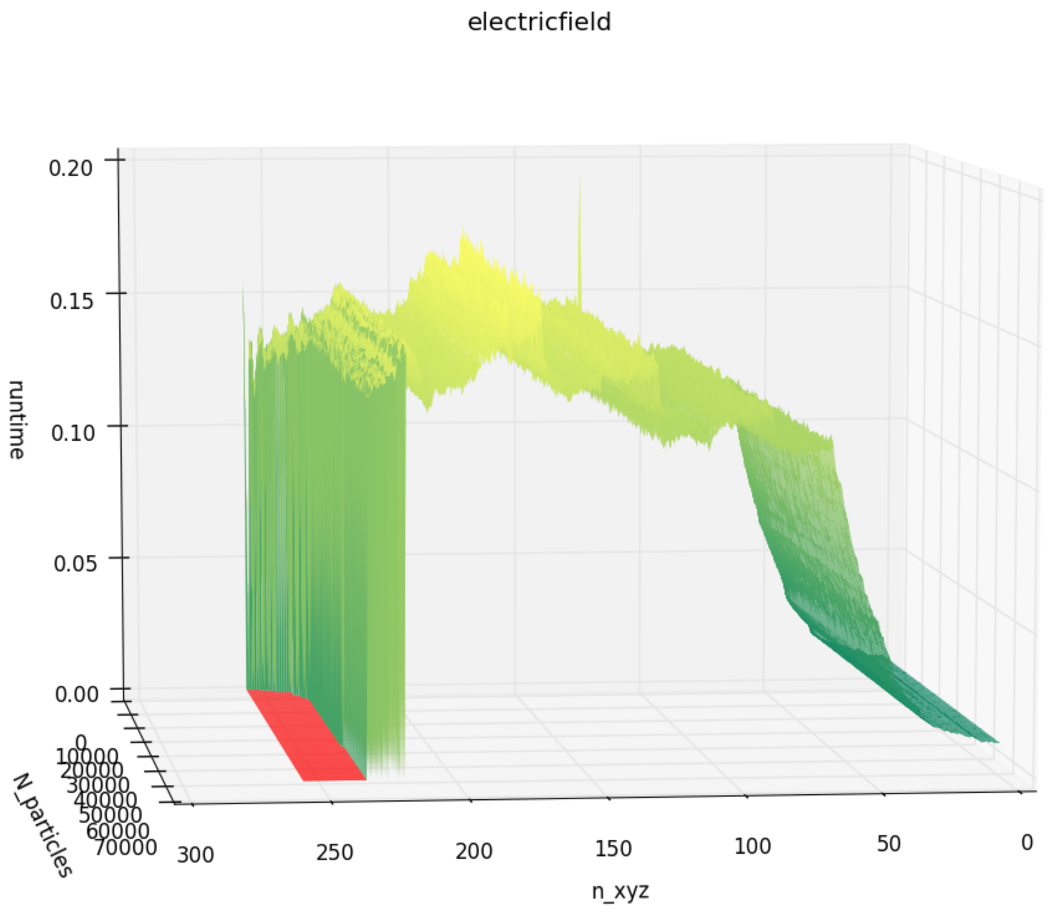


Figure 5.11: Runtime of the electricFieldFromPotential() kernel, as a function of both grid resolution and number of particles.

