Convolutional Neural Network w/ Intel Thread Building Blocks

ECE 5772/4772

List of Authors (Joshua Duncan, Matthew Irvine)
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: joshuaduncan@oakland.edu, Matthewirvine@oakland.edu

***Abstract*— The Focus of this project is to enhance a convolutional neural network (CNN) using intel's thread building block library (TBB) to improve processing efficiency. The integration of TBB into CNN architectures can achieve dynamic task scheduling and better utilization of multi-core processors. All in all, TBB integration in CNNs offers significant advancements in neural network speed and efficiency, suggesting potential applications in broader neural network models and encouraging further development of adaptive algorithms for multi-core machine learning environments.**

I. Introduction

Convolutional Neural Networks (CNNs) have emerged as a cornerstone in the field of machine learning, offering robust solutions in diverse applications ranging from handwriting recognition to medical diagnoses and autonomous vehicle navigation. This project delves into the integration of CNNs with thread building blocks (TBB), aiming to harness their power for more efficient parallel processing in handling complex and large datasets. The motivation behind this project stems from the growing need for faster and more efficient computational capabilities in neural networks. As CNNs become increasingly complex and are tasked with processing vast amounts of data, the demand for reducing computation time without compromising accuracy has become paramount. This is where TBB comes into play. By facilitating dynamic task scheduling and optimized utilization of multi-core processors, TBB promises to significantly enhance the computational performance of CNNs.

This will cover the implementation of a CNN using the MNIST database, which consists of 10,000 training samples. The designed CNN aims to precisely interpret processed images, informing the user of the network's accuracy in recognizing the input. A critical aspect of this study is to compare the computation times of the traditional sequential CNN implementation with the TBB-enhanced version. This comparison is vital in demonstrating the effectiveness of TBB in improving the efficiency of neural networks.

From an educational perspective, this project encapsulates key concepts learned in class, such as 2D image convolution, artificial neural networks, and the usage of TBB to speed up CPU intensive programs. Additionally, it allowed for self-directed learning, particularly in the realm of parallel computing and the practical application of TBB in neural networks. The ensuing report is structured as an expansion of these introductory themes, delving deeper into the technicalities of CNNs, the role and implementation of TBB, and the comparative analysis of the different implementations.

II. Methodology

A. Convolutional Layers

The foundation of the described Convolutional Neural Network (CNN) architecture begins with the meticulous processing of the 28x28 pixel images from the MNIST database, which contains a substantial set of 10,000 images of handwritten digits. These images are stored in a binary input file (bif) format, serving as the primary input map for the first layer of the network. It is at this stage that the CNN commences its critical task of feature extraction from the basic pixel data presented by each image.

Within this layer, six unique feature maps are employed, each associated with a dedicated 3x3 convolutional kernel. These kernels are carefully designed to capture specific attributes from the input images through narrow 2D convolutions. The application of these kernels transforms the original 28x28 input into six new 26x26 feature maps, effectively highlighting various aspects of the input data that are essential for pattern recognition.

Subsequently, a bias term is introduced to each of these feature maps, followed by the application of the Rectified Linear Unit (ReLU) activation function to every pixel within them. The ReLU activation is a critical component in the architecture, as it introduces the necessary nonlinearity to the model by setting all negative inputs to zero. This nonlinearity is what enables the network to interpret and learn the complex patterns inherent in the varied and nuanced shapes of handwritten digits.

The process culminates with a max pooling operation, which selectively condenses the feature maps down to a 13x13 resolution. By retaining only the paramount features, max pooling not only simplifies the information within the feature maps, but also ensures that the resulting output captures the essence of the input image. This step is instrumental in reducing the dimensionality of the data, thereby streamlining the network for the subsequent layers, which are tasked with extracting increasingly abstract features. The integration of the bias term across this process is of particular importance, as it guarantees the activation of neurons even in instances where certain features may not be present, thus providing the network with adaptability and enhanced generalization capabilities.

