Report on the Feasibility of Implementing PIC Codes on a GPU

Joshua Payne

December 20, 2011

Abstract

GPUs have become a very attractive supplement to traditional high performance computing. GPUs have significantly better performance per cost and power consumption. However, GPUs introduce several additional levels of parallelism that must be contended with. New methods must be developed in order to take full advantage of the capabilities of this architecture. This paper explores the application of GPUs to particle tracking codes for plasma physics such as the NTCC Transp module NUBEAM. This code will outline how the ORBALL subroutine of NUBEAM was ported to the GPU with a 13x speedup, as well as present several general guidelines concerning implementing particle tracking codes.

1 Introduction

Simulating plasma behavior can be incredibly difficult. The equations that govern plasma behavior are incredibly non-linear due to significance of self forces. One of the best ways to simulate plasma behavior is by simulating the behavior of individual particles. Tracking the movements and interactions of a small fraction of the $10^20/m^3+$ particles can provide a very accurate representation of the bulk behavior of a real plasma. The problem with the these particle tracking codes is that they require a very large number of particles in order to achieve a reasonable accuracy on the order of tens of millions for small simulations up to tens of billions for large.

The large number of particles that these codes track means that they can be very slow. One way to reduce the computation time is to distribute the particle tracking across multiple processors. The ideal architecture for these codes would have a very large number of simple processors with very fast communication.

Graphical processing units or GPUs are designed to trace rays of light and generate an image. Ray tracing and particle tracking are incredibly similar and because of this, GPUs have the potential to make great particle tracking processors. However, there are some issues with moving to a new architecture. The shear amount of processing power reduces the time of many computations to the point where moving the data to and from the processor is more expensive than the computation. The lack of a large cache means that data access patterns and organization are significantly more important. Coupling GPUs and MPI introduces multiple levels of parallelism that have very different characteristics. Figure 2 illustrates the multiple levels of parallelism that an MPI-GPU system contains.

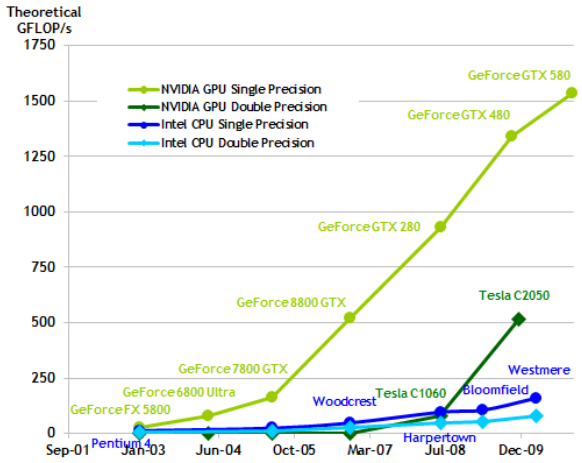

Figure 1: Performance comparison of GPUs vs CPUs.

The key to developing a high performance particle tracking code is properly decomposing and organizing the problem at each level of the multi-parallel tree. The complexity of the particle mover, collision operator, and timescale of the problem play large roles in how the problem is decomposed. This report will focus primarily on the fast ion orbit integration routines of the NTCC transp code NUBEAM [4], but will also draw on several key features of a generalized PIC code.

