Quarter 1 Weekly Reports

Week 1(1/13):

The goal for this week was to review and understand a CUDA simulation example, specifically the particle simulation. This simulates a large number of balls, >32k, colliding with one another inside a fixed sized box shown in fig 1.

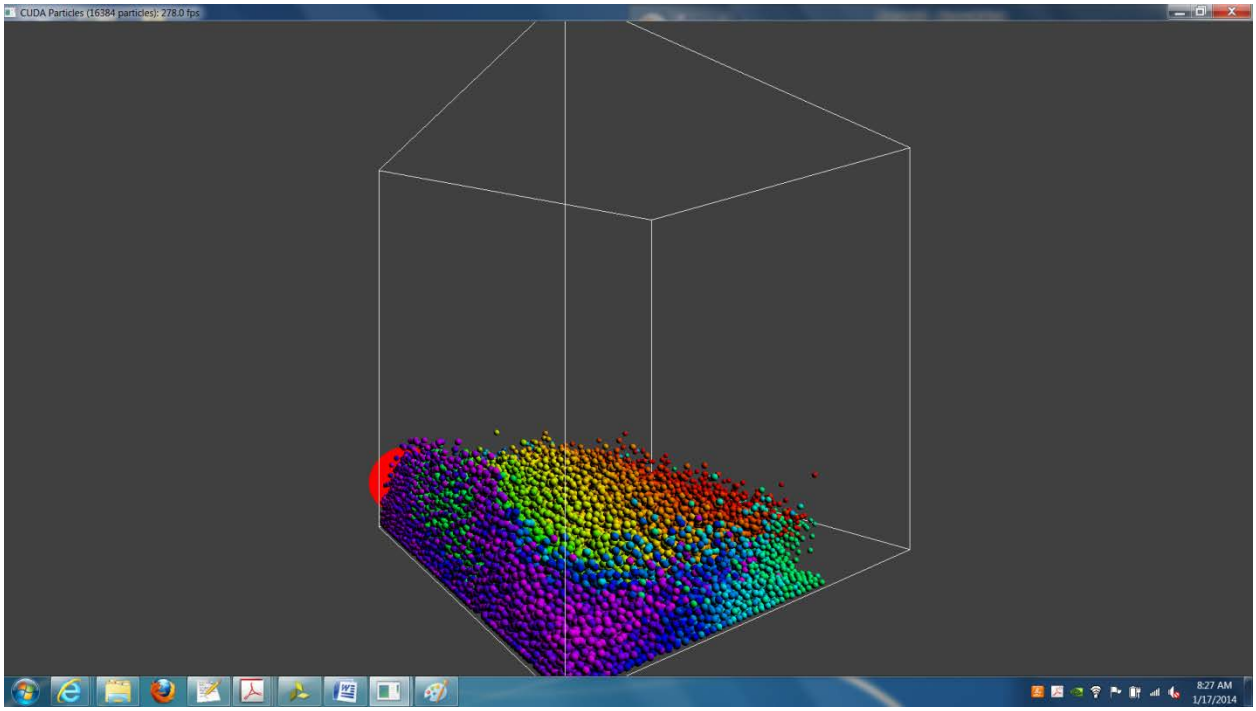

Figure 1, Particle simulation with 32k particles

So over the past week I went over the CUDA specific code which contains all of the most important algorithms for this simulation. The majority of the rest of the code is meant for the graphics, specifically OpenGL which I didn't look over.

There are three specific stages that the program goes through:

1.) Integration
2.) Data Structure Construction
3.) Collision Calculation

Step 1

The first step is integration which is the step where each particles velocity and position are updated for a set time step called deltaTime.

```
vel += params.gravity * deltaTime;
vel *= params.globalDamping;
```

```
pos += vel * deltaTime;
```

The gravity and damping factors are applied to the velocity which are constants given by the user. This function also checks if a particle is touching any of the 6 walls.

```
if (pos.x > 1.0f - params.particleRadius){
        pos.x = 1.0f - params.particleRadius;
        vel.x *= params.boundaryDamping;
}
```

The idea is that if a particle has a position beyond the wall then damping is applied to the particles velocity vector perpendicular to the wall. This will happen until the ball slows and reverses direction in the collide stage.

Step 2

The second step, which is the data structure, is the most complex part of this GPU code. This is because data structures when dealing with any GPU or parallel processor is a completely different animal to sequential processors. This basically means that typical data structures and sorting techniques have to be reworked or rethought to have good performance on GPUs.

