NTNU – Trondheim
Norwegian University of
Science and Technology

Challenges Adapting CUDA PIC Codes to multiple GPUs

Olav Emil Eiksund

Master of Science in Computer Science
Submission date: July 2015
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Challenges Adapting CUDA PIC codes to multiple GPUs

Olav Emil Eiksund

July 18, 2015

Problem description

The Particle-In-Cell method is a particle simulation that includes a PDE solver, with applications in modelling particle acceleration and plasma waves.

This project includes adapting a naive CUDA-implementation of Particle-In-Cell codes for multiple GPUs. Previous work has shown that 3D Particle-In-Cell codes require several orders of magnitude more particles than a 2D version, for which the limited memory of a GPU is a bottleneck. The project includes investigating how additional GPUs may be used to increase possible problem size, and how this will affect performance.

Abstract

A Particle-In-Cell code is a common particle simulation method often used to simulate the behaviour of plasma. In this work, a parallel PIC code is developed in CUDA, with a focus on how to adapt the method for multiple GPUs. An electrostatic three dimensional PIC code is developed, with an FFT-based solver using the cuFFT library.

Several issues related to parallelizing the PIC code are discussed, along with the performance on one and two GPUs compared to the CPU version we developed based on OpenMP and FFTW as a benchmark. For most problem sizes, the application is found to be memory bound, with the speed of the memory interface playing a larger role than the double precision performance. Alternatives to some of the naive solutions are discussed, with suggestions for how the simulation could be implemented for a shared memory computer. Benchmarks were performed on several GPU platforms including Nvidia Tesla K20 and GTX 980, and the challenges of getting the implementations to simulate plasma oscillations are discussed.

Sammendrag

Particle-In-Cell koder er en kjent type partikkelsimulering brukt til å simulere plasmaer. I dette arbeidet utarbeides en parallell PIC kode i CUDA, som videre tilpasses kjøring på flere GPUer. PIC koden er tredimensjonal og elektrostatisk, og benytter en FFT-basert løser implementert gjennom cuFFT-biblioteket.

Ytelsen på en og to GPUer blir sammelignet med ytelsen til en CPU-variant basert på OpenMP og FFTW. Programmet viser seg å være bundet av minneaksesser, hvor hastigheten på minnet dominerer double precision regnehastighet i ytelsespåvirkning. Alternativer for noen av de mindre heldige løsningene blir diskutert, og et forslag til en shared memory-implementasjon blir nevnt. Ytelsestester ble kjørt på Nvidias Tesla K20 og GTX 980 GPU+platformer, og utfordringer rundt det å få implementasjonen til å simulare plasmasvingninger blir diskutert

Acknowledgements

I would like to thank my advisor Anne C. Elster for providing me with this project, and for introducing me to the exciting field of GPGPU and HPC.

My advisor and I would like to thank NVIDIA for their support of the IDI/NTNU HPC-Lab through their NVIDIA GPU Research and Teaching Centers at NTNU, and NTNU for the generous equipment grants to HPC-Lab.

Many thanks to Trygve R. Sørgård for extremely valuable feedback, including proofreading and providing a deeper understading of the physics involved, and many interesting conversations.

My fellow students at the NTNU HPC-lab have provided support and new insight, and many educating experiences. I would also like to thank the students and teachers from NTNU in general, and Computer Science in particular for providing a stimulating environment and many memorable experiences throughout my time at NTNU.

I would also like to thank my family and friends who have guided me down the path that has led me here, and particularly my parents Anne Betty and Gudmund for encouraging my studies.

My time time at NTNU would not be the same without my girlfriend Kathrine, whom I would like to thank for always supporting me and keeping me motivated during difficult times.

Contents

1 Introduction **1**
1.1 Outline 3

2 Background Theory **5**
2.1 Particle-In-Cell codes 5
2.1.1 Particles and Cells 6
2.1.2 Charge density contribution from particles 6
2.1.3 Solving Poisson’s equation for the electric potential 7
2.1.4 Deriving the electric field from the potential 8
2.1.5 Updating particle positions and velocities 8
2.2 The Fast Fourier Transform solver 9
2.2.1 Solving PDEs with the Fourier Transform 10
2.2.2 The FFT algorithm 11
2.3 Parallel computing and GPGPU 11
2.3.1 History and motivation for parallel computing 12
2.3.2 General Purpose computing on Graphics Processing Units 13
2.3.3 Parallel performance measures and potential speedup . . 14
2.3.4 Architectures 15
2.4 CUDA - Compute Unified Device Architecture 17
2.4.1 The CUDA Programming Model 17
2.4.2 Performance factors 20
2.4.3 Alternative CUDA interfaces 21
2.5 cuFFT - the NVIDIA CUDA Fast Fourier Transform library . . . 22
2.5.1 cuFFTXt - Multi-GPU support in cuFFT 22
2.6 Related work 22

3 Implementation **25**
3.1 Goals for the implementation 25
3.2 Simulation overview 27
3.2.1 Setup 27
3.2.2 Simulation loop 28
3.2.3 Cleanup 31
3.3 Parallelization issues 31
3.3.1 Atomic operations 31
3.3.2 Overlapping memory transfers with kernels 31
3.3.3 cuFFTXt limitations 32
3.3.4 Particle migration 32
3.4 Particle tracing 33
3.4.1 Tracer implementation 33
3.4.2 Visualization script 34
3.5 The CPU benchmark comparison 34
3.5.1 OpenMP 34
3.5.2 FFTW 35

4 Results and discussion **37**
4.1 Hardware 37
4.2 Goals 38
4.3 Profiling 38
4.3.1 Single GPU 39
4.3.2 Two GPUs 40
4.3.3 CPU 42
4.4 Simulation performance 43
4.4.1 Particle count 43
4.4.2 Grid z dimension 45
4.4.3 Frame count for 1000 iterations 46
4.5 Stability testing 47
4.5.1 Plasma oscillations 47
4.5.2 Two-stream instabilities 50
4.6 Atomic operations 51
4.7 Particle exchange 52
4.8 Particle partitioning and load balancing 53
4.9 Variant 54

5 Conclusion **57**
5.1 Suggestions for future work 58

Bibliography **59**

A CUDA and cuFFT examples **63**

A.1 CUDA example 64

A.2 cuFFT example 66

List of Figures

2.1 Trilinear interpolation 7
2.2 Conceptual illustration of a CUDA architecture 18

3.1 Illustration of boundary storage 30

4.1 Plot of profiling results, one GPU 39
4.2 Plot of profiling results, two GPUs 41
4.3 Plot of profiling results, CPU 42
4.4 Plot of performance results, particles 44
4.5 Plot of performance results, grid size 45
4.6 Plot of performance results, frame count 46
4.7 Illustration of plasma oscillation 47
4.8 Illustration of oscillation breakdown 48
4.9 Illustration of particle partitioning 56

Chapter 1

Introduction

Particle-in-Cell (PIC) method is a popular method that has been in use in plasma simulations since the mid 1950s. The names stems from modelling each particle (or in plasma simulations letting a ”superparticle” represent a bunch of charges) in a discrete ”cell”. Applications include charged toner particles in a copier, charged particles in aurora borealis (assuming also an electromagnetic field is added, and simulating beam dynamics in particle accelerators. While the scope and size of the simulations have increased over the years, the algorithm is largely the same. It is considered easy to grasp, with the electrostatic version based around solving a single partial differential equation (PDE), and otherwise updating particles using classical mechanics. Advances have been made, either focusing on supporting additional physical phenomena, reducing numerical error, or increasing performance and scope[1].

In electrostatic Particle-Mesh codes, which we will cover in this thesis, the contribution of each of the particles' charges in a given cell to the cell's grid points is calculated at a given time based on the particles' location. The resulting charge density distribution is then used to calculate the electric field based on the PDE resulting from Maxwell's equations. The particles are then ”pushed” by the field according to Newton's law of motion.

Until the past few decades 3D simulations were considered restricted to supercomputers, being too time- or storage intensive. As the algorithm lends itself well to parallel computing, it has seen many adaptations to various parallel architectures.

Since the early 2000s parallel processing have entered mainstream computing, with the majority of PCs using multicore processors, cell phones and tablets

running on multicore processors. Among the more interesting recent advances is the use of the originally entertainment-oriented graphics processing unit (GPU) in high performance computing, and as a component of supercomputers. While they started out as graphics accelerators for 3D games and modelling, the transition from specialized hardware through programmable shader pipelines to general purpose manycore processors have led to their adoption as a parallel computing accelerator of sorts. At a relatively low cost GPUs provide access to a large number of threads with a focus on high throughput.

For this project the goal is to implement a GPU-accelerated Particle-In-Cell code, extending the algorithm and 2D codes developed by my advisor A. Elster [2] to three dimensions. Her code was later adapted for a distributed system using MPI, and for a combination of MPI and OpenMP [3, 4]. The same code has served as the backbone for this project, and although written from scratch in CUDA, the underlying algorithm is the same.

Nvidia's CUDA platform and the CUDA C/C++ language is used for development, and the cuFFT library, which is part of the CUDA Toolkit, is used to implement a PDE solver. In order to achieve a stable simulation, a certain problem size is required, with the number of particles and cells increasing by several orders of magnitude for 3D compared to 2D. Because of the limited memory size available to GPUs, containing the problem might be difficult, and a secondary goal is therefore to implement support for multiple GPUs.

Part of the work done during this project is the extension from two to three dimensions, and in line with comments made by Meyer in [3], an attempt has been made to make the code readable and modular. It is hoped this will make it easier for others to understand, extend and improve upon the work.

The real challenge of this work has been to adapt the code for execution across multiple GPUs. This includes using the cuFFTXt library when multiple GPUs are present, handling boundary conditions and particle migrations.

In order to determine bottlenecks and potential for improvements, some profiling and performance tests have been performed for different configurations. The implementation was tested on a single NVIDIA Tesla K20, an NIVDIA GTX 980, and a system with two NVIDIA Tesla K20s. In addition, an OpenMP+FFTW variant was implemented for CPU as comparison. The major bottleneck has been found to be atomic operations, with memory performance as an important factor. Based on this, several ideas for improvement are suggested, with a focus on how to handle large numbers of particles.

Several attempts have also been made to achieve a stable plasma oscillation, with limited success. Oscillations break down after a variable number of oscillations, and further testing with parameters that satisfy constraints is encouraged.

Ideas for how to do this, is also included.

1.1 Outline

The contents of the rest of this thesis is outlined as follows:

Chapter two will give a brief introduction to Particle-in-Cell codes, deriving the general algorithm from the physics equations. The FFT as a PDE solver is given a brief treatment, and the history behind parallel computing and GPGPU is summarized. Some parallel performance measures and parallel architecture classifications are described. The CUDA GPGPU environment and its FFT library cuFFT are presented, describing the programming model and some performance factors. Finally some forays into parallel PIC codes are mentioned.

Chapter three details the new OpenMP +FFTW and CUDA-based GPU implementations, describing the development goals, overview of the simulation, and various changes made from previous work. Some development choices regarding parallelization issues are presented. Finally the particle tracing and CPU comparison are described.

Chapter four presents tests and benchmarks used to evaluate the implementation, and briefly discusses the results. Potential improvements and solutions for some issues are be presented.

Chapter six contains our conclusions, and provides several pointers for future work.

Chapter 2

Background Theory

This chapter will provide some background information on the Particle-In-Cell method of particle simulation, including how to derive the algorithm from the physical equations. The Fast Fourier transform will also be given a brief introduction, showing how it is used as a PDE solver.

A section is devoted to the history of parallel computing, and GPGPU in particular, and different parallel architectures and ways to measure parallel speedup will be described. Nvidia's CUDA library for general purpose computing on graphics processing units (GPGPU) will be introduced, explaining the programming model and some ways to achieve good performance. The CUDA FFT library cuFFT will be given a brief introduction as well. Finally a brief recap of the history of PIC codes, and some forays into parallelizing them will be mentioned.

2.1 Particle-In-Cell codes

The name "Particle-In-Cell" is said to originate with a research group at Los Alamos in the 1950s, from "investigations into the fluid nature of matter at high densities and extreme temperatures"[5, p314]. Since then the term has come to refer to particle simulations involving charged particles moving within some discrete grid ("cells"). Typical applications for PIC codes have been the study of plasma, originally as a way to visualize what could not be observed, and later to predict the actual behaviour of plasma, for example in beam dynamics.

The code examined in this work is known as an electrostatic Particle-In-Cell

code. The general algorithm is based on Dr. Anne C. Elster's doctoral thesis [2], although extended to three dimensions. See section 2.6 for examples of other works published on PIC codes.

The idea behind PIC codes is that the collection of charged particles create a charge density distribution ρ that sets up an electric field. Charged particles in an electric field are affected by an electric force, causing them to move around, thereby altering the field. The PIC simulation repeats this process for some small time step Δt and traces the particles' movements over time as output. The physics and mathematics behind the steps will be explained in more detail below, but first the particle and field models are examined.

2.1.1 Particles and Cells

The particles are represented as point charges with mass and velocity. While some PIC codes model collisions between particles(see [6]), a common approach is to assume particles with no volume, and the simulation concerned with the general behaviour of the plasma constituted by the particles rather than that of the individual particles. To achieve realistic values for the number of particles, total mass, and charge of the plasma without sacrificing performance, the particle objects of the simulation will often be a so-called "super-particle" with its charge and mass being some multiple of the particle type it models.

The cells in the name refers to the discretized field, represented in three dimensions by a grid of size $N_x \cdot N_y \cdot N_z$. This grid samples the charge density, electric potential and electric field at all its vertices, with greater accuracy for increasing N.

