CME 213, Introduction to parallel computing
Eric Darve
Spring 2024

Stanford University

Homework 2

Total number of points: 180.

In this programming assignment, you will use NVIDIA's Compute Unified Device Architecture (CUDA) language to implement a basic recurrence algorithm and element-wise kernels for the final project. In the process, you will learn how to write general-purpose GPU programming applications and consider some optimization techniques. You must turn in your own copy of the assignment as described below. Although you can collaborate with your classmates on the assignment, sharing solutions is strictly prohibited. If you have any queries regarding the task, kindly post them on the forum.

You can access an NVIDIA GPU, which is needed to complete this assignment, from the login node `cme213-login.stanford.edu`. You can connect to this node using the command `ssh your_sunet_id@cme213-login.stanford.edu`. Each time you login you must run the command `ml course/cme213/nvhpc/24.1`; alternatively, you can add this command to the file `~/.bashrc` to automatically load the required modules every time you login. Run `make` to compile your code, run `make clean` to clear the directory of existing object files and executables, and run `sbatch hw2.sh` to submit your job to the queue. The output will be in `slurm.sh.out`.

For all questions asking to comment on plots, make sure to describe the shape and different regions (such as increasing performance or asymptotic behavior) of the graph and explain why these patterns may emerge.

CUDA

"C for CUDA" is a programming language subset and extension of the C programming language and is commonly referenced as simply CUDA. Many languages support wrappers for CUDA, but in this class, we will develop in C for CUDA and compile with `nvcc`.

The programmer creates a general-purpose kernel to be run on a GPU, analogous to a function or method on a CPU. The compiler allows you to run C++ code on the CPU and the CUDA code on the device (GPU). Functions which run on the host are prefaced with `__host__` in the function declaration. Kernels run on the device are prefaced with `__global__`. Kernels that are run on the device and that are only called from the device are prefaced with `__device__`.

The first step you should take in any CUDA program is to move the data from the host memory to device memory. The function calls `cudaMalloc` and `cudaMemcpy` allocate and copy data, respectively. `cudaMalloc` will allocate a specified number of bytes in the device main memory and return a pointer to the memory block, similar to malloc in C. You should not try to dereference a pointer allocated with `cudaMalloc` from a host function.

The second step is to use `cudaMemcpy` from the CUDA API to transfer a block of memory from the host to the device. You can also use this function to copy memory from the device to the host. It takes four parameters, a pointer to the device memory, a pointer to the host memory, a size, and the direction to move data (`cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost`). We have already provided the code to copy the string from the host memory to the device memory space, and to copy it back after calling your shift kernel.

Kernels are launched in CUDA using the syntax `kernelName<<<...>>>(...)`. The arguments inside of the chevrons (`<<<blocks, threads>>>`) specify the number of thread blocks and thread per block to be launched for the kernel. The arguments to the kernel are passed by value like in normal C/C++ functions.

There are some read-only variables that all threads running on the device possess. The three most valuable to you for this assignment are `blockIdx`, `blockDim`, and `threadIdx`. Each of these variables contains fields x, y, and z. `blockIdx` contains the x, y, and z coordinates of the thread block where this thread is located. `blockDim` contains the dimensions of thread block where the thread resides. `threadIdx` contains the indices of this thread within the thread block.

We encourage you to consult the development materials available from NVIDIA, particularly the CUDA Programming Guide and the Best Practices Guide available at
`http://docs.nvidia.com/cuda/index.html`

Unit test fixtures

Sometimes when testing your code, you may notice that you are doing very similar operations to set up certain tests. In such cases, GoogleTest provides test fixtures that enable you to reuse the same object configuration for multiple tests. For the recurrence problem below, we will employ this approach to streamline our testing procedures. Additional information on test fixtures can be found in the GoogleTest documentation:
`http://google.github.io/googletest/primer.html#same-data-multiple-tests`

Fixture tests must use the macro `TEST_F`. The first argument of the macro must be the name of the test fixture class, `RecurrenceTestFixture`. Using test fixtures, the following sequence of code is executed:

1. A `RecurrenceTestFixture` object is created.
2. The first test
   `TEST_F(RecurrenceTestFixture, GPUAllocationTest_1)`
   runs. This test is able to access objects and subroutines in the test fixture object `RecurrenceTestFixture`.
3. Once the test completes, the test fixture object is destructed.

