Parallelizing the Convolution Operation using MPI

Team Members: Nirmal Krishnan (nkrishn9) and Tanay Agarwal (tagarwa2)

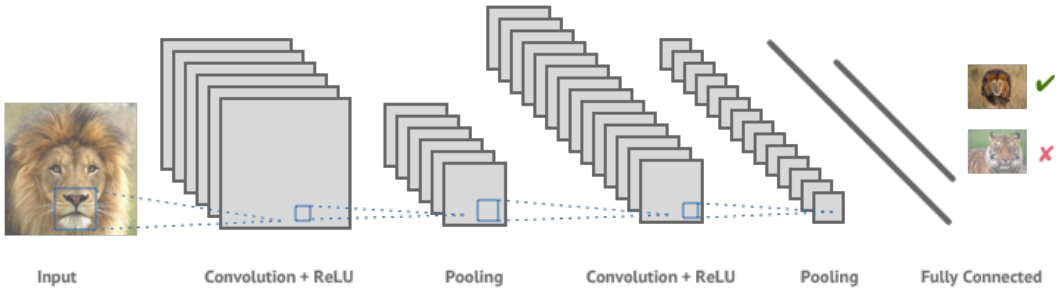
May 9, 2018

1 Introduction

1.1 Background

In the last decade, convolutional neural networks (CNN's) have become instrumental in solving a variety of classical computer vision problems such as image segmentation and identification [1]. The foundations for these networks, like the name indicates, are the convolutional layers. These layers consist of a set of learning filters, called convolutions.

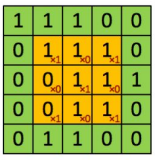
Figure 1: A typical CNN architecture



These convolution filters on an input image of dimensions 30 by 30 might be of size 3-by-3, the kernel size. During the forward pass of the CNN, "we slide (more precisely, convolve) [this 3 by 3] filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position" [2]. In the backwards pass of the model, local and upstream gradients are computed relative to the convolutional filters and their entries are updated using stochastic gradient descent [3].

Specifically, in the forward pass, the kernel convolves each element $A[i][j]$ of the input matrix by localizing the center-element of the kernel onto the element $A[i][j]$. The dot product is then taken between all the elements that the filter covers. For example, in the case of a 3-by-3 kernel, the convolution on the element $A[i][j]$ would involve a dot product with all the elements immediately surrounding $A[i][j]$:

Figure 2: Convolution filter



As you might notice, this can create problems when convolving elements near the edges of the input, because the kernel can't fit inside the bounds of the matrix. To solve this issue, CNNs use what is typically known as zero-padding. All matrix edges are padded with zero entries according to the kernel size, so that all elements of the input matrix can safely be convolved. Thus, the size of the output matrix is the same as that of the input.

Critical to understanding what a convolution filter does is the idea that knowledge is shared in image data. Pixels close to each other form larger abstractions, so the convolution accordingly operates on windows of close by pixels. This allows the deep learning model to generalize and abstract towards lines, shapes, and eventually objects [4].

1.2 Problem Statement/ Thesis

While CNN's are extremely powerful, their operations - especially for the convolution layers - can be computationally expensive. In this paper, we examine the benefits of parallelizing the forward pass of the convolution operation. We hypothesize that parallelizing this operation will result in a sub-linear, but significant performance speedup.

