Implementation and Performance Evaluation of a GPU Particle-in-Cell Code

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 Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

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

Certified by .....
Ian H. Hutchinson
Professor of Nuclear Science and Engineering
Thesis Supervisor

Certified by .....
Kord Smith
Professor of the Practice of Nuclear Science and Engineering
Thesis Reader

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

Implementation and Performance Evaluation of a GPU Particle-in-Cell Code

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 Science and Engineering

Abstract

In this thesis, I designed and implemented a particle-in-cell (PIC) code on a graphical processing unit (GPU) using NVIDA's Compute Unified Architecture (CUDA). The massively parallel nature of computing on a GPU nessecitated the development of new methods for various steps of the PIC method. I investigated different algorithms and data structures used in the past for GPU PIC codes, as well as developed some of new ones. The results of this research and development were used to implement an efficient multi-GPU version of the 3D3v PIC code SCEPTIC3D. The performance of the SCEPTIC3DGPU code was evaluated and compared to that of the CPU version on two different systems. For test cases with a moderate number of particles per cell, the GPU version of the code was 71x faster than the system with a newer processor, and 160x faster than the older system. These results indicate that SCEPTIC3DGPU can run problems on a modest workstation that previously would have required a large cluster.

Thesis Supervisor: Ian H. Hutchinson
Title: Professor of Nuclear Science and Engineering

Thesis Reader: Kord Smith
Title: Professor of the Practice of Nuclear Science and Engineering

Acknowledgments

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

Contents

List of Figures

List of Tables

List of Algorithms

Frequently Used Terms

atomic memory operation Memory access that to a device DRAM or shared memory that is guaranteed to be seen by all threads. Atomic operations lock down a memory address, delaying accesses by any other threads until the operation is complete..

CUBLAS Fully GPU accelerated BLAS package provided with the CUDA toolkit.[16] .

CUDA NVIDIA's Compute Unified Architecture, a general purpose computing architecture for NVIDIA GPUs that uses C as a high-level programming language.

CUFFT Fully GPU accelerated fast Fourier transform package provided with the CUDA toolkit.[17].

CURAND Fully GPU accelerated, high quality pseudo-random and quasi-random number generation library provided with the CUDA toolkit.[18].

CUSPARSE Fully GPU accelerated sparse matrix linear algebra package provided with the CUDA toolkit.[19].

device A system consisting of a single GPU..

global memory GPU DRAM. Large memory space on the GPU, facilitates communication between threads in different thread blocks, and can be accessed by the CPU using cudaMemcpy() operations..

host A system consisting of the CPU and its memory space..

kernel CUDA function that is called by the CPU and executes on the GPU. When a kernel is launched multiple thread blocks are created and executed on the GPU. Kernels execute asynchronously with respect to the CPU code. Multiple kernels can exist simultaneously on the same GPU, and can be called asynchronously with memory transfers between CPU and GPU memory spaces. Kernels are also known as __global__ functions..

shared memory User managed cache on a streaming multiprocessor. Shared memory can be nearly as fast as registers and can be used to facilitate communication between threads in the same thread block..

streaming multiprocessor SIMD processing unit on an NVIDIA GPU. Each streaming multiprocessor (SM) contains a number of streaming processors, also known as "CUDA cores". The SM creates, manages, schedules and executes multiple warps at a time. Many GPUs contain multiple SMs.[14] .

thread block Grouping of threads on the GPU. Threads within the same block can communicate using the __shared__ memory space, and can be synchronized within a kernel using the __syncthreads(); function. Thread blocks exist on a single SM, and can contain anywhere from 1 to 1024 threads, which are divided up into warps for execution. For additional information please see the *NVIDA CUDA-C Programming Guide*[14] .

THRUST Parallel algorithm template library for GPUs and multi-core CPUs.[15] .

warp Grouping of threads on the GPU. One warp consists of 32 threads that execute simultaneously on a single streaming multiprocessor. A warp executes one common instruction at a time, although not all threads in the warp need act on the issued instruction..

Acronyms

CPU Central Processing Unit.

CUDPP Cuda data parallel primitives.

ECC Error-Correction Coding.

GPU Graphical Processing Unit.

MPI Message Passing Interface.

PIC Particle-In-Cell.

SM Streaming Multiprocessor.

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 among 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 modeled.

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 prohibitively 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 principles. This method is called particle-in-cell (PIC), and operates by moving particles on a potential grid and updating that potential with the new

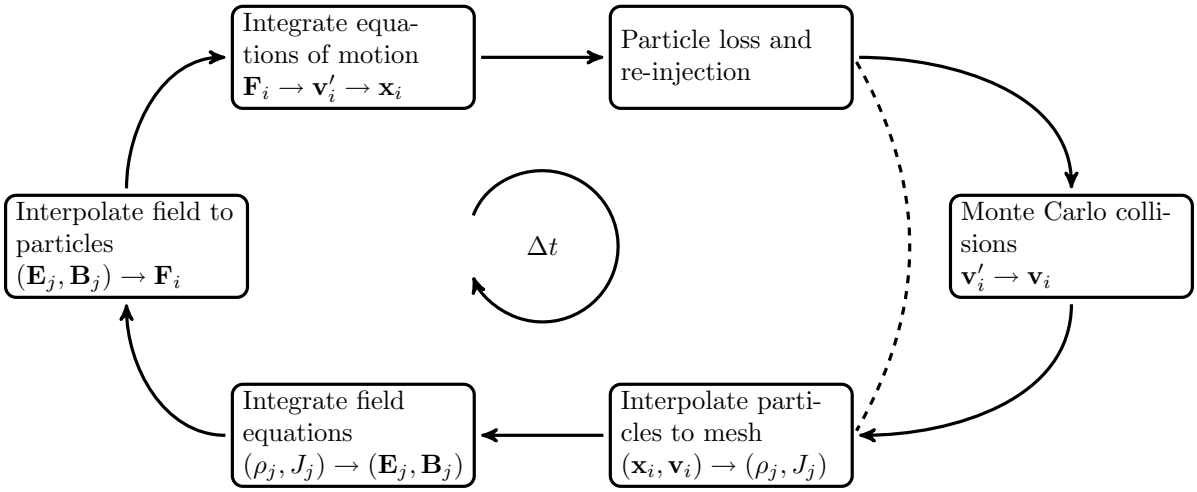

Figure 1-1: Flow schematic for the electromagnetic PIC method. Monte Carlo collisions are optional and not included in all codes.

particle density at every timestep.

The PIC method begins with an initial particle distribution, $\mathcal{P}(\mathbf{x}_i, \mathbf{v}_i)$. which is interpolated to the simulation mesh vertices in order to generate the charge and current density, $\mathcal{V}(\rho_j, \mathbf{J}_j)$. Integrating the charge and current densities defines the electric and magnetic fields, $\mathcal{V}(\mathbf{E}_j, \mathbf{B}_j)$ at the mesh vertices. From here the force, $\mathbf{F}_i$ on each particle is interpolated from $\mathbf{E}_j$ and $\mathbf{B}_j$. The force is then used to advance the positions and velocities of every particle. Following the advancing step, new particles must be reinjected to replace those that left the simulation domain. Collisions can be included after the reinjection step, although not all codes choose to include them. Finally new particle distribution function is interpolated to the mesh and the entire process starts again. Figure 1-1 shows the typical flow for this kind of electromagnetic PIC code.[26]

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 billions of particles on a single core for thousands 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 separate processor. This is of course assuming that the majority of the computational complexity lies in moving the particles and that communication 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 opposite 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 flexibility 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, forgo 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 manufacturers to combine hundreds of processors on a single chip. In order to supply enough data to keep hundreds of processors busy 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

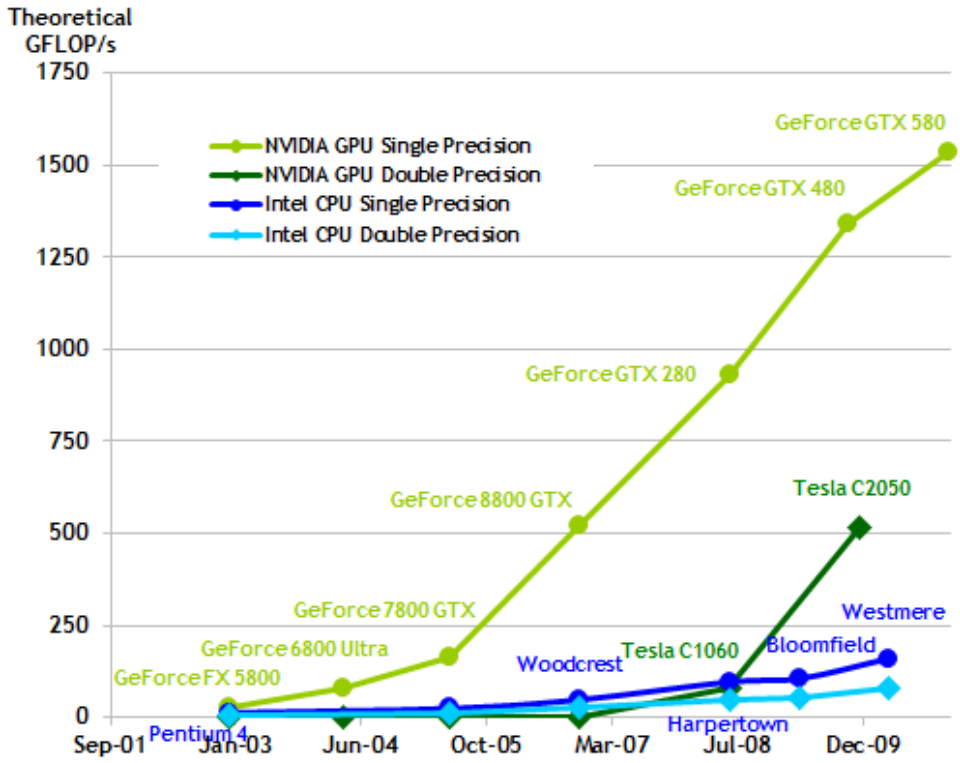

Figure 1-2: Performance comparison of GPUs vs CPUs. GPU performance is continuing to increase at a very rapid pace.[14]

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

All of their advantages 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 outlines 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 number of threads on different CPUs, and using an interface such as Message Passing Interface (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 analogous 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". When all of these systems are used together the resulting architecture has multiple levels of parallelism. The levels parallelism of a multi-GPU architecture are shown in figure 1-3.

The point here is that multiple domain decompositions must be performed in

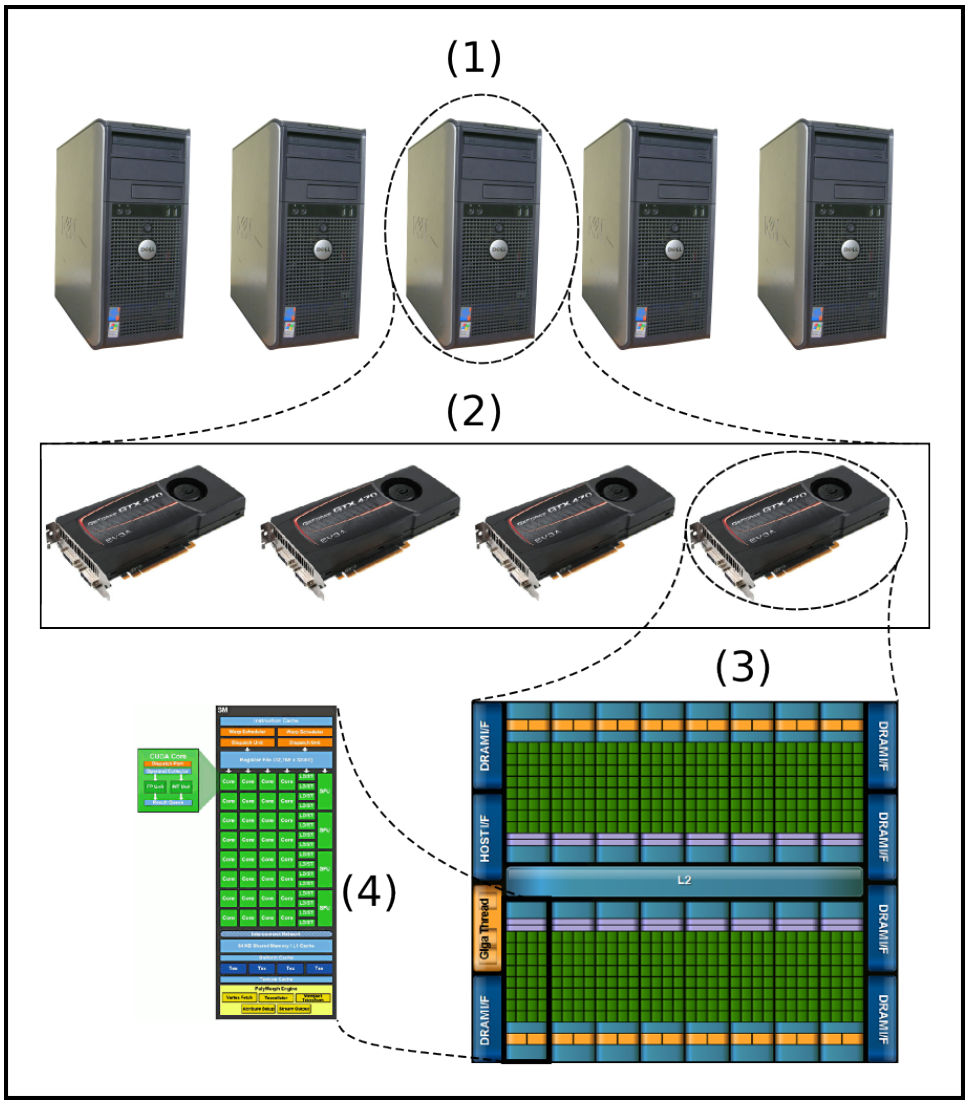

Figure 1-3: Multiple levels of parallelism. (1) Cluster of systems communicating through a LAN. (2) Multiple GPUs per system communicating through system DRAM. (3) Multiple streaming multiprocessors per GPU execute thread-blocks and communicate through GPU global memory. (4) Multiple CUDA cores per multiprocessor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

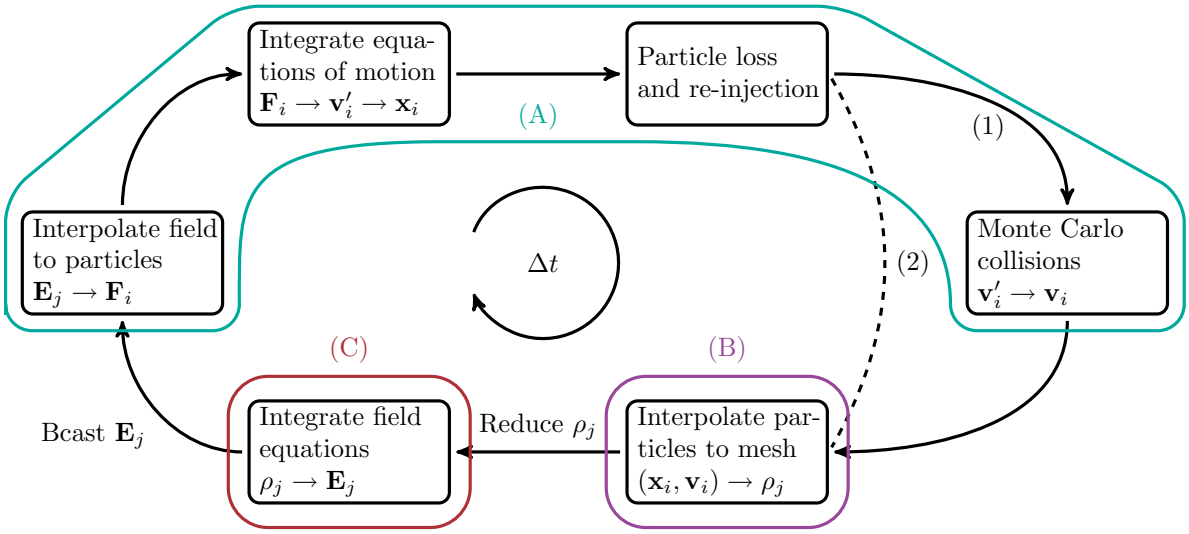

Figure 1-4: Flow schematic for the electrostatic PIC method with parallelizable steps highlighted. (1) and (2) are possible paths that the code can take depending on whether or not collisions are used. (A) Particle steps are embarassingly parallel. (B) Particle to Mesh interpolation steps are difficult to parallelize. (C) Field solving step can make use of parallelized linear algebra libraries.

