

**Homework 4**  
**Due: 6/18/09**

**CIS 665**

**Problem 0 : (0 Points)**

- Set CUDA up: [http://www.nvidia.com/object/cuda\\_get.html](http://www.nvidia.com/object/cuda_get.html)
- Verify your installation by checking out a couple examples in the SDK code examples

**Problem 1: Matrix Multiplication CG vs. CUDA (20 points)**

From assignment 3 you should have a working matrix-matrix multiplication algorithm. Please time (clock\_t, clock() are fine) your code.

Using the format discussed in assignment 2, generate 3 dense matrices of sizes 256x256, 512x512, 1024x1024 of your choosing. (Hint: I would start with the identity matrix to verify your calculation or use matlab to double check your calculations, but anything works to generate it and check it, you can use the CPU implementation in the CUDA example discussed next.)

After you massage the CUDA matrixMul program to handle your input, run the code for the same 3 input matrices and report the timings.

Compare and contrast why one executes faster than the other in your README.

**Extra Credit (5 Points):** Do a similar test for matrix multiplication using the CUBLAS library and do a similar report for the input matrices.

**Problem 2 : Cloth on the GPU using CUDA (25 points)**

Clothing is simulated using large mass-spring systems. Each vertex on the cloth is connected to its surrounding vertices using a mass spring system. This results in a large number of mass-spring simulations that can be handled in parallel. In this assignment you will simulate the mass spring system in CUDA and display the results in OpenGL. The forces that springs exert on connected points are obtained from Hooke's law:

$$F_{ij}^i = F_{ij}(x_i, x_j) = D_{ij} \frac{|l_{ij}| - |l_{ij}^0|}{|l_{ij}|} * l_{ij}$$

$D_{ij}$  – stiffness of the spring connecting points  $x_i$  and  $x_j$

$l_{ij}$  – distance between  $x_i$  and  $x_j$

$|l_{ij}^0|$  - rest length of spring in its initial configuration

$F_{ext}^i$  - the sum of the external forces exerted on the point

Point positions are updated using the Lagrangian law of motion:

$$F_{ext}^i = m_i \ddot{x}_i + c \dot{x}_i + \sum_{j \in N_i} F_{ij}(x_i, x_j) \quad (\text{Equation 1})$$

$m_i$  – mass constant

$c$  – damping constant

$\ddot{x}$  - acceleration

$\dot{x}$  - velocity

$N$  – number of points adjacent to the current point

Using Verlet integration we do not need to explicitly calculate or store the velocities. New point positions  $x_i$  can be computed as:

$$x_i(t + dt) = \frac{F_i^{tot}(t)}{m_i} dt^2 + 2x_i(t) - x_i(t - dt) \quad (\text{Equation 2})$$

Where the total force is computed as:

$$F_i^{tot}(t) = F_{ext}^i - \sum_{j \in N_i} F_{ij} - c \frac{x_i(t) - x_i(t - dt)}{dt} \quad (\text{Equation 3})$$

In CUDA describe the forces for the individual points of the mass spring as locations inside texture memory by creating a 2D memory location of the size of the cloth. Inside the kernel you will implement the following pseudo code

```
// Gather force
for all neighboring mass points j do
    calculate spring force  $F_{ij}$ 
    add  $F_{ij}$  to total force  $F_i$ 
end for
```

```
update vertex position  $x_i$ 
```

This is a 3 pass algorithm.

1. Calculation and accumulation of spring forces at mass points
2. Time integration of mass points
3. Update of point positions

Equation 1 is calculated in the first pass and its results are outputted to a texture.

Equations 2 and 3 are calculated in the second pass and its result is output to the vertex buffer.

In the third pass the vertex buffer is rendered to the screen.

In this approach a surface point needs to maintain its current and last position for time integration, which will be maintained in a texture, its mass as well as a reference to all adjacent points including spring stiffness and rest length. In your implementation all points will have the same mass, spring stiffness and rest length and can therefore be placed in constant memory. We will also describe our cloth as all having the same number of incident edges except for the borders. Make sure to choose a small enough delta time that the simulation remains stable. In order for your simulation to become active you will need to apply a force to the cloth. To do so, add either mouse or keyboard functionality to pull or push the cloth from a single end point. This is done by adding a force to a corner point of the cloth. This force will be propagated through the rest of the cloth by your simulation.

