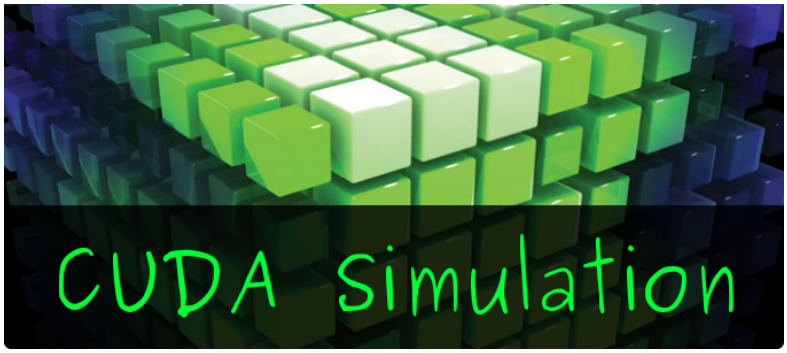

Jedediyah Williams
willij16@rpi.edu

May 2, 2013

1 Introduction

I present multiple small simulators implemented using OpenGL and NVIDIA's Compute Unified Device Architecture (CUDA) for parallel computing on GPU. General purpose GPU computing is becoming more populer, and is currently becoming an important tool in multibody dynamics. For example, the Bullet physics library [1] has been focused heavily on CUDA based implmentations of dynamics functions in its recent releases.

Having never studied parallel computing before, a significant amount of time was spent reading and learning how to properly form kernel functions to fully exploit the architecture of CUDA enabled graphics cards. Thanks go to Josh for lending me "Cuda by Example" [2], which is how I spent most of my reading time and learned most of what I now know about CUDA. NVIDIA's code samples [3] were also helpful when first starting out, particularly "simpleGL" which demonstrates how to use basic interoperability between CUDA and OpenGL.

Since I planned on working with triangle meshes, I had hoped I'd be able to use and modify the code from class assignments, however hash.h will not compile with CUDA because *unordered_map* is not compatible with CUDA. So I had to start from nearly scratch, re-using code snippets where I could.

2 Simulating with CUDA

2.1 Basics of CUDA

Developing in CUDA involves writing heterogeneous code that consists of both traditional C++ as well as kernel code that is compiled to run on a GPU device. Conveniently, CUDA code was designed to use C++ like syntax, giving the programmer the sense that she is writing in a homogeneous environment.

