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

List of Algorithms

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 par-

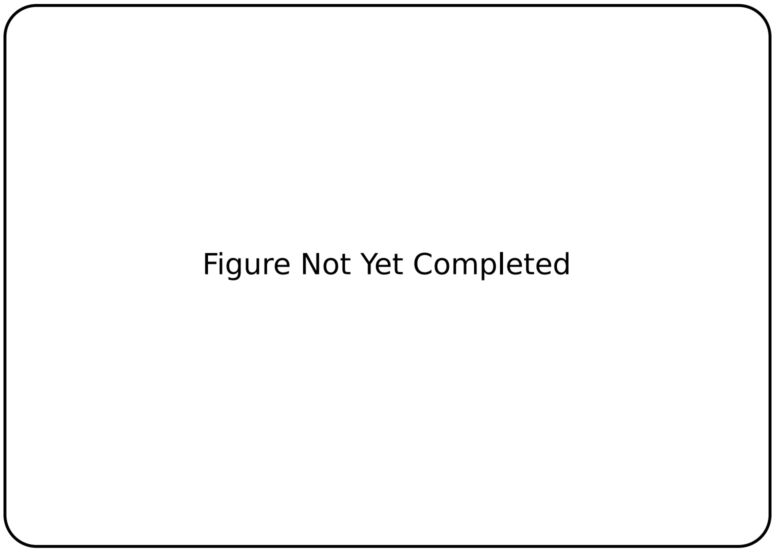

Figure 1-1: Flow schematic for the PIC method. Need to make figure

ticle 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, 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.

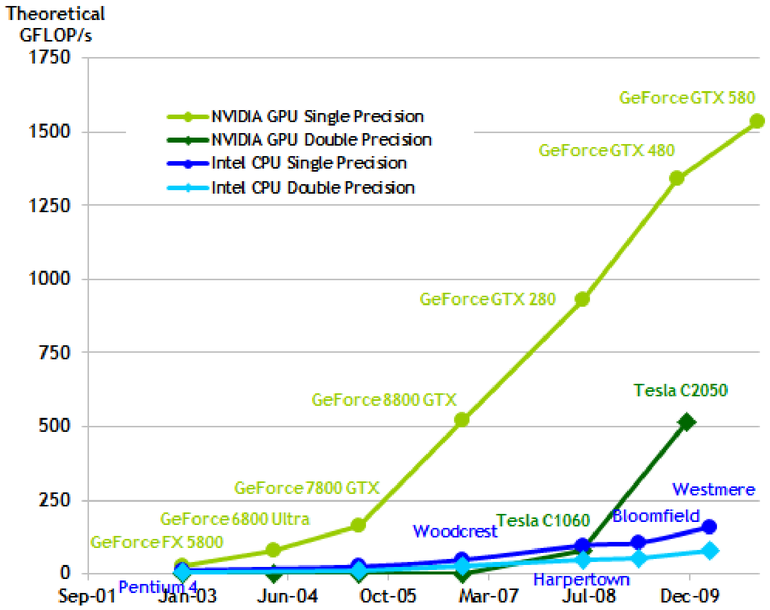

Figure 1-2: Performance comparison of GPUs vs CPUs.

- Easier to upgrade.
- GPUs still improving with Moore's law.

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”.

The point hear is that multiple domain decompositions must be performed in order to fully utilize the capabilities of this system. The coarse decomposition is very similar to that used for MPI systems, but the fine decomposition can be very different due to the significanlty higher memory bandwidth and smaller cache of GPUs.

1.2.1 Parallelization Opportunities in PIC Codes

1.2.2 Current Status of GPU PIC codes

Some work on efficient GPU based PIC codes has already been done. This past work will be briefly introduced here and discused in depth in chapter 3. Burau et al developed a fully relativistic PIC code for a gpu cluster.

1.3 Overview of sceptic3D

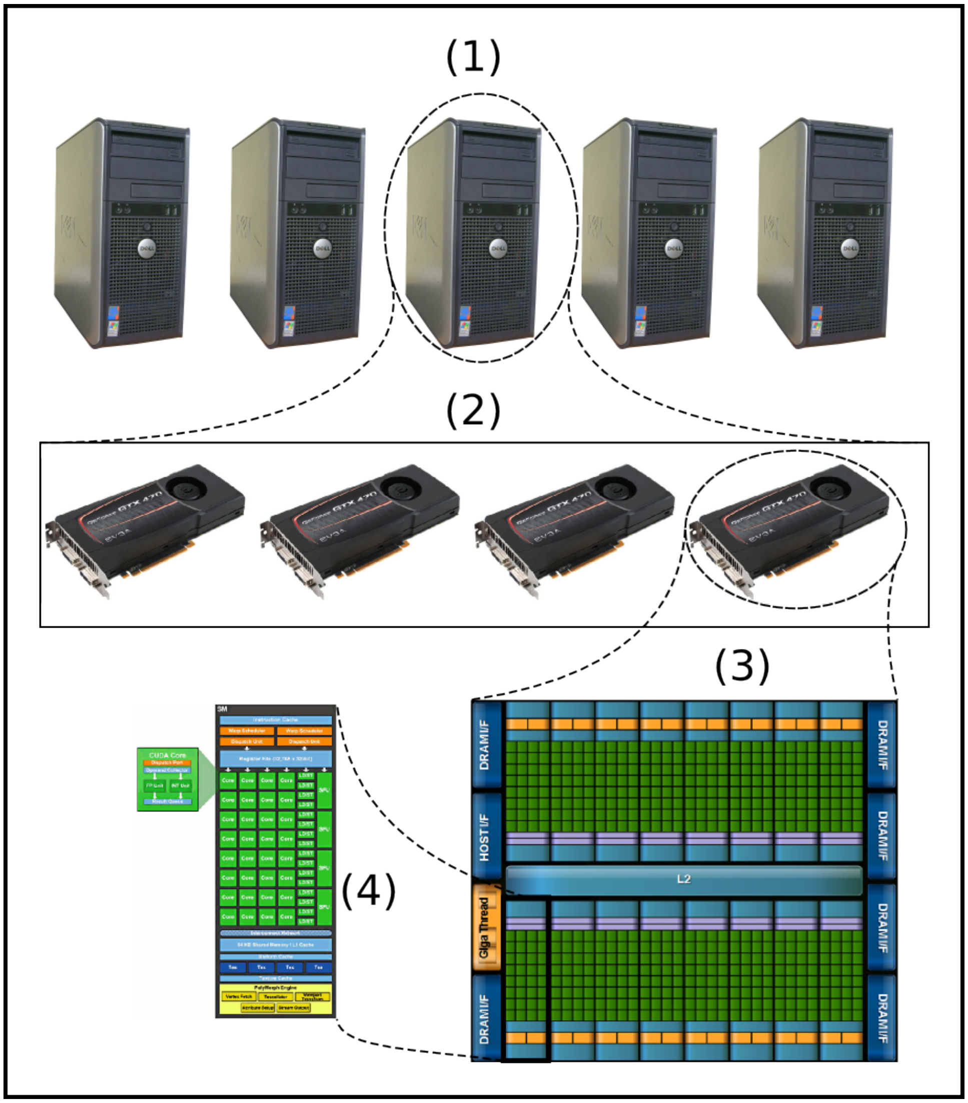

Figure 1-3: Multiple levels of parallelism. (1) Cluster of systems communicating through a LAN. (2) Multiple GPUs per system communicating through system DRAM. (3) Multiple streaming multiprocessors per GPU execute thread-blocks and communicate through GPU global memory. (4) Multiple cuda cores per multiprocessor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

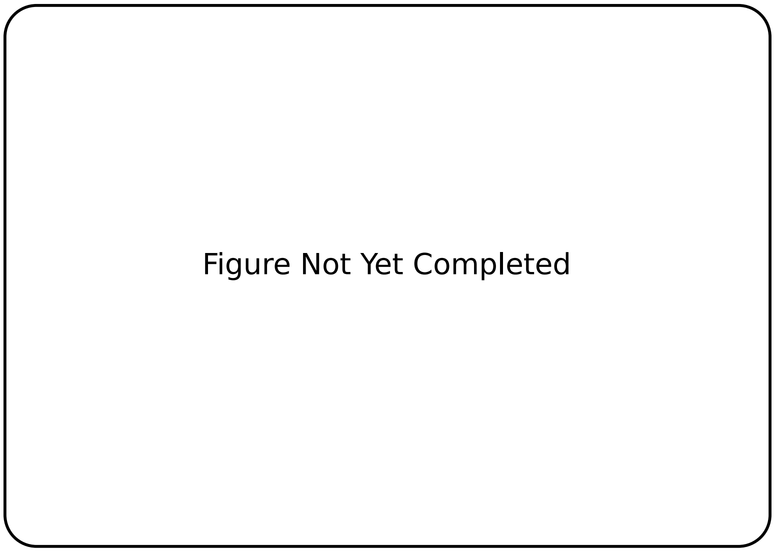

Figure 1-4: Flow schematic for the PIC method with parallelizable steps highlighted. Need to make figure

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 [6, 7, 4, 5].

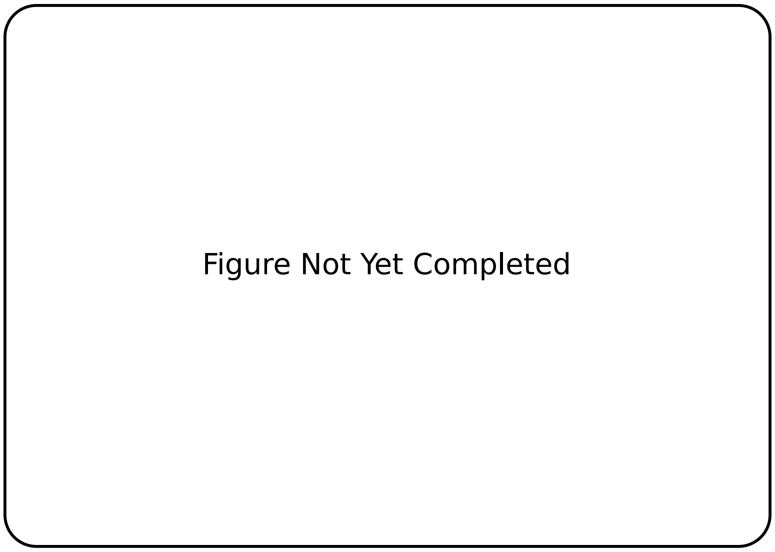

Figure 2-1: Flow schematic for the PIC method with sceptic subroutine names Need to make figure

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

GPU architecture is significantly different than that of a CPU, and thus a high performance PIC code on a GPU is going to look a lot different from its CPU equivalent. Memory access patterns, cache behavior, thread communication, and thread workload all have significant impacts on the performance of GPU codes. This means that porting an existing PIC code to the GPU is by no means straightforward, the data structures and algorithms will likely be different from the original serial code.

Performance is just one facet of the code design, maintaining separate CPU and GPU versions of the same code presents additional problems. Programmers tend to be lazy in that the fewer lines of code that they have to write, the better. If a new feature is desired, then two different implementations of that feature must be written and debugged. From the lazy programmers perspective this is to be avoided as much as possible. Therefore, it is very important that the GPU version of the code utilize as much of the CPU code as possible. This means that interoperability between the CPU and GPU code must be both efficient and fast.

