# ECE 459: Programming for Performance
# Lab 3 — Using the GPU [1]

Lab created by Yuhan Lin; manual by Bernie Roehl
Due: March 11, 2022 at 23:59 Eastern Time

In this lab, you'll learn how to improve performance of a convolutional neural network (CNN) by using the GPU to do some of the work.

## Learning Objectives

- Become familiar with running code on the GPU
- Learn how to use the CUDA interface

## Background

GPUs were originally designed to accelerate computer graphics (hence the name, Graphics Processing Unit), but their architecture makes them suitable for a wide range of tasks that can be parallelized. There are two major programming interfaces between the CPU and the GPU – OpenCL and CUDA. While OpenCL is more portable and runs across a wide range of GPUs, CUDA usually offers better performance and is very widely supported. It's also easier to use. The biggest limitation of CUDA is that it only runs on NVidia GPUs.

## CUDA

The typical CUDA workflow consists of having the CPU initialize the GPU, copy data into the GPU memory, invoke the code on the GPU (which is called a "kernel"), and copy the results from the GPU back into CPU memory. In this lab, the CPU portion of the workflow is handled by a library of Rust bindings called RustaCUDA. The GPU kernel is programmed in CUDA-C, a language similar to C++ with none of Rust's safety guarantees. As such, calling a CUDA kernel from Rust is an unsafe operation and you must carefully coordinate between your Rust code and CUDA-C code to avoid undefined behaviour.

You can find more information about the CUDA-C language here:
`https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

## The CNN

A Convolutional Neural Network (CNN) is a machine learning algorithm that takes an image and passes it through multiple layers for processing, producing a single vector of values as output. In this lab, we'll be using a simplified version of a CNN that consists of only three layers. The inputs to the CNN are the $100 \times 100$ matrices that represent images.

The first layer of the CNN is a Convolution layer consisting of 10 neurons, each containing a $5 \times 5$ matrix called a filter. Each neuron divides the input image into $5 \times 5$ sections and performs a dot product between its filter and each section of the input, producing a $20 \times 20$ output matrix of products.

The second layer is a Rectified Linear Unit (ReLU) layer, which sets any negative values in the Convolution layer output to zero.

The third layer is the Output layer which consists of 10 neurons, each containing a $4000 \times 1$ vector called weights. Each Output neuron flattens all the matrices' output from the previous layer and concatenates them, forming a $4000 \times 1$ vector that it multiplies with its weights via a dot product. Since each Output neuron produces a single value, the entire Output layer produces an output vector of 10 values.

[1] v1, 17Feb22

## The Software

Your program reads in a CNN description and a number of $100 \times 100$ matrices (representing images), and runs the neural network by feeding the matrices through the CNN. The program writes its results to an output file, and reports the elapsed number of microseconds. The output files contain the output vectors for each input matrix.

All the data files (input, output and CNN descriptions) are in CSV (Comma-Separated Values) format, which is basically just a text file where every line contains values separated by commas (just as the name promised!).

We provide you with some starter code, including a main.rs file (which you will not need to modify), a `cpu.rs` file containing the CPU-based implementation, and a `cuda.rs` file containing a skeleton for the GPU-based implementation. There is also a file called `kernel/kernel.cu`, which contains a skeleton for the code that will run on the GPU. There is also a `build.rs` file that gets run automatically at build time to compile the `kernel.cu` file into a `kernel.ptx` file which gets downloaded to the GPU.

**Running things for real.** Note that in order to build the code that runs on the GPU, you must use the correct version of the gcc compiler. Here's a command you can use to set that up:

```
nvcc -compiler-bindir /usr/bin/gcc-6 -ptx nbody.cu
```

There's a bash script in the kernel directory call `setgcc` that does this for you.

The command line for running your program is as follows:

```
cargo run -release - <mode> <cnn_file> <input_file> <output_file>
```

where `<mode>` is either `"cpu"` to run the CPU-based implementation or `"cuda"` to run the GPU implementation. All the files are pathnames. You would typically use the following commands:

```
cargo run -release - cpu input/cnn.csv input/in.csv output/out.csv
```

```
cargo run -release - cuda input/cnn.csv input/in.csv output/out_cuda.csv
```

The program outputs the time spent doing "actual work", i.e. converting the input matrices to the output vectors. This measurement does not include i/o or the overhead of initializing the GPU. As such, the time should be lower for the CUDA version than the CPU version.

## Generating Inputs and Checking Results

In addition to the starter code, we provide two Python scripts. One (generate.py) is used to generate the input and CNN files and write them into the input directory as `in.csv` and `cnn.csv` . The other script (`compare.py`) is used to compare the two files in the output directory (`out.csv` , which is the output of the CPU version, and `out_cuda.csv` which is the output of the GPU version). If your CUDA-based implementation is correct, `compare.py` should not report any differences between those two output files. Note that the output files will not be perfectly identical, which is why we use a script to do the comparison.

The files generated by `generate.py` and the files compared by `compare.py` have fixed names, so it's recommended that you use the same names on the command lines that you use to run the application.

The scripts are written in Python 3, so you can run them using the python3 command (e.g. `python3 compare.py`).

## Rubric

**Implement your code on the GPU (40 marks)** Your code needs to build correctly and run on the GPU.

**Produce correct results (40 marks)** The compare.py script should not report any discrepancies.

**Report (20 marks)**

- 8 marks for a clear discussion of how your kernel works
- 8 marks for a clear discussion of how cuda.rs works with the kernel
- 4 marks for clarity