CIS 565: Assignment 4: CUDA and GPU Computing
Spring 2011
Due Friday, 03/04, using Blackboard by 11:59pm

This assignment covers CUDA programming and concepts of parallel and stream programming.

The assignment includes both programming and written components, make sure to complete them all. Include a README.txt file with your name and username, answers to the written questions, any extra credit you completed, and any implementation or platform notes.

To submit, upload YOUR_LAST_NAME.zip using the Blackboard submission link. YOUR_LAST_NAME.zip should include the GLSL_Samples and your readme file. That is, your submission should look this:

+ YOUR_LAST_NAME.zip
  - README.txt
  + Cuda_Samples
    ... Same setup as before

Part 1: Questions (35 points)

Include answers to the following questions in your readme file.

1. Parallel Reduction (10 points)

   a) Write out each pass, including the final answer, in a parallel reduction using 2x2 blocks to find the sum of the following matrix:

   \[
   \begin{bmatrix}
   0 & 3 & 4 & 6 \\
   1 & 2 & 2 & 8 \\
   3 & 3 & 2 & 1 \\
   4 & 1 & 5 & 6 \\
   \end{bmatrix}
   \]

   b) Assuming this runs on the GPU until a 1x1 block is reached, how many passes are required?

   c) How many total texture reads are required?

   d) How many additions are performed?

   e) What is the arithmetic intensity?

2. Scan (10 points)
a) Compute the inclusive and exclusive scan using addition for the following array:

\[3 2 0 1 6 5 4 2]\n
b) Show each of the three passes for a naive parallel scan using the above array as input.

c) How many total memory are required?

d) How many additions are performed?

e) What is the arithmetic intensity?

3. **Stream Compaction** (5 points)

a) Given an array of floating point values, explain how to use exclusive scan to create a new array containing just the positive values in the same order as the input array. You can use exclusive scan as a primitive; you do not need to explain its implementation.

b) Provide a non-trivial example of your solution to (a) using an input array with eight elements.

4. **GPU Architecture** (10 points)

a) Why do GPUs lack the large caches found on-chip in CPUs?

b) What is a divergent branch? How does the GPU handle it?

c) Explain how a shader/kernel’s working set, e.g., number of registers, affects the GPUs ability to hide memory latencies.

**Programming Component:**

The programming components of this assignment are longer and less structured than previous assignments. You will have to perform more large-scale code decisions, transfer and initialization of data, etc.

From previous experience, the cloth component is more straightforward but harder to get perfect. The K-Means component is harder to conceptually grasp but simpler to implement. If you have trouble getting your platform set up, check the installation notes at the end of the writeup.
**Part 2: Cloth** (30 points)

**Problem 1 (Programming): Cloth on the GPU using CUDA** (20 points)

Work off of the ‘cloth’ project in the ‘Cuda_Samples’ solution. Most of your code should be in cloth_kernel.cu. Use ‘1’, ‘2’, and ‘3’ to change the rendering mode of the cloth.

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:

\[ F_{ij} = F(x_i, x_j) = D_{ij} \times (|l_{ij}| - |R_{ij}|) \times \text{normalize}(l_{ij}) \]

- \(D_{ij}\) is the stiffness of the spring connecting points \(x_i\) and \(x_j\)
- \(l_{ij}\) is the displacement between \(x_i\) and \(x_j\)
- \(|R_{ij}|\) is the rest length of the spring in its initial configuration.

Point positions are updated using the Lagrangian law of motion:

\[ F_{\text{ext}}(i) = m_i \times x_i'' + c \times x_i' + \text{sum}_j(F_{ij}(x_i, x_j)) \]  \hspace{1cm} (Equation 1)

- \(m_i\) is the mass of point \(i\)
- \(c\) is a damping constant
- \(x_i''\) is acceleration (due to gravity, etc.)
- \(x_i'\) is velocity

The sum ranges over adjacent points.

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

\[ x_i(t + dt) = (F_i(t)/m_i) \times (dt^2) + 2 \times x_i(t) - x_i(t-dt) \]  \hspace{1cm} (Equation 2)

Where the total force is computed as:

\[ F_i(t) = m_i \times x_i'' + \text{sum}_j(F_{ij}(x_i, x_j)) - c \times (x_i(t) - x_i(t-dt))/dt \]  \hspace{1cm} (Equation 3)

This equation is the force calculation from before with velocity replaced by an approximation.
In CUDA describe the forces for the individual points of the mass spring as locations inside device memory by creating a 2D memory location of the size of the cloth. Inside the kernel you will implement the following pseudo code

```c
// Gather force
for all neighboring mass points j do
    calculate spring force Fij
    add Fij to total force Fi
end for

update vertex position xi
```

This is a 2 pass algorithm.
1. Calculation and accumulation of spring forces at mass points
2. Time integration of mass points

Equation 3 is calculated in the first pass and its results are outputted to device memory. Equations 2 is calculated in the second pass and its result is output to the vertex buffer to be drawn. Note! You cannot update mass positions in the first pass because other points force calculations may depend on them! A global synchronization is necessary.

In this approach a mass point needs to maintain its current and last position for time integration (which will be maintained in device memory) its mass as well as a reference to all adjacent points including spring stiffness and rest length. In your implementation all points with have the same mass, spring stiffness and rest length and can therefore be placed in constant memory. We will also describe our cloth points 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 be 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 an acceleration 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 2 (Programming): Cloth interactions** (10 points)

Update problem 1 to have the cloth interact with a simple sphere. We have provided a command line switch to the cloth project to switch between this mode and Problem 1. --part1 should run the first problem, --part2 should run this code.

