Simulation of Ion Collection by a Sphere using the Particle-in-Cell Method on a GPU

by

Joshua Estes Payne

Submitted to the Department of Nuclear Engineering
in partial fulfillment of the requirements for the degree of

Masters of Science in Nuclear Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author .....
Department of Nuclear Engineering
May 18, 2012

Certified by .....
Ian Hutchinson
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Simulation of Ion Collection by a Sphere using the Particle-in-Cell Method on a GPU

by

Joshua Estes Payne

Submitted to the Department of Nuclear Engineering
on May 18, 2012, in partial fulfillment of the
requirements for the degree of
Masters of Science in Nuclear Engineering

Abstract

In this thesis, I designed and implemented a compiler which performs optimizations that reduce the number of low-level floating point operations necessary for a specific task; this involves the optimization of chains of floating point operations as well as the implementation of a "fixed" point data type that allows some floating point operations to simulated with integer arithmetic. The source language of the compiler is a subset of C, and the destination language is assembly language for a micro-floating point CPU. An instruction-level simulator of the CPU was written to allow testing of the code. A series of test pieces of codes was compiled, both with and without optimization, to determine how effective these optimizations were.

Thesis Supervisor: Ian Hutchinson
Title: Associate Professor

Acknowledgments

This is the acknowledgements section. You should replace this with your own acknowledgements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past century humanity has become increasingly dependent on the 4th state of matter, plasma. Attaining a better understanding of plasma behaviour and interaction is critical to developing faster computer chips, creating new sources of energy, and expanding humanities influence amoung the stars. One important subset of plasma behaviour is how plasmas interact with solid objects such as dust particles, probes, and bodies traveling through space. These interactions can be very difficult to explore experimentally, and therefore must be modelled.

A plasma's behaviour is heavily influenced by the collective electric and magnetic fields generated by the individual particles that comprise the plasma. This means that plasma behaviour is essentially a very large n-body problem, where for moderately dense plasmas n can be on the order of 10^{20}. No computer currently in existence can store the information for 10^{20} particles, and calculating the interaction of every particle in the set with every other particle would be prohibitvely long. The solution to this problem is to model only a subset of the true number of particles. The modeled behaviour of these particles and their contributions to magnetic and electric fields can be used to statistically infer the behaviour of the rest of the plasma, essentially from first princeiples. This method is called particle-in-cell (PIC), and operates by moving particles on a potential grid and updating that potential with the new particle density at every timestep. The flow of a general PIC code is shown in figure 1-1. The PIC method is a very robust and straightforward scheme for modeling plasma behaviour,

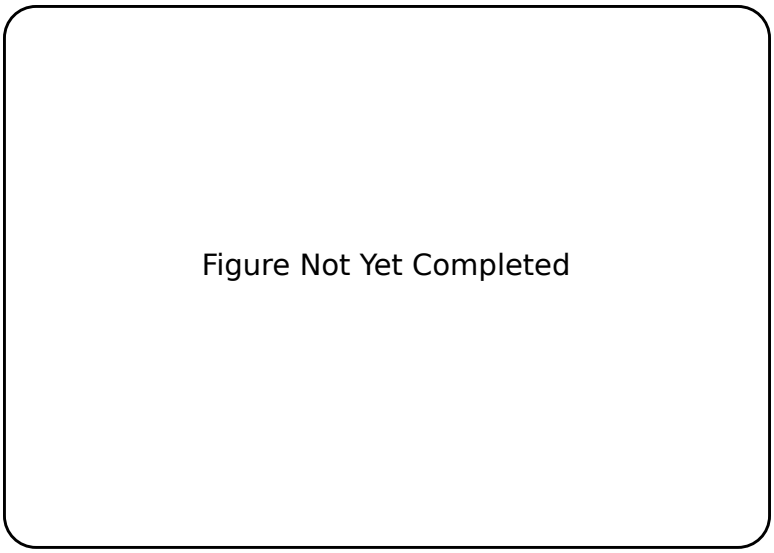

Figure 1-1: Flow schematic for the PIC method. Need to make figure

and is used extensively to model plasmas in complicated systems.

1.1 Motivation

The PIC method is very good at modeling complicated plasma behaviour, however this method still relies on tracking a very large number of particles for good statistics. In order to achieve "good" statistics PIC codes employ millions to billions of particles, which means that these codes can require a very large amount of computation time for each timestep. Running millions of particles on a single processor for hundreds of timesteps is not really feasible, it simply takes too long to compute a solution.