Common uses will typically involve copying data from host memory to device memory, operating on that data in parallel, then copying resultant data from device back to host. An example of this is shown in the complete "hello world" example of a CUDA program in figure 1.

```
__global__ void addVectors( int *d_A, int *d_B, int *d_C ) {
    int t = blockIdx.x*blockDim.x + threadIdx.x; // Thread ID
    d_C[t] = d_A[t] + d_B[t];   // C = A + B
}

int main() {
    const int N = 1000;   // Length of vectors
    int *h_A, *h_B, *h_C; // Host variables
    int *d_A, *d_B, *d_C; // Device variables

    // Allocate both host and device memory
    h_A = (int*)malloc( N*sizeof(int) );
    h_B = (int*)malloc( N*sizeof(int) );
    h_C = (int*)malloc( N*sizeof(int) );
    cudaMalloc( (void**) &d_A, N*sizeof(int) );
    cudaMalloc( (void**) &d_B, N*sizeof(int) );
    cudaMalloc( (void**) &d_C, N*sizeof(int) );

    // Initialize host values
    for (int i=0; i<N; i++) { h_A[i] = i;  h_B[i] = i^i; }

    // Copy A and B from host to device
    cudaMemcpy( d_A, h_A, N*sizeof(int), cudaMemcpyHostToDevice );
    cudaMemcpy( d_B, h_B, N*sizeof(int), cudamemcpyHostToDevice );

    // Call CUDA kernel to add A+B -> C
    addVectors<<<1,N>>>(d_A, d_B, d_C);

    // Copy result C from device to host
    cudaMemcpy( h_C, d_C, N*sizeof(int), cudaMemcpyDeviceToHost );

    for (int i=0; i<N; i++) printf("C[%d] = %d\n", i, h_C[i]);

    // Free memory
    free(h_A); free(h_B); free(h_C);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

Figure 1: Example CUDA program.

2.2 Additional Benefits of GPU Programming

In addition to the benefits of parallel simulation, running simulations in CUDA provides other benefits as well. In particular, because CUDA kernel code is being executed on a GPU, it has access to the graphics memory. Because of this, it is easy to for exapmle manipulate VBO vertex data directly from the GPU itself instead of relying on updates from a CPU process. This technique is called interoperability, and is employed in all of the CUDA simulations presented here.

3 Particle Based Cloth Simulation

The first parallel simulation I implemented used the mass-spring model of cloth described by Provot [4]. This is a model that was implemented on CPU for our second assignment, is highly parallelizable, and is straight-forward. These attributes made it a great first simulation since I had a serial implementation to compare to, and was practically guaranteed good speedup results. The model uses a regular grid of point mass particles attached to neighboring particles by virtual springs. Three different spring linkages are used for a given particle point P:

- *Structural* springs which connect to the particles directly above, below, left, and right of P.