Performance and maintenance are the two key issues that were considered when designing sceptic3Dgpu. Some of these issues have been investigated previously, although the amount of research in this area is still very small. To make matters worse, the specific techniques used are rapidly evolving with every new generation of graphics card. It is unlikely that the pace of GPU hardware evolution will slow in the near future. Spending large amounts of time optimizing algorithms for the current generation of hardware is inadvisable, and therefore the design of the code should focus on utilizing techniques that emphasize the underlying principles of GPU design or utilize library functions that will be optimized for each generation of hardware.

The goal of this chapter is to outline various design options for implementing the various steps of the PIC algorithm on the GPU and explore the pros and cons of each option. Solutions used by other researchers will be outlined and evaluated based on their applicability to sceptic3D and their applicability to PIC codes in general. To accelerate these evaluations a simple 3D sandbox PIC code was implemented on the GPU in addition to several other basic comparison codes.

3.1 GPUPIC Sandbox and the big questions

The first step in the development of sceptic3Dgpu was to create a very simple, generalized pic code that performed the major steps of the PIC algorithm and implement it in CUDA. This simple code, we'll call it GPUPIC_testbed is designed without making any assumptions about the physics of the system. GPUPIC_testbed operates in Cartesian coordinates with periodic boundary conditions. We do not really care too much about the field solve since in the serial version it takes a very small amount of time compared to the particle advance and charge assign steps. By recognizing the low priority of the field solve we really only need to characterize the performance of the following 5 steps:

1. Read the particle data
2. Read the Potential data for that particle
3. Move the particle
4. Write the new particle data back to the particle list
5. Interpolate Charge to Mesh
6. Goto 1

Component	Runtime (ms)
Particle data read, move, and write	375
Potential Grid Read	467
Charge Assign	1.143e4
Total	1.227e4

Table 3.1: Total Execution times for 100 iterations of the key steps of the move kernel at three different optimizations.

The first implementation of this code was very naive. The only real difference from a serial version was the density array update, which used atomic updates on global memory in order to prevent memory collisions between multiple threads. Other than that the code boiled down to unrolling the loop over all of the particles into one particle per thread. The runtime breakdown of this code for a 32^3 grid and 4.2 million particles is shown in table 3.1.

As you can see, the particle move and the potential read are very similar, but the charge assign is very slow. Determining how we can better adapt the charge assign to the GPU is our first major challenge. Several ways of dealling with the issue of the charge assign will be discussed in the following section. Some of the other issues that will be discussed in this chapter are:

- Particle Data Structure: Is it better to use an Array of structures, like the fortran code, or a Structure of Arrays?
- How do we handle divergent processes in the advancing routine, such as losses, reinjections, and collisions?
- At what point does the field solve become a dominant cost?
- Are there any new issues that arise from solutions to the other issues?

3.2 Charge Assign

There are two different ways to approach the charge assign, one in which information is "pulled" from the particles by the vertices, and one in which data is "pushed" by the particles to the vertices. Let G represent a grid of domain D of dimension d comprised of all vertices $v_s \in$ D. We can define some distribution function $f(v_s)$ at each of the vertices which is the sum of some function $\mathrm{K}(v_s, p_i)$, where p_i is the position of particle i. Given these definitions the algorithms for the particle pull and particle push method are algorithms 3.1 and 3.2 respectively.

Algorithm 3.1 Particle Pull Method of charge deposition. From Stantchev et al. [11]

```
// Loop over the verticies first
for all vertex $v_s \in G$ do
    find $\mathcal{P}(v_s)$
    f($v_s$) $\leftarrow$ 0
    for all $p_i \in \mathcal{P}(v_s)$ do
        f($v_s$) $\leftarrow (f)(v_s) + \mathrm{K}(v_s, p_i)$
    end for
end for
```

Algorithm 3.2 Particle Push Method of charge deposition. From Stantchev et al. [11]

```
// Loop over the verticies first
for all vertex $v_s \in G$ do
    f($v_s$) $\leftarrow$ 0
end for
for all particle $p_i \in$ D do
    find $\mathcal{V}(p_i)$
    for all $v_s \in \mathcal{V}(p_i)$ do
        f($v_s$) $\leftarrow (f)(v_s) + \mathrm{K}(v_s, p_i)$
    end for
end for
```

As pointed out by [11] each method has its advantages and disadvantages. For an algorithm consisting of N particles and k grid vertices the advantages and disadvantages are as follows:

The particle pull method

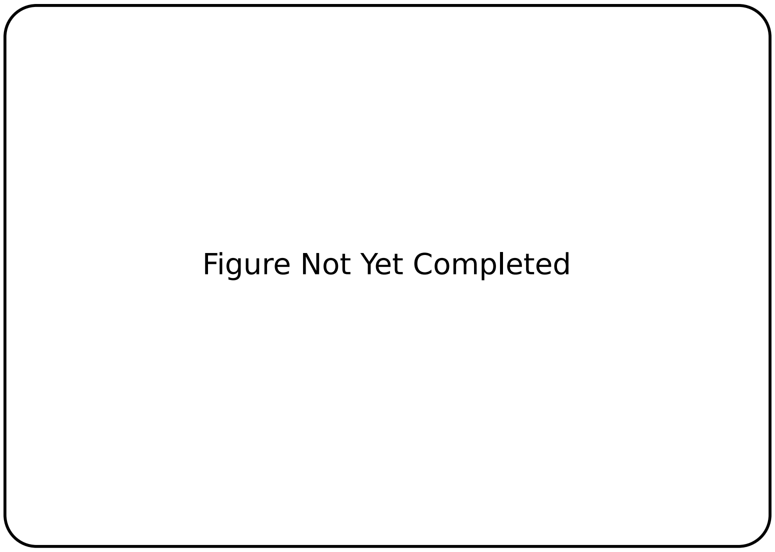

Figure 3-1: Atomic Memory collisions

- requires $\mathcal{O}(2^d N + k)$ read write operations
- $\mathcal{P}(v_s)$ is expensive to retrieve dynamically unless particles are organized

The particle push method

- requires $\mathcal{O}((2^d + 1)N)$ read/write operations
- $\mathcal{V}(p_i)$ is easily computed dynamically from the particles coordinates

The charge assign that we implemented in the sandbox PIC code is a particle push to an array in global memory. In that implementation we used atomic opperations to prevent memory collisions. Looking back at table 3.1 we notice that the charge assign step constitutes about 93% of the total runtime. Unfortunately this poor performance is a result of serialization caused by the atomic updates. Additionaly, since the grid is far too large to fit in shared memory these updates must be performed on global memory, which has much higher latency and lower bandwidth. When a thread attempts to update a value in memory and finds that it is locked it must then repeat the process until it succeeds. Every failed update represents an additional slow global memory access that is essentially wasted.

The technique applied for MPI codes is parallel reduction. Each thread deals

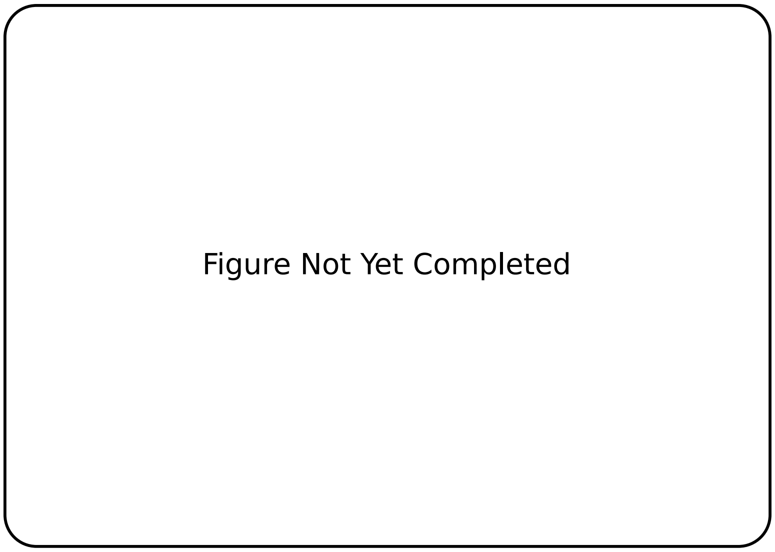

Figure 3-2: One thread per cell

with a subset of the particle list and tallies up the contributions of that list to some array in memory private to a single thread. Once every thread has recorded the contributions from their subset of the particle list a parallel reduction is performed in order to quickly sum up the contributions from all threads. The problem with directly applying this solution to the GPU is that when a thread reads in a particle the thread must be able to account for every possible location that the particle can contribute to. With a completely random particle list any given particle can contribute to any element of the grid. However, say a thread knows that every particle that it reads in will only contribute to one element of the grid. This thread now only has to keep track of a single value, since it knows that every particle it sees will only contribute to this value. When it comes time for all of the threads to contribute to the final result each thread provides the full answer for a single element. This is essentially the particle pull method described in algorithm 3.1 without the need to to retrieve $\mathcal{P}(v_s)$ dynamically. One of the main benefits to this method is that it significantly reduces the memory requirements of each thread but imposes the constraint that a thread is given only particles that exist within its domain. We will worry about this additional constraint later.

Now consider this, the MPI code works well for a few randomly ordered sets

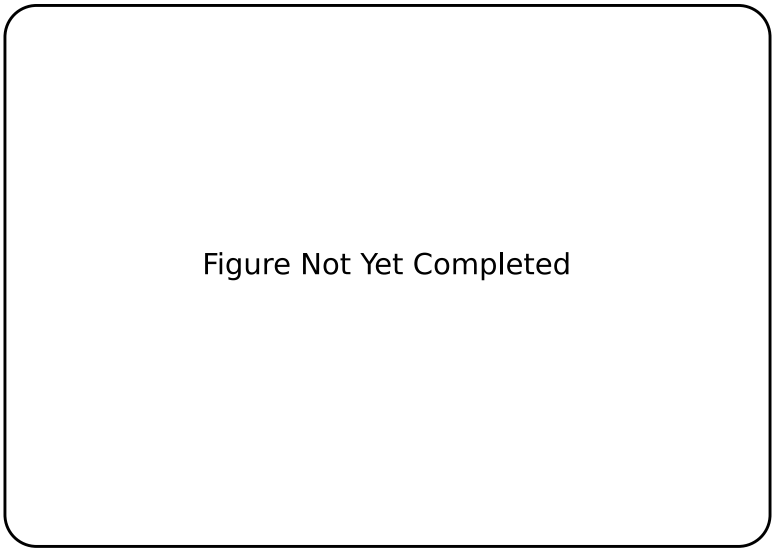

Figure 3-3: MPI and One thread per cell

of many particles, or objects, manipulated by a small number of threads. The decomposition technique works for a small number of organized sets of a few objects manipulated a large number of threads. If we think of threads operating on small groups of particles as objects and we replace every instance of 'objects' with 'threads' in the previous two sentences we end up with an interesting situation. Apply the MPI technique to a few randomly ordered sets of many threads each operating on a small number of particles. Essentially if we want to run really large particle lists we can divide up the list amongst several nodes. Each node uses many threads to process a small ordered subset of this list and contribute to the full array. Once every node has completed its own tally the standard MPI technique is used to gather the tallies of all the nodes. This is an excellent example of multi-grained parallelism. The level consisting of multiple nodes is coarse parallelism while the node level is a finer level of parallelism.