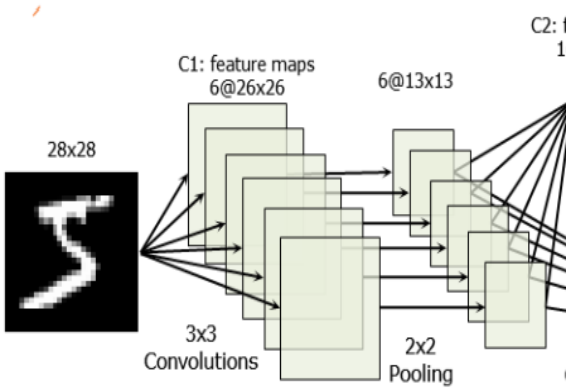

Figure 1. First Layer Architecture

Building upon the first layer of the CNN, the second layer further refines the feature extraction process. It takes the six input maps – which are the outputs from the first layer – and expands the network's depth by generating 16 feature maps. This increase in depth is achieved through the use of an intricate array of kernels; specifically, each of the 16 feature maps is produced by convolving the six input maps with a corresponding set of six 3x3 kernels. This means a total of 96 distinct kernels are utilized at this stage, each contributing to the extraction of intricate features, resulting in feature maps of size 11x11.

Once the convolution process is complete, the information from each input map is amalgamated, summing the convolutions to form a single comprehensive feature map. This summing process is key to integrating the various detected features into a unified representation. Following convolution, each pixel within these feature maps is processed by adding a bias term and then passed through a Rectified Linear Unit (ReLU) activation function. This step is crucial for introducing nonlinearity to the network, enabling the model to learn complex patterns and dependencies in the data.

The final step in the second layer is the application of max pooling, which aggressively downsamples each feature map to a size of 5x5. This operation retains only the most prominent features from the preceding convolutions, thereby reducing the dimensionality and computational complexity of the network. It also provides the added benefit of making the network's learned features invariant to minor translations, aiding in the robustness of pattern recognition.

This second layer is pivotal in the CNN architecture, as it builds on the primary features extracted in the initial layer and begins to form more abstract representations of the input data that are crucial for the network's ability to perform complex image recognition tasks.

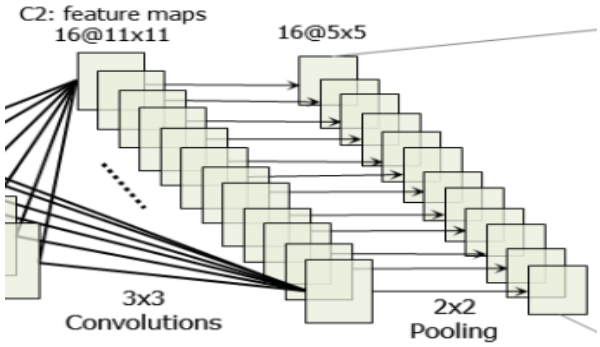

Figure 2. Second Layer Architecture

B. Fully connected network

The fully connected network in the described Convolutional Neural Network (CNN) architecture consists of three layers, each playing a pivotal role in pattern classification. The first layer of this network is a dense layer that interfaces with the feature maps produced by the preceding convolutional and pooling layers. It has 400 inputs, which correspond to the flattened output of the previous layer, and it provides 120 outputs. This layer is responsible for interpreting the features extracted by the convolutional layers and beginning to synthesize this information to identify more complex patterns.

Transitioning to the second layer of the fully connected network, it receives the 120 outputs from the first layer as its inputs and provides 84 outputs. The significant reduction in dimensionality from the first to the second layer reflects a further consolidation of information, as the network focuses on the most salient features that are critical for classification.

The third and final layer of the fully connected network serves as the decision-making component of the CNN. This layer takes the 84 inputs from the second layer and narrows them down to 10 outputs. These outputs typically correspond to the class scores for a classification task, which, in the case of the MNIST dataset, represent the ten possible digits (0 through 9) that the network is attempting to recognize.