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 significantly higher memory bandwidth and smaller cache of GPUs.

1.2.1 Parallelization Opportunities in PIC Codes

Most of the steps in the PIC method can be parallelized, although some steps are more difficult than others. Figure 1-4 highlights the steps of the PIC method that are parallelizeable, and groups them according to how they can be parallelized. The particle steps, group (A) in the figure, are probably the easiest steps to implement in parallel. In theory every particle could be advanced by a separate processor or thread. In the case of GPUs it is entirely feasible to assign a single thread to every particle. There are several issues that arise when implementing steps that contain a large amount of execution divergence on the GPU. The particle loss / re-injection step, and the collision step are two such subroutines in which the execution paths of particles that are being lost, re-injected, or colliding will diverge greatly from the

majority of the particles. In these cases it is best to operate on subsets of the particle list in which all particles of the subset follow the same execution path. While we have not yet discussed them, there are ways in which subsets of the particle list can be operated on efficiently without large amounts of execution divergence.

Parallelizing the particle to mesh step, (B) in figure 1-4, is a bit trickier, depending on the amount of memory available per thread. In the case of multi-CPU implementations, each thread has enough memory to store a copy of the entire grid. This thread can process a particle that exists anywhere on the grid, and map its contribution to the grid without any memory conflicts. The case is very different for GPUs, which have nowhere near enough memory to support replicating the entire grid for every thread. Therefore the key to parallelizing the particle-to-mesh interpolation step on the GPU is determining way to efficiently avoid memory conflicts, using some kind of atomic memory operation or domain decomposition.

The last step (C), the field solve, can be fairly straightforward to parallelize. The field solve involves integrating Maxwell's equations over the entire grid. In the case of electrostatic codes this is simply solving Poisson's equation. Poisson's equation is a straightforward linear equation that can be solved using a number of parallel $Ax = b$ solvers, popular ones being preconditioned bi-conjugate gradient solvers and spectral solvers.

Essentially the vast majority of the PIC algorithm is well suited for implementation on the GPU. There are some challenges to overcome, such as how to perform the charge assign, and how to efficiently handle re-injections and collisions. Currently GPU computing is in its infancy, and the solutions to these challenges are evolving rapidly. In the following section we will take a brief look at some of the solutions that others have developed to overcome these issues.

1.3 GPU PIC Code Development

Before we dive into past research into implementing PIC on GPUs it would be useful to know a little more about general purpose graphical processor (GPGPU) computing. Currently there are two frameworks for writing GPGPU programs, the Open Computing Language (OpenCL) initially developed by Apple Inc., and the NVIDIA Corporation's Compute Unified Architecture CUDA.[20, 14] Each framework has its own strengths and weaknesses. CUDA is a software environment that allows developers to use C as a high-level programming language. Developing in CUDA C is straightforward and is very well documented and supported by NVIDIA. The CUDA toolkit and SDK provided by NVIDIA include decent debugging and profiling tools as well as a large number of fully GPU accelerated libraries. These libraries include:

- THRUST (template and algorithm library)[15]
- Cula (linear algebra library)
- CUBLAS (Basic Linear Algebra Subprograms)[16]
- CUFFT (Fast Fourier Transform (FFT) Library)[17]
- CUSPARSE (Sparse linear algebra routines)[19]
- CURAND (high-quality pseudo-random and quasi-random numbers)[18]

The primary downside to CUDA is that it is restricted to NVIDIA GPUs, and cannot be used for other hardware platforms. OpenCL on the other hand supports multiple computing platforms and has been adopted by Intel, Advanced Micro Devices, Nvidia, and ARM Holdings. However, developing in OpenCL can be somewhat cumbersome compared to CUDA, especially when it comes to debugging and profiling. OpenCL also lacks mature libraries similar to those provided with CUDA. For these reasons, and the fact that the primary compute-specific cards are the NVIDA Tesla cards, most GPU PIC code development has been done using CUDA.[20, 14]

1.4 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 discussed in depth in chapter 2. One of the earliest efforts in developing plasma PIC simulations on the GPU is that of George Stantchev et al. In 2008 Stantchev et al published a paper in the *Journal of Parallel Distributed Computing*[25] outlining the development of a fast density accumulation method for PIC implementations on the GPU. Stantchev's paper was one of the first to identify the issue of parallelizing the density update and develop a very efficient solution. The solution involves defining clusters of grid cells and sorting the particle list according to which cluster they belong to. The cell clusters are small enough to fit into shared memory and updated pseudo atomically using a now obsolete thread tagging technique. The simplicity and efficiency of this method has led to its use in many other GPU PIC implementations. Stantchev also identified the issue of the particle sort step required by their density accumulation method. Developing an optimized sort has since become one of the biggest challenges for GPU PIC implementations.

Also in 2008, Dominique Aubert et al published their development of a GPGPU galactic dynamics PIC code.[2]. Aubert et al adapted their entire code for GPUs and tested several field-solving and density-accumulation techniques. The first density accumulation techniques used relied on using the radix sort provided by the CUDA Data Parallel Primitives (CUDPP) library to order the particle data and perform a histogram. The second technique used the GPU to find the nearest cells for each particle and then perform the histogram on the CPU. As for field solving techniques, GPU versions of both FFT and multi-grid Poisson solvers were used. The FFT solver made use of the CUFFT API provided by the CUDA Toolkit. The multi-grid (MG) solver was a from scratch GPU implementation of the MG solver outlined in [23]. Aubert et al achieved speed ups of 40x for the FFT and 60x for MG, and approximately 20x for the velocity and position updates, but the histogramming was on the order of 5-10 times *slower* on the GPU. They acknowledge that their histogramming method is too slow, and propose using an improved technique following that

developed by Stantchev et al.

A further step in GPU PIC codes was taken in 2010 by the simulation code PIConGPU[3], developed by Heiko Burau et al. PIConGPU was one of the first scalable GPU PIC implementations involving multiple nodes. PIConGPU is a fully relativistic, 2D electromagnetic code used to simulate the acceleration of electrons in an under-dense plasma by a laser-driven wakefield. Burau et al also identified the issue of parallelizing charge and current density accumulation on SIMD architectures and came up with a unique solution, a linked particle list. This linked particle list is one of the core features of PIConGPU and is geared towards maintaining localized data access for each kernel. In order to facilitate scalability over multiple nodes the simulation volume is domain decomposed and distributed across multiple GPUs. Each subdomain is bordered by a region of guard cells to facilitate particle transfers between subdomains. The results of this research indicated that single GPU PIC algorithms can be scaled to multi-node clusters despite high host-device communication latency.

In March, 2010 NVIDIA released the first GPUs utilizing the Fermi architecture. Fermi boasted several key improvements for scientific computing over the G80 and GT200 architectures. These improvements included [13]:

- Configurable L1 and unified L2 caches.
- 8x the double precision performance of the GT200 architecture.
- Error-Correction Code (ECC) support.
- Improved atomic memory operation performance.
- Full IEEE 754-2008 32-bit and 64-bit arithmetic floating point precision compliance.

The significant increase in double precision performance and addition of ECC support are huge wins for scientific computing on GPUs. With the launch of Fermi, work on GPU based PIC codes increased significantly. Another 2D relativistic GPU PIC code was developed by Paulo Abreu et al in late 2010.[1] Abreu et al chose an approach to the charge and current accumulation which was in some regards

opposite of that employed by Stantchev et al. This approach was based on using pseudo atomic memory operations that actually consisted of two **atomicExchange**() operations. In order to minimize atomic collisions the particle list is ordered such that the probability that two threads within a warp were operating on particles in the same cell is low. The initial particle distribution is prepared in way to minimize this probability, but this distribution can be upset throughout the simulation. If the distribution degradation reaches a certain threshold then a redistribution of the particle list takes place. This redistribution was handled by a sorting operation, using the CUDPP radix sort, and a *stride* distance. The resulting code had a relatively fast particle advance but the current deposition was rather slow. The current deposition step constituted approximately 67% of the code runtime for 1.2 million particles on a 128^2 grid.

One of the more unique GPU PIC implementations was developed by Rejith Joseph et al and published at the 2011 *IEEE International Parallel & Distributed Processing Symposium.*[11] This code, a GPU implementation of the XGC1 code differed from previous codes in two ways. First, XGC1 uses an asymmetric triangular mesh, unlike most GPU PIC codes which use symmetric rectangular meshes. Second, there are no limits imposed on particle movement, which makes the lessons taught by the code more applicable in general.

The first point, the use of a triangular mesh is interesting because most of the time a symmetric rectangular mesh is preferred on the GPU. As in other PIC codes, in order to facilitate a fast particle to grid interpolation it was important that all of the triangles that a thread-block might access fit into the limited shared memory. This limit, imposed by the size of shared memory, necessitated that the primary mesh be partitioned into smaller regions. Joseph et al tried three different partitioning strategies, density based, uniform, and non-uniform. The density based strategy partitioned the mesh such that each region had approximately the same triangle density. The uniform approach divided the mesh into large uniform rectangular regions. Lastly the non-uniform partitioning was a sort of hybrid of the uniform and density

approaches, using rectangular regions that varied in size depending on the granularity of the triangle density.[11]

The goal of these partitioning schemes was to take care of load balance on the GPU. As it turns out, the automatic load balancing provided by the GPU hardware is very good. When Joseph et al compared triangle search times between the different partitioning schemes the uniform scheme was the fastest for all numbers of thread blocks, but for very large numbers of blocks the Uniform and the Non-Uniform schemes converged to roughly the same execution time. This brings up a very interesting point that can teach us a valuable lesson; namely, the GPU tends to like very simple problems and often times increasing the complexity results in reduced performance. While complicated performance enhancing techniques work well on serial architectures, often they will only hurt you when implemented on the GPU.

The final code that we will look at was published in the *Journal of Computational Physics* by Xianglong Kong et al in 2011.[12] This code was a 2D3V fully relativistic electromagnetic code that achieved speed-up factors of 81x and 27x over the CPU runtime for cold plasma runs and extremely relativistic plasma runs respectively. This code approached the issue of maintaining an organized particle list very differently than previous codes. The approach used in this code is more akin to message passing than sorting, and will be explained in more detail in chapter 2. The performance of this code greatly depended on the fraction of particles that crossed from one sub-region into the next, or the crossing fraction η. The achieved run-time per particle-step was 4.6 ns for $\eta = 0.5\%$, and up to 9.15 ns for $\eta = 7.5\%$.

1.5 SCEPTIC3D

Now that we have discussed previous works in the realm of GPU PIC codes, it is time to introduce the code that we chose to implement a GPU version of, SCEPTIC3D. SCEPTIC3D is three dimensional hybrid PIC code specifically designed to solve the problem of ion flow past a negatively biased sphere in a uniform magnetic field. The

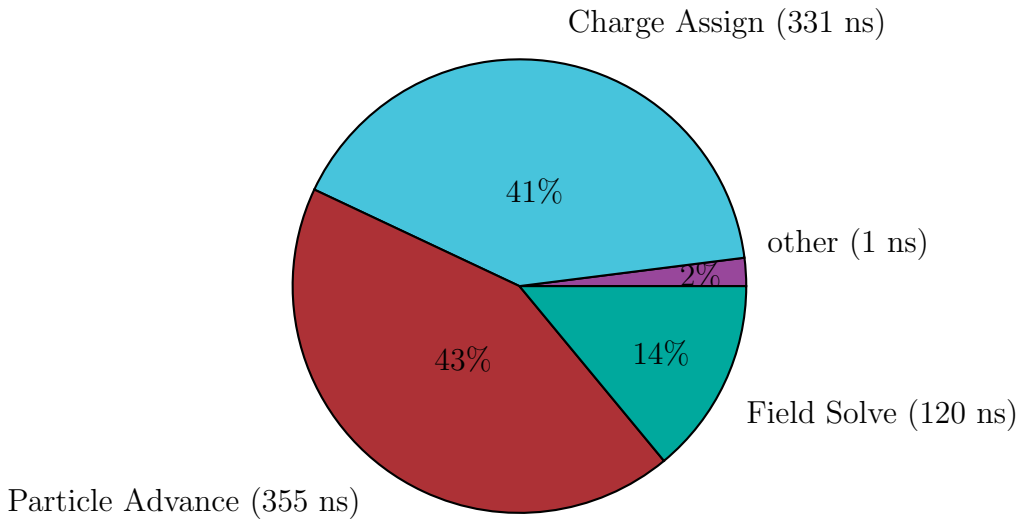

Figure 1-5: Breakdown of SCEPTIC3D runtime costs by subroutine. This is for 12.5 million particles on a 64^3 grid. Times are in ns per particle step.

current version of the code was derived from the 2D/3v code SCEPTIC which was originally written by Ian Hutchinson [9, 10, 7, 8]. The code is written in FORTRAN, and has been previously parallelized using standard MPI communication interface. Uses of SCEPTIC3D include exploring probe-plasma interactions, modeling the behaviour of particle grains in dusty plasmas, and determining the interactions between objects in space and their environment. As previously mentioned, the PIC method requires a large number of particles in order to reduce statistical noise. In the case of SCEPTIC3D, runs typically consist of 50 million particles. Moving 50 million particles on a 64^3 grid takes roughly 11 seconds per step using 4 cores of an Intel i7 930 processor. The fact that SCEPTIC3D has already been parallelized using MPI makes it far easier to develop a multi-GPU implementation.

1.5.1 CPU Code Profiling

In figure 1-5 we break down the runtime of SCEPTIC3D into the main subroutines. This figure indicates that porting the charge assign and particle advancing steps should be the main priority. Of course, if we hope to get a performance boost of more than 3x for both the particle advancing and charge assign steps, then we will have

to worry about the field solve as well. However, it should be noted that the balance between the particle and field costs depends on the number of particles per cell. In the case of figure 1-5 we used roughly 48 particles per cell, generally a larger fraction is preferable in order to reduce particle noise.

In the end the GPU implementation of SCEPTIC3D is very different from any other GPU PIC implementation. While we will focus a great deal on performance, we will also place a great deal of emphasis on implementation difficulty and applicability to future PIC implementations on the GPU. We will determine how critical the performance of the field solver is when the run time of other steps is significantly reduced. Lastly, we will look at how primarily graphics related GPU features can be utilized by PIC codes.

Chapter 2

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. Maximizing the amount of code that can be used for both the CPU and GPU versions also helps minimize the number of bugs and can help make the code easier to read. 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.

2.1 GPUPIC Sandbox

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

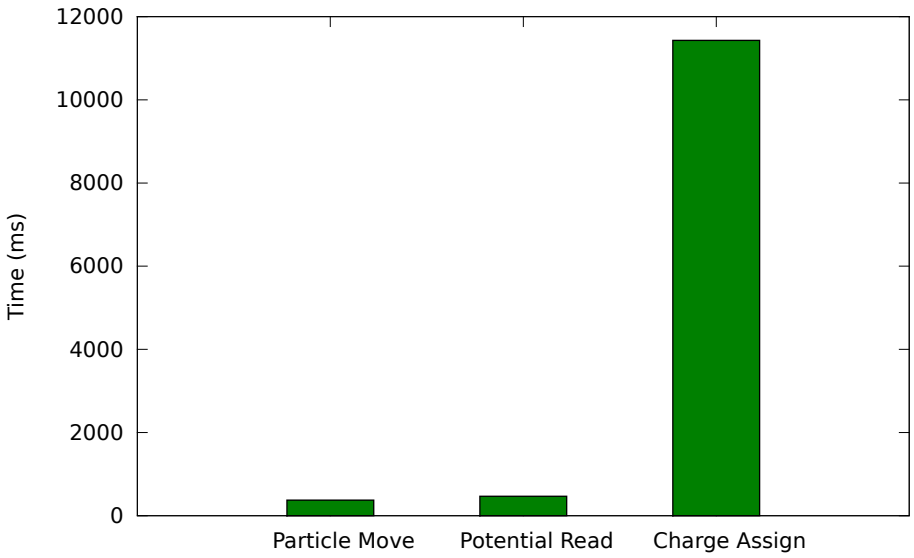

Figure 2-1: Total Execution times for 100 iterations of the key steps of the move kernel. The charge assign dominates the run time by a very large margin.

6. Repeat

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 figure 2-1.

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?

2.2 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. $\mathcal{P}(v_s)$ is a list of all particles contributing to vertex v_s and $\mathcal{V}(p_i)$ is the list of all vertices that particle p_i contributes to. Given these definitions the algorithms for the particle pull and particle push method are algorithms 2.1 and 2.2 respectively.