This sequence is repeated for all subsequent fixture tests `TEST_F`.

Problem 1 Recurrence

The purpose of this problem is to give you experience writing your first simple CUDA program. This program will help us examine how various factors can affect the achieved performance.

Inspired by the Mandelbrot Set, we want to perform the following recurrence for several values of c:

$$z_{n+1} = z_n^2 + c.$$

z is in general a complex number but for simplicity we will use `floats` in this homework. For each value of c, you can study the sequence z_n. If z_n does not diverge (starting from $z_0 = 0$) then the point c belongs to the Mandelbrot set. In Figure 1, the coordinates of each pixel correspond to the real and imaginary parts of c. The color of a pixel is determined by computing the smallest iteration n for which $|z_n| > 2$. One can prove that if $|z_n| > 2$ for some n then $|z_n| \to \infty$ as $n \to \infty$. The recurrence is done for a maximum of `num_iter` iterations, and the values of the c's are set in `initialize_array()` in `main_q1.cu`.

Code

You should be able to take the files we give you and type `make main_q1` to build the executable. The executable will run, but since the CUDA code hasn't been written yet (that's your job), it will report errors

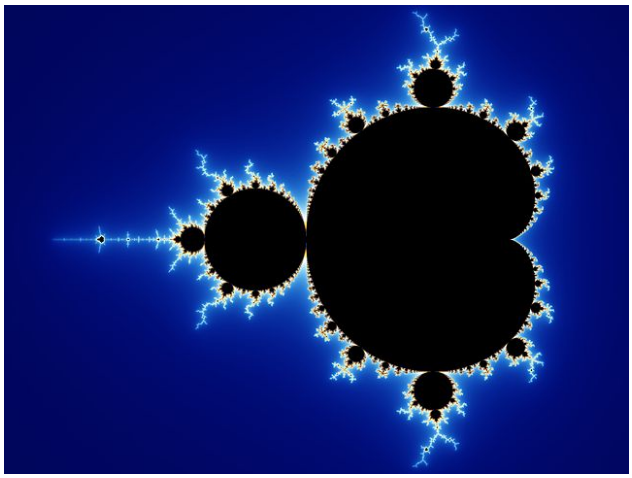

Figure 1: Mandelbrot Set. Source: Wikipedia. The black points in the image belong to the Mandelbrot set. The sequence z_n does not diverge for the corresponding c. Although not obvious, the Mandelbrot set is a connected set.

and quit. All locations where you need to write code are noted by a `TODO` in the comments. For this problem we provide the following starter code (* means you should *not* modify the file):

- `main_q1.cu`—This is the main file. We have already written most of the code for this assignment so you can concentrate on the CUDA code. We take care of computing the host solution and checking your results against the host reference. There is also code to generate the tables you will need to do the benchmarking questions. You will do questions 1 and 2 in this file.
- `recurrence.cuh`—This file already contains the necessary function headers—do not change these. You should fill in the body of the kernel and launch the kernel from `doGPURecurrence()`.
- `*test_recurrence.h`—This file contains the functions we will use to check your output. Do not modify this file.
- `*Makefile`—`make run1` will build and run the binary. `make main_q1` will build the binary. `make clean` will remove the executables. You should be able to build and run the program when you first download it, however only the host code will run.
- `hw2.sh`—This script is used to submit jobs to the queue. You need to comment out the other lines in the file if you only want to run `./main_q1`.

Question 1.1

10 points. Allocate GPU memory for the input and output arrays for the recurrence within the test fixture constructor. Free this GPU memory at the end in the destructor. This code (approx. 4 lines) should be in `main()` in `main_q1.cu`.

Question 1.2

10 points. Implement `initialize_array()`, the function that initializes an array of a given size. The values are random floats between -1 and 1. These will be the constants c in the recurrence. This code should be in `main_q1.cu`.

Question 1.3

20 points. Implement the recurrence kernel and launch it. These should be implemented in `recurrence()` and `doGPURecurrence()` respectively in `recurrence.cuh`. You can see a CPU implementation of the recurrence in `host_recurrence()` and a sample launch of the kernel in `main()`, both in `main_q1.cu`. Add the output of the code (it should be 2 tables) to your PDF submission. The whole run may take 10 minutes.