In order to increase performance, a divide and conquer approach is used. This means the box is divided into a grid of squares, shown in fig 2, which can contain a maximum number of particles at one time.

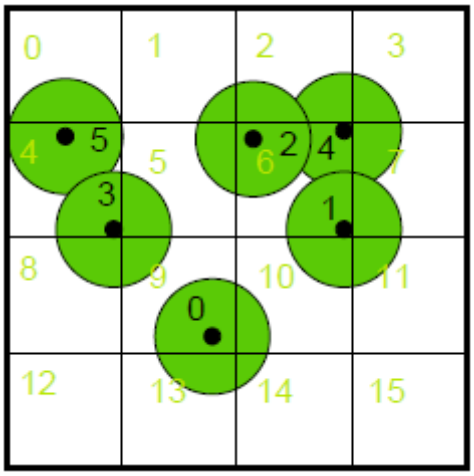

Figure 2, Grid data structure inside cube, cells hold particles

Each time deltaTime occurs the grid is re-calculated, ie each particle is placed in a maximum of four different cells. The program also offers a different approach which uses a hash table to organize the particles. The two approaches depend on what hardware is used, the first method only works for newer generations(GTX 200 series and above).

The algorithm has three steps:

1.) Calculate hash values of particles
2.) Sort particles by fast-radix sort
3.) Find start of each cell in list

The hash values in this program end up just being the cell id that that particle is in.

```
int3 gridPos = calcGridPos(make_float3(p.x, p.y, p.z));
uint hash = calcGridHash(gridPos);

gridParticleHash[index] = hash;
gridParticleIndex[index] = index;
```

Two functions are called, the first is calcGridPos which takes the x,y,z cordinate and finds the cell number. The second is calcGridHash which just calculates the index of where the particle is entered into the global array.

The second step calls a library function for CUDA to implement a fast-radix sort. This sorts the particles by both the cell and particle number. At the end of the sort, the array is scanned to store where each cell ends and begins. This is done to make the collision calculations quicker due to faster memory accesses.

Step 3

The final step is the calculation of all possible collisions of particles hitting other particles or against the walls of the box. Each particle occupies a single thread, so each thread calls these functions independently. The first function called is collideD(), which iterates over all neighboring cells to the cell that the particle is currently in. This means it iterates 3x3x3 cells around the current. This is to check if a particle collides with other particles in other cells, this is the case where the particle maybe on the edge of the cell.

```
for (int z=-1; z<=1; z++){
        for (int y=-1; y<=1; y++){
                for (int x=-1; x<=1; x++){
                        int3 neighbourPos = gridPos + make_int3(x, y, z);
                        force += collideCell(neighbourPos, index, pos, vel, oldPos,
                                oldVel, cellStart, cellEnd);
                }
        }
}
```

The function collideSphere() is called inside collideCell() when two spheres are inside of a specified constant distance. This function uses the DEM method, I'm not sure what this is, more research is needed. Supposidly this is to reduce the complexity of the calculations.

```
float3 collideSpheres(float3 posA, float3 posB,
                      float3 velA, float3 velB,
                      float radiusA, float radiusB,
                      float attraction){

    // calculate relative position
    float3 relPos = posB - posA;

    float dist = length(relPos);
    float collideDist = radiusA + radiusB;

    float3 force = make_float3(0.0f);

    if (dist < collideDist){
        float3 norm = relPos / dist;

        // relative velocity
        float3 relVel = velB - velA;

        // relative tangential velocity
        float3 tanVel = relVel - (dot(relVel, norm) * norm);

        // spring force
        force = -params.spring*(collideDist - dist) * norm;
        // dashpot (damping) force
        force += params.damping*relVel;
        // tangential shear force
        force += params.shear*tanVel;
        // attraction
        force += attraction*relPos;
    }
    return force;
}
```

This force is summed as a total of all collisions that occur with the current particle. This force is returned to the collideD() function which applies this force to the velocity of the current particle.

```
newVel[originalIndex] = make_float4(vel + force, 0.0f);
```

The constants for the spring, damping, shear, attraction etc. are listed below

```
float timestep = 0.5f;
float damping = 1.0f;
float gravity = 0.0003f;
int iterations = 1;
int ballr = 10;
```

```
float collideSpring = 0.5f;;
float collideDamping = 0.02f;;
float collideShear = 0.1f;
float collideAttraction = 0.0f;
```

These values can be changed using sliders during the simulation. The rest of the program is dedicated to user input and the graphics of the simulation which is not the focus of this week.