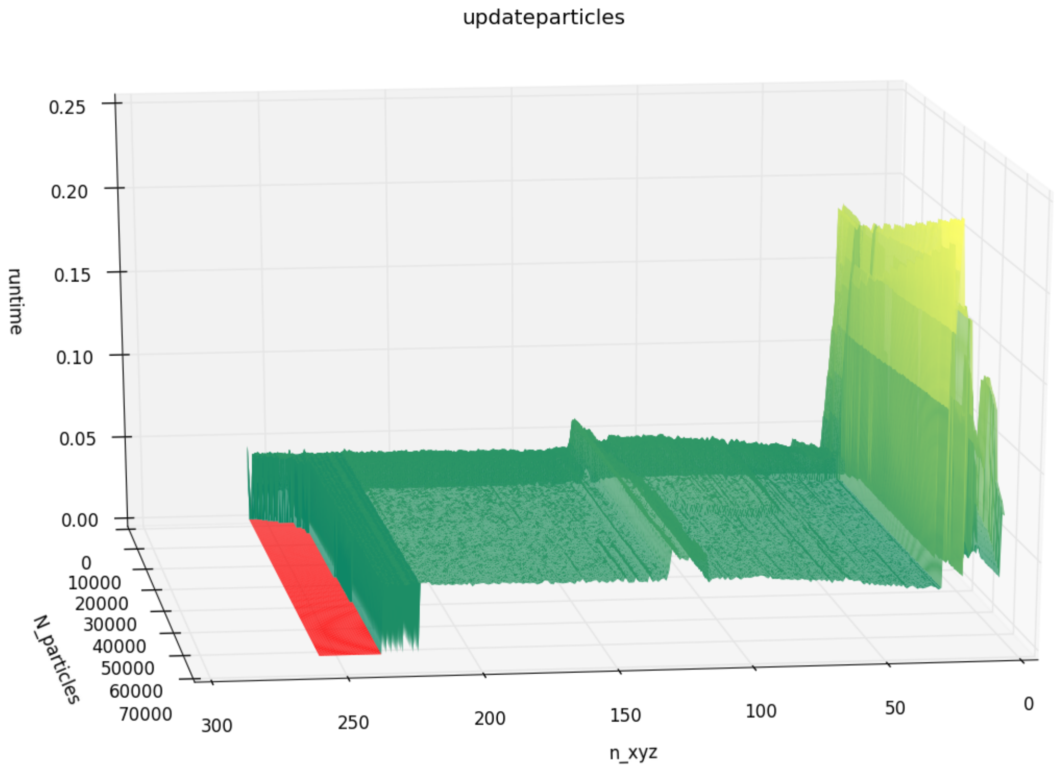


Figure 5.12: Runtime of the updateParticles() kernel, as a function of both grid resolution and number of particles.

Chapter 6

Discussion

6.1 Results

Some important characteristics noted during testing were:

- The main factor in determining runtime is the grid resolution.
- The number of particles has a comparatively small performance impact.
- Kernel run times are relatively constant for increasing resolution.
- The vast majority of the computational work is done in the solver step.
- Our SOR solver has a high constant factor, but runtime remains more or less constant for different grid resolutions.
- GPU memory size is a limiting factor as far as problem size goes, preventing grid resolution much beyond 256^3 for mid-end cards, and about 512^3 for high-end cards. Particle number also has a noticeable impact on memory usage.

That execution time of charge distribution kernel decreases with larger resolution might seem counterintuitive, but since it is called on a per-particle basis an increase in number of grid elements will have no impact on the work load. However, because the charge distribution kernel writes to some grid element based on the particles' position, this makes fewer particles write to the same element. By reducing these conflicts threads no longer have to wait for access, and thus performance is improved.

Since the kernels don't appear to scale poorly with resolution the answer must be to improve the solver's scaling. As the implementation ensures data alignment regardless of n_x, n_y and n_z, and cuFFT is noted to perform best for dimensions on the form $2^a \cdot 3^b \cdot 5^c \cdot 7^d$, it seems prudent to enforce this restriction on the grid resolution. Even for values on this form, even ones tend to perform better than odd ones, suggesting that having a power of two as a factor is desirable.

Even so, the FFT is an $O(n \cdot log\, n)$ complexity algorithm, while the SOR is relatively unaffected by the resolution. Since the SOR is done in-place memory is much less of a factor than for the FFT solver, and it handles large grids much better. As an element only reads from its neighbors, there is no increase in workload per thread when the number of elements is increased. With the number of SOR iterations fixed at 128 runtime seems almost constant at 25 ms, other than between

While resolution is still limited by memory, a measure that mey increase performance tenfold while reducing memory footprint is to switch from double to single precision floating point values. For the GeForce series of GPUs the number of operations per clock cycle is significantly lower for double precision. For compute capability 5.0 128 single precision operations compared to 1 double precision, while for 3.0 the numbers are 192 and 8 [11, sec. 5.4.1]. In addition to freeing up memory for additional resolution, there is a marked increase in processing power, utilizing far more of the potential of the GPU.

While this holds true for consumer-grade GPUs, the Tesla series of accelerators has far better double precision performance, with those of compute capability 3.5 capable of running 64 double precision operations per cycle. Though, if single precision data turns out to be sufficiently accurate, the benefit of reduced memory footprint may still justify the trade off. An interesting test in this regard would be to compare the accuracy loss of switching to single precision versus the gain from increasing resolution. Testing this implementation on a Tesla card would also be interesting, given the card's double precision performance.

6.2 Implementation flaws, and potential fixes

Listing some of the flaws to take into account when evaluating the implementation:

- No use of shared memory. Important when working with CUDA, avoiding global memory accesses by utilizing the much faster on-chip shared memory, this should be one of the first improvements worked on. Especially the SOR-kernel should gain performance this way.

- Kernel fusion. The update particle and distribute charge kernels are called sequentially, with no dependencies on the first to finish execution before starting the second, and one could easily avoid expensive API calls by fusing these kernels together. Looking at the performance metrics however, The performance benefit would be minimal compared to optimizing the solver. Whether the code becomes more or less organized by fusing them is another matter.

- The red-black SOR currently consists of two consecutive kernel calls with an offset of one element. While this makes the implementation easy to understand a way of fusing these calls should be investigated. A argument in favor of the current implementation is that kernel calls are a trivial way to ensure the device wide synchronization needed.

