Princeton Plasma Physics Laboratory

PPPL- 4996

PPPL- 4996

Multi-threaded GPU Acceleration of ORBIT with Minimal Code Modifications

Ante Qu, Stephane Ethier, Eliot Feibush and Roscoe White

FEBRUARY, 2014

Prepared for the U.S. Department of Energy under Contract DE-AC02-09CH11466.

Multi-threaded GPU Acceleration of ORBIT with Minimal Code Modifications

Ante Qu[1] Stephane Ethier[2] Eliot Feibush[2] Roscoe White[2]

August 16, 2013

Abstract

The guiding center code ORBIT [1] was originally developed 30 years ago to study the drift-orbit effects of charged particles in the strong guiding magnetic fields of tokamaks. Today, ORBIT remains a very active tool in magnetic-confinement fusion research and continues to adapt to the latest toroidal devices, such as the NSTX-Upgrade, for which it plays a very important role in the study of energetic particle effects. Although the capabilities of ORBIT have improved throughout the years, the code still remains a serial application, which has now become an impediment to the lengthy simulations required for the NSTX-U project. In this work, multi-threaded parallelism is introduced in the core of the code with the goal of achieving the largest performance improvement while minimizing changes made to the source code. To that end, we introduce compiler directives in the most compute-intensive parts of the code, which constitutes the stable core that seldom changes. Standard OpenMP [2] directives are used for shared-memory CPU multi-threading while newly developed OpenACC [3] directives and CUDA Fortran [4] code are used for Graphics Processing Unit (GPU) multi-threading. Our data shows that the fully-optimized CUDA Fortran version is 53.6 ± 0.1 times faster than the original code.

1 Introduction

1.1 ORBIT

Magnetic fields in circular and bean-shaped tokamaks can be conveniently described in terms of perturbations of an equilibrium field with magnetic surfaces. A set of Hamiltonian guiding-center drift equations can be written using canonical variables very closely related to the magnetic coordinates used in equilibrium and stability analysis. The purpose of the guiding-center code ORBIT, written in Fortran 77, is to use these equations to study and simulate the behavior of particle orbits for different input magnetic fields [1]. More recently, there has been a need to study the behavior of particle loss, and this requires the simulation of larger numbers of particles to acquire good statistics. These simulations currently take several days of CPU time on a single processor core, so a significant speedup is needed to reduce that run time to a few hours or even minutes. Parallelization is the only way to achieve this kind of speedup. We want, however, to achieve this with little to no impact on how the developers and users currently interact with ORBIT. To that end, we make sure that the code modifications are minimal and affect only the core of the source

[1] Princeton University

[2] Princeton Plasma Physics Lab

code. This part of the code, which essentially constitutes the particle-advancing algorithm, is very stable and no longer touched by the developers.

1.2 Shared-Memory Multiprocessing

Shared-memory multiprocessing is a form of parallelism where multiple processor cores have access to a unified shared memory. This architecture allows the multiple cores to exchange data through a single shared memory. This form of parallelism is in contrast to distributed parallelism, which requires explicit messages to be sent across a network in order for processor cores on different nodes to exchange data. The Open Multi Processing (OpenMP) Application Program Interface (API) is a set of specifications of compiler directives in C/C++ and Fortran that support shared-memory multiprocessing [2]. In Fortran, OpenMP directives are added as comments to the source code so that it can be compiled with or without implementing OpenMP. The simplicity and flexibility of the OpenMP interface makes it an ideal candidate for a method to implement shared-memory multiprocessing with minimal modifications to the code.

1.3 General-Purpose Computing on Graphics Processing Units (GPGPU)

In recent years, many scientific codes have been shown to achieve impressive speedups on graphics hardware, which is normally optimized for fast rendering of images through massive multi-threaded parallelism. There now exist on the market so-called “General-Purpose Graphics Processing Units”, or GPGPUs, which are tailored for computation rather than image rendering. GPGPUs differ from their graphical display counterparts mainly by the availability of double-precision arithmetics, memory error correction capability, and larger on-board memory. While CPUs are optimized for fast serial processing, GPUs are processors with a large number of smaller cores optimized for massively parallel processing using multi-threading and Single Instruction, Multiple Data (SIMD) processing (also known as vector processing). GPGPUs consist of many “stream” processors that perform instructions known as kernels. Each kernel is a set of instructions for all the threads to perform, and the GPU is most efficiently used by scheduling a thread every time another thread is idle (for example, while waiting for a memory fetch). As a result, GPU parallelism often involves creating orders of magnitude more threads than cores in order to saturate the computational power. These threads are assigned in groups, called “blocks”, to stream processors, and threads in each block are allowed to share memory and synchronize since they run on the same multiprocessor. Several options exist for programming the GPU with Fortran. OpenACC, which is similar to OpenMP, is a set of specifications of compiler directives for programming accelerator devices in which directives are added into the code as comments [3]. Compute Unified Device Architecture (CUDA) Fortran is another API that allows for lower-level memory management and device-code programming [4].

2 Background

2.1 Structure of ORBIT

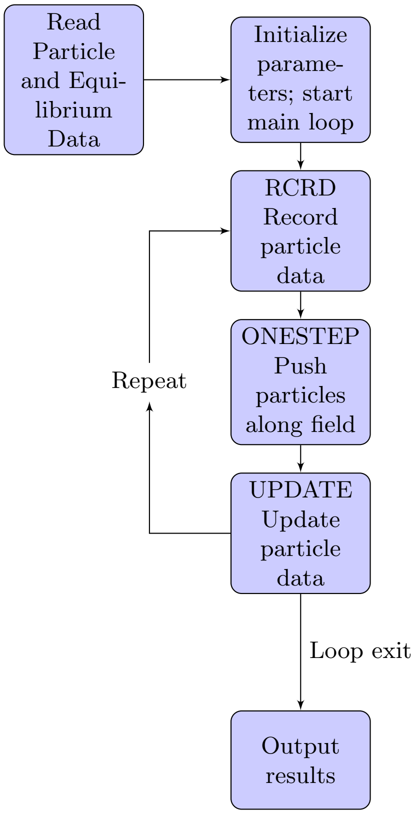

Read Particle and Equilibrium Data
Initialize parameters; start main loop
RCRD Record particle data
ONESTEP Push particles along field
UPDATE Update particle data
Repeat
Loop exit
Output results

Figure 1: The structure of the original ORBIT code. The main loop increments by a step each time, and loops over particles are inside the RCRD, ONESTEP, and UPDATE subroutines.

During an execution of ORBIT, the run begins by initializing the user-set parameters. Then it reads in magnetic field, perturbation, and particle data to initialize the field splines and particle coordinates. Afterwards, it enters a main loop that loops over timesteps. During each step, the particles are advanced in time according to their equations of motion in the magnetic field, and phase space data, related to that motion, are aggregated at the end of the step. This main time-advanced loop is the most compute-intensive part of the code and thus the focus of the majority of our acceleration techniques. After this loop, the particles are analyzed again, and results are written out to files.

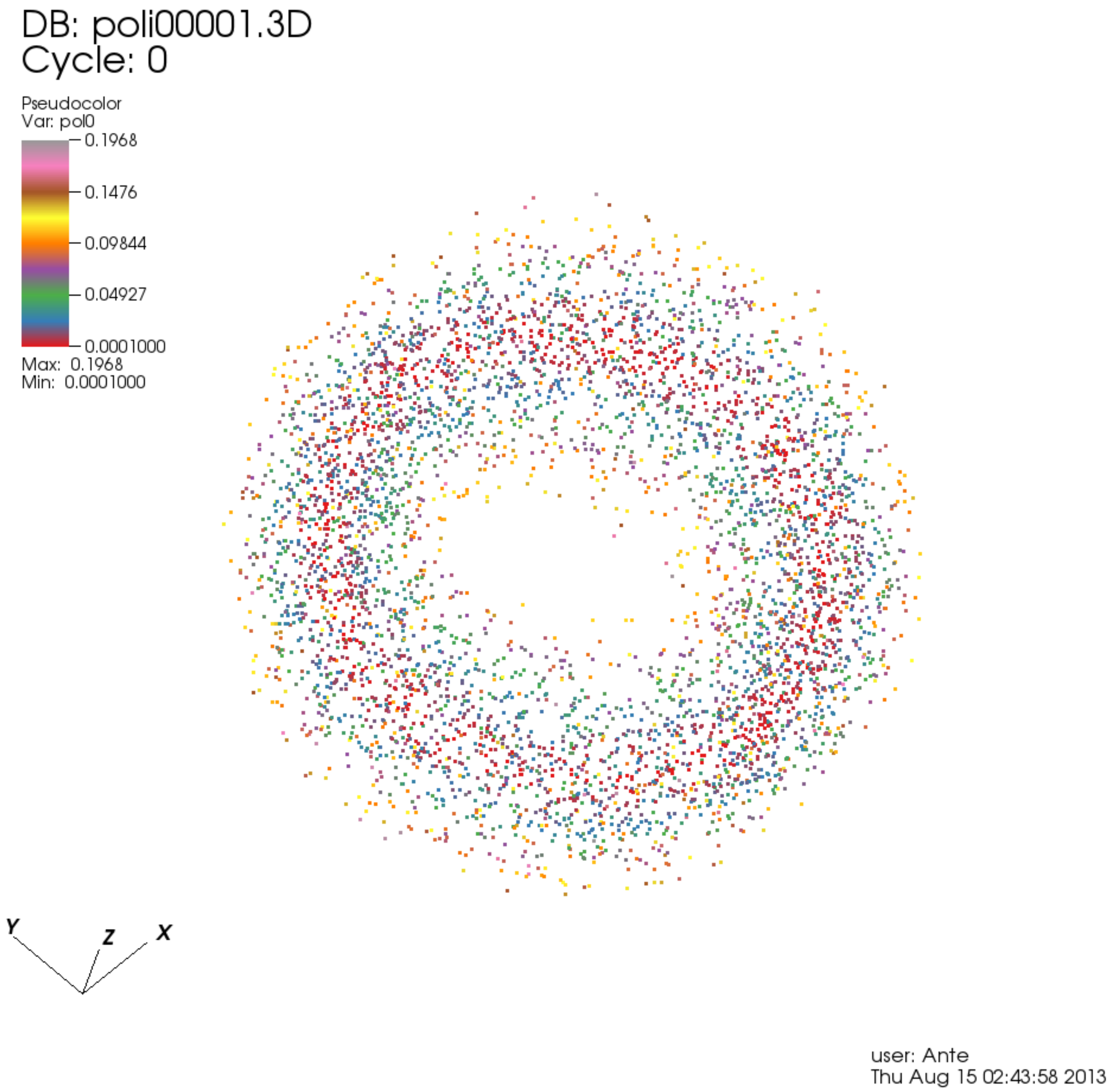


Figure 2: An ORBIT simulation of 5000 particles at its first step. New code was created to make it possible to visualize simulations in progress. “pol0” refers to the particle’s initial poloidal flux.

2.2 OpenMP Overview

Shared-memory parallelism can be rather complex when implemented with explicit calls to the “PTHREADS” (POSIX threads) library [5]. In such cases the programmer has to explicitly manage the threads and their access to memory, making sure to put “locks” on memory locations that could be modified by several threads simultaneously in order to avoid incorrect results. Fortunately, the details of this complexity go away when using OpenMP directives. The programmer still needs to consider potential memory-access conflicts between the threads, but the implementation requires only simple compiler directives. OpenMP threads are created upon entrance of a “parallel region”, defined by the OMP PARALLEL directive. In Fortran, a parallel region can be defined by adding “OMP PARALLEL” before it and “OMP END PARALLEL” after it. Within the region, each thread runs every line of code but on a different processor core. In addition, clauses can be used to specify, for each variable, whether it is private to each thread or shared among the threads. OpenMP also provides the “DO” directive to split and schedule the work of a Fortran “DO” loop among the threads, with clauses that give the flexibility of setting a “static” or “dynamic” work schedule. Other than directives, there are APIs that provide access not only to information such as thread ID and total number of threads but also to environmental variables that can be used to set the maximum thread count and thread stack size.

2.3 OpenACC Overview

While OpenMP simplifies the implementation of multi-threaded parallelism on CPUs, OpenACC tries to do the same for the GPGPU hardware. OpenACC shares many features in common with OpenMP; however, since it is still in its infant stage, it lacks many of the APIs and much of the functionality of OpenMP. In addition, OpenACC has directives for copying data back and forth between the CPU (known as the “host”) and the GPU (known as the “device”). OpenACC parallelism is divided into three levels: gang, worker, and vector, which are meant to correspond well to the structure of GPU parallelism. Upon entrance to a parallel region, OpenACC creates multiple gangs that each execute the code. Gangs cannot communicate with or synchronize with each other. Within a gang, however, one can have workers that can execute code in parallel and synchronize each other. For code that is vectorizable, each worker can then execute the code with vector-level parallelism, which is also known as “data parallelism”. Vector-level parallelism is designed for SIMD processing, where the cores can run the same instruction on multiple units of data. In many OpenACC directives, the programmer can specify how intensely each level of parallelism is to be used, as well as which levels to share the work among.

2.4 CUDA Fortran Overview

CUDA Fortran [4] requires the programmer to generate separate code for the device (the GPU) from the host (the CPU). The separate code is written in one separate module, and the compiler converts it into CUDA code. Each subprogram has a host, global, or device attribute. Only code with a global or device attribute can be run on the device, and only a global or a device subprogram can call a device subprogram. Device code is started by a host subprogram calling a global subprogram with specifications on the dimensions of the kernel (here “dimensions” refer to the number of blocks and threads in each block). In addition, many APIs are provided for getting device configurations, managing data in memory, and controlling thread synchronization.

3 Methods

Our project consisted of two main steps, implementation and testing.