Conclusion

I believe that this may not be a good basis for the ultimate goal of this thesis, but it may be a good start for a simple demonstration of the smooth particle hydrodynamics. Collisions are based on a method called DEM which needs to be researched to see what exactly it is; as short description explains it as a simpler method of calculation.

In this example each particle has a short range of affect on other particles at a distance, basically limited to a radius of a sphere. There is another example called the n-body simulation which apparently takes action at a distance into account which may fit our needs better as a starting point. I propose that next week I will take a look at this example to see what the differences are and if it could be more adaptable for our needs.

Week 2(1/20):

After reviewing the Particles exam the previous week I found that this had referenced another named the "N-body Simulation". This example simulates a number of particles, 'n', which have mass and their interactions with one another, specifically gravity. An example simulation is shown in fig 3 & 4.

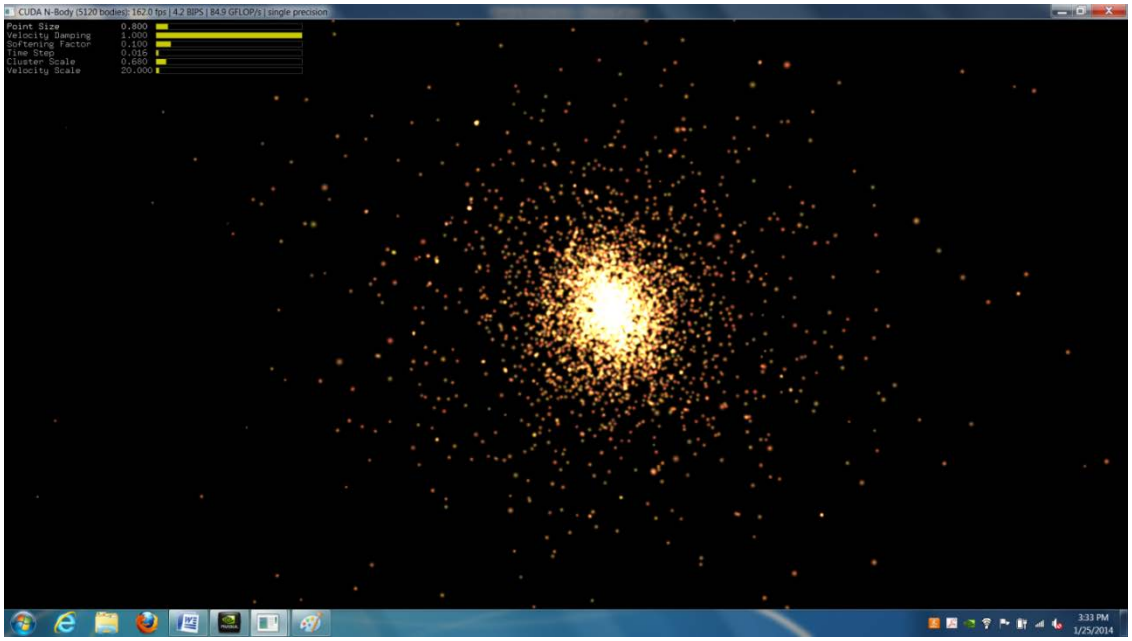

Figure 3 & 4, N-body simulation 5120k masses

The focus of this simulation is large masses, such as solar systems , and interactions between colliding galaxies. The difference between this and the previous example is the distance of interaction. So in the particle simulation the distance of interaction was limited to a few radii

away, but in this one each particle has an effect on every other one. This means that simulation is much more difficult and time consuming because each time step n^2interactions are checked. It seems like this may be a better basis for our goals because the molecules may have a larger distance of interaction than a few particles away.

Equations

The first equation that the program references looks like the equation below:

$$f_{ij} = G\frac{m_i m_j}{\left\|\vec{r}_{ij}\right\|^2}\frac{\vec{r}_{ij}}{\left\|\vec{r}_{ij}\right\|}$$

This equation seems to resemble what I remember as newtons law of universal gravitation, but the difference here is the addition of a direction and the extra fraction. This function is used in the equation:

$$F_i = \sum_{1}^{N} f_{ij} = Gm_i \sum_{1}^{N} \frac{m_j \vec{r}_{ij}}{\left\|\vec{r}_{ij}\right\|^3}$$

This summation is used to calculate the total force and direction on particle 'i' from the N-1 other particles. This may not represent the best performing calculation due to the increased cost as N gets large. The program uses a different function which approximates the summation and treats the points like galaxies.

$$F_i \approx Gm_i \sum_{1}^{N} \frac{m_j r_{ij}}{\left(\left\|r_{ij}\right\|^2 + \varepsilon^2\right)^{3/2}}$$

The constant epsilon is a softening factor, again this is added to simulate large cluster of masses, such as galaxies, which don't collide instead pass through one another. The comments in the program state that they used a leapfrog-Verlet integrator to update the velocities and positions. I'll need to look into this or ask you about it during our next meeting.