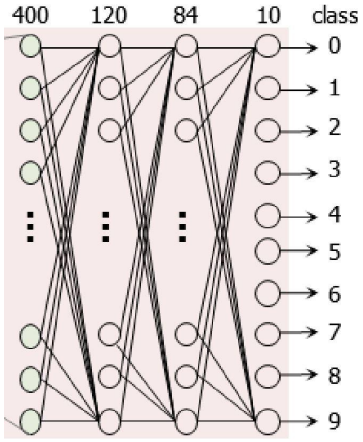

Figure 3. Fully Connected Network Architecture

Each layer within this fully connected network possesses its own set of distinct weights and biases. These parameters are learned during the training process and are essential for the network's ability to make accurate predictions. Weights determine the influence of each input value on the output, and biases allow the network to adjust the output independently of the weighted sum. The training process involves adjusting these weights and biases to minimize the difference between the network's predictions and the actual target values.

The architecture of the fully connected network is critical as it serves to interpret the high-level features extracted by the convolutional layers and translate them into a final prediction. The progression from a high number of inputs in the first layer to a small number of outputs in the last layer reflects the network's process of abstracting raw image data into an understandable classification.

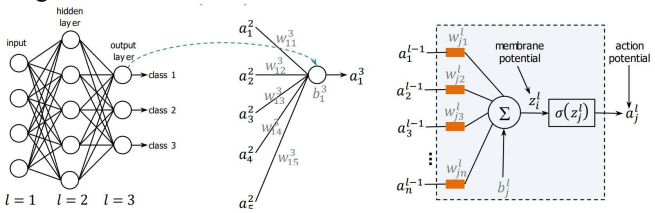

Figure 4. Fully Connected Neural Network Concept

III. Experimental Setup

In this project setup, the Intel Atom N2600 board was utilized, chosen for its computational capability suitable for running the Convolutional Neural Network (CNN). This board, running Ubuntu Linux, provided a stable and efficient platform for executing the C++ code that forms the backbone of theCNN.

A validation process was structured to include an extensive testing phase, where the CNN's predictions were compared against a predefined dataset containing the expected results. This dataset served as the ground truth for the MNIST images the CNN was tasked to classify. Accuracy measurements were taken across 10,000 iterations to ensure statistical significance and reliability of the performance metrics.

IV. Opportunities for parallelization

There are multiple opportunities for parallelization when using intel's TBB library. This project constructed 5 different implementations of the CNN – one of which is the basic implementation discussed earlier in order to gauge how well the CNN would perform when using different parallel techniques.

A. Parallel_for

In addressing the computational challenges of processing a large dataset, particularly one consisting of 28x28 pixel images across 10,000 examples, the implementation of parallel computing techniques is essential. The parallel_for construct plays a pivotal role in this context, effectively preventing the linear increase in computation time characteristic of sequential processing.

The benefits of parallel_for are especially evident in the second layer of our Convolutional Neural Network (CNN), where the generation of 16 feature maps imposes a substantial computational burden. By employing parallelism, one can distribute this workload across multiple cores, thereby significantly accelerating the processing speed. This approach is scalable; as the number of feature maps increases, 'parallel for' can seamlessly handle the intensified workload without necessitating major changes to the codebase.

B. Parallel_reduce

Another function used is the parallel_reduce function. As stated previously, during the second convolutional layer, there is a point at which multiple convolutions will need to to be performed and then summed. This is where the Parallel_Reduce function can be taken advantage of. The purpose of this function is to optimize associative functions such as taking the sum, average, maximum, or performing Boolean Operations.

C. Parallel_pipeline

The next implementation that was investigated was parallel pipelining. The parallel_pipeline architecture that was implemented was divided into three distinct stages, each designed to process different components of the CNN in a concurrent manner.

The first stage is pivotal, ensuring a steady stream of image data is available, feeding subsequent stages without delay from the processing of previous images. This

continuity is crucial, as it eliminates idle time, optimizing the pipeline's operational flow.

