# GPU Implementation of Image Recognition Neural Network Architectures

Justin Garrigus
University of North Texas
National Science Foundation AI REU
Denton, Texas, United States
JustinGarrigus@my.unt.edu

Bora Bulut
Boston University
National Science Foundation AI REU
Boston, Massachusetts, United States
bbulut@bu.edu

***Abstract*—The neural network architectures VGG-16, LeNet, and Alexnet are each implemented in Python and C to compare relative performance when using a CPU-styled and GPU-styled approach to computation. This program uses a modular implementation, making it easy to analyze characteristics like performance bottlenecks or data organization in specific layers or using the GPU simulator GPGPU-Sim. The network is internalized as a list of individual layers, each with an optional list of weights and parameters used to modify input vectors to achieve an output. The C network is loaded from a custom file format generated by the Python program, which is built on the Keras API interface for Tensorflow. The final program accepts a jpg image as input, along with a network file and flag for computing on the CPU or GPU, and provides a list of predictions and percentages on what the network interprets the image to be, along with the time it took for each layer to complete their computation.**

***Keywords—neural network, GPU, VGG-16, LeNet, Alexnet, GPGPU-Sim.***

## I. Introduction

For the 2022 summer National Science Foundation AI REU, we worked to learn about neural networks and how they relate to Graphics Processing Units. To these ends, we developed a software that pulls the weights from trained models built in Python's Keras library, and uses those weights to predict the classification of an input image in the Cuda extension for C. Our final program was modular and easily scalable, allowing a wide array of models to be implemented; we will use the VGG-16, LeNet, and Alexnet architectures as demonstrations of this. This paper demonstrates our findings over this 10-week period, as well as the measured results of our completed program.

### *A. Neural Networks and Image Recognition*

A neural network is classification of program which is capable of learning how to accomplish a goal. It yields a matrix of data when given a matrix as input, and uses an elegant training algorithm to coerce a set of weights to modify inputs to become the intended outputs. Network "models" usually consist of a linear sequence of layers which range in style and purpose. Our program focuses on image-recognition, and networks attempt to predict what an input image consists of. Three networks are currently implemented, with more types being easy to implement; for prototyping purposes, we used Keras to implement our own versions of these networks.

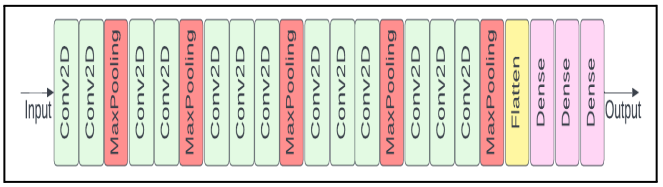


Figure 1: VGG-16 layers.

The main network architecture implemented was VGG-16 (Figure 1). This architecture was chosen first because Keras archived pre-trained weights for it (meaning we would not need to train the model ourselves), but it also offered an elegant solution to the image-recognition problem. It is built in an organized, well-structured way, and the variety of layers contain a breadth of computationally-heavy components we could focus on improving in the next step. Four layer types are introduced: Conv2D, MaxPooling, Flatten, and Dense.

This model accepts full-sized images as an input and assigns it a series of guesses based on 1000 possible labels based on the ImageNet dataset. Inputs are preprocessed by extracting their RGB components in the range [0, 255], subtracting the mean from each component, and switching the B and R fields.

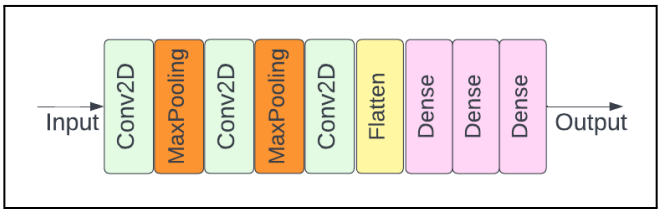


Figure 2: LeNet layers.

LeNet was the second architecture implemented due to the similarities it shared with VGG-16 (Figure 2). It does not introduce any new layers, and is much smaller in comparison. For this reason, it accepts 28x28 black and white images of handwritten digits from the MNIST digit dataset. Inputs are preprocessed by subtracting the mean from each color and dividing the result by the standard deviation of the set; therefore, the new mean of the input would be 0, and the new standard deviation would be 1. The inclusion of this model was meant to show the fluidity of the program, proving it can implement layers in any order and still yield a credible result.