Algorithm 2.1 Particle Pull Method of charge deposition. From Stantchev et al. [25]

```
// Loop over the vertices first
for all vertex v_s ∈ G do
   find P(v_s)
   f(v_s) ← 0
   for all p_i ∈ P(v_s) do
      f(v_s) ← f(v_s) + K(v_s, p_i)
   end for
end for
```

As pointed out by [25] 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

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

Algorithm 2.2 Particle Push Method of charge deposition. From Stantchev et al. [25]

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

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 challenge of the charge assign is that with a completely random particle list any given particle can contribute to any element of the grid. For parallel implementations using MPI, the solution, shown in figure **??**, involves setting aside enough memory to store the entire domain for each thread. Each thread deals 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.

Applying a similar technique on the GPU would require massive amounts of memory. Memory is a valuable commadaty on the GPU, and ideally we want to use as much of it as possible for the particle list. We need to determine a way in which we only need one copy of the density array in global memory. The problem is that if two threads attempt to update the same element in the density array at the same time, such as in figure 2-3, a memory conflict occurs and only one threads contribution will be recorded.

In the sandbox PIC code we used atomic operations to prevent memory collisions

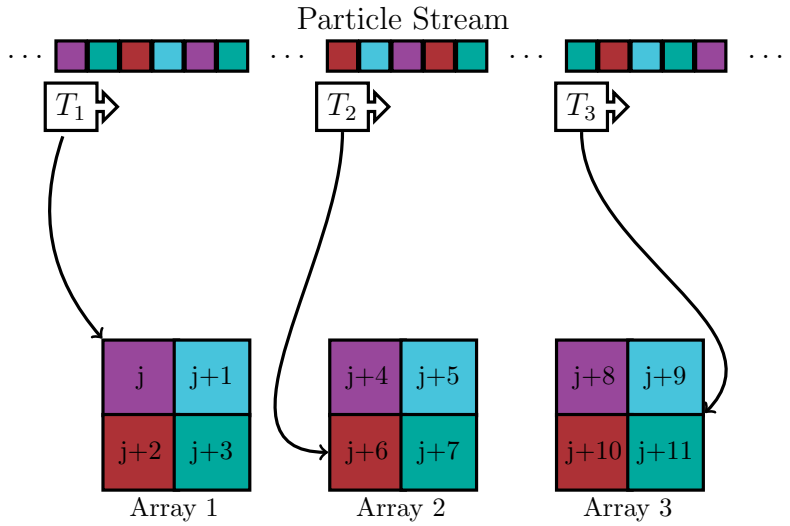

Figure 2-2: Every thread reads in some random subset of the particle list and records the contributions of its subset to a copy of the density array. Once all threads have processed their respective sub-lists a prefix sum is used to condense all of the contributions to a single array.

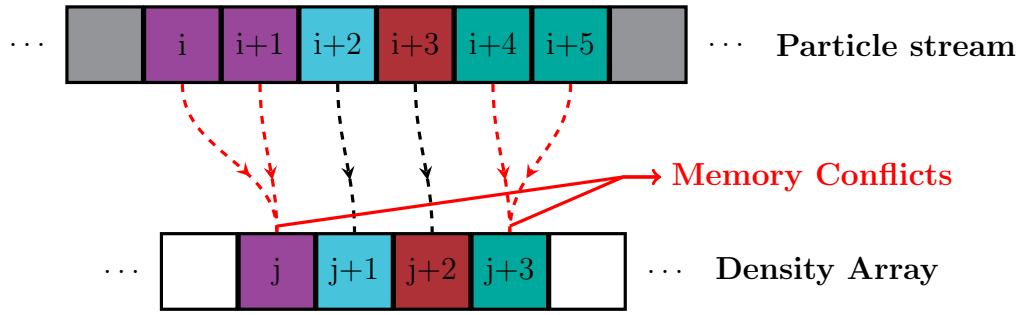

Figure 2-3: One thread is launched for every particle. Conflicts arise whenever two threads attempt to update the same element of the density array at the same time. Without atomic operations only one threads contribution would be counted. Atomic operations can be used in order to ensure that every thread's contribution to the density is accounted for. However, when a memory conflict occurs between threads using atomic operations the execution becomes serialized, significantly reducing the operation's efficiency.

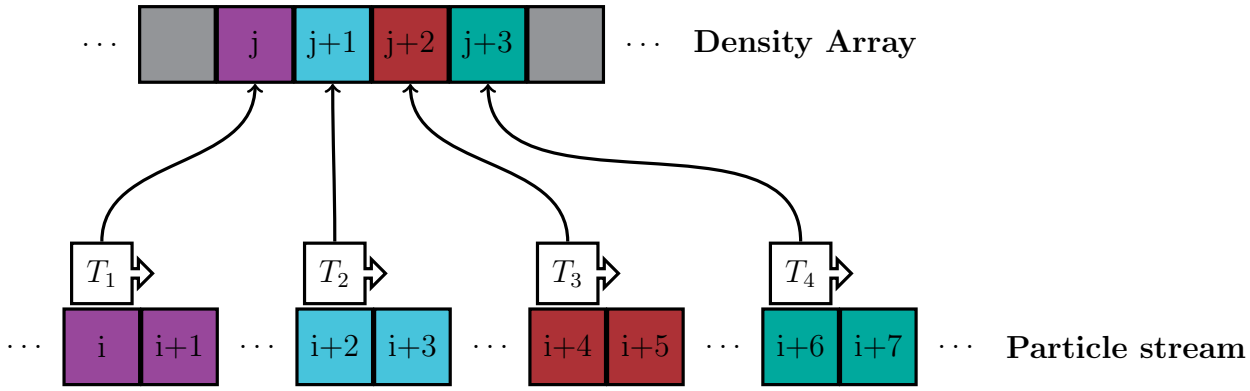

Figure 2-4: One thread per cell. Each thread is responsible for a subset of the particle list. Each subset of the particle list corresponds to a single element of the density array.

during the charge assign. Looking back at figure 2-1 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. Additionally, 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.

However, if we can ensure that a thread knows that every particle it reads in will only contribute to one element of the grid, then operation that the thread has to perform is a simple sum. Since this thread knows that every particle it sees will only contribute to a single value, each thread requires only enough memory for their single 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 2.1 without the need to to retrieve $\mathcal{P}(v_s)$ dynamically. An illustration of this method is shown in figure 2-4. 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 a threads domain. We will worry about this additional constraint later.

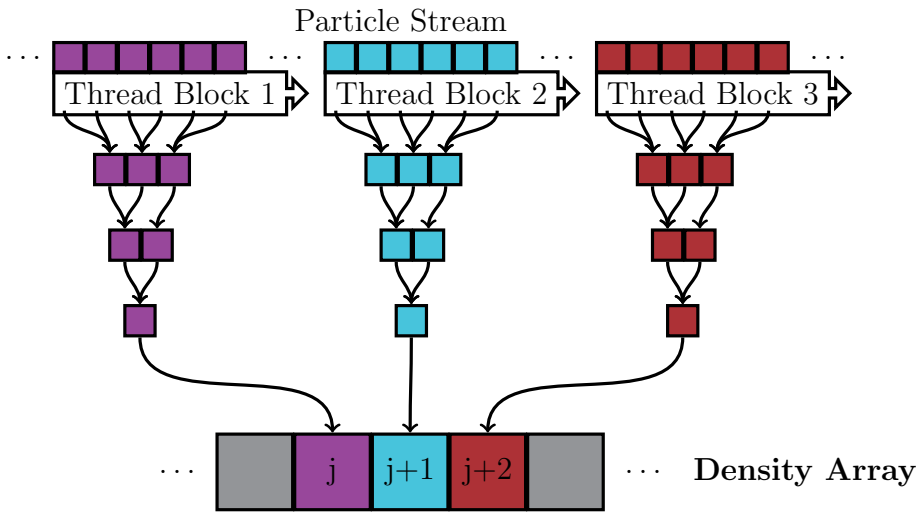

Figure 2-5: One thread block is assigned to a subset of the particle list corresponding to a single element of the density array. Each thread processes and records multiple particles in the sub-list, no particle is processed by two different threads. Threads record the contributions to a private location in shared memory. Once all particles have been processed the contributions from all the threads are reduced to a single value using a prefix sum.

Now, since the operation performed at the thread level is a simple sum there should be a way that we can add an additional level of parallelization. Further parallelization can be accomplished by replacing the sum with a parallel reduction. This method, illustrated in figure 2-5 the particles are sorted into subsets according to which cell they are contributing to. Each subset of the particle list is assigned to a thread block, or multiple thread blocks, and read into shared memory. Once the contributions from all of the particles in the subset have been read into an array in shared memory a parallel reduction is performed to condense the array of values into a single value. In this method there are zero memory conflicts as each thread reading in data from the particle list has a private memory space in which to record the values it reads. This reduction method approaches the limit of how far the particle to mesh mapping can be parallelized, in which every thread must process only a single particle. In this limit, the time complexity of accumulating the contributions of N particles to D elements is $\mathcal{O}(log(N/D))$, if the particles are evenly distributed across each element.

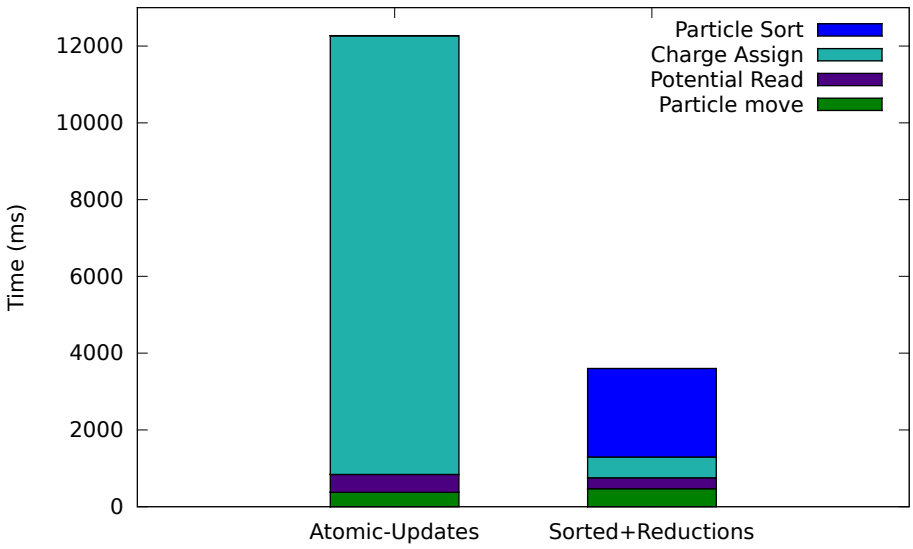

Figure 2-6: Comparison between a global atomic charge assign and a sort+reduce charge assign. Although we introduced an additional step, the total runtime is still far lower for the sort+reduce.

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 figure 2-6.

As you can see from the figure, the charge assign is on the order of 20x faster using the reduction technique, although this speed-up is somewhat offset by the sorting requirement. Sorting the particles also benefits reading the potential during the advancing step. This speed-up 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.

[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

2.2.1 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.[25][12]

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 $nv_{slab} \times$ nthreads floats to store the results of each thread. For a typical NVIDIA GPU with 49kB 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.

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.[25][12] 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.

2.3 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 figure 2-6 we can see that this sort step is now the dominant cost by a large margin[2]. The performance of this sort is undesirable, we would like 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 developed a GPU PIC implementation.

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

- Partial sort using message passing. [12][4]
- In-place particle Quicksort. [25]
- Linked list reordering [3]
- Full Radix Sort-by-key and reorder. [1]

Each of these methods have their own advantages and disadvantages. For the purposes

[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.[24]

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.

2.3.1 Message Passing Sort

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 bin in question is several cells wide. The benefit of considering this case is that only a small fraction of the particles will leave the bin during a given time step, and thus only a small number of particles need to be moved from one bin to the next. Most of the particles will remain in their respective bins 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 bins.

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 2-7 is 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 for a 2D mesh proceeds as follows[12]:

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.
6. For 3D repeat steps 1 to 5 for bi-clusters oriented in the third direction.

In cases where the number of particles in a cluster is greater than the number 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. [12]

This method works well if we assume that a particle will move at most one cluster in any direction in a single time step. If a particle moves more than two clusters, then the sort must be performed twice, one for each step that the particle moves. In the worst case scenario, where a particle can move from one cluster to any other of n clusters, it may take up to n sort steps to put every particle in its proper cluster. Another downside to this method is that it requires extra memory for the buffer. If the buffer is to small then global reorders will be performed more often. If the buffer section is too large then precious memory is wasted.

2.3.2 In Place Particle-Quicksort

The second sort method, developed by George Stantchev et al, eliminates the need for a buffer array, and is applicable to cases in which particles can move any number of clusters. This sort, like the message passing sort, is an "incomplete" which reorders only those particles that have changed bins. For this sort the particle list is divided up into "bins" representing groups of cells to which particles in that bin belong. A separate array keeps track of the indices, called "bookmarks" of the first and last

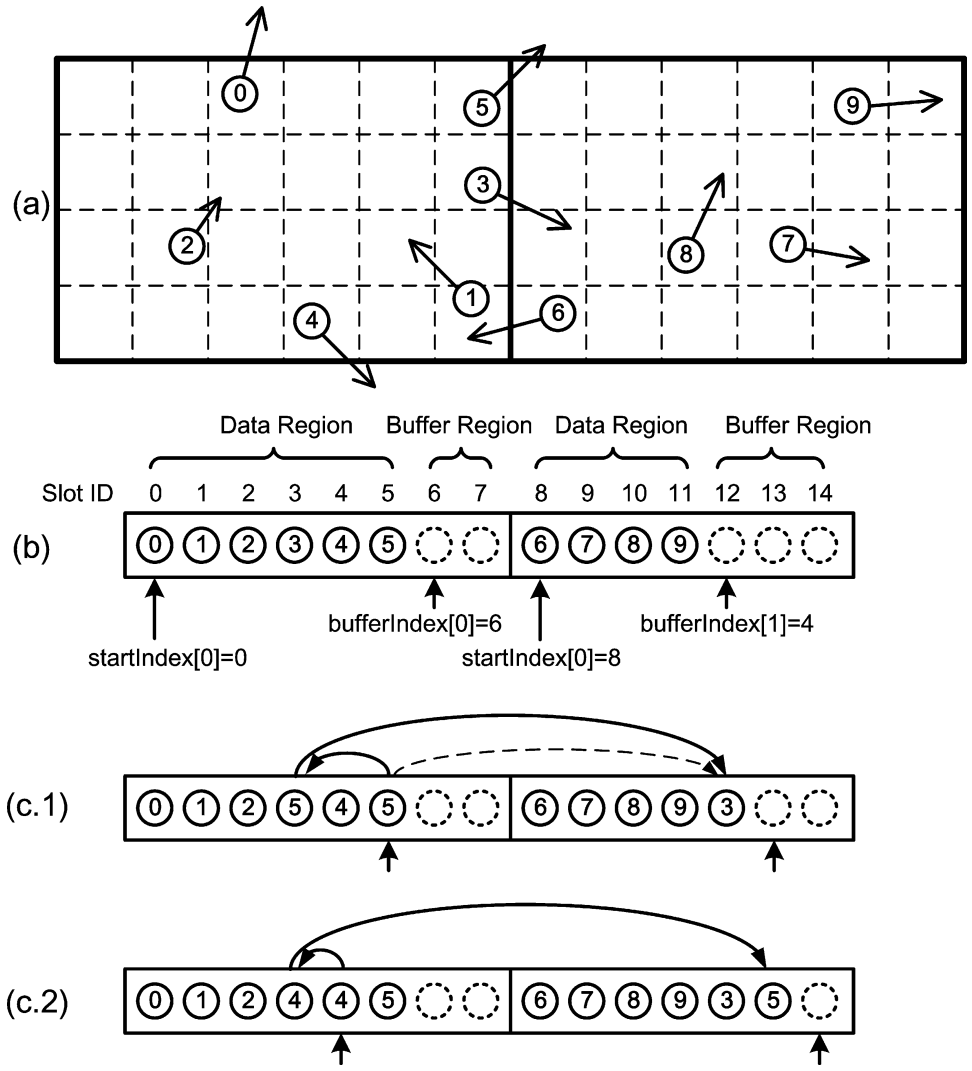

Figure 2-7: Message Passing particle sort. (a) Bi-cluster of cells. (b) Data structure of the bi-cluster particle array. (c.1) Particle 3 and 5 move to the right. Particle 3 is moved first to slot 12 which frees up slot 3. Slot 3, now empty, copies the last particle in the data region, which is particle 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 [12].

particle of each bin. The actual sorting is based on moving particles within their respective bins and manipulating the bookmarks such that in the end all particles are where they belong. The algorithm consists of two steps, a defragmentation step and a rebracketing step.