3.1 Implementation

3.1.1 OpenMP

Shared memory multiprocessing was implemented using OpenMP directives, which allowed for leaving most of the code intact. Another benefit was that debugging, done by checking through the results and comparing them against the results of the serial version, was easy – we simply needed to remove the “-mp” flag on the PGI Fortran compiler for it to ignore the OpenMP directives. The simplest implementation was to add parallel work-sharing regions (PARALLEL DO) around every loop over the particles within the main time-stepping loop. Example code:

```
!$OMP PARALLEL DO PRIVATE(NDUM)
      DO K = 1, NPRT
            NDUM = K * K
            NSQRS(K) = NDUM
      ENDDO
!$OMP END PARALLEL DO
```

Another implementation, designed to minimize the overhead of thread creation and destruction upon entering and exiting parallel regions, involved defining one large parallel region around the main loop, statically calculating the exact particles each thread would work on, and then performing, for each thread, the entire main loop (including the result aggregation) for only those particles. Example code:

```
      STPRT = NPRT * OMP_GET_THREAD_NUM() / OMP_GET_NUM_THREADS() + 1
      ENDPRT = NPRT*(OMP_GET_THREAD_NUM() + 1) / OMP_GET_NUM_THREADS()
      DO K = STPRT, ENDPRT
            NDUM = K * K
            NSQRS(K) = NDUM
      ENDDO
```

After the main loop, the aggregate results from each thread are combined such that they match the results that would have been generated without parallelization. Figure 3 illustrates the implementation with one large parallel region.

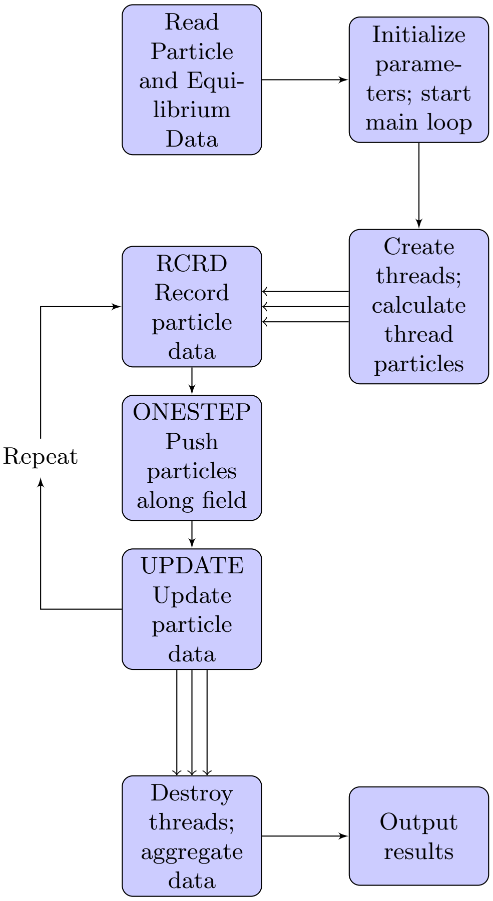

Read Particle and Equilibrium Data
Initialize parameters; start main loop
Create threads; calculate thread particles
RCRD Record particle data
ONESTEP Push particles along field
UPDATE Update particle data
Repeat
Destroy threads; aggregate data
Output results

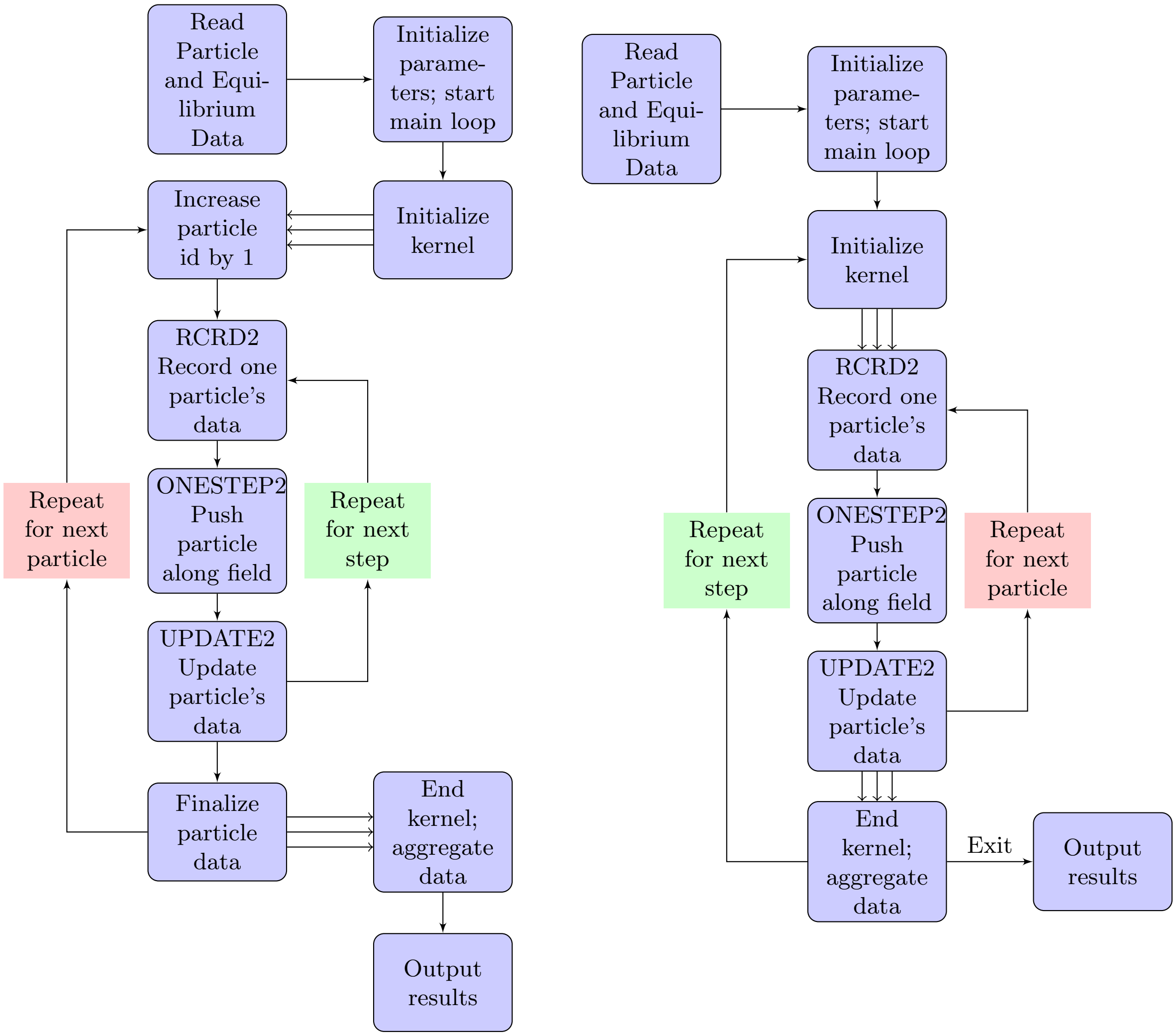
Figure 3: The structure of the OpenMP One-Region code. Threads are only created or destroyed once.

3.1.2 OpenACC

For our GPGPU port, we first investigated OpenACC directives. These directives share earlier-mentioned benefits of OpenMP, which are the low impact on the source code, the ease of debugging, and the straightforward comparison with the serial version of the code by simply removing the "-acc" compiler flag. The simplest OpenACC implementation, similar to the simplest OpenMP implementation, was to add kernels directives around every loop and a data region around the main loop. Example code:

```
!$ACC PARALLEL LOOP COPY(NSQRS) PRIVATE(NDUM)
     DO K = 1, NPRT
          NDUM = K * K
          NSQRS(K) = NDUM
     ENDDO
!$ACC END PARALLEL LOOP
```

This implementation, however, performed poorly due to the much larger overhead of starting each kernel and synchronizing after they completed. The other implementation involved one parallel region around the entire main loop, similar to the one-region OpenMP implementation. Because of the requirements of OpenACC 1.0, the subprograms in the main loop needed to be inlined, so a separate source code was created for the inlined version. In this version, all the loops that cycled through the particles were combined into one master loop that cycled through the particles. In addition, since "complex branching behavior" was not allowed, many modifications were made throughout the code to remove the majority of if-then and goto statements and produce vectorized equivalents. After these modifications, parallel work-sharing (PARALLEL LOOP) directives were added around the particle loop, and REDUCTION clauses were used to aggregate the data after each step. Two versions of the new main-loop were made: the first version moved the loop through the particles outside the time-step loop, so that the entire main-loop was one kernel, and the second version retained the loop through the particles inside the time-step loop, so that a kernel was restarted after every time-step. Figures 4a and 4b illustrate the differences between the two versions.


Read Particle and Equilibrium Data
Initialize parameters; start main loop
Increase particle id by 1
Initialize kernel
RCRD2 Record one particle's data
Repeat for next particle
ONESTEP2 Push particle along field
Repeat for next step
UPDATE2 Update particle's data
Finalize particle data
End kernel; aggregate data
Output results
Read Particle and Equilibrium Data
Initialize parameters; start main loop
Initialize kernel
RCRD2 Record one particle's data
Repeat for next step
ONESTEP2 Push particle along field
Repeat for next particle
UPDATE2 Update particle's data
End kernel; aggregate data
Exit
Output results

(a) The structure of the OpenACC Simple code. The loop that iterates over particles is outside the loop that iterates the steps for each particle, and the kernel directly parallelizes the particles loop.

(b) The structure of the OpenACC Kernels code. The loop that iterates over particles is inside the loop that iterates the steps for each particle, and kernel is restarted for every step.

Figure 4: The structure of different OpenACC implementations. The outside and the inside loops are swapped in the two versions, and so one version has only one kernel while the other version has many kernels. In both versions, the subroutines are inlined.

3.1.3 CUDA

Another method of implementing GPGPU was through CUDA Fortran. Similar to OpenACC, a separate module was created. Subroutine calls are allowed in CUDA, so changes made to the structure of the code during implementation were minimal. The simplest implementation that matches the GPU architecture well was to create one thread for every particle and modify the subroutines to act on only one particle at a time.

```
      CALL STEPKERNEL<<<(NPRT - 1) / 64 + 1, 64>>>
```

CUDA provides for greater control over memory management; however, this greater control also means more work for the programmer and more modifications to the code. First, the constants, magnetic field splines, and particle data were copied into the global memory of the device. In the fully optimized implementation, the splines were bound to textures and accessed in texture memory, and the particle data were loaded into the registers during each kernel. Implementations lacking either optimization were also tested (one without textures, and one without loading particle data into the registers). Next, a kernel was started for a preset number of steps, and within each thread, the particle data were loaded into registers. The thread then called the subroutines for particle pushing, and then some data were aggregated for the block. At the end of the kernel, data were unloaded from the registers into the global memory of the device and kept there. The aggregated data for the block were passed back to the host, and then the host would start the next kernel while aggregating the aggregated data across the blocks. This loop would continue until all the time-steps were completed. Our CUDA Fortran implementation is illustrated in Figure 5.

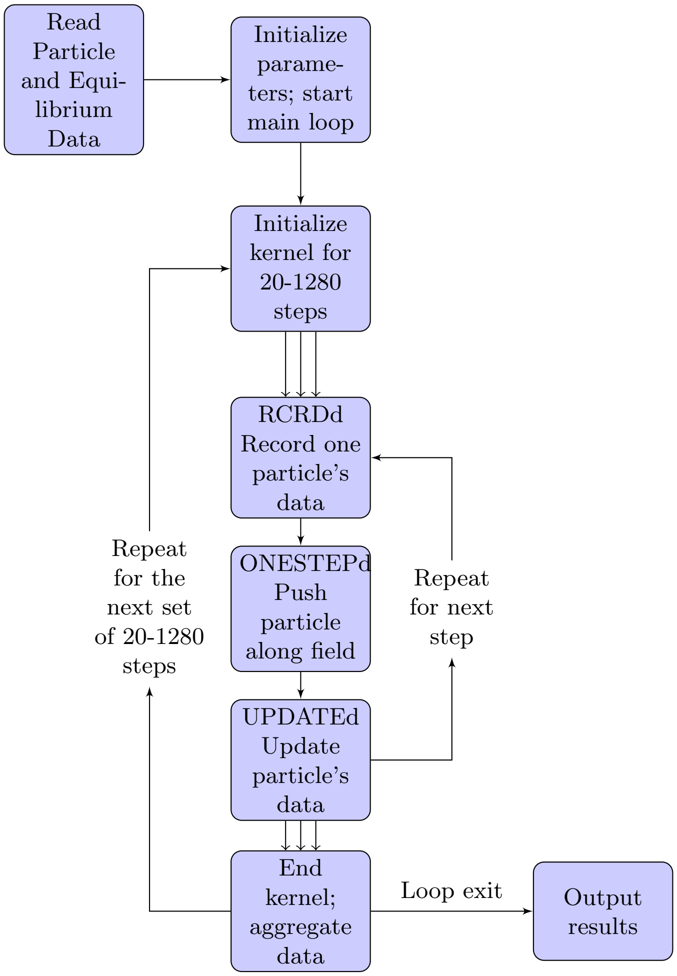

Read Particle and Equilibrium Data
Initialize parameters; start main loop
Initialize kernel for 20-1280 steps
RCRDd Record one particle's data
ONESTEPd Push particle along field
UPDATEd Update particle's data
End kernel; aggregate data
Output results
Loop exit
Repeat for the next set of 20-1280 steps
Repeat for next step

Figure 5: The structure of the CUDA Fortran code versions. This structure is very similar to the OpenACC Simple code structure, except the subroutines are not inlined, and the kernels have a large enough dimension such that each thread only pushes one particle.

