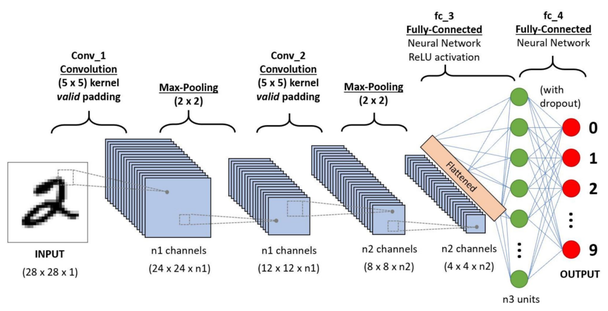
Parallel Implementations of Computer Neural Networks

Jacob Nelson (jacobnelson2018@gmail.com)
Shrutee Rakshit (shruteerakshit@oakland.edu)
ApREECE 2021
Faculty Mentor: Daniel Llamocca

Introduction

Convolutional Networks can be used to classify digital images through a series of mathematical calculations in successive layers. Given the large amount of computations and the large input datasets, this is a suitable candidate for exploiting parallelism. We explore the use of Intel Threading Building Blocks (TBB) for efficient multi-threading implementations with the goal of speeding up processing time. We investigate i) the optimal combination of TBB functions in order to speed up the CNN computations, and ii) the circumstances under which certain combination of TBB functions are better than others (e.g.: CNN architecture, input size). We provide speed-up results that were collected by running our implementations on a variety of computers.

Design



Images processed through a Convolutional Neural Network (Figure 1) go through the Convolutional Layers and the Fully Connected Layers (FCN).

Figure 1) (Image from: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53)

Parallelization was applied with TBB (Thread Building Blocks). It allows multiple tasks to run concurrently. We applied TBB to both the FCN and the CNN, which required some modification of the original C++ sequential code.

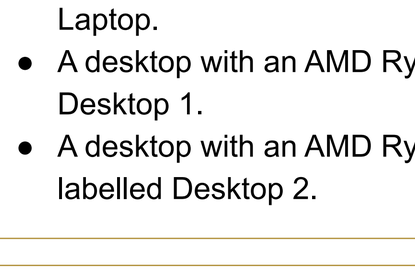


Figure 2 depicts how parallelization is achieved through dividing tasks. TBB algorithms were implemented to both the FCN and CNN.

Figure 2) (Image from: Intel Corporation)

The TBB algorithm parallel_for divides a collection of iterations into tasks, which can then run at the same time.

The TBB algorithm parallel_reduce splits a collection into multiple subranges, the constraint being, however, that the applied operation is associative.

In both the FCN and CNN, parallel_for was applied where accuracy was not affected, and parallel_reduce was applied to a dot product computation.

The FCN and CNN sequential and parallel code were tested on 4 separate machines:

- An Altera DE2i 150 with an Intel Atom N2600 (2 cores/4 threads), labelled Embedded Kit.
- A Dell XPS 13 with an Intel i7-8550u (4 cores/8 threads), labelled Laptop.
- A desktop with an AMD Ryzen 3 3100 (4 cores/8 threads), labelled Desktop 1.
- A desktop with an AMD Ryzen 9 5950x (16 cores/32 threads), labelled Desktop 2.

Results

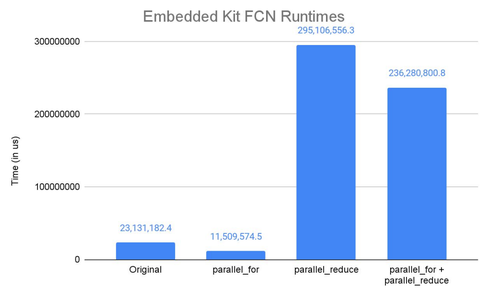


Figure 3A)

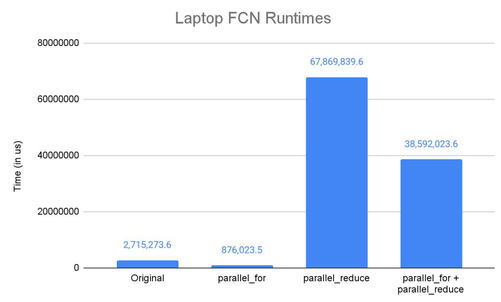


Figure 3B)

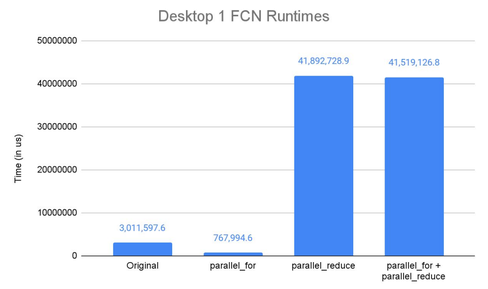


Figure 3C)