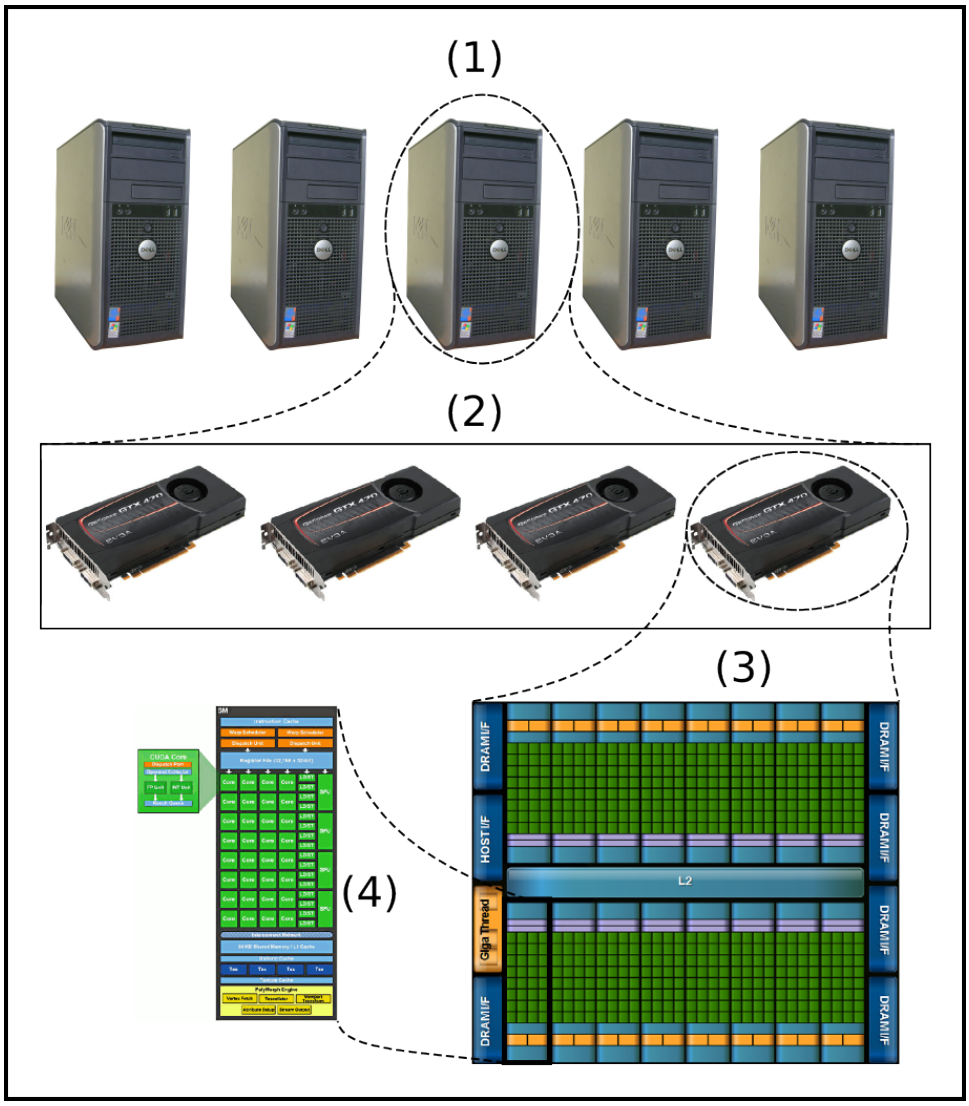

Figure 2: Multiple levels of parallelism. (1) Cluster of systems communicating through a LAN. (2) Multiple GPUs per system communicating through the PCIe bus. (3) Multiple streaming multiprocessors per GPU execute thread-blocks and communicate through GPU global memory. (4) Multiple cuda cores per multiprocessor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

2 NUBEAM: Modeling Fast Ion Orbits

Nubeam is a Monte-Carlo code used to model neutral beam injection in tokamaks. The code consists of two primary subroutines. The first is Depall, which tracks beam neutrals to their ionization point. Depall accounts for about 15% of the total run-time. The second major subroutine is Orball, which handles integrating fast ion orbits until they are neutralized, thermalized, or reach the end of the global time step. Orball is approximately 80% of the total run time. The other 5% of the run time is setup and recalculation of the fields every global time step. Since Orball is 80% of the total run time, it will be the first subroutine to be ported to the GPU.

2.1 Underlying Physics

The ORBALL subroutine handles the following physical operations:

- Ion drifts; $\boldsymbol{v}_E$, $\boldsymbol{v}_{\nabla B}$, $\boldsymbol{v}_\kappa$, $\boldsymbol{v}_p$
- Charge Exchange collisions $\frac{\partial f}{\partial t}|_{cx} = -f/\tau_{cx}(v)$
- Fokker-Planck collisions
- Anomalous diffusion

2.1.1 Ion drift equations

For ions in a magnetized plasma it is useful to use the components of the velocity perpendicular and parallel to the magnetic field. Ions are free to move parallel to the magnetic field, but are confined perpendicularly. Perpendicular movement is characterized by very fast orbits about the magnetic field line. The radius of this orbit is the larmor radius and the center of the orbit is the gyro-center. The net movement of the ion integrated over many gyro-orbits can be characterized by the drift of the gyro-center. The velocity of the gyro-center is given by the equations in figure 3.

The serial implementation of Nubeam has two different schemes for integrating the drift equations. The first uses a single time step in magnetic coordinates, and the second uses rk4 in Cartesian coordinates. The rk4 method also has a time step control that is based on the ion energy balance over the course of the time step. This feature is particularly useful because it helps maintain roughly the same integration error when switching between double and single precision. The fact that the rk4 integrator is in Cartesian coordinates allows for better use of texture memory. Overall the rk4 integrator was determined to be a better choice for the GPU implementation.

$$\boldsymbol{v}_E = \frac{\mathbf{E}\times\mathbf{B}}{B^2} \tag{1}$$

$$\boldsymbol{v}_{\nabla B} = \frac{mv_\perp^2}{2qB}\frac{\mathbf{B}\times\nabla B}{B^2} \tag{2}$$

$$\boldsymbol{v}_\kappa = \frac{mv_\parallel^2}{qB}\frac{\mathbf{R}_c\times\mathbf{B}}{R_c^2 B} \tag{3}$$

$$\boldsymbol{v}_p = \frac{m}{qB}\hat{\boldsymbol{b}}\frac{\mathrm{d}\boldsymbol{v}_E}{\mathrm{d}t} \tag{4}$$

Figure 3: Gyro-center drift equations [6].