After the particle advance some of the particles will have moved to bins with a higher bin index, and some to bins with a lower index. Since the particle list is laid out continiously in linear memory higher bins will be to the right, and lower bins to the left of the current bin. The purpose of the defragmentation step is to organize the particles within a bin into three subsets based on the direction that they are traveling, left, right, or staying. Let $P = [p_{i_1}....p_{i_n}]$ represent a particle bin that maps to a group of cells C with memory index j. The cell cluster that each particle resides in is denoted by $I(p)$. As shown in algorithm 2.3 the de-fragmentation is performed through two loops over all elements in P

Algorithm 2.3 Particle defragmentation. From Stantchev et al. [25]

```
// Loop over particle bins
for all particle bin P_l do
  t_min ← lowest cell in the cluster associated with P_l
  t_max ← highest cell in the cluster associated with P_l
  α ← lowest particle index of P_l
  ω ← highest particle index of P_l
  // Forward Swapping pass
  for all p_i ∈ P_l in ascending order do
    if I(p_i) < t_min then
      swap(p_i, p_α)
      α ← α + 1
    end if
  end for
  // Backward swapping pass
  for all p_i ∈ P_l in descending order do
    if I(p_i) > t_max then
      swap(p_i, p_ω)
      ω ← ω − 1
    end if
  end for
  // Now α and ω are the new temporary bookmarks indicating the boundaries
  between migrating and non-migrating particles in bin P_l
end for
```

Once the defragmentation step is complete the subset of the particles that are migrating must be moved to their new homes and the particle bin bookmarks updated. This step, called particle rebracketing and shown in algorithm 2.4, is performed first for all odd l and then for all even l. Each rebracketing step starts by taking two adjacent particle bins, P_l and P_{l+1}. Particle p_{ω_l} in bin P_l is swapped with particle $p_{\alpha_{l+1}}$ in bin P_{l+1}. After the swap α_{l+1} is decreased by 1 and ω_l is increased by 1. This is repeated until $I(p_{\omega_l}) = I(P_l)$. The final bookmark l, the final particle in bin P_l, is given by the original values of α_{l+1} and ω_l, $l = \alpha^0_{l+1} - l + \omega^0_l$. The process is then repeated for all even values of l.

Algorithm 2.4 Particle ReBracketing. From Stantchev et al. [25]

```
// Odd Particle bins
for all particle bin P_l where l is odd do
   α ← α^0_{l+1}
   ω ← ω^0_l
   while I(p_{ω+1}) ≠ I(P_l) do
      swap(p_ω, p_α)
      α ← α − 1
      ω ← ω + 1
   end while
   // Last particle l in bin P_l is set to:
   l ← α^0_{l+1} − l + ω^0_l
end for
// Repeat for even bins
```

So far the method presented only works for particles moving only one bin to the left or right, but this can be expanded to include all possible magnitudes of particle movement. This is accomplished through hierarchical binning. The idea here is to create a binary tree of bins. At the first level of the tree the domain is divided into two bins, particles are sorted into the two bins using the defragmentation and rebracketing techniques described above. Once this is done each bins in the domain is cut in half and the process is repeated until the desired number of bins is reached. Traversing the entire tree takes $\mathcal{O}(N\log C)$ with C being the total number of bins and N the total number of particles. [25]

Now when this technique is implemented in CUDA each bin is assigned to

a single thread block for the defragmentation stage and two bins are assigned to each thread-block for the rebracketing stage. Unfortunately this means that at the first level of the hierarchal binning the GPU is severely underpopulated. In cases where particle behavior is reasonably well defined we can use clever geometries for the binning tree such that the first level has multiple bins. An example of this would be dividing up a three dimensional grid into slabs and assuming that particles can only transverse one slab in a time step. One reorder pass is performed for the slab level, followed by a hierarchical method for sorting within the slab.

This method is rather promising, it is applicable to a large variety of cases, and can be made significantly faster if assumptions about the system are integrated into the sort. A second benefit to this method is that it does not require a buffer, so more memory for particles. The downside to this algorithm is that it is very difficult to implement on the GPU and is rather poor in the general case.

2.3.3 Linked List Ordering

A third method of maintaining particle order, used by Heiko Burau et al is based on never moving the particle data from its original position in memory. Instead, each cell contains a pointer to the last particle in its local linked list, and each particle contains a pointer to its predecessor in its cell's linked list. When a particle changes cells it is deleted from the old cell's linked list and appended to the new cell's linked list. Particle list insertions and deletions are handled by atomic memory operations. [3]

There is one major issue with this approach, namely, the reordered access pattern can severely reduce performance due to un-coalesced memory accesses. Burau notes that this does in fact occur with the current deposition routine being the most affected routine. Unfortunately it is dificult to perform a meaningful comparison between this approach and other approaches with the information in the paper "PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster". [3]

2.3.4 Full Sort using Sort from the THRUST Library

The final and most general option is performing a full sort using the radix sort provided with the THRUST library.[15] The organization of the particle list is similar to the message passing and QuickSort methods, that is, particles are grouped into bins that represent clusters of cells. Particles are sorted by populating an array containing the bin indicies of all of the particles along with an array containing the indicies of all the particles in the particle list. The binID / particleID arrays are used in the THRUST sort_by_key() function. Once the particle indicies have been sorted, a kernel is launched in which each thread reads in a particle ID and then uses that ID to copy particle data from the original list into the new sorted list.

With the newer versions of the THRUST sort this method can be very fast, as well as simple to implement. This method is also very general and applicable to all PIC codes as it is completely independent of the physics of the problem at hand. On the other hand, the generality of this method can be a down side as there is no way to improve this routine based on the physics of the system. The message passing sort and QuickSort methods can perform better than the full sort in the cases where the maximum distance that particles will travel in a given time step is known to be small. If this is true then then the message passing sort and QuickSort methods will take fewer iterations. In cases where particle movement is completely un-defind then the THRUST sort wins out. A comparison of the THRUST sort and the Stantchev sort can be seen in figure 2-8.

The data shown in figure 2-8 was generated using a simple code that populates a partially ordered particle list and then sorts this list. The amount of data for each particle is similar to that of SCEPTIC3D, six floats for position and velocity, and one integer to keep track of bin index. The "grid" is represented by a series of bins, there is no need to represent individual cells for this comparison. The partial ordering is achieved by looping over the particle list and assigning each particle p_i to bin floor($\frac{N_{part}}{G}i$). Where N_{part} is the total number of particles, G is the size of the

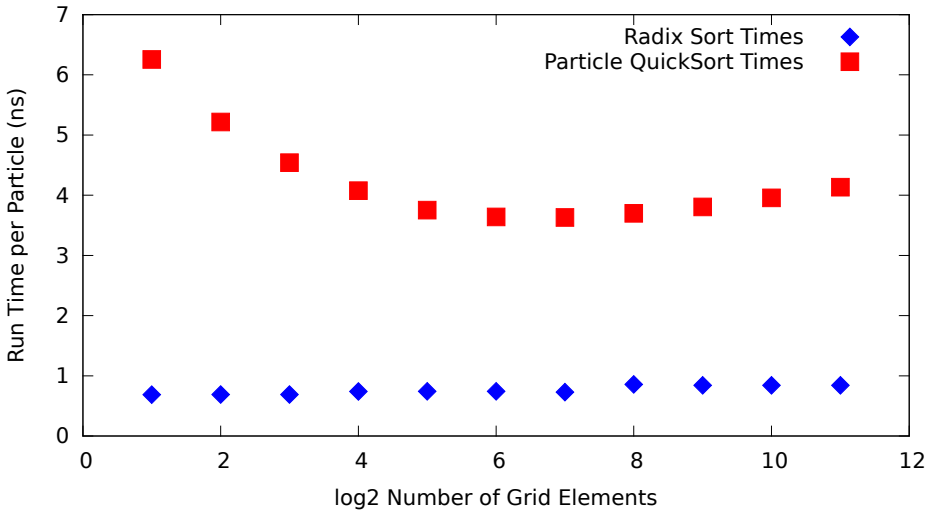

Figure 2-8: Comparison of Particle QuickSort and the THRUST radix sort. The latest implementation of the THRUST radix sort is very fast for a generalized problem. However, the Particle QuickSort can perform significantly better if constraints on particle movement are

grid, and i is the particle index. The "partial" part of the ordering is achieved by drawing a random number for each particle, if this number is less than R, then a second random number is drawn to determine if the particle is being placed in the previous or next bin. The boundary conditions are periodic such that particles in bin 0 that are being placed in a lower bin end up in bin $G-1$ and particle in bin $G-1$ that are being shifted up will end up in bin 0. The number R can be adjusted in order to adjust the fraction of particles that are moving. For the data shown in figure 2-8 $R = 0.2$, $N_{part} = 2^{24}$, and G ranges from 2^1 to 2^{12}.

Overall the thurst sort has the benefits of being easy to implement, very general, and is part of an externally maintained library that will be updated as GPU hardware changes. Fortunately the THRUST sort is also much faster than the radix sort from the NVIDIA GPU Computing SDK 3.1. The performance of the THRUST sort is fast enough that it is no longer the dominant cost. The downside to this sort is that it also requires allocation of buffer memory. The amount of memory required for this buffer array can be reduced by using separate arrays for each element of the particle list, which will be discussed in more detail in the next section. For now the THRUST

sorts benefits of simplicity and "fast enough" make it the winner of the particle sort competition.

2.3.5 Spatial Indexing

One way that the sort performance can be improved is through the use of space filling curves, such as a Z-order curve for cell-cluster indexing. Utilizing a space filling curve for the cluster indexing preserves the spatial ordering of the clusters in their layout in memory. This aids the sort by reducing the distance that particles must move in memory whenever they change clusters. We will use a z-order curve to index the cell clusters, since it is fairly simple to implement and will provide some performance boost to the code.

2.4 Particle List Structure

Another major design question is what kind of data structure should be used for the particle list. In the serial version of SCEPTIC3D the particle data is laid out in a 6xN array of reals in fortran. In C this layout corresponds to an array of six element structures. The array of structures format performs well on the CPU since only one particle is being operated on at a time, this means that data from only a single particle will be required at a given time. On the GPU things are a bit different. When a data access is performed on the GPU a group of 32 threads, a warp, executes memory requests for the same element, but from 32 different particles. If these addresses are not sequential then the request will be divided up into as many 128 byte cache-line transactions as it takes to fulfill the requests from all 32 threads. In kernels where all six elements will be used soon after one another, such as in the move kernel, this is not a big issue. When the request for the first element is read from global memory, the data for the other 5 elements will also be retrieved from global memory. For the most part cache hits are almost as fast as registers, so subsequent requests for the other

```
class XPchunk // Array of Structures
{
public:
  float x,y,z,vx,vy,vz;
};

class XParray // Structure of Arrays
{
public:
  float* x,y,z,vx,vy,vz;
};
```

Figure 2-9: Array of Structures and Structure of Arrays

5 elements will result in fast cache hits. Unfortunately this is not true of kernels in which only a subset of the elements is required, such as when calculating the particle bin index of each particle with relies only on the particle's position. [14]

The alternative is to use a structure of arrays, which corresponds to the transpose of the particle list structure in the fortran version of SCEPTIC3D. The main benefit of a structure of arrays is that reading in a single element of the particle list for 32 particles corresponds to a 128 byte transfer, the size of the cache line transaction. None of the bandwidth is wasted. This only takes 6 reads to global memory, no reads to cache. Reading in the elements of the array of structures results in at least 6 reads to global memory for the first element, followed by 5 reads from cache in the best case scenario. In order to test this we ran our toy GPU PIC code with both an array of structures (AoS) and a structure of arrays (SoA). The results of this test are shown in figure 2-10.

The biggest benefit to using the the structure of arrays is the smaller buffer required for the particle sort. Using a structure of arrays we only need a 1xN array of floats to sort each element of the particle list into. We would need a 6xN floats for sorting the array of structures. This approach scales well with more complicated particles that must keep track of more information. For SCEPTIC3D we have a total of 6 floats for spatial and velocity components, one 32-bit integer for the particle index, one float to keep of previous time step for reinjections, and a 16-bit integer for the

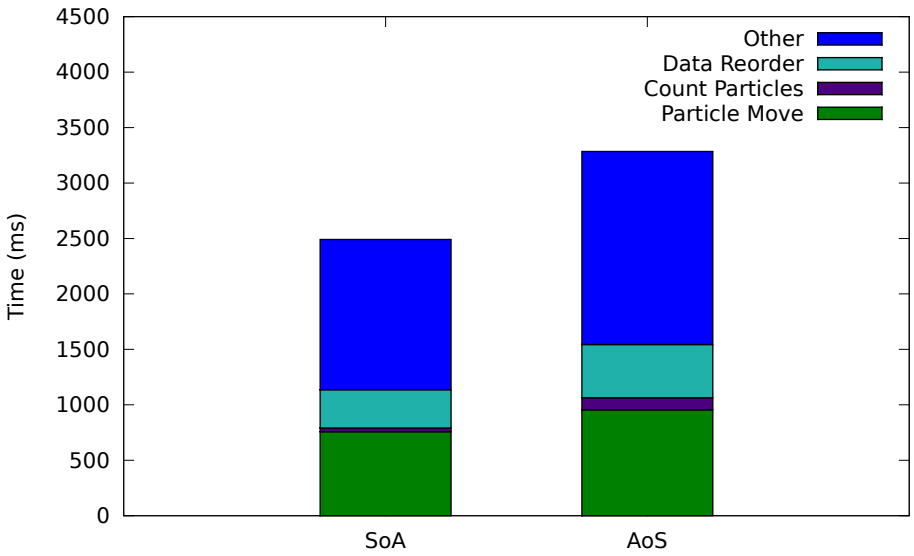

Figure 2-10: Execution times of main steps for Array of Structures and Structure of Arrays. Count Particles and Data Reorder are steps used for a sorted particle list. Count Particles counts the number of particles in each sub-domain. Data Reorder reorders the particle list data after the binindex / particle ID pair have been sorted by the radix sort.

binID. Using a structure of arrays the memory requirements imposed by the sorting step only increase the memory required per particle by 12%. The standalone buffer array can also be useful for other uses, such as stream compactions and reductions of diagnostic outputs.

The main downside of the structure of arrays approach is that a transpose is required for particle data transfers between the fortran SCEPTIC3D code and the GPU code. Performing this transpose on the GPU would require a large amount of memory, which would reduce the total number of particles that could be run. Performing it on the CPU will be more costly computationally, but will provide better performance overall.

2.5 Particle Advancing

Implementing SCEPTIC3D's particle advance on the GPU is fairly straightforward, and for the most part is identical to the CPU implementation. The primary design challenge here is dealing with reinjections.

The SCEPTIC3D particle advance algorithm is as follows:

Algorithm 2.5 SCEPTIC3D Particle Advancing

for all $p_i \in$ particles **do**
  $p_i \leftarrow \text{move}(p_i, \Delta t)$
  while $p_i \rightarrow \mathbf{x} \notin$ Domain **do**
    $\Delta t_{prev} \leftarrow \Delta t - \frac{\|\text{exit} - \mathbf{x}_i^0\|}{\|\mathbf{v}_i\|}$
    $p_i \leftarrow \text{Reinject}()$
    $p_i \leftarrow \text{move}(p_i, \Delta t_{prev})$
  end while
end for

Reinjections on the CPU are performed whenever a particle leaves the computational domain. When this event occurs, the code first determines the exact point during the timestep that the particle left the grid. From there a new particle is reinjected by calculating a new position and velocity as described in [21]. The new particle is then advanced the remainder of the times step, performing a second reinjection if the new particle also leaves the domain.

Unfortunately this method does not work well on the GPU. If this same algorithm were to be used on the GPU it would result in a large amount of warp divergence. Calculating new random positions and velocities is also expensive. Ideally we would have all of the particles be moved, and then operate on the subset of particles that need to be reinjected. In CUDA this is done by compacting the particle stream down to a list that contains only particles that are undergoing reinjection. This compacted particle list is still a particle list, and can be advanced in the same manner as the main list.