We can take this methodology even further on the GPU by recognizing that we can parallelize the single element summations using reductions. Taking this to the limit of one thread per particle on the GPU we end up with each thread block, or several blocks, is responsible for a subset of the particle list. All of the particles in the block's list will contribute to the same element. The threads within each block

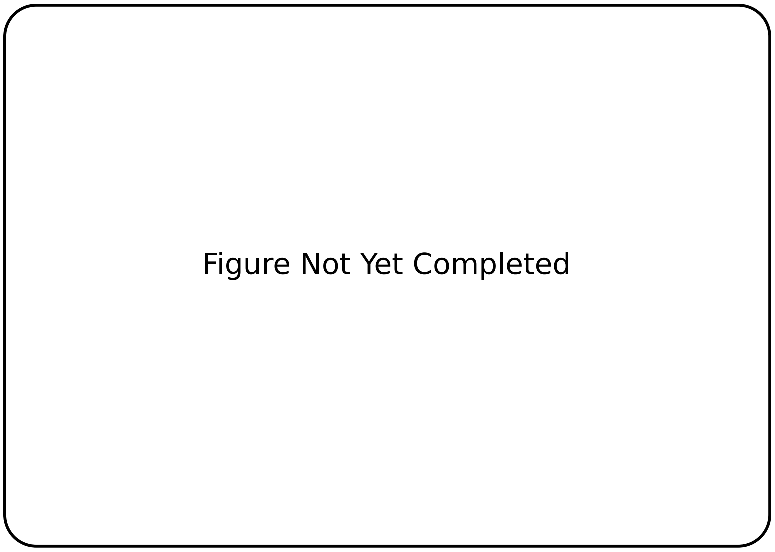

Figure 3-4: Three levels of parallelism for the charge assign

Component	Atomic-Updates (ms)	Sorted+Reduction (ms)
Particle data read, move, and write	375	468
Potential Grid Read	467	285
Charge Assign	1.143e4	542
Particle List Sort	0	2.305e3
Total	1.227e4	3600

Table 3.2: Total Execution times for 100 iterations of the key steps of the move kernel at two different optimizations.

read in their particles contribution to that element into shared memory. With all of the data in shared memory a very fast parallel reduction can be performed.

We implemented this technique in the sandbox PIC code and compared the runtime of the reduction particle-pull to the atomic particle-push. The results of this comparison can be seen in table 3.2.

As you can see from the table, the charge assign is on the order of 20x faster using the reduction technique, although this speedup is somewhat offset by the sorting requirement. Sorting the particles also benefits reading the potential during the advancing step. This speedup is a result of increased cache hits due to all threads within the same thread block accessing the same addresses in the potential array.

Although we have successfully reduced the cost of the charge assign we have introduced an additional cost of a sorting step[1]. In the case of the sandbox code the sort step accounts for roughly 70% of the runtime. Fortunately several other projects have figured out that there are ways reduce the sorting costs while maintaining some of the performance achieved by utilizing a sorted particle list.

3.2.1 Other Codes

There are several papers which point out that sorting by cell at every time step is not entirely necessary for the particle-pull method. It is possible to minimize the sorting requirement by expanding the sorting bins to include multiple cells, or rather, by dividing the simulation space into slabs composed of multiple cells. The advantage to this technique is sorting is only required between slabs, but not within the slabs.[1]

This slab method, as described by Abreu et al, is used on a one thread per slab basis. One thread for each slab loops over all of the particles that belong to that slab, contributing to an array that is the same size as the slab. Once a thread completes its particle loop it writes the portion of the array that it is responsible for to the main array, using atomic operations for guard cells.[1] Similar approaches are used by Stantchev et al and Kong et al.[11][8]

Unfortunately it is difficult to apply the reduction version of the particle push to the slab method. The reason this is difficult boils down to limited shared memory. Consider a slab with nv_{slab} vertices. In order for the reduction to work we need to have $nv_{slab}*$nthreads floats to store the results of each thread. For a typical NVIDIA GPU with 49k shared memory per streaming multiprocessor and 128 threads per block, we are limited to a slab of about 96 vertices per slab. This amounts to 9 cells per slab for a 3D grid, or about 3 fewer steps for a radix sort.

[1] It should be noted that the sort used here is an older version of the radix sort, newer versions, such as the thrust implementation are much faster

The approaches used by Kong and Stantchev is a sort of hybridization of the push and pull algorithms. Here the grid is domain decomposed into sub-domains and each sub-domain assigned to a thread-block. Each thread-block has an array representing the distribution function for that sub-domain allocated in shared memory. Particles are ordered in the particle list according to what sub-domain they reside in. Within each sub-domain the charge assign is performed like a particle push. Threads loop through a subset of the particles, check which vertices that particle is updating, and update the distribution function at those vertices.

In order to avoid memory collisions both Kong and Stantchev use a technique similar to atomic operations. The technique that they used is called thread-tagging and is no longer needed due to the addition of atomic operations for shared memory.[11][8] This approach has several advantages over the reduction technique, the primary reasons being lower order sorting keys and slightly easier implementation. The disadvantage of this approach is that because it relies on atomic operations there is no guarantee that the results are deterministic since the order of the atomic operations is undefined.

3.3 Particle List Sort

In the previous section we discussed what is required for an efficient charge assign on the GPU. In order to massively parallelize the charge assign and avoid memory collisions the particle data must be organized. Unfortunately this means introducing a new step in the PIC method, a sort step. Looking back at table 3.2 we can see that this sort step is now the dominant cost by a large margin[2]. Operating with one step consuming 64% of the total run time is unacceptable, we need to figure out a better way of keeping the particle data organized than this radix sort. Fortunately this problem has been explored in great detail by just about everyone else who has

[2] Please note that these results are for the radix sort outlined in the NVIDIA GPU computing SDK version 3.1, developed by N. Satish et al.[10]

developed a GPU PIC implementation.

Particle sorting for GPU PIC codes basically comes in four flavors:

- Partial sort using message passing. [8][3]
- In-place particle Quicksort. [11]
- Linked list reordering [2]
- Full Radix Sort-by-key and reorder. [1]

Each of these methods have their own advantages and disadvantages. For the purposes of this code we are looking for a method that is fast for a broad range of applications and does not depend too greatly on the specifics of the problem.

3.3.1 Message Passing

Going back a section to the charge assign we concluded that sorting by cell is unnecessary. Instead we are ordering the particles according to a group of cells called bins. For most cases the dimensions of the bin will be greater than the average distance that a particle will travel in a given time step. There are cases in which the particle path length will be smaller than the size of a cell, however this is much more likely if the domain in question is several cells wide. The benefit of considering this case is that only a small fraction of the particles will leave the domain during a given time step, and thus only a small number of particles need to be moved from one domain to the next. Most of the particles will remain in their respective domains and therefore do not need to be sorted. Instead of a full particle sort we want a method that will partially sort the particle list, only handling the particles that changed domains.

One partial sorting method is similar in principle to message passing. The particle list is divided up into sections according to domain. Whenever a particle leaves its current domain it is flagged. Flagged particles are then moved to different sections of the particle list through some kind of buffer. There are currently two approaches to this. The approach taken by Kong et al, illustrated in figure 3-5 is

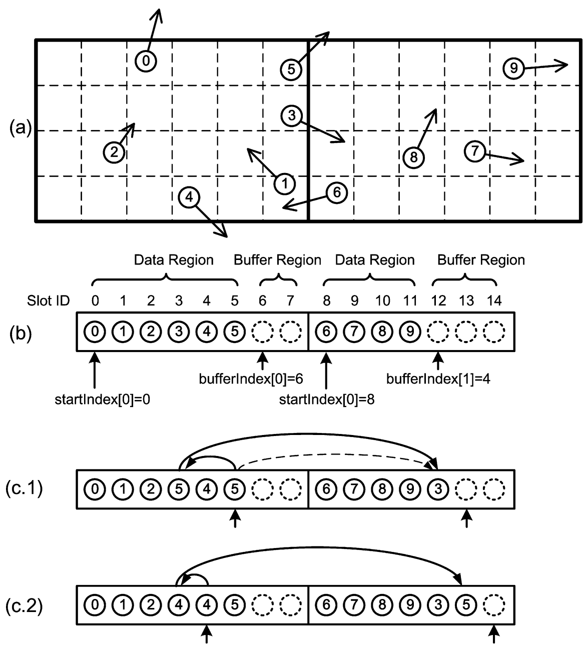

Figure 3-5: Kong particle sort. (a) Bi-cluster of cells. (b) Data structure of the bi-cluster particle array. (c.1) Particle 3 and 5 move to the right. Particle 3 is moved first to slot 12 which frees up slot 3. Slot 3, now empty, copies the last particle in the data region, which is particle 5. Particle 5 is also moving to the right so slot 3 copies it to slot 13 and replaces the contents of slot 3 with particle 4. Image taken from [8].

based on integrating the buffer into the particle list. The particle list is structured such that each sub-domain's section of the particle list is divided into two regions, a data region and a buffer region. Using this particle list structure as a foundation, the rest of the sorting algorithm proceeds as follows[8]:

1. Sub-domains, referred to by Kong as clusters, that are adjacent horizontally are grouped into pairs called bi-clusters. This first step is odd cells on the left, even cells on the right.
2. Particles that are moving from the left cluster to the right are copied from the left cluster into the buffer section of the right cluster.
3. Step 2 is then repeated for particles moving from the right cluster to the left.
4. Repeat steps 1 to 3 for bi-clusters for even cells on the left and odd cells on the right.
5. Perform steps 1 to 4 for vertically oriented bi-clusters.

In cases where the number of particles in a cluster is greater than the number

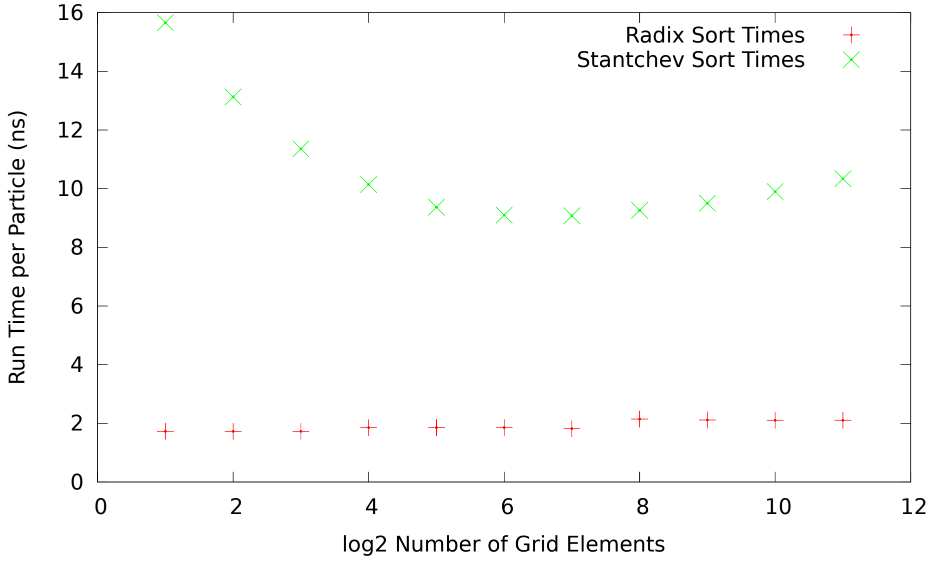

Figure 3-6: Comparison of Stantchev Sort and the thrust radix sort.

of slots in that cluster a global data reorder must be performed. This reorder is only performed when a particle to slot ratio exceeds a certain limit. When a cluster exceeds this threshold the code increases the buffer for this cluster by reducing the buffer of other clusters. This operation is carried out through a sequence of calls to $cudaMemcpy()$ where a section of data from end of a cluster is copied to the end of the buffer of the previous cluster. The start index of the adjusted cluster is then shifted by the amount of memory copied. [8]

The second approach used by Decyk et al breaks the relocating particles into two groups, those moving within a thread group and one for all the rest. The buffer for the second group is stored in a different array from the main particle list.