One way to reduce the total run time of PIC codes is to parallelize them. Since PIC codes operate on the fact that the potential changes little over the course of a single timestep, each particle can be assumed to be independent of its neighbors. This leads to a situation that is trivially parallel. In theory a machine with a million processors could run every particle on a seperate processor. This is of course assuming that the majority of the computational complexity lies in moving the particles and that comunication between processors is very fast.

1.1.1 GPUs vs CPUs

The ideal computing system for a particle in cell code should have a large number of relatively simple processors with very low communication costs. Traditional CPUs are just the oposite of this. CPUs tend to have 4-8 complicated processors that are very good at performing large operations on small sets of data, but very slow when it comes to communicating between multiple processors. CPUs are designed to be able to actively switch tasks on the fly. This makes them very good at simultaneously running web-browser, decoding a video, and playing a video game. However, this flexiblity requires a large number of cycles to switch between tasks, and a large amount of cache to store partially completed tasks.

Graphical processing units, or GPUs, forgoe the flexibility of CPUs in favor of more raw processing capability. Reducing the size of the cache and employing single instruction multiple data (SIMD) parallelism allows GPU manufactures to combine hundreds of processors on a single chip. In order to supply enough data to keep hundreds of processors GPUs also have a very large data channel between the processors and DRAM. All of these features are chosen to create a math processor that excels at tasks where each processor operates on data that is invisible to the other processors. These features give GPUs a significant raw floating point performance advantage over CPUs as seen in figure 1-2.

The hardware in GPUs is tailored to excel at performing tasks such as ray-tracing, which is very similar to particle moving. Therefore it is by no means unreasonable to conclude that GPUs can be very good PIC code processors. The advantages that GPUs have over CPUs for scientific computing include:

- Higher performance per cost.
- Higher performance per watt.
- Easier to upgrade.
- GPUs still improving with Moore's law.

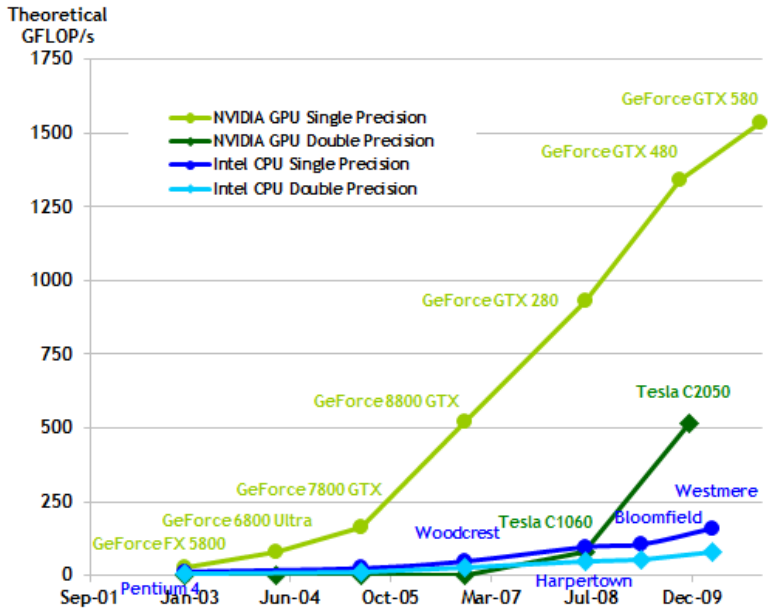

Figure 1-2: Performance comparison of GPUs vs CPUs.

All of which are observed when comparing the CPU and GPU versions of the same PIC code. While these advantages are very promising there are also several disadvantages to GPU computing:

- Increased code complexity.
- Smaller memory space.
- Smaller cache.
- Slow communication between CPU and GPU.
- Most developed GPU language is an extension of C.
- Algorithms can be very dependent on hardware configuration.

The key to developing efficient PIC algorithms that utilize GPUs lies in balancing the work between the two architectures. Some operations will be easier to implement on the CPU and be just as fast as the GPU while others will be significantly faster on the GPU. Partitioning the code between the different architectures begins to outline a very important aspect of parallel computing, multiple levels of parallelism.

1.2 Multiple Levels of Parallelism

Currently most parallelization is done by dividing up a task between a bunch of threads on different CPUs, and using an interface such as MPI to allow those threads to communicate. This network of threads has a master node, usually node 0, which orchestrates the communication between the other nodes. This is analgous to how a single CPU-GPU system operates. The CPU is the "Master" and serves as a communication hub for groups of execution threads on the GPU called thread blocks. Each thread block is itself a cluster of threads that can communicate through a memory space aptly named "shared memory".