Chapter 3

Implementation

3.1 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. We define $n_{virtual}$ the ratio of a grid dimension and a bin dimension. These constraints are expressed mathematically through the grid dimensions, n_r, n_θ, n_ψ, and the bin sub-domain dimensions, nb_r, nb_θ, nb_ψ. As shown in equation 3.1, the ratio in each direction must be equivalent and be a power of 2.

$$\frac{n_r}{nb_r} = \frac{n_\theta}{nb_\theta} = \frac{n_\psi}{nb_\psi} = n_{virtual} \tag{3.1}$$

The second constraint on the grid dimensions is set by the hardware. The goal is to maximize the shared-multiprocessor occupancy for the charge assign stage of the code. Given that each thread block has the maximum number of threads, 512, and each thread requires roughly 25 registers, the maximum number of thread blocks that can exist simultaneously on a single SM is 2. This means that each thread 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 thread blocks per SM provides each block with 24576 bytes of shared memory each, or 6144 floats per block. The maximum that all three bin dimensions can be is 18. For the sake of simplicity these hardware constraints set nb_r, nb_θ, and $nb_\psi \leq 18$. For now none of the bin dimensions are allowed to exceed 18, even if another is smaller.

A third, loose constraint can be set in order to force a minimum number of thread blocks for the charge assign. The command line option "–minbins#" sets the parameter $n_{virtual} = \#$. This is useful in ensuring that enough thread blocks 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.

3.2 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.

3.3 Charge Assign

As previously mentioned, the charge assign is one of the most difficult functions to parallelize. The naive 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 decomposition where each thread block represents a separate sub-domain. The actual charge deposition method in this scheme is very similar to the naive approach, with a key difference being that all the threads in the thread block are operating on shared memory. Once all particles in the sub-domain have contributed to the 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.

3.3.1 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 3.1. 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 3-1.

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 must know all of the particles that reside within the region defined by that sub-

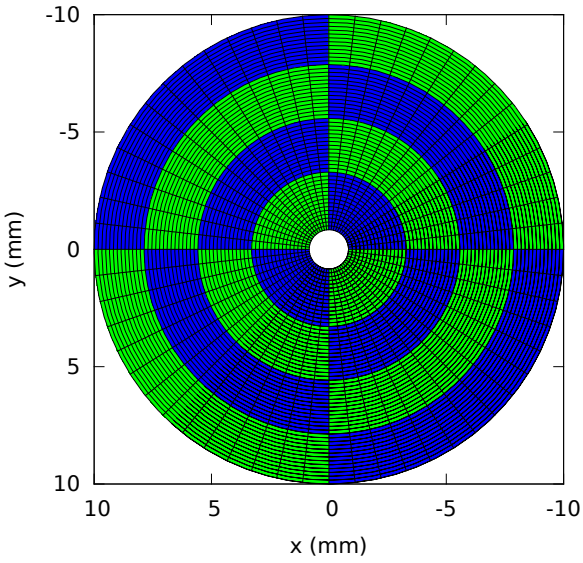

Figure 3-1: Graphical Representation of domain decomposition and ParticleBin organization. The domain is divided up such that every region, highlighted by alternating blue and green fill, is the same size. Additionally the number of bins in any given direction is a power of two. This figure shows what 4^3 bins looks like.

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.

3.3.2 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 of 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 3.1.

The spatial origin of the bin is hashed using a z-order curve and stored as a

Algorithm 3.1 ParticleBin Bookmark Calculation

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

  binID = ParticleList.binID[threadID]
  binID_left = ParticleList.binID[threadID − 1]
  binID_right = ParticleList.binID[threadID + 1]
  if binID ≠ binID_left then
    ParticleBins[binID].ifirstp = threadID
    ParticleBins[binID_left].ilastp = threadID − 1
  end if
  if binID ≠ binID_right then
    ParticleBins[binID].ilastp = threadID
    ParticleBins[binID_right].ifirstp = threadID + 1
  end if
end for
```

16-bit unsigned-integer. This 16-bit integer is refered to as the binID and is used 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 3.4. A 16-bit unsigned integer is used for several reasons. First the sorting method detailed in section 3.4 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.

3.3.3 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 3.2.

Each thread block reads in 512 particles at a time, although only 32 particles, a

Algorithm 3.2 GPU Charge Assign

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

warp, are processed in parallel within the block. Each thread within this warp loops 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:

$$N = \frac{31!}{(31-n)!G^n} \qquad T(n) \propto \frac{c}{32-n} \tag{3.2}$$

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

$$T(G) \propto c \cdot \sum_{n=1}^{31} \frac{1}{32-n} \frac{31!}{(31-n)!G^n} \tag{3.3}$$

This behavior can be seen roughly in 4-11. 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.

3.4 Particle List Sort

As previously mentioned in section 3.3 an ordered particle list must be maintained in order for the charge assign to be fast. The particle list sort, algorithm 3.3 consists of three distinct subroutines, populating the key/value pairs, sorting the key/value pairs, and finally a payload move.

Algorithm 3.3 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.

```
// 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 3-2: Thrust Sort Setup and Call

3.4.1 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:

$$ib_r = \frac{i_r}{nb_r}; \qquad ib_\theta = \frac{i_\theta}{nb_\theta}; \qquad ib_\phi = \frac{i_\phi}{nb_\phi} \tag{3.4}$$

The resulting block coordinates are then hashed using a z-order curve and stored as 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.

3.4.2 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[15]. The snippet of the sort code used in SCEPTIC3DGPU is shown in figure 3-2.

3.4.3 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 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 3.4 reorders only a single element of the particle list structure at a time.

Algorithm 3.4 GPU Payload Move

```
for all member ∈ 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 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 swapping 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%.

3.5 Poisson Solve

The field solve used in SCEPTIC3D involves solving the electrostatic Poisson equation:

$$\nabla^2\phi = \frac{n_e - n_i}{\Lambda_{De}^2} \tag{3.5}$$

Where ϕ is the electrostatic potential, $n_{e,i}$ the electron, ion charge densities normalized to the background charge density N_∞, and Λ_{De} is the unperturbed electron Debye length $\Lambda_{De} = \sqrt{\varepsilon_0 T_e / N_\infty e^2}$. Using the SCEPTIC3D approximation that the electron density can be taken to be a Boltzmann distribution, and normalizing ∇ to $1/R_p$ the probe radius, the Poisson equation can be rewritten as

$$\nabla^2\phi = \frac{exp(\phi) - n}{\lambda_{De}^2} \tag{3.6}$$

We are changing n_i to n in order to avoid confusion with the use of i as a spatial index. Assuming that the potential varies little from one time step to the next we can expand equation 3.6 about the known potential ϕ^* yielding

$$\nabla^2\phi_{i+1} = \frac{exp(\phi^*)[1 + (\phi_{i+1} - \phi^*)] - n}{\lambda_{De}^2} \tag{3.7}$$

Equation 3.7 can be linearized into the form $A\phi + \omega = \sigma$ using finite volumes. Following the derivation for the linear operator A from [22] we get:

$$\begin{aligned}(A\phi)_{i,j,k} = a_i\phi_{i+1,j,k} - b_i\phi_{i-1,j,k} + c_{i,j}\phi i, j+1, k - d_{i,j}\phi_{i,j-1,k} \\ + e_{i,j}(\phi_{i,j,k+1} - \phi_{i,j,k-1}) - [f_{i,j} + exp(\phi^*_{i,j,k})]\phi_{i,j,k}\end{aligned} \tag{3.8}$$

The full derivation of equation 3.8 along with the definitions of the vectors a

through f can be found in [5].

Now that we have our linear operator defined, along with the vectors ω and σ we can see that this is a simple sparse linear system. These system is fairly easy to solve using any sparse linear system solving scheme. One of the easier schemes to implement on the GPU is the preconditioned biconjugate gradient method *PBCG*. This method is easy to parallelize due to the simplicity of the operations involved, namely *PBCG* consists solely of sparse matrix-vector products, vector dot products, and multiplication of vectors by a scalar. All of these operations can be implemented efficiently on the GPU.

We can make an additional optimization of the PBCG solver by recognizing that only one element of our linear operator A changes every time step. This means that we can make use of some of the read-only memory spaces on the GPU, such as texture memory. Since texture memory is cached spatially in its own on-chip cache, we hope that we can significantly reduce the number of global memory transactions for all of the matrix multiplications. We ended up implementing two versions of the Poisson solve, one that stores both A and ϕ^* as textures, and one that stores them in global memory. The version to be used is determined at compile time with the use of pre-processor macros. The only concern with the texture implementation would be the cost of calling cudaBindTextureToArray(). However, since only one diagonal of A and and ϕ^* change over the course of the time step, the binding cost should not be an issue. Using textures for arrays in other parts of the code is also possible. Since we are using textures for the potential in the field solve it is very simple to also use a texture based potential for the particle advancing. The actual gain of using textures will be discussed in chapter 4.

We will not develop GPU implementations of the pre-processing steps required to prepare the ion density n from the particle-density interpolation since those steps do not represent a significant computational cost. Additionally, the particle-density array already exists in host memory due to the fact that an MPI reduce must be performed to gather this array from multiple GPUs. Since this data is already in

host memory, and the computational time is low, there is very little reason to rewrite these routines for the GPU.

3.6 Particle List Advance

Moving the particles on the grid starts with determining the acceleration of the particle. This is calculated by interpolating the gradient of the potential, $\nabla\phi$, from the spherical mesh using the same methods as the cpu code. The new position of the particle is then calculated using the leap-frog method:

$$\begin{aligned} \mathbf{v}' &= \mathbf{v} + \mathbf{a}\Delta t \\ \mathbf{x}' &= \mathbf{x} + \mathbf{v}'\Delta t \end{aligned} \tag{3.9}$$

Additional details of the particle advance can be found in reference [21] section 3.1.2.

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

3.6.1 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 of checking whether or not a particle has left the grid must also calculate the position of the particle when it crossed the boundary. Since the boundaries are both spheres, we can apply a very common ray tracing technique used to calculate a line-sphere intersection.

The equation for the path of a particle with initial position $\mathbf{p}_i$ and final position

$\mathbf{p}_{i+1}$ is given by;

$$\mathbf{l} = \frac{\mathbf{p}_{i+1} - \mathbf{p}_i}{\|\mathbf{p}_{i+1} - \mathbf{p}_i\|}\mathbf{x} = d\mathbf{l} \tag{3.10}$$

Combining equation 3.10 to the equation of a sphere with radius r centered at the origin, and expanding we get:

$$\begin{aligned} \|d\mathbf{l} - (-\mathbf{p}_i))\|^2 &= r^2 \\ d^2\mathbf{l}^2 + 2d(\mathbf{l} \cdot \mathbf{p}_i) + \mathbf{p}_i^2 &= r^2 \end{aligned} \tag{3.11}$$

We can then use the quadratic equation to solve for d. There are a few qualifiers, namely, $(\mathbf{l} \cdot \mathbf{p}_i)^2 - \mathbf{p}_i^2 + r^2 \geq 0$. The distance to the nearest intersection is simply the smallest positive solution of d. A particle leaves the domain if $d^2 \leq \|\mathbf{p}_{i+1} - \mathbf{p}_i\|^2$. If this condition is true, then the particle's new position p_{i+1} is changed such that $d^2 = \|\mathbf{p}_{i+1} - \mathbf{p}_i\|^2$. The particle's position is changed in order to accommodate some of the diagnostic outputs discussed in section 3.6.3.

3.6.2 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 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 would require 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:

Algorithm 3.5 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 ⊂ exited → reinjected_particles)

   // Recursively call this algorithm on the reinjected particles
   reinjected_particles.advance()

   Stream_Expand(reinjected_particles → exited ⊃ particles)
end if

return
```

A graphical representation of algorithm 3.5 can be seen in figure 3-3, which also includes the handling of collisions. For the handling of both collisions and reinjections four separate steps are needed. Step (1) determines whether or not a particle will be colliding, when the collision happens, and flags that particle in the list. Step (2) advances the particle's positions and velocities, and flags all particles that leave the simulation domain. Particles undergoing collisions in step (2) are only advanced to the time of collision, and particles that leave the domain are advanced to the point where they exit the domain. Particles that are flagged for a collision, but leave the domain before the collision takes place do not undergo a collision, but are flagged for reinjection. At this point we have several groups of particles, two that require special

operations and additional advancing, and a third that is completely finished. In step (3) we separate these groups of particles and into compacted collision and reinjection lists, and a sparse main list. The compacted lists are operated by the collision and reinjection operators respectively and then moved recursively. Once the recursive moves have been completed the particles in the two sub-lists are placed back in their original places in the main list by step (4). It should be noted that collisions have not yet been implemented in the GPU code, but the data handling infrastructure needed to support them as shown in figure 3-3 has been implemented. Since the collision and reinjection operators are completely independent, we could make use of multiple CUDA "streams" such that the two processes could be carried out asynchronously on the same GPU.

Compacting some subset of a parent list is a fairly easy parallel operation called stream compaction. Stream compaction, shown in figure 3-4 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 preceding elements yields a unique number that can be used as an index in a new array.

The new positions and velocities for reinjected particles are taken from a pre-calculate pool of new particles. This pool is approximately $1/10^{\text{th}}$ the size of the main particle list, and is repopulated prior to every advance step. Prior to the first advance step the entire pool is populated, but subsequent steps only refill slots that have been used in reinjections. This "pool" method was chosen over implementing the reinjection calculations on the GPU for reasons of code maintenance. There are currently several different reinjection schemes that SCEPTIC3D can call. Maintaining versions of those routines in both fortran and CUDA would be more cumbersome than the small benefit that calculating them on the GPU would provide.

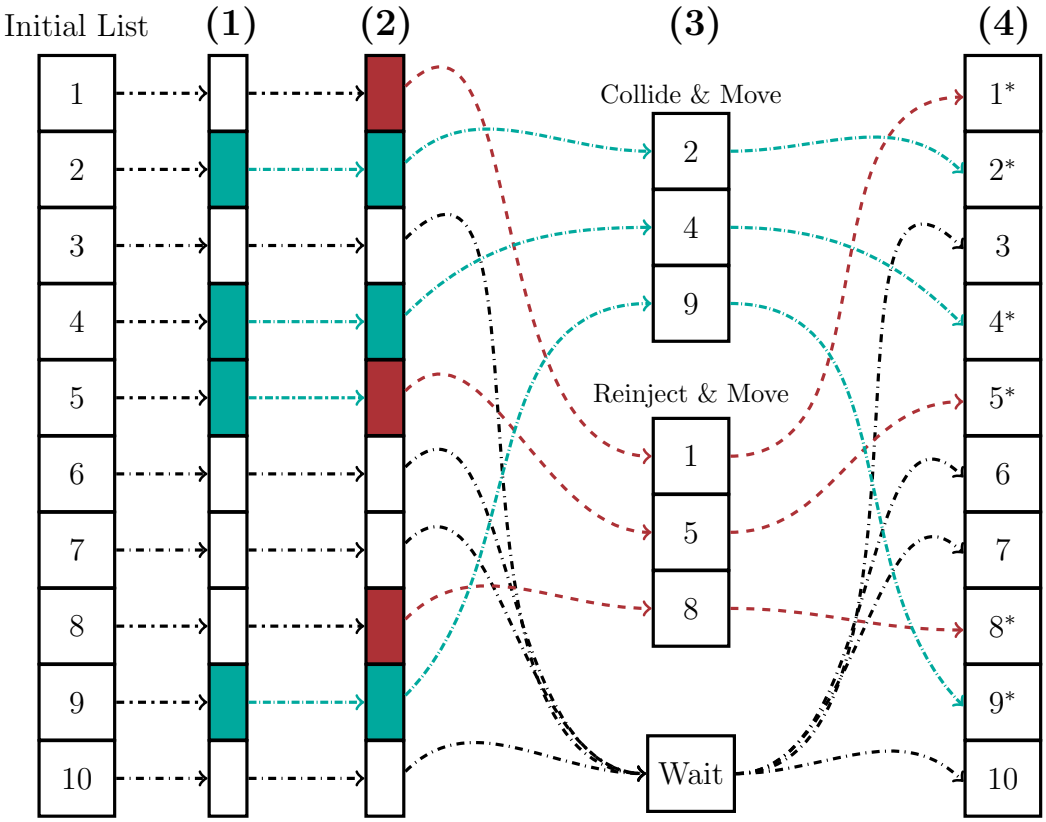

Figure 3-3: Illustration of GPU particle advancing algorithm with handling of reinjections and collisions. All of the particles to be advanced begin in the same list. Prior to updating the particle's positions and velocities, step (1) determines which particles will undergo collisions, and when those collisions will occur. Between steps (1) and (2) the particle's positions and velocities are advanced. After the advance some particles will have left the domain, these particles are flagged in step (2), with exit flags overriding collision flags. After step (2) two stream compactions are performed in order to condense the re-injection and collision lists. In step (3) the collision and reinjection operators are applied to these two lists respectively, and the entire move method is called recursively on each of the lists. Once all recursive steps have finished the collision and re-injection lists are merged back into the main array in step (4).

