CME 213, Introduction to parallel computing
Eric Darve, Kingway Liang
Winter 2020

Neural Networks on CUDA
Part II: starter code, grading details, instructions

In this second part of the final project, we provide further details about the grading policy and introduce you to the starter code. You can also find instructions for running and profiling the code on GCP VM and submitting your work.

1 Grading details

Please refer to Part I for an overall grading information. Here we explain in detail how we determine the correctness of the code and test the performance. We have setup four test cases (with corresponding grading modes in the code) for testing correctness and performance. These test cases or grading modes can be run by passing command line arguments to the program. More details about them are given in later sections.

1.1 GEMM correctness

Since the GEMM function is a building block of any neural network implementation and will be an important tool in your arsenal, we test the GEMM implementation separately from the overall code testing. We have provided a function prototype called `myGEMM` for you in `gpu_func.cu`, which takes inputs as two scalars α, β, three matrices A, B, C, and returns the result of $D = \alpha\, A\, B + \beta\, C$ in C (in place).

Your job is to fill in this function, and we will test your implementation on two sets of inputs that are relevant to this project: $A \in \mathbb{R}^{800\times784}, B \in \mathbb{R}^{784\times1000}$; and $A \in \mathbb{R}^{800\times1000}, B \in \mathbb{R}^{1000\times10}$. You are welcome to, but you don't have to use this myGEMM function in your parallel training; this is only for the purpose of grading.

We test this correctness by running grading mode 4, which runs the myGEMM function alone. This myGEMM function is called only by rank 0 in the grading mode, i.e., for this part you just need to write kernels to do GEMM on a single GPU.

1.2 Overall correctness

In large neural network problems, a common issue encountered is the aggregation of rounding errors or inconsistencies. Unfortunately, the implementations of several operations are not exactly same on CPU and GPU. Some of the sources for differences include `exp()` operations used in Softmax and Sigmoid functions, FMA (fused multiply add), the order of operations etc. There are some differences at the hardware level of implementation too. These discrepancies are usually of the order of 10^{-16} for double precision calculations. However, such discrepancies can build up over time. In general, as the learning rate gets larger, the instability of the algorithm due to roundoff errors is high. These discrepancies might not lead to any parameter blow-up, but might create significant differences between the CPU and GPU solutions. This makes determining correctness challenging.

In order to tackle this, we have setup three test cases for determining correctness in the form of grading modes. In all those modes, a max norm of the difference between final CPU and GPU results (parameters $W^{(1)}$, $W^{(2)}$, $b^{(1)}$, $b^{(2)}$) is considered. If this max norm is greater than a threshold, which is 10^{-7}, for any case, your code will fail the correctness test for that case. The actual max norm values we get are much lower than this, but we want to provide some leeway in this regard and have relaxed the threshold. Apart from passing the three correctness tests, the precision on the validation set of the CPU and GPU implementations must be very close.

The hyper-parameters for the three test cases are as follows

1. Low learning rate: 0.001; large # iterations: 40 epochs

2. Medium learning rate: 0.01; medium # iterations: 10 epochs

3. High learning rate: 0.025; small # iterations: 1 epoch

The grading modes 1, 2 and 3 run the above three test cases respectively.

Note: In order to get full credit on overall code correctness, all test cases above must meet the threshold by running a fully parallel code on an arbitrary number of processes between 1 and 4 (or CPU threads) using MPI and CUDA. If the code is running on a single GPU or is not using GPUs (just MPI), you will lose a significant portion of the grade. Similarly, if you are running four processes but only one of them is using GPUs, you will again lose points. Here, when we say running on GPUs, we expect that **all** the GEMM, Softmax and Sigmoid calculations be done on GPUs.

Note: For your convenience, we have provided a function to output the differences between the serial and parallel versions into a file, and you can use this by passing the debug flag `-d` when running your code. Details of this debug mode can be found in subsection 3.2.

1.3 GEMM Performance

This refers to the performance of your `myGEMM` function. To test this we run the code in grading mode 4. The grade for this will be based on the performance of your GEMM function (in terms of the time taken) relative to other students in the class. The exact method for calculating this relative grade will be determined by us later depending on the range of performances we get.

In the code, we run this myGEMM function repeatedly for a number of iterations. This has been currently set to 10, but we might change this based on the performance we see in the submissions. We believe that this should not affect your implementation.

Caveat: If your GEMM implementation does not pass the GEMM correctness test, you will not receive any points for performance.

1.4 Overall Performance

This refers to the performance of your full NN code. Here we use the default settings of the program for benchmarking the performance (time taken). Here again, the grade is based on your performance relative to other students in the class. The exact method for calculating this relative grade will be determined by us later depending on the range of performances we get.

Caveat: If you do not pass the overall correctness tests, points will be deducted.