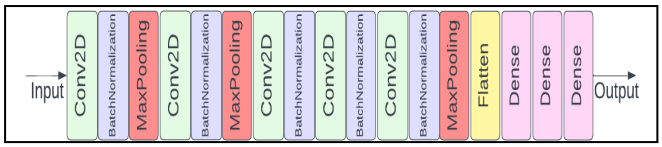

Figure 3: Alexnet layers.

The final architecture implemented was Alexnet, chosen due to its importance in the image-recognition field (Figure 3). While VGG-16 was considered to be a direct improvement of Alexnet on the basis of organization and accuracy, we still chose to incorporate Alexnet due to its complexity. It introduces a new layer named BatchNormalization, and many model hyperparameters have additional adjustments. Inputs are preprocessed in a similar manner to LeNet by subtracting the mean and dividing by the standard deviation of the set. This model was not included pre-trained in Keras, which required us manually train it on our own dataset; we chose to use the CIFAR-10 image dataset which contains 32x32 pixel images spread across ten classes out of storage concerns.

### *B. GPU, Cuda, and GPGPU-Sim*

Besides basic implementation of network architectures, we also sought to explore improvements to be made in terms of GPU programming. The GPU is similar to the CPU in many ways, but the largest difference lies in the number of threads that can be simultaneously executed. Typically, a standard CPU-centric program is limited by the number of cores present on the hardware; for instance, a dual-core CPU typically can only run two sets of instructions at a time. More threads can be created (the Windows operating system has several thousand running at any moment), but the CPU accomplishes this by switching its "focus" between each thread. To summarize, a CPU can only simultaneously execute a small number of instructions at any moment.

Alternatively, the GPU specializes in running a much larger amount of threads, each executing their own instructions at the same time. The GPU is specifically optimized for executing the *same* instruction on different sets of data giving it the name SIMD: single-instruction, multiple data. The GPU is capable of running different instructions per-thread, but performance is optimal if each thread is synchronized. This concept is immensely useful, for example, in matrix multiplication.

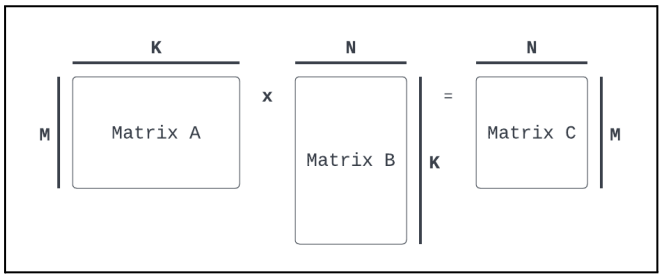

Figure 4: Matrix multiplication between Matrix A (with dimensions MxK) and Matrix B (with dimensions KxN) resulting in Matrix C (with dimensions MxN).

Matrix multiplication is a relatively repetitive process, featuring a dot-product between a row and a column, where:

$$C_{ij} = A_i \bullet B_j \qquad (1)$$

This equation is applied for each cell in the C matrix. For instance, if matrix C is 500x500 cells large, then 25000 dot product operations are performed. The CPU would accomplish this task by executing one operation at a time, using several for-loops to go through each cell in the resulting matrix and choosing rows and columns from the inputs to calculate dot products of. As such, the time complexity of a matrix is $O(M * N * K)$.

Since GPUs can quickly dispatch a large number of threads, a different approach can be taken to solving the problem: instead of looping through each cell in the final matrix, we can create one thread per cell, and have each concurrently calculate a single dot product operation. Assuming that there are enough threads available, then the new time complexity is $O(K)$ since a single loop is required to calculate the dot product of a cell. This is a tremendous improvement; increasing the M or N dimensions of the input matrices has no bearing on the execution time since the time it takes to start threads is constant regardless of the number of threads started. While this is only a simple example of the benefit of GPU programming, there are many other diverse circumstances they can be applied.