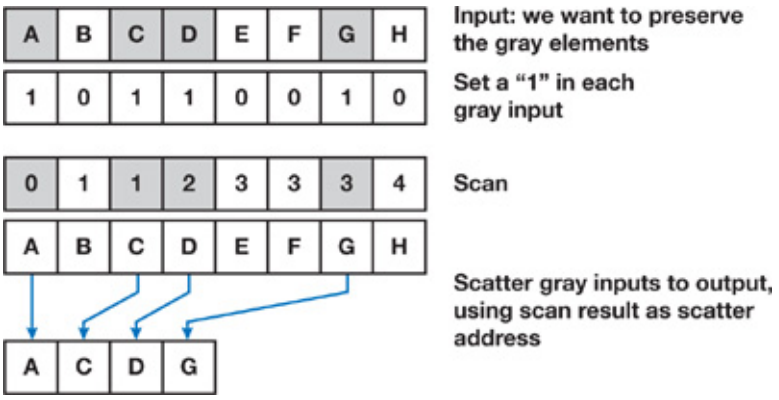

Figure 3-4: Stream Compaction from GPU Gems 3 [6]

3.6.3 Diagnostic Outputs

There are several outputs from the particle advancing routine that are desirable to keep in the GPU code. These outputs include;

- Total momentum transferred to the probe.
- Total momentum lost to particles leaving the system.
- Total momentum added to the system from reinjections.
- Total energy transferred to the probe.
- Total number of particles lost to the probe.
- Spatial distribution of energy transferred to the probe.
- Spatial distribution of momentum transferred to the probe.
- Spatial distribution of density transferred to the probe.

For the single value quantities, such as the quantities of total momentum, energy, and particle count are tallied using atomic operations in shared memory. Since the fraction of particles that leave the domain during a timestep is small, $\leq 7\%$, we can just use conditional statements within the primary move kernel to perform the tallies. While this is not strictly optimal, it does not have a very large impact on the overall performance of the move kernel.

Chapter 4

Performance

Unless otherwise specified the following tests were performed using two CPU cores or two GPUs, with MPI as the interface between multiple threads. The machine specifications for system 1 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 Supercomputer 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 code. These runs were performed on 2 GPUs with 17 million particles per GPU and a grid size of 64^3 using two different GPUs and CPUs. The first setup has already been mentioned and the second setup, system 2, is comprised of 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

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

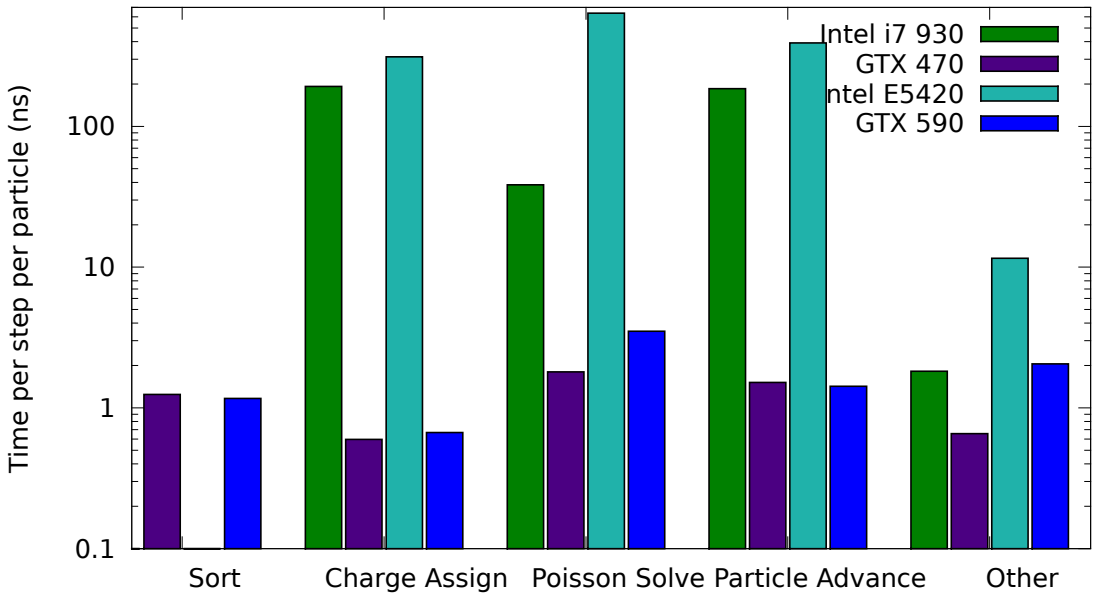

Figure 4-1: CPU and GPU Runtime comparison for a GTX 590 vs an Intel(R) Xeon(R) CPU E5420. Test was performed using 2 MPI threads handling 17 million particles each on a 64^3 grid.

GB ram. Figures 4-1 and 4-2 show the run times and speedups for the CPU and GPU on both of these systems. It is important to note that system 2 has a slower processor and lower memory bandwidth than system 1. The lower CPU performance combined with a faster GPU gives system 2 much higher speedup values compared to system 1.

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 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 figure 4-1 indicate, that with the latest THRUST sort, speeding up the particle list sort is no longer a major issue. The sort step could be improved by

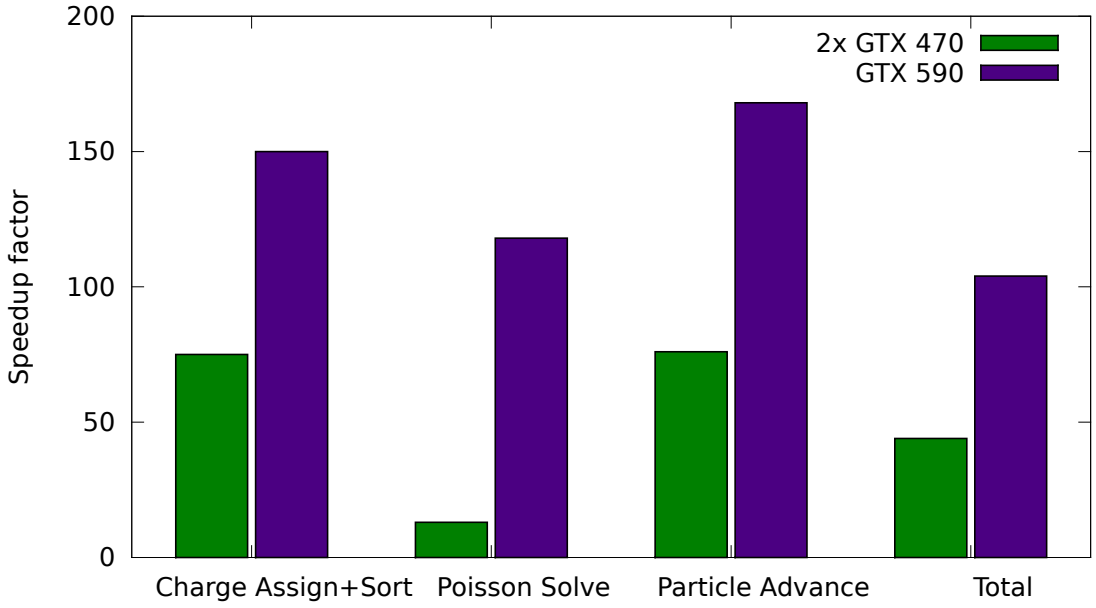

Figure 4-2: CPU and GPU Speedup comparison for a GTX 590 vs an Intel(R) Xeon(R) CPU E5420. Test was performed using 2 MPI threads handling 17 million particles each on a 64^3 grid. The difference in results is due primarily to the different hardware present on the two systems. The system with the GTX 590 is older than that used with the 2x GTX 470's

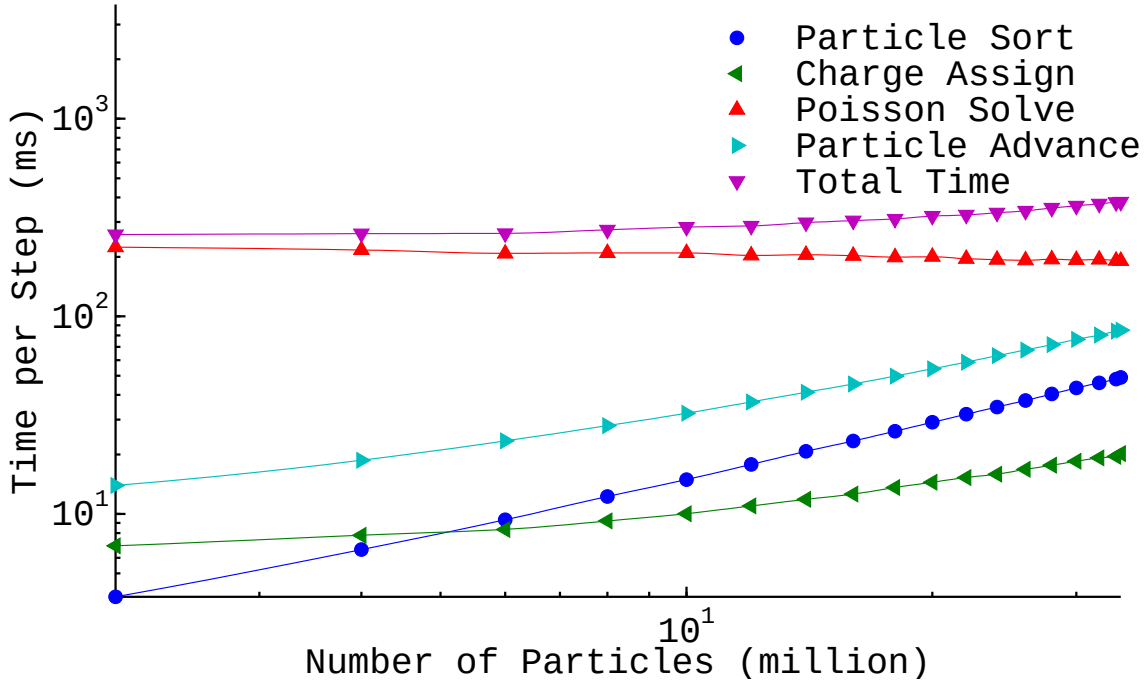

Figure 4-3: Number of Particles Scan on a 128x64x64 grid. Far more particles are needed in order to prevent the Poisson solve from being the dominant cost.

taking into account problem specific properties of a given pic code, but considering the ease of use and generality of the THRUST sort, it is unlikely that developing an optimized problem-specific sorting routine would really be worth it.

4.1 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 4-3 was performed on a 128x64x64 grid, and figure 4-4 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 4-5.

In the case of the 128x64x64 grid the Poisson solve is by far the most expensive

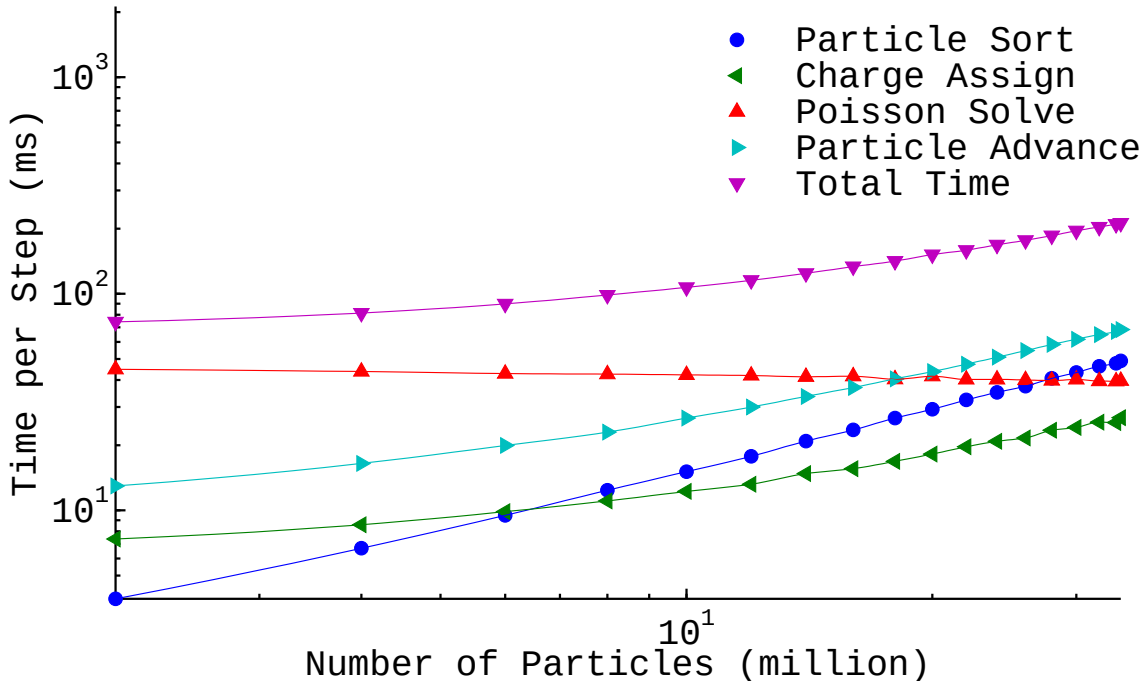

Figure 4-4: Number of Particles Scan on a 64x32x32 grid. At about 20 million particles, the particle moving steps become the dominant costs.

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 and particle sort for more than 15 million particles. Perhaps the most interesting behavior is best observed in the speedup factor, figure 4-5, 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 4-3 and 4-4 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 4-5 it is not unreasonable to conclude that with an even larger performance boost

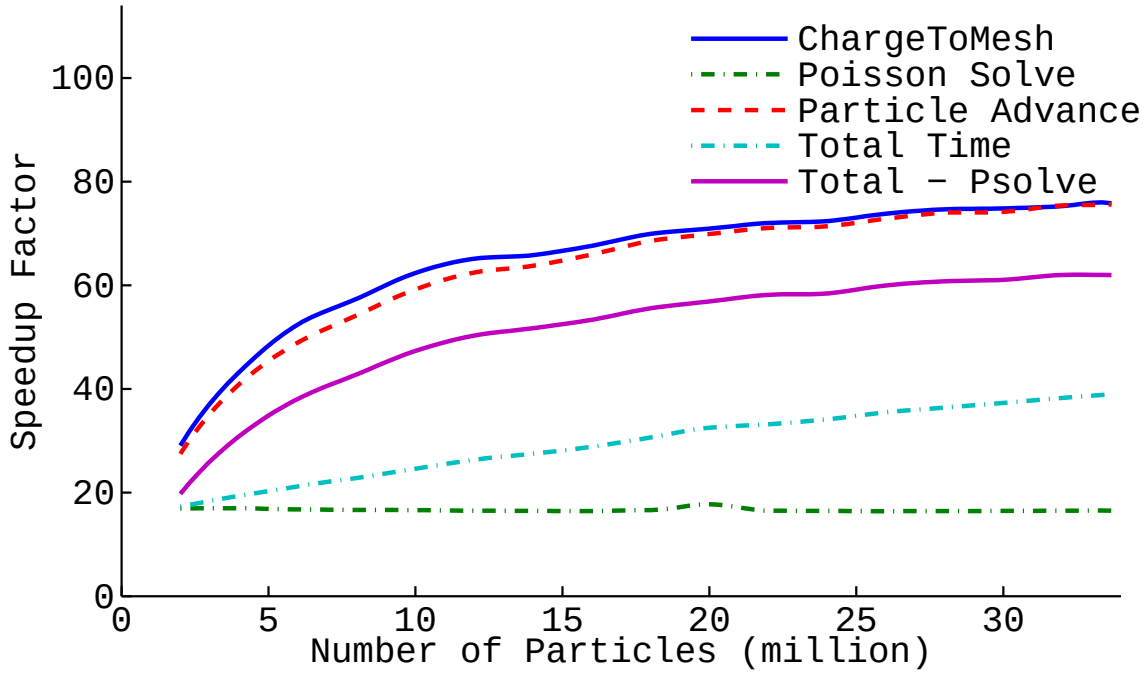

Figure 4-5: Speedup factor Number of Particles Scan on a 128x64x64 grid. The speedup factors for both the charge assign and particle advance are very similar, and follow similar trends. The total speedup is greatly reduced in this case because of the relatively high cost of the Poisson solve.

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

4.2 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.

4.2.1 Absolute 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 4-6 through 4-10. 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.