2.1.2 Charge density contribution from particles

All particles contribute charge to their neighbouring grid vertices, in proportion to the distance between them. For a three dimensional simulation trilinear interpolation is used, weighting the contribution according as seen in figure 2.1. The sum of a particle's contributions thus equal the particle's charge, conserving the total charge of the system. As an example the contribution of a particle with charge ρ_p to a neighbouring vertex $\rho_{i+1,j,k}$ is shown in equation 2.1.

$$\rho_{i+1,j,k} = \frac{\rho_p}{(h_x - a) \cdot b \cdot c} \tag{2.1}$$

h here represents the size of the cell, with h_x being its length along the x axis, found by dividing the length by the number of grid vertices. a, b and c

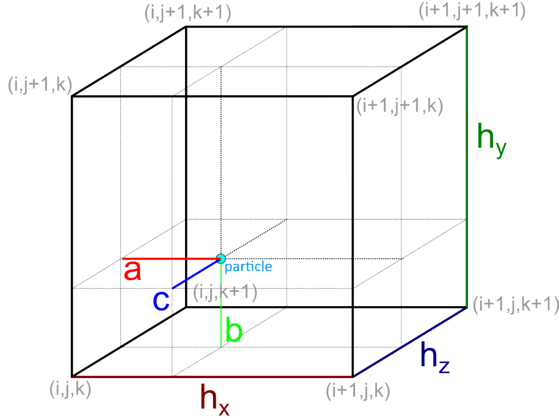

Figure 2.1: Trilinear interpolation. A particle (light blue) positioned at (a, b, c) within a cell. Grid vertices are indicated in grey coordinates.

describes the particle's position within the cell. a is calculated by subtracting the position of the cell wall from the particle's position.

$$h_x = L_x / N_x a \qquad = p_x - i \cdot h_x \tag{2.2}$$

2.1.3 Solving Poisson's equation for the electric potential

Poisson's equation $\nabla^2 u = f$ and the special case of Laplace's equation $\nabla^2 u = 0$ are arguably the most important elliptic partial differential equations for practical applications [7, p910]. They appear in fluid dynamics, astronomy, electromagnetism and electrostatics, mechanical engineering, and potential fields in general.

Equation 2.3 shows Poisson's equation for electrostatics, and relates the charge density ρ of the field to the electric potential Φ. For the derivation of the equation see [2, s3.2.1].

$$\nabla^2 \Phi_{x,y,z} = -\frac{\rho_{x,y,z}}{\epsilon_0} \tag{2.3}$$

To solve the equation for the potential Φ some numerical solver must be chosen. Under the assumption of periodic boundary conditions, a spectral solver based on the Fourier transform is chosen. The Fourier transform is known

to have efficient numeric implementations, and will give exact solutions for a periodic system.

In three dimensions the spatial derivative is $\nabla^2\Phi = \frac{\delta^2\Phi}{\delta x^2} + \frac{\delta^2\Phi}{\delta x^2} + \frac{\delta^2\Phi}{\delta x^2}$. The Fourier transform of a the derivative $\frac{\delta^2\Phi_{x,y,z}}{\delta x^2}$ is $k_x^2 \cdot \Phi^2_{k_x,k_y,k_z}$, which transforms equation 2.3 into 2.4, a second degree polynomial equation in place of a second order PDE.

$$(k_x^2 + k_y^2 + k_z^2) \cdot \hat{\Phi}_{k_x,k_y,k_z} = -\frac{\hat{\rho}_{k_x,k_y,k_z}}{\epsilon_0} \tag{2.4}$$

$$\hat{\Phi}_{k_x,k_y,k_z} = -\frac{\hat{\rho}_{k_x,k_y,k_z}}{(k_x^2 + k_y^2 + k_z^2) \cdot \epsilon_0} \tag{2.5}$$

This equation is then solved by dividing by k^2 and scaling by $\frac{1}{\epsilon_0}$. After performing the inverse transform and performing any necessary normalization, the field is solved for the electric potential.

2.1.4 Deriving the electric field from the potential

The electric field is related to the potential by equation 2.6, and in the discrete case can be derived using first order finite differences along each axis. The electric field at a vertex $E_{i,j,k}$ is then as shown in equations 2.7, 2.8, and 2.9, represented as a three component vector.

$$E = -\nabla\Phi \tag{2.6}$$

$$E_x[i,j,k] = \frac{\Phi[i-1,j,k] - \Phi[i+1,j,k]}{2 \cdot h_x} \tag{2.7}$$

$$E_y[i,j,k] = \frac{\Phi[i,j-1,k] - \Phi[i,j+1,k]}{2 \cdot h_y} \tag{2.8}$$

$$E_z[i,j,k] = \frac{\Phi[i,j,k-1] - \Phi[i,j,k+1]}{2 \cdot h_z} \tag{2.9}$$

2.1.5 Updating particle positions and velocities

When accelerating the particles only the electric forces acting on them are considered here. The gravitational forces between electrons are insignificant in

comparison[1].

The electric force affecting a point charge q at position p in an electric field E is $F_E = -q \cdot E(p)$. Dividing by the particle's mass yields the acceleration, and the particle can then be moved using the classical mechanics equations 2.10 and 2.11:

$$v_n = v_{n-1} + a_n * \Delta t \tag{2.10}$$
$$p_n = p_{n-1} + v_n * \Delta t \tag{2.11}$$

The argument could be made that the relativistic velocity equation should be used instead, but for simplicity the Newtonian mechanics have been used. The assumption is made that most velocities will be well below a significant fraction of the speed of light.

Like [2] and many other PIC code implementations [8] a leap-frog integration scheme is used to move the particles, with the velocity update lagging half a time step behind the position update (see equations 2.12 and 2.13). The leap-frog algorithm is more accurate than the simple Euler integration of equations 2.10 and 2.11, with a limited and self-cancelling error when $\omega \cdot \Delta t \leq 2$, where ω is the plasma frequency (see section 4.5.1 for more) [9].

$$v_{n+\frac{1}{2}} = v_{n-\frac{1}{2}} + a(p_{n-1}) * \Delta t \tag{2.12}$$
$$p_n = p_{n-1} + v_{n+\frac{1}{2}} * \Delta t \tag{2.13}$$

2.2 The Fast Fourier Transform solver

Under periodic boundary conditions a PDE can be solved using an FFT solver, which is both efficient and accurate. This section will recap the mathematics behind how the Fourier Transform can be used to solve a PDE, and give a brief introduction to the idea behind the Fast Fourier Transform Algorithm.

The Fourier Transform is shown in equation 2.14, with the inverse transform in 2.15. For a function of time $f(t)$ its transform is considered to be a function of

[1] The gravitational force between two electrons yields values on the scale of $F_G = G\frac{m_e^2}{r2} \approx \frac{5.538 \cdot 10^{-71} Nm^2}{r^2}$, while the electric force will be $F_E = k_e \frac{q_e^2}{r^2} \approx \frac{2.307 \cdot 10^{-28} Nm^2}{r^2}$. This means that the electric force between two electrons will be roughly $4.166 \cdot 10^{42}$ times stronger than the gravitational force regardless of the distance between them.

temporal frequency $\hat{f}(\omega)$. Similarly for a function of space $f(x, y)$ the transform $\hat{f}(\xi_x, \xi_y)$ represents the spatial frequency. For this reason the space described by the transformed variables ξ_x and ξ_y is often called the frequency domain.

$$\mathcal{F}\big(f(x)\big) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) \cdot e^{-i\xi x} dx = \hat{f}(\xi) \tag{2.14}$$

$$\mathcal{F}^{-1}\big(\hat{f}(\xi)\big) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \hat{f}(\xi) \cdot e^{i\xi x} d\xi = f(x) \tag{2.15}$$

2.2.1 Solving PDEs with the Fourier Transform

In order to find an expression for the Fourier transform of the derivative of a function f, we can derive the inverse transform from equation 2.15:

$$\frac{d}{dx}\big(f(x)\big) = \frac{d}{dx}\Big(\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \hat{f}(\xi) \cdot e^{i\xi x} d\xi\Big) \tag{2.16}$$

$$f'(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \hat{f}(\xi) \cdot \frac{d}{dx}\big(e^{i\xi x}\big) d\xi \tag{2.17}$$

$$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \hat{f}(\xi) \cdot i\xi e^{i\xi x} d\xi \tag{2.18}$$

$$= \mathcal{F}^{-1}\big(i\xi \hat{f}(\xi)\big) \tag{2.19}$$

and by Fourier transforming both sides

$$\mathcal{F}\big(f'(x)\big) = i\xi \hat{f}(\xi) \tag{2.20}$$

The process can be repeated to show that the general rule for the transform of n'th and second order derivatives are

$$\mathcal{F}\big(f^{(n)}(x)\big) = (i\xi)^n \cdot \hat{f}(\xi) \tag{2.21}$$

$$\mathcal{F}\big(f''(x)\big) = -\xi^2 \cdot \hat{f}(\xi) \tag{2.22}$$

For a second order PDE such as Poisson's equation the result of Fourier transformation is a much more easily solvable polynomial equation, shown in equation 2.23. Solving for Φ then consists of dividing the right hand side by $-\xi^2$ and performing the inverse Fourier Transform.

$$\mathcal{F}\big(\nabla^2\Phi\big) = -\xi^2\hat{\Phi} = \frac{-\hat{\rho}}{\epsilon_0} \tag{2.23}$$

Equation 2.24 generalizes this to multiple dimensions, and for three dimensions in particular it can be seen that a division of ρ by $\epsilon_0 \cdot (\xi_x^2 + \xi_y^2 + \xi_z^2)$ is necessary.

$$\Phi = \frac{-\hat{\rho}}{\epsilon_0 \cdot \sum_n \xi_n^2} \tag{2.24}$$

2.2.2 The FFT algorithm

The Discrete Fourier Transform is a discrete analogue of the Fourier Transform from equation 2.14. The one dimensional version is seen in equation 2.25, where ω is the Nth root of unity. The DFT will not be described in detail here, see [7] for a thorough explanation. Since for every output ξ_k each input x_n needs to be evaluated, the DFT has a computational complexity of $O(N^2)$.

$$\xi_k = \sum_{n=0}^{N-1} x_n \cdot \omega^{kn}, k \in 0..N-1 \tag{2.25}$$

The Fast Fourier Transform is the name of an optimized algorithm with a complexity of $O(N\log N)$. Different FFT algorithms have been invented for various purposes, but they are most commonly implemented as a recursive algorithm. The most common is the algorithm described by J.W.Cooley and John Tukey in [10], known as the Cooley-Tukey FFT algorithm.

Dividing the transform into two transforms of size ${}^{N}\!/_{2}$ is called a radix-2 FFT, while the more general case of dividing the transform by the factors of $N \neq 2^n$ is called a mixed-radix FFT. A more detailed explanation can be found in [10] or [7], but the general idea is to split the computation to avoid recomputing the same values due to symmetry and periodicity. By exploiting these properties of the transform, the complexity $O(N\log N)$ is achieved.

2.3 Parallel computing and GPGPU

Parallel computing may refer to any form of processing in which multiple operations are performed concurrently. This can range from vectorizing single instructions to running ten thousands of threads on thousands of nodes on a

supercomputer. In recent years, GPGPU has become a popular form of parallelism.

2.3.1 History and motivation for parallel computing

When Gordon E. Moore discussed the possibility of ”Cramming More Components onto Integrated Circuits” in 1968[11], besides reasoning why and how the number of transistors on a chip would increase over time, the difficulties associated with it such as power dissipation received only a small mention. Moore expected that integrated circuits would allow for sufficient cooling, and that the shrinking size of the components would allow the circuit to run faster under the same power.

While this is true, and Moore’s law regarding increasing component density on circuit boards has held true for longer than the ten year period he estimated, the increase in processing power slowed during the late 1990s and came to a halt in 2004. A visible sign of this was when the successor to Intel’s previous Pentium 4 processor was cancelled in favour of a transition to a dual core architecture[12].

Important reasons for this happening were the fact that memory speeds were lagging significantly behind processor speeds already, to the point where further increase in CPU clock frequency would result in it running idle while waiting for data from memory, with the cache hierarchy no longer able to hide the latency ([13] provides a more detailed explanation, and considers whether the wall was hit when expected). Memory *bandwidth* was increasing by at least the square of the *latency*.

Memory latency had been partially hidden by increasing the instruction level parallelism, but attempting to extract further parallelism would yield diminishing returns[14].

Increasing clock frequency and transistor density both contributed to CPU power consumption increasing rapidly, to the point where traditional cooling techniques would suffice no longer[14]. Counteracting this power increase by reducing voltage would no longer be an option either, due to issues such as subthreshold leaking and electromigration becoming prominent with insufficient voltage. More transistors could be placed on the chip than could be powered on.

Rather than designing increasingly complex and massive single core CPUs the trend from the early 2000s became multicore processors, with multiple independent processing cores on a single chip. By adding additional cores the theoretical processing power of the CPU could be further increased, assuming all cores could be kept busy.

Multithreading has therefore become an important aspect of modern applications programming, running independent operations in threads that execute concurrently on multiple cores. Examples of applications that commonly are multithreaded are operating systems, video and image processing applications, computer games, and other applications with heavy real time data processing requirements.

Even before consumer computers made the jump to multicore, academia and industry have made use of parallel platforms such as computer clusters and supercomputers. The different varieties of parallel architectures will be described below, but for this project a particular brand of parallel computing is of interest, GPGPU.

2.3.2 General Purpose computing on Graphics Processing Units

General-purpose computing on graphics processing units (GPGPU) is a fairly recent trend in high performance and parallel computing, where programs are accelerated by running heavy computation on a *graphics processing unit* (GPU) rather than the more traditional CPU. Characterized by a large number of slower cheaper compute cores, modern graphics cards can deliver massive parallelism at a comparatively low cost.