3.2 Testing

To assess the performance, we tested ORBIT on 500 toroidal transits of the "nplot = 2" option, which studies diffusion and particle loss. The default particle count (nprt) for this setting is 5000. However, we also tested nprt = 25000 since the goal was to significantly increase the number of particles, and speedup results might differ as the GPU got more saturated. We followed the following procedure:

1. Run the original ORBIT code three times on a test server for nprt = 5000 and three times for nprt = 25000.

2. For each implementation, determine a set of possible configurations to test.

3. Run each configuration three times for nprt = 5000 and three times for nprt = 25000.

4. Choose the configuration with the best average, and compare separately the runtime for nprt = 5000 and nprt = 25000 to the original ORBIT code.

By testing each configuration three times, the standard error of the average is expected to be small enough such that any selection bias from picking the best configuration would not be misleading in its expected performance.

For OpenMP, the configurations we tested varied the maximum threads allowed, from 1 to 64. For OpenACC, the configurations we tested were the choice of accelerator (between Fermi and Kepler) and the choice of loop configuration (which loop is on the outside). For CUDA, the configurations we tested were the choice of accelerator, the block size (number of threads per block), and the GPU to Host Reporting Interval (which sets frequency at which kernels synchronize and report to the host).

To assess the correctness, we used the Linux "diff" command to count the number of particles that differed between the distf.plt (which is the list of final particle data of remaining particles), lostf.plt (the list of particles that exited), dist.plt (the list of initial particle data), and diffusion.plt (the diffusion data at various time steps) outputs of the original and each implementation.

4 Results

After performing our testing procedure, these were the average loop runtimes, in seconds, of the best configuration for each implementation:

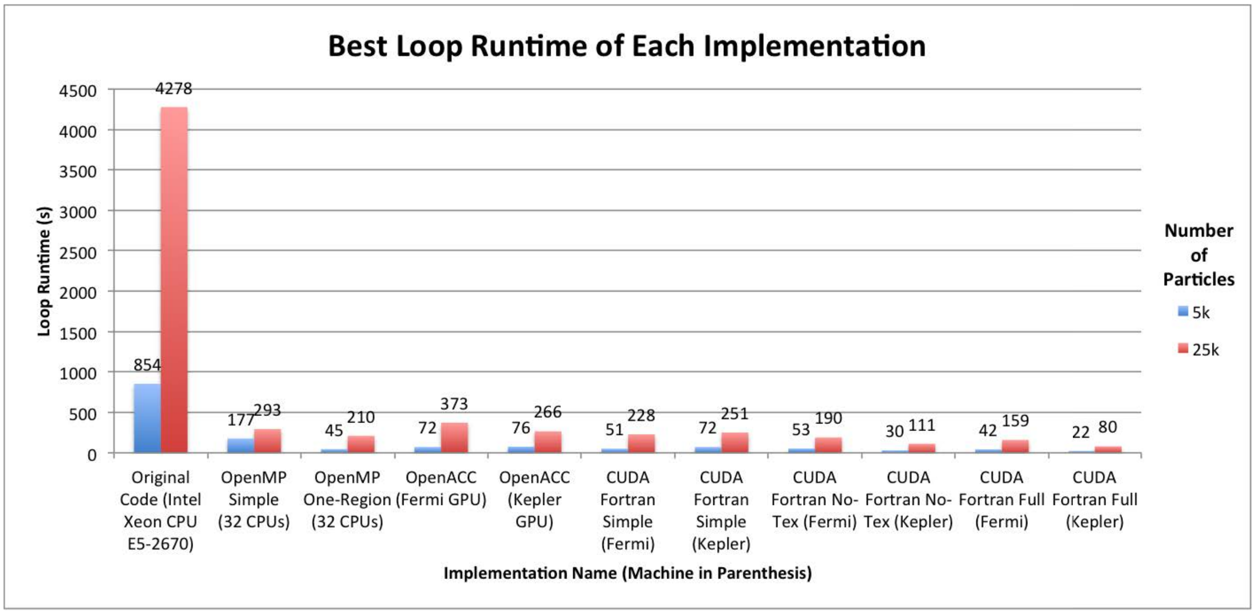


Figure 6: Bar graph of loop runtime in seconds. The original code benchmark times are the left-most bars.

As we can see, the loop runtimes of all the accelerated versions were small compared to the benchmark loop runtime. We can see the differences between them better by plotting the speedups, which are essentially the reciprocals of these runtimes, scaled to the benchmark 5k and the benchmark 25k runtimes:

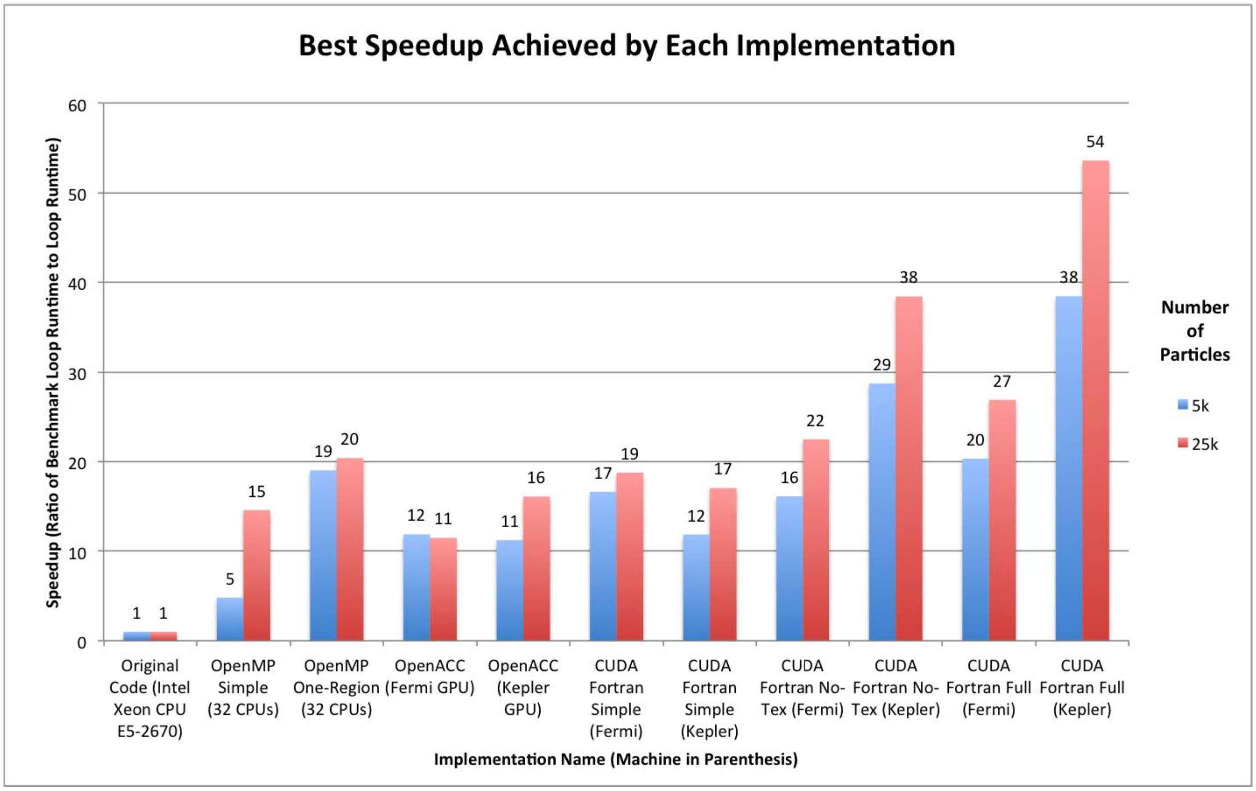


Figure 7: Bar graph of speedups achieved. These are calculated by taking reciprocals of the runtimes and scaling them with the original code runtimes.

With the best implementation, we achieved a 53.59 ± 0.07 times speedup for 25k particles. While the original code took a bit over an hour to run on the Intel® Xeon® CPU E5-2670, it now takes only 80 seconds to run on the NVIDIA Tesla K20c GPU.

4.1 Benchmark

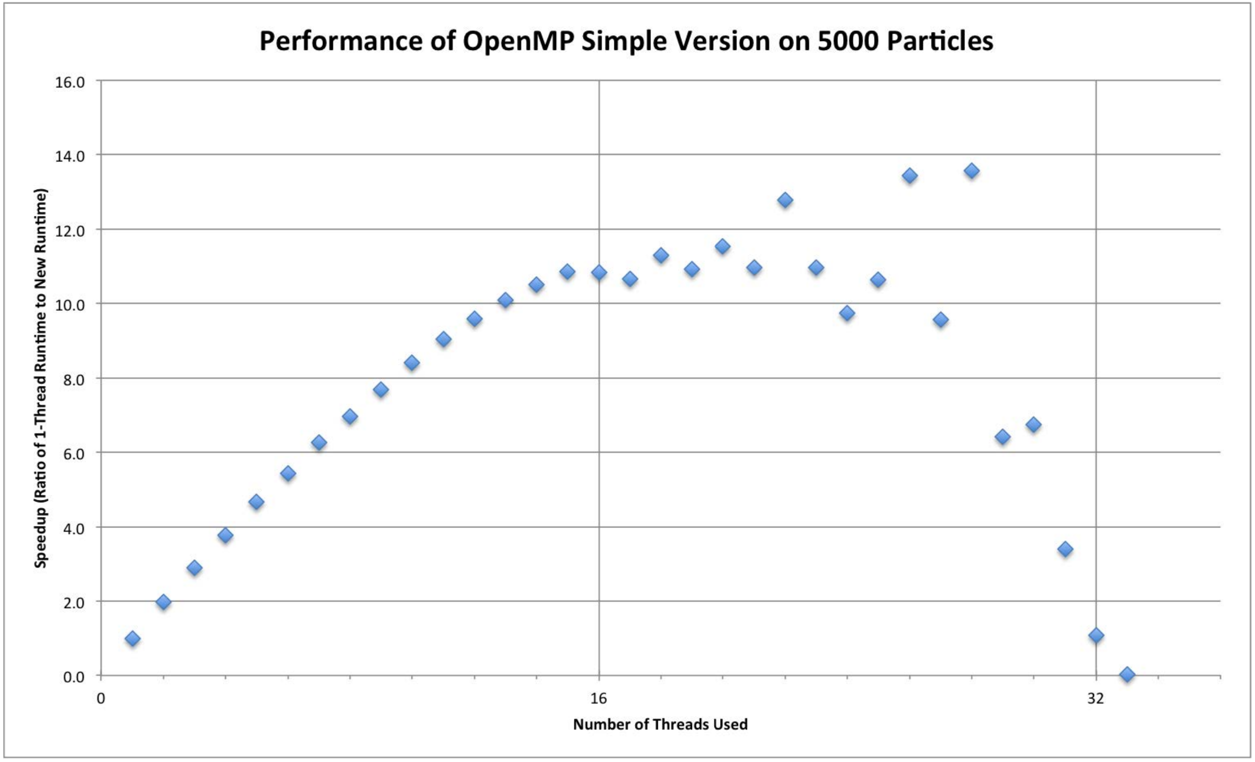
We tested the original ORBIT code on a separate computing node that uses Intel® Xeon® CPU E5-2670 (Sandy Bridge). For runtime, the code was tested under the "nplot = 2" setting for 500 toroidal transits (ntor), three times using 5000 particles and three times using 25000 particles. The average (median) loop runtime for 5000 particles was 0.2373 ± 0.0001 hours, and the median loop runtime for 25000 particles was 1.1882 ± 0.0004 hours. For 5000 particles, the results for ntor = 5, 25, 50, 75, 100, 200, 300, 400, and 500 were saved as the benchmark for testing the correctness of the other implementations.

4.2 OpenMP

We tested two implementations of OpenMP, one by systematically adding directives around every loop, called the "simple" version, and one by combining everything into one parallel region, called the "one region" version. In all our OpenMP implementations, static scheduling was used. We tested each version under the "nplot = 2" setting for 500 toroidal transits.

4.2.1 OpenMP Simple Version

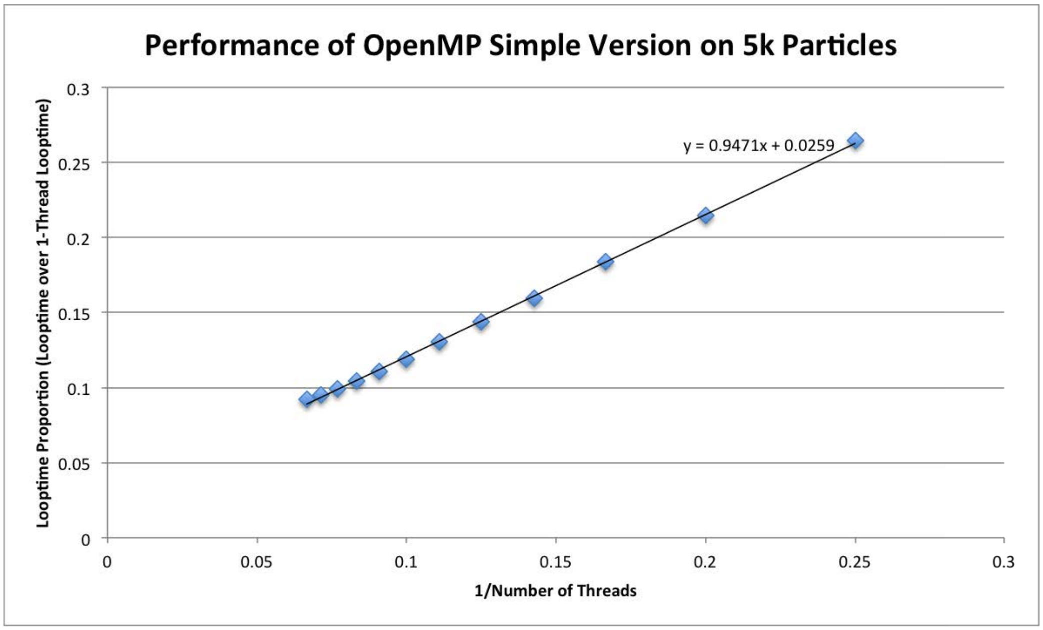
We tested the speedup of this version on 32 requested processors on the gpusrv computing cluster, which uses Intel® Xeon® CPU E5-2670 processors. Each possible number of threads, from 1 to 33, was tested three times, and the runtime for the main loop was recorded. Each data point represents the median of the three corresponding runs.