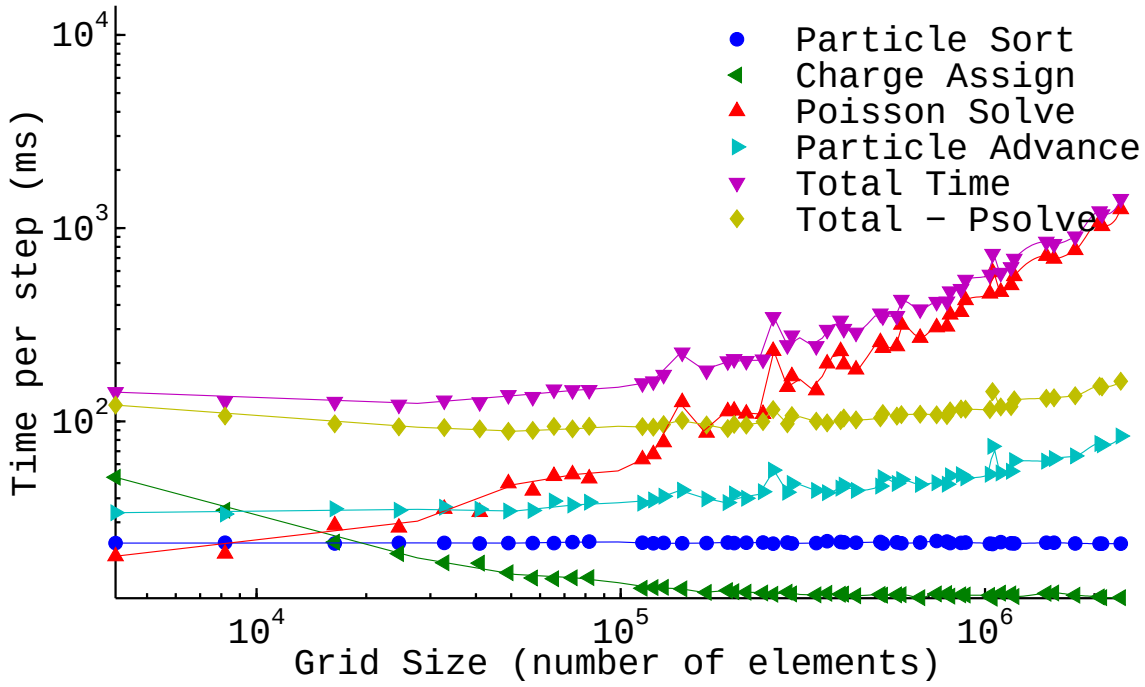

Figure 4-6: Gridsize Scan with 16 million particles, and 8^3 bins

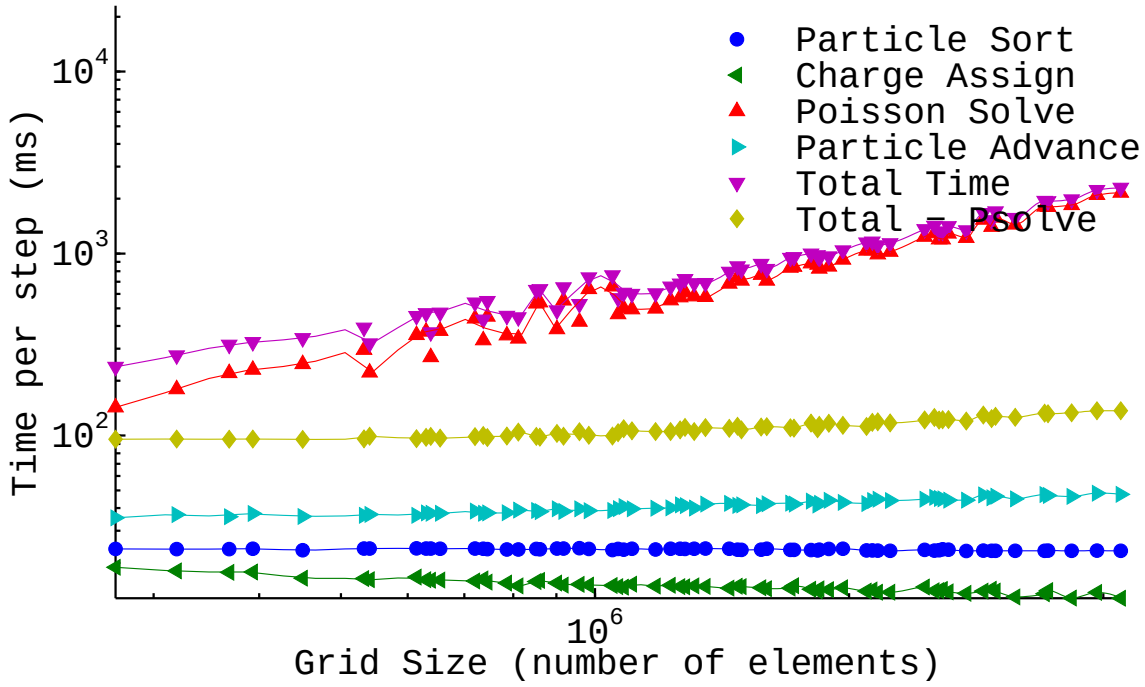

Figure 4-7: Gridsize Scan with 16 million particles, and 16^3 bins. The total runtime is entirely dominated by the Poisson solve.

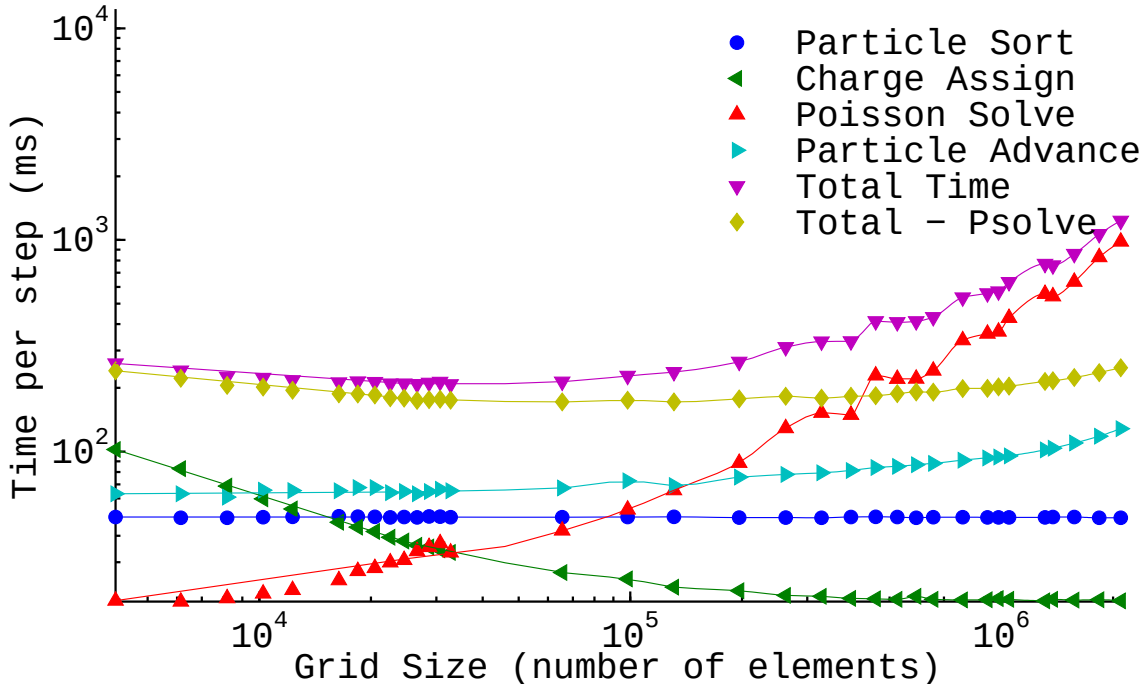

Figure 4-8: Gridsize Scan with 34 million particles and 8^3 bins. Note how when the contribution from the Poisson solve is removed there is a minimum at about 10^5 elements.

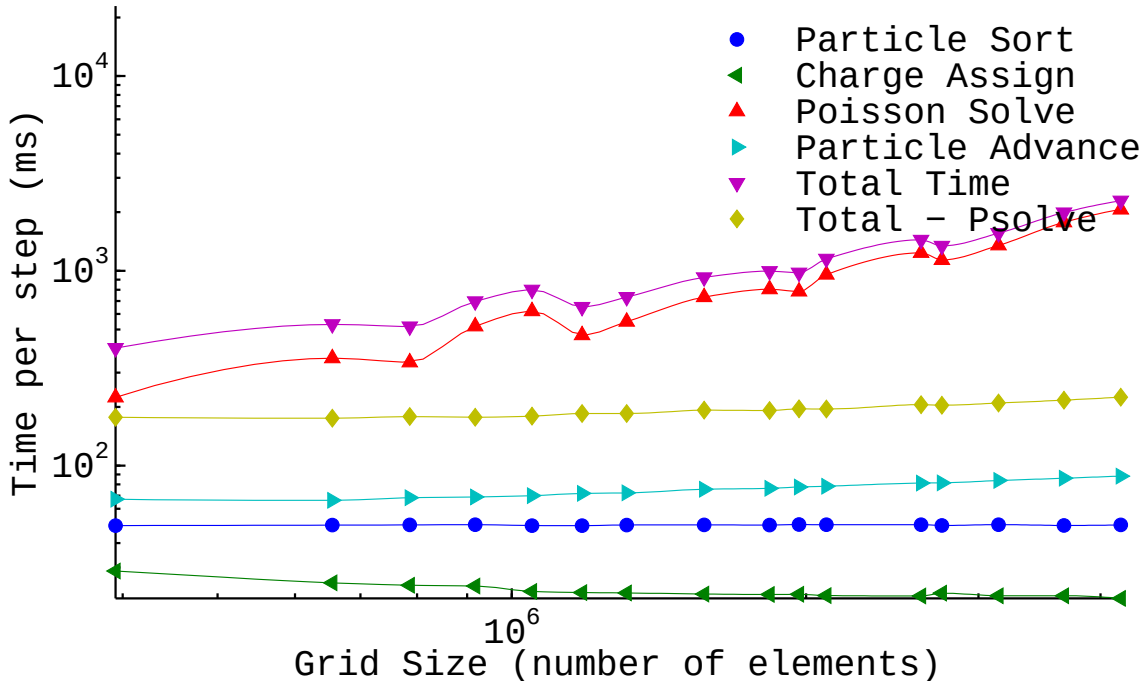

Figure 4-9: Gridsize Scan with 34 million particles and 16^3 bins.

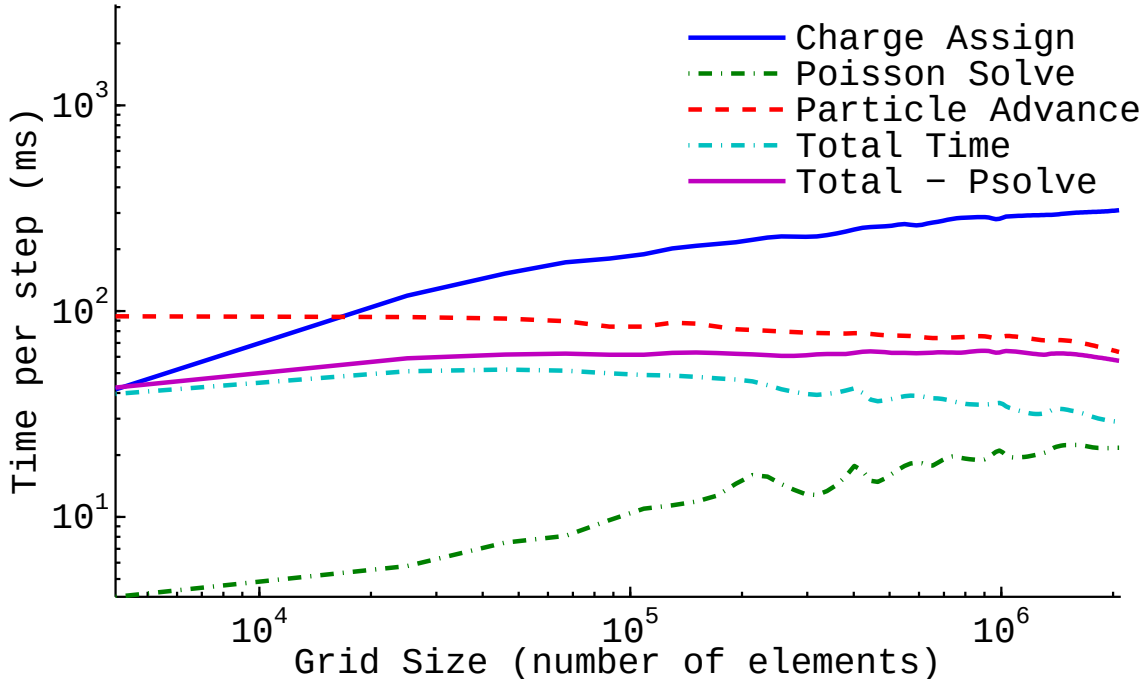

Figure 4-10: Gridsize Speedup Scan with 34 million particles and 8^3 bins. The total runtime is entirely dominated by the Poisson solve.

The more subtle scalings of the particle advance and charge assign can be seen in figures 4-6 through 4-10 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.

4.2.2 Threadblock Sub-Domain Size

In chapter 3 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 4-11 shows the time per step of each routine vs the size of the sub-domain. Correcting for the Poisson solve we do in fact see a minimum in the run time.

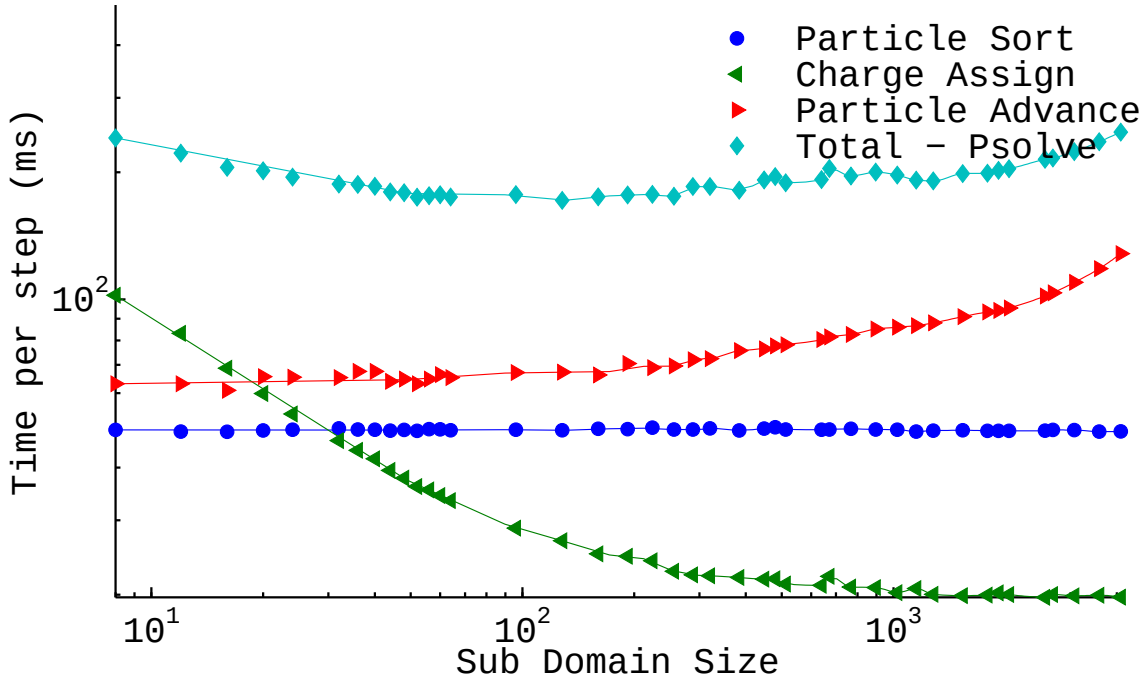

Figure 4-11: Sub Domain Size scan, also known as bin size, for 34 million particles. Note the minimum in the total - psolve run time.

4.3 Kernel Parameters Scan

One important performance consideration for GPU computing is ensuring that every thread performs enough work to warrant the cost of its creation. For the case of the advancing kernel we adjusted the number of particles processed by each thread. This is essentially a parameter that could be optimized with through auto-tuning, and will likely vary based on the hardware configuration. We performed such a scan with the advancing kernel, ranging the number of particles processed per thread from 3 to 16. The results of this scan can can be seen in figure 4-12.