Question 1.4

10 points. Run the same setup but with the number of blocks to be 72, the number of iterations to be 40,000, and the array size (number of constants we test) to be 1,000,000 (this code has already been written for you). Vary the number of threads per block as 32, 64, 96, …, 1024. Take the table that is generated and plot the performance in TFlops/sec vs. the number of threads. Comment on and explain the shape of the graph.

Question 1.5

10 points. Run the same setup with the number of threads per block to be 256, the number of blocks to be 576, and the array size (number of constants we test) to be 1,000,000 (this code has already been written for you). Vary the number of iterations as in the code. Take the table that is generated and plot the performance in TFlops/sec vs. the number of iterations. Comment on and explain the shape of the graph.

Problem 2 Benchmarking with Strided Memory Access

For this problem, we will benchmark our device by performing strided memory accesses. The file `benchmark.cuh` performs a benchmark using two very long input arrays x and y, as well as an output array z, by computing $z[i] = x[i] + y[i]$ at stride lengths between 1 and 32. That is, $z[i] = x[i] + y[i]$ for $i \in \{0, 1, 2, 3, \dots\}, i \in \{0, 2, 4, 6, \dots\}, \dots, i \in \{0, 32, 64, 96, \dots\}$.

For this problem, we provide the following starter code (* means you should not modify the file):

1. `*main_q2.cu`—sets up the CUDA runtime and launches your benchmarking kernel with stride lengths in 1…32

2. `benchmark.cuh`—this is the file you will need to modify and submit. Do not change the function headers but fill in the bodies and follow the hints/requirements in the comments.

3. `*Makefile`
   `$ make main_q2`
   will build the benchmarking binary. You can run it using `sbatch hw2.sh` (see item 4 in this list) or `srun -p gpu-turing -G 1 ./main_q2`.
   `$ make clean`
   will remove the executables. You should be able to build and run the program when you first download it. However, your results will be incorrect as your kernel won't be performing any memory accesses.

4. `hw2.sh`—This script is used to submit jobs to the queue. You need to comment out the other lines in the file if you only want to run `./main_q2`.

Question 2.1

5 points. Perform the strided memory access in benchmark.cuh. Then, in the terminal, run `make benchmark`. In your writeup, display the results on a semilogy plot of throughput in GB/s as a function of stride length. Do not comment on the plot under this part.

Question 2.2

5 points. Comment on and explain the shape of the graph. Why do we observe the trend that we do as the stride length increases?

Neural Networks on CUDA

In the final project, you will use both CUDA and MPI to implement a parallel training process for a two-layer neural network which can identify digits from handwritten images (a specific case of image classification problem). Neural networks are widely used in machine learning problems, specifically in the domains of image processing, computer vision, and natural language processing. There is a flurry of research projects on deep learning, which uses more advanced variants of the simpler neural network (NN) we cover here. Therefore, being able to train large neural networks efficiently is important and is the goal of this project.

Data and Notation

We will be using the MNIST [1] dataset, which consists of greyscale 28×28 pixel images of handwritten digits from 0 to 9. Some examples of this dataset are shown in Figure 2.

Figure 2: Examples of MNIST digits

The dataset is divided into two parts: 60,000 images in the *training* set and 10,000 images in the *test* set. We will use the training set to optimize the parameters of our neural network (described below), and we will use the unseen *test* set to measure the performance of the trained network. We denote the i^{th} image sample in the training set as $(x^{(i)}, y^{(i)})$ where $x^{(i)}$ denotes the image and $y^{(i)}$ denotes the class label (the digit shown in the image).

Neural Networks

Neurons

To describe neural networks, we begin by describing the simplest neural network, which comprises a single "neuron. " Figure 3 illustrates a single neuron.

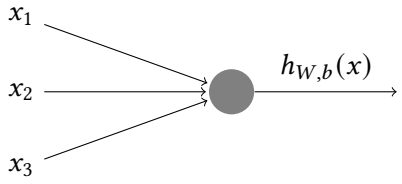

Figure 3: A single neuron.