While the usage of GPUs are not limited to matrices, an additional optimization can be applied in regards to matrix multiplication specifically. Ceratin Nvidia GPUs come equipped with a device called a Tensorcore, which are optimized to calculate the result of the matrix operation:

$$D = A \times B + C \qquad (2)$$

There are several constraints that must be applied (the dimensions of the input matrices must be a multiple of 16, and each matrix must consist of 16-bit floating point values), but the result is staggering performance far exceeding that of the CPU or GPU methods discussed previously.

Cuda offers an API for programming GPUs. It is built on top of C, and can be executed from a native C program with the help of some compiler tricks. Additionally, a separate program named GPGPU-Sim is used to simulate GPUs or to profile different GPU architectures to identify bottlenecks, cache hit/miss rates, execution time, and more when used with a Cuda program. These tools were all used to develop the program and ensure it performed the best it could.

## II. Motivation and Objectives

Neural networks feature a paradigm similar to what GPUs favor; layers typically consist of a repetitive loop of instructions being applied across a set of data. For this reason, we wanted to research how much of a benefit could be gained by shifting a CPU-centric image-prediction algorithm over to the GPU.

There were several other goals we had in mind when planning for this project. We wanted our program to be general enough to work for different types of network architectures to improve its benefit; because of our modular approach to implementation (where networks are internalized as a collection of independent layers), it is very easy to execute and profile a unique network so long as the model is sequential and the layer type has an implementation.

Additionally, since the program is split into clear easy-to-read subsections, parsing important data using GPGPU-Sim is much easier than it would be with other existing neural network related libraries like cuBLAS or

cuDNN. The simple yet straightforward approach to program organization offers an easy way to identify areas to improve, apply changes, and profile the specific sections they apply. This is something that would be difficult to do otherwise.

Finally, we chose to develop the majority of the program in C as opposed to strictly in Python so we can compare cross-language performance as well. Keras takes a very high-level approach to deep-learning, intentionally obscuring the intricacies of network design and reducing complex algorithms down to simple invokable functions. This idea would be beneficial to most programmers wanting to quickly create and prototype network designs, but we wanted to take a deeper approach to AI research, diving into the instruction-level details on layer implementation. To these ends, our approach took a much more comprehensive look at creating fast, space-efficient programs for GPUs.

## III. Methodology

Before we began planning for the project, we created a preliminary program which tested the potential performance benefits of utilizing the GPU. The program featured an artificial neural network (in other words, a network featuring only Dense layers) of varying size, and compared how quickly a CPU would complete the feedforward operation versus a GPU doing the same operation. The results of this (discussed in the results section) were promising, and prompted us to pursue the project idea further.

The next step was to create the Python backend. Our goal was to use Cuda for C, but we wanted to avoid training a network from scratch. This prompted us to use Keras since it had pre-trained models available for us to use, so we decided on extracting a model, saving it to a custom file format, and later on loading that file from C. We created this model save/load function first, but in the process determined that we should treat Python like a "sandbox" for testing implementations for specific layers. Since Python allows us to quickly prototype code, generalizing unimportant sections and catching bugs easier with a stack trace instead of C's core dump, it was logical to first create an example of our program in Python before doing a line-for-line translation over to C. Our approach was to incrementally replace Keras's version of the layers with our own version, piece-by-piece until the entire program consisted entirely of our own code with minimal usage of Tensorflow functions attached, and then doing the same from Python to C. After different designs and attempts, we were able to implement the four initial layers from VGG-16 (Figure 1):

- *Conv2D*: this layer operates by sliding a "kernel" consisting of scalar values across a subset of the input matrix, calculating the dot product of the scalars and input and placing the result in the output slide. This layer type introduced the biggest bottleneck to our program; some instances feature billions of floating-point operations (FLOPs).
- *MaxPooling*: these feature to coalesce and "summarize" the convolutional output by passing forward the maximum values to the next layer. These serve mainly to halve the size of the input matrix (for instance, layer 2 represents a 224x224 pixel image, while layer 4 is 112x112 pixels in dimensions).
- *Flatten*: this operation simply transforms the matrix input into a single-dimensional vector for the sake of the next layer type.
- *Dense*: this layer consists of a classic matrix multiplication and addition. It multiplies the input (a one-dimensional vector) against a matrix of weights, and adds another matrix of weights to the result. Dense layers also introduce a considerable bottleneck, as the matrix multiplication occurs between matrices thousands of elements wide.