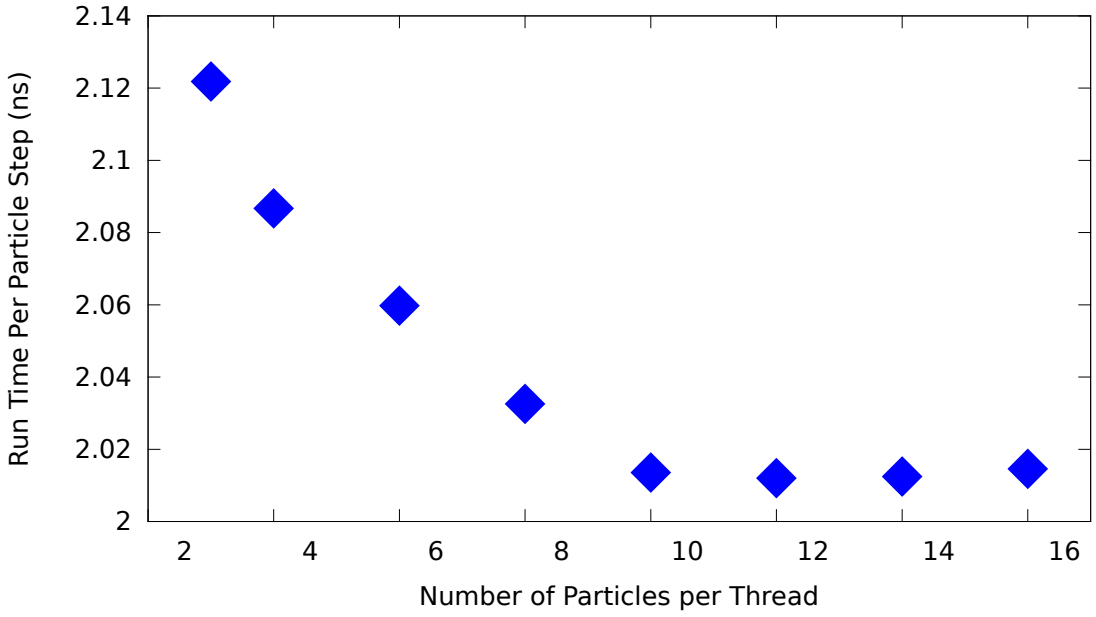

Figure 4-12: Adjusting the amount of work per thread for the advancing kernel. Increasing the number of particle per thread has a very small effect on the runtime of the advancing kernel, although in order to run this many particles in the first place we need to run at least 2 particles per thread, since the maximum number of threads that can be launched in a single kernel is 8.3 million when using 128 threads per block.

As we can see from the figure, there is a minimum around 10 particles per thread, which is about 5% faster than the 3 particles per thread case. This adjustment had

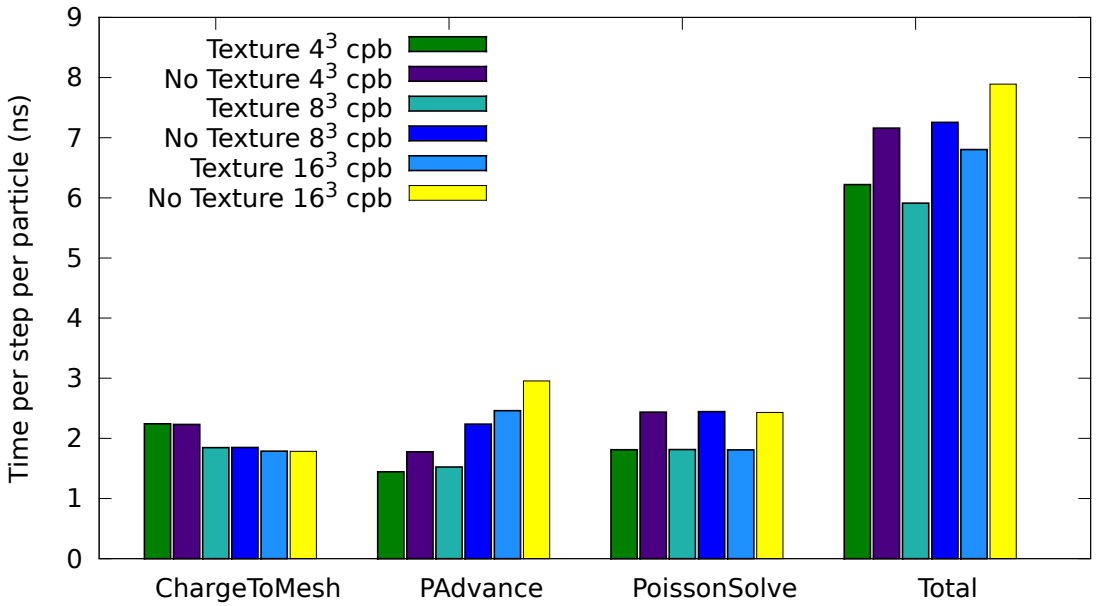

Figure 4-13: Comparison between texture enabled kernels at various bin sizes. Tests used 42 million particles with 200 time steps on a 64^3 grid.

a small, but still noticeable effect.

4.4 Texture Performance

One of the more interesting concepts that we investigated was the use of textures as a storage structure for both the potential, and the Poisson solve sparse matrix diagonals. Several comparisons between the two can be seen in figures 4-13 and 4-14. These tests were performed using 42 million particles with 200 time steps on 128^3 and 64^3 grids. We also varied the number of cells per sorting bin (cpb) in order to determine how the performance gain from using texture memory varies with how many possible mesh points an execution block of particles can access.

As you can see from figures 4-13 and 4-14, utilizing texture memory does provide a noticeable performance boost for the Poisson solve, and a small boost for the particle advance. Additionally, it is interesting to not that for particle advancing varying the number of cells per bin from 4^3 to 8^3 does not change the run time of the texture run.

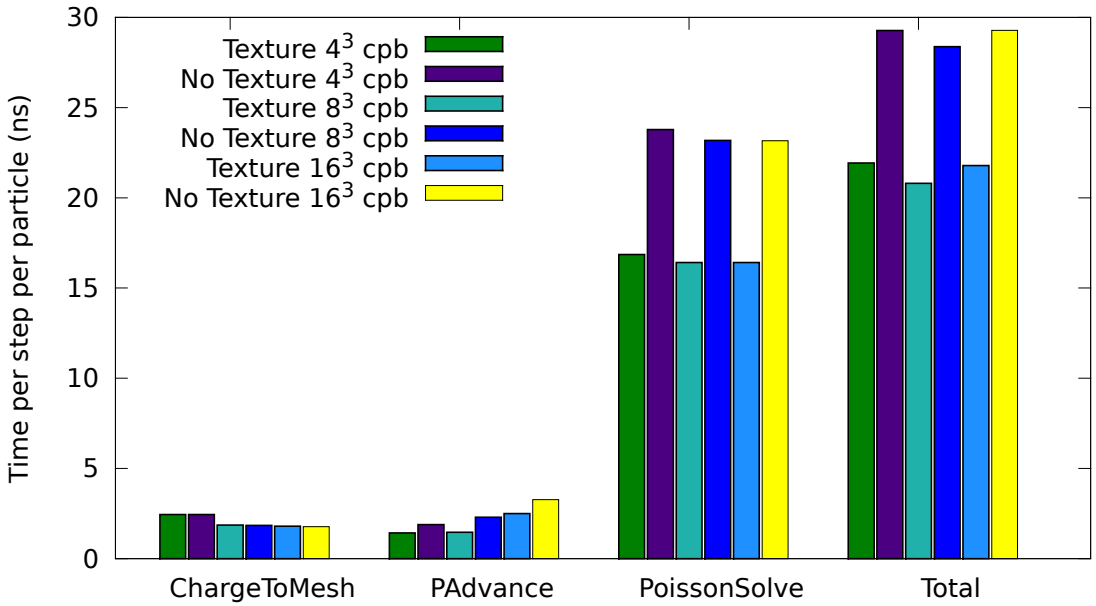

Figure 4-14: Comparison between texture enabled kernels at various bin sizes. Tests used 42 million particles with 200 time steps on a 128^3 grid.

The particle advance without textures becomes slower as the number of cells per bin increases, but there is a threshold for the texture based routine.

It is likely that more data structures could be changed to textures, or surfaces for read/write data structures. The Poisson solve in particular could benefit from additional texture/surface memory usage. The primary barrier to this is not performance, but implementation difficulty. Textures are not easy to incorporate into code due to the fact that they cannot be passed as function arguments, and must be declared within the same file scope as they are used. Another downside to textures is the process of "binding" global memory to the texture reference, which can be expensive.

Chapter 5

Conclusion

5.1 Review

Towards the end of chapter 1 we stated the primary goals of this project. We set out to develop a multi-GPU version of SCEPTIC3D using very generalized techniques. We came away with concrete conclusions concerning the performance impacts of using a full radix sort. We also explored ways to minimize warp divergence while handling reinjections and collisions using stream compaction and recursion. Lastly, we investigated the benefits of utilizing texture memory for constant arrays that are frequently accessed. In the end, the lessons that we learned from the exploration of each of these topics enabled us implement a GPU PIC code that attained a upwards of 100 times the performance of its CPU based counterpart.

While the end results were impressive, we learned that a PIC code implemented on the GPU can differ greatly from the same code implemented on a CPU. Over the course of this project we analyzed the methods used by others in the development of GPU PIC implementations. The main issue that was identified by most groups was how to efficiently implement the particle to mesh step on the GPU. The conclusion that we, and many of the papers we reviewed, came to was that the particle data must be sorted in order to ensure that the majority of the accumulation can be performed

using fast shared memory. So far nearly every paper that we reviewed developed their own techniques to keep the particle list sorted, with each technique optimized to fit a specific problem. While these specialized methods can achieve higher performance, more generalized methods tend to be easier to implement, and if they can be based on external libraries, easier to maintain.

Some of these past works produced very generalized methods. The particle-mesh integration technique developed by Stantchev et al is a good example of a broadly applicable technique. We aimed to develop more general techniques for other steps of the PIC method. We presented a fast, very general sort based on the radix sort provided by the THRUST library. While the costs associated with this sort are not negligible, the combined costs of the sort and charge assign are still comparable to the costs of the advancing step and the field solve steps. The other general technique that we developed was a way to minimize warp divergence when dealing with reinjections and collisions. The specific algorithm presented works well when the number of collisions and reinjections are small, but can be applied more generally by changing the conditions for placing a particle in a sub-list.

The final unique property of the GPU PIC implementation presented here is the use of texture memory as the primary data structure for the potential and the backbone of the Poisson solver. We concluded that texture memory should be used as a data structure for spatially organized data. Of course there are drawbacks to texture memory, primarily that the very strict usage properties can make it difficult to integrate with the rest of the code.

5.2 Implications

We were able to achieve speedup factors of 160 versus older CPUs and factors of 60 versus newer Intel i7 CPUs. With these speedup factors it is possible to attain cluster level performance on a multi-GPU workstation for SCEPTIC3D. In the tests we performed, we ran 50 million particles per GTX 590. There are commercially

available workstations that support up to 8 graphics cards. Populated with 8x GTX 590 graphics cards, one of these workstations would be the equivalent of a 2560 core cluster for the purposes of running SCEPTIC3D. Essentially a GPU enabled SCEPTIC3D can attain cluster level performance on a single workstation.

While we managed to improve the performance of the particle advancing and the particle-mesh interpolation steps by factors of 122 and 104 respectively, we ran into the issue where the Poisson solve became the dominant cost. As we saw in chapter 4 when the grid becomes much larger than 64^3 the cost of the Poisson solve dominates. If we start to consider even larger grid sizes, we eventually run into the limit where we are using more memory for the grid than the particle data. At that point we would have to decompose the domain across multiple GPUs such that each GPU only moved particles on part of the grid. This would greatly complicate the entire system because particles that leave one sub-domain would have to be passed from one GPU to another. Smaller PIC codes should not have to worry too much about this, although methods will have to be developed to handle this efficiently for larger codes.

One of the biggest implications of this work is its effect on the particle sorting step. Previous groups attempted to solve the sorting question using the radix sort implementation provided by the CUDPP library. However, this implementation was unacceptably slow, and combined with the poor memory access patterns of older cards, the library sort was out of the question. Recently the radix sort, now provided by the THRUST library, has been significantly improved. This fact, combined with the improved memory access patterns has resulted in a radix based particle sort method with significantly better performance. The performance of the THRUST sort is currently good enough that there is little reason to develop a specialized sorting technique.

5.3 Future Work

Within the scope of the GPU implementation of SCEPETIC there is at least one additional physics component that has not yet been included due to time constraints, the collision operator. While we have laid out the framework for implementing collisions on the GPU we have not yet implemented the entirety of that framework. Additionally, the re-injection particle list is currently calculated on the CPU, and a portion of this 'pool' is transfered to the GPU during the re-injection phase. Refilling the particle list on the CPU currently costs about $1/3^{\text{rd}}$ the cost of the particle advance. Implementing this on the GPU would significantly reduce the cost of this step, as well as eliminate the particle list transpose and host to device memory copy currently required. Further improvements to SCEPTIC3D could be made by optimizing the advancing kernel, specifically optimizing the force interpolation function to require fewer registers.

Pertaining to GPU PIC implementations in general, more work needs to be done concerning large grid sizes. As previously stated, two of the major issues facing larger scale GPU PIC implementations are multi-GPUs domain decomposition and optimizing large field solves. Developing asynchronous techniques that perform computations on both the GPU and CPU simultaneously is another high priority.

Appendix A

Tables

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

Table A.1: Run Time breakdown for different steps of the TOYGPUPIC code

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 A.2: Total Execution times for 100 iterations of the key steps of the TOYGPUPIC code at two different optimizations.

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 A.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.

Component	CPU (ns)	GPU (ns)	Speedup
Sort	0	1.247	-
Charge Assign	192.055	0.597	321x
Charge Assign & Sort	192.055	1.843	104x
Poisson Solve	38.398	1.803	21x
Particle Advance	185.245	1.517	122x
Total[1]	417.519	5.820	71x

Table A.4: 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 21 million particles each on a 64^3 grid.

Component	CPU (ns)	GPU (ns)	Speedup
Sort	0	1.166	-
Charge Assign	312.210	0.669	389x
Charge Assign & Sort	312.210	1.835	170x
Poisson Solve	637.349	3.500	182x
Particle Advance	391.335	1.426	274x
Total[1]	1352.461	8.818	153x

Table A.5: 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.

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] Dominique Aubert, Mehdi Amini, and Romaric David. A Particle-Mesh Integrator for Galactic Dynamics Powered by GPGPUs. *Proceedings of the 9th International Conference on Computational Science*, 55444:874–883, 2008.

[3] 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.

[4] 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.

[5] CB Haakonsen. *Ion Collection by a Conducting Sphere in a Magnetized or Drifting Collisional Plasma*. Masters of science, Massachusetts Institute of Technology, 2011.

[6] Mark. (NVIDIA Corporation) Harris, Davis) Sengupta, Shubhabrata. (University of California, and Davis) Owens, John D. (University of California. Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39. Addison-Wesley Professional, 2007.

[7] 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.

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

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

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

[11] Rejith George Joseph, Girish Ravunnikutty, Sanjay Ranka, Eduardo D'Azevedo, and Scott Klasky. Efficient GPU Implementation for Particle in Cell Algorithm. *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 395–406, May 2011.

[12] 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.

[13] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: FERMI. Technical report, NVIDIA, 2009.

[14] NVIDIA Corporation. NVIDIA CUDA C Programming Guide Version 4.1. Technical report, NVIDIA, 2011.

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

[16] NVIDIA Corporation. CUDA Toolkit 4.1 CUBLAS Library. Technical Report January, NVIDIA, 2012.

[17] NVIDIA Corporation. CUDA Toolkit 4.1 CUFFT Library. Technical Report January, NVIDIA, 2012.

[18] NVIDIA Corporation. CUDA Toolkit 4.1 CURAND Guide. Technical Report January, NVIDIA, 2012.

[19] NVIDIA Corporation. CUDA Toolkit 4.1 CUSPARSE Library. Technical Report January, NVIDIA, 2012.

[20] NVIDIA Corporation. OpenCL Programming Guide for the CUDA Architecture. Technical report, NVIDIA, 2012.

[21] L. Patacchini. *Collisionless Ion Collection by a Sphere in a Weakly Magnetized Plasma by Leonardo Patacchini.* PhD thesis, Massachusetts Institute of Technology, 2007.

[22] L. Patacchini. *Collisionless ion collection by non-emitting spherical bodies in E x B fields.* PhD thesis, Massachusetts Institute of Technology, 2010.

[23] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes.* Cambridge University Press, Cambridge, third edition, 2007.

[24] 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.

[25] 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.

[26] J P Verboncoeur. Particle simulation of plasmas: review and advances. *Plasma Physics and Controlled Fusion*, 47(5A):A231–A260, May 2005.