The graphics pipeline on early GPUs contained specialized hardware for common graphics processing tasks. Traditionally data was passed from the CPU to the GPU where it was processed in several steps, rendered, and output to some display device. Over time the specialized hardware was replaced by a more flexible programmable pipeline, with support for programmable shaders working on common graphical primitives.

Around 2001 these programmable shaders enabled researchers to perform the first experiments with GPU computing, by defining the data and problems in terms of graphics primitives and shader programs. In order to alleviate some of the inconvenience of this convoluted programming model GPU-specific languages and libraries were written, effectively providing an abstraction over the shader language. Examples of such languages were Sh and Cg [15]. As mentioned by [16] implementing simulations using these shader languages placed severe limits on what was achievable.

As the industry as a whole began shifting its focus towards multithreading and parallel computing GPUs were also becoming more powerful and easier to program. By the late 2000s GPUs had shifted from being graphics pipelines to

generic stream processors, with increasingly high parallelism and general programmability. In recent years GPGPU has been found to perform outstandingly for certain types of problems, with potentially much higher efficiency than comparable CPU implementations[17]. GPGPU is still recognized as a field with higher demands in terms of programming effort, and many problems will be better suited to the more general domain of the CPU.

2.3.3 Parallel performance measures and potential speedup

When the performance of a program or system is measured the goal is often to find the improvement over a previous candidate. For parallel systems in particular, the speedup compared to the serial execution is often of interest, or the performance relative to a different parallel configuration. For parallel systems the speedup as a function of the number of processing cores is typically of interest. Two different definitions of parallel execution time has led to two well known yet different rules for parallel speedup.

In both of these the speedup of a program a with runtime T_a over a program b is defined as $S_{a,b} = \frac{T_b}{T_a}$, the number of times a can be executed while b is running. In addition to the number of processing cores n another important parameter is what fraction of the program is inherently serial, and which thus yields no speedup from adding additional cores. One the best known rules in this regard is Amdahl's law.

Amdahl's law defines the speedup as the serial execution time divided by the time it would take a parallel processor with n cores to do the same work. Assuming a serial fraction P_s and a parallel fraction P_p, with execution time for a single core T_1:

$$S_{n,1} = \frac{T_1}{T_n} = \frac{T_1}{T_1 \cdot (P_s + \frac{P_p}{n})} = \frac{1}{P_s + \frac{P_p}{n}} (= \frac{n}{P_s \cdot n + P_p}) \tag{2.26}$$

Letting $n \to \infty$ it is apparent that the speedup is limited by the serial fraction, with the maximal speedup proportional to $^1/_{P_s}$. See [18] for the original argument. What is known as Gustafson's law was proposed as an answer to Amdahl's law in [19].

Gustafson's law instead defines the speedup as the time it would take a single core to perform the work the parallel processor is capable of doing divided by the parallel processing time. The difference here is essentially that Gustafson

assumes the parallel part of the program can be scaled according to the number of processors rather than remaining fixed. By performing more work in a single execution the overhead from the serial fraction will no longer dominate as the number of cores increase.

$$S_n = \frac{T_1}{T_n} = \frac{P_s + P_p \cdot n}{P_s + P_p} = \frac{P_s + P_p + P_p \cdot (n-1)}{1} = 1 + P_p \cdot (n-1) \quad (2.27)$$

2.3.4 Architectures

There are several options for classifying parallel computer architectures, based on programming model, hardware, communication patterns and so on. In this section the scheme known as Flynn's taxonomy will be introduced first, followed by a look at some different memory architecture classes.

Flynn's taxonomy

Since first proposed by Michael J. Flynn in 1972, the scheme known as Flynn's Taxonomy has been used to classify parallel architectures based on how instructions and data are shared between processors[20].

	Single data	*Multiple data*
Single instruction	SISD	SIMD
Multiple instruction	MISD	MIMD

SISD The single instruction, single data architecture represents a traditional sequential computer, any single-core CPU.

SIMD Single instruction, multiple data-architectures contain several processors performing the same operations on different data. Array processors are an important example here, and in particular GPUs. The same instructions are run on several processors in parallel, with each operating on its own data set. Typically consisting of an array of smaller processors that each operate on a small number of elements from a larger shared array, devices with these architectures are often called array processors.

MISD An uncommon architecture, the MISD does not utilize parallelism to improve performance, as it has the same throughput as a single code (SIMD)

processor. Instead of multiple cores processing more data, this case runs different instruction on the same data stream in parallel. This can be useful for systems with low fault tolerance, where processing a set of data on different systems (with different instructions) can reveal errors.

MIMD Both multicore CPUs and most current supercomputers fit this category, with multiple cores running instructions independently, on separate data.

Other Variant extensions have been proposed later, including the SPMD (single program, multiple data) and MPMD (Multiple program, multiple data). SPMD is a subcategory of MIMD where a single program is run on multiple processors in parallel. Instances of the program on different processors are not guaranteed to run concurrently however, and SPMD is therefore more flexible than SIMD.

Memory architectures

Parallel systems are also often categorized based on their memory architecture. As an example, while a multicore CPU and a supercomputer both fit the MIMD label, they will typically have a vastly different configuration, in particular with regard to memory spaces, and of course number of cores.

Shared memory Most multicore personal computers and workstations employ a shared memory architecture (SMA), where threads in programs share a memory space. This means that threads within a process can read and write the same data. Communication between threads is therefore fast.

While a shared memory may be easier to envision and program for, it becomes increasingly complex when more cores are added. Memory access arbitration will become harder, and core will have to wait longer for their request to return. Race conditions are more prevalent, but all data is available to each core.

Distributed memory Supercomputers and computer clusters in general often consist of a large number of nodes. Each node may consist of several cores and accelerators, and has a private memory. Nodes in a distributed system may themselves be shared-memory systems.

By distributing the memory, the number of cores competing for memory access is limited to the number of cores per node, thus improving scalability

drastically. Doubling the number of nodes in the system will have no effect on the local memory latency of any core.

Memory access types Parallel architectures can also differ in how memory accesses are performed, notably whether they have a uniform or non-uniform memory access (NUMA) from different processors. Numerous variations exist in regard to configuration of memory and cache handling.

2.4 CUDA - Compute Unified Device Architecture

The Compute Unified Device Architecture (CUDA) is a proprietary GPGPU framework developed by Nvidia. While CUDA is a relatively young technology, first appearing in 2006, it has already seen widespread use in industrial and scientific computing. This section will give an introduction to the CUDA programming model, device architecture and mention some key factors affecting the performance of CUDA applications.

While CUDA applications can be developed using various languages and APIs this section will mainly concern the CUDA C/C++ language and the CUDA Runtime API. See section 2.4.3 for other ways to accelerate applications using the CUDA framework. The CUDA model is explained in detail in [21].

2.4.1 The CUDA Programming Model

The CUDA architecture employs SPMD parallelism, executing the same program across multiple *Streaming Multiprocessors*(SMs), typically targeting different elements of the same data stream. Compared to a typical CPU architecture this leads to increased usage of hardware for actual computing rather than control logic and cache handling.

The multiprocessors themselves operate in a scheme called *SIMT* (Single Instruction, Multiple Threads), where hundreds of threads execute the same instruction concurrently. Threads are organized in blocks, and executed together on a an SM. Warps of 32 threads share control logic and therefore also execute all instructions together. See figure 2.2 for an illustration.

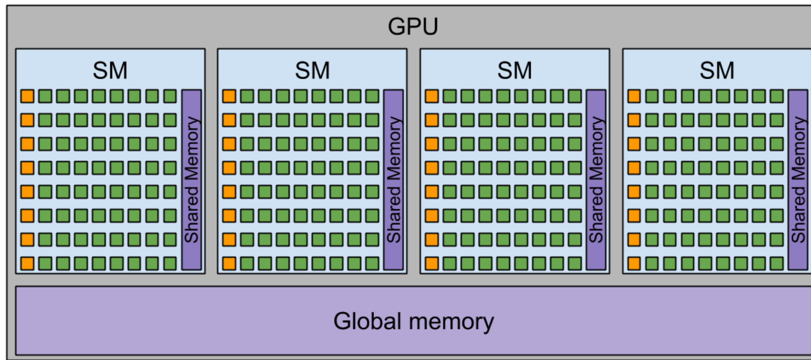

Figure 2.2: Conceptual illustration of the CUDA architecture, with a warp size of 8 and 4 SMs for clarity. Threads (green) in a warp share control logic (orange). All threads in a block execute on the same SM and have access to the same shared memory. All blocks on a GPU have access to global memory.

Device code

A CUDA C/C++ application typically consists of host code serving as a master thread which launches device code in functions called kernels. The host performs any necessary set-up such as allocating memory and transferring data to device memory, and then launches the device kernels with a specified number of threads. The kernels then process the data in some way, typically operating on one data element per thread, using relatively simple code with limited branching and looping. After the kernels have executed, the host then transfers the result back to host memory, frees resources and ends execution. See section A.1 for a simple CUDA example.

Threads

As mentioned, the kernels are called while specifying the number of threads. Organized as a number of threads in a block, in a grid of blocks, the thread hierarchy in CUDA somewhat reflects the hardware implementation. A relic of its origin in graphics programming CUDA organizes threads and blocks using three dimensional coordinates, allowing a natural mapping from thread index

to coordinates.

Each thread has a private set of registers it operates on, while threads are executed in sets of 32 called *warps*. Each warp executes in lock-step with all threads in a warp executing all instructions (SIMD). It is thus of interest to limit divergent branching within warps, as the whole warp must execute code for all branches.

A thread block consists of up to 1024 threads, ideally a multiple of 32 since they are executes as warps of 32. Threads in a block are executed at the same time within a single SM, and can therefore synchronize with each other, and have access to the same *shared memory*.

Blocks are organized within a grid, containing all threads belonging to the same kernel launch. Blocks are distributed among SMs, execute in batches, and therefore share neither on-chip resources nor any form of synchronization other than kernel launch and end time. All blocks in a grid will execute on the same device however, and therefore have shared access to the global memory available on that device.

CUDA streams represent another level of hierarchy, allowing multiple kernel executions (grids) in separate streams. While all kernels launched in a stream are executed sequentially, kernels in separate streams may execute in any order relative to each other, or even concurrently. When using CUDA with more than one GPU, kernels will be issued to separate streams, as they are tied to a device.

Memory

The different levels of the memory hierarchy correspond to the thread hierarchy. Registers are the fastest memory resource available to threads. They come in a severely limited quantity, 255 per thread for current architectures, but have no latency. Registers can only be read by the owner (with some exceptions, see [22]) and share the lifetime of the thread.

Shared memory is a slower and larger memory that resides on-chip, and is shared between all threads in a block. Access latency is on the order of 10 cycles, and size on the order of 64KB per SM. Where registers are private, shared memory is typically used to communicate between threads in a block, or to store data common to several threads, reducing the load on the global memory by preventing unnecessary memory accesses. Shared memory has the lifetime of its block, so to keep any data it must be written to global memory.

The global memory is large, typically 2 to 24 GB depending on the device, but typically has a latency of several hundred cycles. All threads on a device can access global memory, and it is persistent, allowing communication between

blocks and kernels. While the global memory in general is accessible by threads on the device it is contained, threads from different devices may access memory on other devices under certain conditions [21, s3.2.6.4].

Atomic operations are a common way of avoiding write conflicts when multiple threads in a shared memory context need to write to the same address. By guaranteeing only one thread can update an address at a time, no writes are lost to overwriting. While double precision atomic operations are limited to the atomic compare-and-swap, others can be implemented using the atomicCAS, as seen in the example below (based on one given in [21]).

```
__device__ double atomicAdd(double *addr, double value) {
  unsigned long long int* ull_addr = (unsigned long long int*)addr;
  unsigned long long int updated = *ull_addr, compare;

  do {
    compare = updated;
    updated = atomicCAS(ull_addr, compare,
      __double_as_longlong(value + __longlong_as_double(compare)));
  } while (compare != updated);
  return __longlong_as_double(updated);
}
```

This example also demonstrates why the atomic operations are slow; the loop runs until the value is successfully swapped. When a large number of threads attempt to write to the same address the operation can take a long time to complete. Because the CUDA architecture executes warps together, a single thread stuck in the while loop will stall the entire warp.

2.4.2 Performance factors

The long latency between global memory requests and the availability of the requested data is somewhat hidden by the CUDA architecture. Whenever a warp is stalled waiting for data, the SM switches context to a warp ready to execute. The context for all executing warps are stored throughout their lifetime, so that this switching has a minimal performance impact, while serving to keep the SM busy, thus hiding the memory latency.

A big factor in this is *occupancy*. If no other warp is available while the first is waiting the SM will be idle until a warp is ready to continue. For this reason a CUDA device will have better performance if the total number of warps is higher than the number which can execute concurrently, the SMs need idle warps to switch to in order to achieve maximum throughput. The number of threads that

can be stored in an SM at any time depends on the architecture, model, register usage and thread dimensionality. While many of these vary greatly from case to case, a rule of thumb is to have 128 to 512 threads per block (see table 5.1 of [23] for a comparison of utilization for various architectures).

Overlapping kernel execution with host-device memory transfers is another way to prevent downtime. Assuming the data used by the kernel and memory transfer is separate, issuing these to separate threads will allow them to execute concurrently. For this reason many CUDA API functions, especially those dealing with device memory, have synchronous and asynchronous (non-blocking) versions. Of course, this also allows the host thread to perform other tasks while the kernel and memory transfer are executing.