2.1.2 Charge Exchange collisions (CX-collisions)

Most particles will check for a charge exchange collision every time step, in which case it is not a large performance hit to run the check kernel for every particle. However, only a small subset of the particles will actually spawn neutral particles that need to be tracked and redeposited or destroyed. Every time an ion crosses the midplane the code roles the dice and determines how many steps after the midplane crossing are to pass before checking for a cx collision. The first charge exchange kernel determines for every particle whether or not a cx check needs to be done at this time step.

The next kernel, beamcx, calculates the total charge exchange cross section and the probability of a charge exchange event. If a cx event occurs beamcx then calculates the number of neutrals spawned as a function of the particles weight and the time since the last charge exchange event. These neutrals then have to be passed off to another kernel, nutrav in order to determine if and where they are re-ionized.

Generally very few particles actually spawn neutrals. This is a case where it is advantageous to split off this subset of the particle list, apply the nutrav operator on that subset, and then merge the new ions with the main list. This can be accomplished with a general GPU computing technique call stream compaction, which will be explained later in this paper.

2.1.3 Fokker-Planck Collisions

Fokker-Planck collisions are collisions between two charged particles. For the purposes of fast ion tracking these collisions result in pitch angle and velocity scattering. The average change in these quantities is given by the following relationships:

$$\langle \Delta v^2 \rangle = \frac{2\delta t}{\tau_s} \left(\frac{T_e}{m_b} + \frac{v_c^3}{v^3} \frac{T_i}{m_b} \right) \tag{5}$$

$$\langle \Delta \zeta^2 \rangle = \delta t \nu_{ii} \left(1 - -\zeta_0^2 \right) \tag{6}$$

The new velocity and pitch angle is then chosen from Gaussian distributions of width $\langle \Delta v^2 \rangle$ and $\langle \Delta \zeta^2 \rangle$.[4]. Determining which particles are going to undergo fp collisions is very similar to the method used to determine cx collisions. The main difference is that in the current implementation particles undergoing fp collisions are not split off from the main particle list.

2.1.4 Anomalous Diffusion

Anomalous Diffusion occurs in tandem with Fokker-Planck collisions. The anomalous diffusivity is simply an externally set time varying radial profile. All fast ions that undergo FP collisions and do not thermalize are passed to the anomalous diffusion operator, which is in the same kernel as the FP collisions.[5]

2.2 General GPU Implementation Requirements

In order to support all of the underlying physics each particle must store 29 reals and 19 integers. A large amount of the underlying physics is dependent on the atomic species of the particle, this means that the particle list should be multi-dimensional, the first dimension is the particle species, the second is the particles index within that species list. The serial implementation of nubeam allows for each particle to take a different number of total time-steps, from about 5000 on average up to 15000. For the GPU code every particle should take roughly the same number of time steps. If 80% of the particles finish within 1/3 of the time steps a lot of GPU time is wasted.

On the mesh side there are a number of 2D fields that are frequently interpolated using bi-cubic splines. Textures are excellent for these because textures are cached spatially, which means that field data can be read in as few as 4 texture fetches, as opposed to 8 for a regular array. Textures are used for field interpolation and coordinate system mapping.

2.3 toyGPUPIC

Nubeam is a very complicated code, and as such it can be very difficult to test multiple implementation methods. This is where a very simple particle-in-cell code comes in handy. Particle-in-cell codes track particles on a grid and sum the charge and current density profiles at every step. These density profiles are then used to recalculate the potential mesh that moves the particles. The typical procedures of a PIC code are as follows:

1. Integrate the equations of motion.
2. Handle Collisions and reinjections.
3. Interpolation of charge and current sources to the field mesh.
4. Solve for Electric and Magnetic fields from the charge and current sources.
5. Interpolation of the fields from the mesh to the particle positions.
6. goto 1

Nubeam employs all of these steps save the field solve. Therefore a an efficient implementation of a simple particle-in-cell code that excludes the field solve should be representative of an efficient implementation of Nubeam. This simple PIC code referred to as toyGPUPIC.

The toy PIC code uses a particle described by 6 floats and 1 integer. The 6 floats are the 3-component position and velocity, and the integer is the binindex, which indicates what sub-domain the particle resides in. The toyGPUPIC code consists of 5 basic 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. Update the density array

The code was run using 4.2 million particles on a 32x32x32 grid for 100 time-steps. The execution times were recorded using NVIDIA's Compute Visual Profiler as well as calls to timer functions. This code served as a testbed for many of the implementation considerations.

3 Implementation Considerations

The nature of GPU data access and execution control means that close attention must be paid to data organization and execution paths. This section will discuss choices of data structures, data ordering, and execution control.