2 Methods

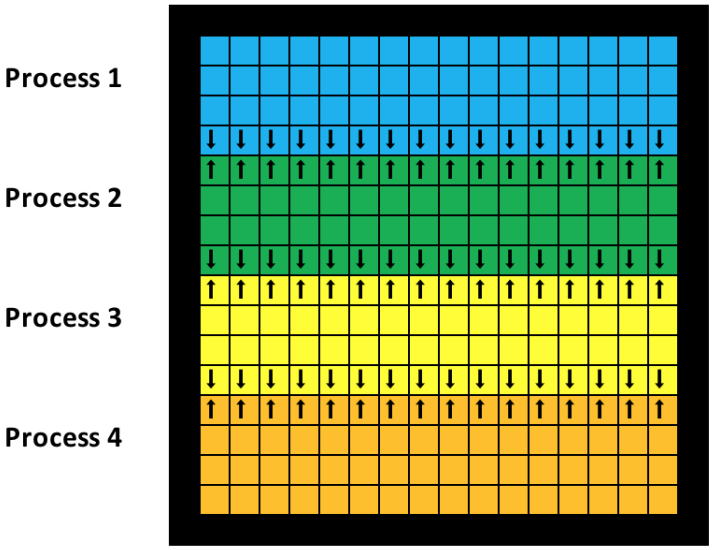
2.1 Algorithm Design

We implemented a deadlock-free approach for the forward-pass of the convolution filter. As discussed in the introduction, the convolution filter operates over the entire image based on the kernel defined by the user. Each convolution performed by the kernel is independent of the matrix elements that are not overlapped by the kernel at that moment in time. Therefore, this operation is obviously parallelizable by allowing different processes to manage different portions of the input image and then aggregating the results. In order to divide the image, we used a strip-based decomposition, where each process performs convolutions on different chunks of rows.

More specifically, we divide our input matrix such that each process is equally assigned a contiguous set of rows. Each process can calculate the convolution output for its subgrid independently of the other processes, except for its top and bottom rows. These rows depend on other processes because when the kernel is localized onto these row elements, it overlaps into a subgrid that is assigned to a different process. Therefore, we "pad" each subgrid with extra upper and lower rows. We then employ a message passing scheme (similar to that of Assignment 3) to update these pad row values so that the corresponding subgrid values can be populated correctly and completely [5]. The number of required pad rows depends on the kernel size: for example, we need only 1 upper and lower pad row for a 3-by-3 kernel, whereas we need 3 upper and lower pad rows for a 7-by-7 kernel.

Since each MPI node is enumerated with an ID, we use these to control our message passing. In order for our approach to be deadlock-free, every node that is sending a message must be paired with a node that is receiving at the same time. Therefore, we let the odd-numbered nodes send their actual top and bottom rows while the adjacent even-numbered nodes receive (and store this information in the pad rows). Then, we let the even-numbered nodes send their actual top and bottom rows while the odd-numbered nodes receive. This ensures that each send is paired with a receive, and that each process has all the information required to independently perform the convolution. Once all processes independently compute their subgrid outputs, this information is aggregated by the master 0th node to produce the final output of the convolution forward pass. The diagram below shows how the convolution operation is divided for a 16 by 16 image with a 3 by 3 kernel size and 4 processes.

Figure 3: Strip-based decomposition for convolution



2.2 Experimental Setup

In this study, we wrote our algorithm in C using the MPI framework for parallel design. Our experiments were tested on a c5.2x large AWS instance with 8 vCPUs and 16 GB of memory running Ubuntu 16.04 LTS.

To explore the effect of parallelization on performance, we tested different combinations of hyperparameters and measured speedup. Specifically, we varied input size, kernel size, and the number of forward passes (iterations). When varying a parameter, we keep the other two parameters fixed. These fixed values are 1024-by-1024, for the input size, 15-by-15 for the kernel size, and 1000 for the number of forward passes. For each combination of parameters, we measured runtime on 1, 2, 4, 8, and 16 processes to obtain speed-up charts (see below).

Runtime was measured using the unix bash function "time" which measures the entire lifecycle of the application, including startup. Each data point in the charts below was calculated using the median of 15 runs to ensure consistent and reliable results.

