School of Physics and Astronomy

EPSRC Scholarship Summer Project

An Investigation Into Particle Tracking and Simulation Algorithms Using GPUs

James Henderson
September 2010

Abstract

Over recent years Graphical Processing Units (GPUs) have demonstrated their great ability in scientific calculations. They have the capability to carry out tasks in parallel, enabling huge speed-ups of calculations. This project investigates the potential speed-up using GPUs when calculating a particle's trajectory through a magnetic field. The results show that, when considering many particles, a speed-up of 32x can be achieved on the GPU versus a serial processor.

Declaration

I declare that this project and report is my own work.

Signature: Date:

Supervisor: Prof. Philip Clark 10 Weeks

Contents

1 Introduction **2**

2 Background **2**
2.1 The Runge-Kutta Method 2
2.1.1 Error Analysis 4
2.2 Geant4 4
2.3 Parallel Computing 5

3 Method **5**

4 Results & Discussion **7**
4.1 Accuracy 10

5 Conclusion **12**

6 Acknowledgements **12**

A Appendix **13**
A.1 CPU comparason 13
A.2 C++ code 14
A.3 CUDA code 19

1 Introduction

In recent years Graphical Processing Units (GPUs) have proven extremely useful in complex scientific calculations. They have been found to speed-up program times by huge amounts and improve what we can achieve in the same timescale. This project investigated potential speed-up of a section of the particle tracking toolkit Geant4 [1], used to simulate particle trajectories in medical, space and high-energy physics. The section of Geant4 code investigated was the Runge-Kutta algorithm: a numerical differential equation solver, which was ported (translated) into GPU code and any speed-up compared to a serial version of the code.

2 Background

2.1 The Runge-Kutta Method

The Runge-Kutta 4th-order (RK4) method is used to step a particle through a differential equation of motion. In this case the Lorentz equation was used to move a particle through a magnetic field.

$$\mathbf{F} = m\mathbf{a} \quad = \quad q \cdot (\mathbf{E} + \mathbf{v} \times \mathbf{B})$$

$$\mathbf{a} = \frac{d\mathbf{v}}{dt} \quad = \quad \frac{q}{m} \cdot (\mathbf{E} + \mathbf{v} \times \mathbf{B}) \tag{1}$$

Where $\mathbf{F}$ denotes the force on the particle, m - mass, q - charge, $\mathbf{E}$ - the electric field, $\mathbf{B}$ - the magnetic field and $\mathbf{v}$ - the velocity. It is worth noting this project found the trace of the particle's velocity, but it is simple to find position if we know both the initial position and velocity at each step.

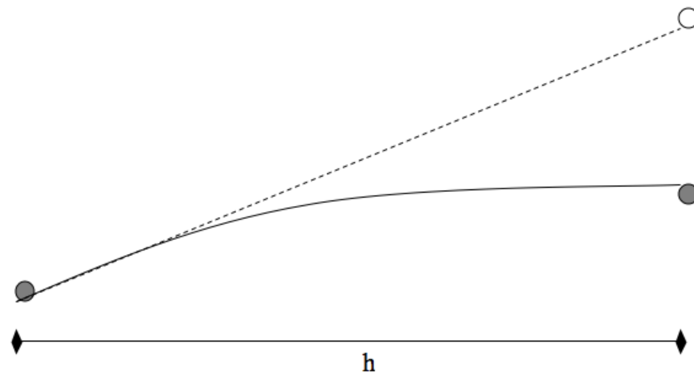

Figure 1: The simple Euler method of integration, the gradient is found at the starting point and the next point is found by the equation of a straight line. The step is denoted h and the solid line shows the true trajectory, the dashed line shows the flawed, calculated trajectory.

The most basic integration method is simple Euler integration [2], where the gradient ($\mathbf{a}$ in equation 1) is found at a point and the trajectory is modified to the next step by the

equation of a straight line. Figure 1 shows one step of the Euler method, it can be seen that the main problem is that the gradient can change over the distance of one step. To solve this problem we can simply reduce the step-size but this means we must do more calculations, which takes longer.

Other possible numerical differential equation solvers include [3]:

- Backwards Euler: the gradient used to progress a step is the gradient at the endpoint of the step.
- Trapezoidal Method: the gradient used is an average of the forwards and backwards Euler methods.
- Midpoint Method: the gradient used is an average of the forwards and backwards Euler methods as well as the gradient at a half step.

The RK4 method is accepted as giving the most accurate results for the least calculations. Figure 2 demonstrates how the RK4 method works.

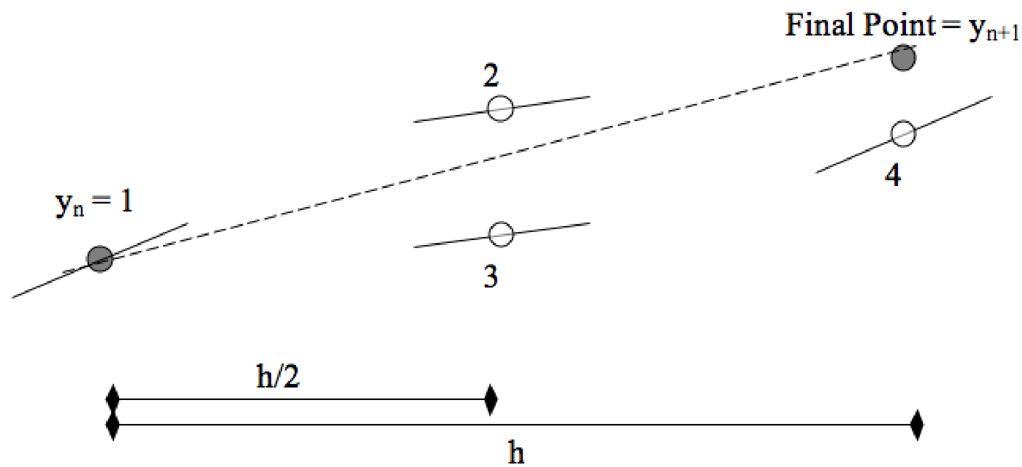

Figure 2: The Runge-Kutta 4th order method of integration. Gradients are found at 4 points and averaged over to take us to the next step. The full step is denoted h.

- The first gradient (m_1) is found at the initial point (point 1).
- Point 2 is found using simple Euler and a half step (h/2) from point 1 and gradient m_1.
- The second gradient (m_2) is now found at point 2.
- Point 3 is found using simple Euler and a half step (h/2) from point 1 and gradient m_2.
- m_3 is found at point 3.
- Point 4 is found using simple Euler and a full step (h) from point 1 and gradient m_3.

- m_4 is found at point 4.
- A weighted averaged of the gradients is taken and the final point is found by Euler from point 1, see equation 2, where the gradients are denoted as ms.

$$y_{n+1} = y_n + h \cdot (m_1 + 2m_2 + 2m_3 + m_4) \tag{2}$$

2.1.1 Error Analysis

The problem remains from the Euler method where the step size may be too large and miss some change in the gradient. To cover that problem, the RK4 method does error analysis that calculates the y_{n+1} value in one RK4 evaluation and then again in two evaluations at half the step size, see Figure 3. The two results for y_{n+1} are compared and if similar enough the step is accepted, if not the step-size is decreased and the step re-evaluated. Figure 3 shows the one big evaluation done in faded black and the two smaller evaluations done in red. The difference in the results of the evaluations is shown as d in the figure.

There are other possible, more sophisticated, error analysis methods that can be found in Numerical Recipes in C [3], but as the purpose of this project is to prove the worthiness of porting the RK4 algorithm to GPU, rather than to provide a highly sophisticated working RK4, it was decided to incorporate the half step analysis.

