Parallelising Convolutional Neural Networks using openMP and MPI

Naveen Bharadwaj

Summary

Machine learning is widely used to improve the efficiency of several applications with convolutional neural networks at the forefront of these gains. CNN's are used in Image Recognition, Instance Segmentation, Nuclei Detection in Medical AI, and several other applications. I have noticed that the main operation encompassing the CNN's is a 7 nested for- loop with a few mathematical operations carried out inside the loop. In this particular project, I plan to explore OpenMP and MPI methodologies to find opportunities in parallelising these CNN applications.

Timeline/Roadmap

- Understanding the applications of Convoluted Neural Networks in detail.
- To explore the opportunities presented by a Shared Memory Architecture to accelerate the CNN application process.
- To explore the opportunities presented by a Distributed Memory Architecture to accelerate the CNN application process.

Reference & Methodology

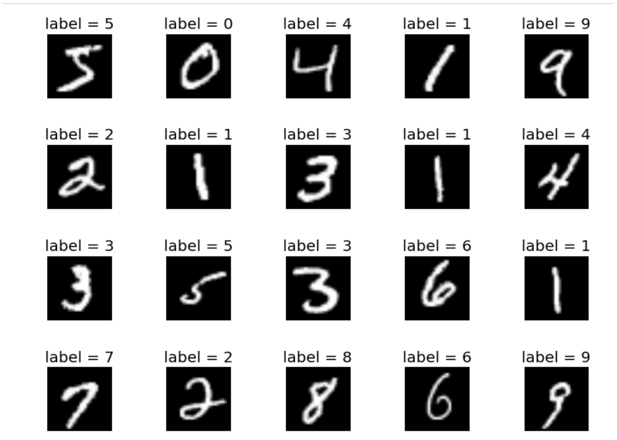
- **Programming language:** C++

An example of the 7-dimensional CNN loop nest:

```
for n = 1 to N
for k = 1 to K
 for c = 1 to C
for w = 1 to W
for h = 1 to H
for r = 1 to R
for s = 1 to S
out [n] [k] [w] [h] +=
 in [n] [c] [w+r-1] [h+s-1] * filter [k] [c] [r] [s];
```

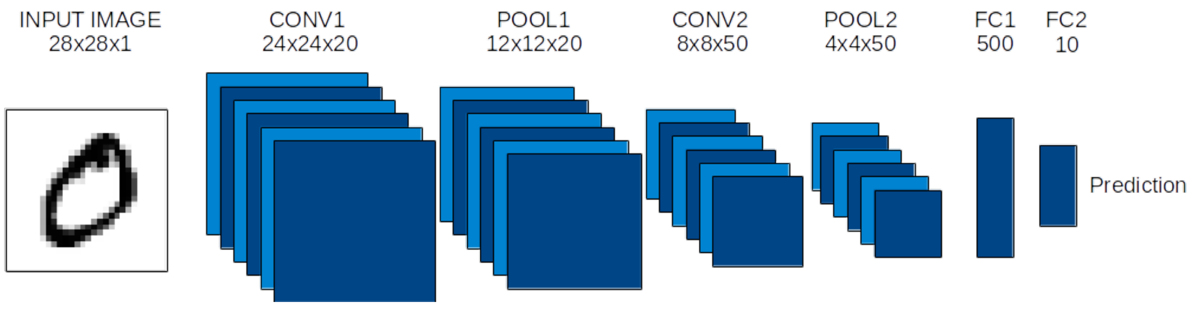
Introduction

A significant trend in high-performance and embedded applications is developing: many of the emerging high-performance and embedded applications, from image/video/audio recognition to automatic translation, business analytics, and all forms of robotics rely on machine learning techniques. Due to its inherent parallelism, GPUs have attracted a lot of attention, and is the flagship processor for CNN training and inference. For now, I have implemented the fast processing of inputs (feed-forward) rather than training (backward path) on the accelerator.