The results of the profiling at each optimization step can be seen in table 1. The rest of this section will discuss what changes were made at each optimization step and the reasoning behind those changes.

3.0.1 Particle List Structure

The largest structure in the code is the particle list. A total of 19 integers and 29 reals must be stored for every particle. The question here is whether to store the all the particles as an array of structures, or a structure of arrays. Two versions of the toyGPUPIC code were implemented with the following particle data structures:

```
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;
};
```

The kernel run times for the main components of each version of the code are shown in table 1.

Essentially the run times for the structure of arrays are faster than the run times for the array of structures. These results, and the significantly faster particle counting

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 1: 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.

routine can be explained by how the GPU accesses global memory. When a thread in a given warp wants to access data in global memory it sends a request to a warp scheduler along with all the other threads in the warp. The warp scheduler then processes all of the global memory accesses by fitting them into the minimum number of 128-byte cache line reads needed to contain all of the memory addresses, call it n. The amount of data read into cache is then $128n$-bytes. This is fine if all of the threads are accessing data that is sequentially addressed in memory, but when the size of the data type does not fit well into the 128-byte cache line there is wasted bandwidth. Additionally, in the case of the Count Particles function, each thread only requires 1 element of the particle data, the binindex. The code for this kernel is shown in figure 3.0.1.

A particle list as a structure of arrays is always preferable to an array of structures, especially when only a subset of the particle properties are required.

3.0.2 Data ordering

Data ordering is another very important part of a high performance GPU code. As mentioned in the previous section, GPU memory accesses are made in 128-byte chunks. If global memory accesses are not coalesced into the minimum number of 128-byte chunks then precious memory bandwidth is wasted. However, coalesced memory access is not the only thing that data organization can affect. Data ordering also has a significant impact on what algorithms can be used. One example is the particle counting kernel shown in figure 3.0.1. Counting the number of particles in a bin when the data has been sorted by bin is very easy. If the particle list were unsorted then every bin might have to go through the entire particle list, or a subset of the list. This would require the entire particle list to be read from global memory multiple times. Another method is to have every particle atomically update a counter for its own bin. Too many threads trying to update the same counter simultaneously serializes the execution of the code, which defeats the purpose of parallelizing it. With a sorted list the entire particle list has to be read only once. Since the adjacent binindexs fall within the same cache-line, each particle only has to go to the cache to get the index of the particle next to it. This can also be done with shared memory, basically a user managed cache.

In order to test the impacts of data organization, sorting the particle list in particular, several different organization methods were implemented.

As shown by table 2, the largest contribution to the run time of this implementation

```
__global__
void count_particles(XPlist particles, Particlebin* bins)
{
        int idx = threadIdx.x;
        int gidx = idx+blockIdx.x*blockDim.x;

        int nptcls = particles.nptcls;

        uint binindex;
        uint binindex_left;
        uint binindex_right;

        if((gidx < nptcls-1)&&(gidx > 0))
        {
                binindex = particles.binid[gidx];
                binindex_left = particles.binid[gidx-1];
                binindex_right = particles.binid[gidx+1];

                if(binindex_left != binindex)
                {
                        bins[binindex].ifirstp = gidx;
                        bins[binindex_left].ilastp = gidx-1;
                        bins[binindex].binid = binindex;
                }

                if(binindex_right != binindex)
                {
                        bins[binindex].ilastp = gidx;
                        bins[binindex_right].ifirstp = gidx+1;
                        bins[binindex].binid = binindex;
                }

        }

        if(gidx == 0)
        {
                binindex = particles.binid[gidx];
                bins[binindex].ifirstp = gidx;
                bins[binindex].binid = binindex;
        }
        if(gidx == nptcls-1)
        {
                binindex = particles.binid[gidx];
                bins[binindex].ilastp = gidx;
                bins[binindex].binid = binindex;
        }
}
```

Figure 4: Count Particles Kernel. This kernel counts the number of particles in each bin by having 1 thread per particle, then having each thread ask the thread next to it if their particles belong in the same bin. If they do not belong in the same bin then that thread knows it is the last or first particle in its bin.

Component	Un-Optimized (ms)	Sorted (ms)	Sorted+Shared (ms)
Particle data read, move, and write	375	375	468
Potential Grid Read	467	342	285
Density Update	1.143e4	1.004e4	542
Particle List Sort	0	2.305e3	2.305e3
Total	1.227e4	1.308e4	3600

Table 2: Total Execution times for 100 iterations of the key steps of the move kernel at three different optimizations.

is the density array update. The time spent on the density array update constituted about 93% of the total time spent on the move kernel. When implementing the move algorithm in parallel, there will be multiple threads attempting to update the density array simultaneously. Problems arise when one thread overwrites or ignores the density array update of another thread. In order to ensure that all updates are done correctly, the updates must be done atomically. This causes the code to become serial for the density update, significantly reducing any speed up that would otherwise be gained. Therefore, it is very important that these atomic operations be avoided and if possible replaced by another method for updating the density array.