The maximum speedup is 13.6±3.1 times faster than the one-thread speed, at 28 threads. Note that this number is different from the reported speedup in the overall results plot (Figure 7), because this speedup compares the 28-thread runtime against the 1-thread runtime rather than against the benchmark.

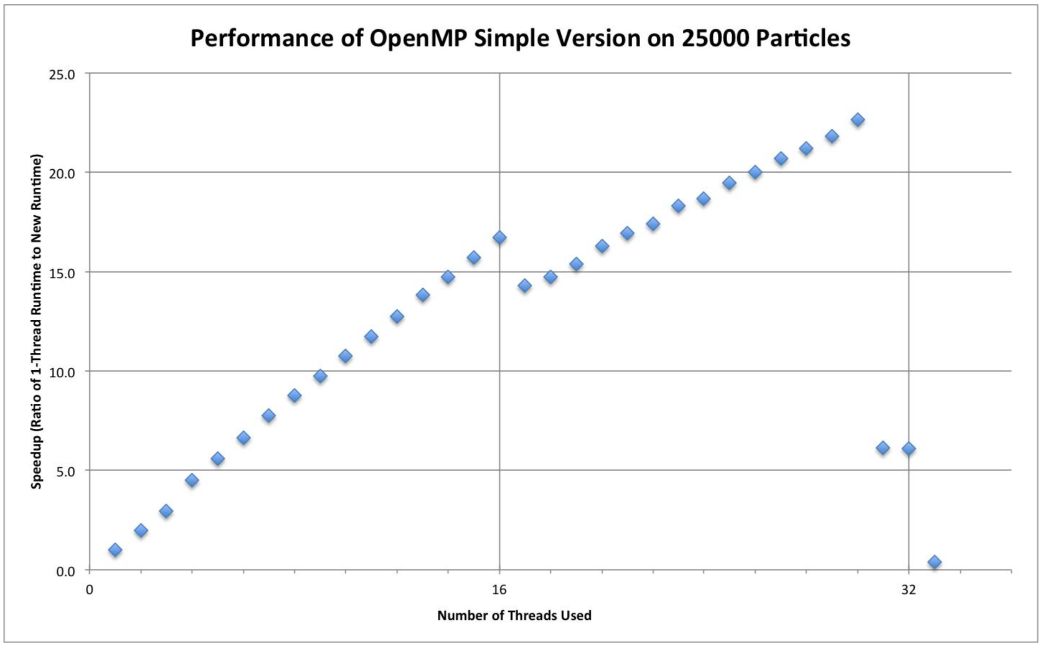
As we can see, as the number of threads gets larger, the speedup increase slows down. The speedup starts to roughly approach a horizontal asymptote, implying there is a significant overhead and non-parallelizable (serial) region. Assuming that the runtime is of the form $T = O + L/N$, where T is the Total Loop Runtime, O is the sum of the overhead and nonparallelizable runtime, L is the parallelizable runtime, and N is the number of threads, we can find out the overhead and non-parallelizable runtime, O, by plotting T vs. $1/N$ and finding the T-intercept.

Since the NUMA architecture has about 16 processors per node, we choose $N = 16$ as the maximum we need for this plot. In addition, the first three points would be too influential (to the slope), so we can start at $N = 4$. This is the plot of T vs. $1/N$, for N from 15 to 4:



The T-intercept, which represents the size of the overhead and the non-parallelizable region, is 0.0415 ± 0.0022, or 3.1% of the original 1-thread looptime, implying a theoretical speedup upper bound of 32.3 times. The large overhead is mostly due to the fact that there are multiple parallel regions.

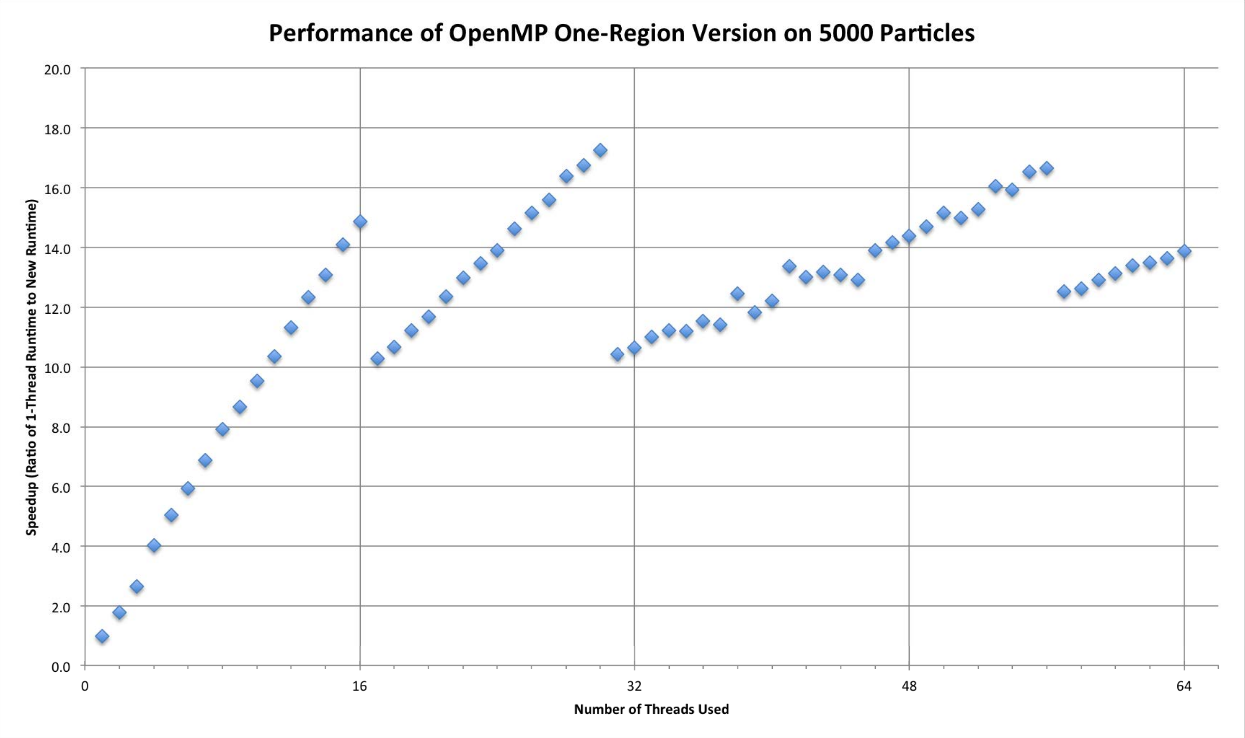
We also analyzed the speedup with 25k particles, to see how well it scales:



The maximum speedup is 22.7±0.0 times, at 30 threads. (Also different from the Figure 7 number, for the same reason as for the 5k particle tests.) The speedup appears to get slightly better with 25k particles, and the data points for 15 or more threads become less bumpy and stick much closer to their respective lines.

4.2.2 OpenMP One-Region Version

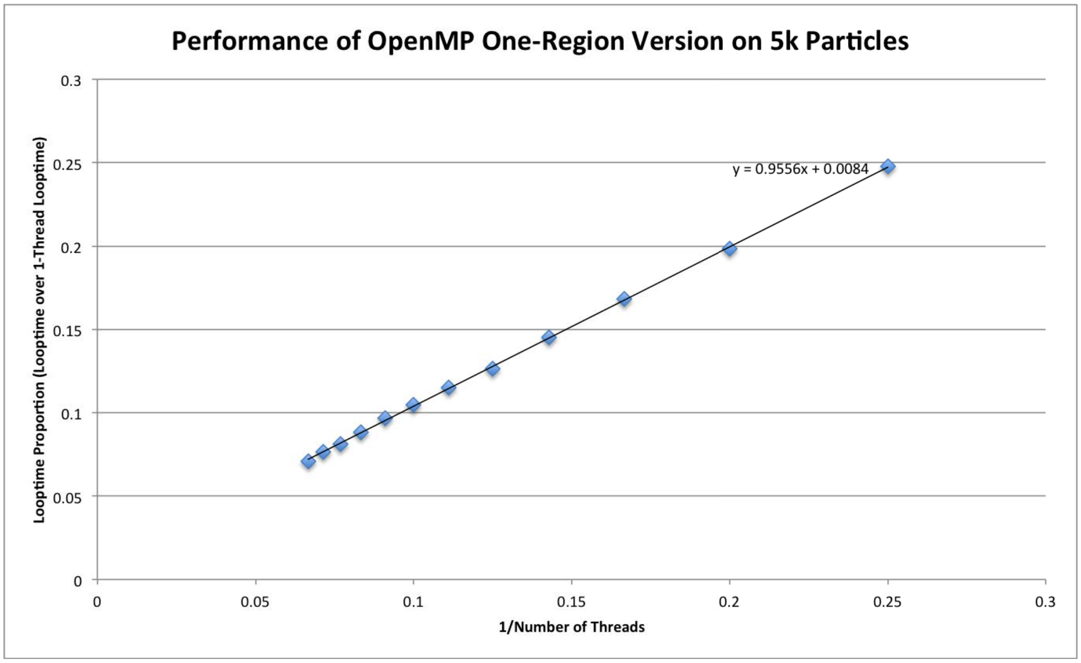
We tested the speedup of this version on 32 requested processors on the gpusrv computing cluster, which uses Intel® Xeon® CPU E5-2670 processors. Each possible number of threads, from 1 to 64, was tested three times, and the runtime for the main loop was recorded. Each data point represents the median of the three corresponding runs.



The maximum speedup is 17.3 ± 0.1 times the one-thread speed, at 30 threads.

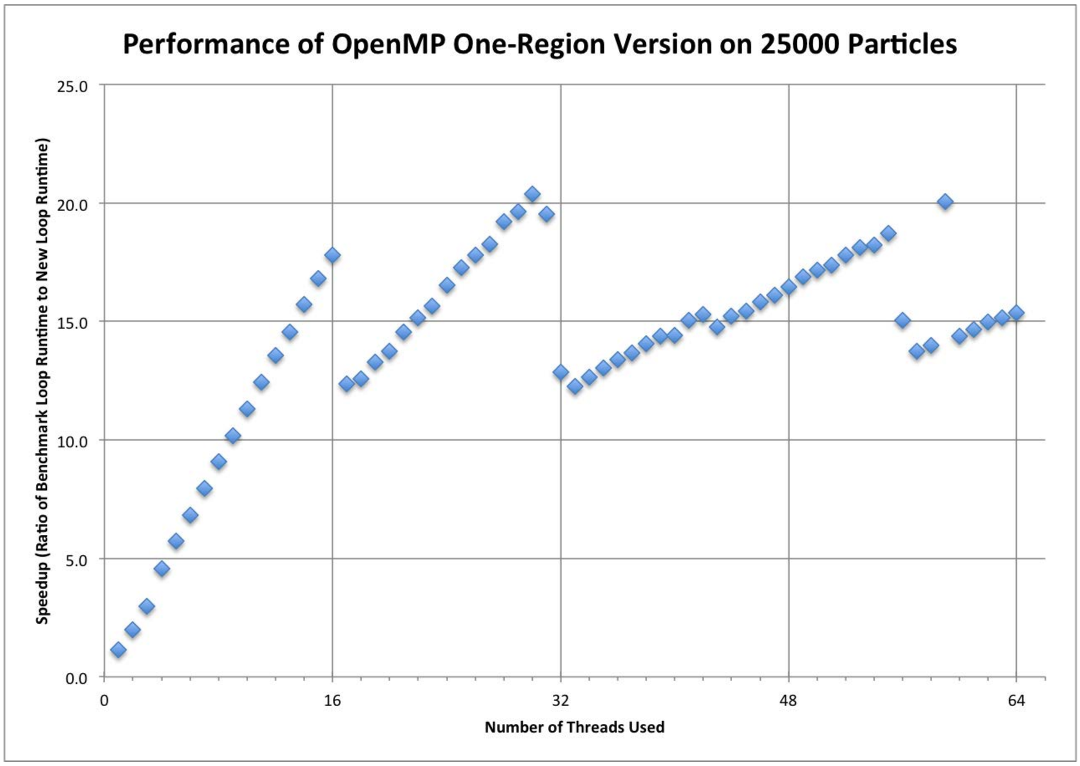
Since we used static scheduling, every thread has an equal workload assigned. As we can see from the data, each time the threads use a new NUMA (Non-Uniform Memory Access) node, the few extra threads placed into the new node slow down the entire run because of delayed access to the shared memory. If the work were scheduled dynamically, we might be able to see a more even curve with the runtime, but the straight lines help us see the NUMA architecture of the server.

Using the same exact analysis of the overhead, the T-intercept is now significantly less:



As we can see, the T-intercept is 0.0076 ± 0.0006, which is only a third of that of the OpenMP Simple version. The overhead and the non-parallel region, therefore, are only 0.87% of the original looptime, implying a theoretical speedup upper bound of 115 times.

We also analyzed the speedup with 25k particles, to see how well it scales:



The maximum speedup is 20.4±0.1 times, at 30 threads. The speedup appears to get slightly better with 25k particles, and the data points form more linear patterns, with less bumpiness, indicating that the speed evens out over the course of calculating the increased number of particles.

4.2.3 Correctness

The outputs of the code from both OpenMP versions matched those of the original ORBIT code. We checked diffusion.plt, dist.plt, distf.plt, lost.plt, and orbit.out.

4.2.4 Summary and Discussion

Since the scaling for the first 15–16 threads were almost linear in all implementations, ORBIT worked very well with OpenMP. OpenMP can be used to easily divide the work among many processors in the same node. The One-Region version scaled better, and thus we will use OpenMP One-Region in the final implementation for runs where no GPU is available.