Sample of MNIST Dataset

Even though Deep and Convolutional Neural Networks come in various forms, they share enough properties that a generic formulation can be defined. In general, these algorithms are made of a (possibly large) number of layers; these layers are executed in sequence so they can be considered (and optimised) independently. Each layer usually contains several sub-layers called feature maps; we then use the terms input feature maps and output feature maps. Overall, there are three main kinds of layers: most of the hierarchy is composed of convolutional and pooling (also called sub-sampling) layers, and there is a classifier at the top of the network made of one or a few layers. The role of convolutional layers is to apply one or several local filters to data from the input (previous) layer. Thus, the connectivity between the input and output feature map is local instead of full. The role of pooling layers is to aggregate information among a set of neighbour input data. In the case of images again, it serves to retain only the salient features of an image within a given window and/or to do so at different scales. An important side effect of pooling layers is to reduce the feature map dimensions. Convolution and pooling layers are interleaved within deep hierarchies, and the top of the hierarchies is usually a classifier. This classifier can be linear or a multi-layer (often 2-layer) perceptron. Unlike convolutional or pooling layers, classifiers usually aggregate (flatten) all feature maps, so there is no notion of feature maps in classifier layers.



1. LeNet-5

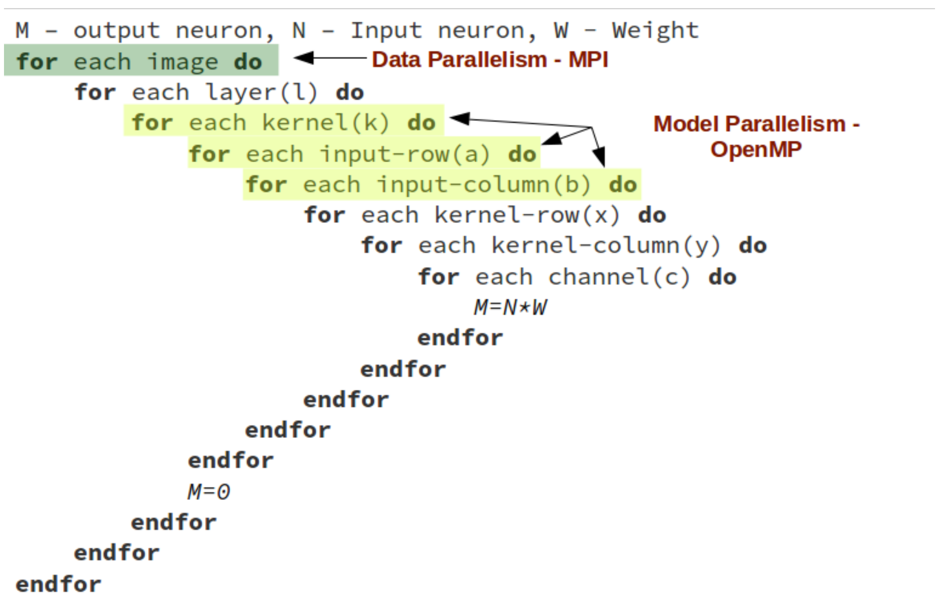
Implementation — OpenMP

Sequential execution of convolutional layer in a CNN is shown in 1. Each layer is executed sequentially. As kernels and channels constitute to most of the operations in convolution layers, it makes sense to parallelise these two first. Algorithm in 4 shows a way to parallelise. The first level OpenMP parallelism is done across the kernels. Table 1 shows the time taken in performing inference on 10,000 images sequentially as well as different types of OpenMP parallelisms. Column 3 of the table (Parallel Kernels) shows the improvement in performance by implementing parallelism across kernel. It can be observed that doing so gives us a speedup of more than 2 times. We then try to parallelise the channels. The channels are the inner most loop of all the nesting in our convolution algorithm. Also there is accumulation performed across the channels. So I added a reduction statement to the parallel for loop for accumulation. Column 4 (parallel kernel and channels) shows the time taken by such a method. It can be observed that parallelising both kernels and channels makes the algorithm perform worse than performing sequentially. The main reason for such a drop is false sharing.

```
M - output neuron, N - Input neuron, W - Weight
for each image do
    for each layer(l) do
        for each kernel(k) do
            for each input-row(a) do
                for each input-column(b) do
                    for each kernel-row(x) do
                        for each kernel-column(y) do
                            for each channel(c) do
                                $M=N*W$
                            endfor
                        endfor
                    endfor
                endfor
            endfor
            $M=0$
        endfor
    endfor
endfor
```