This is a thread communication problem, and one of the best places to look to for answers to thread communication problems is shared memory. Shared memory is as fast as registers if bank conflicts are minimized, and can be accessed by all threads in a given thread-block. Since thread execution within a thread-block can be synchronized, safe updates to shared memory can be ensured. The problem is that shared memory is rather small, a measly 48kb for a single multi-processor, meaning that 48kb can be split between multiple thread-blocks. Realistically this leaves about 16kb or so shared memory for a thread-block at maximized sm occupancy.[1] This means that if the grid is large then it will not fit into shared memory, so shared memory can only be used if all particles within the thread-block belong to a specific sub-domain. This means that a block level domain-decomposition is required for fast density updates.

In toyGPUPIC this sub-domain is the size of a single cell, which means that the particles must be sorted by individual cell. This has both pros and cons. The positive effect is that with only 8 nodes to update an 8x512 fits fine in shared memory, meaning that each thread can read a particle and assign its weighted density to each of the nodes in its own section of shared memory. Once all spots in the shared memory array have been filled, a parallel reduction for each node can be done. Reductions on shared memory are very fast, meaning that even though it is technically more computational complex, the fact that it is done in parallel makes it worth it. An example of GPU reduction is shown in figure 5.

There are a few downsides to sorting by each individual cell. The first is non-uniform particle distributions. If some cells contain significantly more particles than other cells, then the code has to wait on those cells. Another issue is the sorting itself. Sorting by individual cell for a 32^3 grid means that a radix sort[2] has to go through 15 bits. Smaller

[1]Each SM is limited by 1536 threads, 6 thread blocks, 48kb shared memory, or 32k 32-bit registers.

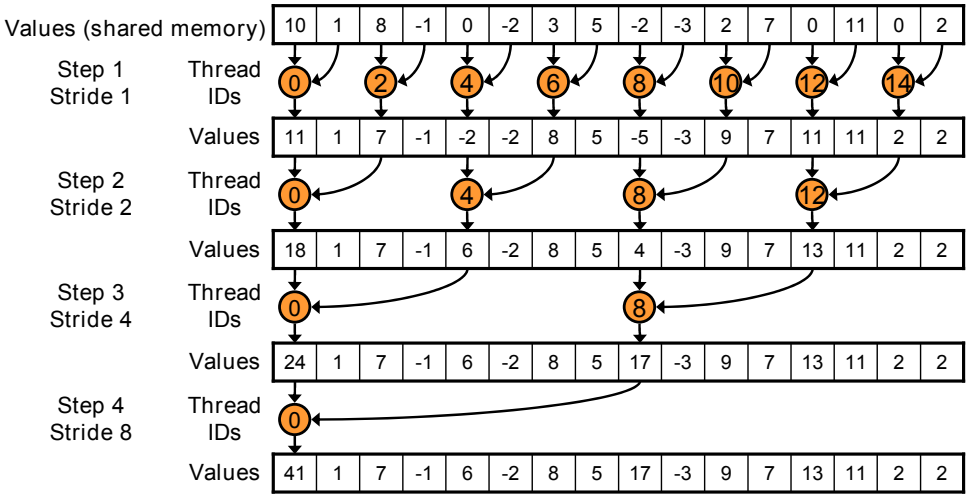

Figure 5: Parallel Reduction with interleaved addressing.[1]

cells also means that the ratio of surface area to volume is greater, which means that the particle flux into and out of the cell is larger. Larger particle fluxes between cells means that more particle data has to be moved around at each sort. The key to solving this is a hybrid approach. Operations on shared memory are fast. Sorting ensures that all particles within a group belong to the same sub-domain. Having a larger number of grid elements per sub-domain means that reductions are not as viable, but what about shared atomic operations? Atomic operations can serialize the code, but when they are done on shared memory by a few hundred threads instead of a few million, they are significantly less costly. This is the method that was used in the sceptic3D code.

3.0.3 Execution Control

The final implementation guideline concerns balancing a threads workload and the cost of just launching it. This means that in some cases it is fine to treat each thread as an ideal parallel processor that only performs a single task. However, if the workload is light, it can be advantageous to treat each thread as a serial processor that does the work of several threads. A prime example of this is the move kernel in sceptic3Dgpu. Tests were run using 2 GPU runs of sceptic3Dmpi with 8 million particles per gpu and 20 time steps per run. The number of particles per thread for the move kernel (padvnc) was varied from 1 to 10. The results can be seen in figure 6.

4 Results

Detailed performance analysis of Nubeam is not available at this time. However, a preliminary results are available. Detailed performance results of toyGPUPIC are available and will be presented in the following section.