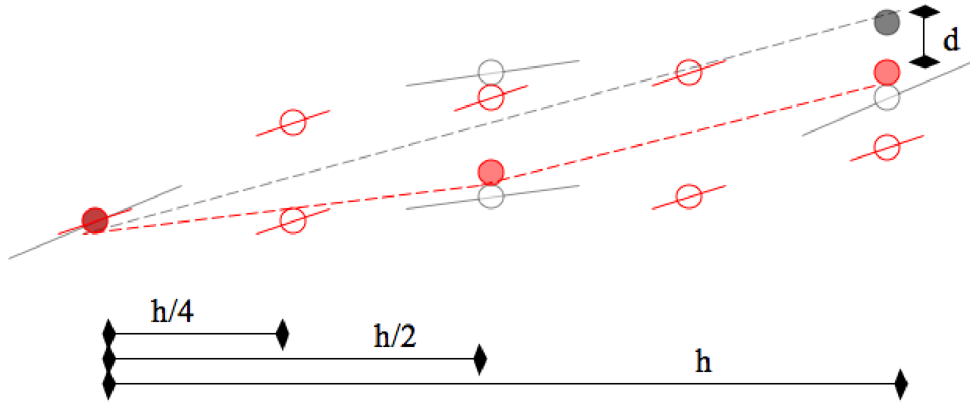

Figure 3: The error analysis of RK4. The full step is done as one full evaluation (faded black) and then again as two half evaluations (red). The difference in the y direction is shown as d.

2.2 Geant4

To make the study as relevant as possible, the RK4 algorithm in Geant4 was investigated. Geant4 has several types of RK4 functions, some designed to repeat the trajectory if a particle is stuck in a loop and others for constant field etc. The one most closely followed in this project was the G4ClassicalRK4 method, which inherits from the G4MagErrorStepper method [4]. The G4ClassicalRK4, takes a 'dumb step' forwards in

time for the particle, the G4MagErrorStepper class then evaluates this step to see if it is acceptable and within the error tolerance for the setup. This is the approach that was followed through the project, of step and evaluation. This means that whilst the code could probably be improved by an experienced programmer, the concept and hence results will be relevant, if not exact.

2.3 Parallel Computing

GPUs were designed to process data for many thousands of display pixels as quickly as possible and to take this burden away from the Central Processing Unit (CPU) [5]. They are an array of many thousands of processors all designed to carry out simple calculations at the same time (in parallel). This contrasts with the CPU who's job it is to carry out complex tasks in serial. Until recently CPU manufacturers have been able to increase clock speeds on CPUs and hence instructions are executed faster and so programs run faster. CPU development has decelerated due to limits with clock speed: overheating and data transport speed, so program developers are looking towards carrying out tasks in parallel to speed them up. This is more complex than simply increasing clock speed as the programs must be re-written to work on the GPUs.

At the time of writing, GPUs have the potential to perform around 10 times the number of floating point calculations per second, than a CPU. This give us the ability for huge improvement in computation time if the code is ported correctly.

3 Method

The RK4 algorithm is an inherently serial process; the calculation of a step requires us to know where we are starting from so we must have done the preceding step first. Therefore, it was not possible to simply calculate all the steps independently on the GPU and fix them all together later.

It was first decided to port the gradient calculations to GPU and try to calculate gradients (accelerations in this case) for all possible velocities. Due to only having a finite number of processors on the GPU there clearly had to be some granularity in the steps between the velocities used. Then once all the accelerations had been found on the GPU, they were transferred back to the CPU, in a large matrix, and the RK4 calculations were carried out where the CPU only had to find which gradient it required and look it up rather than calculate it by itself. During a meeting with John Apostolakis, a leading author in the Geant4 project, he suggested that the gradient calculations are fairly simple and would take a tiny amount of time to carry out on the CPU. Therefore he thought this strategy would provide us with no speed-up at all.

Therefore, it was decided that the best way to speed-up the trajectory tracking process was to port the error analysis section to GPU as well as doing each component (x, y and z) in parallel.

It was decided that the GPU code should evaluate a range of step-sizes at once so that if the largest step size was within an acceptable error of the smallest (and hence most accurate) step size, the trajectory could move on a greater amount and hence a smaller number of evaluations were required. To this end, a parameter called 'spread' was added

to the code, which if it were small, a small range of steps would be evaluated at once. For example; is spread = 1, step sizes of 2x, 1x, and 1/2x the step-size for the previous step were found. If spread = 3; 8x, 4x, 2x...1/4x, 1/8x the previous step-size were evaluated.

During testing of this strategy it became clear that spread must equal 1 to obtain maximum speed-up as otherwise the larger steps would be waiting for some time until the smaller steps completed. For example, if spread = 3, then the 8x step would be completed in one evaluation. The 4x is two and the 2x in four etc. The 1/8 step would require sixty-four evaluations until it reached the same point as the 8x step had in one evaluation. This clearly wasted time, so from this point on, spread was set to 1.

To port the error analysis, the one full evaluation was carried out in parallel with one of the half evaluations. Then the other half evaluation was done, meaning that where the serial code took the time of 3 evaluations, the GPU code took 2. Evaluating the components in parallel meant the GPU was 3 times faster in theory, than the CPU. Programs were written in CUDA - a GPU language designed by nVidia [6], and C++ - to run in serial and compare with the CUDA run times. Copies of both codes can be found in the appendix at A.2 and A.3.

The test case that was chosen to do the timed runs on was a charged particle in a magnetic field with an initial velocity in the x direction only. The magnetic field was confined to the y direction so that, according to the Lorentz equation (1) the particle's velocity should oscillate between the x and z directions in a sinusoidal fashion. This meant that the number of peaks in the sinusoidal curve would give a representation of the amount of computation that had to be carried out. For example; if there was a weak magnetic field and the trajectory had 5 peaks, the processor would have to do much less work, as it could take bigger steps, than if the magnetic field strength was increased and there were 500 peaks. The particles were tracked as the x-axis value (time) ran from 0 to 100. As it was decided to run over multiple particles, the input for each particle should be different to avoid the processors doing the exact same calculations multiple times. To this end, an input file was created where the first number determined the number of particles to be run and the remaining numbers described the respective particles initial velocities in the x direction.

The tests were carried out on 4 different processors; 2 GPUs and 2 CPUs. The GPUs were the nVidia Tesla C1060 [7], a specialist GPU card designed for scientific calculations and the nVidia GeForce 9400M [8], a standard GPU designed for laptop graphics. The CPUs were the 2 GHz Core 2 Duo Intel processor and the AMD Athlon Dual Core 5200+. The Intel processor proved to be much faster than the AMD, hence as we are comparing the GPU speeds with CPU speeds, it is fair to compare to the fastest CPU available to us. Therefore, the AMD has been omitted from the plots in the results section but an example plot can be found in the appexndix A.1.

4 Results & Discussion

As the GPU is physically located away from the CPU, data must be transferred between the two devices. This meant that the data transfer overheads made the CPU anywhere up to 70x faster, depending on the particle setup, than the GPU when tracking a single particle. It was however clear that the GPU was significantly faster at calculating the trajectory when it did have access to the data. This suggested that for multiple particles the GPU would be faster which was indeed the case.