In our custom implementation of each layer, we needed to be sure an equivalent C implementation was possible: this required we remove all references to code that would be difficult to translate. The only libraries we allowed ourselves to use in this section were Numpy for their usage of the *ndarray* data structure; this had a close comparison to C arrays and Python did not have an easily-accessible array format naturally. Although, some complications were later introduced from this decision, which will be discussed later.

Due to several factors, the Python implementation tended to execute very slowly. This is largely due to the design of the language itself, introducing hurdles that make programming easier but slow the execution considerably. One method we utilized to prototype the program quicker was a "scaled probability" approach: we considered how many operations we could do in a given time period, and consistently performed that many operations at a maximum. For instance, if we could perform 10 operations/second, while wanting to spend no more than 3 seconds calculating each layer, and the first convolutional layer had 80 operations, then the resulting probability was:

$$p = (rate * time)/operations$$

$$p = (10 * 3)/80 = 0.375$$

This example probability would mean that we would only calculate 37.5% of the total operations in order to compute the layer in 3 seconds. Every time we encounter an operation, we would poll a random value and compare it with the probability; if the random value is less than the probability, we would go ahead with calculation; if the value was greater than the probability, we would use the *actual* value Keras generated for the layer. In this case, this would mean 37.5% of the output is a result of our own original calculations, while the other 62.5% of the output consists of values we know are already correct. This time-saving measure proved to be extremely effective at prototyping the Python implementation quickly: due to the nature of neural networks, small changes in values lead to potentially big effects, meaning we would immediately notice if our attempt at coding a layer was incorrect.

After the Python version of these layers were finished, we created an identical program in C which loads an example image, loads the weights of the pre-trained network, constructs the model from the weights, supplied the input image to the first layer, performed each layer's respective operation passing the output of one layer to the input of the next, and parsed which labels the network predicted the input image to represent.

As mentioned earlier, we used some functions from Numpy in order to implement the Python version of the program; these functions would each need to be re-implemented in C. The biggest accommodation to be

made was in storage: Numpy's *ndarray* structure was different from C in its ability to create arrays of arbitrary dimension, which could be emulated through the usage of unique functions we created for this project. As a result, we have a C implementation of the *ndarray* which contains methods for padding multidimensional arrays, copying, indexing, and creating iterators.

After the C program yielded a sufficient degree of accuracy for an image prediction, we moved on to re-implementing certain layers in Cuda. The performance-critical sections were the Dense and Conv2D layers: optimizing Dense layers would involve creating a fast matrix multiplication algorithm which we already discussed previously, but Conv2D would involve a new, but similar, GPU-based algorithm. The idea was to have one thread run per cell in the output array; a cell is yielded by finding the dot product of the scalar "kernel" with a subsection of the inputs, so this operation would run several thousand times all at once with the GPU as opposed to sequentially as it would be on the CPU. Although, a new problem introduced itself at this stage: since the GPU works best when each thread is executing the same instruction, it became important to pad the inputs to the convolutional layer to prevent the usage of divergence-inducing conditionals.

In the GPU, threads can only be dispatched in groups of at most 1024; a group of threads in this context is called a *block*. Also, each block run must have the same number of threads in it. Since each thread should calculate exactly one output value, then the number of threads should exactly equal the number of outputs, which in our case meant it must be aligned with a width of 1024. Although, most of the layers failed to meet this restriction, which required either the usage of conditionals in the GPU function (have each thread check if their position was out of bounds), or by padding the input. Padding was seen as a more reasonable option performance-wise: if the output size of a convolutional layer was, for instance, 4000, we would simply round the number up to the nearest multiple of 1024, which in this case is 4096. Those extra 96 threads would therefore successfully operate on data, but the data would be unused in any following step in the GPU.