2 Starter-code

The starter code integrates the GPU CUDA code and other C++ code. The GPU code is first compiled by `nvcc` into object files, and then linked with other parts of the project and libraries by `mpic++` linker. The project is using the `Armadillo` library for matrices and vectors. The details about the files are below. Those marked with a star (*) will not be submitted by the submission script. You are free to modify those files for debugging purposes, but make sure you test with the original version of those files before you submit. In the other files, you may write any number of functions you wish to.

Note: Please make sure you adequately comment your code and also structure it well. Although we will avoid going through the code in detail, it will help us read your code in case we have to.

- *`create_vm_final_project_4gpu.sh` and *`create_vm_final_project_1gpu.sh`: Scripts to create Google cloud VM. These files are on the `stanford-cme213.github.io` GitHub repository, under `Code/`
  `https://github.com/stanford-cme213/stanford-cme213.github.io/tree/master/Code`
  The scripts install all necessary libraries and download MNIST dataset for you on the VM. The only difference between the two scripts is the number of GPU. If you wish, you can develop your kernels on the VM with 1 GPU to conserve credit. Follow the instructions listed after running the script. You can safely ignore the warnings about
  `You have selected a disk size...` and `Disk size: '40 GB' is larger than...`
  Once the VM is up, please log in and run
  ```
  $ download_starter_code
  ```
  This will download the starter code, with all the files listed below, from GitHub on the VM.

- *`main.cpp`: This is the main file for the project. You do not need to change this file except for your own debugging purposes.

- `gpu_func.cu`, `inc/gpu_func.h`: You should implement your GPU CUDA wrapper functions and kernels in `gpu_func.cu` and declare them in `inc/gpu_func.h`. This separates the source code so that `nvcc` only compiles the CUDA code into object files, which can be linked into other parts of the project by the `mpic++` linker.

- *`inc/neural_network.h`: This file contains a basic C++ class to implement the two layer neural network. Note that all members in `neural_network` are declared to be public, and you can access them directly, which allows an easier MPI implementation than with a more encapsulated class.

- `neural_network.cpp`: This file already contains a serial implementation of the neural network. Your objective is to fill the `parallel_train` function with the parallel implementation.

- *`utils/tests.cpp` *`utils/tests.h`: These files contain the tests used for determining correctness and testing performance.

- *`utils/common.cpp`, *`utils/common.h`: These files contain common operations on `arma::mat` that may be useful. You can make your own GPU CUDA implementation accordingly in `gpu_func.cu`.

- *`utils/test_utils.h`: This file contains helper functions useful for debugging and testing, e.g., a function to compare a memory space representing a matrix with an Armadillo Matrix to check if the GPU implementation is correct.

- *`utils/mnist.cpp`, `utils/mnist.h`: These files contain code that reads in the MNIST dataset.
- *`Outputs` folder: All the output files go into this folder. There is another folder named CPUmats inside this folder. All the CPU matrices that are written out during debug mode go into this folder.
- *`obj` folder: All the object files generated during compilation will be stored here.

3 Instructions

3.1 Suggested order of implementation

1. Before implementing anything, be sure you understand how everything will work together. You don't need to know exactly you will implement everything, but you should have a high level understanding; otherwise you might make assumptions during implementation that is incompatible with the overall computation and it will be difficult to debug.
2. Implement the GPU kernels. Remember to test on multiple matrix sizes to ensure your GPU kernel handles different cases well. You may choose to implement a single-GPU version of the full code as well.
3. Validate your parallel algorithm by implementing a "pseudo-parallel" code. This means: divide the data into different parts, but have one process perform the calculation. This does not yet involve MPI, but serves to validate your parallel data decomposition.
4. Implement the MPI version.
5. Optimize your GPU kernels.

3.2 Running instructions

To run your compiled code using a single process and GPU, use

```
./main [args]
```

To run your compiled code using `N` processes and GPUs, use

```
mpirun -mca btl ^openib -n [N] ./main [args]
```

The command line arguments for `main` are explained in the next section. The argument
`-mca btl ^openib`
is optional and simply removes some of the warnings when running `mpirun`.

3.3 Command line arguments

We provide several useful command line arguments for main:

- `-n num` to change number of neurons in the hidden layer to `num`
- `-r num`, `-l num` to change `reg` and `learning_rate`
- `-e num`, `-b num` for `num_epochs` and `batch_size`

- `-s` to run the sequential training together with your parallel training to compare their performance.
- `-d` for the debug mode. This mode is for the convenience of debugging your code: it will output the differences of the parameters between the CPU version and the GPU version into a file. For the first time, you need to run the debug flag together with the serial flag: `-sd`, and this will write the parameters from the CPU version for the first batch of each epoch into files. For later runs (with the same hyper-parameters), you need to use the debug flag. This automatically uses the already stored CPU files so that you need not wait for the CPU code to run.
- `-p num` to print debug output and files every `num` iterations; This overrides the default setting of writing only for first batch of each epoch.
- `-g option` for grading mode. Options are 1, 2, 3, 4. Options 1, 2, 3 run the three test cases for checking correctness, and option 4 runs the GEMM case.