While the techniques above are useful to hide latency, accessing global memory correctly will yield even better results. The architecture is optimized for SIMD-style computing, where access to consecutive addresses by threads in a warp will be significantly faster than random accesses. Because memory transactions are 32-, 64- and 128-byte wide segments, memory accesses that are aligned to these sizes will yield better performance. Note that the smallest transaction is for the equivalent of eight 32-bit integers, meaning a memory access for a smaller size will waste throughput.

Structuring memory accesses so that consecutive threads access consecutive addresses is called a *coalesced* memory access. Coalesced memory accesses yield better performance, particularly for data that is aligned in memory[21, p. 5.3.2].

As mentioned above, shared memory is commonly used to reduce the number of global memory accesses. An ideal example is for image processing, where by copying the thread block's segment of an image into shared memory, all threads in the block has fast access to the whole segment.

2.4.3 Alternative CUDA interfaces

Support for OpenCL, an open source alternative to CUDA, is built on top of the low-level CUDA architecture on Nvidia GPUs, providing a different interface to much of the same functionality[21]. OpenACC is a compiler directive-based programming standard, aiming to provide simple parallelization of accelerators, similar to the OpenMP standard for multicore parallelization. See Nvidia's CUDA Language solutions[2] page for more information.

[2] https://developer.nvidia.com/language-solutions

2.5 cuFFT - the NVIDIA CUDA Fast Fourier Transform library

Nvidia's Fast Fourier Transform library cuFFT is available as part of the CUDA development kit. cuFFTW is a second library which supplies similar functionality, but through an interface that mimics that of the FFTW[3] library to ease conversion of code.

For input sizes consisting of powers of small primes the Cooley-Tukey algorithm is used to generate efficient transforms on up to 512 million elements. Transforms of up to three dimensions are supported, for complex and real valued data. While the FFT exhibits a degree of parallelism by allowing sub-problems to be computed in parallel, cuFFT's forte is in batched transforms, where multiple data sets are transformed via a single call, minimizing overhead.

2.5.1 cuFFTXt - Multi-GPU support in cuFFT

Recent versions of the library have improved and extended the multi-GPU capabilities of the library as well. As of CUDA 7.0 the multi-GPU functionality (referred to as cuFFTXt) supports execution across two or four GPUs, with certain additional restrictions placed on combinations of transform and input sizes. In particular, multidimensional transforms have a minimum size of 64 along the x and y dimensions, and only support in-place complex-to-complex transforms.

Another major difference is that while the single-GPU transform can operate on any array matching the transform size, for multiple GPUs a special data structure must be allocated that contains pointers to data on the various GPUs, array sizes and information about the library. The result of a single 2D or 3D transform will also be divided along the y axis instead of along the x axis. See section A.2 for examples on use.

2.6 Related work

The following are examples of PIC codes being adapted to different parallel architectures.

[3] The Fastest Fourier Transform in the West

Parallelization Issues and Particle-In-Cell Codes [2], Dr. Anne C. Elster's Ph.d. thesis from 1994, is a detailed account of a parallel electrostatic PIC code running on a SMP architecture. Beyond giving an elaborate introduction to the physics behind the algorithm, load balancing across processors and techniques to achieve this are discussed.

***Emerging Technologies Project: Cluster Technologies* PIC codes: Eulerian data Partitioning** [3] by Jan C. Meyer describes an attempt to adapt the algorithm from [2] for a message-passing architecture using MPI. The performance on a supercomputer and a PC cluster was compared.

Parallelizing Particle-In-Cell Codes with OpenMP and MPI [4] is Nils Magnus Larsgård's master's thesis from 2007, which also builds upon the same algorithm. The topic here is parallelization using both OpenMP and MPI on cluster computers. In particular, the goal was to find which configuration of OpenMP and MPI lead to the best performance.

A General Concurrent Algorithm for Plasma Particle-in-Cell Simulation Codes [24] is the name of a paper on a one dimensional particle in cell code written for parallelization on a distributed memory architecture, and accounts for the details on splitting the problem into subproblems in an efficient manner, and the distribution of these subproblems across the processors.

Particle-in-Cell Charged-Particle Simulations, Plus Monte Carlo Collisions With Neutral Atoms, PIC-MCC [6] gives a detailed review of the history of PIC codes from ca 1950 to 1990. Attempts to add particle collisions to the simulation are mentioned, and the combination of PIC code with Monte Carlo collisions is the main focus.

Dynamic Load Balancing for a 2D Concurrent Plasma PIC Code [25] extends [24] to 2D with a focus on load balancing. Load balancing was found to be most useful when the particle update step dominated the solver step. Since their FFT-based solver exhibited far less efficient parallelism they found that load balancing was counter productive compared to static partitioning.

Adaptable Particle-in-Cell algorithms for graphical processing units [26] implemented a simple 2D PIC code with the goal of running it on different platforms.

Particle-in-cell plasma simulation on heterogeneous cluster systems [1] developed a 3D PIC plasma simulation, and using OpenMP and OpenCL supports execution on either CPUs or GPUs in a cluster.

Chapter 3

Implementation

The Particle-In-Cell method implementation will be described in this chapter. We will also briefly discuss issues with dividing the implementation across multiple GPUs, and the implementation and possible alternatives for the particle tracer. Finally a simple OpenMP/FFTW port of the implementation issues described.

3.1 Goals for the implementation

The following is a summary of the goals and motivations behind some important decisions taken during implementation.

Particle simulation on a graphics card

The primary goal is to implement a functioning Particle-In-Cell code in CUDA, capable of simulating the expected behaviour of some system. The target simulation is plasma oscillation, as described by [2].

In addition, the implementation should support acceleration by more than one GPU, to investigate what benefits this could yield. The CUDA Fast Fourier Transform library cuFFT is used for the PDE solver, as it has support for multiple GPUs, and should integrate better with a CUDA application than most other libraries.

Problem size

An important question to be answered is whether the hardware and memory of a GPU is sufficient to execute and contain a PIC code of sufficient size and accuracy.

According to [1] some problems for PIC codes require a number of particles on the order of 10^9 and a grid of some 10^8 elements. A quick calculation shows that 10^9 3D particles using double precision data types require at least 44.7 GB of storage alone. With the largest available memory of even high-end Nvidia Tesla GPUs at around 12 or 24 GB a simulation of those sizes are still out of reach. See section 4.5.1 for more on the scale of the simulation.

For performance reasons, particles and simulation grid are all kept in GPU memory when possible. A larger number of particles could be supported by keeping the majority in host memory, transferring them to the GPU and processing them in turn. The simulation grid is still restricted to the size of the GPU memory however, unless a more low level implementation of the solver is used. To make use of the cuFFT libraries without too much overhead, and particularly because of the high cost assiciated with data transfers between host and GPU memory, the problem size will be restricted to whatever can fit in GPU memory.

To increase the available memory, and thus potentially the scale of the simulation, support for additional GPUs will be added. Whether the additional available memory can be of use is an issue that will be investigated.

Performance

While optimal performance is not a primary goal for this implementation, decent performance relative to a comparable multithreaded C++ implementation is expected. In addition, the performance of the multi-GPU configuration compared to the single-GPU one is of interest. Some overhead due to communication between the GPUs is expected, but whether there is enough of a benefit from splitting the FFT computation across them to outweigh this will be of interest.

Modularity and readability

Besides goals for increasing problem size and decent performance, a tertiary goal has been to write readable and modular code, with more loosely coupled code. Ideally this should result in a more readable code, while also being easier to extend with further functionality, such as adding other solvers, or target applications.

As mentioned by [3, s6.2] the use of explicit data structures rather than arrays of primitives helps increase the modularity of the program. While data structures were used to represent both particles and grid elements in [3], the field has in this implementation been left as separate arrays for charge density and the electric field. The advantage of this is that the electric potential can be calculated in-place by the solver, reducing the memory footprint. In addition the cuFFT library does not support strided transforms across multiple GPUS as of yet.

The implementation also aims to be operating system agnostic, tested on both Windows and Ubuntu. The cuFFT library still has some restrictions on GPU configuration based on OS as of CUDA 7.0.

3.2 Simulation overview

The structure of the program is as follows:

1. Setup
2. Distribute charge from particles to grid
3. Solve charge distribution for electric potential
4. Calculate electric field in grid
5. Update particles based on electric forces
6. Repeat from 2 until finished
7. Cleanup

Each of these steps are covered below.

3.2.1 Setup

Setup consists of reading the simulation configuration file `settings.cfg`, setting up the solver, allocating and initializing particles, and opening the trace file.

The configuration file uses the INI format[1], and the PropertyTree class from the Boost library[2] is used to read the file. Other values that depends on those read from the file are then computed and stored in a Cfg object.

The number of GPUs on the system is also read and stored in this object, but can be overridden by setting the multi option in the config file to 0, overriding the detected number of GPUs and setting it to 1. Based on this value the appropriate solver is then set up, as cuFFTXt employs a different interface than the standard cuFFT library, and requires different initialization. The FFT plan is created, and both the grid and work area for cuFFT is allocated.

Particles are initialized using a function and kernel particular to the application. For plasma oscillation the particles' positions should be regularly spaced in two planes. The spacing in either direction as well as the number of rows and columns in each plane is calculated, and a kernel creates a particles at each position, with no initial velocity. The distribution of particles along either axis of the plane is attempted to match the grid resolution in that direction.

A particle tracer object is also created and a trace file opened, see 3.4 for details. The tracer object uses its own set of CUDA streams to overlap its data transfers with kernel computation.

3.2.2 Simulation loop

The simulation loop is run a number of times specified in the config file. While previous work defined the simulation time as a parameter, the runtime is instead controlled here, with a simulation time equal to the number of iterations times the length of the time step.

An iteration of the simulation loop consists of the steps below; the charge distribution kernel, the forward cuFFT call, the solver kernel, the inverse transform, the electric field kernel, the particle update kernel, and depending on the configuration and current iteration particles might be traced.

In addition, boundary exchange and particle migration is handled for a multi-GPUs system. Because both data types and cuFFT calls are different, two different simulation loop functions have been implemented, simplifying the single-GPU version.

[1] INIfileonWikipedia. https://en.wikipedia.org/wiki/INI_file
[2] http://www.boost.org/

Charge distribution kernel

The kernel is called for each particle and opens by finding the neighbouring grid vertices based on current position, trilinearly interpolating as shown (2.1).

Because the equations for the weights and array offsets for each neighbour are the same for this and the particle update kernel the code for finding these has been separated into a helper function.

To update each neighbour while avoiding overwriting values from other particles, atomic updates of grid values are used. This severely limits the performance of the kernel, and alternatives have been investigated.

For the multi-GPU case, particles that are positioned between vertices on two GPUs (see figure 3.1) write their contribution to a temporary boundary array. After the kernel has completed this boundary is then forwarded to each appropriate device, and the values are added to the vertices they represent.

Field solver

The solver step consists of a forward FFT, a solver kernel, and an inverse transform. The solver kernel divides each grid value by $\epsilon_0 \cdot (k_x^2 \cdot k_y^2 \cdot k_z^2)$. In addition, it normalizes the result by $N_x \cdot N_y \cdot N_z$ as required by the transform. k_x can be calculated from the x axis coordinate i as $k_x = i \cdot \frac{2\pi}{l_x}$ where l_x is the metric length of the simulation space in x. Since only the coordinates varies for different threads, the rest is precomputed and stored in the config object for the kernels.

Electric field

The electric field is derived according to the equations in section 2.1.4. For each grid element the difference between neighbours' potential divided by the distance between them is calculated along all axes.

Boundary potential values are stored in the same boundary arrays, and distributed to the other devices for multi-GPU configurations. Afterwards the boundary array is populated with the electric field values along the boundary.

Updating the particles

Called per particle, the electric field values of neighbour vertices are interpolated in the same way as for the charge distribution kernel. Scaling by ${}^{\rho}/_{m}$ yields the particle's acceleration. The velocity and then position are updated accordingly. Particles that go out of bounds are "wrapped" so they re-enter the simulation

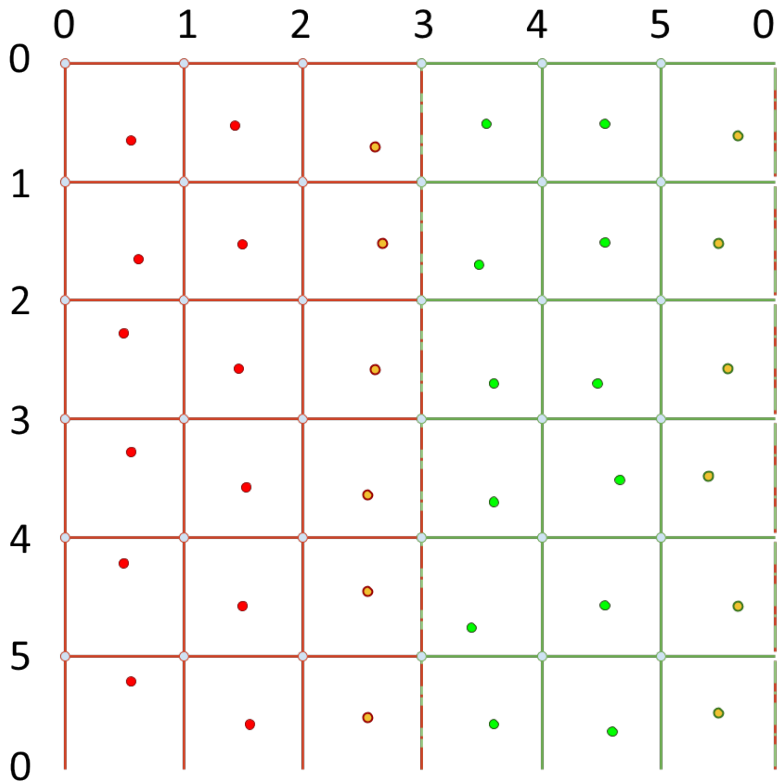

Figure 3.1: Illustration of particles in a grid split between two GPUs. Column 0 and 3 are duplicated into temporary buffers, indicated by the dotted line. The yellow particles lie on the boundaries between GPUs and need to access these buffers to write their contributions.

space on the other side, keeping with the periodicity of the simulation. As for charge distribution, particles in space between nodes from different GPUs read their electric field values from a temporary buffer.