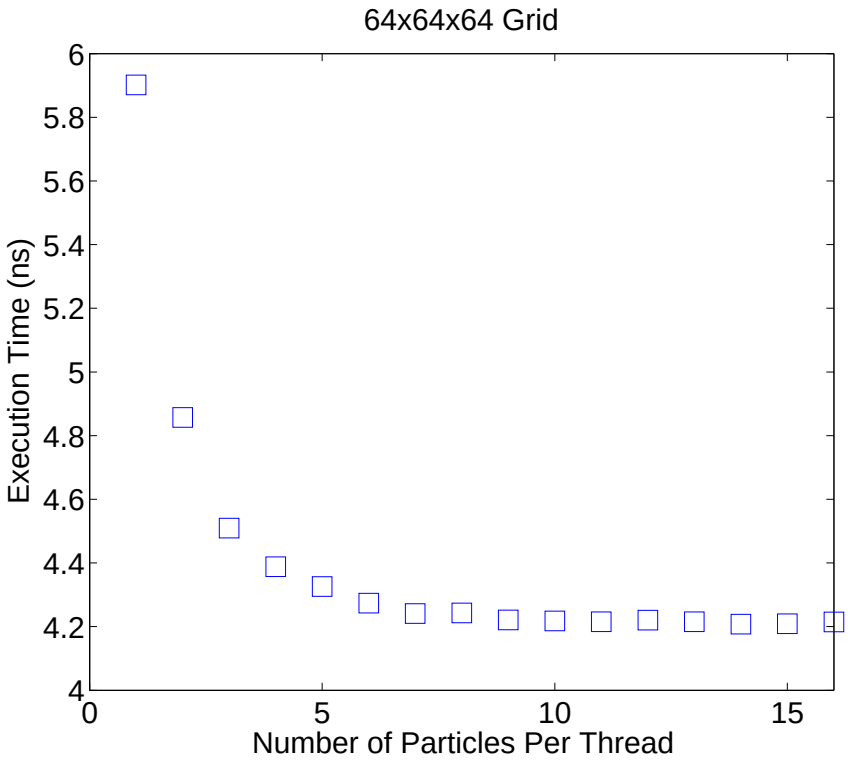

Figure 6: Varying the number of particles each thread handles in the particle moving kernel. This is for 2 gpus with 8 million particles per gpu on a 64^3 spherical grid. Note how it is significantly faster to run multiple particles per thread. This is because the work required to move 1 particle does not significantly outweigh the cost of launching the thread.

4.1 Nubeam Results

So far only the orbit integration routines have been fully debugged. The charge-exchange, Fokker-Planck, and anomalous diffusion routines have been implemented but are still buggy. Results for just the orbit integration routines are as shown in table 3. Figure 7 shows what the particle movement and magnetic coordinate mesh in nubeam look like.

Number of Particles	CPU time(s)	GPU time(s)
100,000	662	48
200,000	1001	78

Table 3: Roughly 13x speedup for the orbit routine, or about 3x speedup for Nubeam as a whole. This is for 1 CPU vs 1 GPU.

4.2 Performance of toyGPUPIC

The first performance measurements were done in a system parameters scan, which involved varying the particle count and the grid size. The number of particles ranged from 1024 to 4.2 million. Two grid sizes were used for these scans, an 8x8x8 grid, and a