All of the flagged particles are condensed and written to some kind of buffer using stream compaction. Once all particles that are being relocated have been transferred to the buffer and all threads have been synchronized particles in the buffer array are broadcast to the empty slots in the primary array.

3.3.2 Costs and Benefits

3.3.3 Other Codes

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

3.3.4 In house tests

3.4 Particle List Structure

3.4.1 Other Codes

3.4.2 In house tests

Component	SoA (ms)	AoS (ms)	Speedup (SoA vs AoS)
Particle data read, move, and write	758	955	1.26x
Count Particles	32.7	109	3.35x
Data Reorder	346	480	1.38x
Total CPU run time	2491	3284	1.31x

Table 3.3: Execution times of main steps for Array of Structures and Structure of Arrays. Count Particles and Data Reorder are steps used for a sorted particle list. Count Particles counts the number of particles in each sub-domain. Data Reorder reorders the particle list data after the binindex / particle ID pair have been sorted by the radix sort.

3.5 Particle Advancing

3.5.1 Assumptions

3.5.2 Other Codes

3.5.3 Reinjections and Diagnostics

3.6 Poisson Solve

3.6.1 Desired Performance

3.6.2 Performance vs Implementation Difficulty

3.7 Grid Dimension Constraints and Handling

Chapter 4

Implementation

4.1 Constraining Grid Dimensions

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 threadblocks 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.

4.1.1 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

As previously mentioned, the charge assign is one of the most difficult funcitons to parallize. The niave approach of applying a thread to every particle and atomically adding each particles contribution to an array in global memory is very slow. Grouping the particles spatially allows the majority of the atomic operations to be done in the context of shared memory which is much faster than global memory. The resulting algorithm resembles basic domain decompositon where each thread-block represents a seperate sub-domain. The actual charge deposition method in this shceme is very similar to the niave approach, with a key difference being that all the threads in the thread block are operating on shared memory. Once all particles in the subdomain have contributed to grid in shared memory it takes only a small number of global memory accesses to write the contributions of a large number of particles to the main array χ.

4.3.1 Domain Decomposition

The primary grid is decomposed into sub-domains of size nb_r, nb_θ, nb_ψ. The methods for determining the size of the sub-domains is outlined in section **??**. The indexing of the sub-domains is done using a z-order curve in order to preserve spatial locality of the sub-domains in memory. This is done in an attempt to reduce the mean distance that particles must be moved in memory during the sort phase. A graphical representation of this is shown in figure 4-1.

In addition to representing a sub-section of the computational mesh, each sub-domain must have a section of the particle list associated with it. The sub-domain

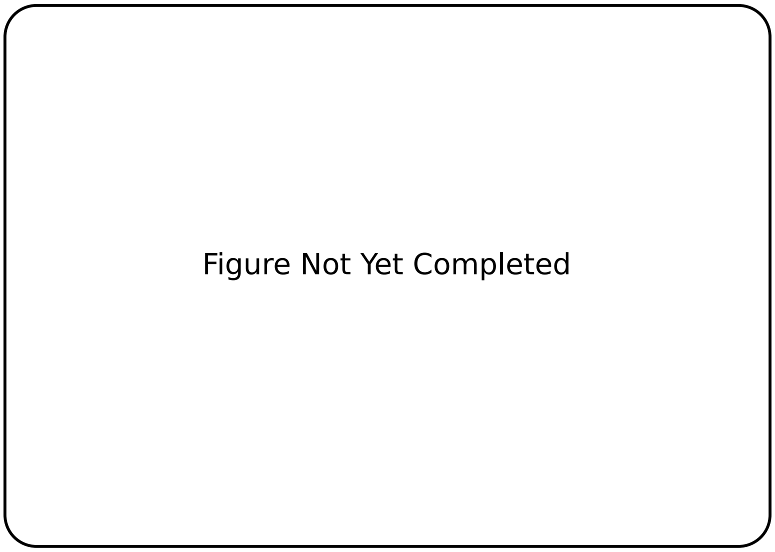

Figure 4-1: Graphical Representation of domain decomposition and ParticleBin organization. Need to make figure

must know all of the particles that reside within the region defined by that sub-domain. In essence each sub-domain represents a bin of particles that corresponds to some spatial location, hence the use of "ParticleBin" as the naming convention for these object.

4.3.2 Particle Bins

The *ParticleBin* object keeps track of all of the particles that reside in the region of space that the *ParticleBin* represents. For the sake of simplicity all of the particle bins are the same size spatially, which means that the *ParticleBin* object only has to keep track the section of the main particle list that the bin represents and the spatial origin of the bin.

In the context of the particle list each bin represents a pair of bookmarks that bound a section of the particle list. The bookmarks for each bin are calculated after the particle list is sorted by algorithm 4.1.

The spatial origin of the bin is hashed using a z-order curve and stored as a 16-bit unsigned-integer. This 16-bit integer is refered to as the binID and is used for

Algorithm 4.1 ParticleBin Bookmark Calculation

```
for all threadID = 0 → ParticleList.nptcls in parallel do

    binID = ParticleList.binID[threadID]
    $\text{binID}_{\text{left}}$ = ParticleList.binID[threadID − 1]
    $\text{binID}_{\text{right}}$ = ParticleList.binID[threadID + 1]
    if binID $\neq$ $\text{binID}_{\text{left}}$ then
        ParticleBins[binID].ifirstp = threadID
        ParticleBins[$\text{binID}_{\text{left}}$].ilastp = threadID − 1
    end if
    if binID $\neq$ $\text{binID}_{\text{right}}$ then
        ParticleBins[binID].ilastp = threadID
        ParticleBins[$\text{binID}_{\text{right}}$].ifirstp = threadID + 1
    end if
end for
```

determining the region of the domain that a bin is responsible for as well as a sorting key for the particle list. Calculating the binID will be discussed in more detail in section 4.4. A 16-bit unsigned integer is used for several reasons. First the sorting method detailed in section 4.4 is dependent on the number of bits of the sorting key. Second, the upper bound on the grid size set by using a 16-bit integer to store the z-order hash is much larger than the largest grid size that would need to be run. For a 16-bit integer this upper bound is 512^3 grid points. The third reason for using a 16-bit integer is that it also reduces the memory requirements of the particle list by about 5%, which does help when trying to run as many particles on the GPU as possible.

4.3.3 Particle Push

Now that the particles are organized spatially in memory, it is trivial to assign a single thread block to a region of space and corresponding particle bin in order to perform the particle push. This process is rather simple and is outlined in psuedo code in algorithm 4.2.

Each thread block reads in 512 particles at a time, although only 32 particles, a warp, are processed in parallel within the block. Each thread within this warp loops

Algorithm 4.2 GPU Charge Assign

```
for all $ParticleBin \in Grid$ in parallel do
    \\ Inside the threadBlock with ID blockID
    __shared__ float $subGrid(nb_r, nb_\theta, nb_\psi)$
    for all $node \in subGrid$ in parallel do
        $node = 0$
    end for
    __syncthreads()
    for all $particle \in ParticleBin$ in parallel do
        cell = $particle.cell - ParticleBin.origin$
        for all $node \in cell$ do
            atomicAdd($subGrid(cell, node)$, weight($node$))
        end for
    end for
    __syncthreads()
    \\ Write block results to global memory
    for all $node \in subGrid$ in parallel do
        atomicAdd($Grid(blockID, node)$, $subGrid(node)$)
    end for
end for
```

over the 8 nodes that bound the cell that contains the particle being processed. The nodes reside in a shared memory array, and are updated with the weighted particle data atomically. Once all of the nodes for a given particle have been updated the thread will retrieve a new particle from global memory. This process is repeated by all of the threads in the block until every particle in the particle bin has been processed. Once all of the particles have been processed the block then atomically updates the nodes in global memory with the values stored in shared memory.

The atomic operations in this algorithm lead to some very interesting time complexity behavior. In essence this algorithm is being executed on a machine with 32 processors. The time complexity of this scenario is $\mathcal{O}(\frac{c}{p})$, where c is constant and p is the number of available processors. When two processors attempt to atomically update the same memory address, one of the processors must wait until the other is finished. This means that one processor is effectively lost for a 1-way conflict.

The mean number of n-way atomic conflicts N in a warp over a sub domain of size G, and the execution time T is given by:

$$N = \frac{31!}{(31-n)!G^n} \qquad T(n) \propto \frac{c}{32-n} \tag{4.2}$$

This means that the total time complexity of this algorithm with respect to the sub domain size G is:

$$T(G) \propto c \cdot \sum_{n=1}^{31} \frac{1}{32-n} \frac{31!}{(31-n)!G^n} \tag{4.3}$$

This behavior can be seen clearly in 5-10. This algorithm on the GPU can perform the particle push up to 200x faster than the CPU version of the charge assign. However, this algorithm relies on the particle data being ordered spatially, which contributes to the run time. The method used to maintain an ordered particle list on the gpu will be discussed in the following section.

4.4 Particle List Sort

As previously mentioned in section 4.3 an ordered particle list must be maintained in order for the charge assign to be fast. The particle list sort, algorithm 4.3 consists of three distinct subroutines, populating the key/value pairs, sorting the key/value pairs, and finally a payload move.

Algorithm 4.3 Particle List Sort Overview

```
Populate_KeyValues(Particles, Mesh, sort_keys, sort_values)
thrust::sort_by_key(sort_keys,sort_keys+nptcls,sort_values)
Payload_Move(Particles, sort_values)
```

This method of maintaining particle list order was chosen because it is a good balance between simplicity and performance. An additional benefit of this routine is that it uses the sort from the thrust library, which is maintained by NVIDIA.

4.4.1 Populating Key/Value Pairs

The first step in sorting the particle list is ensuring that the key/value pairs needed by the sorting routine are populated. The sorting key for a particle is the index of the particle bin that the particle belongs to. Sorting values are simply the position of the particle in the unsorted list.

Calculating the particle bin index, or binid, begins with calculating the mesh cell that the particle resides in. This cell described by coordinates i_r, i_θ, i_ϕ. The coordinates of the particle bin that a given cell resides in is given by:

$$ib_r = \frac{i_r}{nb_r}; \qquad ib_\theta = \frac{i_\theta}{nb_\theta}; \qquad ib_\phi = \frac{i_\phi}{nb_\phi} \tag{4.4}$$

The resulting block coordinates are then hashed using a z-order curve described in appendix A to give the binid. Each thread calculates the binid's for several particles and stores them in the sort_keys array. Once a thread has calculated the binid for a particle it also stores the index of that particle as an integer in the sort_values array.

4.4.2 Sorting Key/Value Pairs

The key/value pair sorting is done using the thrust library sort_by_key template function. This function is provided by NVIDIA with CUDA. The thrust sort is a radix sort that has been optimized for NVIDIA GPUs[9]. The snippet of the sort code used in sceptic3Dgpu is shown in figure 4-2.

4.4.3 Payload Move

The payload move is responsible for moving all of the particles from their old locations in memory to the new sorted locations. The idea is simple, each thread represents a slot on the sorted particle list. Threads read in an integer, the particleID, from

```
// wrap raw device pointers with a device_ptr
thrust::device_ptr<ushort> thrust_keys(binid);
thrust::device_ptr<int> thrust_values(particle_id);

// Sort the data
thrust::sort_by_key(thrust_keys,thrust_keys+nptcls,thrust_values);
cudaDeviceSynchronize();
```

Figure 4-2: Thrust Sort Setup and Call

the values array that was sorted using the binid's. This integer is the location of a given threads particle data in the unsorted list. Data at index particleID is read in, and stored in the new list at index threadID. While the idea is simple, this algorithm would require a completely separate copy of the particle list, a lot of wasted memory. However, since the particle list is set up as a structure of arrays, there is something that can be done to significantly reduce the memory requirements. The method, outlined in algorithm 4.4 reorders only a single element of the particle list structure at a time.