3.2.3 Cleanup

After the simulation is completed the final particle traces are written to file and the trace file is closed. The FFT plan is destroyed, and other allocated memory is freed.

3.3 Parallelization issues

This section will mention some of the important parallelization issues faced during implementation of the algorithm. Notably, two different kinds of parallelism occur, shared memory parallelism within a GPU, and distributed memory parallelism when more GPUs are involved. Each of these have their own issues.

3.3.1 Atomic operations

For simplicity and readability the implementation currently uses the slow atomic operations. Their performance decreases with the number of conflicts, but require less overhead in design and maintenance during development. The issue is discussed further, with suggestions for improvements, in section 4.6.

3.3.2 Overlapping memory transfers with kernels

Because of how the CUDA architecture is designed, with high throughput-high latency memory buses, it is advantageous to overlap data transfers with kernel execution when possible. By using the asynchronous variant of the memory management interface, and executing kernels in a separate CUDA stream, the host will not have to wait for either to finish before issuing new requests.

In order to benefit from this, the data dependencies within the program must allow overlapping. The data transfers involved in the simulation loop are either part of the particle tracer described below, or boundary distribution between kernels. The latter typically leaves little room for overlap, since the transferred data is written by the preceding kernel and read by the succeeding one, see table 3.1 for the lifetime of each data array.

Particle tracing however depends on data written by the particle update kernel, and must complete before the next particle update. The data transfer can therefore be overlapped with both the other kernels and also the writing of previous particle traces to file.

functions \ arrays	particle	ρ, Φ	E	boundary
chargeDistribution	read	update		write
boundary, rho		update		read
solverKernel		update		
boundary, phi		read		write
electricfield		read	write	read
boundary, E			read	write
particleUpdate	update		read	read
particleExchange	update			
trace particles	read			

Table 3.1: Illustration of the lifetime of each data array.

The best candidate for overlapping these transfers are likely the solver, done automatically by the cuFFT library, and based on results from code profiling (see 4.3) it appears this is well utilized.

3.3.3 cuFFTXt limitations

While there are many options with regard to particle migration and load balancing in general, the cuFFTXt library sets restrictions. In particular, the grid is statically distributed among the GPUs involved, which limits the possibilities for dynamic load balancing.

3.3.4 Particle migration

When particles on one GPU enter a part of the grid located on another, the particle must migrate from the memory of the first to that of the second in order to be able to access the correct neighbouring grid elements.

While it is easy to determine whether a migration is necessary (using a simple check of the position coordinate against the domain of the GPUs), the actual transfer of particle data is more complicated. First consider the case where all particles migrate from one GPU to another. Either the correct amount of memory must be allocated in-loop to be able to house the data, or room for all particles must be allocated on all GPUs (more advanced schemes will be mentioned in the following section). Assuming then that all GPUs have room for all particles (which yields ${}^{1}/_{n_{GPU}}$ occupancy of the allocated memory), how should the migration occur?

The implementation currently uses a simple brute force solution, where all particles are copied to the host memory, sorted according to correct device, and copied back to the appropriate device memory. This is a simple solution, but considering that plasma oscillation along one axis, with the grid divided along another ideally has *no* particles migrating, it is wasteful to transfer all particles every iteration. Some suggestions for improving this are mentioned in section 4.7

3.4 Particle tracing

Without some sort of output, the simulation would serve no other purpose than as a work load to benchmark a system by. Because the output can be used to verify simulation accuracy, and because the process of writing data to file has an impact on performance itself, some support for writing particle data to file is needed. The simulation is built around plasma oscillation, with the particles' positions as the main output.

3.4.1 Tracer implementation

A goal for the particle tracer was that it should not distract from the algorithm, ideally handling the tracing process behind the scenes. To limit the impact particle tracing has on the general performance, the maximum number of particles traced can be specified. This allows the visualization script mentioned below to animate the movements of the particles without choking on the sheer amount of data. For the same purpose, a frame option is also given in the configuration settings, which limits how many times the tracer is called.

Writing data to file

Two sets of particle storage arrays plus an output array are allocated by the tracer. By using the CUDA host memory allocation function the resulting array is pinned in host memory, facilitating faster transfers between it and the GPU it is associated with. Using an asynchronous transfer of data from the GPU to one of the associated arrays, data already copied to the other can be written to file. The idea is to overlap the transfer of frame n with the writing of frame $n-1$ to file. By handling this in a separate thread the performance could be increased further, with the simulation running while particles are written.

An important assumption here is that the number of particles written is limited, as pinned host memory remains in RAM at all times. Since the total

size of the particle array easily reach several GB in size this will be a problem. If an application needs to output *all* particles in a simulation with millions of particles, a different solution must be sought.

Output format

The output data itself has a simple structure. Position values are formatted to a precision of 12 digits and separated by commas, particles are separated by a semicolon, and frames separated by a newline. While this does not conform to a typical CSV format, it makes the process of separating the data easier.

3.4.2 Visualization script

For demonstration and troubleshooting purposes a relatively simple python script uses the numpy and matplotlib packages to animate the movements of the particles over the course of the simulation. The animation can either be shown directly or stored as a movie file. For performance reasons the number of particles and frames should be limited working with this script, and to visualize millions of particles over longer stretches of time a more sophisticated is in order.

Some options that seems interesting at this point would be animating the data in openGL or similar library, with hardware support. Performance should be greater, but this also allows investigations into interactive visualization or even real-time animation of particle movements.

3.5 The CPU benchmark comparison

In order to compare performance on one or more GPUs with a CPU implementation, the code was ported relatively straightforward to a OpenMP + FFTW application. The CPU version uses the same configuration file and produces output in the same format.

3.5.1 OpenMP

OpenMP is a multiprocessing API for a shared memory system such as a multicore CPU. The transition from CUDA to OpenMP was surprisingly simple. The general idea was to replace kernel calls with simple function calls, and in the kernels add a for loop over the relevant arrays. OpenMP then parallelizes the loop when "`#pragma omp parallel for`" is placed on the line preceding it. While

a more sophisticated and optimized parallelizations scheme can no doubt be found, the result serves the purpose as a CPU comparison that makes use of all available cores to accelerate the computation.

3.5.2 FFTW

FFTW is one of the most efficient and popular FFT libraries available. As cuFFT was designed to feel familiar to FFTW developers[27] it was less surprising that a transition the other way would also be simple. The process mostly consisted of replacing the various cuFFT API calls with the FFTW equivalent.

Chapter 4

Results and discussion

Different tests are run in order to evaluate the implementation. For a measure of scalability, the division of labour among kernels for increasing numbers of particles is looked at in section 4.3, using profiling tools.

Section 4.4 provides the timing results for various problem sizes and configurations, as well as each configuration's performance on a large sale simulation.

Section 4.5.1 briefly explains plasma oscillations, and the motivation behind using it as a test for numerical stability. Some of the constraints regulating possibly parameters will be presented, along with a brief discussion of the results from testing. The two-stream instability test is also briefly introduced.

Some potential solutions and ideas for improvement will then be presented in the last few sections.

4.1 Hardware

The GPU used for testing is the Tesla K20 GPU Accelerator [28]. The test machine is equipped with two of these GPUs, an Intel i7 3770k CPU[29], and 32GB RAM from Kingston in four DIMM's clocked at 1333MHz. Each Tesla GPU is equipped with 5GB GDDR5 RAM, supporting PCI Express 2.

Although the CPU in question, the Intel i7 3770k, cost roughly $^1/_{9.5}$ of what the Tesla K20 did at launch[30, 29], the comparison will hopefully show the different tendencies of GPUs and CPUs. To make up for the difference in cost, some of the tests will be run on the more recent consumer GPU GTX980, with a cost of around 1.6 of the CPUs cost[31].

This GPU has a much lower double precision performance, being aimed at 3D graphics and driving a desktop screen, and therefore targets the same market segment as the 3770k (enthusiast desktops). With only 4GB of GDDR5 RAM it has a lower capacity, but the memory interface is clocked higher, with roughly double the throughput of the Tesla K20[32], and supports PCI Express 3.0. The 980 test machine equipped with an Intel i5 3470 CPU, which should perform comparably to the 3770K in a single-threaded application.

4.2 Goals

When testing the implementation the performance of a single GPU versus multiple GPUs is a main focus. To provide some context the CPU implementation will be run with the same parameters. The goal is to determine some qualities of the implementations:

- performance - how fast a given simulation can be run
- scalability - how variance in the problem size affects the performance
- accuracy - whether the implementation accurately simulates expected behaviour for the particles

4.3 Profiling

To determine the impact each step of the algorithm has on performance for various configurations, a profiling tool can be used to measure the execution time spent on each function. Nvidia provides a visual profiler with its Toolkit that also provides information on occupancy and overlapping memory transfers [33].

For traditional CPU code a profiler tool such as gprof[1], or tools in the valgrind[2] suite may be used. Due to its simplicity gprof was chosen for this task. While it gives only an estimate of the time spent in different functions, and has certain issues that need to be taken into consideration[3], the tool is accurate enough to provide the info sought here.

[1] https://sourceware.org/binutils/docs/gprof/

[2] http://valgrind.org/

[3] gprof assumes that a function's execution time is unrelated to where it was called, and divides its measured execution time among the callers proportionally to the number of times it was called by each.

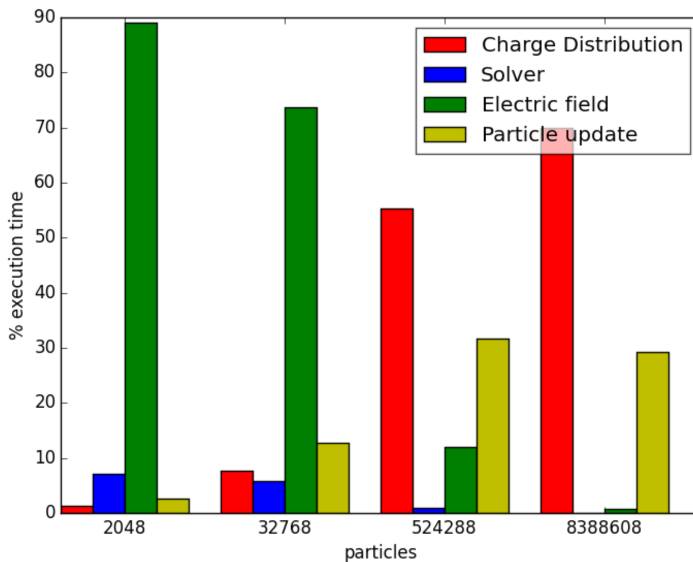

Figure 4.1: Percentage of execution time per kernel for different particle counts, on one GPU.

It is worth noting that code run with a profiler will never yield optimal performance, as profiling code is either injected into the program, or the program is run on emulated hardware. The timing results for profiled code will therefore not be accurate with regard to the production code, but may still be useful in determining bottlenecks or suboptimal code.

The simulation settings for profiling has 100 iterations of the simulation loop with 10 frames for the particle tracer. The code will be profiled for a grid of 64^3, while the number of particles will be varied between tests, in order to see the effect it has on execution time of the different kernels.

4.3.1 Single GPU

The profiling results for a single GPU are illustrated in figure 4.1. It shows that although both charge distribution and particle updates have many uncoalesced memory accesses in the same pattern, and the particle update kernel does more

computational work, the atomic operations involved with charge distribution means it is still slower. The number of conflicting atomic operations should increase with the number of particles per cell, as more particles will write their contribution to the same grid vertex at any time. This could be the cause of the charge distribution kernel ”stealing” a fraction of the time from particle updates in the last case. While both kernels have an increased work load as a result of the increased number of particles, the charge distribution kernel is also affected by the resulting increase in particle density. When the number of particles is lower, it can be seen that the particle updates consume more execution time, as might be expected.

That the kernel deriving the electric field from the potential dominates the solver might not be surprising given that the solver kernel operates on a single data point, while the former reads six neighbours. However, the timing result for the solver also includes the execution time of the cuFFT calls involved, implying that it is very efficient. A simulation on a 64^3 grid would require on the order of $2.6 \cdot 10^6$ particles, which would likely yield results somewhere between those for 524288 and 8388608 particles. Even with the particle kernels dominating the grid kernels, between 0.7% and 12.0% is still enough that an attempt to optimize the electric field derivation using shared memory to reduce the number of loads, especially since these loads are already coalesced, and neighbouring values typically correspond to threads in the same block.

4.3.2 Two GPUs

The results in figure 4.2 are generally very similar to those for the single GPU profile. The solver step appears to consistently consume more of the execution time. Because the solver kernel does no work depending on the number of devices involved this is likely due to cuFFT doing more overhead when working across devices, performing more expensive peer-to-peer memory transfers.

Without looking at actual timing data, it would be hard to say whether this decreases the performance, but the only case in which the solver's fraction of execution time would increase while it performs better is if all the other functions' performance improves even more. This is very unlikely to be the case, and more likely the multi-GPU solver performs worse. A possible cause for this is that the cost of peer-to-peer data transfers outweighs the benefits of the additional compute power available for this grid size.