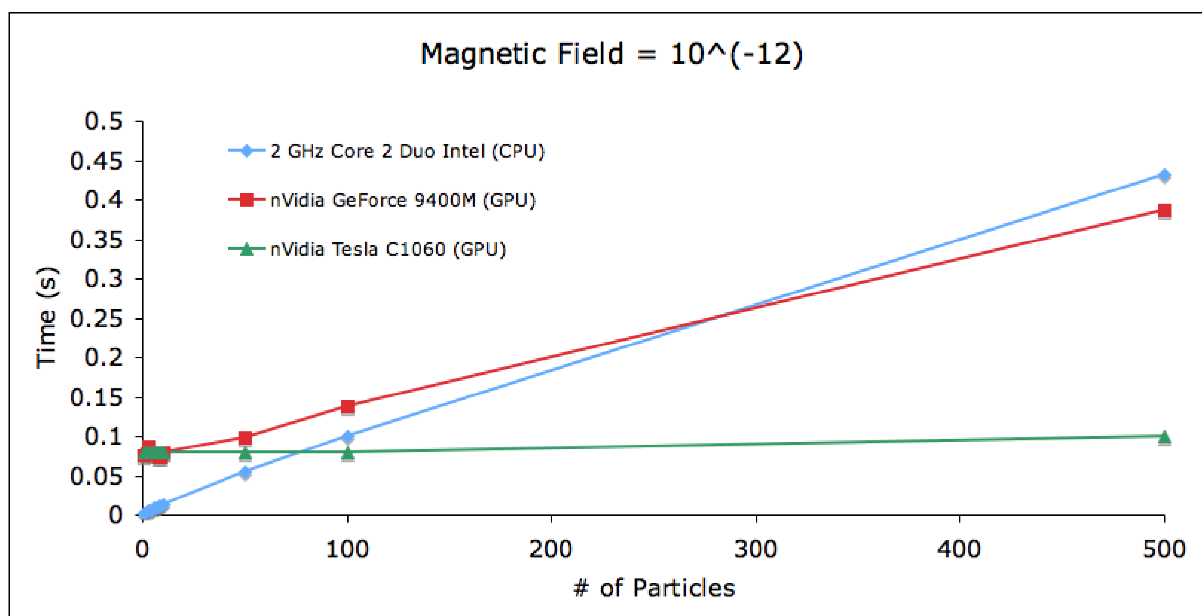

Figure 4: The timing results from magnetic field = 1×10^{-12}. The CPU time increases linearly whereas the GPUs become relatively faster with the number of particles.

In Figure 4 it can be seen that the CPU time increases linearly from zero as it has no transfer overhead, whereas the GPU times have a minimum value due to this transfer. This data is best displayed at particles/second as then we can see the vast improvements the GPUs make as the number of particles is increased.

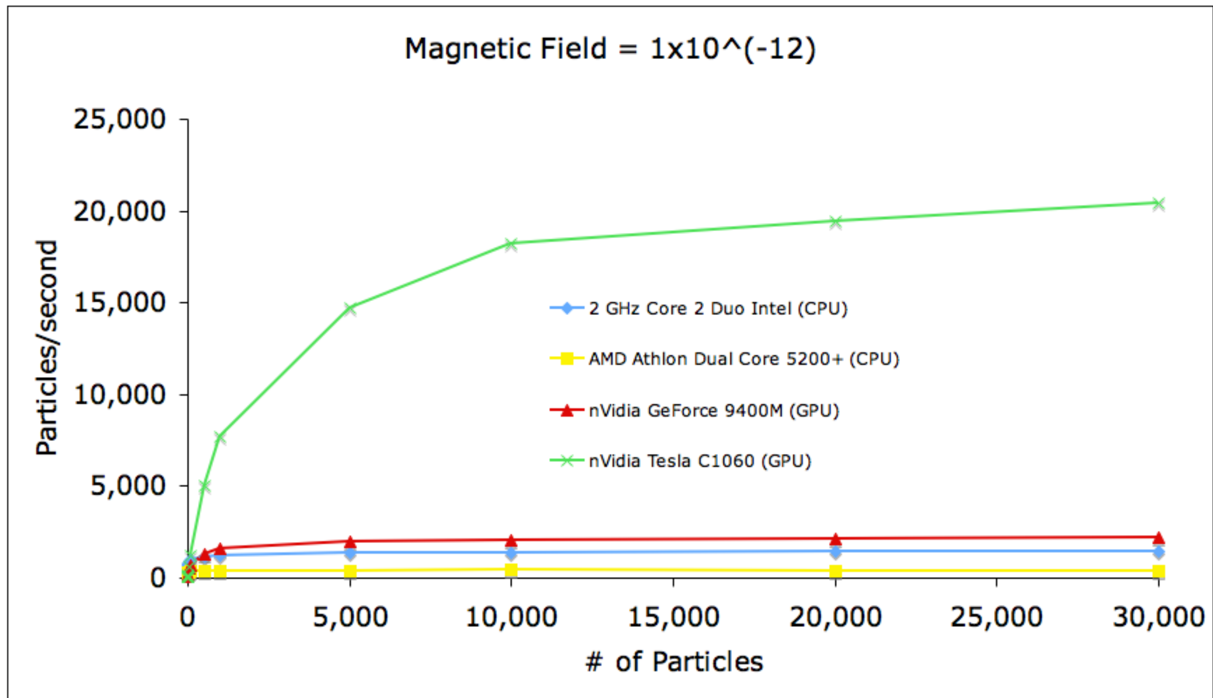

Figure 5: The timing results from magnetic field = 1 $\times 10^{-12}$. Displayed as number of particles versus particles processed per second.

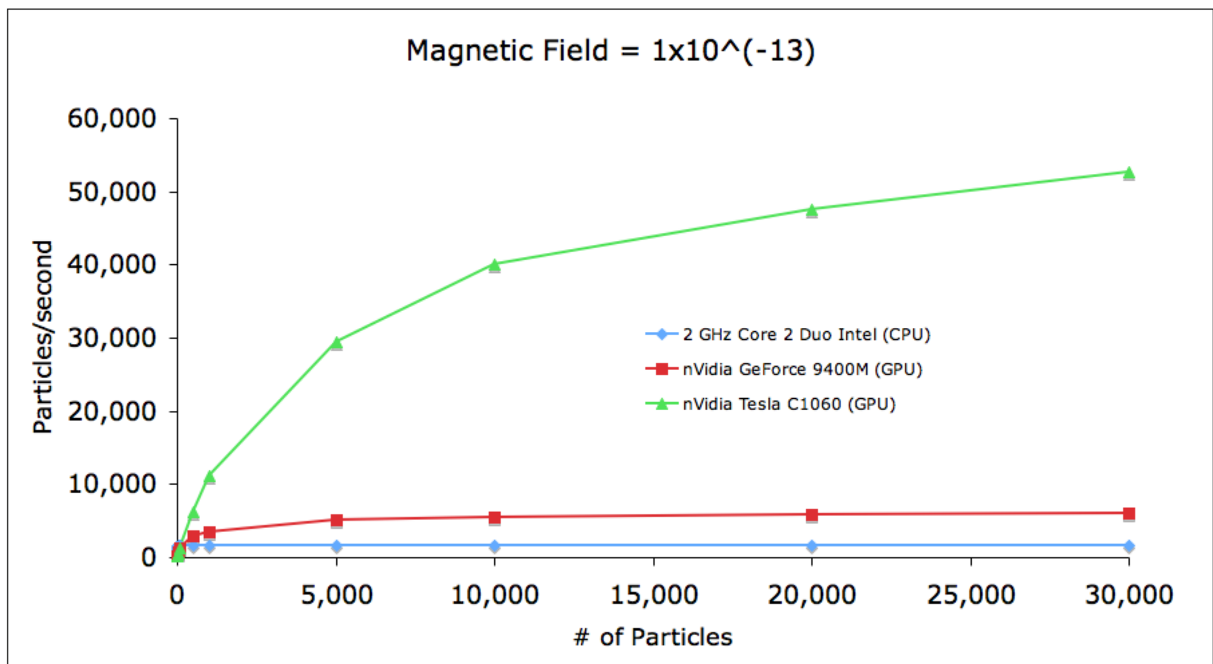

Figure 6: The timing results from magnetic field = 1 $\times 10^{-13}$. Displayed as number of particles versus particles processed per second.