Algorithm 4.4 GPU Payload Move

```
for all $member \in XPlist$ do
  float* idata = member
  float* odata = XPlist.buffer
  reorder_data(odata,idata,particleIDs)
  member = odata
  buffer = idata
end for
```

Essentially the idea is that a great deal of time and memory can be saved by statically allocating a "buffer" array that is the same size as each of the data arrays. During the payload move each data array is sorted into the buffer array. Some pointer magic is performed, the old buffer array becomes the new data array, and the old data array becomes the buffer for the next data array. For sceptic3Dgpu this implementation of the payload move only increases the particle list size by about 8.6%.

4.5 Poisson Solve

4.6 Particle List Advance

Moving the particles on the grid is fairly straightforward. The process starts with determining the acceleration of the particle. This is calculated by interpolating the potential, ϕ, from the spherical mesh using the same methods as the cpu code. The new position of the particle is simply $\vec{x}' = \vec{x} + \vec{v}\Delta t + \frac{1}{2}\vec{a}\Delta t^2$. A more detailed description of the basics of the particle advance can be found in reference **??** section 3.1.2.

While the implementation of the basic physics of the particle advance remains the same, there were several interesting issues. Quickly determining whether a particle has crossed one of the domain boundaries, contributing to diagnostic outputs, and handling reinjections were the main issues.

4.6.1 Checking Domain Boundaries

In order to correctly contribute to the diagnostic outputs, the location where a particle left the domain must be known. This means that the process or checking whether or not a particle has left the grid must also calculate the position of the particle when it crossed the boundary. This is handled differently for the inner and outer boundaries.

The outer boundary is fairly straightforward. A particle has left the domain if the radial position of the particle r is greater than the maximum radius of the domain r_{max}. The

4.6.2 Diagnostic Outputs

4.6.3 Handling Reinjections

Once it has been determined that particles have left the grid, new particles must be reinjected to replace them. In the serial version of the code this is handled by simply calling a reinjection subroutine that determines the new particle's position and velocity. Once the new position and velocity has been found the particle is moved for the remainder of the time step, and be replaced by a new particle if the reinjected particle leaves the domain.

Performing reinjections in this manner on the GPU would introduce very large divergences in warp execution as well as very uncoalesced memory accesses. Eliminating the warp divergences requires that all of the threads in a warp be operating on reinjected particles. Reducing the uncoalesced memory accesses would require that all of the reinjected particles be adjacent in memory. Since we already have an object with methods that can move a list of particles and handle reinjections, all we really need is some method by which we can efficiently and reversibly "pop" a subset of the particles in the main list to a secondary list. From there we can perform the particle advance on the secondary list, and place the results back in the empty particle slots in the main list. The resulting advancing algorithm is as follows:

Compacting some subset of a parent list is a fairly easy parallel operation called stream compaction.

Stream compaction is a process by which a random subset of a list can be quickly copied to a new list in parallel. It only works for some binary condition, such as an array of length nptcls, where each element is 1 for particles that have left the domain, and 0 for all others. For each 'true' element taking the cumulative sum of all proceding elements yields a unique number that can be used as an index in a new array.

On top of this, there are several different methods for calculating the positions

Algorithm 4.5 Particle Advancing Algorithm

```
// Update The particle positions and check domain boundaries
GPU_Advance($particles, mesh, Exit\_Flags$)

Prefix_Scan(Exit_Flags)

nptcls_reinject = Exit_Flags[nptcls-1]

if nptcls_reinject $> 0$ then

    reinjected_particles.allocate(nptcls_reinject)

    Stream_Compact(particles $\subset$ exited $\rightarrow$ reinjected_particles)

    // Recursivley call the Advance on the reinjected particles
    reinjected_particles.advance()

    Stream_Expand(reinjected_particles $\rightarrow$ exited $\supset$ particles)
end if

return
```

and velocities of reinjected particles, and reproducing each of those methods on the GPU would require a large amount of effort for very little gain.

Chapter 5

Performance

Unless otherwise specified the following tests were performed using two CPUs or two GPUs, with MPI as the interface between multiple threads. The machine specifications are as follows:

- CPU: Intel Core i7 930 @ 2.8GHz.
- Memory: 12GB (3 x 4GB) DDR3 - 1333MHz ECC Unbuffered Server memory.
- GPUs: 2x EVGA GeForce GTX 470 1280MB, 607 MHz / 1215 MHz, Graphics / Processor Clock.
- Motherboard: ASUS P6T7 WS Cupercomputer Intel x58.

Typical total[1] speedups on this setup are on the order of 40x. A detailed breakdown of the run times per particle per time step and the speedup achieved by the GPU can be seen in table 5.1. This run was performed on 2 GPUs with 17 million particles per GPU and a grid size of 64^3.

The initial results indicate that a very high speedup was achieved for the charge assign and particle advance routines. It should also be noted that ordering the particle data allows for an incredibly fast charge assign. After accounting for the time that it

[1] Total run time includes MPI reduces and various other subroutines that were not ported to the GPU

Component	CPU (ns)	GPU (ns)	Speedup
Sort	0	1.428	-
Charge Assign	150.265	0.577	260x
Charge Assign & Sort	150.265	2.005	75x
Poisson Solve	40.347	3.045	13x
Particle Advance	188.177	2.475	76x
Total[1]	380.635	8.695	44x

Table 5.1: CPU and GPU Runtime comparison for 2 GTX 470's vs an Intel(R) Core i7 930 Test was performed using 2 MPI threads handling 17 million particles each on a 64^3 grid.

takes to sort the particle list, the speedup is a more modest 75x. In other codes the primary concern has been how to quickly and efficiently keep the particle list sorted. The results in table 5.1 indicate, that with the latest thrust sort, that speeding up the particle list sort is no longer a major issue. The sort step could be improved by taking into account problem specific properties of a given pic code, but considering the ease of implementation and generality of the thrust sort, it is unlikely that developing an optimized problem-specific sorting routine would really be worth it.

We also compared the performance between single and double gpu cards, as well as performance on different CPU architectures. Table 5.2 shows the run times for the CPU and GPU on a machine with 2x Intel(R) Xeon(R) CPU E5420 @ 2.50GHz and 1x NVIDIA GeForce GTX 590. The GTX 590 is a double GPU card with 2 x 512 processing cores clocked at 630 MHz, and 2x 1.5 GB ram.

Component	CPU (ns)	GPU (ns)	Speedup
Sort	0	1.272	-
Charge Assign	312.210	0.802	389x
Charge Assign & Sort	312.210	2.075	150x
Poisson Solve	637.349	5.393	118x
Particle Advance	391.335	2.325	168x
Total[1]	1352.461	12.958	104x

Table 5.2: CPU and GPU Runtime comparison for a GTX 590 vs an Intel(R) Xeon(R) CPU E5420. Test was performed using 2 MPI threads handling 17 million particles each on a 64^3 grid.

5.1 Particle list size scan

The following tests were performed to explore the dependence of sceptic3Dgpu's runtime on the total number of particles in the simulation for two standard grid sizes. Figure 5-1 was performed on a 128x64x64 grid, and figure 5-2 was performed on a 64x32x32 grid. Since the run times for the GPU and the CPU vary by such a large degree, a comparison between the two architectures is represented by the speedup factor, $\tau_{\text{cpu}}/\tau_{\text{gpu}}$. The speedup factor as a function of the total number of particles is shown in 5-3.

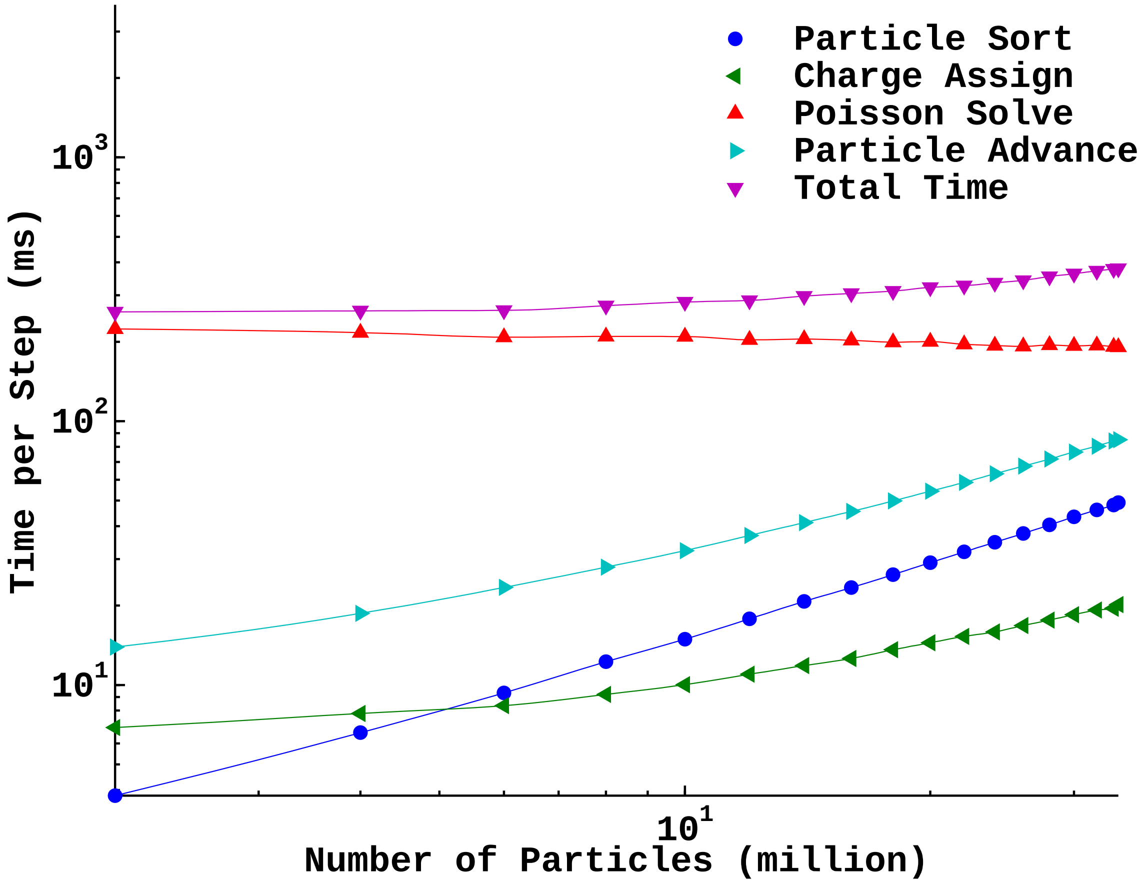

Figure 5-1: Number of Particles Scan on a 128x64x64 grid

In the case of the 128x64x64 grid the poisson solve is by far the most expensive computation for all ranges of particles. For a smaller grid, 64x32x32, the poisson solve dominates for fewer than 15 million particles, but drops below the particle advance