Based on the profiler output it was also apparent that for higher particle counts, particle exchange appeared to consume an increasing amount of time compared to the rest simulation loop. For the last row, the time spent on

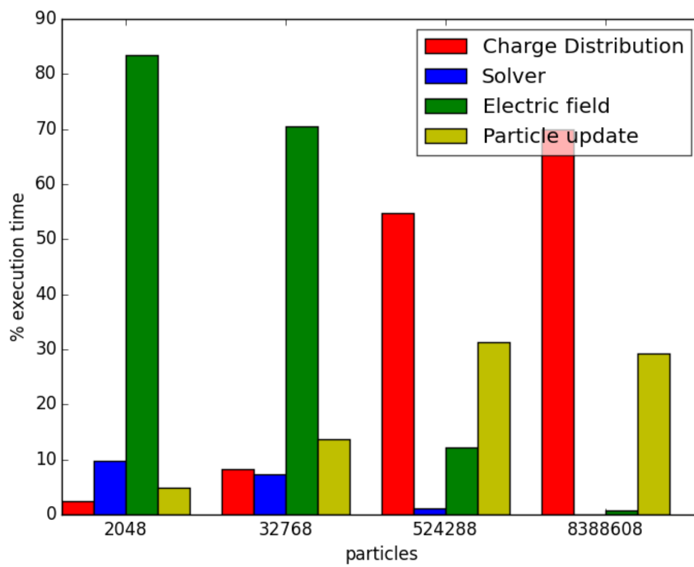

Figure 4.2: Percentage of execution time per kernel for different particle counts, on two GPUs.

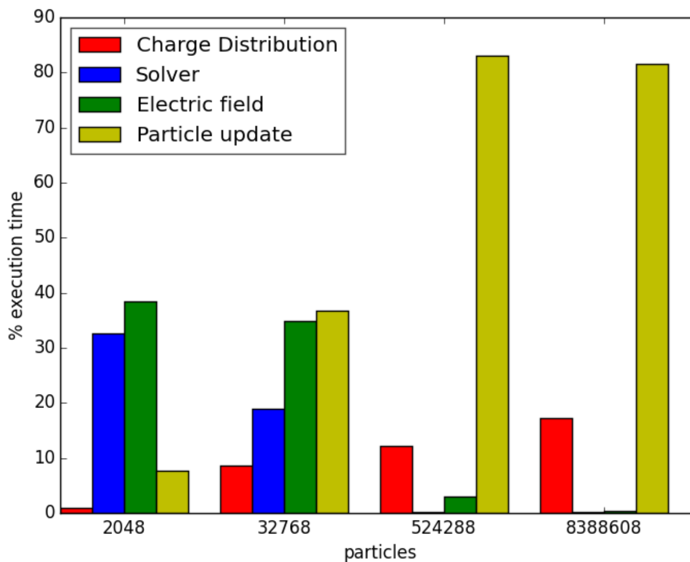

Figure 4.3: Percentage of execution time per kernel for different particle counts, on CPU.

particle exchange was nearing $^1/_3$ of the simulation loop execution time. Most of this time was shown to be the synchronous data transfers between device and host, and by making these asynchronous, the time spent on particle exchange was cut to less than $^1/_4$.

4.3.3 CPU

While the steps of the algorithm handling particles still dominate for a higher particle density, the trends shown in figure 4.3 paint a different picture than for the GPUs. Perhaps most striking is the fact that the particle update and not the charge distribution dominates. Because a multithreaded program running on a multicore CPU runs significantly fewer threads at a time compared to a GPU the effect of the atomic memory accesses could be expected to have less of an impact, with fewer concurrent conflicts.

Another difference here is that the solver step consumes a great deal more of

the execution time in all cases, compared to the electric field kernel. It could be that cuFFT is more efficient than FFTW relative to the rest of the program, or that the electric field kernels neighbouring values are stored in cache, yielding the performance improvement that might be offered by using shared memory in CUDA.

4.4 Simulation performance

As a more accurate measure of actual performance than code profiling, the different configurations will be timed during normal optimized execution. The goal for these benchmarks is to find a valid comparison between the single-GPU, double-GPU and CPU implementations.

A measure of scalability can be found by measuring execution time for each hardware configuration for various problem sizes. With 1000 iterations of the simulation loop the implementation will be timed for different numbers of particles, grid length in the z dimension, and number of particle tracing frames. The results are shown in the tables below. Default parameters are a grid size of 64^3, particle count of 32768 and 10 frames traced out of 1000.

4.4.1 Particle count

The results for variations in particle count are illustrated in figure 4.4. From 2048 to 32758 particles the GPUs show little difference in execution time, which might be explained by the high number of threads needed for the GPUs to reach peak utilization. For high particle counts however, the execution time increases drastically, suggesting that both number and density of particles is affecting performance.

The CPU execution time in comparison scales linearly, showing no signs of congestion.

Because of its poor double precision performance, being primarily a consumer product not targeted at scientific computing, the GTX 980 might have been expected to be outperformed by the far costlier and more industrially oriented Tesla card. However, as we see from figure 4.4, it performs just as well, and even better in some cases, than either Tesla card configuration. An educated guess says this might be due to the GTX 980's bandwidth being 224 GB/s to the Tesla K20's 104 GB/s, and that the implementation is memory bound rather than waiting for computation.

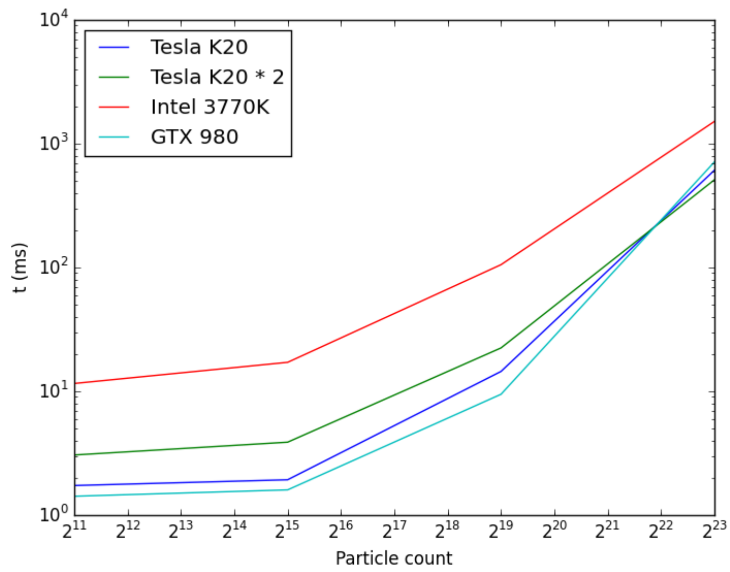

Figure 4.4: Performance as a function of the particle count.

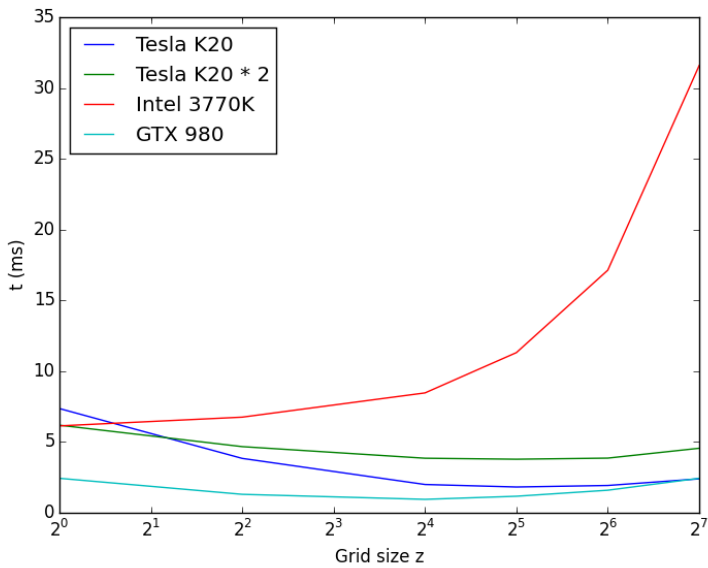

Figure 4.5: Performance as a function of grid size along the z dimension.

It is interesting that the GTX 980, which is otherwise fastest, comes in last for the highest particle count tested. The double Tesla K20 configuration, while otherwise slowest of the GPUs, is fastest here. It might be that the benefit of servicing the particles through two memory interfaces (with effectively double the memory throughput) outweighs the cost of transferring the particles to the host for sorting, when the number of particles is so high. Why the GTX 980 performs worse than the single Tesla K20 card is more uncertain, especially since its better memory specification might be the reason it performs well otherwise.

4.4.2 Grid z dimension

For the CPU configuration the increase in runtime is more or less linearly proportional to the increase in grid size (note that the x axis in figure 4.5 uses a logarithmic scale). For the GPU configurations however, the runtime *decreases* for increasing grid size, up to a point. A good candidate for the cause of this is that by increasing the grid size, the particle density is decreased, thereby

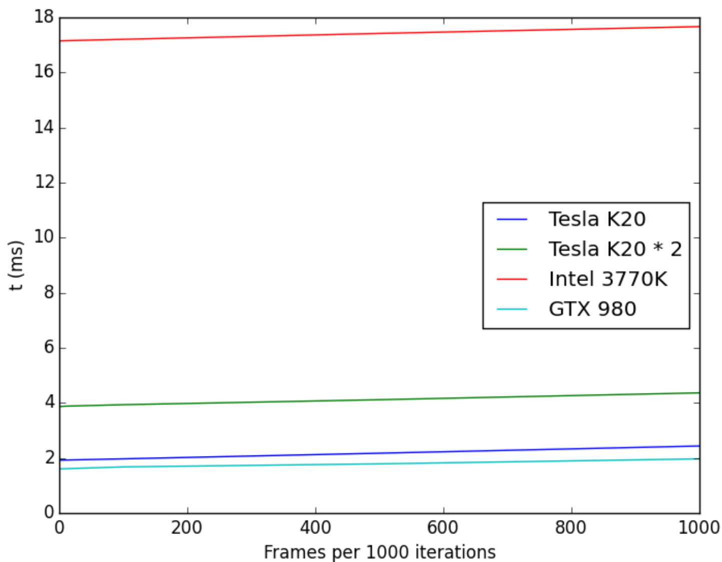

Figure 4.6: Effect of frame rate on performance.

improving the performance of the atomic operations in the charge density kernel. For $z = 128$ the runtime again increases, as the cost of computing a larger transform and more threads appear to outweigh the benefits of reducing the particle density further.

Again the GTX 980 outperforms the Tesla K20 for all the lower values, showing a halved execution time for the first three columns.

4.4.3 Frame count for 1000 iterations

The results from figure 4.6 show that for all configurations, the increase in execution time is directly proportional to the number of frames traced. Unsurprisingly, given that the amount of work done per trace is independent of the number of traces, so long as the size of the trace data is small enough to complete transferring before the next is ready.

Note that the particle tracer uses roughly the same amount of time for each configuration ($\sim$ 0.5ms for 1000 frames), which might mean that most of the

time is spent writing to file rather than transferring data.

4.5 Stability testing

In addition to any potential performance increase, the implementation should display sufficient numerical accuracy to provide a useful result from the simulation. In order to confirm numerical correctness the behaviour of the electrons over time must be observed. Rather than evaluating the numerical values and comparing them to some precomputed truth, the general behaviour of the particles in an application will be tested, looking at its ability to simulate physical phenomena. The primary focus during development has been to simulate plasma oscillations.

4.5.1 Plasma oscillations

To explain plasma oscillations, first consider two electrons in a 1D vacuum. Because they have equal charge they will repel each other, with a force inversely proportional to the distance between them. Under our assumptions of periodicity in the simulation space however, the two electrons will soon approach each other "from the other side", and therefore slow down and eventually reverse their direction again. This behaviour will repeat, with both electrons oscillating back and forth.

The PIC code implemented ought to be able to simulate this behaviour when certain constraints on the input parameters are satisfied. Elster[2] successfully demonstrated plasma oscillations as part of her work, with lines of particles in a 2D space. Figure 4.7 illustrates the behaviour in 2D. For 3D there will be two sheets of particles in a 3D space.

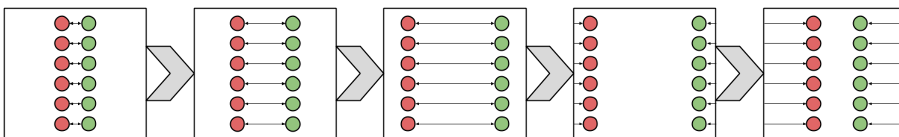

Figure 4.7: An illustration of how two lines of particles, red and green, will repel each other and oscillate back and forth. The sequence illustrated will repeat until some instability occurs.

It is important that the particles are equally spaced. Because the space is

periodic along all axes, a particles will see its sheet as infinite in all directions. In a regularly spaced grid the particle will observe equal electric forces from all directions within the plane, summing to zero. With all movement restricted to that caused forces outside the particle plane, all particles should oscillate back and forth as for the single electron case mentioned above.

Because even double precision floating point numbers have a limit to their precision, the oscillation cannot continue indefinitely in practice. Even so, under sufficient conditions the implementation should be able to produce several periods of the expected behaviour. Figure 4.8 illustrates an oscillation breaking down, where the small displacement of some particles leads to further instability and chaos.

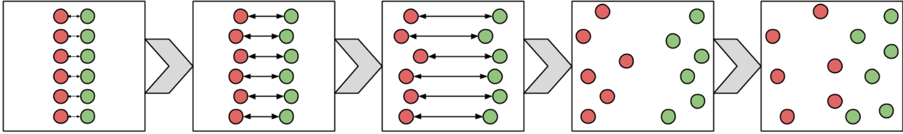

Figure 4.8: An illustration of oscillation breakdown.

The plasma frequency The plasma frequency ω_p appears in the constraints below. It is defined as

$$\omega_p = \sqrt{\frac{N_p q^2}{m\epsilon_0}} \tag{4.1}$$

where N_p is the number of particles with, mass m and charge q[34, 35].