3.4 Profiling instructions

For profiling, we will use nvprof (NVIDIA Profiler) and nvvp (NVIDIA Visual Profiler). Here is a quick tutorial. You will run nvprof on the VM to generate profiling data. To visualize the profiling, you will use nvvp, which can be run (i) remotely on the Google Cloud VM, (ii) remotely on rice, or (iii) locally on your machine.

nvprof: To profile the code, use nvprof, the command line profiler

- To profile **multiple MPI processes:**
  ```
  mpirun -mca btl ^openib -np 4 nvprof --output-profile profile.%p.nvprof ./main [args]
  ```
- Profiling kernel(s) in depth can take a very long time, use the `-e` flag in the arguments for main to limit the number of epochs, e.g. `./main -e 1`
- To profile **a specific kernel** in depth in one process:
  ```
  sudo -E /usr/local/cuda-10.2/bin/nvprof --kernels <KERNEL NAME> --analysis-metrics
  --output-profile profile.%p.nvprof ./main [args]
  ```
- To profile **all kernels** in depth in one process (**this may take hours**):
  ```
  sudo -E /usr/local/cuda-10.2/bin/nvprof --analysis-metrics
  --output-profile profile.%p.nvprof ./main [args]
  ```
- You'll get one profiling output file for each MPI process. The file is tagged with the process ID. Make sure to keep your profiler outputs organized, or it might get hard to figure out the profile files from your latest run.

Running nvvp remotely: In order to use nvvp on the VM or on rice, you need to ssh using the `-Y` option (to enable graphics forwarding). Further, you need XQuartz for Mac (or some form of X11) installed on your local system.[1] If running on rice, you need to transfer the profiling output files to rice.

[1] After installation, XQuartz is located in `/Applications/Utilities/`.

nvvp on Google Cloud VM:

- Make sure you log into the GCP VM `project` with X11 forwarding. Log in with
```
gcloud compute ssh <VM name> --ssh-flag="-Y"
```
- Run the following one-time setup on VM before invoking nvvp
```
sudo update-alternatives --config java
```
Then select option 2 (the option for java-8).
- Run the NVIDIA Visual Profiler. On GCP VM, you invoke nvvp using
```
nvvp &
```

nvvp on rice:

- Log into rice with X11 forwarding
```
ssh <your SUNet ID>@rice.stanford.edu -Y
```
- Run
```
module load cuda
```
- Launch nvvp
```
nvvp &
```

Installing nvvp locally: You can install and run nvvp on your local machine.[2] You need to install the CUDA Toolkit 10.2, not the Driver or Samples. Note the requirements: Windows 7, 8.1, or 10; Linux; Mac OSX 10.13. Mac OSX Catalina is version 10.15, unfortunately.

You do not need a GPU for this on your local machine. To install, open the link in the footnote, select your OS type and download the installation file. You need to follow the same instructions for running the command line profiler on the VM; then load the profiler outputs into nvvp.

Using nvvp: In the opened window, choose File → Import → Nvprof → Multiple Processes, or Single Process (depending on what you profiled), and Browse to select all the profiling output files for timeline data → Finish, then proceed with the analysis. We will have a hands-on workshop on profiling.

3.5 Submission instructions

1. Make sure your code compiles on GCP and runs.
2. The writeup should be written in `prelim_report.pdf` and `final_report.pdf` for the Preliminary and Final report respectively. Please upload the PDF file to Gradescope.
3. The project should be submitted using the submission script on `cardinal`. The submission script must be run on `cardinal.stanford.edu`.
4. Copy your submission files to `cardinal.stanford.edu`. You can use the following command in your terminal:
```
scp <your submission file(s)> <your SUNetID>@cardinal.stanford.edu:
```

[2] `https://developer.nvidia.com/cuda-downloads`

5. The submission script will then copy the files below to a directory accessible to the CME 213 staff. Only the following files will be copied. Make sure these files exist and that no other files other than those provided in the starter code are required to compile and run your code. In particular, 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:

   ```
   gpu_func.cu
   gpu_func.h
   neural_network.cpp
   ```

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

6. To submit, type:
   ```
   /afs/ir.stanford.edu/class/cme213/script/submit.py final_part1 <directory with your submission files>
   ```
   for Part 1, and
   ```
   /afs/ir.stanford.edu/class/cme213/script/submit.py final_part2 <directory with your submission files>
   ```
   for Part 2.