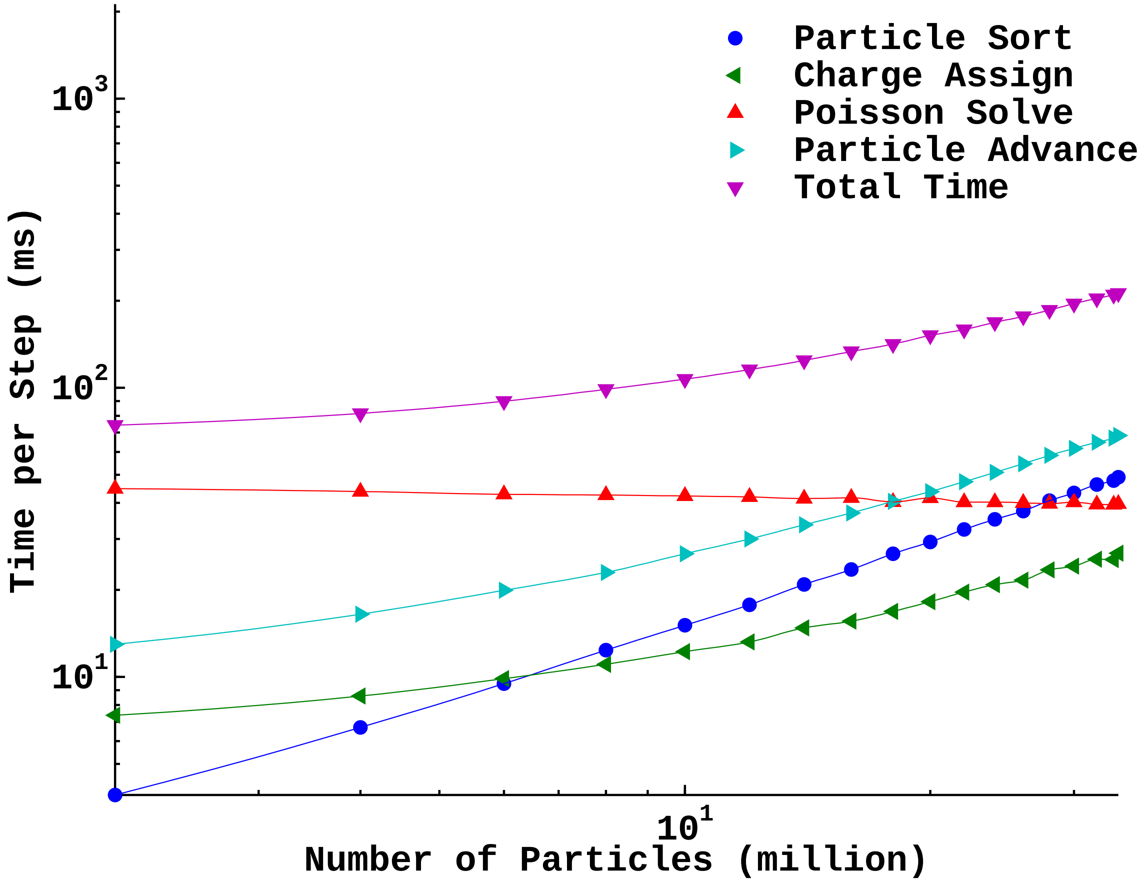

Figure 5-2: Number of Particles Scan on a 64x32x32 grid

and particle sort for more than 15 million particles. Perhaps the most interesting behavior is best observed in the speedup factor, figure 5-3, which shows a very steep rise in the speedup factor below 10 million particles. This behavior indicates that anything fewer than 10 million particles will not saturate the gpu. This behavior can also be seen in the figures 5-1 and 5-2 by the fact that the particle advance, charge assign, and total time converge to linear behavior at large numbers of particles.

A second interesting characteristic is the fact that the speedup factor curves do not fully flatten out between 10 million particles and 30 million particles. This is due in part to some small cpu costs within these routines, namely host-device transfers of data that does not scale with the number of particles. For the GTX 470 with 1280 MB of memory 17 million particles is about the limit for a single gpu. Looking at figure 5-3 it is not unreasonable to conclude that with an even larger performance boost

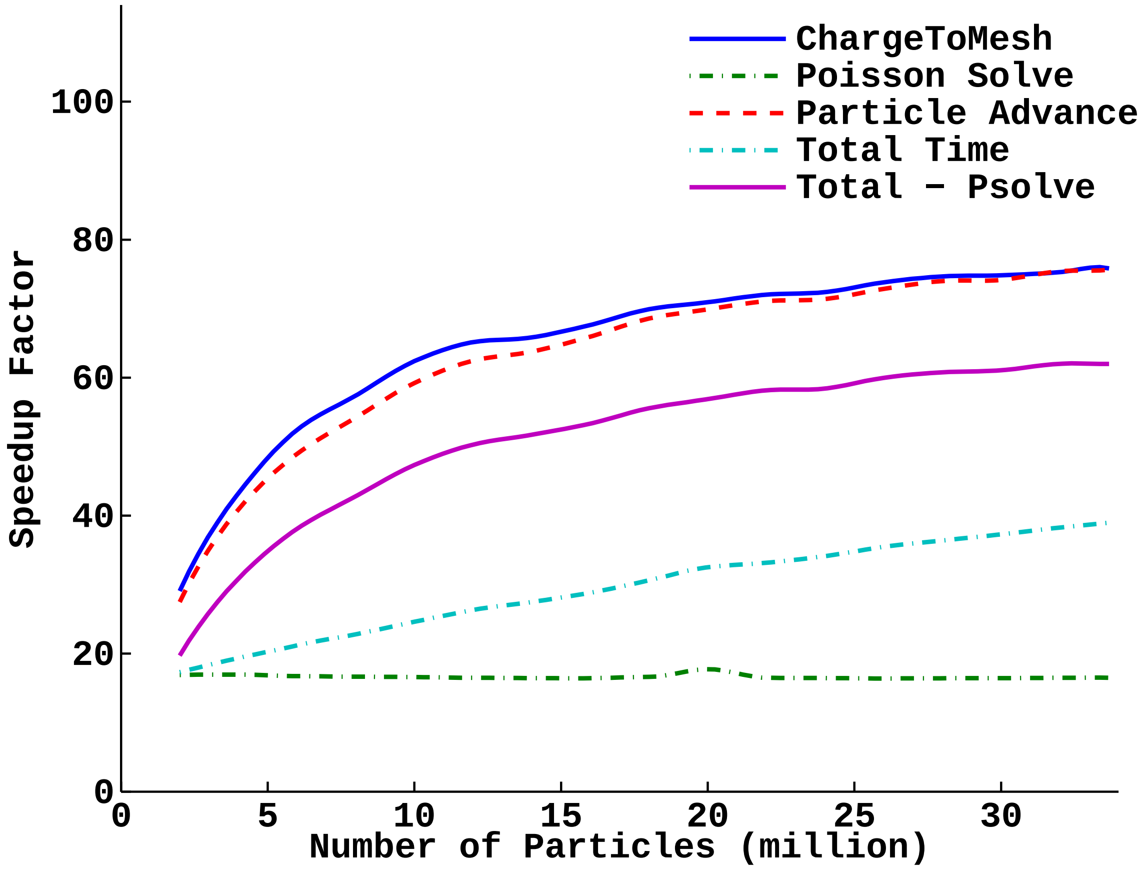

Figure 5-3: Speedup factor Number of Particles Scan on a 128x64x64 grid

can be attained simply by increasing the amount of available device memory.

5.2 Grid Size scan

So far the results indicate that the GPU is very good at moving the particles and writing the density array. In fact the GPU is so good at this that it is wasteful to not run as many particles on the gpu as physically possible. This brings us to the second main parameter of interest, the grid size. Generally speaking we would expect to see three of the subroutines display scaling with gridsize, but through different mechanisms. The poisson solve should scale roughly linearly with the number of grid elements, while more subtle scalings are dominant for the charge assign and particle advancing routines.

5.2.1 Absolue Size

In order to get a reasonable idea of how sceptic3Dgpu scales with grid size three sweeps of the grid size parameter were performed using 8, 16, and 34 million particles. There are two separate plots for each particle number due to the fact that for large grid sizes the number of bins must be increased in order to account for shared memory size restrictions. Since some of the scalings for the particle advance and charge exchange depend primarily on the number of elements per bin and not the absolute grid size plotting the results for 8^3 bins and 16^3 bins would be misleading.

The primary routine of interest here is the poisson solve, which takes roughly $\sqrt[3]{G}$ iterations of operations that are roughly $\mathcal{O}(G)$, where G is the total number of grid elements. This scaling can be seen clearly in figures 5-4 through 5-9. However, much like the number of particles scaling of the particle advance and charge assign, there is a region in which the GPU poisson solver is not saturated and beats the normal scaling. Once the GPU is saturated the poisson solve behaves as expected, scaling roughly linearly with the total number of grid elements.