N-Body Algorithm

The data structure used to store each pair of interactions is a two dimensional array N*N in size. The row index corresponds to the particle being looked at and the subsequent columns in that row are the forces applied to the particle. This means that the sum of the columns in a specific row 'i' is the total force on particle 'i'. GPU hardware requires the software developer to divide the algorithm into blocks and threads. In the case of this example, each mass is a thread and the grid can be split into smaller sections for the blocks.

The first function call is to the integrateNbodySystem() whose main purpose is check the system status, run the block and thread size calculations, and start the integration kernals.

```
int numBlocks = (deviceData[dev].numBodies + (blockSize-1)) / blockSize;
int sharedMemSize = blockSize * 4 * sizeof(T); // 4 floats for pos

integrateBodies<T><<< numBlocks, blockSize, sharedMemSize >>>
        ((typename vec4<T>::Type *)deviceData[dev].dPos[1-currentRead],
        (typename vec4<T>::Type *)deviceData[dev].dPos[currentRead],
        (typename vec4<T>::Type *)deviceData[dev].dVel,
       deviceData[dev].offset, deviceData[dev].numBodies,
       deltaTime, damping, numBodies);
```

Thread and block size is hardware dependent, so this is never a fixed constant value. Since each thread is a particle, the integrateBodies kernal is called for each thread within a block; All of the parameters are dealing with memory and block size.

```
typename vec3<T>::Type accel = computeBodyAccel<T>(position, oldPos,
        totalNumBodies);

typename vec4<T>::Type velocity = vel[deviceOffset + index];

velocity.x += accel.x * deltaTime;
velocity.y += accel.y * deltaTime;
velocity.z += accel.z * deltaTime;

velocity.x *= damping;
velocity.y *= damping;
velocity.z *= damping;

// new position = old position + velocity * deltaTime
position.x += velocity.x * deltaTime;
position.y += velocity.y * deltaTime;
position.z += velocity.z * deltaTime;

// store new position and velocity
newPos[deviceOffset + index] = position;
vel[deviceOffset + index]    = velocity;
```

The first line is a function call to computeBodyAccel() which returns a vector for the particles acceleration. This is used in the calculation for the velocity and position vector to update the particle after a time step.

The next function I looked as is the computeBodyAccel() which is listed below:

```
typename vec3<T>::Type acc = {0.0f, 0.0f, 0.0f};

for (int tile = 0; tile < gridDim.x; tile++){
```

```
            sharedPos[threadIdx.x] = positions[tile * blockDim.x + threadIdx.x];
            __syncthreads();

            // This is the "tile_calculation" from the GPUG3 article.
            #pragma unroll 128

            for (unsigned int counter = 0; counter < blockDim.x; counter++){
                  acc = bodyBodyInteraction<T>(acc, bodyPos, sharedPos[counter]);
            }

            __syncthreads();
      }
      return acc;
```

The biggest part of this function is the for loop in the middle which runs through the block and calculates all of the acceleration vectors for the pairs.

The last function in the CUDA kernal of this example is the bodyBodyInteraction() which is where the function listed earlier is actually calculated.

```
// r_ij  [3 FLOPS]
r.x = bj.x - bi.x;
r.y = bj.y - bi.y;
r.z = bj.z - bi.z;

// distSqr = dot(r_ij, r_ij) + EPS^2  [6 FLOPS]
T distSqr = r.x * r.x + r.y * r.y + r.z * r.z;
distSqr += getSofteningSquared<T>();

// invDistCube =1/distSqr^(3/2)  [4 FLOPS (2 mul, 1 sqrt, 1 inv)]
T invDist = rsqrt_T(distSqr);
T invDistCube =  invDist * invDist * invDist;

// s = m_j * invDistCube [1 FLOP]
T s = bj.w * invDistCube;

// a_i =  a_i + s * r_ij [6 FLOPS]
ai.x += r.x * s;
ai.y += r.y * s;
ai.z += r.z * s;
```

First the distance vector is calculated by subtracting the $b_j - b_i$ where 'i' is the current particle. The next two lines are the $\|r_{ij}\|^2$ and ε^2 calculations which are fairly straightforward. The rest is the same sort of thing, just calculating the equation.