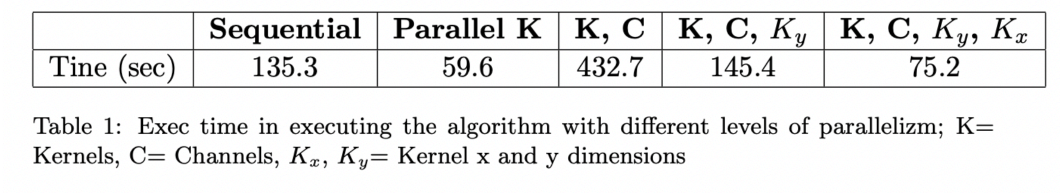
1. Pseudo-code for sequential implementation

The inner most loop has a sum += operator. When thread i writes data into sum, as a result of the += operator, the cache line holding that part of the sum becomes dirty. The cache coherency protocol then invalidates all copies of that cache line in other cores. Other cores now have to refresh their copy from the main memory of some deeper level of cache resulting in additional time. I tried to schedule the loop with a specific chunk size to avoid such a false sharing scenario. But as I have considered LeNet which doesn't have bigger channel sizes, such a method was not giving me any major improvement in performance. Another alternative was to parallelise the upper nested loop as well. Column 5 and 6 shows the improvement in performance if the reduction is performed across a much deeper level of loop. That is, the channel for loop and the above two for loops (kernel x and y dimensions) are parallelised using the collapse() operator. This gave me less scope of false sharing thereby giving better performance. But this improvement is not comparable to just parallelising kernels. Hence I stuck to parallelising kernel, and input feature maps for better performance improvements. But for layers with deeper channels, parallelising the channels with the reduction clause will give significant improvements in time taken.



```
M - output neuron, N - Input neuron, W - Weight
for each image do
    for each layer(l) do
        for each kernel(k) do
            for each input-row(a) do
                for each input-column(b) do
                    for each kernel-row(x) do
                        for each kernel-column(y) do
                            for each channel(c) do
                                $M=N*W$
                            endfor
                        endfor
                    endfor
                endfor
            endfor
            $M=0$
        endfor
    endfor
endfor
```

2. Pseudo-code for parallel implementation



	Sequential	Parallel K	K, C	K, C, K_y	K, C, K_y, K_x
Tine (sec)	135.3	59.6	432.7	145.4	75.2

Table 1: Exec time in executing the algorithm with different levels of parallelizm; K= Kernels, C= Channels, K_x, K_y= Kernel x and y dimensions

Table 1

Implementation — MPI

As it can be seen in the pseudocode in 2, there are multiple levels of parallelism that can be achieved while executing a convolutional neural network. As explained in the previous section, I have used OpenMP to parallelise processing of individual layers. A dataset like MNIST has tens of thousands of images to be processed. We make use of this characteristic of CNNs to parallelise it one step further – parallel execution of multiple images. This is referred to as Data Parallelism in 4. Each MPI process is allocated a bunch of images to process on the same model set. This enhances performance, as two different levels of parallelism are in place; model parallelism and data parallelism.

Results

The output of this program is attached along with the code file . MPI outperforms openMP significantly. Also attached with the file is the initial sequential code for reference. I have included four shell files as parameters to run the program. Also, I wasn't able to produce precise results owing to the machine's limited computational power. However, running the program on a well equipped server can yield relatively better results.

System Configuration

CPU : AMD Ryzen 7 5800X 8-Core Processor (16 threads)
GPU: NVIDIA 3070Ti
Memory: 16 GB