15-418 Final Project Writeup

Summary

We built a framework that facilitates running computations across multiple GPUs and displaying results in a web browser. We then created three demos that utilized this framework and ran them on the GTX 480s in the Gates clusters.

Background

Framework

We decided to create this framework because we saw lots of uses and advantages for this framework. Firstly, we believe that it makes application development much easier because our framework is platform-agnostic, which means that once a program is built to run on our framework, it will run on any device that has a web browser. The front-end for each application that runs on our framework is written with HTML5 technologies. This makes it easier to develop GUIs for any kind of application since web technologies were built from the ground up for interaction and several libraries and extensions are available. This also allows applications to display output in a wide variety of formats, ranging from text to images to WebGL.

For the framework, we encountered the following difficulties:

1) Inter-GPU communication
   We had to facilitate communication between multiple GPUs that were each located on different machines.

2) Passing raw data between GPUs
   JavaScript’s arrays are not space-efficient (JSON objects are transferred over network as ASCII, and this includes numbers). This meant that we had to send binary data between GPUs to save space. However, sending binary data to the browser is difficult since WebSockets currently do no have widespread binary support. This meant we had to convert binary data sent to the browser, e.g. images or an array of floats, to Base64 before sending it over. This increases the size of the data by 33% and introduces overhead for converting to and from Base64.

3) Writing CUDA bindings
   There were several difficulties involved: learning about the internals of Chrome’s JavaScript engine, getting JavaScript to hold references to CUDA data structures,
transferring data to and from GPU memory, etc. We had to use the CUDA driver API instead of the CUDA runtime API, which was difficult because it is a very low-level interface to running code on the GPU.

4) **Network issues**
Depending on the display format and output being shown, we expected to have issues with network bandwidth. It would be difficult to send images at 30fps across the network, so we would have to come up with strategies for improving the speed, such as JPEG compression.

5) **Message passing**
This was challenging to implement because there are several connections that need to be managed between browsers, web servers, and GPU servers. We needed to set up message passing protocols and handle the transfer of large packets that TCP splits up into chunks.

*Demos*
We chose demos that showcased various aspects of our framework.

Mandelbrot
Mandelbrot, shows that our framework can display high FPS animations, and we also chose to emphasize user interactivity in this demo with a vibrant rendering of the Mandelbrot fractal.

Stars
Stars showcases more computationally intense loads and multi-user interaction. Users can interact with the simulation by clicking on the screen, which sends out a repulsion force that causes stars in the nearby area to move away. Multiple users can do this at the same time, and everyone’s session is updated in real-time. We believe that this demo is an example of how our framework can be used for scientific applications.

Parachute
Parachute demonstrates how our framework handles multi-GPU simulations and multi-user interactions. Every user is assigned a corner of the parachute, and each user can interact with the parachute by moving their assigned corner around. Everyone sees these interactions in real-time.

*Approach*

*Framework*
Our framework is composed of multiple components, each with its own purpose. This diagram shows the ways the different components interact.
Overview of the framework as a whole.

**Web Server – web_server.js**
The web server serves static pages over HTTP. Once the JavaScript on the static page has loaded, it connects to the browser via WebSockets and acts as an intermediary for communicating with the GPU servers. When a browser connects, it decides whether the browser is trying to join an existing session or if it should start a new one. If it needs to start a new session, then it schedules the job the least busy GPU servers. The web server is also responsible for acting as a GPU server manager so it keeps track of which servers are connected.

**GPU Server – gpu_server.js**
The GPU server does three things. It waits for messages to start or stop jobs from the web server. It also receives messages from the web browser via the web server. Additionally, it launches application modules, and it also facilitates message passing between GPU nodes.

**Message Passing**
The GPU servers and web server communicate with each other through TCP sockets. The GPU server exposes the following functions to each application that it is running:

1) Send – Takes in which node ID to send to and a message type
2) Receive – Waits until a message of a given type from a certain node ID is received

These functions will perform synchronous sends and receives, and the messages that are passed can include both JSON data (easy for structured data passing but not space efficient for arrays), and binary data (for passing data as raw bytes). Messages that are received but not yet consumed get put on a queue.
Communication within the framework.

The diagram above shows how the different components of the framework connect to each other. All the web browsers connect to the web server via HTTP and WebSockets. The web server connects to the GPU servers through TCP, and the GPU servers communicate with each other through TCP as well. The GPU servers then load the application modules and call the 4 functions that the modules must implement. Multiple GPU servers may be running the same application, and each GPU server may be running multiple applications.

**Job Scheduler**

The job scheduler will schedule new jobs such that the least busy GPUs will work on the job. For example, a job that requires n GPUs will use the n least busy GPUs. We can also run several different jobs simultaneously. The GPUs will automatically share jobs if there is more than one job scheduled to it.