At this point, the GPU version of VGG-16 was completed. The rest of the project was spent repeating the same steps for LeNet and Alexnet; since these architectures were both structurally very similar to VGG-16, it took considerably less time to do, and demonstrated the flexibility of the program in its ability to reuse code. Although, Alexnet introduced a final unique layer type which needed to be added:

- *BatchNormalization*: this layer essentially performs the preprocessing step again but makes the mean and standard deviation be weighted values instead of 0 and 1 respectively. It finds the mean and standard deviation of the inputs but it adjusts them by parameterized weights. Then, the layer subtracts each value by the adjusted mean, and divides the result by the adjusted standard deviation. The result of this operation is that the output layer has a mean and standard deviation equal to the values the layer specified during training.
- *Dropout*: this layer type was only included in the Python version. The original Alexnet paper specified the usage of a Dropout layer to help in the training step; it works by disabling random sets of inputs in order to make the training a bit faster, and it does not have any operation during normal inference. Since we did not focus on training for the C version of the project, this layer was omitted.

Finally, once each network was completed and tested, it was time to implement the improved matrix multiplication with Tensorcores. This would introduce a small improvement to only the Dense layer, since it was the only layer which explicitly contained matrix multiplication, but it was an improvement nonetheless. Nvidia released an example test file, *cudaTensorCoreGemm.cu*, which implemented a matrix-multiplication algorithm using Tensorcores, so our first job was to strip that example of its unimportant parts and to wrap it in an easy-to-invoke function. The biggest concern was that of copying: in order to simplify invocation of these functions, it was necessary to assume data had a consistent location. GPU and CPU data are separated, and transfer between the GPU and CPU takes a considerable amount of time, so this causes a problem that would need to be addressed in the future. In the final program, gigabytes of weights are transferred before each invocation of the matrix multiply function which stalls the image prediction by a huge amount, but the matrix multiply operation itself does see a slight performance improvement as expected.

## IV. Results

It was initially mentioned that before the program began, we ran a series of preliminary tests to profile potential performance gains in using GPUs versus CPUs in regards to neural networks.

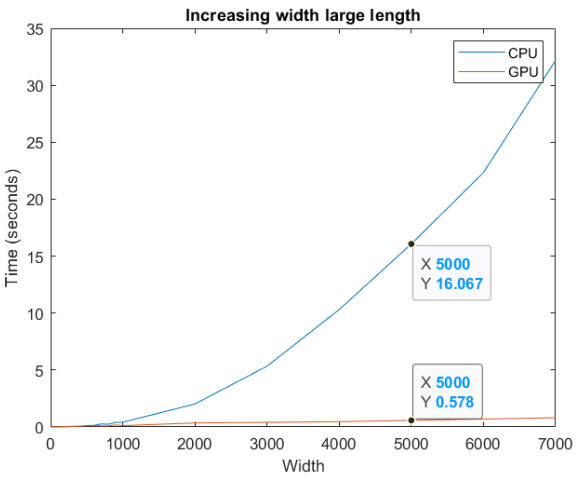


Figure 5: A graph displaying the performance of a network with many dense layers. The execution time on the CPU at different widths is in blue, contrasted with the execution time on the GPU in red.

Figure 5 displays how GPUs are much better suited than CPUs for networks with a large number of nodes. The highlighted point says that a network with many layers in it, with each layer containing 5000 nodes, completes a feedforward operation in 16.067 seconds on the CPU and 0.578 seconds on the GPU. As the number of nodes in any given layer increases, the computation time increases exponentially for the CPU while it increases very slowly linearly for the GPU. These results were very promising for the rest of the project; it suggested that a mass-thread method of problem solving was much more effective at

solving repetitive problems than a multidimensional for-loop based approach. We would refer to this example more in the future when we developed our network design.

After programming the project and verifying that the predictions of the input images matched the expected results from Keras, we executed each version of the project at each stage of completion and compared their execution time in generating a set of predictions. These versions are made of the Python version, whose implementation was an original recreation of Keras; the C version, which was a line-for-line recreation of the Python version; the Cuda version, which replaced the Dense and Conv2D functions with GPU-performant versions; and the Tensorcore version, which replaced the Dense GPU function with an even better GPU-performant version. The program was timed based on the layer operation itself, and the timing does not factor in any setup or cleanup routines that immediately precede or follow the layer operation. For instance, it was mentioned previously that Tensorcores require allocating several gigabytes of memory before invoking any function: for our purposes, we did not include this allocation time and only included the time of the matrix multiplication itself because an ideal program would be designed to avoid any dynamic allocation at prediction-time. The following table demonstrates the times retrieved from each program execution:

| **VGG-16** | ***Prediction Time (s)*** | ***Speedup*** |
|---|---|---|
| Python (CPU) | 12633.41 | —- |
| C (CPU) | 1328.21 | 89.49 |
| Cuda (GPU) | 2.68 | 99.98 |
| Tensorcore (GPU) | 2.45 | 99.98 |