This neuron is a computational unit that takes as input (x_1, x_2, x_3) and outputs

$$h_{W,b}(x) = f(Wx + b) = f\Big(\sum_{i=1}^{3} W_i x_i + b\Big)$$

where $f : \mathbb{R} \to \mathbb{R}$ is a non-linear 'activation' function, W is the weights of the neuron, and b is the bias of the neuron. The pair $p = (W, b)$ is referred to as the parameters of the neuron.

For this project, we set $f(\cdot)$ to be the sigmoid function

$$f(z) = \sigma(z) = \frac{1}{1 + \exp(-z)}.$$

The derivative of the sigmoid function with respect to its input is

$$\frac{\partial \sigma(x)}{\partial x} = -\frac{1}{(1 + \exp(-x))^2} \frac{\partial \exp(-x)}{\partial x} = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \sigma(x)(1 - \sigma(x));$$

we will use this fact repeatedly in the following sections. Other common activation functions include $f(z) = \tanh(z)$ and the rectified linear unit (ReLU) $f(z) = \max(0, z)$. These are illustrated in Figure 4.

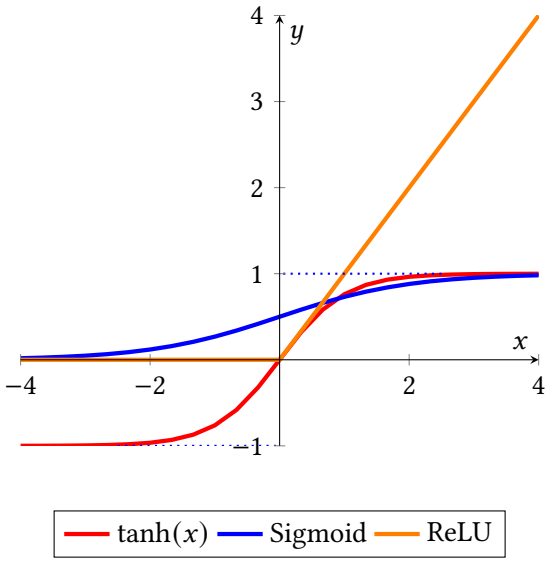

Figure 4: Examples of three activation functions: $\tanh(x)$, $1/(1 + \exp(-x))$ (sigmoid), and the rectified linear unit (ReLU).

A single neuron can be trained to perform the task of binary classification, Consider the example of cancer detection, where the task is to classify a tumor as benign or malignant. We can provide as input $x = ($ size of the tumor, location of tumor, length of time the tumor has existed), and if the label is

$$y = \begin{cases} 1 & \text{the tumor is malignant} \\ 0 & \text{the tumor is benign} \end{cases}$$

we can say that the neuron predicts that the tumor is malignant if and only if $f(Wx + b) > 0.5$.

Since the value of $f(Wx + b)$ depends on the sign of $Wx + b$, the neuron effectively partitions the input space $\mathbb{R}^3$ using a 2-dimensional hyperplane. On one side of the hyperplane $f(Wx + b) > 0.5$, and on the other $f(Wx + b) < 0.5$. Through an optimization process referred to as 'training', we want to find values of the parameters W and b such that the hyperplane represented by the neuron is as close as possible to the 'true' hyperplane.

More generally, we want to find values of the parameters W and b such that the network's predictions are 'good' on an unseen test set, since this would imply that our choice of model (here, a neuron with

certain values of W and b) is close to the 'true' model. It is insufficient to observe good predictions on the training set, since sufficiently complex networks can be trained to make perfect predictions on the training set but they perform much worse on unseen data, implying that the trained model is not close to the 'true' model.

In the project, we want to perform the task of multi-class classification, rather than binary classification. Instead of a simple true/false output, we need to decide which digit from 0 to 9 is shown in the input image.