The rest of the program deals with graphics which again I'm not going to look at right now because it isn't important. The important part is the actual calculation which is inside the CUDA kernal.

Conclusion

The main purpose of this example is to show that the GPU can handle an n-body simulation which means each particle or body in the simulation interacts with every other body. To make the calculations quicker a modified version of the universal law of gravitation is used which approximates the calculation. The only problem is that collisions are not taken into account, but this may be implementable in the future. As of now it would be good for simulation of a static object, but maybe not a mass which interacts with something in the outside world like a surface.

Goal for next two weeks

I think a good thing to do is to modify this program to either include collisions or to modify it to fit our need more. Maybe add a surface to collide against, but maybe not implement collision with other particles. We can discuss this at the next meeting.

Week 3(1/27)

The goal for this week was to find or write the equations that make the particle simulation work. These are important because they are the key to finding an entry point to the program in order to adapt this to our needs. The problem is that the calculation is done over several steps which means in multiple functions. The other issue is that each direction, x, y and z are not explicitly defined in calculations. I believe that float3 and float4 handle addition and multiplication, kind of like a library function for regular arithmetic operations.

There are two main things that I looked at, the first being the equations and the second is what float3 and float4 represent. The first thing I looked at was float3 which is important to understand first so that possible vectors can be identified in the equations later. From the nvidia website, the data fields are listed as 'x', 'y', and 'z' which are seen in figure 5.

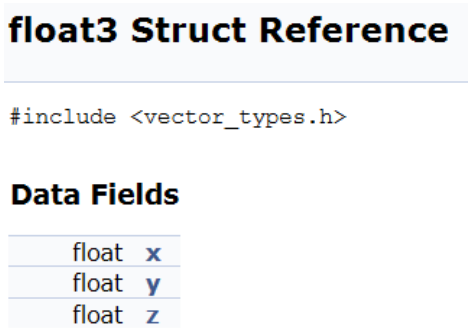

Figure 5, float3 documentation on NVIDIA website showing x,y,z

This is the first good sign that a float3 data type is meant to be a vectors direction which simplifies the equations considerably in the code. Looking more closely at the code in this light shows that CUDA supports +/-, *, and dot products between vectors. It also has a length function which returns the magnitude or length of the vector. Most of these operators and functions are used in the calculation which is why the code is simplified.

The 'integration' function is the first called which calculates the new position and updates the velocity with respect to gravity and air resistance. In the equations below 'GD' is a constant called GlobalDamping, 'G' is a gravity constant, and 'dt' is a small change in time or deltaTime. The updated velocity and positions are needed for the collision function later in stage three.

$$\vec{\boldsymbol{v}}_{\boldsymbol{t+\Delta t}} = GD(\vec{\boldsymbol{v}}_{\boldsymbol{t}} + G\Delta t)$$

$$\begin{aligned}\overrightarrow{\boldsymbol{pos}_{\boldsymbol{t+\Delta t}}} &= \overrightarrow{\boldsymbol{pos}_{\boldsymbol{t}}} + \overrightarrow{\boldsymbol{v}_{\boldsymbol{t+\Delta t}}} \cdot \Delta t \\ &= \overrightarrow{\boldsymbol{pos}_{\boldsymbol{t}}} + \Delta t[GD(\vec{\boldsymbol{v}}_{\boldsymbol{t}} + G \cdot \Delta t)] \\ &= \overrightarrow{\boldsymbol{pos}_{\boldsymbol{t}}} + GD(\vec{\boldsymbol{v}}_{\boldsymbol{t}}\Delta t) + GD(G \cdot \Delta t^2)\end{aligned}$$

Each velocity and position is stored in a float3, thus a vector. These are stored back in the hash-table by the function I went over during the first week, so I won't go over this again.