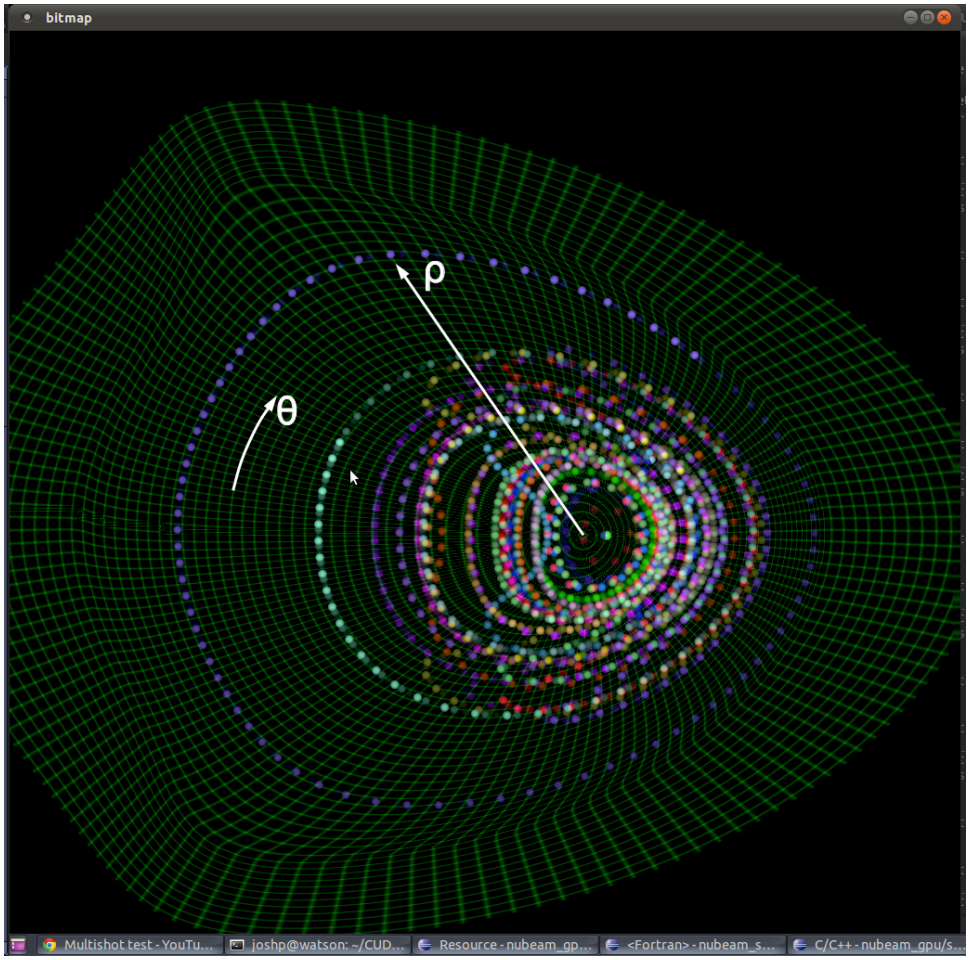

Figure 7: Visualization of fast ion orbits calculated by gpu Nubeam. Video can be found here

16x16x16 grid. The hardware utilized was 1 NVIDIA GTX 470 video card. The GTX 470 has 1280MB global memory and 448 cuda cores. The computer's CPU was an intel i7 930 with a clock speed of 2.8 GHz. The time measured was the total execution time for 100 iterations of a move kernel and all of its supporting kernels. The results of these scans can be seen in figures 4.2 and 4.2.

The cpu comparison was run on the same hardware with 4.2 million particles on a 32x32x32 grid. The results can be seen in table 4. The completely un-optimized method was 78 times faster than the CPU implementation. The first level of optimization involved using the brute force method, but on a sorted particle list. Including the sorting time, this implementation was 60 times faster than the CPU implementation. The second level of optimization makes use of shared memory for the grid data and for updating the density array. This implementation was 150 times faster than the CPU implementation, including the sorting time, and the time for several support kernels.

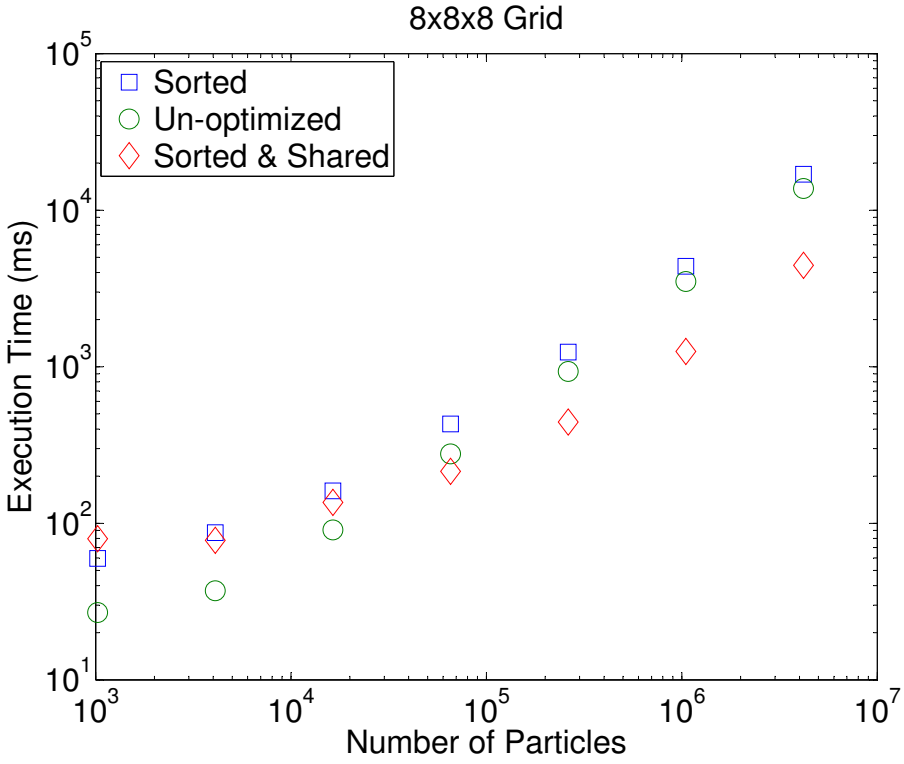

Figure 8: System Scan on an 8x8x8 grid. Execution time is for 100 iterations including the sort time and time to run several supporting kernels.