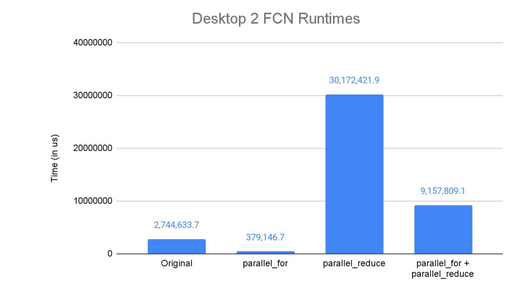


Figure 3D)

Figures 3A, 3B, 3C, and 3D represent the testing data from the FCN. Compared to the sequential code, here we see speedups across the board with parallel_for, but slowdowns from both parallel_reduce as well as parallel_reduce + parallel_for. parallel_for saw speedups from 2.00x on the Embedded Kit to 7.24x on Desktop 2. parallel_reduce saw slowdowns ranging from 10.99x on Desktop 2 to 77.47x on the Laptop. This reduction in speed is because the dot product calculations were simple, and the overhead to utilize parallel_reduce was greater than the potential speedups. Parallel_reduce + parallel_for saw slowdowns ranging from 3.34x on Desktop 2 to 44.05x on the Laptop.

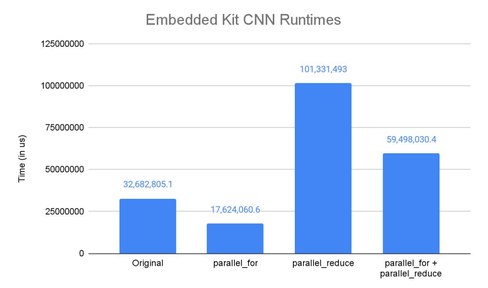


Figure 4A)

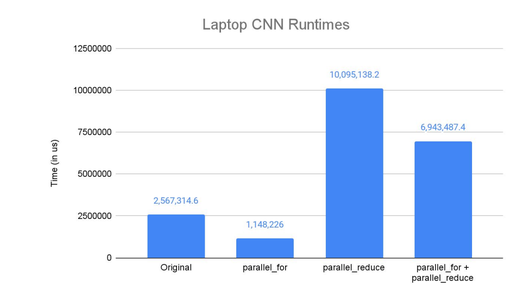


Figure 4B)

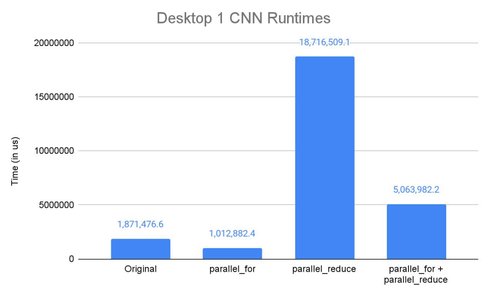


Figure 4C)

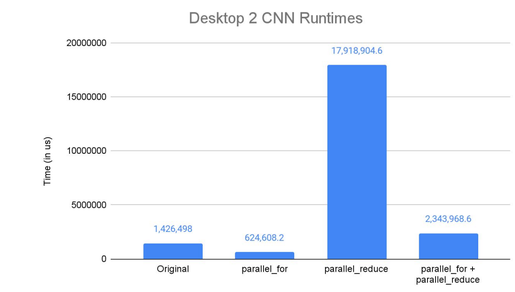


Figure 4D)

Figures 4A, 4B, 4C, and 4D represent the testing data from the CNN. Similar to the FCN, compared to the sequential code we see a speedup with parallel_for, and slowdowns with parallel_reduce and parallel_reduce + parallel_for. parallel_for saw speedups from 1.85x on Desktop 1 to 2.28x on Desktop 2. parallel_reduce saw slowdowns ranging from 3.10x on the Embedded Kit to 12.56x on Desktop 2. Here again the slowdowns are due to the dot product calculations being too simple of a workload to properly utilize parallel_reduce. parallel_reduce + parallel_for saw slowdowns ranging from 1.64x on Desktop 2 to 2.71x on Desktop 1.

Conclusion

The parallel_for algorithm in FCN and CNN resulted in overall speedup.

The parallel_reduce algorithm in both the FCN and CNN resulted in an overall slowdown.

The combination of parallel_for and parallel_reduce in both the FCN and CNN also slowed down execution time, however not as much as parallel_reduce alone.

Speedup of execution time would assist in applications such as AI or vehicle image processing.

This program was supported by the National Science Foundation Grant No EEC-1659650. Additional support was provided by the School of Engineering and Computer Science at Oakland University.