# GPU Implementation of Image Recognition Neural Network Architectures

Justin Garrigus, Bora Bulut, Hui Zhao

Department of Computer Science

Neural network models typically consist of a large volume of data transformed with a repetitive set of instructions, which sometimes results in poor performance on serial devices like CPUs. Alternatively, GPUs are specialized to dispatch many threads which can each process a block of data simultaneously. We compared the performance of three image recognition architectures when implemented on the CPU versus the GPU and created a modular program which can be used in conjunction with GPU simulators to profile different low-level aspects of the execution. Our results demonstrated a tremendous improvement in speed when networks are implemented on the GPU compared with the CPU.

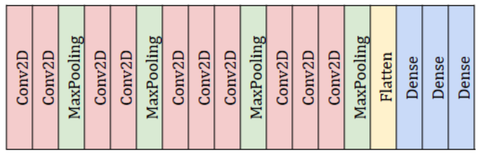


Figure 1: Layer configuration of the VGG-16 image recognition neural network architecture.

## Motivation

We wanted to observe the efficiency of GPUs and learn ways to optimize neural network-related computations through concurrent programming techniques. Networks implemented on the CPU waste a considerable amount of resources processing data sequentially and single-threaded. Due to the GPU's ability to quickly create tens of thousands of threads at once, with each thread executing instructions on their own partition of data, it became apparent of the potential speed improvement that could be gained. Coupled with a simulator of a GPU named GPGPU-Sim, we can profile an execution of our model as well. Our objective was to compare the performance of three image recognition networks: VGG-16, Alexnet, and LeNet.

## Matrix Multiplication

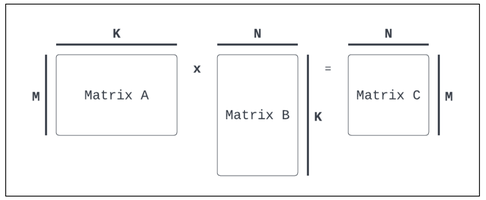


Figure 2: Standard matrix multiplication. Matrix A (with dimensions MxK) is multiplied with matrix B (with dimensions KxN) to yield matrix C (with dimensions MxN).

- A cell in matrix C is the dot product of a row from matrix A and a column from matrix B:

$$C_{ij} = A_i \cdot B_j$$

- A single-threaded approach would be $O(M * N * K)$ time complexity. Three for-loops would be required:

```
for i in range(M):
    for j in range(N):
        C[i,j] = 0
        for k in range(K):
            C[i,j] += A[i,k] * B[k,j]
```

Figure 3: Example algorithm to multiply two matrices.

- A different algorithm on the GPU would create one thread per output cell in matrix C. Each thread would calculate a single dot product concurrently. The outer two for-loops above can be replaced with multithreading.
- A multi-threaded approach would be $O(K)$ since the length of the dot-product vectors are the only factor that contribute to time. Increasing *M* or *N* would create more threads at no expense to time.

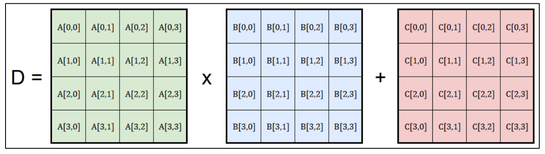


Figure 4: Another approach to processing matrices. Through Tensorcores, matrices are multiplied and accumulated concurrently by splitting cells into chunks.

## Results

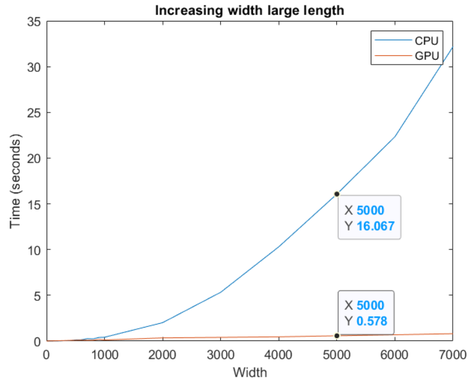


Figure 5: A fully-connected network with many layers and 5000 nodes in each layer takes 16.067 seconds to complete on the CPU and only 0.578 seconds to complete on the GPU.

For each of the network architectures, four versions were created and compared: one version was created in Python, one in C, one with convolutional and fully connected layers recreated with Cuda (a GPU programming interface), and one with fully connected layers utilizing Tensorcores.

| Model comparison | VGG-16 (s) | LeNet (s) | Alexnet (s) |
|---|---|---|---|
| Python (CPU) | 12633.41 | 0.56 | 1371.79 |
| C (CPU) | 1328.21 | 0.35 | 150.89 |
| Cuda (GPU) | 2.68 | 0.30 | 1.05 |
| Tensorcore (GPU) | 2.45 | 0.33 | 1.01 |

Figure 6: The duration of a single image prediction for each model implementation. Python is the slowest, while hardware-accelerated C through the usage of Tensorcores is generally the quickest.

## Conclusion

- There's a small overhead involved when creating threads on the GPU, so performance improvement is only evident when the size of the network is large.
- Threads on a GPU are capable of executing their own unique instructions, but synchronized threads each executing the same instruction yields the fastest execution speed.

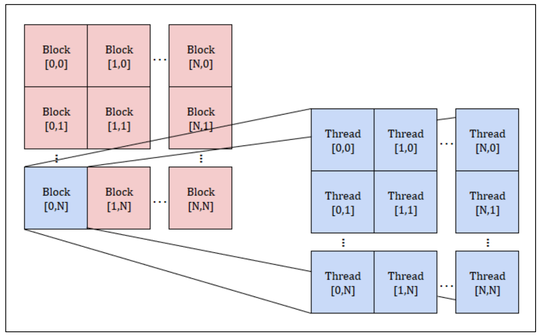


Figure 7: GPU layout. Threads are organized into blocks, which are each organized in a grid.

- Padding is sometimes necessary to ensure conditionals are removed from the GPU code to help satisfy the previous constraint. An arbitrary number of threads cannot be created, they must be aligned across blocks.
- When all constraints are met, GPU-optimized code **vastly outperforms** CPU-only code in relation to programs with **large amounts of data processed in repetitive ways**.

## Future Work

- Our program should be updated to accommodate for more diverse model types (namely models which are non-linear or recursive).
- Additionally, further optimizations can be made to improve data movement; copying data between the CPU and GPU incurs a significant time penalty.