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

Lab created by Yuhan Lin; manual by Bernie Roehl
Due: March 18, 2024 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

Lab video. `https://www.youtube.com/watch?v=s3hCPNnUIcc`

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 (a "kernel") on the GPU, 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`

Here's a resource we pointed at last year:
`http://users.wfu.edu/choss/CUDA/docs/Lecture%205.pdf`

Here's another resource about parallel reductions (dividing and conquering) in CUDA:
`https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`

And just for fun: `https://www.youtube.com/watch?v=-P28LKWTzrI`

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 100×100 matrices that represent images.

[1] v3, 9Feb23

Mathematically, our CNN network is transforming a 100×100 input matrix into a 10-element vector by performing a bunch of matrix operations on the input. The input typically represents an image, and the output represents probabilities of classifications. For example, if we're using the CNN classifying images of digits, then each element of the output vector will represent the probability of the image being a given digit. In the real world, CNN networks are used to make these types of classifications, but for the purposes of the lab, you can think of it as a bunch of number crunching that'd be much more suited for the GPU than the CPU.

The first layer of the CNN is a Convolution layer, consisting of 10 neurons, each containing a 5×5 matrix called a filter. Each neuron divides the input image into 5×5 sections and performs a dot product between its filter and each section of the input, producing a 20×20 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, consisting of 10 neurons, each containing a 4000×1 vector called weights. Each Output neuron flattens all the matrices' output from the previous layer and concatenates them, forming a 4000×1 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.

The Software

Your program reads in a CNN description and a number of 100×100 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 `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. Once again, you can change the starter code, but don't change what the marker sees when running your code.

Running things for real. 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 -ptx nbody.cu
```

There's a bash script in the kernel directory, `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. If you're curious about what's slow, you can put `nvprof` in front of `cargo run`.

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

We're again requiring a commit log message (instead of a report). Commit messages should argue for why your change should be merged but imply less boilerplate than reports.

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

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

Commit Log (20 marks) 8 marks for a clear discussion of how your kernel works, how you know that it works (and how you know about its performance), 8 marks for a clear discussion of how `cuda.rs` works with the kernel, 4 marks for clarity.

What goes in a commit log

Here's a suggested structure for your commit log message justifying the pull request.

- Pull Request title explaining the change (less than 80 characters).
- Summary (about 1 paragraph): brief, high-level description of the work done.
- Tech details (3–5 paragraphs): anything interesting or noteworthy for the reviewer about how the work is done.
- Something about how you tested this code for correctness and know it works (about 1 paragraph).
- Something about how you tested this code for performance—although it doesn't actually have to be faster, you do have to evaluate performance (about 1 paragraphs).

Write your message in the file `commit-log/message.md`.

Clarifications

Can you clearly tell me which lectures are relevant to this lab? Lectures 21 and 22.

nvcc: command not found. Make sure you're on an `ecetesla` machine (or maybe `eceubuntu4`).

I'm not close enough, says `compare.py`. Are you using `double`s to represent numbers?

I'm confused about the syntax for `<<< ... >>>`. Check out
`https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#execution-configuration`.

The arguments are gridsize, blocksize, shared_mem, stream.

The third number (0) is the shared parameter. This defines the size of dynamic shared memory for the stream, but we don't need it here. The Rustacuda documentation (and/or source code!) can give you some guidance about its use if you have a use for it.

The second number (256) is the block size. The first parameter is the grid size, not the number of grids. The grid size tells us how many blocks are in there, and its number of points divided by 256, plus 1 just in case it does not divide evenly (which for something like 100k points, it does not). So for 100k points, it's 391 blocks.

We don't use the grid dimension because it's not really necessary in the calculation. `gridDim.x` would tell us the number of blocks in the grid in the `x` dimension, but that is not a useful figure in the calculation so we don't ask for it.

*****.** In Rust, multi-dimensional arrays are represented as a large 1D array, so your CUDA array representation has to match it. I would pass in the ConvLayer pointer as `const double*`. Since the dimensions of the convolution array are `(CONV_LAYER_SIZE, FILTER_DIM, FILTER_DIM)`, a 3D array access of `conv_layer[i][j][k]` should be written as `conv_layer[i*FILTER_DIM*FILTER_DIM + j*FILTER_DIM + k]` in your kernel.

Weird compile error. "I try to compile `let result_conv = launch!(self.module.convolution_layer...` and it doesn't work, so I'm sad."

Use a local variable to store `self.module` and launch that. For more details: `https://docs.rs/rustacuda/0.1.2/rustacuda/macro.launch.html`.

Weird nondeterministic behaviour that's changed by adding an unused variable in the kernel. Check your kernel for any out of bounds accesses on your buffers. Also, to reiterate, make sure you're using "double" as your floating point data type in the kernel. Others have reported that moving their declaration of the input device box above the stream and module declarations makes things more deterministic:

```
let mut input_matrix = DeviceBox::new(input).unwrap(); // <-- first
let mut layer2_output = DeviceBuffer::from_slice(&[0.0f64; 4000]).unwrap();
```

Go figure.

Other weird behaviour. Make sure you're synchronizing at the appropriate times (which may be fewer than you think). Word to the wise[2]: "CUDA operations issued to the same stream always serialize." That is, you only need to call synchronize() once per stream. Weird things may happen if you call it more than once.

[2] `https://forums.developer.nvidia.com/t/cudadevicesynchronize-needed-between-kernel-launch-and-cudamemcpy/44494`

I wish I could use a debugger. We had mixed reports from last year. People said print statements worked. You can use 'cuda-gdb' instead of the default 'rust-gdb' too.

I'm printing things but it's not quite right. Be sure to use %.2f to print a **d**ouble (sorry).

Multiple kernels? Multiple calls to the kernel? Yeah, go ahead.

Hey, that could be slow. Does performance actually count? Again, no (wait for Lab 4); just use the GPU in a sufficiently parallel way. But we may be sad. Unless you're not actually using the GPU. In which case you may then be sad. To expand on that:

You're expected to utilize the GPU's parallelism to a reasonable degree, so don't put everything on the same thread, and make sure the GPU solution is faster than the CPU solution. Consider a divide-and-conquer strategy for summing elements, since that's faster than atomicAdd. I'll use dot product as an example.

Let's say you need to do a dot product between vectors A and B, which are both 4000 elements long. The first step is to calculate the element-wise product C of A and B, which is easily parallelizable across 4000 threads (thread n calculates `C[n] = A[n]*B[n]`). The second step is summing C, which is the more complicated part. To do this we split C into 200 equal segments with 20 elements each, and assign each segment to a different thread (200 threads in total). Each thread sums its segment (20 elements), so the 200 threads produce another vector D of 200 elements where `sum(D) = sum(C)`. We can repeat the last step to reduce D to a 10-element vector, which can then be summed to the dot product.

An analogy would be if you wanted to collect and accumulate all the garbage across N households efficiently. First you'd send N workers to collect the garbage of all households (that's the easily parallelizable part). Then you dump the N bags of garbage into M landfills, where $M < N$. Finally, you assemble the garbage from all M landfills into a single garbage processing plant, completing the procedure.

Are constants constant? We won't change the constants in the CNN file, i.e. you can hardcode them if you want.

Do we commit compiled files to git? (`kernel.ptx` and `output/out_cuda.csv`) Doesn't matter, we'll regenerate.

Appendix: Taking arrays as arguments in C instead of raw pointers

This appendix is authored by previous ECE 459 student Aidan Wood, who was kind enough to allow us to include this in the description.

Arrays in C aren't really truly "arrays" in the sense the existence of a C array doesn't actually mean a contiguous block of memory the same dimension as the array actually exists in memory. An array is C is just a pointer to a position in memory where the programmer has stated "hey I have a memory position, and here's some information about how I like to think about the data around it".

Array bounds in C only serve as an "ease of use" feature for the programmer, when you define an array: `type arr[10]`, the value of `arr` is initialized to the memory location of element 0, and then when you index it with some value x: `arr[x]` this corresponds to the following operation `*(arr + sizeof(type) * x)` or "take the pointer `arr`, increment it by the size of a single element in the array (`sizeof(type)`) x times, and then dereference the next `sizeof(type)` bytes at that location, therefore producing the value at this position in the array".

Now, this logic also extends to multidimensional arrays in C, multidimensional arrays do not actually exist because memory is 1D contiguous, so an array `type arr[size_0][size_1][size_2]............` and then index it with some values `x_0, x_1, x_2.....`, this corresponds to `*(arr + (sizeof(type) * size_1 * size_2 * ....) * x_0 + (sizeof(type) * size_2 * .....) * x_1 + (sizeof(type) * .... ) * x_2 + .....)`

Now, what this means for the assignment is YOU DO NOT HAVE TO DO THE ARRAY INDEXING MANUALLY ON POINTERS, C is very very forgiving on type conversions, without any explicit casting you can bind a pointer of any type to an array of the same type with any dimensions, so when you pass in your device pointers to your CUDA kernels, you don't need to take them in as raw pointers, you can take them in as multidimensional arrays and then use those arrays to do the indexing for you: lets say your convolution layer for example.

Many people probably have something like

```
extern "C" __global__ void ConvolutionLayer(
    double* input_data, double* filters, double* output_data
)
```

Now this is fine, but you're gonna have to do manual 1D indexing off of each of these pointers which is more work than it has to be, instead, we know what the underlying memory layouts are (what dimensions the arrays were allocated as) so we can just tell C to interpret these pointers as those arrays, so for `input_data` we know its a 100×100, `filters` is a 10×5×5, and `output_data` is a 10×20×20 so instead we can write:

```
extern "C" __global__ void ConvolutionLayer(
    double input_data[100][100], double filters[10][5][5], double output_data[10][20][20]
)
```

Now if we want to access data in the arrays instead of having to do the relatively annoying and tedious manual 1D indexing we can get C to do the work for us by using the array indexes, so if you want to access the bottom right element in the filter of the 7th filter you just write `filters[6][4][4]` instead of `filters[6*25 + 4*5 + 4]`.