1.2.1 Parallelization Opportunities in PIC Codes

1.3 Overview of sceptic3D

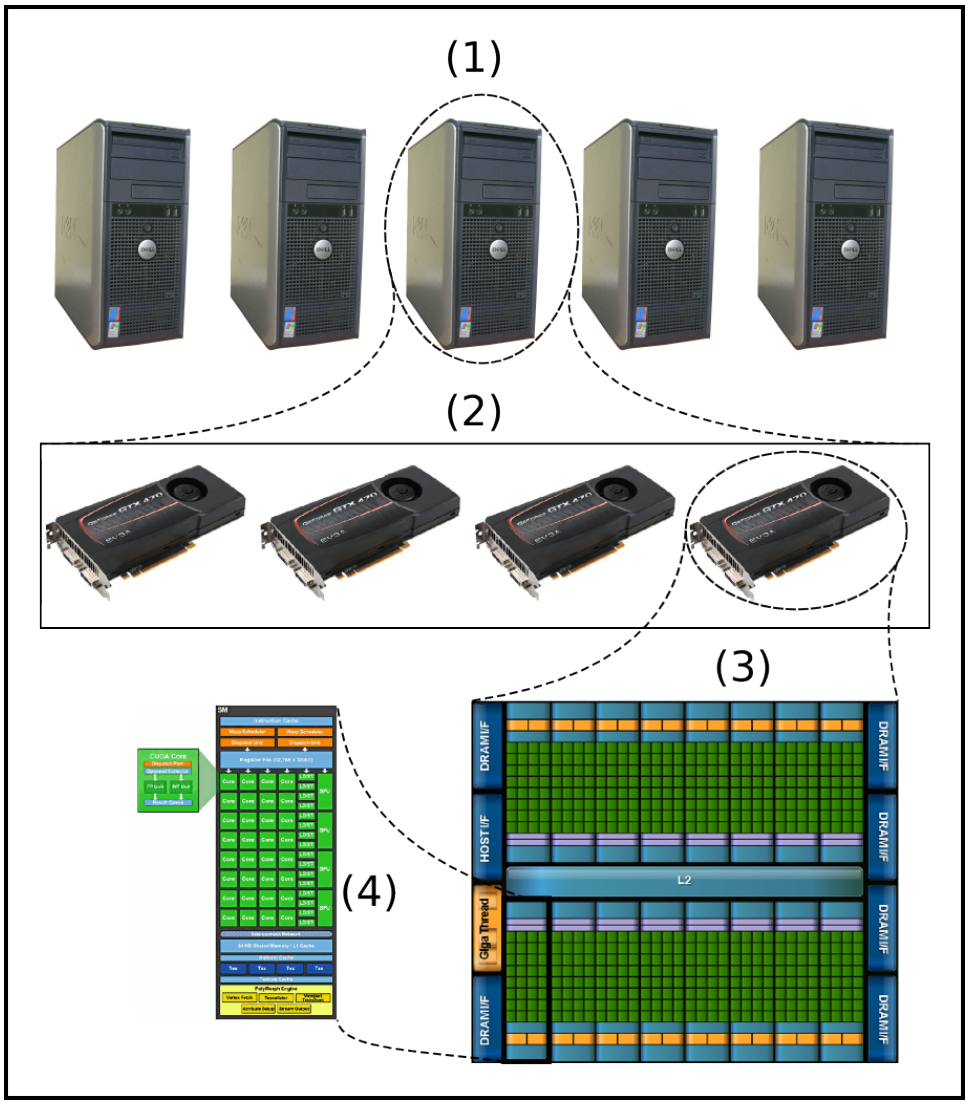

Figure 1-3: Multiple levels of parallelism. (1) Cluster of systems communicating through a LAN. (2) Multiple GPUs per system communicating through system DRAM. (3) Multiple streaming multiprocessors per GPU execute thread-blocks and communicate through GPU global memory. (4) Multiple cuda cores per multiprocessor execute thread-warps and communicate through on chip shared memory.