4.3 OpenACC

We tested two implementations of OpenAcc 1.0 using the pgf 90 compiler: one with the particle loop outside the stepping loop, called the "simple" version, and one with the stepping loop outside the particle loop, called the "kernels" version. We tested each version under the "nplot = 2" setting for 500 toroidal transits on two types of Nvidia GPUs: the Tesla M2070 "Fermi" (on gpusrv) and the newer Tesla K20c "Kepler" (on cppg-6).

On the Fermi GPU with nprt = 5000, the "simple" OpenACC version achieved a speedup of 11.9 ± 0.0 times, while nprt = 25000 achieved a speedup of 11.5 ± 0.1 times. Under the same conditions, the "Kernels" version achieved an absolute speedup of 11.7 ± 0.0 times for nprt=5000, and 11.2 ± 0.1 times for nprt = 25000. The closeness of the speedup values in both cases indicates that 5000 particles are roughly enough to saturate the Fermi GPU. The absolute speedup achieved, however, is lower than that of 30 OpenMP threads on the Intel Sandy Bridge E5-2670 mentioned earlier.

For the newer Kepler GPU with nprt = 5000, the simple version achieved an absolute speedup of 11.2 ± 0.0 times, while nprt = 25000 achieved a speedup of 16.1 ± 0.0 times. The "Kernels" version reaches 10.9 ± 0.0 times and 15.8 ± 0.0 times, respectively. The fact that the Kepler scaled better for nprt = 25000 in both cases means that 5000 is not enough to saturate the Kepler GPU. While the speedup for nprt = 25000 is significantly better on this latest GPU hardware, it is still lower than the absolute speedup of 30 OpenMP threads on the Intel Sandy Bridge processor.

The corresponding speedup values of each test were roughly within 2% of each other. The closeness of the speedups between the "kernels" and the "simple" versions shows that splitting up the particle loop into many parallel kernels does not make it significantly slower. Therefore, there is hope for passing aggregated data from each step out from the GPU onto the CPU. However, as will be described later, we failed to do this because the overhead from creating new parallel regions to aggregate this data is too large.

4.3.1 Performance Discussion

The following issues with OpenACC 1.0 directly affected the performance and speed of the accelerated code:

1. It is not possible to indicate which values to load from global memory into shared, cache, or registers. Since subroutines are inlined, it is not possible to create a separate subroutine, in which particle data are loaded as scalars into registers, for each particle to run. Instead, the particle data must remain as arrays.

2. Complex branching behavior must be removed (possibly through vectorization), which in some sections severely slows the code down.

3. It is not possible to program block-level synchronization (sync must be grid-level).

The following issues also hindered the use of OpenAcc 1.0 for our purpose of accelerating legacy code:

1. Lack of flexibility (e.g., in not allowing branching) makes it very difficult to code for multiple possible settings – this is doable, but either the same subroutine would need to be copied many times or many compiler settings would need to be added.

2. The significant overhead for starting parallel and kernels regions makes it very difficult to perform diagnostics during the run.

3. Inlined code is hard to read, and not easily modified (for example, some subroutines are called more than once, so to modify the code, one would need to modify it in every instance where the subroutine appears.)

4.3.2 Correctness

For the “nplot = 2” setting, data is aggregated in a subroutine called rcrd2. For the OpenACC implementations, we failed to produce the correct output for diffusion.plt using an inlined version of the rcrd2 subroutine. Because of the reasons described in the previous section, we decided it was not worth further correcting the rcrd2 subroutine.

A few of the 5000 particles end up being incorrect, and the number of particles whose results differ increases as the number of toroidal transits increases. The source of this discrepancy appears to be the different floating-point standards used by the Nvidia GPU hardware. One of the main contributions to this discrepancy is the added Fused Multiply-Add (FMA) operation, which is not used in the original benchmark.

Here is a comparison demonstrating this issue:

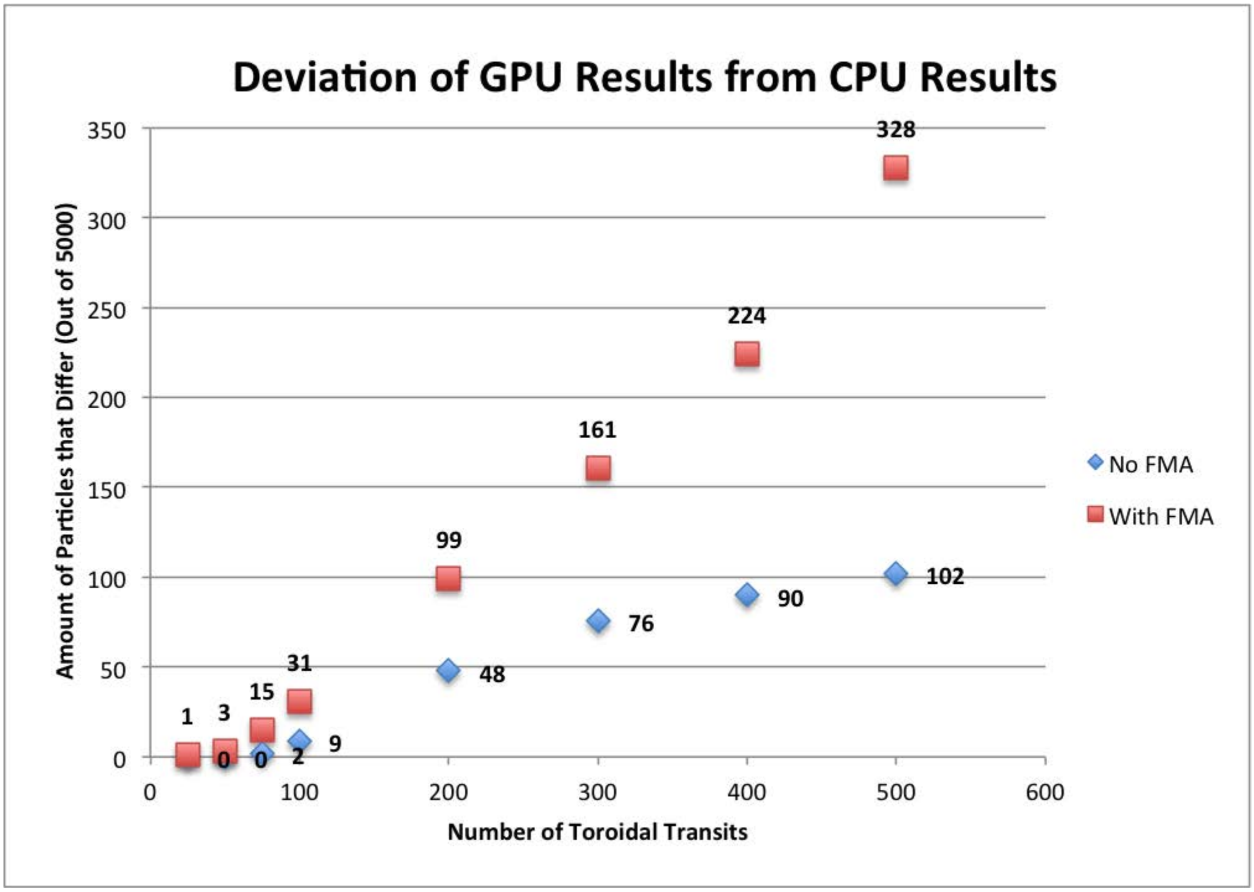


Figure 8: Number of particles that differ in distf.plt, for nprt = 5000. Each toroidal transit takes 200 steps.

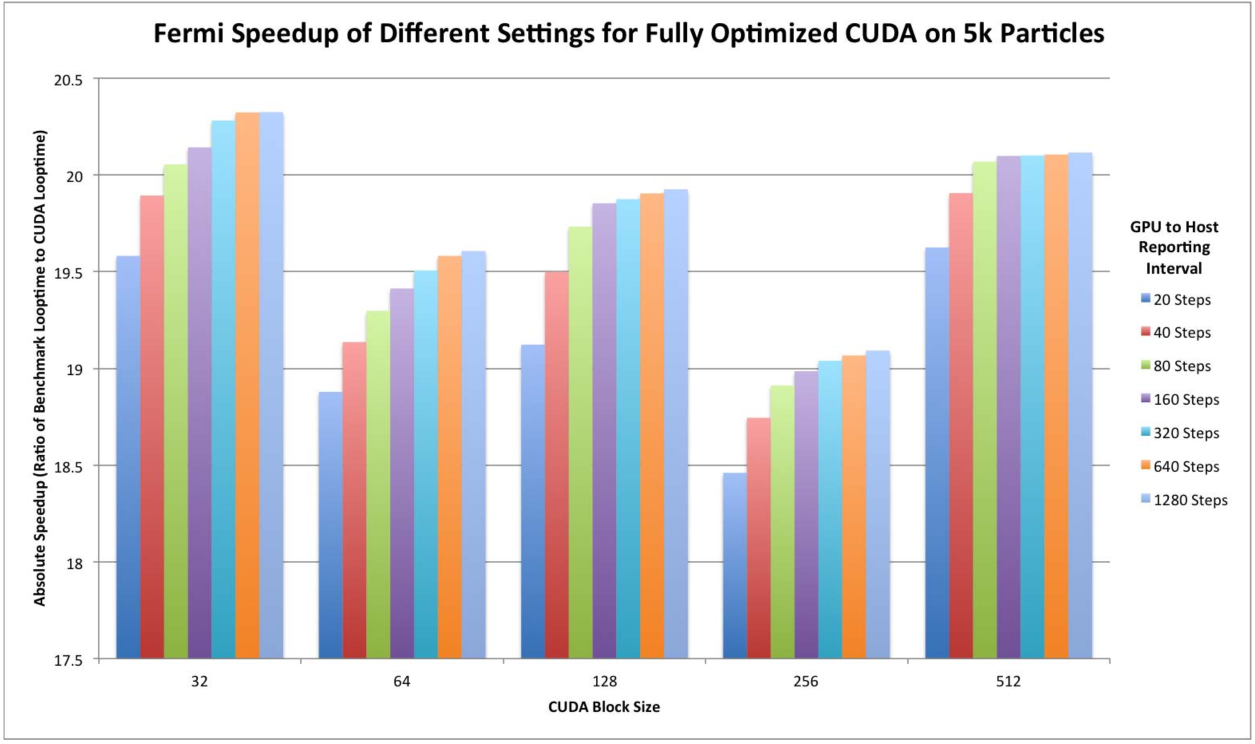
According to this data, removing the FMA operations removed two-thirds of the particles that differed. The other one-third is suspected to be caused by differences in satisfying IEEE floating-point roundoff standards.

4.4 CUDA

We tested three implementations of CUDA Fortran using the pgf90 compiler. The first one, the "fully optimized" version, binds textures to the splines and loads all the particle data into registers during each kernel. To confirm that these two optimizations were the right decisions, we also tested a "no textures" version where the splines are not bound to textures (and everything else is identical to the fully optimized version) and a "simple" version where the particle data are not loaded into registers (and everything else is identical to the fully optimized version). We tested each version under the "nplot = 2" setting for 500 toroidal transits.

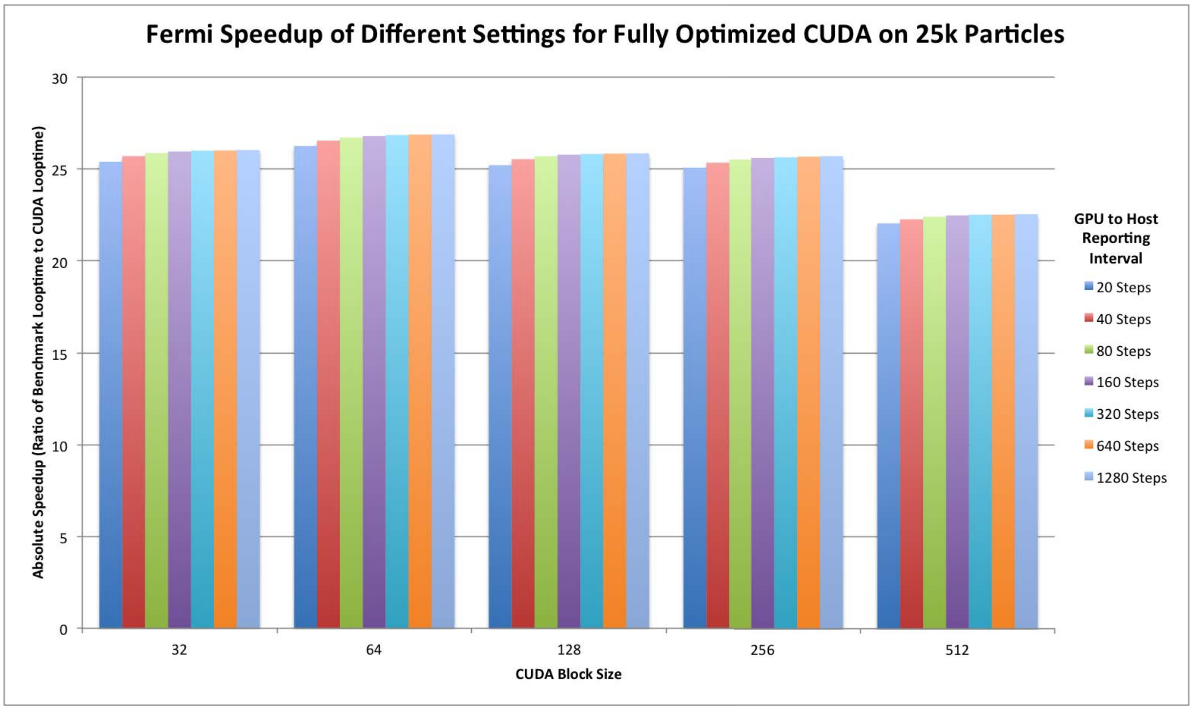
4.4.1 Fully Optimized CUDA Version

We tested the fully-optimized version on both a Fermi GPU (NVIDIA Tesla M2070 on the gpusrv cluster) and a Kepler GPU (NVIDIA Tesla K20c on the cppg-6 server). We tested this code for different block sizes, from 32 to 512, going by a factor of 2 each time, and for different GPU-to-Host Reporting Intervals, from every 20 steps to every 1280 steps, going by a factor of two each time. On the kepler, the kernels did not launch for block size = 512, so the data for this block size were thrown out. Each setting was tested three times, and each bar in the following plots represent the median of the three corresponding runs. These are the results for 5000 particles on the Fermi:



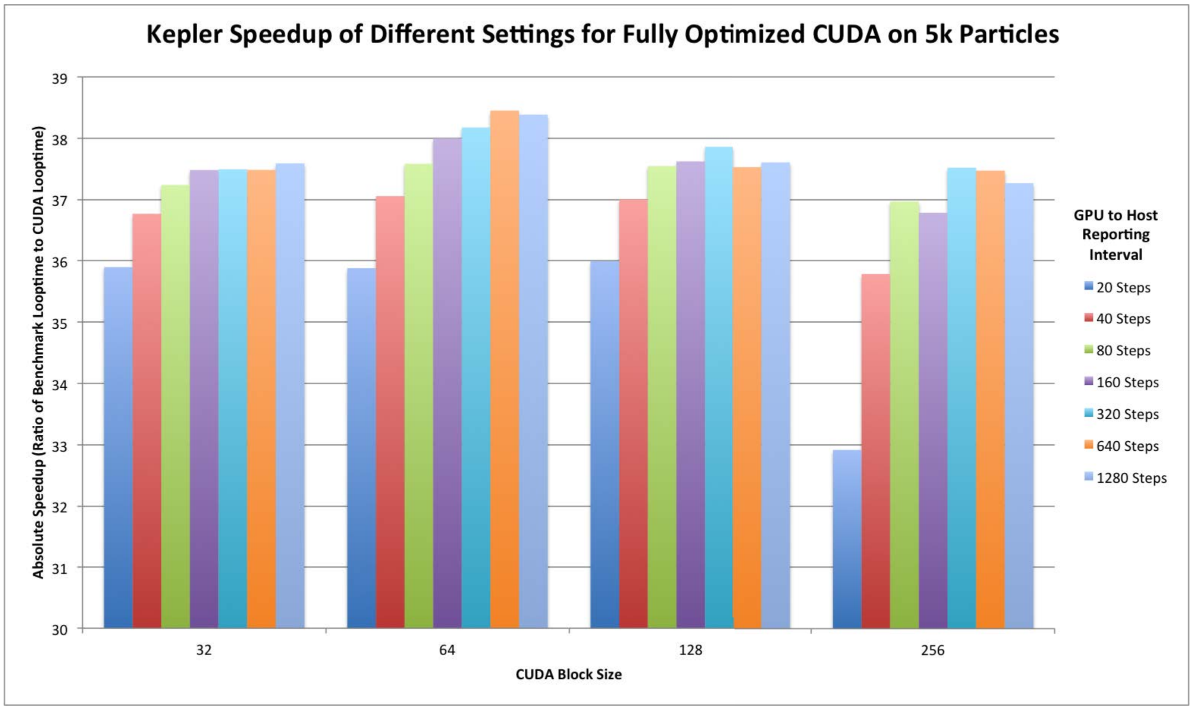
As we can see, the smallest block size and the largest reporting intervals produce the best results. The speedups for the smallest block size, however, fall sharply as the reporting intervals get smaller, because the kernels are synchronized more frequently and data are only aggregated to the block level on the GPU; thus with a small block size there are more data to copy back and aggregate between each kernel. In addition – and this applies to every block size – the largest reporting intervals perform best because synchronization between blocks is the least frequent.

Let us examine the results for 25000 particles on the Fermi:



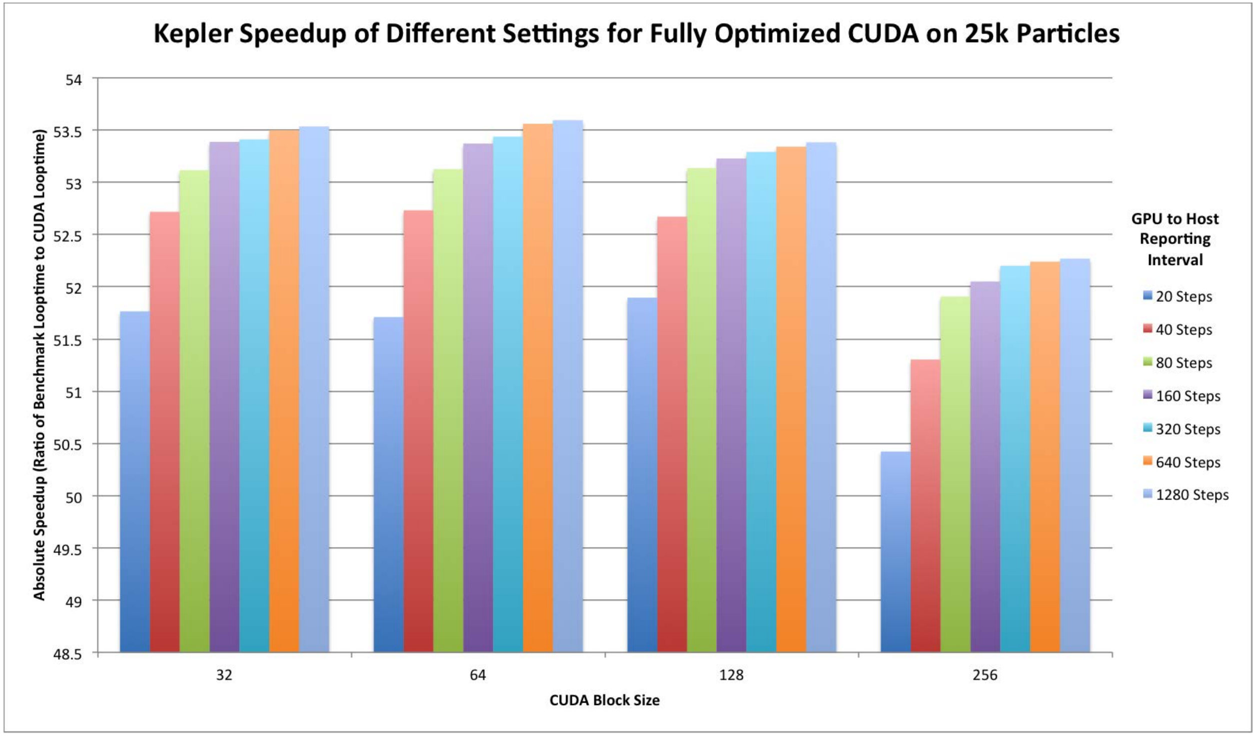
We find a similar effect as with the 5k particle results, except the block size of 64 is slightly better. In addition, the absolute speedups significantly improved (by about 35%), indicating that the GPU was not saturated with 5000 particles.

Now let us examine the results for 5000 particles on the Kepler:



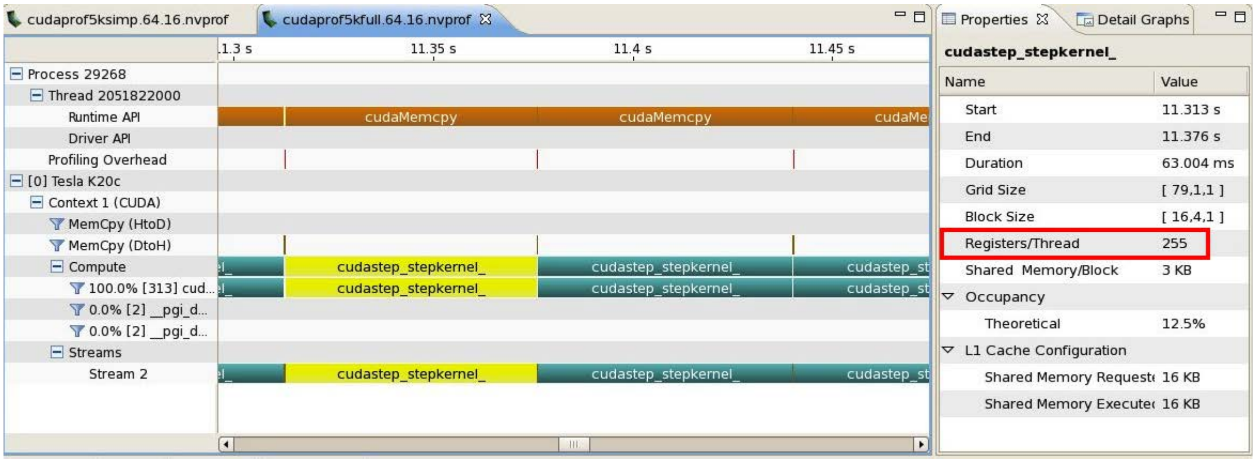
It appears that on the Kepler, as with the 25k particles ran on the Fermi, the block size of 64 is the best. This is most likely because our implementation of reduction only runs half the threads, and as a result, 64 is the smallest block size that can use a full warp to perform reduction. In addition, the reporting interval of 640 steps is better than 1280 steps; however, this is by less than a percent and thus is not significant.

Results for 25000 particles on the Kepler:



We see similar effects here, and we note that the absolute speedups improved again by about 40%, indicating that 5000 particles are not enough to saturate the GPU.

Upon examination of the execution profile generated by the NVIDIA "nvvp" profiling tool, we learned that the memory copyback for the data aggregation takes a negligible amount of time, which is good, and that this implementation uses all 255 available registers:



cudastep_stepkernel_

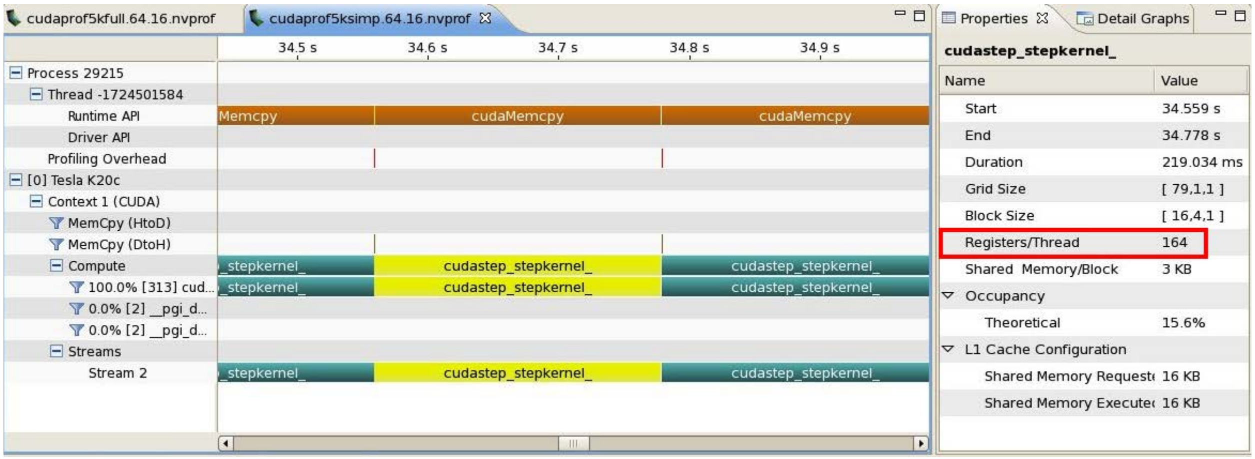
Name	Value
Start	11.313 s
End	11.376 s
Duration	63.004 ms
Grid Size	[79,1,1]
Block Size	[16,4,1]
Registers/Thread	255
Shared Memory/Block	3 KB
Occupancy	
Theoretical	12.5%
L1 Cache Configuration	
Shared Memory Request	16 KB
Shared Memory Execute	16 KB

Figure 9: An NVIDIA Visual Profiler (nvvp) profile of a CUDA Fully-Optimized Version run.

In order to test if a version that uses fewer registers would perform better, we created a simple version, where the particle data is left in global memory during the main loop.