- Particle sorting. Currently the particles maintain order in the particle array, and the same particles are put in a warp together every iteration. This might be wasteful, since particles on opposite sides of the grid might execute together, thus leading to scattered memory access. By sorting particles so that particles close together are executed together, we might get coalesced memory access, or even avoid different warps reading the same data. When writing data in the charge distribution kernel we would like to reduce the number of writes to global memory by handling writes to the same element in shared memory. See Elster's original work[1, sec. 4.4] for more details on particle sorting.

6.3 Parallelizing PIC codes for CUDA

Overall the particle-in-cell simulation seems well suited for parallel computing. All steps can be parallelized, and there is little time spent in sequential parts of the code. Considering Gustafson's law (sec. 2.3) we see that one can easily increase the parallel work without scaling the serial part, thus allowing unlimited speedup in theory. Limiting factors are of course physical memory and processor throughput.

More specifically for the CUDA architecture, the mapping of a simulation step seems intuitive. Especially the grid-based kernels map easily, given the GPU's predisposition towards 3D coordinates and stencil based computation. An issue here is with the pitched pointers used to ensure data alignment. Using pitched pointers means accessing data using pointer arithmetic that produces cluttered code. While this means we have fine grained control over data location, it also leaves more room for error, and pointer calculation was indeed a major source of bugs during development. Additionally, if we restrict the grid dimensions to a power of two for optimal FFT performance, data alignment is already taken care of (assuming sufficiently large dimensions), and normal arrays could be used instead.

If we want a grid resolution larger than what is possibly on GPU memory alone there is the possibility of storing the complete grid in host memory and transferring back and forth every iteration, but this will likely become prohibitively expensive considering the amount of data to be transferred. Other options are upgrading to a larger GPU memory, or adding more GPU's to the system. By storing parts of the grid on different devices we only need to transfer values on the border between them. Communication speed between devices is still an issue, but technologies like Nvidia's NVLink shows potential in this regard, promising speeds of up to 5 to 12 times what we would get over the PCIe 3 bus [12]. For a 2 GPU 3D FFT of size 256^3 the speedup of using NVLink is apparently nearly 2.25 over a PCIe configuration.

That the cuFFT library is readily available is another benefit of using CUDA. In CUDA version 6.5 callback functions were added to FFT execution calls, since one usually follows a transform with some operation on the transformed data. This way one avoids shifting control to the CPU before running the solver kernel, and using this for both the forward transform-solver and inverse transform-electric field pairs of kernels, we avoid two of these calls per iteration. The cuFFT library currently only supports acceleration on two GPUs, so this limits hardware scaling somewhat if this library is to be used. An additional limit as that the entire transform must fit in memory of the GPU's involved[13, sec. 2.8.4].

The SOR also translates nicely to CUDA, especially if we optimize it using shared memory. One bottleneck is the error checking, where the current implementation instead uses a fixed number of iteration. Checking whether error is sufficiently small for all elements requires a global maximum reduction. Since this can be expensive for a large number of elements, finding a number of iterations that is sufficient for convergence is a preferable solution. As the amount of work done by the kernel is relatively low, an increase in the number of iterations may well outweigh the cost of calculating the error every iteration.

6.4 Solvers

A straightforward comparison of the solvers' performance would be unfair considering that cuFFT is a highly optimizing library while the SOR makes no use of shared memory yet. For all grid sizes 128 and below the FFT-based solver completely outperforms the SOR, with results for 64 and below being an order of magnitude lower. It is therefore surprising to see that the SOR has a lower runtime for $n = 256$, about half that of the FFT. In addition the SOR can be expected to handle irregular (even prime) grid dimensions far better, and since cuFFT resorts to a slower less accurate algorithm for large primes [13, sec. 2.12], a lot of the difference would be made up even for smaller grid sizes.

If the SOR is upgraded with shared memory one should also identify the best possible configuration of elements per warp/ block. Since an element reads three values from its own row plus an element from the rows above, below, in front and behind, a total of five rows must be read to compute a value. By letting a warp handle a single row we reduce the number of coalesced memory access to five rows. As an example lets compare configurations of threads per warp of (8, 8, 4) and (256, 1, 1), both handling 256 elements. (8, 8, 4) reads eight elements from $8 \times 4 = 32$ rows, all of which are likely scattered across the array. (256, 1, 1) has coalesced reads from it's own row, plus four others for a total of five. Assuming dimensions of 256^3 this will also make the upper middle and lower rows successive in memory, resulting in only three scattered accesses.

Since the FFT is quite communication intensive the performance benefit of switching to single precision data types is likely greater than for the SOR kernel, and a new comparison using an optimized SOR kernel and single precision data is warranted. Nevertheless, both solvers seem to perform comparably for problem sizes fitting in GPU memory, and the choice is still one of selecting the one appropriate given the boundary conditions of the problem.