The values for the magnetic field were chosen as they gave a reasonable number of oscillations of the velocities without taking too much time to gather data. The nVidia GeForce 9400M appears to increase linearly, as shown in Figure 4, this is probably as it

is a very small GPU, so has a limited number of processors that will be saturated early on.

Figures 5 and 6 display the number of particles processed per second by the various processors. These plots show how the nVidia Tesla can compute the trajectories of many more particles per second than the CPUs. The smaller nVidia GeForce also manages to compute more particle trajectories per second than the CPUs despite it being a widely available GPU designed for graphics processes, not calculations. Hence even small, non-specialised GPUs can give us speed-ups over CPUs.

Due to finite numbers of processors on the GPUs, there was a maximum number of particles that they could cope with. This was found to differ on each GPU as would be expected. The maximum number of particles investigated was 50,000, at which point the nVidia Tesla showed a speed-up of 32x over the Intel CPU, under certain particle conditions.

As we increase the magnetic field strength, and hence the number of peaks that the particle's velocity oscillates through, all of the compute times increase. If we consider the magnetic field strength of 1×10^{-8} the particles are oscillating 10,000 times more often than for a field os 1×10^{-12} so the compute times reflect this. Even for these very strong fields, the GPU times out-shine the CPU times, see Figure 7

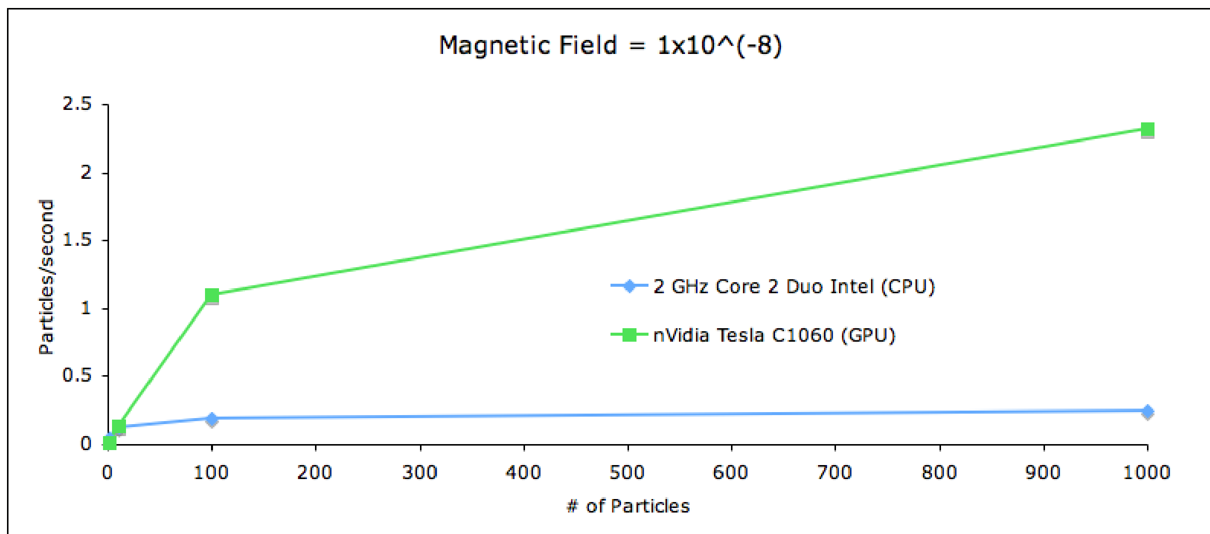

Figure 7: The timing results from magnetic field = 1×10^{-8}. Only the nVidia Tesla and the Intel CPU have been plotted as the other processors took much too long when computing these trajectories.

4.1 Accuracy

Whilst the timing results obtained are important, it is also necessary to look at how accurate the output plots are. For the test case described in section 3, the correct output plot for initial velocity = (1, 0, 0) and the magnetic field strength in the y direction = 1 $\times 10^{-12}$ should look as in Figure 8. The data displayed in Figure 8 was computed using the parallel CUDA code, but for the same conditions the serial C++ code produces the exact same results.

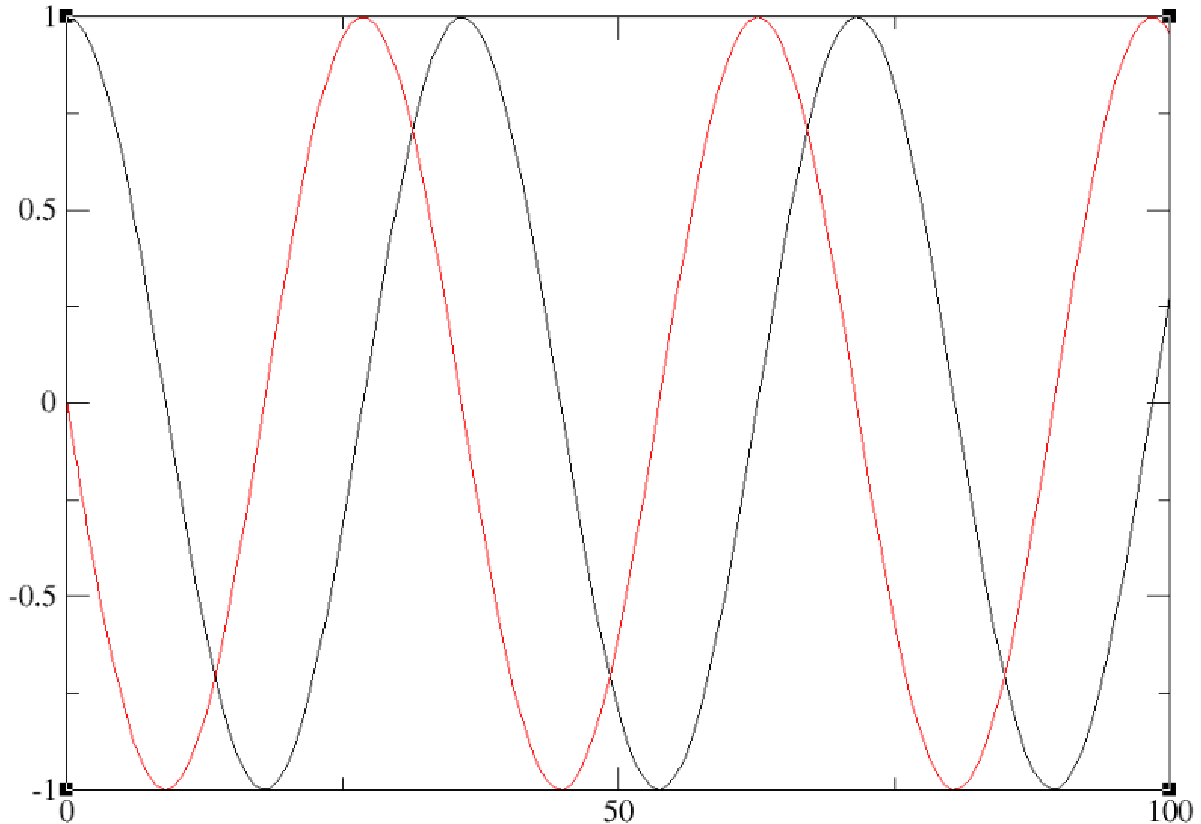

Figure 8: The correct output plot for initial velocity = (1, 0, 0). The x velocity is shown in black and the z velocity is in red. Y velocity remains at 0 throughout. Time runs along the x-axis.