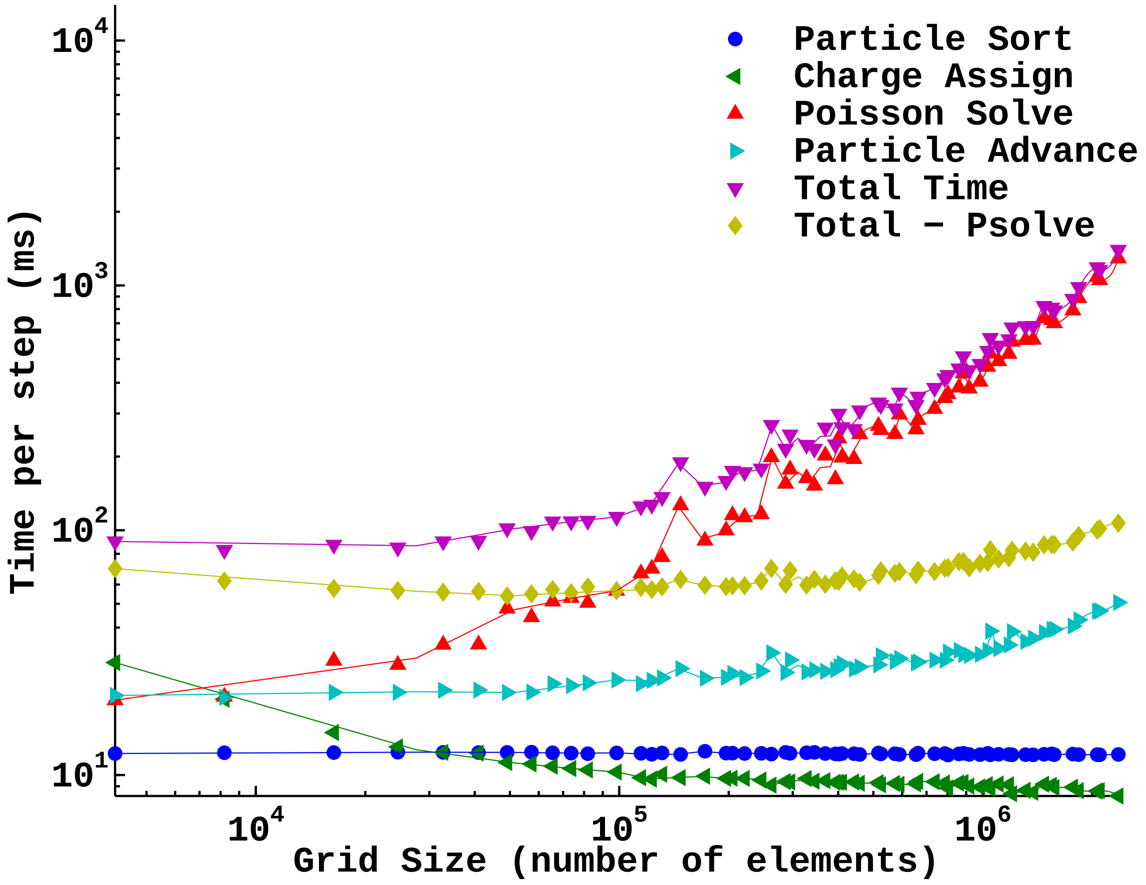

Figure 5-4: Gridsize Scan with 8 million ptcls, and 8^3 bins

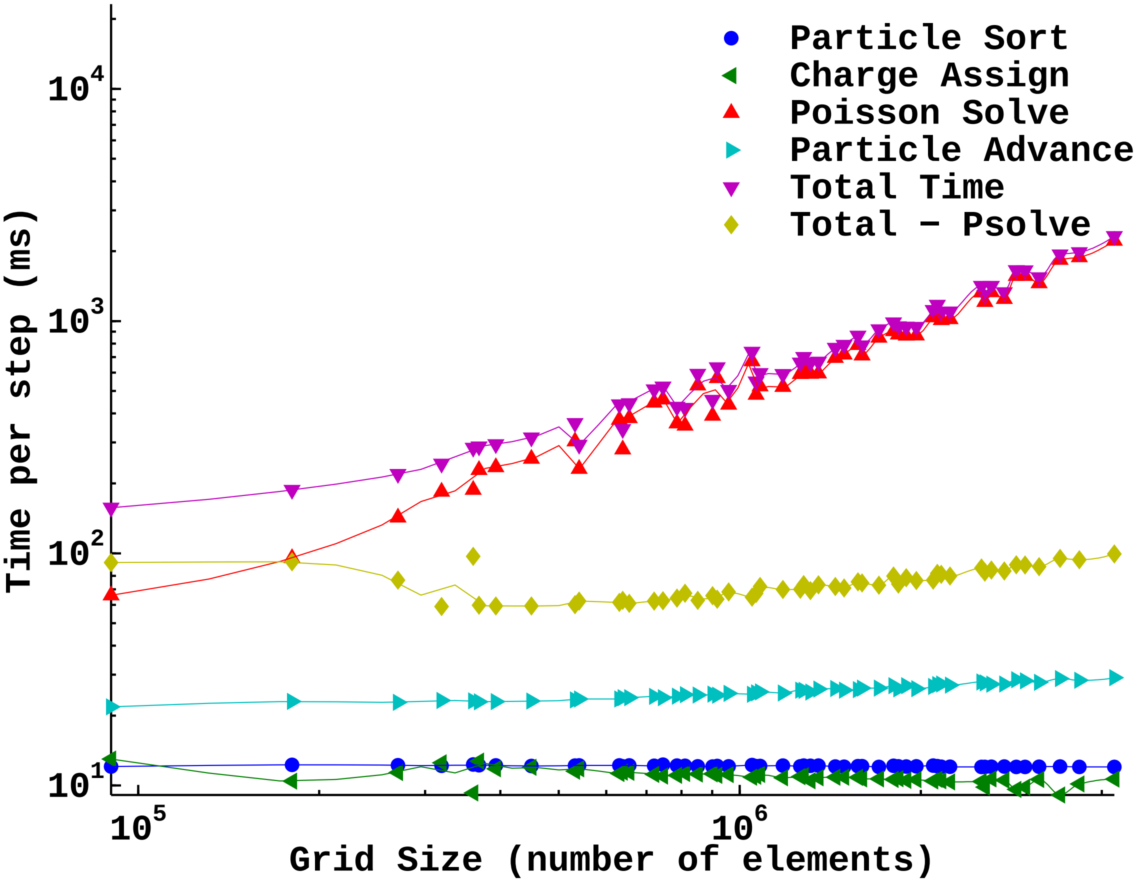

Figure 5-5: Gridsize Scan with 8 million ptcls, and 16^3 bins

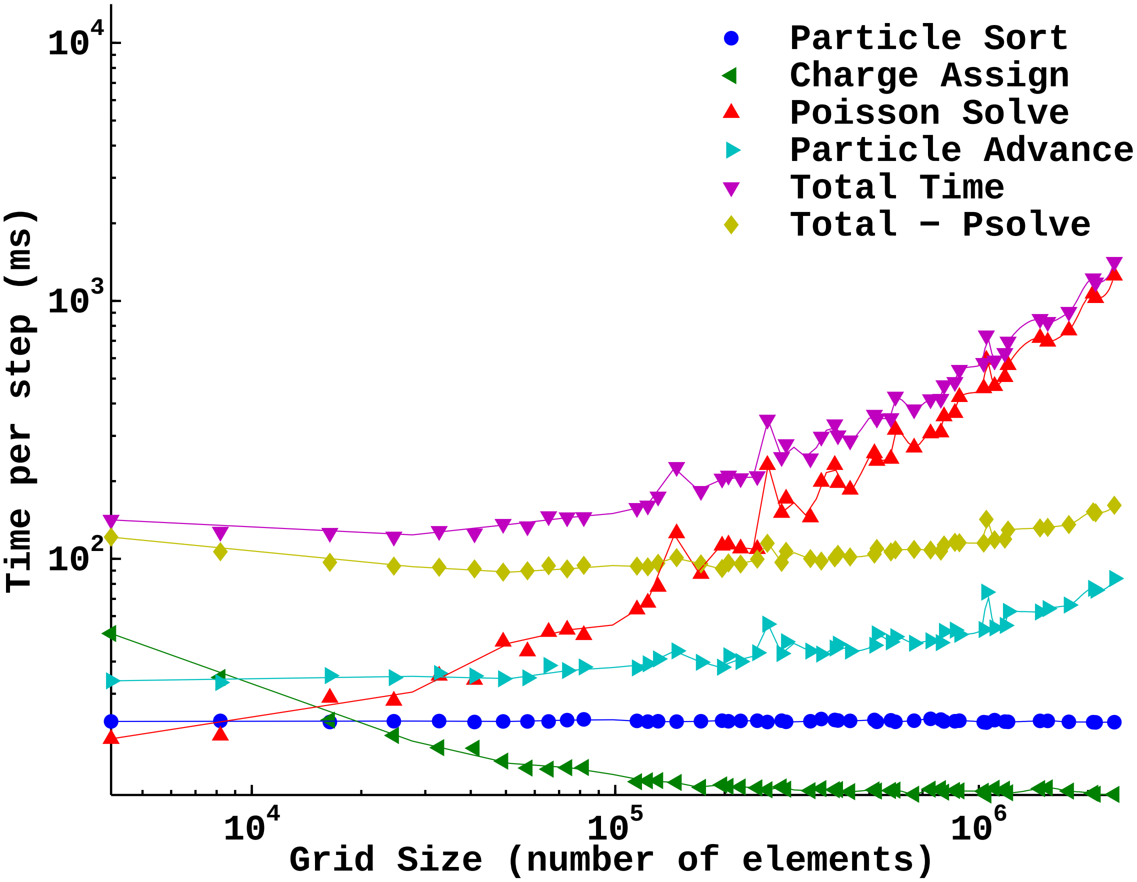

Figure 5-6: Gridsize Scan with 16 million ptcls, and 8^3 bins

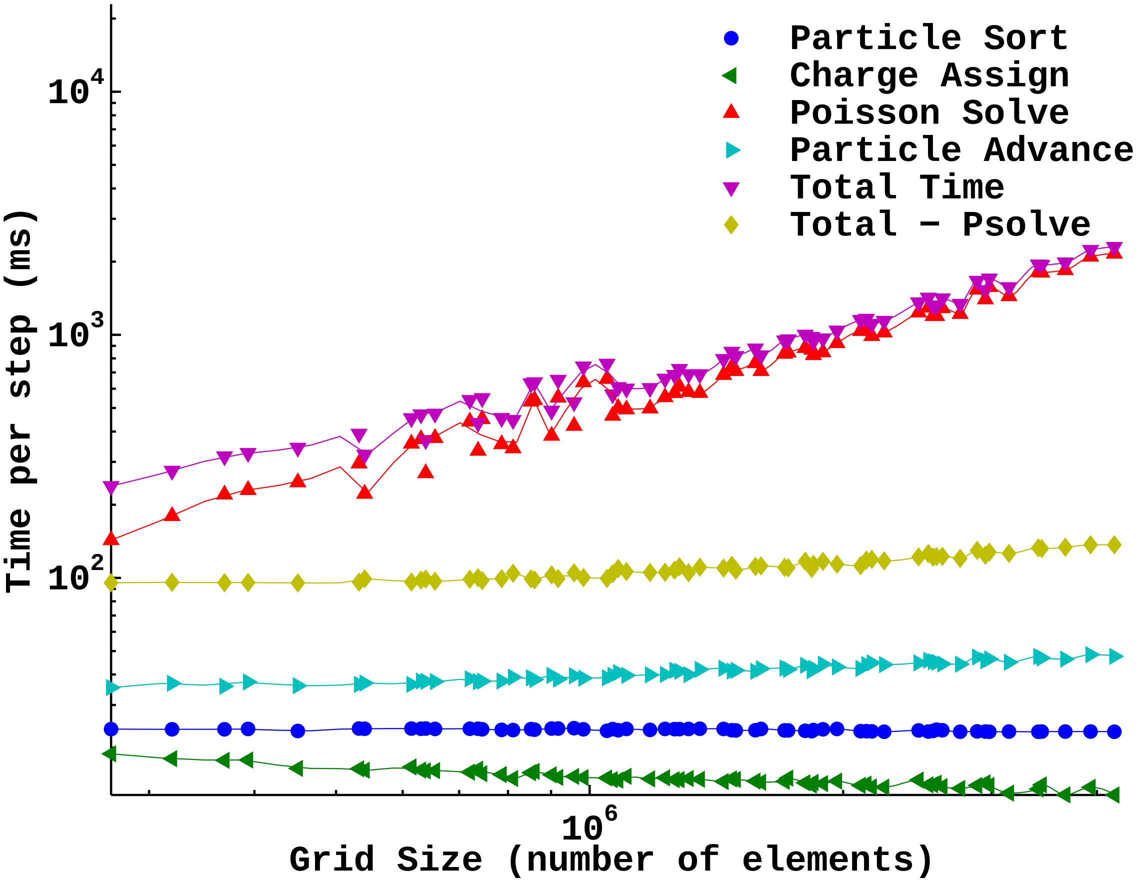

Figure 5-7: Gridsize Scan with 16 million ptcls, and 16^3 bins

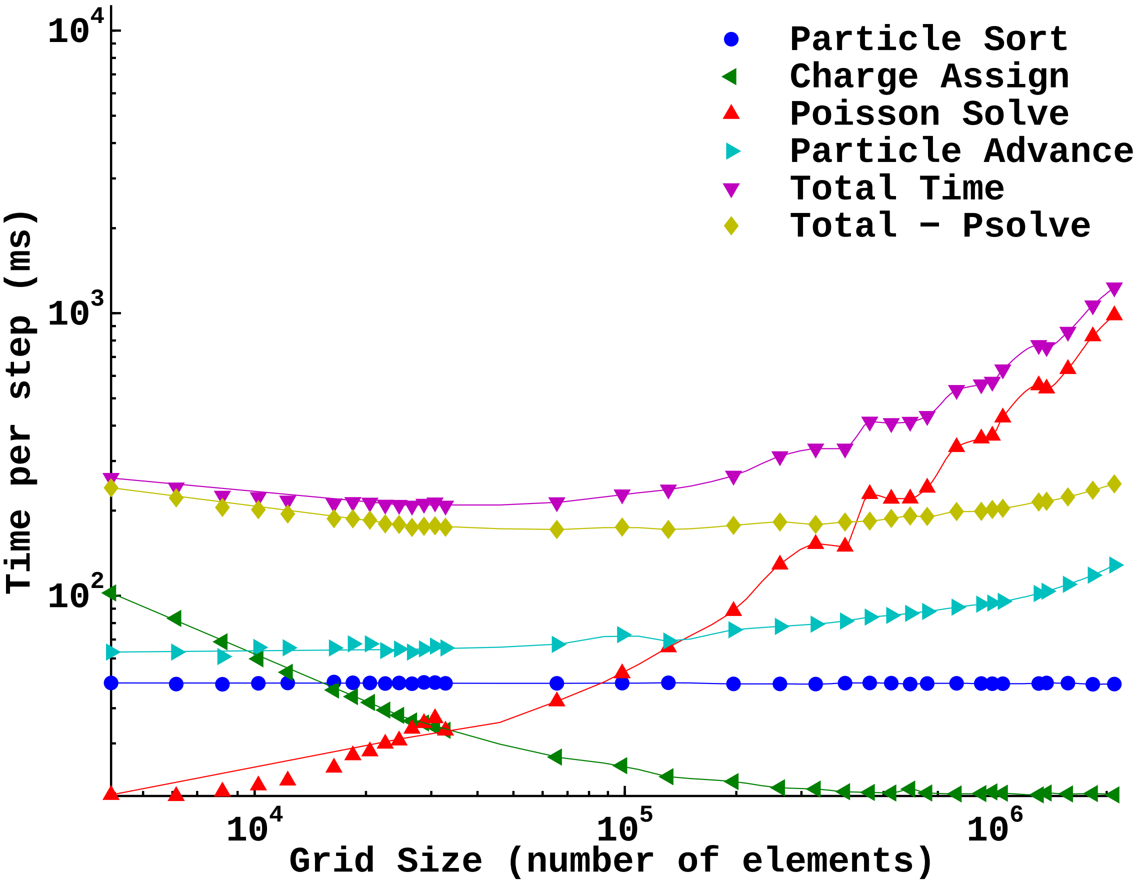

Figure 5-8: Gridsize Scan with 34 million ptcls, and 8^3 bins. Note how when the contribution from the poisson solve is removed there is a clear minimum at about 10^5 elements.

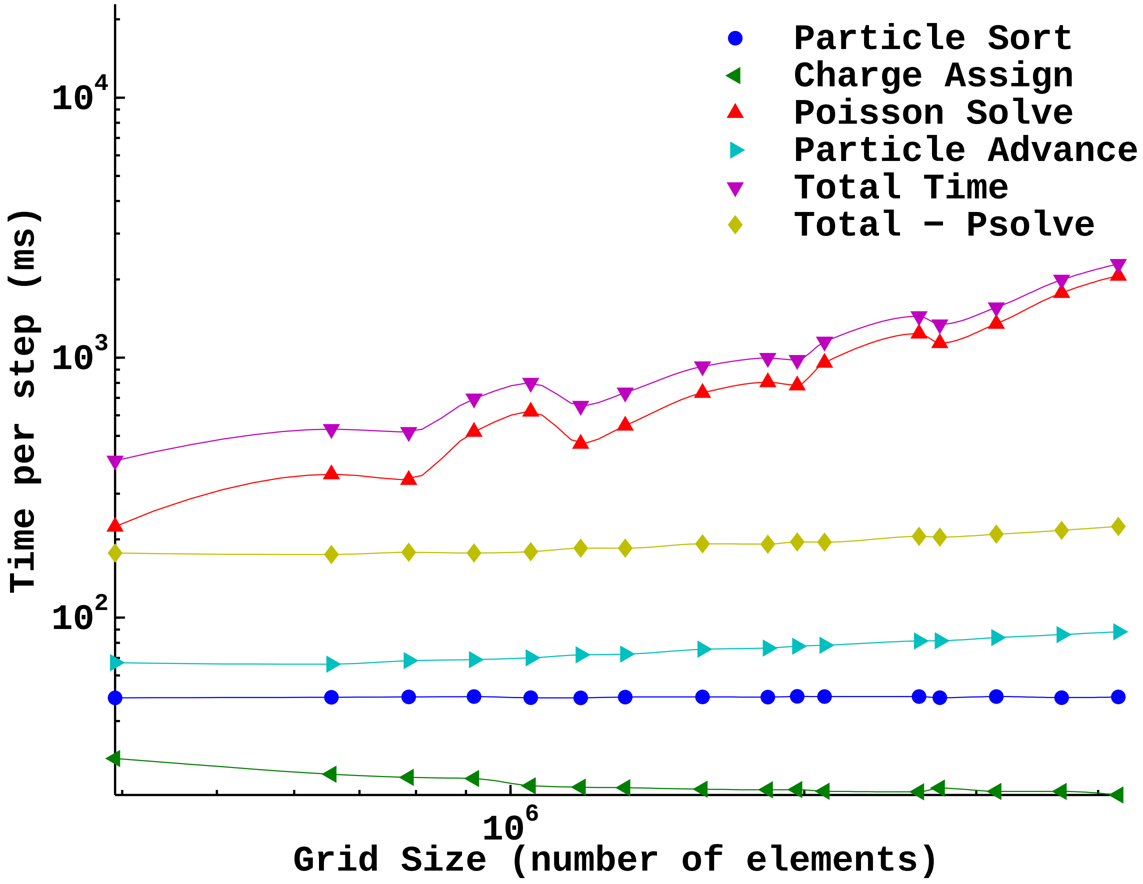

Figure 5-9: Gridsize Scan with 34 million ptcls, and 16^3 bins

The more subtle scalings of the particle advance and charge assign can be seen in figures 5-4 through 5-9 in the cases where there are only 8^3 bins, but do not continue their trends for 16^3 bins. This behavior is due to the fact that these are not scalings with the absolute grid size, but rather scalings with sub-domain size.

Another point of interest is the scaling of the particle sort, note that it only scales with the number of particles and is completely independent of grid size. One might expect to find some small scaling based on the distance that particles have to be moved during the sort stage, or that with fewer sub-domains the radix sort would have fewer digits to process, but this is not the case. This means that some improvement can be made to the sort, namely using the number of bins as the upper limit on the bits for the radix sort to process. Hopefully this kind of feature will be available in future releases of the thrust library.

5.2.2 Threadblock Sub-Domain Size

In chapter 4 we discussed the scaling of both the particle advance and the charge assign subroutines with grid size. Smaller grids should lead to more atomic conflicts in the charge assign and thus longer run time. On the other hand, for the particle advance smaller grids mean that a larger fraction of the grid can be stored in cache, which leads to fewer global memory accesses. Taking these two effects together, we should see a clear minimum in the data. Figure 5-10 shows the time per step of each routine vs the size of the sub-domain.