4.4.2 Simple CUDA Version

The nvvp profile from the simple version shows that it uses significantly less registers:



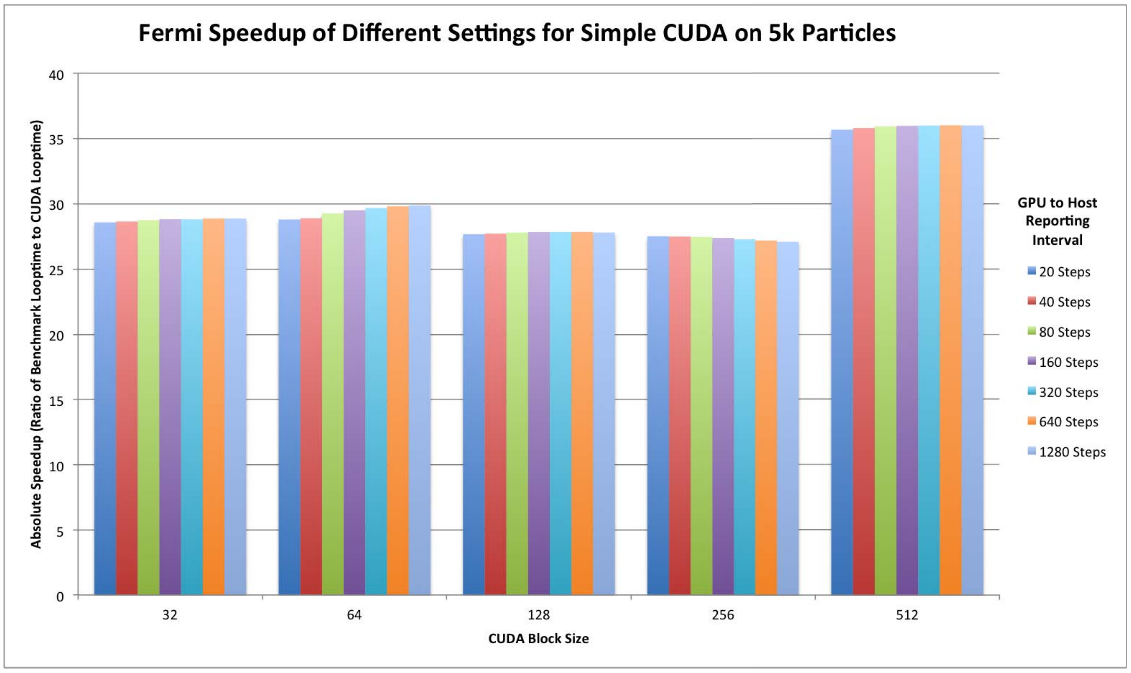
cudastep_stepkernel_

Name	Value
Start	34.559 s
End	34.778 s
Duration	219.034 ms
Grid Size	[79,1,1]
Block Size	[16,4,1]
Registers/Thread	164
Shared Memory/Block	3 KB
Occupancy	
Theoretical	15.6%
L1 Cache Configuration	
Shared Memory Request	16 KB
Shared Memory Execute	16 KB

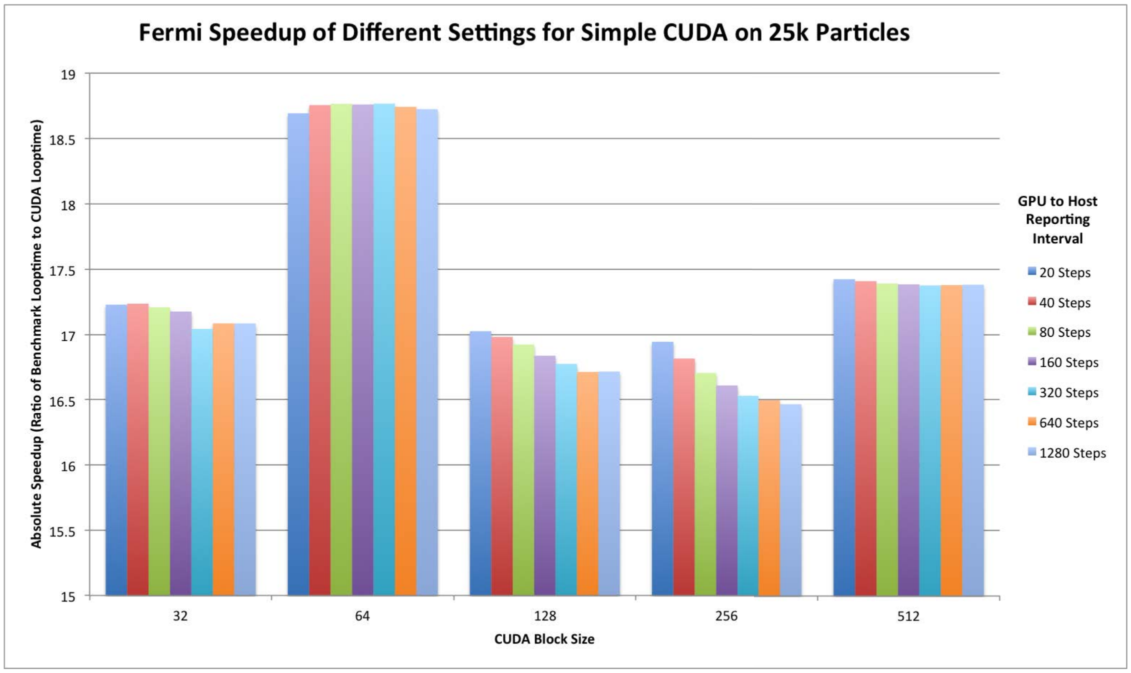
Figure 10: An nvvp profile of a CUDA Simple Version run. We can see in the red box that it uses 164 registers.

Runtime results of the simple CUDA version:

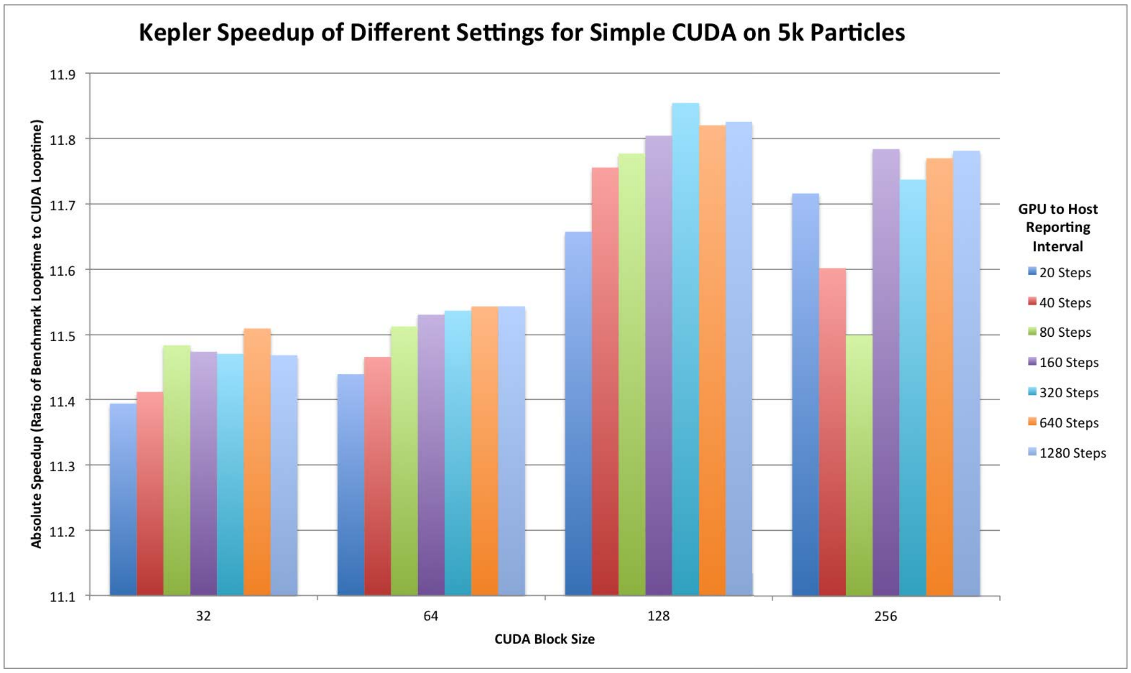
These are the results for 5000 particles on the Fermi:



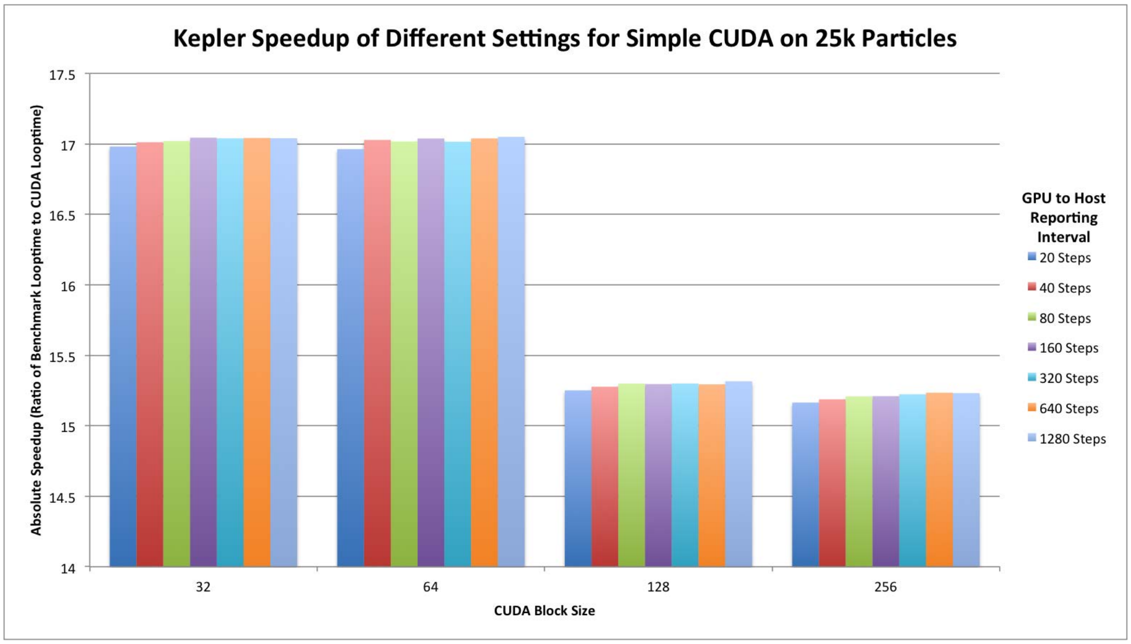
These are the results for 25000 particles on the Fermi:



These are the results for 5000 particles on the Kepler:



These are the results for 25000 particles on the Kepler:

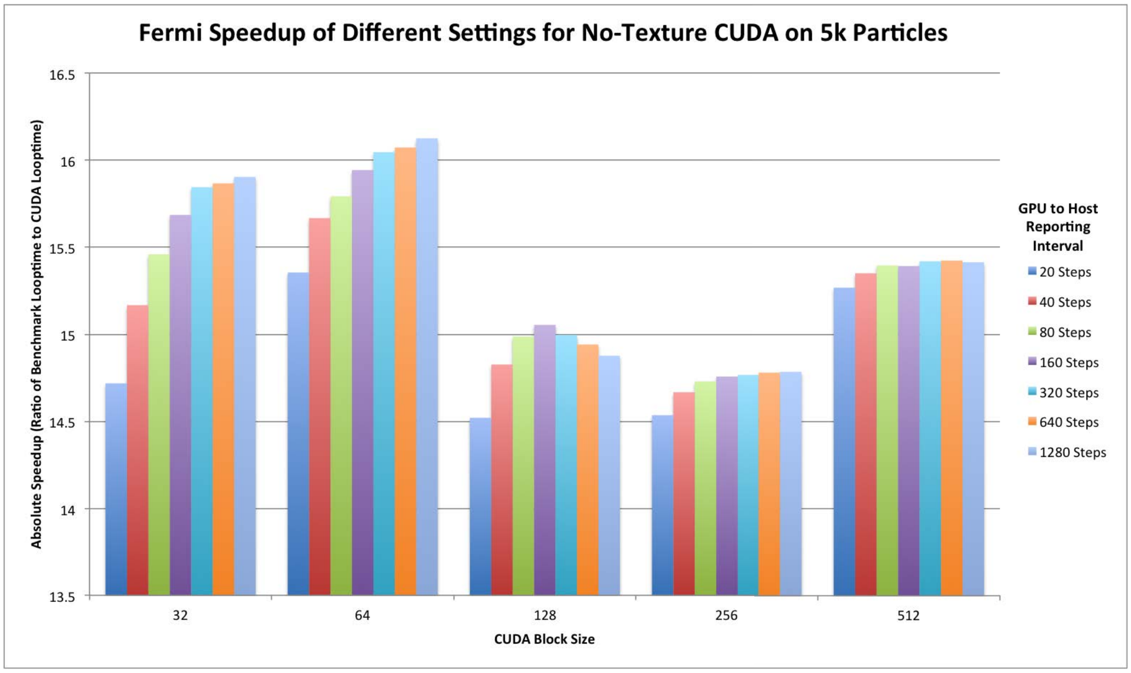


As we can see for the Kepler, the simple version produced results that ran at one-third the speed of the fully optimized version. For the Fermi, the simple version also produced significantly worse results for 25000 particles, and 20% worse results for 5000 particles. Overall, working with the particle data in global memory did not improve performance, and produced results similar to OpenACC 1.0 speedups, which on the other hand further supports the hypothesis that the OpenACC code simply loads particle data into global memory, without further support for memory management.