Despite the C++ code producing the same results under the above described conditions, when it was calculating multiple particles trajectories the results be less pleasing. When the results of calculating 30,000 particle trajectories under magnetic field = 1 $\times 10^{-12}$ are compared between C++ and CUDA, the C++ is clearly not very accurate, see Figure 9. These are close to the conditions under which we obtained maximum speed-up, only now with the magnetic field 10 times stronger. Under these exact conditions we obtain 13x speed-up.

Figure 9 shows both the C++ and CUDA output plots. The CUDA plot has a consistent period over the whole graph whereas the C++'s period is changing. Whilst this clearly points to a programming error in the C++ coding, where the error analysis has a bug, this leads us to the conclusion that the CUDA code produces much more

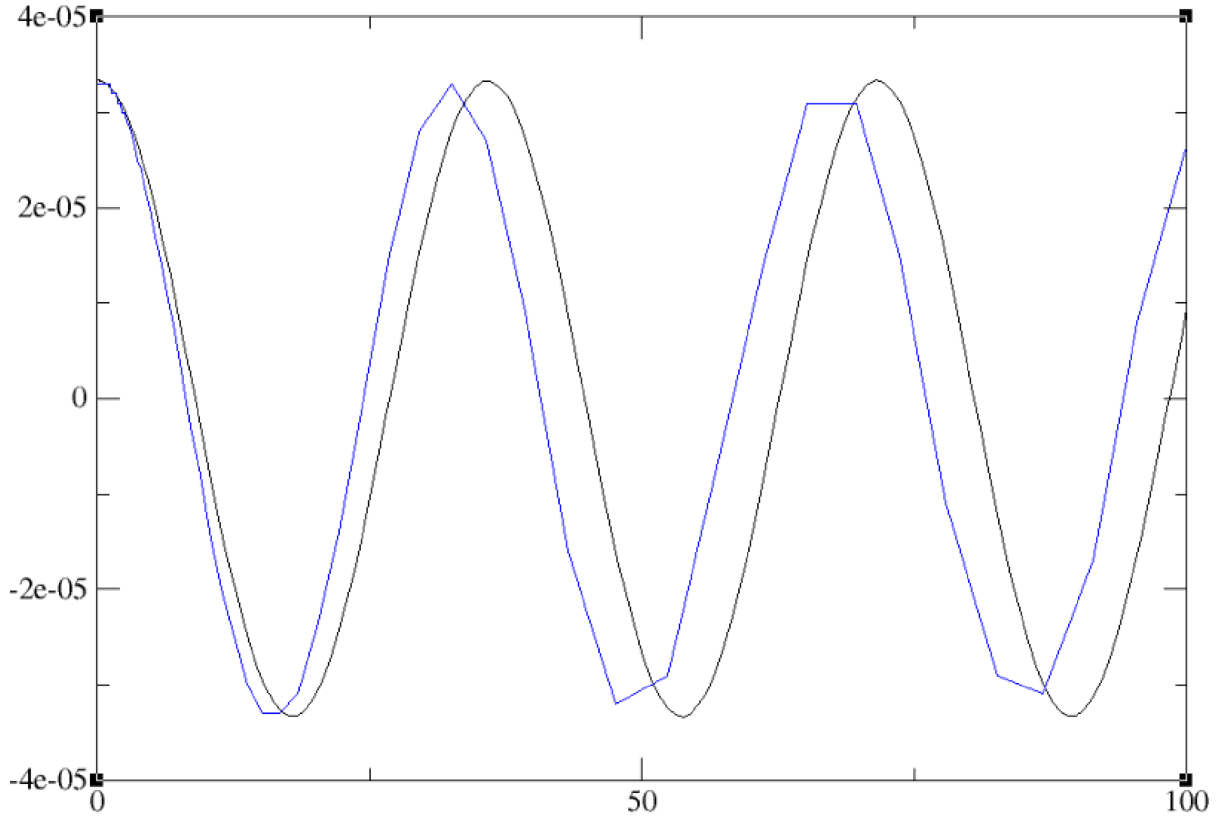

Figure 9: The output plots obtained from the final particle tracked during a 30,000 particle run. The smooth black plot shows the CUDA output and the jagged blue plot is the C++ output. Both plots show the x velocity, under initial velocity = (1/30,000, 0 ,0) and magnetic field in the y direction = 1 $\times 10^{-12}$.

accurate plots, 13 times faster than the C++ code.

Also worth discussing is the fall off of the sinusoidal traces over time. As our error analysis works by checking the next point is within a certain tolerance of our most accurate result, small errors creep into the plots and are exaggerated over time. This results in the peaks and troughs of the sine waves falling towards zero over time.

Note both Figures 10 and 11 have the same axis so it is clear to see the C++ output falls off more over the same time. This again proves that the CUDA code has produced a more accurate plot in a shorter space of time. A more talented programmer could improve the C++ code to be more accurate but it would be sensible to assume that this improved code would take the same time or longer as the current C++ code, so the speed-ups obtained in this paper would hold or be improved.

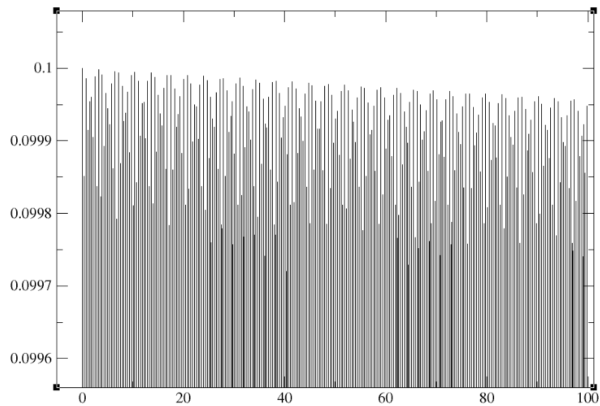

Figure 10: CUDA output's fall off over time.

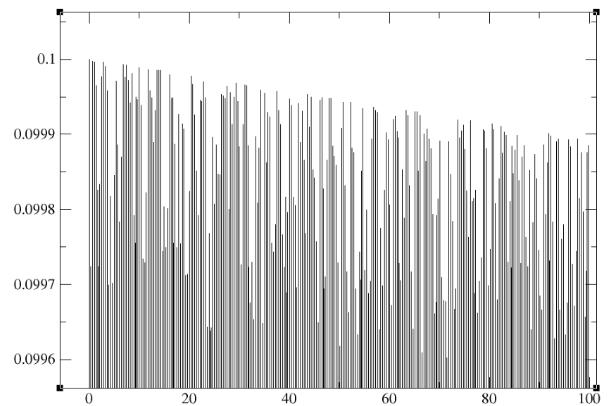

Figure 11: C++ output's fall off over time.

5 Conclusion

The GPU has again proven its worth in detailed scientific tasks and has provided a speed-up of 32 times under certain conditions, over the serial CPU. This could be improved by an experienced programmer as the author has very limited experience in the field of GPU programming. Yet even so, this report has shown that with a little research and time even an inexperienced programmer can achieve 32x speed-up of a very relevant piece of code. Whilst the serial code produced for this project had some flaws that caused it to produce in-accurate results for high particle numbers, one can reasonably assume that if these issues were to be resolved, the serial code would take longer. Hence, it is sensible to say that higher speed-ups could be obtained from a more detailed investigation of this subject. This investigation has shown that parallelising what is an inherently serial algorithm can indeed result in significant speed-ups and has proven how vital GPU program development is for scientific computations.