Fully-connected feedforward neural network

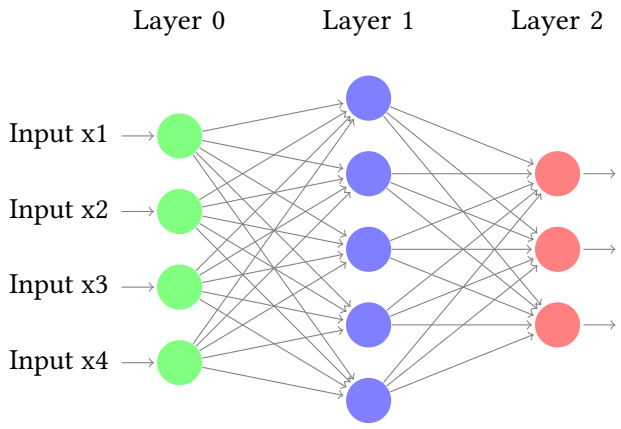

Figure 5: Fully Connected Feedforward Neural Network with 2 layers

Figure 5 shows a fully-connected feedforward neural network with 1 input layer, 1 hidden layer, and 1 output layer. We call such a network a two-layer neural network (ignoring the input layer as it is trivially present). Let us denote the input as $x \in \mathbb{R}^{1\times d}$, the number of neurons in layer i as H_i, and the parameters of layer i as $(W^{(i)}, b^{(i)})$.

In our problem, we are trying to determine the digit associated with each image. We will call this digit the "label" associated with the image. The total number of labels is denoted C. In our case $C = 10$, since we are trying to determine digits 0 to 9.

Recall that the parameters of a single neuron are $W \in \mathbb{R}^{1\times d}$ and $b \in \mathbb{R}$, i.e., W is a vector of the same dimensionality as the input and the bias is simply a scalar. Therefore, we can represent the parameters of layer i of the neural network in matrix form as $W^{(i)} \in \mathbb{R}^{H_i \times H_{i-1}}$ and $b^{(i)} \in \mathbb{R}^{H_i \times 1}$. Similarly, if we had N input vectors, we denote the input collectively as $X \in \mathbb{R}^{N\times d}$.

In Figure 5, $d = 4$, $H_1 = 5$, $H_2 = 3$, $W^{(1)} \in \mathbb{R}^{5\times 4}$, $b^{(1)} \in \mathbb{R}^{5\times 1}$, $W^{(2)} \in \mathbb{R}^{3\times 5}$, $b^{(2)} \in \mathbb{R}^{3\times 1}$.

The last layer is special. This is the output of our network. In the project, we have $C = 10$ output nodes. Each node represents a possible digit. We will see later on how the output vector $\hat{y}$ can be interpreted to determine the digit that is predicted by the network for an input image.

Feed-forward

The nice thing about neural networks is that they are highly modular. Layer L_i does not need to know whether its input is the input layer itself or the output of L_{i-1}. L_i computes its activations as $a^{(i)} = f^{(i)}(Wa^{(i-1)} + b^{(i)})$, with $a^{(0)} = x$, where $f^{(i)}$ is the non-linearity used by L_i (sigmoid, by default). Feed-forward is the process of computing the activations of all neurons in the network layer-by-layer, from $i = 1$ to $i = 2$ in our case.

Let us perform the feed-forward for the network in Figure 5:

$$z^{(1)} = W^{(1)}x + b^{(1)}$$
$$a^{(1)} = \sigma(z^{(1)})$$
$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$$
$$\hat{y} = a^{(2)} = \text{softmax}(z^{(2)})$$

$\hat{y}$ is the output of the network. Note that we have represented the linear transformation of the L_i by $z^{(i)}$. This will help us in the following sections. The softmax function is defined by:

$$\text{softmax}(z^{(2)})_j \stackrel{\text{def}}{=} P(\text{label} = j|x) \stackrel{\text{def}}{=} \frac{\exp(z_j^{(2)})}{\sum_{i=1}^{C} \exp(z_i^{(2)})}$$

This equation is saying that the probability that the input has label j (i.e., in our case, the digit j is handwritten in the input image) is given by $\text{softmax}(z^{(2)})_j$. Therefore, our predicted label for the input x is given by:

$$\text{label} = \text{argmax}(\hat{y})$$

This is basically the digit the network believes is written in the input image.

Training

Recall that our objective is to learn the parameters of the neural network such that it gets the best accuracy on the *test* data. Let y be the one-hot vector denoting the class of the input, i.e., $y_c = 1$ if c is the correct label, 0, otherwise. We want $P(\text{label} = c|x)$ to be the highest (e.g., close to 1).