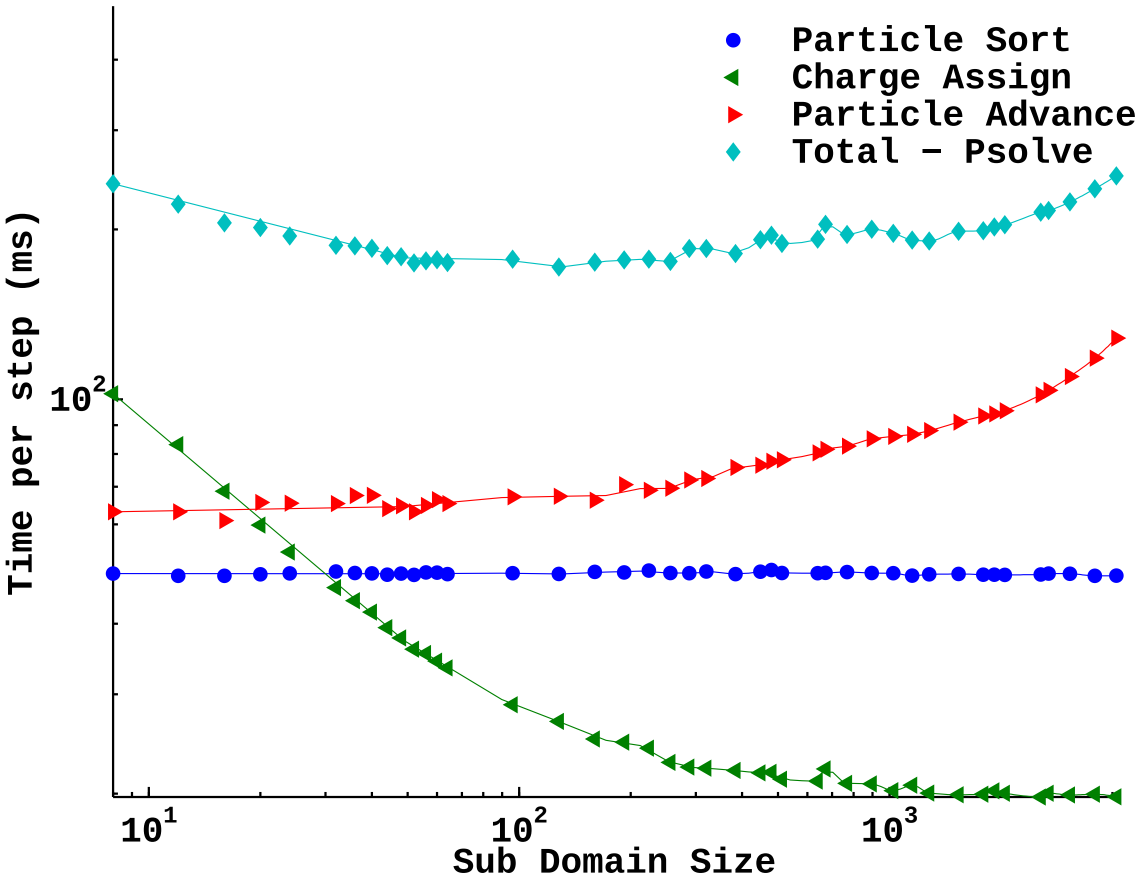

Figure 5-10: Sub Domain Size scan, also known as bin size, for 34 million particles. Note the minimum in the total - psolve run time.

5.3 Kernel Parameters Scan

Chapter 6

Conclusion

Bibliography

[1] Paulo Abreu, Ricardo a. Fonseca, João M. Pereira, and Luís O. Silva. PIC Codes in New Processors: A Full Relativisitic PIC Code in CUDA-Enabled Hardware With Direct Visualization. *IEEE Transactions on Plasma Science*, 39(2):675–685, 2011.

[2] Heiko Burau, Renée Widera, Wolfgang Hönig, Guido Juckeland, Alexander Debus, Thomas Kluge, Ulrich Schramm, Tomas E Cowan, Roland Sauerbrey, and Michael Bussmann. PIConGPU : A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. *October*, 38(10):2831–2839, 2010.

[3] Viktor K. Decyk and Tajendra V. Singh. Adaptable Particle-in-Cell algorithms for graphical processing units. *Computer Physics Communications*, 182(3):641–648, March 2011.

[4] 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.

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

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

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

[8] Xianglong Kong, Michael C. Huang, Chuang Ren, and Viktor K. Decyk. Particle-in-cell simulations with charge-conserving current deposition on graphic processing units. *Journal of Computational Physics*, 230(4):1676–1685, February 2011.

[9] NVIDIA Corporation. Thrust Quick Start Guide. Technical Report January, 2011.

[10] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. *2009 IEEE International Symposium on Parallel & Distributed Processing*, (May):1–10, May 2009.

[11] G Stantchev, W Dorland, and N Gumerov. Fast parallel Particle-To-Grid interpolation for plasma PIC simulations on the GPU. *Journal of Parallel and Distributed Computing*, 68(10):1339–1349, October 2008.