- *Shear* springs which connect particles one diagonal element away from P.
- *Flexion* springs which are similar to structural springs but connect two elements away from P instead of one.

At every simulation step, the internal forces of each particle point P_{ij} are calculated based on the distance l to each connected particle, as well as the rest length l^0 to each respective particle. After the forces are calculated for each particle, we doubly integrate the acceleration to get the new position of the particle. However it is quite possible, in fact likely that after this stage, particles have moved away from some particles to positions which will generate large forces in the next time step. This results in So a correction is iteratively applied in which we sequentially test every pair of connected particles and project their positions to a maximum stretch of 1.1 times their rest length l^0 if this constraint is violated. This sequence of steps is outlined in algorithm 1.

Algorithm 1 Provot Mass-Spring Model for Rigid Cloth Behavior

1. for each timestep of size Δt
2. for each particle P_{ij}
3. $\mathbf{F}_{ij}(P_{ij}) = -\Sigma_{k,l \in R} K_{i,j,k,l}[l_{i,j,k,l} - l^0_{i,j,k,l} \frac{l_{i,j,k,l}}{||l_{i,j,k,l}||}]$
4. for each particle P_{ij}
5. $\mathbf{a}_{ij}(t + \Delta t) = \frac{1}{m} F_{ij}$
6. $\mathbf{v}_{ij}(t + \Delta t) = \mathbf{v}_{ij} + \Delta t \mathbf{a}_{ij}(t + \Delta t)$
7. $P_{ij}(t + \Delta t) = P_{ij}(t) + \Delta t \mathbf{v}_{ij}(t + \Delta t)$
8. for i from 1 to 6
9. for each pair of connected particles P_k and P_l
10. if $l > 1.1 l^0$
11. move P_k and P_l toward each other a distance $\frac{l - 1.1 l^0}{2}$

The parallel implementation of this algorithm in CUDA still calculates all of the values as the serial algorithm above for lines 3, 5, 6, and 7. The one implementation difference is that in the parallel version, each particle is simultaneously determining the correction vector in line 11 based on the position of

all its neighbors current positions before updating to a new position. This is of course faster because it is done in parallel, but it is conveniently also more stable! As a result, the GPU implementation was able to handle much larger meshes without becoming unstable. The largest mesh I ran had 1,000,000 particles and took approximately 0.2 seconds per simulation step.

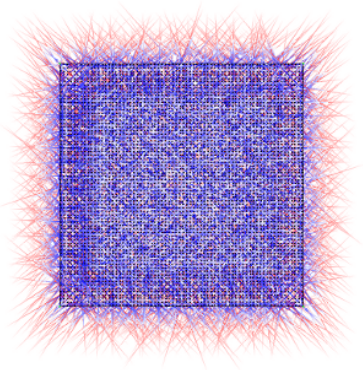

Figure 2: View of 10,000 particle cloth becoming unstable on CPU.

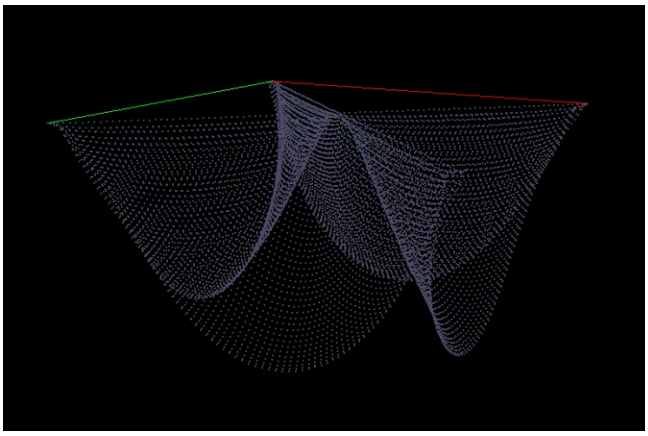

Figure 3: 10,000 particle cloth rendering on GPU.

3.1 Comparison of CPU vs. GPU

A benchmark comparison was done in order to compare the performance of the two different implementations of the cloth simulation. The following initial simulation parameters were used for both:

- Timestep = 0.001 seconds