The next function is the actual collision calculation which calculates pair-wise collisions in each cell; each cell can hold up to four particles. Vectors are marked with bold font with the arrows above them. All vectors are stored as float3s except the last equation below which is a float4.

$$dist = length(relPos)$$
$$\overrightarrow{\boldsymbol{relpos}} = \overrightarrow{\boldsymbol{posB}} - \overrightarrow{\boldsymbol{posA}}$$

$$\overrightarrow{\boldsymbol{norm}} = \frac{\overrightarrow{\boldsymbol{relPos}}}{dist}$$

$$\overrightarrow{\boldsymbol{relVel}} = \overrightarrow{\boldsymbol{velB}} - \overrightarrow{\boldsymbol{velA}}$$
$$\overrightarrow{\boldsymbol{tanVel}} = \overrightarrow{\boldsymbol{relVel}} - \overrightarrow{\boldsymbol{norm}} \cdot (\overrightarrow{\boldsymbol{relVel}} \bullet \overrightarrow{\boldsymbol{norm}})$$

$$x = collideDist - dist$$
$$\zeta = Damping\ ratio\ (0 \rightarrow 1.0)$$
$$k = spring\ constant\ (0 \rightarrow 1.0)$$
$$\tau = shear\ constant\ (0 \rightarrow 0.1)$$
$$\alpha = attraction\ (0 \rightarrow 0.1)$$
$$g = gravity\ constant\ (0.001 \rightarrow 0.003)$$

$$\overrightarrow{\boldsymbol{force_{i\ on\ 0}}} = -kx \cdot \overrightarrow{\boldsymbol{norm}} + \zeta \cdot \overrightarrow{\boldsymbol{relVel}} + \tau \cdot \overrightarrow{\boldsymbol{tanVel}} + \alpha \cdot \overrightarrow{\boldsymbol{relPos}}$$

$$Total\ Force\ on\ Particle_0 = \sum_{i=1}^{N} \boldsymbol{force_{i\ on\ 0}}$$

After the force on a single particle is calculated in collideCell(), the function returns and adds this to the velocity that was calculated in the integration stage. This doesn't make much sense to me, but maybe its correct. This velocity is then placed in the original index then the process starts over again for a new deltaTime.

$$\vec{\boldsymbol{v}}_{\boldsymbol{t+\Delta t}} = GD(\vec{\boldsymbol{v}}_{\boldsymbol{t}} + Gdt) + \sum_{i=1}^{N} \boldsymbol{force_{i\ on\ 0}}$$

So the three most important final equations are listed below:

$$\overrightarrow{\boldsymbol{pos_{t+\Delta t}}} = \overrightarrow{\boldsymbol{pos_t}} + GD(\overrightarrow{\boldsymbol{v_t}} \cdot \Delta t) + GD(G \cdot \Delta t^2)$$

$$\overrightarrow{\boldsymbol{force_{i\ on\ 0}}} = -kx \cdot \overrightarrow{\boldsymbol{norm}} + \zeta \cdot \overrightarrow{\boldsymbol{relVel}} + \tau \cdot \overrightarrow{\boldsymbol{tanVel}} + \alpha \cdot \overrightarrow{\boldsymbol{relPos}}$$

$$\vec{\boldsymbol{v}}_{\boldsymbol{t+\Delta t}} = GD(\vec{\boldsymbol{v}}_{\boldsymbol{t}} + G \cdot \Delta t) + \sum_{i=1}^{N} \boldsymbol{force_{i\ on\ 0}}$$

The program reference something called the DEM which was used in the calculation process. DEM stands for the discrete element method which is used to simulate a large group of particles. This is not just one set of equations, but rather consists of many different methods. It is used to calculate particle directions using Verlet integrator. I'm not sure how to tell what is part of the DEM and if these equations represent anything significant or existing.

Conclusion

This week's goal was to define the equations used in the program and what corresponds to the DEM. I have listed the equation above but may need more guidance about the math and physics behind these equations. No information or questions have been brought up about this algorithm that I can find.

Week 4+5(2/4->2/18)

The goal for these two weeks was to document the code, create a flow chart, and make modifications to the sliders to view more drastic changes. The first portion of documentation involves writing a structure like listed below:

```
/**
        Function that iterates over all of the particles in a cell and calculates all
        collisions with the original particle. This function is called for each of the eight
        cells around the center cell where the particle is located. Forces are summed and
        returned to the collideD function.

        @param gridPos cell number in grid in x,y,z coordinates
        @param index index of particle being looked at by thread
        @param pos vector containing the position of the current particle in x,y,z
        @param vel vector containing velocity vector of current particle
        @param *oldPos pointer to array containing all of the position vectors indexed by the
                particle number
        @param *oldVel pointer to array containing all of the velocity vectors indexed by the
                particle number
        @param *cellStart pointer to array containing the start pointer of each cell in the arrays
        @param *cellEnd pointer to array containing the end pointer of each cell in the arrays

        @return force sum total of all forces on particle from other particles in current cell
*/
```

This layout is useful because certain software can automatically generate a documentation file for each function. These descriptions are used for each gpu function in the .cu and as well as the cpu code in .cpp files.

