Bottleneck analysis of matrix multiplications and CNN inference

Abhishek Ranjane
Courant Institute of Mathematical Sciences,
New York University
arr7980@nyu.edu

Mohamed Zahran
Courant Institute of Mathematical Sciences,
New York University
mmz1@nyu.edu

Abstract. Parallel computing has become prevalent in all modern computing systems. We see the efficient implementation of parallel programs with tools such as OpenMP. However, the challenge with writing parallel code is writing code efficiently that scales well, and avoids bottlenecks. In this paper we implement two types of matrix multiplications - dot product and element-wise multiplication. In addition to that, we also implement a simulation of the inference done by Convolutional Neural Networks. We then induce various types of bottlenecks and analyze them with profiling tools to get useful insights.

Keywords: Bottleneck analysis, OpenMP, Matrix Multiplication, CNN Inference.

1 Introduction

Highly scaled Matrix multiplications are gaining importance since the advent of Deep Learning. Making the Parallelization of these programs more efficient can allow us to use deep learning more easily in practical applications. Thus, this paper analyses bottlenecks that can hinder the application of these programs and reveals interesting insights.

In this paper, 3 different types of programs are implemented using OpenMP. OpenMP is an API that provides programmers directives to parallelize programs. The first implementation parallelizes Matrix multiplication. Artificial bottlenecks are induced in it, such as critical regions and file I/O. Then the program is tested against different threads and at scale. The second implementation is a parallel implementation of element wise multiplication that has an induced bottleneck of not using the collapse optimization. Lastly, the project implements one layer of a *convolutional neural network.* This consists of piece-by-piece element-wise multiplication by sliding a fixed size kernel over the input matrix. The resultant products are summed and assigned to the resultant matrix.

The paper tests the program using profiling tools such as omp_get_wtime and gprof to get statistics on the program, such as function call graph, time spent on each function and most importantly, the speedup achieved by each program while changing the number of threads and scale of input.

The paper is Structured into 6 sections. Section 2 surveys the literature on this topic. Section 3 and 4 explain the proposed idea and how it was implemented. Section 5 analyzes the results of the profiling tools. Section 6 concludes the paper.

2 Literature Survey

2.1 Profiling

The research in the area of bottleneck analysis has been conducted by exploring and implementing different profiling tools such as LiMiT and Gprof[2]. OpenMP provides its own time measuring tools as well. OpenMP provides a tool omp_get_wtime which returns elapsed wall clock time in seconds. This has been a really good device to analyze bottlenecks as it is very easy to implement. However, with OpenMP, the program has to make system calls which affects the time required by the program. Gprof is a profiler that analyzes the Unix program and returns a call graph. This has allowed researchers to view exactly which function is throttling. However, the resulting data for Gprof is not exact, rather a statistical approximation.

2.2 Scaling Multi-threading applications

Scaling Multi-threaded applications is growing in importance as the scale of programs is increasing. We see the work of Jos´e A. Joao, et al. who have implemented a Bottleneck Identification and scheduling system [4]. This type of implementation is very useful in Asymmetric Chip Multi-Processor (ACMP). The BIS can identify and accelerate any type of bottleneck and increases as the number of cores increase.

The work of John Demme, et al., have analyzed bottlenecks using precise event counting on onchip performance counters. These tools provide interesting insights to parallelizing MySQL and Firefox, which were initially obscured.

2.3 Parallelizing inference for CNNs

Due to the growing popularity in the field of Deep Learning, CNNs and parallelizing CPU inference and parallelizing them has grown in importance. Most of the work for training and testing Deep Learning systems has been done on GPU, using the CUDA architecture due to their highly parallel processing capabilities. The work of Muhammadjon Musaev, et al.[5] focuses on accelerating the training of CNNs on CPUs. They also use OpenMP for parallelizing the training on images that range from 16x16 to

256x256. GPUs have proven to be far more superior to train neural nets in the past few years. However, in our work we focus on the inference side of implementing the CNN. This can be a cost effective measure and a highly versatile use-case for scenarios where GPUs cannot be accessed but CNN predictions are required.