- Particle mass was set equal to 0.1 * $MESH_{width}$ * $MESH_{height}$

- Gravitational acceleration was 9.8 m/s^2 in the $-y$ direction
- Spring constants $K_{structural} = 5$, $K_{shear} = 2$, $K_{bend} = 2$
- Points along the diagonals of the mesh were set as fixed
- 10 simulation iterations were completed for each graphics rendering

Several trials (about 10) were run for various square meshes, and the results were averaged per mesh size. The metric used was the amount of time in seconds to complete the first 1000 simulation steps. These results are presented in figures 4 and 5.

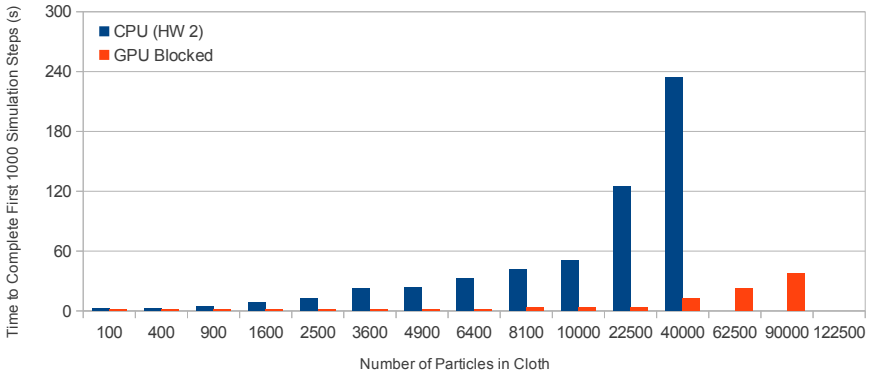

Figure 4: Performance of CPU and GPU implementations of cloth simulation.

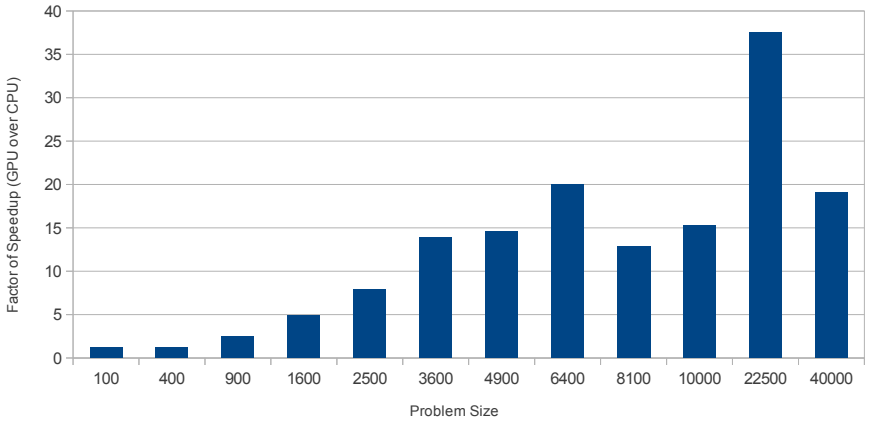

Figure 5: Factor of speedup for GPU over CPU in cloth simulation.

The results are what we expect: a quadratic performance decrease for the CPU implementation as a function of the grid width, and a significant speedup

in the GPU over the CPU as the grid size increases. The GPU performance is precisely constant for the first six trials, taking 1.65 seconds to complete the first 1000 timesteps. For problems of size 40,000 and up, the GPU performance time approximately doubles for each of the few remaining trials.

The reason that the GPU implementation does not remain constant is that a hardware limit is reached: I was using a EVGA GeForce GTX 680 graphics card, which has 8 multiprocessors $\times$ 192 CUDA cores per multiprocessor for a total of 1536 CUDA cores. During my experiments I was using block sizes that were multiples of 5, with 1024 threads per block. So as soon as the number of threads exceeded $5 * 1536 = 7680$, more threads needed processing than there were available cores. In retrospect, now that I understand the CUDA architecture better, I believe I was leaving a fraction of the CUDA cores unused and I could have increased the GPU performance by using block sizes that were multiples of 8 instead of 5.

4 Rigid Body Simulation

The particles in the cloth simulation were straight forward to represent since each particle is an individual body in a grid structure. For N particles which all have equal mass, we use an array to store position, velocity, acceleration, and force, *all* of which have size $3N$. Indexing such a representation is simple. Rigid bodies composed of triangle meshes provide a greater challenge. Separate meshes don't necessarily have the same number of vertices, edges, nor faces. This will bring us back to the idea briefly mentioned in section 2.2, CUDA interoperability.

4.1 Rigid Body Representation

To start, there are several attributes that each body must store for use in dynamics. An array of the following *Body* struct is used to store information for each body.