In order to find a set of constants that give the sphere more solid like properties, the sliders range has to be increased beyond the standard. This was difficult to locate as the documentation is non-existent and where they are doesnt mention anything about sliders. The code is listed below

```
params = new ParamListGL("misc");
params->AddParam(new Param<float>("time step", timestep, 0.0f, 1.0f, 0.01f, &timestep));
params->AddParam(new Param<float>("damping"  , damping , 0.0f, 10.0f, 0.1f, &damping)); //Original 1.0f, 0.01f
params->AddParam(new Param<float>("gravity"  , gravity , 0.0f, 0.01f, 0.001f, &gravity)); //Original 0.001f,
0.0001f
params->AddParam(new Param<int> ("ball radius", ballr , 1, 20, 1, &ballr));
params->AddParam(new Param<float>("collide spring" , collideSpring , 0.0f, 10.0f, 0.01f, &collideSpring));
//Original 1.0f, 0.001f
params->AddParam(new Param<float>("collide damping", collideDamping, 0.0f, 1.0f, 0.01f, &collideDamping));
//original 0.1f, 0.001f
params->AddParam(new Param<float>("collide shear"  , collideShear  , 0.0f, 1.0f, 0.01f, &collideShear)); //original
0.1f, 0.001f
params->AddParam(new Param<float>("collide attract", collideAttraction, 0.0f, 10.f, 0.1f, &collideAttraction));
//original 0.1f, 0.001f
```

These values are updated with each time step, so before each collision these values are checked and possibly changed.

The other goal was to increase the number of particles so a sphere of particles can impact a thick block of particles. This is relatively simple, all that has to be done is changing the constant value shown below:

```
#define NUM_PARTICLES   32384
```

This could be increased to whatever is needed, so when I start using the sever I should be able to simulate over 1 million particles easily.

The big issue I have run into is compiling these programs. When I do compile I get no errors and an executable is created. The problem is that the GPU is not utilized other than for the graphics as shown in figure 6.

