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

May 11, 2018

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 the cluster 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 testcases (with corresponding grading modes in the code) for testing correctness and performance. These testcases 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: $A \in \mathbb{R}^{800\times 784}, B \in \mathbb{R}^{784\times 1000}$; and $A \in \mathbb{R}^{800\times 1000}, B \in \mathbb{R}^{1000\times 10}$. 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 testcases 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 testcases above must meet the threshold by running a fully parallel code on 4 GPUs through 4 different processes (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 `g++` 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.

- `*init.sh`: This file contains the script to download the MNIST dataset and install Armadillo. This only needs to be run once. Please see the running instructions before you run this script.
- `*run.sh`: This file contains the script to run the program using SLURM. See 3.2 for further details.
- `*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 `g++` 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. 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.
2. 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.
3. Implement the MPI version.
4. Optimize your GPU kernels.

3.2 Running instructions

1. To use nvcc on the cme213-cluster, you must have the correct modules loaded. If you type the command

   ```
   $ module list
   ```

   the output should be:

   ```
   Currently Loaded Modules:
     1) gnu/5.4.0   2) cuda/8.0   3) openmpi/1.10.6
   ```

   If not, unload and load the appropriate modules using for example

   ```
   $ module unload gnu7/7.3.0
   $ module load gnu/5.4.0
   ```

   You can copy and paste these two lines to your ~/.bashrc file so they will be loaded automatically when you connect on the cluster. Other useful commands include:

   ```
   $ module avail
   $ module -h
   ```

2. With the correct modules loaded, run

   ```
   ./init.sh
   ```

   This downloads the MNIST dataset and installs the Armadillo library. You only need to do this the first time after you download the code.

3. Edit the job script `run.sh` to add command line arguments or change number of processes you want to run with. By default, we request for 4 processes on a single node in the cluster and request for 4 GPUs. The single node is to reduce MPI overhead. Communication across nodes is slower than within a single node. Note that the program prints out the number of MPI processes and CUDA devices used in the very beginning to help you make sure you are running it correctly.

4. Submit the job script `run.sh` using `sbatch` as follows

   ```
   sbatch run.sh
   ```

   You can check whether your job is still running via the command

   `squeue` or `squeue -u <your SUID>`.

   You can kill your running jobs using command

   ```
   scancel <Your job ID>
   ```

3.3 Command line arguments

We provide several useful command line arguments to 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 just 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.

All options are optional.

3.4 Profiling instructions

As nvvp (NVIDIA Visual Profiler) may come handy in the coming optimization part of the project, here is a quick tutorial for using nvvp on the cluster. In order to use nvvp on the cluster, you need to ssh using the `-X` or `-Y` option (to enable graphics). Further, you need XQuartz for Mac (or some form of X11) installed on your local system.

Note: You can also install and run nvvp on your local machine.[1] You only need to install the CUDA Toolkit, not the Driver or Samples. You do not need a GPU for this on your local machine. You need to follow the same instructions below for running the command line profiler on the cluster and then load the profiler outputs to nvvp.

- Make sure you log in to icme cluster with X11 forwarding. Log in with

  ```
  ssh -Y <your SUID>@cme213-cluster.stanford.edu
  ```

- To profile the code, comment out the default run command. Then, use `nvprof` to generate the profiling output. `nvprof` is the command line profiler for CUDA. The command that we run is

  ```
  MV2_USE_CUDA=1 mpirun -np 4 nvprof --output-profile profile.%p.nvprof ./main [args]
  ```

  You can also profile a single kernel with the following command

  ```
  MV2_USE_CUDA=1 mpirun -np 1 nvprof --kernels gpu_GEMM --analysis-metrics
     --output-profile GEMMmetrics.out.%p.nvprof ./main [args]
  ```

  `MV2_USE_CUDA=1` makes MVAPICH2 CUDA aware.

- Submit the job script with the profiling command uncommented and default command commented using

  ```
  sbatch run.sh
  ```

- You'll get one profiling output file for each MPI process. The file is tagged by the process ID. Make sure you keep your profiler outputs organized or it might get hard to figure out your profile files from your latest run.

- Run the NVIDIA Visual Profiler. On the cluster, it is invoked using

  ```
  nvvp &
  ```

- In the opened window, choose File → Import → Nvprof → Multiple Process, and Browse to select all the profiling output files → Finish.

[1] `https://developer.nvidia.com/nvidia-visual-profiler`

3.5 Submission instructions

1. Make sure your code compiles on `cme213-cluster` 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 a 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:
   ```

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. Type:

   ```
   /usr/bin/python /usr/class/cme213/WWW/script/submit.py final_part1 <directory with your files>
   ```

   for Part 1

   Type:

   ```
   /usr/bin/python /usr/class/cme213/WWW/script/submit.py final_part2 <directory with your files>
   ```

   for Part 2

7. You can submit at most 10 times before the deadline; each submission will replace the previous one.
8. For the preliminary part there will be a 10% penalty per 24 hours for late submission. We will not accept submissions that are submitted past one late day. You have NO late day for the final report.