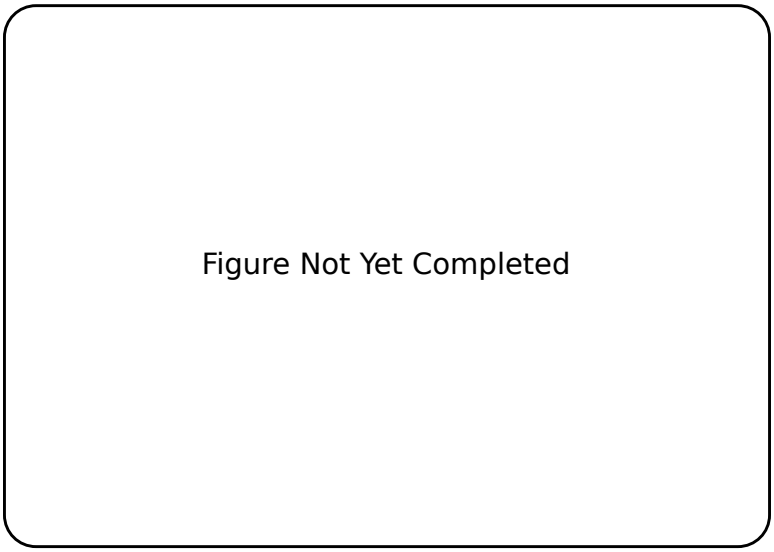

Figure 1-4: Flow schematic for the PIC method with parallelizable steps highlighted. Need to make figure

Chapter 2

Sceptic3D

Now that Sceptic3D is three dimensional hybrid PIC code specifically designed to solve the problem of ion flow past a negativley biased sphere in a uniform magnetic field. The current version of the code was derivied from the 2D/3v code SCEPTIC which was originally written by Hutchinson [3, 4, 1, 2].

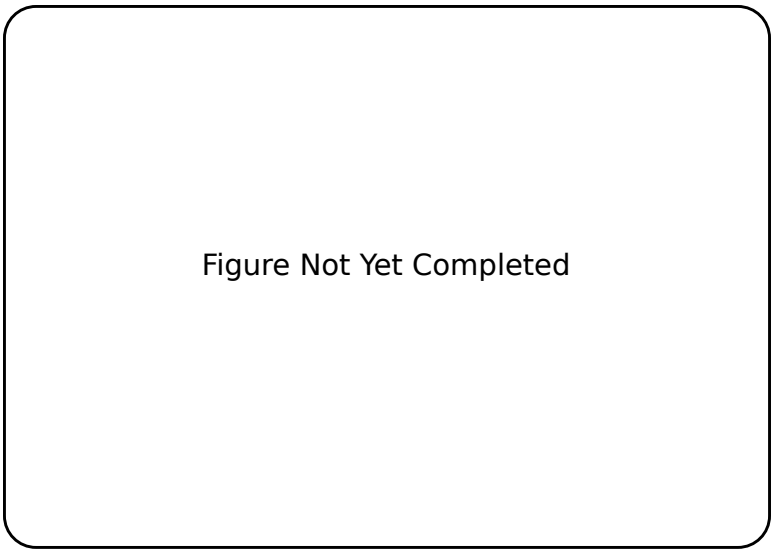

Figure 2-1: Flow schematic for the PIC method with sceptic subroutine names Need to make figure

2.1 Basic Code Structure

2.1.1 Charge Assign Details

2.1.2 Poisson Solve Details

2.1.3 Particle Advancing Details

2.2 CPU Code Profiling

2.3 Overview of sceptic3Dgpu Goals

2.3.1 Main Routines

2.3.2 Supporting Routines

2.3.3 Challenges to overcome

Chapter 3

Design Options

3.1 Particle List Structure

3.1.1 Other Codes

3.1.2 In house tests

3.2 Charge Assign

3.2.1 Naive Atomic Approach

3.2.2 Other Codes

3.3 Particle List Sort

3.3.1 Costs and Benefits

3.3.2 Other Codes

*Stantchev Particle Binning *Kong Particle Passing *Linked Particle List

3.3.3 In house tests

3.4 Particle Advancing

3.4.1 Assumptions

3.4.2 Other Codes

3.4.3 Reinjections and Diagnostics

3.5 Poisson Solve

3.5.1 Desired Performance

3.5.2 Performance vs Implementation Difficulty

3.6 Grid Dimension Constraints and Handling

Chapter 4

Implementation

4.1 Constraining Grid Dimensions

4.1.1 Constraints

There are two constraints that the grid dimensions must conform to. The first is set by the requirements of a simple z-order curve, the second is set by the size of the on chip shared memory. These constraints are expressed mathematically through the grid dimensions, n_r, n_θ, n_ψ, and the block subdomain dimensions, nb_r, nb_θ, nb_ψ.

$$\frac{n_r}{nb_r} = \frac{n_\theta}{nb_\theta} = \frac{n_\psi}{nb_\psi} = n_{virtual} \tag{4.1}$$

Where $n_{virtual}$ is the number of blocks that the grid is divided into in any dimension. In order to fully satisfy the constraints for a simple z-order curve, $n_{virtual}$ must be a power of 2.