The results of your simulation are the vertices of the cloth. Transfer this result to OpenGL using a vertex buffer and display your results. Play around with the various constants to see how they affect your cloth simulation.

Problem developed from Mass-Spring Systems on the GPU., Joachim Georgii, Westermann, Rudiger, 7 July 2005

**Problem 3 Cloth interactions** (15 points)

Update problem 1 to have the cloth interact with a simple sphere. Have the cloth's initial position be above the sphere. Take into account two forces. Force 1 is gravity pushing down onto the cloth. Force 2 is the sphere pushing back on the cloth when the cloth hits it. You can calculate if the cloth intersects the sphere by viewing the sphere as a point with a radius.

Show the cloth interacting with the sphere in your OpenGL display.

**Problem 4: CUDA SCAN** (20 points)

**Read:**

Read the white paper (Slides) on Parallel Reduction by Mark Harris found at the link below:  
[http://developer.download.nvidia.com/compute/cuda/1\\_1/Website/projects/reduction/doc/reduction.pdf](http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf)

Describe how you would implement the following code using Parallel Reduction (SUM) and/or Parallel Prefix (SCAN).

1.  
Find the min of an array of numbers

2.  
`r = C;`  
`for(i = 0; i < n; i++) {`  
    `r /= X[i];`  
`}`

Where  
C is a constant,  
X is an array of constants

3.  
`for(i = 1; i < n; i++) {`  
    `r[i] = A*r[i-1] + X[i]`  
`}`

Where  
A is a constant,  
r is an array result  
X is an array of constants

## Problem 5: K-Means (20 points)

Implement the K-means for 3 dimensions in CUDA

K-means, as described in class, is used to place a set of numbers into bins. In your implementation you will use the colors of an image as the numbers and you will create 16 bins to sort the colors into. The initial bin values can be random numbers, although in a real scenario the bins would be chosen from numbers that exist in the image.

To Implement:

Look at the example SimpleTexture for an example of loading PGM images and using cudaArrays in CUDA. Based on the example, write an image loader that will load a PGM image into a cudaArray. This will be the input for your K-means algorithm.

Initialize the bins for your algorithm (using cudaMalloc). The bins will be held in their own global memory array. Each bin value holds an RGB value so they are float3s. In your K-means algorithm there will be 16 bins.

The algorithm performs multiple iterations of a two pass algorithm. In the first pass, data points are loaded from the image and are checked against a bin. You must use shared memory to hold the bin values in this pass and sync threads when appropriate. In K-means, each value of the data is being tested against each bin value to find the minimum distance between the data point and the bin value. To determine the distance between the two points use the following equation:

$$\text{Dist\_sqrd} = (d.r-b.r)^2 + (d.g-b.g)^2 + (d.b-b.b)^2$$

where d is the data point, and b is the bin value. (the r,g,b are the dimension of the value).

The bin closest to each point is saved in a bin index array which is the same size as our image.

In the second pass we would like to sum the image points associate with a bin to get the new bin's value. To do this, for each image point we lookup the associated bin value that we placed in the bin index array during the previous pass. Any data points associated with the bin we are currently trying to sum we place the image point's value into a new array. Any points not of that bin value, we place a 0. We then perform parallel reduction on the array to get the new bin value. The image points are float3s if we have the array we used to sum made a float4 and we place a 1 in that last vector value for any image points that are of that bin, that fourth vector value will provide the total number of image points that are associated with that bin. This pass is performed once for each bin value. The algorithm is then rerun for the new bin values.

(Note: You must implement your own version of parallel reduction. You can look at how others have done it, but what you hand in must be your own code. I know where the versions are online, the point here is to teach you something very important on the GPU not be an exercise in googling.)

Display the result of your algorithm in an OpenGL window.

Time your code using the timing methods included in the CUDA SDK. To compare results against your classmates, the timer should start BEFORE you transfer data from main memory to the GPU and end once the algorithm completes. Perform a total of 5 iterations before displaying in the OpenGL window.

The fastest implementation will get an additional 5 points extra credit for fun. Use some of the speedups discussed in class to improve your results.