Method	CPU Time (ms)
Un-optimized	9790
Sorted	12834
Sorted & Shared	5101
CPU	766302

Table 4: CPU and GPU run time comparisons. Times are for 100 iterations of 4.2 million particles and a 32x32x32 grid.

5 Conclusions

GPUs are very well suited to particle tracking codes. While tracking the particles is trivially parallel, getting meaningful information out of the tracked particles, such as the distribution function, induced charge and current distributions, etc. is not. In order to achieve high performance out of a particle tracking code it is recommended that the following guidelines are taken into consideration:

- Particle list should be sorted spatially.
- Particle list should be a structure of arrays, not an array of structures.
- Depending on the complexity of the kernel, each thread should process multiple particles in order to hide launch latency.
- Spatially organizing data is very important.

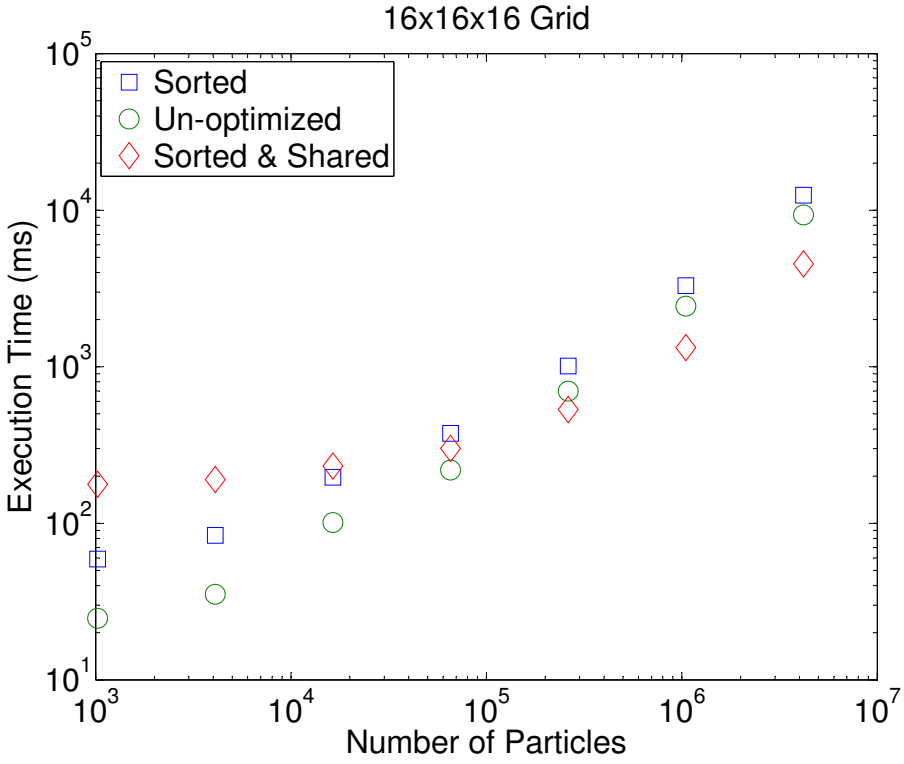

Figure 9: System Scan on an 16x16x16 grid. Execution time is for 100 iterations including the sort time and time to run several supporting kernels.

6 Code Sources

1. nubeam_gpu code can be found at: `https://github.com/spad12/GPU_Nubeam`

2. nubeam_gpu requires a working installation of nubeam, which can be found here: `http://w3.pppl.gov/ntcc/NUBEAM`

3. toy PIC code can be found at: `https://github.com/spad12/GPUPIC_testbed`

4. sceptic3Dgpu code can be found here: `https://github.com/spad12/sceptic3D/tree/gpu`

References

[1] Mark Harris, Optimizing Parallel Reduction in CUDA, `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`

[2] Satish, N., Harris, M., and Garland, M. "Designing Efficient Sorting Algorithms for Manycore GPUs". In Proceedings of IEEE International Parallel & Distributed Processing Symposium 2009 (IPDPS 2009).

[3] L. Patacchini; I. H. Hutchinson Fully self-consistent 3D modeling of spherical Mach-probes in ExB fields. *IEEE International Conference on Plasma Science*, 2009.

[4] R. J. Goldston; D. C. McCune New Techniques for Calculating Heat and Particle Source Rates due to Neutral Beam Injection in Axisymmetrix Tokamaks. *Journal of Computational Physics*, 1981.

[5] Alexei Pankin; Douglas McCune; Robert Andre; Glenn Bateman; Arnold Kritz; The tokamak Monte Carlo fast ion module NUBEAM in the National Transport Code Collaboration library. *Computer Physics Communications*, 2004.

[6] J. Freidberg. *Plasma Physics and Fusion Energy*. Cambridge University Press, 2000.

[7] NVIDIA Corporation NVIDIA CUDA C Programming Guide. 5/6/2011.