CME 213, Introduction to parallel computing
Eric Darve
Spring 2021

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.

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: $A \in \mathbb{R}^{800\times 784}, B \in \mathbb{R}^{784\times 1000}$; 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 this 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), and the order of operations. These discrepancies are usually of the order of the machine roundoff errors. 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. The hyper-parameters are varied for each of the three grading modes 1, 2 and 3. Please see `main.cpp` for details. 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, 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.

In order to get full credit on overall code correctness, all test cases above must meet the threshold by running a fully parallel code with a number of processes equal to **1, 2, and 4** 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.

Case of 3 GPUs (bonus points +5): The case of 3 GPUs is more difficult. Recall that the full dataset of images needs to be split into batches. The total number of images is equal to 60,000. They are split into batches of size 800. Here is a function that calculates the size of each batch:

```
int get_batch_size(int N, int batch_size, int batch) {
  int num_batches = (N + batch_size - 1) / batch_size;
  return (batch == num_batches - 1) ? N - batch_size * batch : batch_size;
}
```

Each batch of size 800 then needs to be divided into mini-batches that are assigned to each GPU. When you select 3 GPUs, it won't divide evenly. GPU 0 will have 267 images, GPU 1 will have 267 images, and GPU 2 will have 266 images ($800 = 267 + 267 + 266$). Here is a function that calculates the size of the mini batch for each rank:

```
int get_mini_batch_size(int batch_size, int num_procs, int rank) {
  int mini_batch_size = batch_size / num_procs;
  return rank < batch_size % num_procs ? mini_batch_size + 1 : mini_batch_size;
}
```

More importantly, using `MPI_Scatter` is a little different since `MPI_Scatter` assumes that each chunk of data has the same size. For the case of 3 GPUs, as noted above, the chunks have different sizes (267 and 266). As a result, you should use `MPI_Scatterv` which allows specifying chunks of different sizes. You will find documentation for these functions at these pages:
`https://www.open-mpi.org/doc/v4.1/`
`MPI_Scatter`
`MPI_Scatterv`

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.

In previous years, we were running the calculation using the double precision type `double`. However, the Turing GPUs have low performance in double precision so we switch the code to single precision with the type `float`. The file `utils/types.h` is managing the switch from `double` to `float`. For flexibility, we define the type `real` and make it equal to `float` or `double` using the flag `USE_DOUBLE`. With the current setup in the starter code, the type `real` is equal to `float`.

Note: Please make sure you adequately comment your code and also structure it well. This will help us read your code.

- *`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/mnist.cpp`, `utils/mnist.h`: These files contain code that reads in the MNIST dataset.
- *`utils/types.h`: This file contains code to define the type to use for the neural network—we are using single precision `float`s.
- *`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 have a clear idea of the overall organization of the calculation and how things will be done, at least at the organizational level. This will help you start in the right direction and make the correct assumptions when organizing your calculations.
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 a starting point.
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 only. This does not yet involve MPI, but serves to validate your parallel approach and data decomposition strategy.
4. Implement the MPI version with communication and using multiple ranks. Make sure your code works with 1, 2, (3,) and 4 ranks.
5. Optimize your GPU kernels. Use shared memory. The primary metric to improve is the arithmetic intensity, that is you should try to reduce the memory traffic between global memory and register files.
6. Optimize the MPI communications and parallelization strategy to minimize communication between MPI ranks.

3.2 Compiling and running instructions

In order to compile the code, you will need to load the following modules:

```
# Modules for final project
module load cuda/11.0
module load openmpi/4.1.0
module load armadillo/10.4.1
```

You may find it convenient to copy these lines at the end of the file ~/.bashrc in your HOME directory on icme-gpu. If you do so, the modules will be automatically loaded every time you log on the computer so you don't have to do it manually.

To compile the code, just run

```
$ make
```

in the directory containing the final project files.

To run your compiled code, run `sbatch run.sh`. Within the script, to use a single process and GPU, use

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

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

```
mpirun -n [N] ./main [args]
```

The command line arguments for main are explained in the next section.

3.3 Command line arguments

We provide several useful command line arguments for main:

- `-n int` to change number of neurons in the hidden layer to `num`
- `-r float`, `-l float` to change `reg` and `learning_rate`
- `-e int`, `-b int` 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 the file `Outputs/CpuGpuDiff.txt`. For the first time, you need to run the debug flag together with the serial flag: `-sd`; this will write the parameters from the CPU version for the first batch of each epoch into files (see directory `Outputs/CPUMats`). For later runs (with the same hyper-parameters), you can use the debug flag only (without `-s`). This automatically uses the already stored CPU files (`Outputs/CPUMats`) so that you need not wait for the CPU code to run.
- `-p int` to print debug output and files every `num` iterations. This overrides the default setting of writing only for first batch of each epoch.
- `-g int` 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 Nsight Systems and Nsight Compute. Install the Nsight Systems and Compute GUI using the instructions below.

1. Create a developer account.
2. Download Nsight Systems (one of Linux/Windows/macOS Host).
3. Download Nsight Compute locally from here.
4. Follow the installation instructions.

Nsight Systems Generate the file `nsys.qdrep` by

1. modifying `run.sh` by adding `nsys profile -o nsys --trace cuda,mpi mpirun` in front of the `mpirun` command and
2. running `sbatch run.sh`.

Copy this file over to your local machine. Open the Nsight Systems application, click File → Open, and select `nsys.qdrep`. Follow this link for more details.

Nsight Compute Get the file `ncompute.ncu-rep` by

1. modifying `run.sh` to run `ncu -o ncompute ./main -g 1` and
2. running `sbatch run.sh`.

Copy this file over to your local machine. Open the Nsight Compute application, click File → Open, and select `ncompute.ncu-rep`. Follow this link for more details.

Tip: 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`

3.5 Submission instructions

1. Make sure your code compiles on `icme-gpu` 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:
   ```
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.h
   gpu_func.cu
   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.