Subsequent to image loading, the convolutional layers come into play during the second stage. Here parallelization is leveraged to expedite the workflow, allowing for the simultaneous processing of multiple images, thereby extracting primary features with increased efficiency.

The final stage of this pipeline is the fully connected network stage, which synthesizes the high-level features delineated by the convolutional layers to formulate the final prediction.

In an effort to refine the parallel pipelining approach, the three-stage pipeline was expanded into a four-stage construct. This iteration maintains the original pipeline's structure but segregates the convolutional layers into two separate stages. The modified second stage now singularly manages the first convolutional layer, aiming for a more balanced computational distribution and augmented parallelism.

V. Results

A. *Accuracy measurement*

The expected result of this thorough testing regimen was to ascertain a 96.97% accuracy in the CNN's predictions, proving the model's effectiveness in image recognition tasks. The accuracy metric would be a reflection of the CNN's ability to generalize from the training data and correctly classify new, unseen images.

B. *Sequential Implementation*

The first implementation of the CNN model recorded a processing time of 35.5 seconds. This metric is significant as it provides a baseline for the computational demand of the network, particularly highlighting the second layer's intensive calculations.

The second layer, due to its convolutional operations involving multiple feature maps and kernels, is inherently the most computationally intensive part of the CNN. This layer's performance is a critical factor in the overall execution time, as it engages in a high volume of matrix multiplications and nonlinear activation functions, which are resource-demanding processes. In total for each of the 10,000 images, the program creates 16 feature maps. These maps come from multiplying each of the 6 original images by kernels, and then summing up the result. So suffice it to say, there is a great deal of computing in the second layer.

Therefore, the time of 35.5 seconds is reflective of the computational workload required by the second layer to perform feature extraction and transformation across the entire dataset. This understanding is essential for identifying potential optimization opportunities in subsequent implementations of the network.

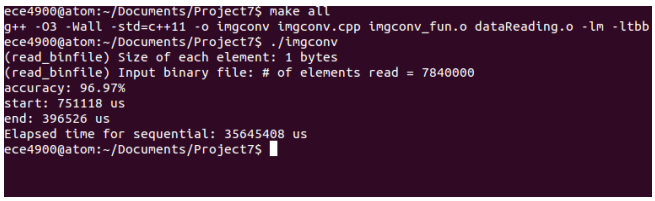

Figure 5. Sequential

C. *Parallel_For implementation*

Through the application of parallel processing, a 2.28-fold increase in speed was achieved, effectively cutting the computation time in half compared to a sequential approach. This improvement underscores the effectiveness of parallelism in enhancing the performance of CNNs, particularly when dealing with large datasets where the demand for computational resources is high.

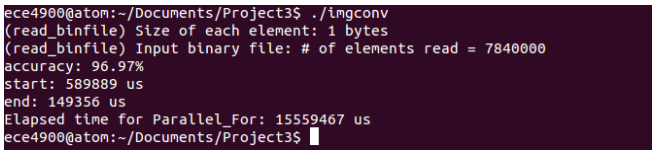

Figure 6. Parallel_For

D. *Parallel_Reduce implementation*

The parallel_reduce function was used to assist with the summation of the six different matrices. For all sixteen maps, six different convolutions on an image, then after all of these are summed, the data is sent to the reduction class and summed. The result of this implementation is a time of 19.17 seconds which is a significant increase over the sequential. That being said, this implementation is also built off of the parallel_for implementation, meaning that by adding the parallel_reduce, the program actually slows down by almost 4 seconds. There is room for improvement here however. One way would be to remove the parallel_for from the second layer being removed. This in theory should cause a small increase in speed. More elements could also be sent to the parallel_reduce class at the same time, meaning that the overall time needed for parallel_reduce would be decreased as it would have to be used less. The images being sent are only 11x11 pixels in size, so many of them are needed to make parallel_reduce worth using.