3 Results

Figure 4: Speedup with Varying Kernel Size

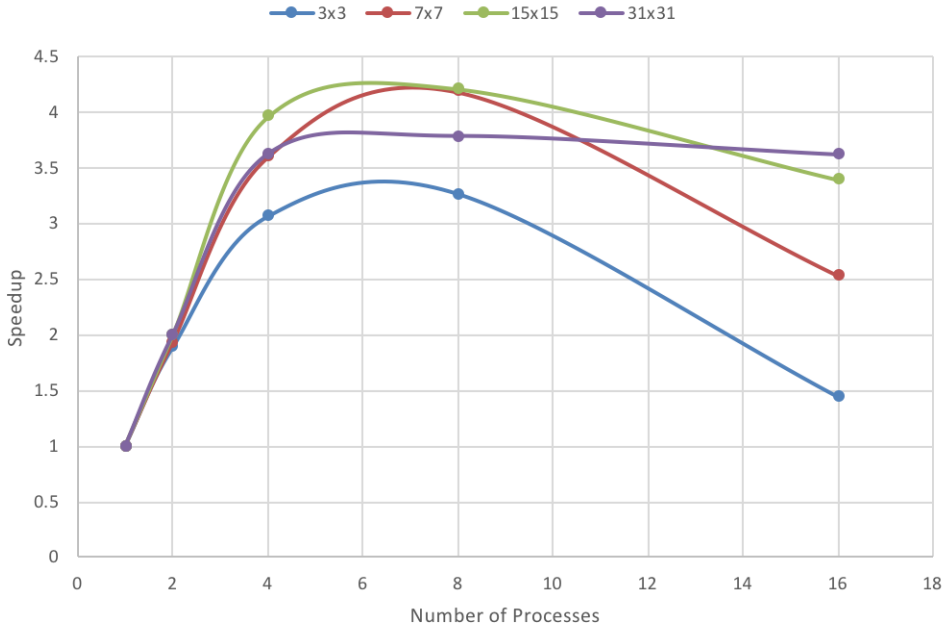


Figure 5: Speedup with Varying Input Size

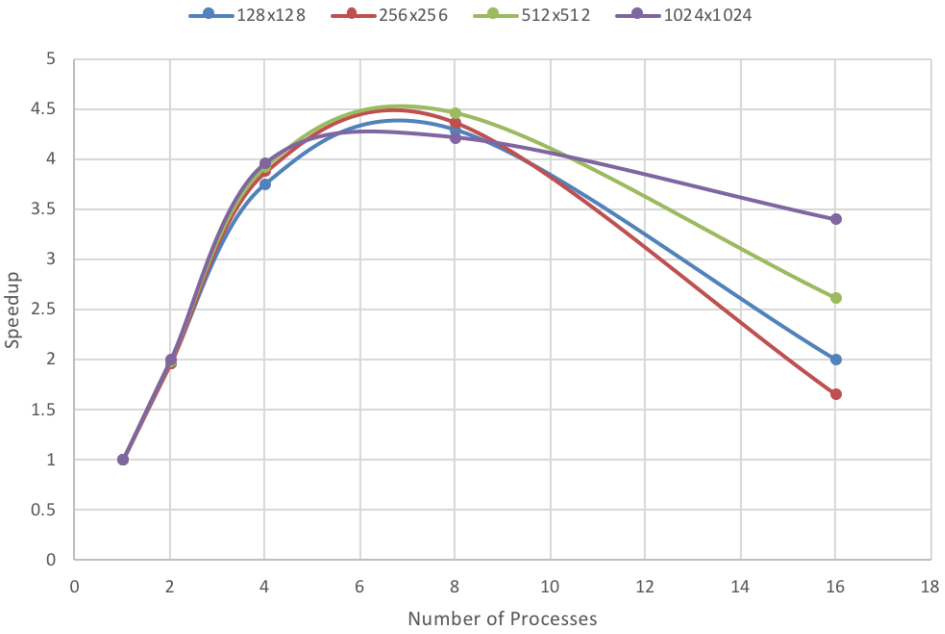
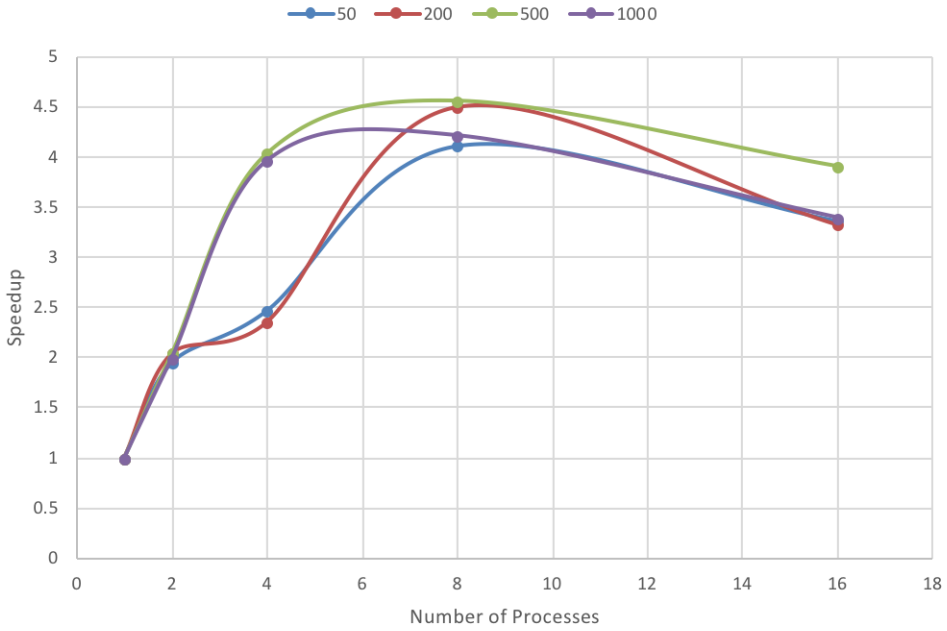