Constraints

Dr. Anne's thesis [2] describes plasma oscillation test for a two dimensional system, with two infinite electron rods oscillating in place of the sheets. As part of the testing, this 2D plasma oscillation will be attempted in the 3D code. She used a 1024 by 1024 grid, with 400000 particles. The time step required for the simulation to run stable is there calculated based on constraints given by [36] for a 1D simulation:

1. $H \leq \lambda_D$
2. $\omega_p \cdot \Delta T \ll 2$

3. $L \gg \lambda_D$

4. $N_p \lambda_d \gg L$

The first constraint declares that the resolution of the simulation grid should be sufficiently small, the distance between two grid vertices should be no larger than the Debye length, the distance a particle's influence reaches [37].

Constraint two requires the plasma frequency multiplied by the time step to be distinctly smaller than 2[9].

The third constraint requires the length of the simulation space to be far longer than the Debye length.

Finally, number four requires a number of particles per Debye length.

These requirements may be generalized to more dimensions by stating them for each dimension, as done by Elster [2] for two dimensions and here for three. However, according to Peratt [5] that when upgrading the number of dimensions for a simulation, each additional dimension requires an increase in the number of particles by around two orders of magnitude. Bastrakov [1] mentions that there are problems known requiring $\sim 10^9$ particles on a $\sim 10^8$ cell grid, but a configuration on that scale would require around 50GB storage for the data alone, and is thus beyond the reach of a simulation running on GPUs with 5GB RAM each.

Comparing the values for the 2D simulation in [2] and the 3D simulation from [1], there are $\sim 10^5$ particles to $\sim 10^6$ cells in 2D, and $\sim 10^9$ particles to $\sim 10^8$ cells in 3D, an increase on the order of two for particles relative to cells. If the same ratio is used to find the largest problem that fits in the memory of a Tesla K20, the results is around $\sim 10^{7.95}$ particles and $\sim 10^{7.22}$ cells, such as 89458688 particles on a $128^2 \cdot 1024$ or 256^3 grid.

A 3D electromagnetic PIC code also implemented in CUDA by [38] mentions that a significant restriction is the Courant restriction, $c \cdot \Delta t < {}^{h}/_{\sqrt{3}}$. This equates to $\Delta t < h \cdot 1.93 \cdot 10^{-9}$, meaning a larger grid requires a smaller time step to achieve numerical stability.

Results

While the simulation has been run with many different configurations, no absolute success has been achieved as of yet, with the oscillation breaking down at some point. The likelihood of a stable oscillation increases for larger numbers of particles and grid elements, and smaller time steps. For a larger problem each iteration takes longer to compute, and when decreasing the time step a larger number of iterations is needed to progress the simulation. This of course means

that a higher resolution simulation will take longer to finish, and because its success or failure is hard to determine while it is running, each attempt can take more than 24 hours from start to end.

To determine success or sufficient parameters two approaches are considered. The first is to include a real time view of the result, indicating success or failure while running. Although more useful, this still requires human interaction during simulation, as automatic verification of the output seems hard to achieve. The other option is to draw up a new set of constraints for a 3D PIC code, based on physics rather than an interpretation and superposition of simplified constraints for one dimension. A deeper understanding of the implications of grid resolution, particle count, time step, and simulation expanse is recommended.

Whether the absence of a stable oscillation thus far is simply a result of testing with insufficient parameters or a sign of the GPU's capacity being too small to provide a sufficient grid resolution and particle number is uncertain, but it should be noted that the substantial runtime required to achieve each result means a limited number of combinations have been tested so far, and many of them with the focus of detecting more severe issues. Further tests with increased time resolution in particular is therefore suggested, as the results have been promising, with up to five periods of expected behaviour.

Further testing

While the 3D oscillation is of interest itself, running a 2D plasma oscillation along each axis would be interesting as well, especially with multiple GPUs, to see whether any axis is favoured. Performance should be best so long as the grid is split among GPUs across the bands, and few particles move across device boundaries, but oscillation along all axes should be stable.

Because particle number appears to severely impact execution time per iteration, and cell count places restrictions on the length of the time step, running tests with smaller problem that satisfy the constraints might be prudent.

4.5.2 Two-stream instabilities

In addition to plasma oscillation, Elster used a two-stream instability test to verify numerical stability. The concept behind the test and suggested test conditions will be outline below, although the tests were never run for the current work.

The particles are uniformly distributed throughout the simulation space, or plane in 2D, and are split in two sets with opposite initial velocity. The two streams of moving charges will be unstable, as the charges repel each other, and when some charge slow a bit it will amplify the effect, as the charge density increases. As the amount of particle bunching increases, so does the effect it has on incoming particles, eventually providing sufficient electrical force to invert the velocity of incoming particles. When plotting velocity against coordinate of the axis the particles move along, a characteristic ”eye” will appear [4].

The appearance of these eyes within some time would serve as a confirmation of numerical stability. For further confirmation, this test should be performed with streams directed along each axis in turn. After implementing a non-naive particle migration scheme one would expect performance to take a hit with streams running along the x axis. By running this test one could confirm whether the division and boundary handling affects stability or not. Even with worse performance due to particle migration it should yield the same output as for another axis.

4.6 Atomic operations

Based on the timing results from the previous chapter, it appears that the atomic operations from distributing the charge to the grid vertices is a bottleneck, at least when the particle density is high. For applications such as plasma oscillation most particles are confined within a small area of the grid, with most particles writing to a small subset of the grid vertices.

Nvidia's Kepler architecture significantly improved the performance of atomic operations on global memory[39], while Maxwell improved upon atomic shared memory operations[40, s1.4.3.3]. Indeed, by first writing to shared memory and then from the partially accumulated value to global memory one could improve the performance somewhat. A prerequisite for this improving the performance is however, that threads in a block largely access a limited number of addresses, and that accesses to an address is limited to a low number of blocks. The result would be that most conflicting atomic operations would be to fast shared memory, while accesses to slow global memory would succeed more often in comparison.

By sorting the particles into some order, assumptions could be made regarding the number of grid vertices accessed by the threads in a block, and the

[4] An example of such an eye can be seen appearing in this video: `https://www.youtube.com/watch?v=__7GQS15IdE`. Link confirmed to be available on 2015-07-17

probability of reducing duplicate accesses would be lower than for a block handling particles at random locations. While the atomic updates to global memory perform worse for higher particles densities, the benefit of first accessing shared memory will be higher for increasing density, as fewer global memory accesses are needed.

For low particle densities the number of different addresses per block would be higher, and if all particles in a block end up writing to different elements, they might not all fit in the shared memory of a block (depending on the block size). In such a case there would be no or less benefit in reducing the writes in shared memory. Some scheme will need to be developed in order to decide which thread reads a given value into shared memory and writes it to global memory, thread structure and particle sorting.

4.7 Particle exchange

The computational cost of the naive particle exchange is the cost of transferring all particles from all devices to host memory, plus one host to host copy per particle. Using synchronous data transfers, the transfer time will increase with additional GPUs, while the host operations are unaffected. Using asynchronous transfers, all devices can transmit particles concurrently, with an increasing number of GPUs only reducing the average number of particles per GPU.

In order to facilitate asynchronous transfers however, the active host memory must be pinned. Because the particle count for each device is unknown ahead of time, and to avoid reallocating the host arrays for each exchange, two arrays of size $n_{particles}$ will remain allocated per GPU during the simulation. When the number of particles reaches 10^8 this means roughly 9GB of pinned host memory *per GPU*!

It is apparent that this is not a scalable solution, as for just two GPUs the particle exchange alone will consume 18GB of host memory at all times. Even with the availability and relatively low cost of memory seen today, a more sophisticated solution is needed. It is important to note that the arrays in question are only used as temporary storage for sorting the arrays according to device. In the typical scenario, most particles will remain on the current device at the end of each iteration, with only a few moving across the border to a neighbour. Performing the expensive procedure above even when *no* particles require transfer seems especially wasteful, and the wish is therefore to transfer only those particles which are located on the wrong device.

By partially sorting the particles on each GPU according to which GPU

they should reside on, the number of transferred particles will likely be cut drastically. The time spent in the particle exchange would then be variable between iterations as a variable amount of particles need transfer, but even the worst case of all particles moving should yield no worse result than the naive variety. The latency of a peer-to-peer device is lower than using the host as a temporary storage, but may still be significant even for a low number of particles. The peer-to-peer transfers can be executed asynchronously without allocating a huge pinned host array however, and the slow sorting on the host is avoided.

The sorting algorithm should be one suited for sorting large arrays in-place, optimized for input where only a few elements need to be swapped. For the exchange, the only information of interest is whether a particle is on the correct device or not, two particles position relative to each other is not of interest.

4.8 Particle partitioning and load balancing

Even with a better particle exchange scheme, the added GPU memory is poorly utilized, as all GPUs allocate memory for all particles. The only space benefit of adding an additional GPU is then to split the grid across them, and with the particle count dominating the grid size, this is a poor comfort.

In order to reduce the particle storage overhead some sort of load balancing is needed in order to partition the particles and grid among GPUs. Ideally the device particle arrays are each allocated to fit $N_{particles}/N_{GPUs}$ particles, allowing full utilization of an additional GPUs memory. To achieve this particles would be sorted along the axis on which the grid is divided between GPUs, and divided into groups with the same memory footprint. This sorting would need to sort all particles relative to each other, unlike that of the particle exchange scheme mentioned, but like that of the proposed improvement for atomic operations.

Splitting the particles evenly among the GPUs is a good start, but because the distribution of particles along the axis is possibly uneven, some particles might not have access to write to the grid vertices they distribute charge to. As mentioned cuFFTXt requires a certain division of the grid among GPUs, but this can be solved by expanding the existing potential boundary and temporary charge storages. Particles write charge to the grid when possible, and otherwise to the boundary array, which is then transferred to the GPU containing the corresponding grid data, and similarly for the transfer of either potential or electric field values in return. Figure 4.9 illustrates the extended borders (dotted

red) when particles are become distributed.

The additional memory required for these boundary arrays factor into the footprint of the division, since the more the particle boundaries are skewed from the grid boundaries, the larger these boundary arrays become. The ideal division of particles therefore depend on the actual distribution of particles along the axis, and the calculation of such a division is left for future work.

Particle partitioning for 2D PIC codes on a multicore architecture is discussed in more detail by Elster[2]. Some of the schemes discussed, such as fixed processor partitioning and partial sorting, are possible, and very similar to the variant implemented or discussed above. The best solution from that work however, a double pointer scheme, is less suited for GPUs. The currently implemented sorting on the CPU is similar, but less elegant, and partial sorting on the GPU making use of the GPUs' massive parallelism seems likely to yield better results.

4.9 Variant

A completely different approach would be to split work between the GPU and CPU. The per-particle steps involve rather unordered memory accesses, with conflicts between threads. Grid kernels on the other hand are regular, ordered, and relatively independent. Recognizing this, and that particles generally have a larger memory footprint than their grid, handling and storing the particles on the host, and the grid on the GPUs, could potentially allow for better performance while increasing problem size further.

The algorithm would then be to write charge densities to an array, transfer it to device memory, solving for potential and deriving the electric field, then transferring it back to the host and updating particles there. The performance hit from transferring the grid to and from device memory will of course impact performance to some degree. There is also the potential to utilize the hardware in question better, by having the host do particle tracing while the GPU is computing the electric field, and letting the GPU render trace data in real time, if possible.

Such a solution could also scale well for additional CPUs, in a shared memory computer such as those produced by NUMASCALE[5], one of which was originally a target hardware for this project. Such a system consists of a large number of cores, with a large shared memory, and possibly one or more GPUs. While cuFFTXt supports a maximum of four GPUs as of CUDA 7.0, One could

[5] https://www.numascale.com/

perform the transform using a lower level approach, wither by combining multiple cuFFT calls and distributing intermediate results manually, or implementing a transform specific to the target application. Another alternative is to use another language such as OpenCL for which there exists similar FFT libraries.

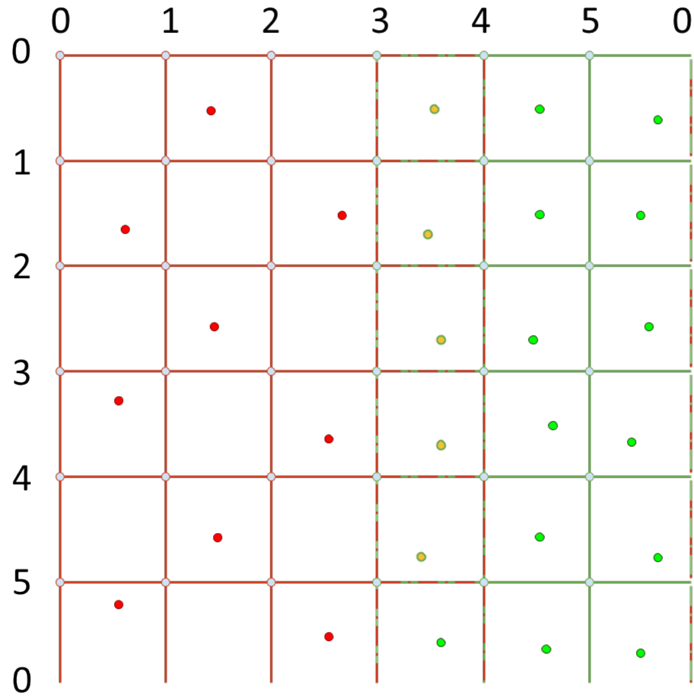

Figure 4.9: Illustration of particle distribution versus from grid division. Grid vertices in column 0 are duplicated in temporary buffers on GPU 2 (green), while vertices from both columns 3 and 4 need temporary buffers on GPU 1 (red). This is because the yellow particles, while in simulation space belonging to GPU 2, are stored and processed on GPU 1 due to the partitioning.