Table 1: Performance comparison between different implementations of VGG-16.

| **LeNet** | ***Prediction Time (s)*** | ***Decrease (%)*** |
|---|---|---|
| Python (CPU) | 0.56 | —- |
| C (CPU) | 0.35 | 37.50 |
| Cuda (GPU) | 0.30 | 47.43 |
| Tensorcore (GPU) | 0.33 | 41.07 |

Table 2: Performance comparison between different implementations of LeNet.

| **Alexnet** | ***Prediction Time (s)*** | ***Decrease (%)*** |
|---|---|---|
| Python (CPU) | 1371.79 | —- |
| C (CPU) | 150.89 | 89.00 |
| Cuda (GPU) | 1.05 | 99.92 |
| Tensorcore (GPU) | 1.01 | 99.93 |

Table 3: Performance comparison between different implementations of Alexnet.

As expected, the GPU versions vastly outperform the CPU variants to a substantial degree. Although, it is important to note the timing differences between the C and Python version as well: remember that the only difference between those versions was their programming language, as the code itself was a near line-for-line recreation. This demonstrates that Python is not the best language to use in performance-critical situations, and that a well-designed C program will almost certainly out-perform an equivalent Python program. Although, even more impressive is the time difference between C and Cuda. An image prediction would take 20 minutes on a device that used only the CPU while taking about two seconds with the GPU enabled. Finally, there is a tiny performance gain in using the Tensorcores, which may still be optimized further by recognizing areas for improvement in the Nvidia sample file. It's important to note how LeNet doesn't see that large of a benefit from GPU usage: there is a small overhead to starting processes on the GPU and utilizing Tensorcores, and the small size of LeNet may not justify its usage. Regardless, when it comes to networks that are large in size, GPUs offer a decisive advantage.

## V. Conclusion

The GPU is clearly the better option to use when it comes to processing a lot of data in a repetitive way. This harkens back to the reason GPUs were made in the first place: to quickly solve equations related to graphics and mapping textures to the screen. Hardware designers were presented with a problem involving a large number of inputs and needed to present a solution that would uniformly apply code across large volumes of data, and the GPU was the best solution they came up with. For our purposes, we haven't used the GPU for graphics-processing at all since our implementation of neural networks is applied in a much more general sense, but the same principles apply. The program we created can be used in many more diverse areas than just image recognition, so long as it can consist of the layers we've presented so far. Additionally, layers are represented as a collection of weights and a pointer to a function; as long as new layers follow this format as well, then the usability of the program can expand even further. Finally, the readability of the code and the design of the program would make it very easy to do further research into areas of improvements for neural networks: for instance, if we wanted to analyze how performance would improve if the size of the data was modified or if the alignment of the data was changed, then the program coupled with GPGPU-Sim would be a great fit for diagnostics.

## VI. Further Studies

There are several areas we will be improving the project in the future. Firstly, the program in its current state copies a significant amount of data between the GPU and the CPU during a single image prediction. Ideally, the only data transfer that should occur should happen at the start of the program (to load the weights to the GPU), at the start of image prediction (to load the image to the GPU), and at the end of image prediction (to transfer the output of the network back to the CPU). This would take a lot more organizational effort to achieve, but it is certainly possible if the time is put in.

Alongside this, another improvement to be made would be in the diversity of models. The three architectures shown off all follow a sequential format, where data travels in a straight line from layer to layer. Some network architectures like GoogLeNet do not follow this format, and thus would

be impossible to represent in the current program. To these ends, it would be beneficial to extend the capabilities of the program to support these new networks so that new bottlenecks can be identified in these layer designs using GPU simulators.

Finally, Tensorcores are another aspect of the project that can be improved. We so far explored how Dense layers can easily utilize matrix multiplication, but the Conv2D layers can also be visualized as a matrix multiplication itself, so long as the data is formatted correctly. Convolutional layers currently cost the most amount of time in the program, meaning even a small optimization could improve the execution time drastically, but more testing should be done to verify this.