```
struct Body {   int bodyID;
                int dynamicsID;
                uint local_start, local_end;
                float mass;
                float4 quat;
                float3 position,
                lin_velocity,
                rot_velocity,
                lin_acceleration,
                rot_acceleration,
                force,
                torque;
                float3 Fext, Text; };
```

The array of body information is copied to GPU global memory during initialization of the simulation, after all bodies have been created.

Vertex information for each body is stored twice: once as a constant stacked vector of local vertex positions for all bodies, and a second time as graphics data that is created as a VBO. The local coordinates are copied to GPU memory along with the body information, and used during simulation to update the world coordinate vertices stored as graphics data.

4.2 Collision Detection

Idealy, I would have liked to implemented mesh-mesh collision detection, but that is a difficult problem beyond the scope of this project. Additionally, acceleration data structures like a k-d tree or even a uniform grid would be very useful during broad-phase collision detection.

The current state of the rigid body simulator does not include multibody dynamics, it only contains planar collision with a ground plane. At each timestep, each vertex (for all bodies) determines its height. If the height is negative, it atomically adds a force and torque proportional to the penetration depth to the corresponding body's applied force. For a body with center of mass located at C, and a force F applied at point p on the body, let r be the vector from C to p. The resultant force on the body is simply F, and the torque τ is given by $\tau = r \times F$.

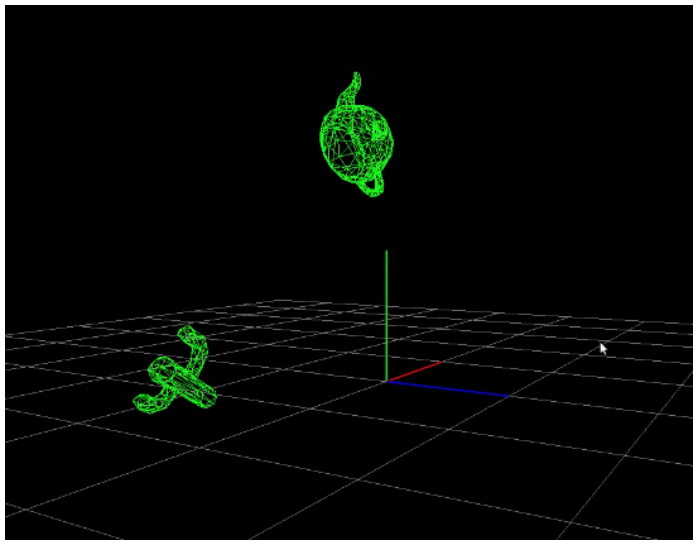

Figure 6: Example simulation of a cactus and Utah teapot.

5 Dynamics Solution

Although a proper dynamics formulation for multi-bodies is not yet implemented, one of the first sets of CUDA kernel functions I wrote were variations on the Gauss-Seidel method for iteratively solving systems of linear equations. All variations solved the same type of problem: given square matrix A, vector b, and initial guess x_0, return an approximate solution x where $Ax + b = 0$.

Algorithm 2 Parallel Gauss-Seidel

Given matrix A, vector b, and initial guess x_0 of length N
Let x_1 and r be vectors of length N
    $t \leftarrow$ thread ID
    for i from 1 to $Max\_Iterations/2$
        $r \leftarrow b[t] + A_{t,1:N} \cdot x_0$
        $x_1[t] \leftarrow x_0[t] - r/A_{t,t}$
        Synchronize_Threads()
        $r \leftarrow b[t] + A_{t,1:N} \cdot x_1$
        $x_0[t] \leftarrow x_1[t] - r/A_{t,t}$
        Synchronize_Threads()

Algorithm 3 Parallel Successive Over-Relaxation (SOR)

Given matrix A, vector b, and initial guess x_0 of length N
Let x_1 and r be vectors of length N
Let λ be a tunable parameter between 0 and 2
    $t \leftarrow$ thread ID
    for i from 1 to $Max\_Iterations/2$
        $r \leftarrow b[t] + A_{t,1:N} \cdot x_0$
        $x_1[t] \leftarrow x_0[t] - \lambda r/A_{t,t}$
        Synchronize_Threads()
        $r \leftarrow b[t] + A_{t,1:N} \cdot x_1$
        $x_0[t] \leftarrow x_1[t] - \lambda r/A_{t,t}$
        Synchronize_Threads()

Algorithm 4 Parallel Projected Successive Over-Relaxation (pSOR)

Given matrix A, vector b, and initial guess x_0 of length N
Let x_1 and r be vectors of length N
Let λ be a tunable parameter between 0 and 2
    $t \leftarrow$ thread ID
    for i from 1 to $Max\_Iterations/2$
        $r \leftarrow b[t] + A_{t,1:N} \cdot x_0$
        $x_1[t] \leftarrow max(0, x_0[t] - \lambda r/A_{t,t})$
        Synchronize_Threads()
        $r \leftarrow b[t] + A_{t,1:N} \cdot x_1$
        $x_0[t] \leftarrow max(0, x_1[t] - \lambda r/A_{t,t})$
        Synchronize_Threads()