6 Acknowledgements

Thanks to Professor Philip Clark with orientation of the theory and a great deal of assistance through out the course of the project. Thanks also to Andy Washbrook for help with many struggles with the computers and to John Apostolakis for guidance on the project direction.

A Appendix

A.1 CPU comparason

After [3] collecting some data it was clear that the Intel CPU decisively out-performed the AMD CPU (see section 3). Therefore, as the project was geared to compare times from GPUs to CPUs, it was reasonable to compare the GPU speed-up times to the fastest CPU times we had. It was decided to therefore omit the AMD data from all the plots in the results section. An example of a plot that includes the AMD data is shown in Figure 12, showing the AMD processor to be much slower than the other 3.

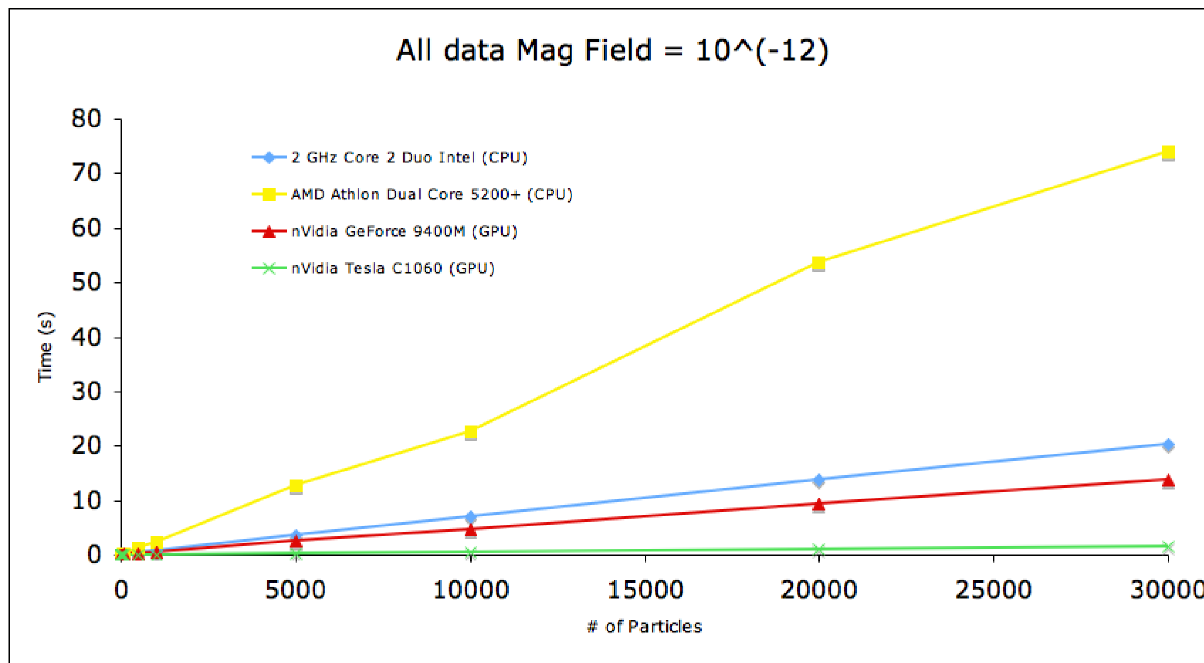

Figure 12: An example plot showing all 4 processors compared. The AMD is clearly the slowest.

A.2 C++ code

```
#include <stdio.h>
#include <math.h>
#include <ctime>
#include <stdlib.h>



#define FIRST_STEP 0.0001
#define TOLERANCE 1*pow(10.0,-7)

void RK4( const float B_fieldD[], float tempYPointsD[], float
    first_step[], const float velocityD[])
{
  float grad_vel0[3];
  float grad_vel1[3];
  float grad_vel2[3];
  float grad_vel3[3];
  float temp_vel1[3];
  float temp_vel2[3];
  float temp_vel3[3];
  int i, j;

  const float charge = -1.6*pow(10.0,-19);
  const float mass = 9.11*pow(10.0,(-31));
  const float step = first_step[0];
  const float halfstep = step/2;
  ;
  for ( i = 0 ; i < 3; i++)
    tempYPointsD[i] = velocityD[i];

  //-------Step 1---------
  grad_vel0[0] = charge/mass * (velocityD[1] * B_fieldD[2]  -
     velocityD[2] * B_fieldD[1] );  //get gradient 0
  grad_vel0[1] = charge/mass * (velocityD[2] * B_fieldD[0]  -
     velocityD[0] * B_fieldD[2] );
  grad_vel0[2] = charge/mass * (velocityD[0] * B_fieldD[1]  -
     velocityD[1] * B_fieldD[0] );

  for ( j = 0 ; j < 3 ; j++)
    {
      temp_vel1[j] = velocityD[j] + halfstep * grad_vel0[j];
         /* move temp velocityD to velocityD + s/2 * gradient0
         */
    }
```

```
  //--------Step 2----------
  grad_vel1[0] = charge/mass * (temp_vel1[1] * B_fieldD[2] -
     temp_vel1[2] * B_fieldD[1]);  //get gradient 1
  grad_vel1[1] = charge/mass * (temp_vel1[2] * B_fieldD[0] -
     temp_vel1[0] * B_fieldD[2]);
  grad_vel1[2] = charge/mass * (temp_vel1[0] * B_fieldD[1] -
     temp_vel1[1] * B_fieldD[0]);

  for ( j = 0 ; j < 3 ; j++)
    {
      /* move temp velocityD to velocityD + s/2 * gradient1 */
      temp_vel2[j] = velocityD[j] + halfstep * grad_vel1[j];
    }
  //--------Step 3----------
  grad_vel2[0] = charge/mass * (temp_vel2[1] * B_fieldD[2] -
     temp_vel2[2] * B_fieldD[1]);  //get gradient 2
  grad_vel2[1] = charge/mass * (temp_vel2[2] * B_fieldD[0] -
     temp_vel2[0] * B_fieldD[2]);
  grad_vel2[2] = charge/mass * (temp_vel2[0] * B_fieldD[1] -
     temp_vel2[1] * B_fieldD[0]);

  for ( j = 0 ; j < 3 ; j++)
    {
      /* move temp velocityD to velocityD + s/2 * gradient2 */
      temp_vel3[j] = velocityD[j] + step * grad_vel2[j];
    }
  //--------Step 4------------
  grad_vel3[0] = charge/mass * (temp_vel3[1] * B_fieldD[2] -
     temp_vel3[2] * B_fieldD[1]);  //get gradient 3
  grad_vel3[1] = charge/mass * (temp_vel3[2] * B_fieldD[0] -
     temp_vel3[0] * B_fieldD[2]);
  grad_vel3[2] = charge/mass * (temp_vel3[0] * B_fieldD[1] -
     temp_vel3[1] * B_fieldD[0]);
  //-------Final point---------
  for ( j = 0 ; j < 3 ; j++)
    {
      tempYPointsD[ j ] = tempYPointsD[ j ] + step/6.0 * (
         grad_vel0[j] + grad_vel3[j] + 2 * (grad_vel1[j] +
         grad_vel2[j]) );
    }

}


void Stepper(const float B_field[], float timestep[], float
   velocity[], float printstep[]) //Function to setup and call
```

```
   kernal
{

  int i;

  float wholestep_y[3];
  float halfstep_y[3];

  float timestepD[1];



 restart:

  float halfstepD[1];
  float velocityD[3];
  float wholestep_yD[3];
  float halfstep_y1D[3];
  float halfstep_y2D[3];
  float B_fieldD[3];

  timestepD[0] = timestep[0];
  halfstepD[0] = timestepD[0]/2.0;

  for(i=0;i<3;i++){
    velocityD[i]=velocity[i];
    B_fieldD[i]=B_field[i];
  }

  RK4(B_fieldD, wholestep_yD, timestepD, velocityD);
  RK4(B_fieldD, halfstep_y1D, halfstepD, velocityD);
  RK4(B_fieldD, halfstep_y2D, halfstepD, halfstep_y1D);

  for(i=0;i<3;i++){
    wholestep_y[i]=wholestep_yD[i];
    halfstep_y[i]=halfstep_y2D[i];
  }

  if ( ( fabs(wholestep_y[0] - halfstep_y[0] ) < TOLERANCE ) &&
     ( fabs(wholestep_y[1] - halfstep_y[1]) < TOLERANCE ) && (
     fabs(wholestep_y[2] - halfstep_y[2]) < TOLERANCE )  )
    {
      for ( i = 0 ; i < 3 ; i++)
        velocity[i]= wholestep_y[i];
      //       printf(" Increasing the timestep.\n    ");
      printstep[0]=timestep[0];
```

```
        timestep[0] *= 1.1;
      }
    else{
      timestep[0] *= 0.8;
      // printf("Decreasing the timestep.\n");
      goto restart;
    }

  }


  int main()
  {

  float velocity[3];
  int looper;
  int upper;
   clock_t begin = clock();
     scanf("%i", &upper);

    for(looper=0; looper<upper; looper++)
      {

        printf("Tracking particle %i.\n", looper);
        FILE *velocityfile;
        velocityfile = fopen("++Velocity.dat","w");

        float B_field[3];

        B_field[1]=1.0*pow(10.0,-10);


        //-------------------Start of velocity tracking-----------
        //   printf("Starting RK4 calculations.\n");
        scanf("%g", &velocity[0]);
        velocity[1]=0;
        velocity[2]=0;
        float time = 0;
        float timestep[1];
        float printstep[1];

        timestep[0]=FIRST_STEP;
        printstep[0]=timestep[0];

        for (time = 0 ; time < 100 ; time += printstep[0]  )
          {
```

```
        //printf("************************        TIME = %g \n",
            time);
        Stepper(B_field, timestep, velocity, printstep);
        fprintf(velocityfile, " %f %f \n", time, velocity[0]);
      }

      fclose ( velocityfile );
    }
  printf("Time elapsed for C++ calculations: %g\n", ( ( std::
      clock() - begin ) / (double)CLOCKS_PER_SEC ) );

  return 0;

}
```