**Multi-User Sessions**

We allow multiple browser sessions to interact with the same instance of an application. The session is identified by an id that can either be provided by the creator of the session, or a random string of 8 hexadecimal characters is used. We also keep track of which browsers are connected and which session they are connected to, and we need to route messages to and from the browser by which session they are connected to.

**Pipelining**

We used pipelining to make single-GPU jobs execute at high framerates. Our pipeline consists of three stages:

1) Compute a frame of data  
2) Compress the frame into JPEG  
3) Transfer JPEG to browser

These stages can be executed simultaneously, which means that three frames are being worked on at the same time. However, Stage 1 may execute far more quickly than stage 3. Instead of waiting for stage 3 to finish, we allow stage 1 to execute multiple times
before moving to stage 2 so that the pipeline won’t be blocked. For example, in the stars simulation, we are able to update the stars faster than we transfer to the client so we could use a smaller dt for a more accurate simulation.

**Node-CUDA**

This consists of CUDA bindings for Node.js, and allows you to run CUDA programs from JavaScript. It contains functions for the following:

1) Creating CUDA contexts
2) Retrieving device information
3) Loading CUDA modules (.cu files that were compiled into either .cubin or .ptx)
4) Allocating/deallocating memory
5) Copying memory to/from Node.js Buffer objects
   a. Buffer objects allow raw data to be stored efficiently. While this feature is not available in JavaScript natively, Node.js provides this feature. It is similar to allocating more memory using malloc.
6) Get references to global functions in CUDA modules
7) Execute the global functions
   a. Requires coping the arguments into a buffer and aligning them properly

For this, we started with Kashif Rasul’s code (https://github.com/kashif/node-cuda). His code could only create CUDA contexts and allocate/deallocate memory. It did not have the ability to launch kernels or copy memory. A lot of the code had to be rewritten and/or restructured. In the end, the changes we made were contributed back and kept under the MIT license.

**Demos**

We parallelized the computations across pixels, with each block being a 32x32 set of pixels.

When the user is interacting with the application, Mandelbrot sends back JPEG images that are smaller but lower quality. When the user stops interacting, Mandelbrot switches to lossless PNG images and uses 16x antialiasing.

**Stars**

We split the work into three stages:

1) Compute forces
   a. Parallelized across stars, each star on its own thread, 32 threads per block
2) Update positions of the stars (using the forces from 1)
   a. Parallelized across stars, each star on its own thread, 32 threads per block
3) Rendering the stars onto an image
   a. Parallelized across pixels, 32x32 set of pixels per block
Parachute
We split the parachute into n equally sized pieces, with n being the number of GPUs doing calculations for a given session. Each GPU’s work was divided into four steps.

1. Compute forces
2. Send forces on the right edge to slice on the right
3. Update positions
4. Send new positions of left edge to slice on the left

A visualization of the work being done on each GPU

The force computation and position update steps were parallelized across the points of the parachute mesh.

Afterwards, every other GPU sends points to node 0, and then node 0 sends all positions to the web client. Finally, the web client renders the points using WebGL (using Three.js).

Results
We succeeded in creating a framework that did what we had envisioned in the beginning, as well as demos that utilized this framework.

Node.js made it easy to tie together all the different technologies we used: WebSockets (using the Socket.io library), WebGL (using the Three.js library), CUDA bindings for Node.js (https://github.com/r2jitu/node-cuda), and PNG/JPEG compression libraries for Node.js.
Using JavaScript for the front-end and for the application modules also allowed us to write far fewer lines of code. Our framework also makes it extremely easy to display graphics and to create user interfaces, because interfaces can be made in HTML5 as opposed to SDL or other C/C++ libraries.

Our system is also very easily scalable; all that needs to be done is to run a web server on a head node. GPU servers can also be elastically added or removed, and through Node.js, it is easy to load modules and code.

In the end, we encountered problems that we did not anticipate in the beginning, the main one being destroying jobs. Destroying jobs is important because the GPU has limited memory and no swapping ability, so freeing everything that was allocated was necessary. This meant that if a server crashed, we needed to kill the session by stopping the job on all the GPUs it was running on. We also needed to free memory that JavaScript holds onto by removing all references to killed jobs.

We made a frame rate timer (frame_timer.js) and tried to put some work into security so that application modules wouldn’t be able to do anything malicious to the GPU server, but that was hard to guarantee.

If we were to continue our project further, we would work on improving multi-GPU communication and increasing the frame rate. Currently, our single-GPU implementation is well-pipelined and has high frame rates, but our multi-GPU implementation doesn’t. In fact, our multi-GPU communication is synchronous, so a lot of time is spent waiting and synchronizing with other GPUs.

References

Node-CUDA
<https://github.com/kashif/node-cuda>

Mandelbrot Set
<http://en.wikipedia.org/wiki/Mandelbrot_set>

Three.js
<https://github.com/mrdoob/three.js/>

List of Work Done

Framework (back-end) – Jitu Das
Demos (front-end) – Bertha Lam