Without going into the mathematical details, we will use the following general expression to determine the error of our neural network. This expression turns out to be the most convenient for our purpose:

$$\text{CE}(y, \hat{y}) = -\sum_{i=0}^{C-1} y_i \log(\hat{y}_i)$$

CE stands for cross-entropy. Since y is a one-hot vector, this simplifies to

$$\text{CE}(y, \hat{y}) = -\log(\hat{y}_c)$$

We can observe that *CE* is 0 when we have the optimal answer $\hat{y}_c = 1$. Similarly, *CE* is maximal ($+\infty$) when $\hat{y}_c$ is 0. This corresponds to a neural network that is "sure" that the digit is *not* c (maximally wrong).

The total cost for N input data points (such that the cross-entropy of the i^{th} training vector is denoted as $CE^{(i)}$) is:

$$\text{cost} = J(W, b; x, y) = \frac{1}{N}\sum_{i=1}^{N} CE^{(i)}(y, \hat{y})$$

The above cost measures the error, i.e. our "dissatisfaction", with the output of the network. The more certain the network is about the correct label (high $P(y = c|x)$), the lower our cost will be.

Clearly, we should choose the parameters that minimize this cost. This is an optimization problem, and may be solved using the method of Stochastic Gradient Descent (described below).

Our neural network applies a non-linear function to the input because of the sigmoid and softmax functions. When optimizing the neural network, we often add a penalization term for the magnitude of W

in order to control the non-linearity of the network. If we make W smaller, the network becomes 'more linear' since $Wx \approx \sigma(Wx)$ when $Wx \approx 0$. Despite the possibility of making W too small and the fact that there is no rigorous justification for this penalization, it is found to work well in practice. With the penalization term, the cost function becomes

$$\boxed{J(W, b; x, y) = \frac{1}{N} \sum_{i=1}^{N} CE^{(i)}(y, \hat{y}) + \frac{\lambda}{2} \|W\|_2^2} \tag{1}$$

where $\|W\|_2^2$ is the sum of the l^2-norm of all the weights W of the network, and λ is a hyperparameter that needs to be tuned for best performance. In our implementation, only the weights W are penalized, not the biases b.

Gradient Descent

Gradient Descent is an iterative algorithm for finding local minima of a function. For our case,

$$\boxed{p \leftarrow p - \alpha \nabla_p J} \tag{2}$$

where α is the learning rate that controls how large the descent step is. $\nabla_p J$ is the gradient of J with respect to the network parameters p.

In practice, we often do not compute $J = \sum_{i=1}^{N} \mathrm{CE}^{(i)}$ since this requires computing $\mathrm{CE}^{(i)}$ for all $i = 1, \ldots, N$. Instead, we divide the input into 'mini-batches' containing M images and process one mini-batch at a time until all images are processed. For each mini-batch we calculate $J_{\mathrm{mb}} = \sum_{i=k}^{k+M} \mathrm{CE}^{(i)}$ (where $x^{(k)}$ is the first image in the mini-batch), and update the network parameters p according to the update rule

$$p \leftarrow p - \alpha \nabla_p J_{\mathrm{mb}}. \tag{3}$$

This algorithm is also called Mini-batch Gradient Descent. See Listing 6 for the pseudo-code, where an 'epoch' refers to a single iteration over all N images and corresponds to $\lceil M/N \rceil$ updates to the parameters p. This approach usually leads to faster convergence than Batch Gradient Descent (or simply Gradient Descent) since we update the network coefficients more than once per epoch.

```
epoch = 0
while epoch < MAX_EPOCHS:
  batches = split(training_samples, M)
  for batch in batches:
    p = p - step * gradient(batch)
  epoch += 1
```

Figure 6: Mini-batch Gradient Descent

Backpropagation

Backpropagation is the process of updating the neural network coefficients. This involves computing the gradient of the multi-variable loss function using the chain rule, to obtain $\nabla_p J$.