A.3 CUDA code

```
#include <stdio.h>
#include <math.h>
#include <ctime>

#define SPREAD  1
#define RANGE (int)pow(2.0,SPREAD)
#define TOLERANCE 1*pow(10.0,-2)

__global__ void Kernal( const float B_fieldD[], float
    tempYPointsD[],const float first_step[], const float
    velocityD[])
{

  int i;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int bx = blockIdx.x;

  //const int numParticles = numParticlesD[0];

  __shared__ float grad_vel0[(2*SPREAD + 1)][3];
                              //Declare the temporary gradients
     and velocities
  __shared__ float grad_vel1[(2*SPREAD + 1)][3];
  __shared__ float grad_vel2[(2*SPREAD + 1)][3];
  __shared__ float grad_vel3[(2*SPREAD + 1)][3];

  __shared__ float temp_vel1[(2*SPREAD + 1)][3];
  __shared__ float temp_vel2[(2*SPREAD + 1)][3];
  __shared__ float temp_vel3[(2*SPREAD + 1)][3];

  const float charge = -1.6*pow(10.0,-19);                    //
     Set the particles attributes, default for an electron
  const float mass = 9.11*pow(10.0,-31);

  const float step = pow(2.0, (SPREAD-tx) ) * first_step[bx];
  const float halfstep = step/2.0;
  const int evaluations = pow(2.0, tx) ;

  tempYPointsD[bx *3*(2*SPREAD+1) + (tx * 3) + ty] = velocityD[
     bx * 3 + ty];

  for ( i = 0 ; i < evaluations ; i++ )
    {
```

```
      //-------Step 1----------
      grad_vel0[tx][ty] = charge/mass * (velocityD[bx*3+(ty+1)
         %3] * B_fieldD[(ty+2)%3]  - velocityD[bx*3+(ty+2)%3] *
         B_fieldD[(ty+1)%3] );  //get gradient 0

      temp_vel1[tx][ty] =velocityD[bx*3+ty] + halfstep *
         grad_vel0[tx][ty];     /* move temp velocityD to
         velocityD + s/2 * gradient0 */
  __syncthreads();
      //--------Step 2-----------
      grad_vel1[tx][ty] = charge/mass * (temp_vel1[tx][(ty+1)%3]
          * B_fieldD[(ty+2)%3] - temp_vel1[tx][(ty+2)%3] *
         B_fieldD[(ty+1)%3]);  //get gradient 1

      temp_vel2[tx][ty] = velocityD[bx*3+ty] + halfstep *
         grad_vel1[tx][ty];
  __syncthreads();
      //--------Step 3-----------
      grad_vel2[tx][ty] = charge/mass * (temp_vel2[tx][(ty+1)%3]
          * B_fieldD[(ty+2)%3] - temp_vel2[tx][(ty+2)%3] *
         B_fieldD[(ty+1)%3]);  //get gradient 2

      temp_vel3[tx][ty] = velocityD[bx*3+ty] + step * grad_vel2[
         tx][ty];
  __syncthreads();
      //--------Step 4-------------
      grad_vel3[tx][ty] = charge/mass * (temp_vel3[tx][(ty+1)%3]
          * B_fieldD[(ty+2)%3] - temp_vel3[tx][(ty+2)%3] *
         B_fieldD[(ty+1)%3]);  //get gradient 3
  __syncthreads();
      //-------Final point---------
      tempYPointsD[ bx *3*(2*SPREAD+1)+ ( tx * 3 + ty) ] =
         tempYPointsD[bx*3*(2*SPREAD+1)+(tx * 3 + ty)] + step
         /6.0 * ( grad_vel0[tx][ty] + grad_vel3[tx][ty] + 2 * (
         grad_vel1[tx][ty] + grad_vel2[tx][ty]) );
    }
   __syncthreads();
}
```

```
void Stepper(float time[], float tempYPoints[], const float
    B_field[], float timestep[], float velocity[], float
    printstep[], float max, float *B_fieldD[], float *velocityD
    [], float *tempYPointsD[], float *timestepD[], dim3 dimBlock,
     dim3 dimGrid, int numParticles) //Function to setup and call
     kernal
{
  int step,component, donesum = 0, particle;

     //Set up and copy from the host, variables to be used on
     the device
  int done[numParticles];

  for ( particle = 0 ; particle < numParticles ; particle++ )
    done[particle] = 0;

  while(donesum<numParticles){
    donesum=0;

    cudaMemcpy(*timestepD, timestep, numParticles * sizeof(float
       ), cudaMemcpyHostToDevice);

    Kernal<<<dimGrid, dimBlock>>>( *B_fieldD, *tempYPointsD, *
       timestepD, *velocityD);

    cudaMemcpy(tempYPoints, *tempYPointsD, (numParticles * 3 *
       (2*SPREAD + 1) * sizeof(float)), cudaMemcpyDeviceToHost);
           //Copy back to the host the new velocities

    //-------------Start error analysis----------------


    for ( particle = 0 ; particle < numParticles ; particle++ )
      {

        if ( done[particle] == 0 ) {

          for ( step = 0 ; step < 2*SPREAD ; step++ )
            {

              if (     ( fabs(tempYPoints[particle*3*(2*SPREAD+1)
                 +step*3] - tempYPoints[particle*3*(2*SPREAD+1)
                 +(2*SPREAD)*3 ]) < TOLERANCE )  &&   ( fabs(
                 tempYPoints[particle*3*(2*SPREAD+1) + step*3 +
```

```
                1] - tempYPoints[particle*3*(2*SPREAD+1)+(2*
                SPREAD)*3 + 1]) < TOLERANCE )   &&   ( fabs(
                tempYPoints[particle*3*(2*SPREAD+1)+step*3+2] -
                 tempYPoints[particle*3*(2*SPREAD+1)+(2*SPREAD)
                *3+2]) < TOLERANCE )   )

                //Select the new velocity that is within the
                error tolerance of the most accurate new
                velocity.
              {
                for ( component = 0 ; component < 3 ;
                   component++)
                                                              //
                   Assign the selected new velocity to the
                   velocity variable
                  velocity[particle*3+component]= tempYPoints[
                     particle*3*(2*SPREAD+1)+step*3+component
                     ];
                printstep[particle] = timestep[particle]*pow
                   (2.0, (SPREAD-step) );                     //
                   Select the timestep used so that it can be