The second constraint on the grid dimensions is set by the hardware. The goal is to maximize the shared-multiprocessor occupancy for the chargeassign stage of the code. Given that each block has the maximum number of threads, 512, and each thread requires roughly 25 registers, then the maximum number of threadblocks that can exist simultaneously on a single SM is 2. This means that each block can be allocated half of the total amount of shared memory on the SM. Compute capability 2.0 GPUs have 49152 bytes of shared memory per SM. Running two blocks per SM

provides each block with 24576 bytes of shared memory each, or 6144 floats per block. The maximum that all three block dimensions can be is 18. For the sake of simplicity this sets nb_r, nb_θ, and $nb_\psi \leq 18$.

A third, loose constraint can be set in order to force a minimum nuber of thread-blocks for the charge-assign. The command line option "–minbins#" sets the parameter $n_{virtual} = \#$. This is useful in ensuring that enough threadblocks are launched to populate all of the SMs on the GPU. To populate all of the SMs on a GTX 470 the code would need to launch at least 28 thread-blocks. For a GTX 580 with 16 SMs 32 thread-blocks are required to fill all of the processors.

4.1.2 Holding to the constraints

4.2 Particle List Transpose

As previously mentioned the particle list structure on the GPU is different than the structure on the CPU. On the GPU particles are stored in a structure of arrays, while on the CPU they are stored in a 6xn array. This means that in order to copy a particle list generated on the CPU to the GPU, or vice versa, the particle list must be transposed. The two main places in the code where this matters is when the particle list is initially populated at the start of the code, and when copying a list of pre-calculated reinjection particles from the CPU to the GPU at every time step during the advancing phase.

The particle list transpose was implemented on the CPU in two different ways depending on the compiler used and the available libraries. A GPU based particle list transpose is significantly faster than a CPU based transpose. However, the GPU has a very limited amount of DRAM compared to the CPU, and it is preferable to use as much of the available GPU memory as possible for the main particle list. In any case transposing the entire particle list only occurs once, but a smaller transpose is performed every time step for reinjected particles. This means that while a faster transpose is preferable, it represents so little of the total computation time that it is

not worth developing a complicated in place GPU transpose.

4.3 Charge Assign

4.3.1 Domain Decomposition

4.3.2 Particle Bins

4.3.3 Particle Push

- Atomic writes to shared memory - Block atomic writes to global memory

4.4 Particle List Sort

4.5 Poisson Solve

4.6 Particle List Advance

4.6.1 Checking Domain Boundaries

4.6.2 Diagnostic Outputs

4.6.3 Handling Reinjections

Chapter 5

Performance

5.1 Test Setup

5.1.1 Parameter Space Explored

5.1.2 Machine Parameters

5.1.3 Memory Bandwidth Comparison

5.2 Particle list size scan

The following tests were performed to explore the dependence of sceptic3Dgpu's run-time on the number of particles per node. These tests were performed using two MPI nodes and a grid size of 64x32x32.

5.3 Grid Size scan

5.3.1 Absolue Size

5.3.2 Ratio of Surface Area to Volume

Global Domain Ratio

Threadblock Sub-Domain Ratio

5.4 Kernel Parameters Scan

5.5 Discussion

Chapter 6

Conclusion

Appendix A

Tables

Table A.1: Armadillos

Armadillos	are
our	friends

Appendix B

Figures

Figure B-1: Armadillo slaying lawyer.

Figure B-2: Armadillo eradicating national debt.

Bibliography

[1] I H Hutchinson. Ion collection by a sphere in a flowing plasma: 3. Floating potential and drag force. *Plasma Physics and Controlled Fusion*, 47(1):71–87, January 2005.

[2] I H Hutchinson. Collisionless ion drag force on a spherical grain. *Plasma Physics and Controlled Fusion*, 48(2):185–202, February 2006.

[3] IH Hutchinson. Ion collection by a sphere in a flowing plasma: I. Quasineutral. *Plasma physics and controlled fusion*, 1953, 2002.

[4] IH Hutchinson. Ion collection by a sphere in a flowing plasma: 2. Non-zero Debye length. *Plasma physics and controlled fusion*, 1477, 2003.