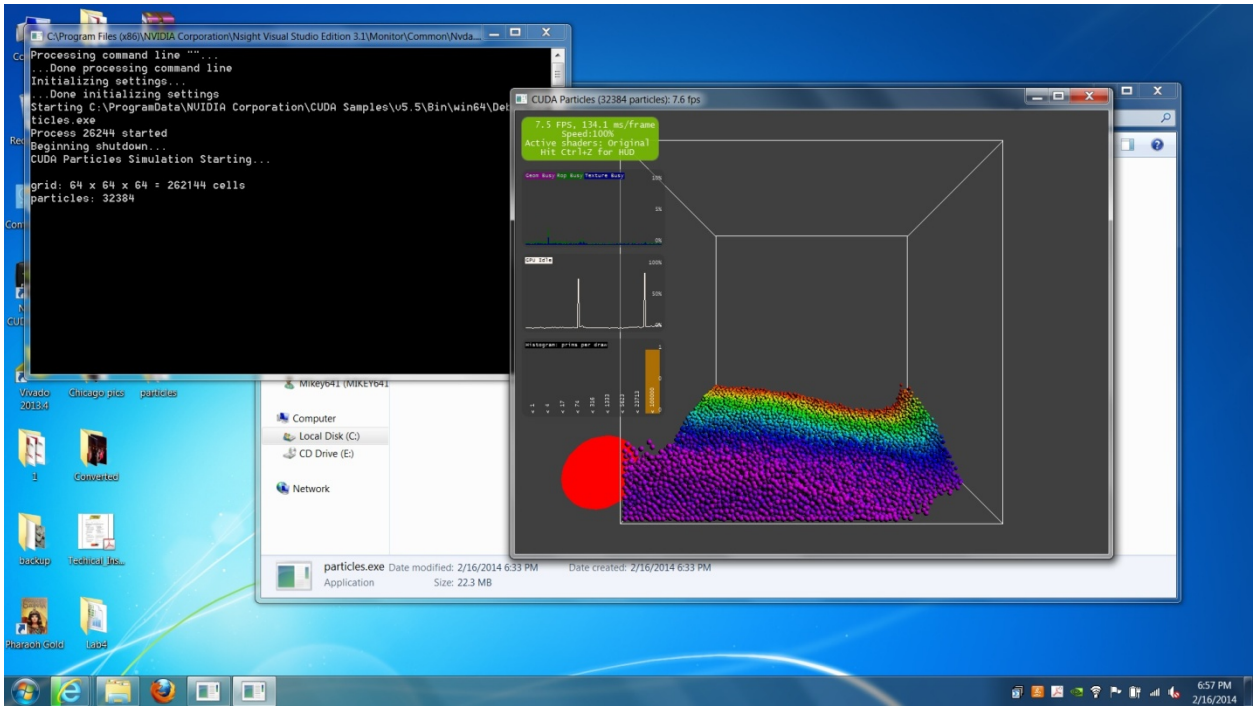

Figure 6, graph showing GPU activity near 0% during my compiled program

The activity graph for the GPU utilization shows almost no activity during the simulation. Upon further investigation the cpu has 25% activity which means one core is at 100% usage. This makes sense because this isnt a multithreaded CPU application so only one core is used. The problem is that the GPU is not used which means the calculation is done completly on the CPU. This ends up with very low performance, around 3-7 frames-per-second versus ~300 for the supplied .exe. The GPU graph should look like the one in Figure 7.

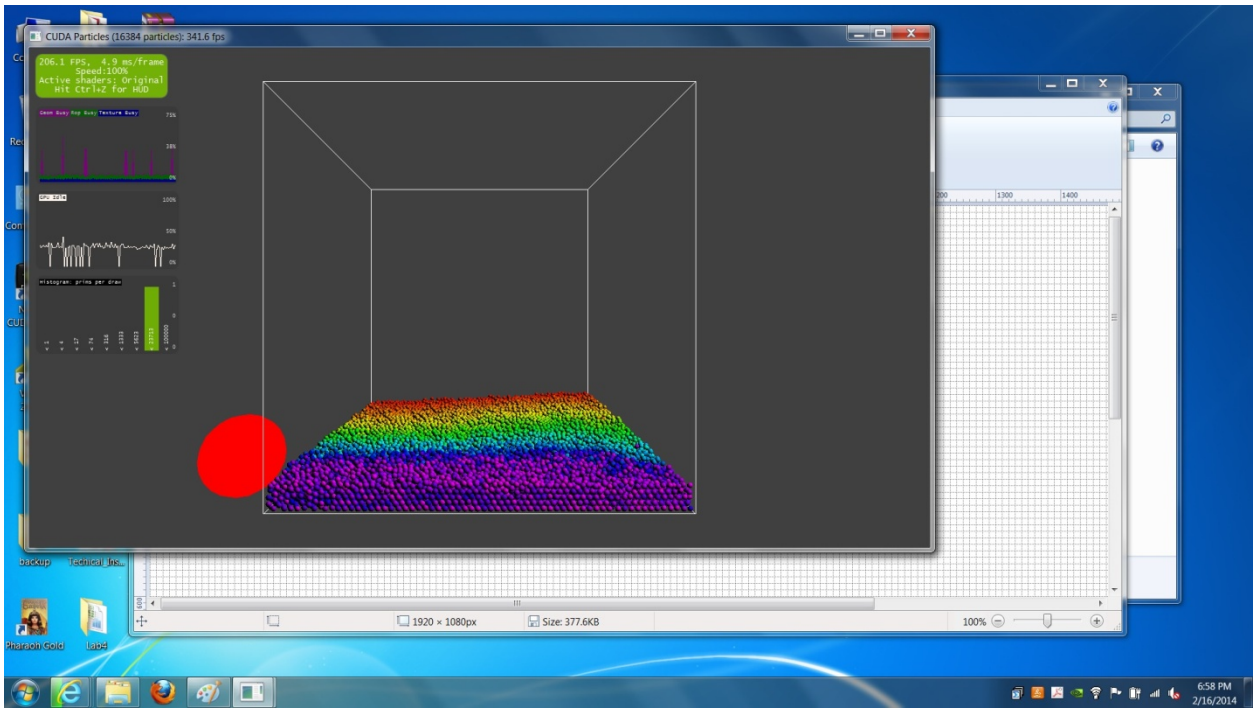

Figure 7, Graph shows up to 50% utilization rather than near 0%

What I need to do is create a makefile for this program to either compile on this machine or the GPU server. The server would need software installed on it to do the compilation and running. I have the SSH client to access the server and Xming to view GUI of programs run on the server. I have the form to sign up for access to the servers on campus I just need signatures and to turn it in to BE 311.