                   printed in main
                time[particle] += printstep[particle];

                if (timestep[particle]*1.1 < max)

                   //Limit max timestep ——— By experiment
                   have deduced that this value of 2*(pow...
                   gives the lowest calculation times
                  timestep[particle] *= 1.1;
                done[particle]=1;
                step = 2*SPREAD;        //Stop the 'step' loop
                   as we have found a suitable step
              }
          }
      }
      donesum += done[particle];
      if(done[particle]==0){
        timestep[particle] /= SPREAD*SPREAD*2   ;
        // printf("Decreasing the timestep of particle %i.\n",
            particle);
      }
    }
  }
}
```

```
int main(void)
{
  //----------------Declare Host variables----------------
   int l, numParticles, particle;

   scanf("%i", &numParticles);          //  -----------------First
      number in input file should be the number of particles
      -------------------

  FILE *velocityfile;
  velocityfile = fopen("VelocityCUDA.dat","w");

  float velocity[numParticles*3];
  float B_field[3];
   float tempYPoints[numParticles*3*(2*SPREAD + 1)];

   for(l=0 ; l < numParticles * 3 * (2 * SPREAD +1 ) ; l++)
     tempYPoints[l] = 0.0;

  B_field[0]=0.;                                                   //
     Set up the magnetic field
  B_field[1]=1.0*pow(10.0,-13);
  B_field[2]=0.;

  int alive[numParticles];
  int alivesum = 0;

  float time[numParticles];
  float timestep[numParticles];
  float printstep[numParticles];

  for(l=0 ; l < numParticles ; l++){
```

```
  scanf("%g", &velocity[l*3]);              //-------------The
     2nd number and beyond should be the initial x velocities
     of the particles----------------
  timestep[l] = (pow(10.0, -15.0)/B_field[1]);
  printstep[l]=timestep[l];
  velocity[l*3+1] = 0;
  velocity[l*3+2] = 0;
  alive[l]=1;
  time[l] = 0;
  alivesum += alive[l];
}

float max = 20 * ( 1.0/(pow(2.0, SPREAD-2))) *(pow(10.0,
   -14.0)/(B_field[1]));

clock_t begin = clock();

//-------------Declare device variables-------------------


float *B_fieldD[3];
cudaMalloc( (void**) &B_fieldD, 3 * sizeof(float));
cudaMemcpy(*B_fieldD, B_field, 3 * sizeof(float),
   cudaMemcpyHostToDevice);

float *velocityD[numParticles*3];
cudaMalloc( (void**) &velocityD, numParticles * 3 * sizeof(
   float));

float *tempYPointsD[(numParticles*3*(2*SPREAD + 1))];
cudaMalloc( (void**) &tempYPointsD, (numParticles * 3 * (2*
   SPREAD + 1) * sizeof(float)));
cudaMemcpy(*tempYPointsD, tempYPoints,( numParticles * 3 * (2*
   SPREAD+1) * sizeof(float)), cudaMemcpyHostToDevice);

float *timestepD[numParticles];
cudaMalloc( (void**) &timestepD, numParticles * sizeof(float))
   ;

dim3 dimBlock((2*SPREAD + 1) , 3);
dim3 dimGrid( numParticles, 1);


//-------------------Start of velocity tracking-----------

printf("Starting RK4 calculations.\n");
```

```
while (alivesum > 0)
  {
    cudaMemcpy(*velocityD, velocity, numParticles * 3 * sizeof
        (float), cudaMemcpyHostToDevice);

    Stepper(time, tempYPoints, B_field, timestep, velocity,
        printstep, max, B_fieldD, velocityD, tempYPointsD,
        timestepD, dimBlock, dimGrid, numParticles);

    if( alive [numParticles-1] == 1)
      fprintf(velocityfile," %g  %g\n",time[numParticles-1],
          velocity[(numParticles-1)*3]);          //Prints out
          the x velocity of the final initial velocity so as to
           compare with C++ graph
    alivesum = 0;

    for(particle=0 ; particle<numParticles ; particle++)
      {
        if ( time[particle] > 100){
          alive[particle] = 0 ;
        }
        alivesum += alive[particle];

      }
  }

cudaFree( *B_fieldD );
cudaFree( *tempYPointsD );
                                                    //Free up the
   temporary device variables
cudaFree( *velocityD );
cudaFree( *timestepD );

printf("Time elapsed for CUDA calculations: %g\n", ( ( std::
   clock() - begin ) / (float)CLOCKS_PER_SEC ) );

int error_code = cudaGetLastError();
cudaError_t error_number = (cudaError_t)error_code;

if( error_code != 0 ){
  printf("Error recieved: %s\n", cudaGetErrorString(
     error_number ));
  printf("Error code: %i\n\n", error_code );
}
```

```
    return 0;
}
```

References

[1] S. Agostinelli and others, "Geant4 — a simulation toolkit", *Nuclear Instruments and Methods in Physics Research*, vol. A, no. 506, pp. 250–303, 2003.

[2] K. F. Riley, M. P. Hobson, and S. J. Bence, *Mathematical Methods for Physics and Engineering*, p. 1021, Cambridge University Press, third edition, 2006.

[3] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes in C.*, Cambridge University Press, second edition, 1992.

[4] http://www.lcsim.org/software/geant4/doxygen/html/classG4MagErrorStepper.html, "Geant4 doxygen", *Last checked on 13/09/10.*

[5] http://gpgpu.org/about, "Gpu information", *Last checked on 13/09/10.*

[6] David Kirk, nVidia, and Wen mei Hwu, "Applied parallel programming", The University of Illinois, 2006-2008, number ECE 498 AL, p. http://courses.engr.illinois.edu/ece498/al/.

[7] http://www.nvidia.com/object/product_tesla_c1060_us.html, "Tesla c1060 information", *Last checked on 13/09/10.*

[8] http://www.nvidia.com/object/product_geforce_9400m_g_us.html, "Geforce 9400m information", *Last checked on 13/09/10.*