Have the cloth’s initial position be above the sphere. Take into account two external
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. You will have to add sphere rendering. Note: getting this simulation to be stable can be quite tricky. It is ok if it eventually blows up, but it should rest stably on the sphere for a good time.

An implementation from last year is pictured here: Sphere-Cloth Video; however, they used a different renderer so your cloth may appear different.

Extra Credit: Advanced Interactions (10 pts):

Add a --part3 to your cloth simulation. Add an additional torus primitive to your cloth sim. Setup a scene where your cloth is resting on a torus, and drop a sphere down through the hole in the torus, tugging the cloth through with it. Your cloth should remain relatively stable through this simulation.

Part 3: K-Means (35 pts)

Problem 1 (Programming): Reduce (15 points)

For the first part of this project we will implement reduce as a subroutine. Our special case will sum a float4 array into a single float4. You must implement your own hierarchical parallel reduction ala Mark Harris Presentation, and your implementation should be careful to synchronize properly. Note that you may have to allocate temporary buffer space, and you will have to perform multiple iterations to accommodate data sets of arbitrary sizes. Be careful with errors relating to non-power-of-2 arrays.

You should start from the ‘reduce’ project in the ‘Cuda_Samples’ solution. Your work should mostly take place in reduce_kernel.cu. There is a small provided test suite to evaluate your code versus a baseline CPU implementation; you may want to add additional corner cases.

Extra Credit: Scan (15 pts):

Implement a parallel scan routine, and baseline CPU implementation, and provide a small test suite.

Problem 2 (Programming): K-Means Image Clustering (20 pts)

K-Means is a generic algorithm to cluster data points into bins. We will be using it to cluster pixels in an image into clusters of colors. It computes both the colors of the clusters, and which pixels belong in each cluster. You should start from the ‘kmeans’ project in ‘Cuda_Samples’ solution. Most of your work should take place in the ‘kmeans_kernel.cu’ file. Currently, there is a simple kernel which just inverts the image color.

The K-Means algorithm performs multiple iterations of a two-pass algorithm. In the first
pass, pixels are binned into clusters based on their similarity to that cluster’s color. In the second pass, cluster centers are re-estimated by averaging their member pixels. These two passes are repeated until convergence. The name of K-means refers to the fact we are computing the best $K$ centers, or means, of the input data, where $K$ is the number of clusters.

We will initialize the colors of the clusters to random float3’s. We will use a fixed $K$ of 16 bins. You should use a small fixed number of iterations (5 is nice). We will use Euclidean distance as our similarity measure between colors.

In the first pass, we need to assign each pixel to a bin based on similarity. You should allocate a 2d array of integers, one integer per-pixel, and a small array of float3’s to represent the bin centers. Compute the similarity of each pixel to all the bin centers and write out the index of the bin to which the pixel is most similar. This has assigned each pixel to a bin for that iteration.

In the second pass, we need to average the colors of the pixels associated with each bin. This ‘re-centers’ the bin to better represent its constituents. There are several ways to do this, but one simple way is to use the reduce function from Problem 1. We will perform 16 iterations over the data, 1 to find the average of each bin.

In each iteration we will first transform the pixels to filter out pixels not in the current bin, and then sum the value of the remaining pixels. For each iteration, we should have a current bin we are working on. Create a 2D buffer of float4s the size of the original image and write out $<0,0,0,0>$ to the buffer if a pixel is not a member of the current bin and $<r,g,b,1>$ if it is. We can look at the indices we computed in the first pass to determine which bin a pixel belongs to.

To average the pixels, we can now sum all the pixels in this (relatively sparse) buffer. (Note, this sums all the entries in the buffer, including the $<0,0,0,0>$ ones. We could perform stream compaction to avoid this cost, but that requires doing a scan operation anyways, which is more expensive). To average the colors of the pixels in a bin, we need not only the sum of their color values but the number of pixels assigned to each bin. We can get this by looking at the 4th component of the sum, since each pixel that was in the bin wrote out $<r,g,b,1>$. Thus, $\text{color\_avg} = \text{sum}\_rgb / \text{sum}\_a$.

Once this is performed for each bin, we have a new ‘center’ for all the bins, and we can repeat the algorithm starting from the first pass.

For the final display after a number of iterations, assign to each pixel the color of its corresponding bin. This should give you a good representation of the image with a limited color palette. Display this in the OpenGL window.

Platform and Installation Notes:

To perform the programming components of this assignment, you need a CUDA-capable Nvidia graphics card, a Windows machine with Visual Studio 2008, OpenGL 3.3, CUDA toolkit 3.2, and up-to-date CUDA drivers. You may also wish to install the CUDA SDK to reference example code and verify that your CUDA installation works.

If you do not have access to such a machine, you may use the computers in the SIG lab or Moore labs. Additionally, you may modify the project setup to make it run on any configuration you want as long as the bulk of the assignment is done via GPU programming; for
example, if you really wish to do this assignment in OpenCL on an arbitrary Linux distribution, you may, but it will probably be a significant amount of work to port. In that case, you will show that your code works via a live demo. If you wish to distribute a ported copy of the base setup to other students (for example, a VS2010 version, Xcode version, or OpenCL version) you may; go ahead and post on the discussion board.

On a similar note, CUDA 3.2 projects do not port cleanly to Visual Studio 2010 (see online discussion). It is however possible to recreate the projects manually in Visual Studio 2010 and import all the files by hand.