Let's compute the gradient for the parameters in the last layer (2) of our network:

$$\frac{\partial \text{CE}(y, \hat{y})}{\partial z_k^{(2)}} = -\frac{\partial}{\partial z_k^{(2)}} \log\left[\frac{\exp(z_c^{(2)})}{\sum_{i=0}^{C} \exp(z_i^{(2)})}\right] = -\frac{\partial\left[z_c^{(2)} - \log\left(\sum_{i=0}^{C} \exp(z_i^{(2)})\right)\right]}{\partial z_k^{(2)}}$$

There are two cases here:

1. Case I: $k = c = y_i$, i.e., k is the correct label

$$\frac{\partial \text{CE}(y, \hat{y})}{\partial z_k^{(2)}} = -1 + \frac{\exp(z_k^{(2)})}{\sum_{i=0}^{C} \exp(z_i^{(2)})} = -1 + \hat{y}_k = \hat{y}_k - y_k$$

2. Case II: $k \neq y_i$

$$\frac{\partial \text{CE}(y, \hat{y})}{\partial z_k^{(2)}} = 0 + \frac{\exp(z_k^{(2)})}{\sum_{i=0}^{C} \exp(z_i^{(2)})} = \hat{y}_k - y_k$$

Therefore, the gradient in vector notation simplifies to

$$\frac{\partial \text{CE}(y, \hat{y})}{\partial z^{(2)}} = \hat{y} - y \tag{4}$$

Recall that $z^{(2)} = W^{(2)} a^{(1)} + b^{(2)}$, such that $z^{(2)} \in \mathbb{R}^{H_2 \times 1}$, $a^{(1)} \in \mathbb{R}^{H_1 \times 1}$ and $W^{(2)} \in \mathbb{R}^{H_2 \times H_1}$. Therefore,

$$\frac{\partial \text{CE}(y, \hat{y})}{\partial W^{(2)}} = \frac{\partial \text{CE}(y, \hat{y})}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(2)}}$$

$$\boxed{\frac{\partial \text{CE}(y, \hat{y})}{\partial W^{(2)}} = (\hat{y} - y)[a^{(1)}]^T} \tag{5}$$

Similarly,

$$\boxed{\frac{\partial \text{CE}(y, \hat{y})}{\partial b^{(2)}} = \hat{y} - y} \tag{6}$$

Going across L_2:

$$\frac{\partial z^{(2)}}{\partial a^{(1)}} = [W^{(2)}]^T$$
$$\frac{\partial \text{CE}(y, \hat{y})}{\partial a^{(1)}} = \frac{\partial \text{CE}(y, \hat{y})}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} = [W^{(2)}]^T (\hat{y} - y)$$

Going across the non-linearity of L_1:

$$\begin{aligned}\frac{\partial \text{CE}(y, \hat{y})}{\partial z^{(1)}} &= \frac{\partial \text{CE}(y, \hat{y})}{\partial a^{(1)}} \frac{\partial \sigma(z^{(1)})}{\partial z^{(1)}} \\ &= \frac{\partial \text{CE}(y, \hat{y})}{\partial a^{(1)}} \circ \sigma(z^{(1)}) \circ (1 - \sigma(z^{(1)}))\end{aligned}$$

Note that we have assumed that $\sigma(\cdot)$ works on vectors (matrices) by applying an element-wise sigmoid, and $\circ$ is the element-wise (Hadamard) product.

That brings us to our final gradients:

$$\frac{\partial \mathrm{CE}(y, \hat{y})}{\partial W^{(1)}} = \frac{\partial \mathrm{CE}(y, \hat{y})}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial W^{(1)}}$$

$$\boxed{\frac{\partial \mathrm{CE}(y, \hat{y})}{\partial W^{(1)}} = \left(\frac{\partial \mathrm{CE}(y, \hat{y})}{\partial z^{(1)}}\right) x^T} \tag{7}$$

Similarly,

$$\boxed{\frac{\partial \mathrm{CE}(y, \hat{y})}{\partial b^{(1)}} = \frac{\partial \mathrm{CE}(y, \hat{y})}{\partial z^{(1)}}} \tag{8}$$

The above equations have been derived for a single training vector, but they extend seamlessly to a matrix of N column vectors. In that case, you need to sum up over all the input images x.

Problem 3 Element-wise CUDA kernels

In each iteration of Mini-batch Gradient Descent, the images in a mini-batch should be distributed evenly among MPI nodes, and computation in the forward and backward passes on each MPI node (such as matrix multiplication and softmax) should be accelerated using GPU kernels.