Figure 6: Speedup with Varying Number of Forward Passes



4 Discussion

In the results section, we see that all of our lines have the same basic shape. We see close to linear speedup from 1 to 4 processes, sub-linear speedup from 4 to 8, and then no/constant speedup from 8 to 16. This is because our experiments were tested on a machine with 8 vCPUs. Therefore, the speedup potentially gained from hyper-threading 8 to 16 processes is likely reduced due to the startup costs associated with that operation.

In figure 4, we can see a variety of kernel sizes being tested on an input size of 1024 by 1024 over 1000 forward passes. We see that the 3x3 kernel is least impacted by parallelization and the aforementioned speedup trends. It is well-known that larger tasks are easier to parallelize than smaller ones and since the 3x3 kernel convolution consists of many small tasks, this is likely what is occurring. A smaller kernel size means that the load on each process is lesser, which decreases the effectiveness of parallelization. On the larger kernel sizes, we see the consistent linear speedup from 1 to 4 processes and then sub-linear onwards.

In figure 5, we can see a variety of input sizes being tested on a kernel size of 15x15 over 1000 forward passes. We see typical results of linear speedup from 1 to 4 process, and sub-linear from 4 to 16. When 16 processes are used, we can more obviously see the effects of the input size on our speed-up results. We see that the larger input sizes perform significantly better at 16 processes. This is because the start-up costs associated with 16 processes are expensive; however, on the larger input sizes, these resources are more effectively being used. This shows that our parallelization scheme is more effective and beneficial when the task at hand is large.

In figure 6, we see a variable number of forward passes being tested on an input size of 1024 by 1024 with a 15x15 kernel. Like figure 4, we can effectively see the benefits of parallelization best when our tasks are large. On the 50 and 200 forward pass parameter settings, we see sub-linear speedup across all processes; however, for the larger parameter settings–500 and 1000– we see linear speedup from 1 to 4 processes and then sublinear from 4 to 16.

Based on our results, we can see that deadlock-free message passing - the principle of parallel computing being investigated in this project - is effective in distributing work among processes and improves performance considerably. We believe that the efficiency of our message passing scheme is a big part of why we get close to optimal speedup up to 4 cores. The size complexity of messages being passed per process per forward pass is $O(NK)$, where N is the input width and K is the kernel width. So, the total complexity of messages being passed per forward pass, $O(PNK)$, increases as the number of processes increases. So, it is imperative that these messages are sent and received efficiently.

Our practical results (shown above) complement the theoretical analyses and expectations of parallel performance for this task. Let N be the input width, K the kernel width, and P the number of processes. Based on our grid decomposition, we know that there are $O(K^2)$ convolution operations per input element, and each process gets $O(\frac{N(N+K)}{P})$ input elements, and so each process does $O(\frac{N^2K^2+NK^3}{P})$ operations per forward pass. So, we would clearly expect to get speedup since increasing the processes decreases the operations per process (and thus, runtime). Based on our practical results, this holds true up to 4-8 processes. Furthermore, our theoretical analysis also leads us to expect that kernel size will have a greater impact on speedup than input size, because it has a higher order of exponentiation. This is also confirmed by our practical results, as the speedup differs most when we vary kernel size.

5 Conclusion

Using the MPI framework, we were able to implement a deadlock-free algorithm that parallelizes the forward pass of convolutional operation in C. Our algorithm improves performance considerably and optimally when 4 processes are used on a machine with 8 vCPUs.

6 Contribution Statement

- Nirmal: Implementation of algorithm and write up.
- Tanay: Implementation of algorithm, trials, and write up.

7 References

1. Applications of Convolutional Neural Networks
2. Convolutional Neural Networks (CNNs / ConvNets)
3. Stochastic gradient descent
4. A Beginner's Guide To Understanding Convolutional Neural Networks
5. Cellular Automata in MPI