Algorithms 2, 3, and 4 are three similar solvers. All three use the same tool in order to allow simoultaneous solution per row: a second solution vector x_1. Each iteration of the loop reads from x_0 and writes to x_1, then reads from x_1 and writes to x_0. Further, threads are synchronized between these two stages, guaranteeing safe read from one vector while writing to the other. The dot product was also implemented as a kernel function.

The first solver is not technically a classical Gauss-Seidel solver since this writing occurs in a different vector and does not forward substitute during iterations. It is in fact a special case of a blocked Gauss-Seidel method with block size of 1.

The second solver uses successive over-relaxation (SOR) in order to converge more quickly. The value of λ used is tunable and must be in the range $0 \leq \lambda \leq 2$. During testing, I found that a value of $\lambda = \frac{\sqrt{2}}{2}$ worked well for problems I was testing, and convergence to 0.001 was usually reached within 5 or 6 iterations.

The third solver is a projected SOR method (pSOR). This is used when we require the result to be non-negative. It uses the same method as SOR, but if a solution element is found to be less than zero, it is "projected" to zero.

6 Kinematic Update

After forces and torques are found for each body, accelerations are determined and integrated to get updated linear and rotational velocities. The quaternion q representing the rotation is updated by

$$q \leftarrow \{cos(|\omega \Delta t|/2), \frac{\omega}{|\omega \Delta t|} sin(|\omega \Delta t|/2)\} * q$$

where ω is the angular velocity and the $*$ operator represents quaternion multiplication.

Once the new center of mass positions and rotations are known, the world-frame vertex positions are updated by transforming the local vertex data stored in global GPU memory and placing the result with the graphics data. The transformation involves translation by the body's position, then rotation by the body's quaternion. I wrote a small but very useful quaternion library in CUDA for use by kernel functions on the GPU.

7 Possible Future Work & Conclusion

I learned a lot about CUDA in the past four weeks, and feel I could effectively use it in research now. Although I wasn't able to implement a complete and robust multibody physics engine for this project, I was happy to get some simple dynamics working and running fully on GPU. In particular, I think OpenGL Interop is interesting and useful. If I continue work on the project, I'd like to look deeper into the work by Tasora and Negrut [5], particularly concerning efficient storage of datastructures.

The following short list is composed of the things I think would be on the soon-to-do list for the rigid-body simulator.

- Script parsing for loading simulation scenes
- Collision detection - Broad phase kd-tree
- Non-penetration and friction constraint dynamics

Video examples of the simulators running are available at the following URL's:
10,000 particle cloth: `http://www.youtube.com/watch?v=X-Yfq-_vmRU`
22,500 particle cloth: `http://www.youtube.com/watch?v=AoeSudIE6Hg`
Bouncing bunny! `http://www.youtube.com/watch?v=bxKYPyj52pU`
Bouncing cactus & teapot: `http://www.youtube.com/watch?v=GcunmvLDmek`
Bunny blooper: `http://www.youtube.com/watch?v=4qBeQct5iow`

References

[1] Erwin Coumans, *Bullet Physics Library: a professional free 3D Game Multiphysics Library*. `code.google.com/p/bullet`

[2] Jason Sanders and Edward Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. NVIDIA Corporation, July 2010.

[3] NVIDIA, CUDA Tookit Documentation. `http://docs.nvidia.com/cuda/cuda-samples/index.html`, 2007-2012.

[4] Xavier Provot, *Deformation constraints in a mass-spring model to describe rigid cloth behavior*. In Graphics Interface, 1995, (147-154), 1995.

[5] Alessandro Tasora and Dan Negrut, *A parallel algorithm for solving complex multibody problems with stream processors*. 2009.

[6] Claude Lacoursiere, *A Parallel Block Iterative Method for Interactive Contacting Rigid Multibody Simulations on Multicore PCs*. pp.956-965 In proceeding of: Applied Parallel Computing. State of the Art in Scientific Computing, 8th International Workshop, PARA 2006, Umea, Sweden, June 18-21, 2006.