You will implement a part of the final project in each homework. In this homework, you will implement classes to manage memory on the GPU for matrices and GPU kernels to accelerate element-wise matrix operations that need to be performed in the forward and backward passes during training. Follow the comments in `gpu_func.h` and `main_q3.cpp` to help guide your implemenations. We will use the dense matrix class and related functions from the Armadillo C++ library; please refer to the documentation at `https://arma.sourceforge.net/docs.html`.

To allow for easily switching between training in single precision or double precision, we have defined the type alias `nn_real` in `common.h`. You should use this type alias instead of using `float` or `double` directly. We have also defined the macros `Log` and `Exp` in `common.h`, which you should use to compute the natural logarithm and exponential within your kernels.

Question 3.1

20 points. Implement the constructors, destructor and the `to_cpu` function for the `DeviceAllocator` class in `gpu_func.cu`.

Question 3.2

10 points. Implement the constructors for the `DeviceMatrix` class in `gpu_func.cu`.

Question 3.3

10 points. Implement the kernel `MatSigmoid` and the kernel wrapper method `DSigmoid` in `gpu_func.cu`.

Question 3.4

10 points. Implement the kernel `MatRepeatColVec` and the kernel wrapper method `DRepeatColVec` in `gpu_func.cu`.

Question 3.5

10 points. Implement the kernel `MatSum` and the kernel wrapper method `DSum` in `gpu_func.cu`.

Question 3.6

10 points. Implement the kernel `MatSoftmax` and the kernel wrapper method `DSoftmax` in `gpu_func.cu`.

Question 3.7

10 points. Implement the kernel `MatCrossEntropyLoss` and the kernel wrapper method `DCELoss` in `gpu_func.cu`.

Question 3.8

10 points. Implement the kernel `MatElemArith` and the kernel wrapper method `DElemArith` in `gpu_func.cu`.

Question 3.9

10 points. Implement the kernel `MatSquare` and the kernel wrapper method `DSquare` in `gpu_func.cu`.

Question 3.10

10 points. Implement the kernel `MatSigmoidBackProp` and the kernel wrapper method `DSigmoidBackprop` in `gpu_func.cu`.

References

[1] Yann LeCun et. al. MNIST. `http://yann.lecun.com/exdb/mnist/`. [Online].

A Submission instructions

To submit:

1. For all questions that require explanations and answers besides source code, put those explanations and answers in a separate PDF file and upload this file on Gradescope.

2. The homework should be submitted using a submission script on `cardinal`. The submission script must be run on `cardinal.stanford.edu`.

3. Copy your submission files to `cardinal.stanford.edu`. The script `submit.py` will copy only the files below to a directory accessible to the CME 213 staff. Only these files will be copied. Any other required files (e.g., Makefile) will be copied by us. Therefore, make sure you make changes only to the files below. You are free to change other files for your own debugging purposes, but make sure you test it with the default test files before submitting. Also, do not use external libraries, additional header files, etc, that would prevent the teaching staff from compiling the code successfully. Here is the list of files we are expecting and that will be copied:

   ```
   main_q1.cu
   recurrence.cuh
   benchmark.cuh
   gpu_func.cu
   main_q3.cpp
   ```

   The script will fail if one of these files does not exist.

4. To check your code, we will run the following on `cme213-login`:
   `$ make`
   This should produce 3 executables: `main_q1`, `main_q2` and `main_q3`.

5. To submit, type:

```
$ /afs/ir.stanford.edu/class/cme213/script/submit.py hw2 <directory with your submission files>
```

B Advice and Hints

- For debugging it will be helpful to limit the number of cases being run to 1. In the recurrence problem, do this by using 1 value instead of the arrays for the 3 for loops.
- If you need some documentation on CUDA, you can look at the documents linked on canvas or visit the CUDA website at `https://docs.nvidia.com/cuda/index.html`.
- An easy way to transfer the table output into a plot is to copy the space-separated program output, paste it into a Google Sheet, highlight the column that contains the data, and click "Data→Split text to columns" in the top banner, then hightlight your new columns and click "Insert→Chart."