Chapter 5

Conclusion

PIC (Particle-in-Cell) codes are often used in plasma simulations to model the movement of charged particles over time. Although easy to grasp and based on solving a relatively small set of equations, they are known to be very compute-intensive. Given the recent use of GPUs as HPC accelerators, this thesis investigated how PIC codes could be ported to one or more GPUs.

In this thesis an algorithm for a 2D electrostatic PIC code was extended to 3D and implemented on Nvidia's CUDA platform, using their cuFFT Fast Fourier Transform library. The implementation was then adapted to make use of multiple GPUs, and ported to OpenMP + FFTW to serve as a benchmark. The implementations were profiled and performance tested to determine bottlenecks and potential for improvement.

A number of issues of varying levels of importance have been uncovered. For the ratio of particles to grid cells described in works such as [1], a dominating factor in determining performance is the particle density. Based on the results gathered, and knowledge of the implementation of the atomic operations in CUDA, it seems clear that a high density of particles leads to poor performance as many particle-threads attempt to update the same vertex. Some candidate solutions have been presented, though further investigation seems warranted. Notably, the CPU implementation was less affected by this issue, having fewer contending concurrent threads.

Specific to the multi-GPU implementation, a more sophisticated particle exchange scheme is needed, and a combination with some sort of load balancing seems viable. The naive variant currently in use gives a choice between a longer runtime, or significantly higher host memory consumption. A proposed solution

where particles are sorted on the device, and then only those particles necessary being transferred seems a good candidate.

Because the grid, and thereby the simulation space, is statically divided among GPUs, it is currently possible that all particles reside on one GPU. This leads to inefficient use of the added memory from additional GPUs(as all GPUs must fit all particles), and might lead to poor processor utilization as well, if only one GPU handles most particles. A load balancing scheme that divides the simulation space between particles based on the memory footprint of each partition is proposed. The particle sorting involved would also serve in one proposed scheme to optimize atomic operations.

5.1 Suggestions for future work

Although the PIC code implementation has not yet been able to provide a stable plasma oscillation, that might be a matter of experimenting further in order to find satisfactory parameters, or there might be an undetected bug in the implementation. In either case, the work done should provide a decent foundation for further extension.

Suggestion for future work include working on optimizing the GPU implementation further, especially atomic operations and particle exchange are good candidates, with the schemes proposed in the previous chapter as a good starting point; sorting of particles using shared memory to accumulate atomic updates within thread blocks, and exchanging only out-of-bounds particles by sorting them on the GPU. A comparison with a cluster implementation would also be in order, looking at performance and capacity versus cost and energy efficiency.

One of the initial goals for this project, implementation on a shared memory cluster computer, would be interesting as well. A division of work between CPU and GPU such as the one described in section 4.9 might be well suited.

Real time visualization of trace data could be useful to receive immediate feedback on the success of a simulation, but would depend on what performance could be achieved with various optimizations.

Bibliography

[1] S. Bastrakov et al. “Particle-in-cell plasma simulation on heterogeneous cluster systems”. In: *Journal of Computational Science* 3.6 (2012). DOI: 10.1016/j.jocs.2012.08.012.

[2] Anne C. Elster, my advisor. “Parallelization Issues and Particle-in-Cell codes”. PhD thesis. USA: Cornell University, 1994. URL: http://www.idi.ntnu.no/~elster/pubs/elster-phd.pdf.

[3] Jan C. Meyer. *Emerging Technologies Project: Cluster Technologies, PIC codes: Eulerian data partitioning*. Tech. rep. Norwegian University of Science and Technology, 2004.

[4] Nils M. Larsgård. “Parallelizing Particle-in-Cell codes with OpenMP and MPI”. Master Thesis. Norway: Norwegian University of Science and Technology, 2007.

[5] Anthony L. Peratt. *Physics of the Plasma Universe*. 2nd ed. New York, USA: Springer New York, 2015. ISBN: 978-1-4614-7819-5. DOI: 10.1007/978-1-4614-7819-5.

[6] C. K. Birdsall. “Particle-in-Cell Charged-Particle Simulations, Plus Monte Carlo Collisions With Neutral Atoms, PIC-MCC”. In: *IEEE Transactions On Plasma Science* 19.2 (Apr. 2, 1991).

[7] Erwin Kreyszig. *Advanced Engineering Mathematics*. 9th ed. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2006. ISBN: 978-0-471-72897-9.

[8] Holger Fehske, Ralf Schneider, and Alexander Weisse, eds. *Computational Many-Particle Physics*. Vol. 739. Lecture Notes in Physics. Springer-Verlag Berlin Heidelberg, 2008. ISBN: 978-3-540-74686-7. DOI: 10.1007/978-3-540-74686-7.

[9] Benedict J. Leimkuhler, Sebastian Reich, and Robert D. Skeel. "Integration Methods for Molecular Dynamics". In: *Mathematical Approaches to Biomolecular Structure and Dynamic*. Ed. by Jill P. Mesirov, Klaus Schulten, and De Witt Sumners. .III. New York, USA: Springer New York, 1996. ISBN: 978-1-4612-4066-2. DOI: 10.1007/978-1-4612-4066-2_10.

[10] James W. Cooley and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. DOI: 10.2307/2003354.

[11] Gordon E. Moore. "Cramming More Components onto Integrated Circuits". In: *Proceedings of the IEEE* 86.1 (1998).

[12] Laurie J. Flynn. *Intel Halts Development of 2 New Microprocessors*. May 8, 2004. URL: http://www.nytimes.com/2004/05/08/business/08chip.html?ex=1399348800&en=98cc44ca97b1a562&ei=5007 (visited on 07/08/2015).

[13] Sally A. McKee. "Reflections on the Memory Wall". In: *CF'04 Proceedings of the 1st conference on Computing frontiers*. New York NY, USA: ACM, 2004. DOI: 10.1145/977091.977115.

[14] Krste Asanovic et al. *The Landscape of Parallel Computing Research: A View From Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, 2006. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[15] Simon Harding and Wolfgang Banzhaf. "Fast Genetic Programming on GPUs". In: *Genetic Programming*. 10th European Conference, EuroGP 2007. (Apr. 11–13, 2007). Ed. by M. Ebner et al. Vol. 4445. Lecture Notes in Computer Science. Valencia, Spain: Springer-Verlag Berlin Heidelberg, 2007, pp. 90–101.

[16] Øystein E. Krog and Anne C. Elster. "Fast GPU+based Fluid Simulations Using SPH". In: *10th International Conference, PARA 2010, Reykjavík, Iceland, June 6-9, 2010, Revised Selected Papers, Part II*. Vol. 7134. Lecture Notes In Computer Science. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-28145-7_10.

[17] Ehsan Totoni et al. "Comparing the Power and Performance of Intel's SCC to Stat-of-the-Art CPUs and GPUs". In: *2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. New Brunswick, NJ, USA: IEEE, 2012. DOI: 10.1109/ISPASS.2012.6189208.

[18] Dr. Gene M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *AFIPS ’67 (Spring) Proceedings of the April 18-20, 1967, spring joint computer conference*. New York, NY, USA: ACM, 1967.

[19] John L. Gustafson. “Reevaluating Amdahl’s law”. In: *Communications of the ACM* 31.5 (1988). DOI: 10.1145/42411.42415.

[20] Michael J. Flynn. “Some Computer Organizations and their Effectiveness”. In: *IEEE Transactions on Computers* c-21.9 (1972).

[21] NVIDIA. *CUDA C Programming Guide*. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (visited on 07/08/2015).

[22] Thomas L. Falch and Anne C. Elster. “Register Caching for Stencil Computations on GPUs”. In: *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. Timisoara: IEEE, 2014. DOI: 10.1109/SYNASC.2014.70.

[23] Shane Cook. *CUDA Programming: A Developer’s Guide to Parallel Computing with GPUs*. Morgan Kaufmann, 2013. ISBN: 9780124159334.

[24] Paulett C. Liewer and Viktor K. Decyk. “A General Concurrrent Algorithm for Plasma Particle-in-Cell Simulation Codes”. In: *Journal of Computational Physics* 85.2 (1989). DOI: 10.1016/0021-9991(89)90153-8.

[25] Robert D. Ferraro, Paulett C. Liewer, and Viktor K. Decyk. “Dynamic Load Balancing for a 2D Concurrent Plasma PIC Code”. In: *Journal of Computational Physics* 109.2 (1993). DOI: 10.1006/jcph.1993.1221.

[26] Viktor K. Decyk and Tajendra V. Singh. “Adaptable PArticle-in-Cell algorithms for graphical processing units”. In: *Computer Physics Communications* 182.3 (2011). DOI: 10.1016/j.cpc.2010.11.009.

[27] NVIDIA. *cuFFT*. URL: http://docs.nvidia.com/cuda/cufft/ (visited on 07/08/2015).

[28] NVIDIA Corporation. *TESLA K20 GPU ACCELERATOR. Board Specification*. 2013. URL: http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v07.pdf (visited on 07/08/2015).

[29] *Intel Core i7-3770K Processor*. URL: http://ark.intel.com/products/65523/Intel-Core-i7-3770K-Processor-8M-Cache-up-to-3_90-GHz (visited on 07/08/2015).

[30] Ryan Smith. *NVIDIA Launches Tesla K20 & K20X: GK110 Arrives At Last*. Nov. 12, 2012. URL: http://www.anandtech.com/show/6446/nvidia-launches-tesla-k20-k20x-gk110-arrives-at-last (visited on 07/08/2015).

[31] Ryan Smith. *The NVIDIA GeForce GTX 980 Review: Maxwell Mark 2*. Sept. 18, 2014. URL: http://www.anandtech.com/show/8526/nvidia-geforce-gtx-980-review (visited on 07/08/2015).

[32] NVIDIA. *GeForce GTX 980 — Specification*. URL: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980/specifications (visited on 07/10/2015).

[33] NVIDIA. *CUDA Toolkit Documentation v7.0*. URL: http://docs.nvidia.com/cuda/ (visited on 07/08/2015).

[34] Grant R. Fowles. *Introduction to Modern Optics*. 2nd ed. Dover Publications, 1989.

[35] Peter W. Milonni and Joseph H. Eberly. *Laser Physics*. John Wiley & Sons, Inc., 2010. ISBN: 978-0-470-38771-9.

[36] R. W. Hockney and J. W. Eastwood. *"Computer Simulation Using Particles"*. New York, NY, USA: Taylor & Francis Group, 1988. ISBN: 9780521803892.

[37] I. H. Hutchinson. *"Principles of Plasma Diagnostics"*. Cambridge, United Kingdom: Cambridge University Press, 2002. ISBN: 9780521803892.

[38] S. J. Cooke et al. "GPU-accelerated 3D Electromagnetic PIC Simulations". In: *2011 Abstracts IEEE International Conference on Plasma Science*. Chicago IL, USA: IEEE, 2011. DOI: 10.1109/PLASMA.2011.5993003.

[39] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK 110/210*. Tech. rep. NVIDIA. URL: http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf.

[40] NVIDIA. *Tuning CUDA Applications for Maxwell*. URL: http://docs.nvidia.com/cuda/maxwell-tuning-guide/ (visited on 07/09/2015).

Appendix A

CUDA and cuFFT examples

The following sections will show some examples of simple CUDA and cuFFT programs, showcasing some distinctive features.

A.1 CUDA example

```
//Kernel definition
__global__ void kernel(double *data){
  //Determine thread index (1D)
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  double val = data[idx];
  data[idx] = val * val;
}

int main(){
  int N = 1024;
  double *h_data, *d_data;

  //Allocate pinned host memory
  cudaMallocHost(&h_data, N * sizeof(double));

  //Allocate device memory
  cudaMalloc(&d_data, N * sizeof(double));

  //Generate data on host
  getData(h_data, N);

  //Blocking transfer of data from host to device
  cudaMemcpy(d_data, h_data, N * sizeof(double),
    cudaMemcpyHostToDevice);

  //Set up thread structure
  threadsPerBlock = 256;
  blocksPerGrid = (N-1) / threadsPerBlock + 1;

  //Call kernel
  kernel<<<threadsPerBlock, blocksPerGrid>>>(d_data);

  //Create new cuda stream
  cudaStream_t transferStream;
  cudaStreamCreate(transferStream);

  //Non-blocking transfer of data from device to host,
  //possible because destination is pinned host memory.
  //Asynchronous transfer must be called to another
  //cuda stream.
  cudaMemcpyAsync(h_data, d_data, N * sizeof(double),
    cudaMemcpyHostToDevice, transferStream);

  //Synchronize with the cuda stream to ensure the
  //transfer is complete.
  cudaStreamSyncronize(transferStream);
```

```
    //{rint output
    print(h_data);

    //Free memory
    cudaFreeHost(h_data);
    cudaFree(d_data);
}
```

A.2 cuFFT example

```
//Create real-to-complex single precision cuFFT plan for
//array of length N
cufftHandle plan;
cufftPlan1d(&plan, N, CUFFT_R2C);

float *d_real;
cufftComplex *d_complex;
cudaMalloc(&d_real, N * sizeof(float));
cudaMalloc(&d_complex, N * sizeof(cufftComplex));
...
//Fill array with data
...
//execute transform on array
cufftExecR2C(plan, d_real, d_complex, CUFFT_FORWARD);
...
//Use transformed data
...
cufftExecR2C(plan, d_complex, d_real, CUFFT_INVERSE);
...
//Free resources
cufftDestroy(plan);
cudaFree(d_real);
cudaFree(d_complex);
```