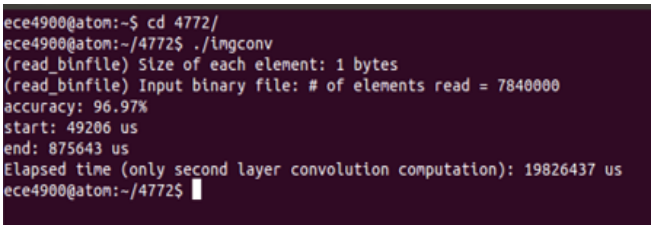

Figure 7. Parallel_Reduce

E. Parallel_Pipeline implementations

In the previous parallel implementations, impressive speed up was achieved relative to the sequential CNN. However, when the computation times were assessed, the results were intriguing. The three-stage pipeline showed a marginal slowdown, approximately one second longer than the parallel_for implementation. The four-stage pipeline extended this delay slightly further. These increases in computation time are likely due to the overhead that multi-stage pipelines introduce. Each additional stage brings synchronization points and data transfers that, despite the benefits of concurrent processing, introduce latency.

Despite these findings, the structured nature of the parallel pipeline holds promise for scalability. It provides a better framework that has the potential to surpass the flat parallel_for architecture, particularly as the complexity and volume of processing tasks escalate. This exploratory study emphasizes the nuanced balance between theoretical and empirical performance in parallel computing architectures.

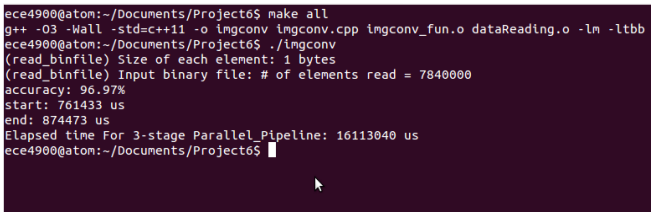

Figure 9. 3-stage Parallel_Pipeline

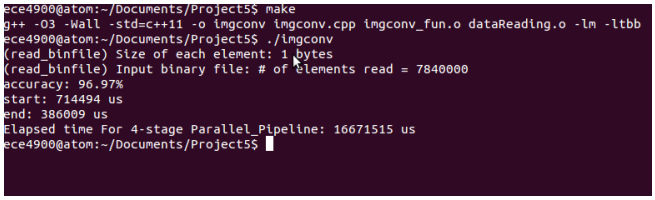

Figure 10. 4-stage Parallel_Pipeline

CONCLUSIONS

Convolutional Neural networks are powerful tools for deep learning applications. Input data is able to be processed with a variety of filters to parse defining features. They are then able to use those key features to determine the optimal choice depending on the input. Their modular structure allows them to be used in a variety of applications such as image processing, but can also be applied in other situations where the data is in a grid-like structure.

Despite their usefulness, these systems come at an intense cost which lies in the processing power needed in order to perform these calculations. This is where parallel processing could be used to speed up the computation. Parallel processing can be applied to many places in a CNN. Being that the convolutional layers need to process a large amount of data, these will be the most beneficial to parallelize.

During this project, it was found that a significant amount of time can be saved with parallel processing. Different methods can be used to do this. The parallel_for function of intel TBB was found to be the most efficient in this case. It was applied over top of both of the first two convolutional layers which allowed for a massive increase in speed. The other functions tried were parallel_reduce and parallel_pipeline. While less effective than the parallel_for implementation, these still resulted in a large increase in speed over performing the calculations sequentially.

REFERENCES

[1] D. Llamocca, *Reconfigurable Computing Research Laboratory.* [Online]. Available: http//www.secs.oakland.edu/~llamocca/index.html. [Accessed: 1-Nov-2023].

[2] R. Gonzalez, Deep Convolutional Neural Networks. [Online]. Available: https://ieeexplore.ieee.org/document/8496892 [Accesed: 15-Nov-2023]