Chapter 7

Conclusion

The following questions were posed in the introduction as a goal for the project:

a. Can the simulation be implemented in CUDA? Easily?

b. Which issues are there when mapping the problem to CUDA? Which solutions or alternatives are available?

c. Which parts of the simulation turn out to be critical paths? Can these parts of the code be improved further?

The answer to a is yes, it is entirely possible to implement particle simulations in CUDA. The CUDA architecture it very well suited for three dimensional computing as a result of it's origin in computer graphics, and with a large number of libraries available the implementation need not be to complex. Whether it is easy to implement largely depends on ones point of view. The implementation is a relatively straightforward translation of math to code, but indexing pitched arrays and managing indexes can be difficult, and often makes up the majority of a kernel. The cuFFT provides a useful framework for GPU-accelerated FFTs, but requires some learning and setup to get working. Compared to OpenCL and MPI a CUDA implementation might be easier to develop, but will be far less portable.

Issues encountered during development are mainly those common to developing for CUDA, such as ensuring coalesced memory access, avoiding write conflicts, and minimize PCIe traffic. Issues related to programming include staying within bounds of arrays, handling pointers correctly, and identifying sources of errors.

As might be expected the solver step of the simulation turned out to be the most compute intensive step. For cuFFT there is only so much one can do to improve performance, but among criteria listed by Nvidia are using single precision data types to reduce bandwidth cost, and keeping transform dimensions a multiple of low primes, ideally a power of two. For the SOR kernel we can use shared memory to reduce memory access time, switch to single precision data types, and identify the lowest number of iterations necessary for the solution to converge with a sufficiently low error. How to distribute elements among warps need to be investigated to find out how to best take advantage of shared memory.

To conclude, the author's opinion is that CUDA can be used to accelerate particle simulations with relatively few issues. The performance of a simulation depends on the accuracy required, more so than for a similar simulations running on the CPU. To get a better measure of potential speedup using CUDA, more optimization of the code is necessary, as well as a comparison with CPU code running on similar hardware.

7.1 Future Work

As discussed, the code is not very optimized, lacking even proper use of shared memory where applicable. In addition to this changing data types from single to double precision and fusing kernels are opportunities that should be investigate. Algorithm wise there may be some gain in devising some clever particle sorting scheme, but impact of this appears relatively low compared to the more critical solver performance. See section 6.2 for details on suggested improvements.

The particle tracing mechanism could likely be improved to minimize performance impact, and more easily facilitate further processing. This includes both how particles are traced during simulation and how trace data is output after the simulation has ended (see also section 3.5). Real time animation of particles and electric field would also be interesting.

Further tests have been suggested in section 4.5, some of these assuming the performance improvements above have been implemented.

Bibliography

[1] Anne C. Elster, "Parallelization Issues and Particle-In-Cell Codes", 1994, Cornell University, USA

[2] Jan C. Meyer, "Emerging Technologies Project: Cluster Technologies, PIC codes: Eulerian data partitioning", 2004, NTNU, Norway

[3] Nils M. Larsgård, "Parallelizing Particle-In-Cell Codes with OpenMP and MPI", 2007, NTNU, Norway

[4] Intel, "About Sh", *last visited on 2 January 2015*

[5] "November 2014", TOP500 The List, November 2014, *last visited 2 January 2015*

[6] Ian Buck, Jeff Nichols and Rob Neely, "GPU Acceleration: What's Next", Video recording: `http://on-demand.gputechconf.com/supercomputing/2014/video/SC401-accelerated-computing-exascale.html`, 2014, Supercomputing 2014, Nvidia

[7] Mark Harris, "Maxwell: The Most Advanced CUDA GPU Ever Made", Nvidia developer zone - Parallel Forall, 18 September 2014, *last visited 2 January 2015*

[8] Jeff Larkin, "OpenMP and NVIDIA", 2013, Super Computing 2013, NVIDIA

[9] "NVIDIA CUDA Compiler Driver NVCC", CUDA Toolkit Documentation, 1 August 2014, *last visited 1 January 2015*

[10] "User Guide for NVPTX Back-end", LLVM Compiler Infrastructure, 29 December 2014, *last visited 1 January 2015*

[11] "CUDA C Programming Guide", CUDA Toolkit Documentation, 1 August 2014, *last visited 2 January 2015*

[12] "NVIDIA NVLINK HIGH-SPEED INTERCONNECT", High Performance Computing, 2014, *last visited 2 January 2015*

[13] Name, "cuFFT", CUDA Toolkit Documentation, 1 August 2014, *last visited 2 January 2015*

All figures have been produced by the author using Inkscape, matplotlib and Google Draw.