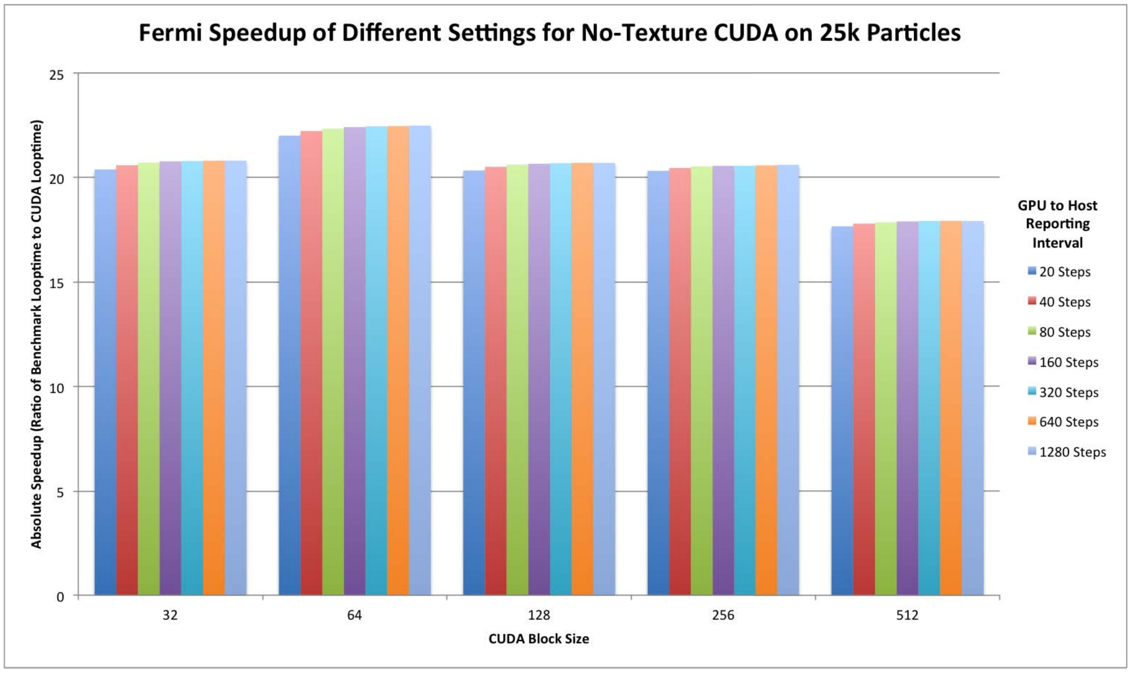
4.4.3 No-Texture CUDA version

To test whether texture binding was necessary, we also implemented a version without texture binding. These are the runtime results of the No-Texture CUDA version:

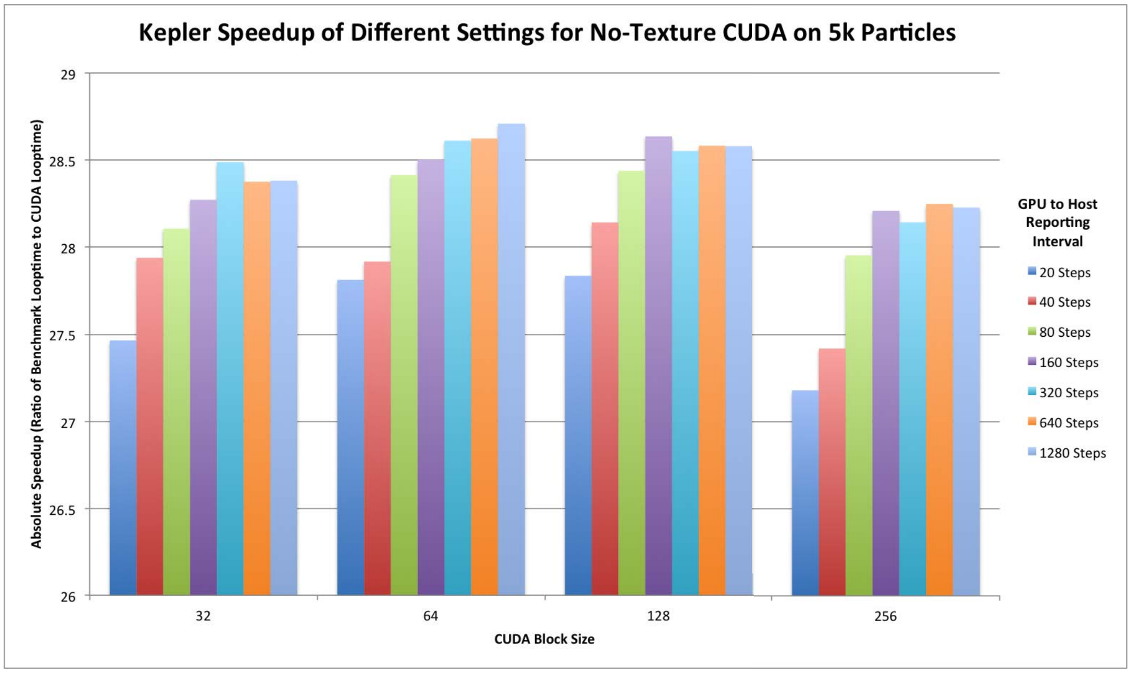
These are the results for 5000 particles on the Fermi without using texture memory:



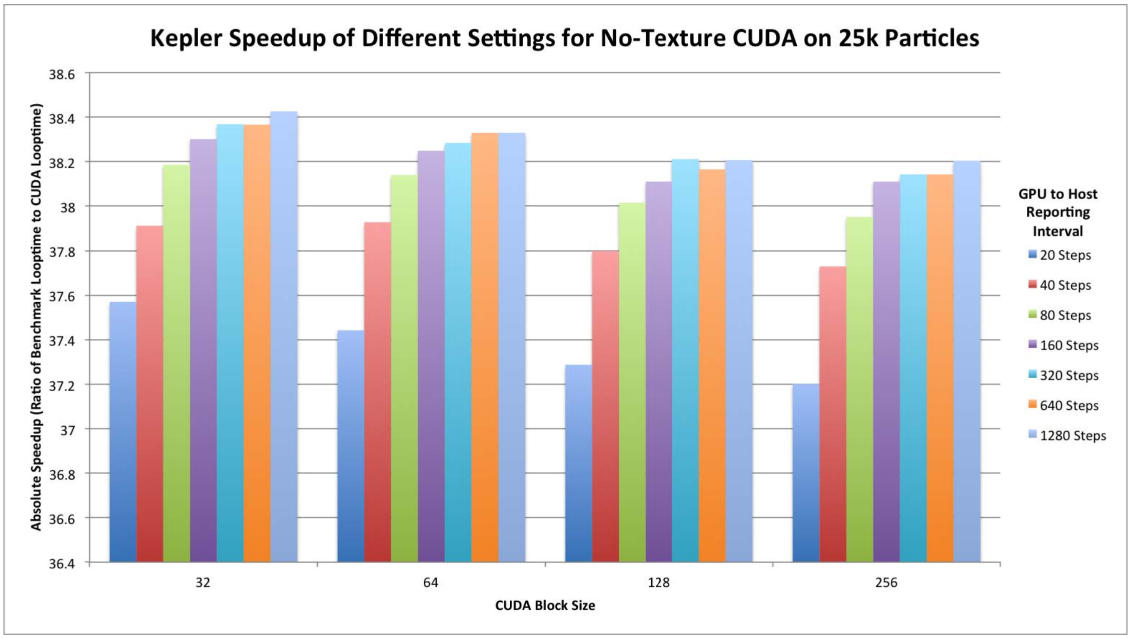
These are the results for 25000 particles on the Fermi (without texture):



These are the results for 5000 particles on the Kepler (without texture):



These are the results for 25000 particles on the Kepler (without texture):



These results indicate that the no-texture CUDA version was only three-quarters as fast as the fully optimized version, indicating that binding textures makes a significant improvement in the speed of the code.

4.4.4 Correctness

The distf.plt results of all three implementations matched those of the OpenACC version, where the "-nofma" compiler flag reduces the number of differing particles to 102 out of 5000. Diffusion.plt now produces correct (but slightly different) results.

4.4.5 Number of Registers Used

Using the NVIDIA Visual Profiler (nvvp), we could check what happens during a run of the fully optimized version versus a run of the simple version. According to Figure 9, the profiler shows that during the fully optimized version, all 255 allowed registers are used. This indicates that some registers are spilling into local memory. In contrast, according to Figure 10, the simple version only uses 164 registers.

While the fully optimized version is much faster than the simple version, possibly because of storing more compute-heavy variables into registers, more work can still be done in reducing registers to prevent this spilling. The fact that the performance is still fast indicates that it is probably only a small number of registers that are spilling, which means this might be easy to fix.

4.5 CUDA Compiler settings

The majority of calculations in ORBIT are double-precision floating-point operations. Both our GPUs, the Fermi and the Kepler, support double precision, since they have Compute Capabilities (cc) 2.0 and 3.5, respectively (Any Compute Capability of 1.3 or higher supports double precision). According to NVIDIA, cc 2.0 has significantly faster double-precision floating-point operation capabilities than of cc 2.5 or cc 3.0. [6] (See Figure 12 in the appendix for the table) Though our Kepler GPU has cc 3.5, which is even faster, our compiler was unable to compile with cc 3.5 settings. So far, in implementations involving the GPU, we compiled with the -Mcuda=cc20 flag, optimizing it for cc 2.0. We tested the cc30 flag on the Kepler and the cc13 flag on the Kepler and the Fermi, and the cc30 flag performed worse by about a factor of 2 and the cc13 flag performed worse by over 10%. (See Figure 11 in the Appendix for runtime data.) In addition, the outputs were identical to the outputs from the cc20 version. Therefore, as long as cc35 is not available as a compiler flag, it is important to compile with cc20 and not cc30.

5 Conclusion and Future Work

The OpenMP One-Region and the CUDA Fortran Fully-Optimized versions were combined into the final code, since the former is the fastest non-GPU version and the latter is the fastest GPU version. The final code would select the implementation based on whether the machine has a GPU.

Future work could attempt to use the newly released OpenACC 2.0 with more success, both in terms of performance and correct results. The OpenACC 2.0 documentation indicates that it supports subroutine calls, and there might be more flexibility allowed with OpenACC.

A more immediate and useful future project could be to reduce the register usage to remove spilling. Other necessities include the implementation of a wider variety of data transfers and data aggregation in between the kernels, so that settings other than “nplot = 2” can be supported. Another useful project would be to implement visualization code such that the GPU can also render particle information to a frame while computing the particle information.

6 Acknowledgements

This work was supported by the National Undergraduate Fellowship, funded by the Department of Energy, and made possible by the PPPL Science Education Department. We also thank Jianying Lang for her expertise.

References

[1] RB White and MS Chance. Hamiltonian guiding center drift orbit calculation for plasmas of arbitrary cross section. *Physics of Fluids*, 27:2455, 1984.

[2] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 3.1 edition, July 2011. (`http://www.openmp.org/mp-documents/OpenMP3.1.pdf`).

[3] OpenACC-Standard.org. *The OpenACC Application Programming Interface*, 1.0 edition, November 2011. (`http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf`).

[4] The Portland Group. *CUDA Fortran Programming Guide and Reference*, 13.6 edition, June 2013. (`http://www.pgroup.com/doc/pgicudaforug.pdf`).

[5] See for example. `http://en.wikipedia.org/wiki/POSIX_Threads` or `https://computing.llnl.gov/tutorials/pthreads/`.

[6] NVIDIA Corporation. *CUDA C Programming Guide*, pg-02829-001 v5.5 edition, July 2013. (`http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`).

7 Appendix

Loop Runtimes of Each Compiler Setting, nprt = 5000

Setting (Device)	Run 1 (hrs)	Run 2 (hrs)	Run 3 (hrs)
cc30 (Kepler)	1.301577E-3	1.302546E-3	1.302392E-3
cc13 (Kepler)	7.063188E-3	7.078032E-3	7.049191E-3
cc13 (Fermi)	1.645337E-2	1.600299E-2	1.604303E-2
Original (CPU)	2.372401E-1	2.372656E-1	2.374618E-1

Figure 11: Raw data of runtimes of the CUDA Fully-Optimized Version, with the best block and reporting intervals, for Compute Capabilities other than 2.0. Notice that if we divide the CPU runtimes by each of these data points, we find that the speedups are significantly lower than those of cc20 reported earlier.

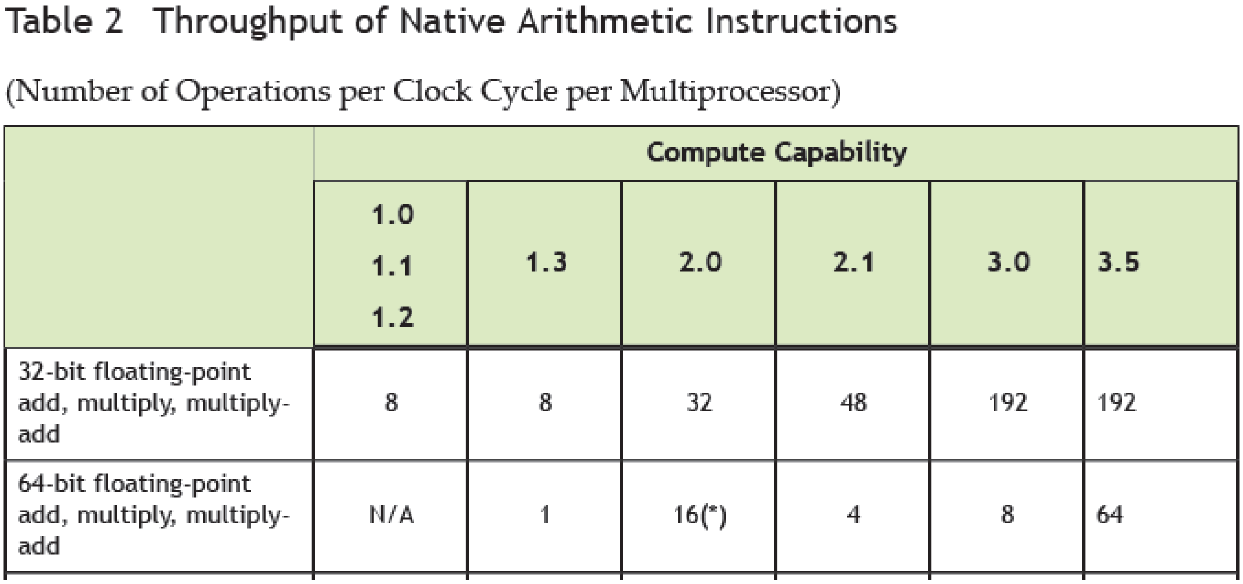


Table 2 Throughput of Native Arithmetic Instructions

(Number of Operations per Clock Cycle per Multiprocessor)

	Compute Capability					
	1.0 1.1 1.2	1.3	2.0	2.1	3.0	3.5
32-bit floating-point add, multiply, multiply-add	8	8	32	48	192	192
64-bit floating-point add, multiply, multiply-add	N/A	1	16(*)	4	8	64

*Throughput is lower for GeForce GPUs.

Figure 12: This is a table of the arithmetic instructions throughput of different NVIDIA GPU compute capabilities, taken from the CUDA C Programming Guide. [6]

The Princeton Plasma Physics Laboratory is operated
by Princeton University under contract
with the U.S. Department of Energy.

Information Services
Princeton Plasma Physics Laboratory
P.O. Box 451
Princeton, NJ 08543

Phone: 609-243-2245
Fax: 609-243-2751
e-mail: pppl_info@pppl.gov
Internet Address: http://www.pppl.gov