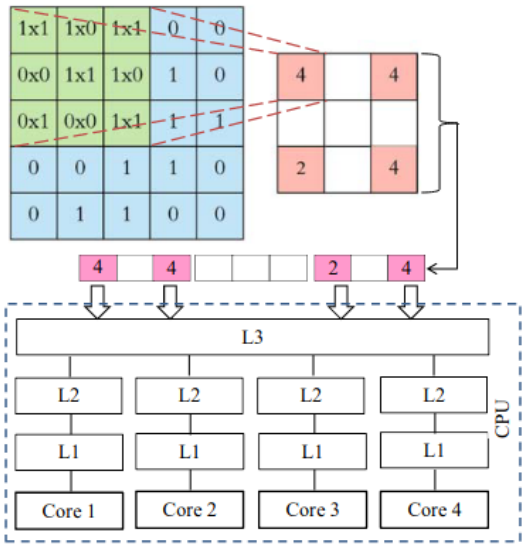

Fig. 1. Scheme of allocating threads on different CPU core

3 Proposed Idea

Due to the growing popularity and use of deep learning systems, there is a lot of value in identifying the bottlenecks in neural networks, so that we can accelerate it. Most neural nets are trained on GPUs which are better for large scale multiplications and mathematical operations. However, GPUs are an expensive resource. Thus, while they are irreplaceable for training neural nets, such as CNNs, we can train CNNs, save their weights and load them at inference time to make minimal predictions. Thus analyzing the performance of convolutional neural networks on CPUs can be very beneficial in deploying cheap and/or portable servers that are void of GPUs.

The idea with analyzing the bottlenecks for matrix multiplication was that most neural networks consist mainly of layers of matrix multiplications, followed by an activation layer and optionally followed by optimizing techniques such as Max-Pooling. Once the convolutional neural network is trained and the weights are loaded, these are the only operations that are required to make predictions. Thus, we are not concerned with calculating Loss and backpropagating that loss over the weights. Our goal was to simulate a model inference layer in C on a CPU using OpenMP and analyzing the bottlenecks.

3.1 Matrix dot product

The first program that we examine is the matrix dot product. This is a standard matrix multiplication that involves multiplying the i^{th} row of a matrix to the j^{th} column of another matrix and summing all the products to get the element in the i^{th} row and j^{th} column of the resulting matrix. Thus, it takes 3 nested for loop: 2 for the rows and columns and 1 for adding them up. We have 2 functions that we analyze. First, initialize_matrices() is used for filling matrices with values. Second, matmul() which does the actual matrix multiplication.

Both of these functions are parallelized using OpenMP. And we measure the timing and percentage of total time for each function.

There are 2 induced bottlenecks:

a. I/O overhead
b. Critical regions

We use a c program to generate numbers that fill in a text file. The I/O version of the program reads the numbers from this file and the initialize_matrices() function puts the values into the matrices in a parallelized fashion. The objective here is to observe if we see a significant difference in the speedup in the input version versus the normal one, where the matrices are assigned values in the program itself.

For the program with a critical region, we want to show that adding a critical region locks up frequently used resources and gives worse performance than the sequential version. Thus we place the matrix multiplication operation in the critical region to observe the difference.

3.2 Matrix element-wise product (Scalar product)

Element-wise matrix multiplication involves each element of the matrix multiplied by the corresponding element in the other matrix. It runs much faster than the dot product as there are only 2 nested for loops that are parallelized. There are 2 different types of programs in the implementation. First implementation involves simple symmetric matrices that work similar to the dot product implementation. Second implementation involves matrices having different numbers of rows and columns.

The objective for the element-wise implementation was to observe the difference between collapsing and not collapsing the for loops. OpenMP provides the collapsing functionality, which can be used to distribute the threads over more than one for loop. This functionality can be leveraged here effectively because the nested loop is not dependent on any outer loop. A similar example is also demonstrated with the neural network implementation.

In the collapsed version we use collapse(2) which divides the threads equally over the nested loop. The proposed idea here is that if the value of P (number of rows) are very less, say P=5, and the number of columns N are very high, that is if the matrix is narrow, then upon usage of more than 5 threads, all the remaining threads will go to waste. Thus

collapsing should prevent that.

3.3 Convolutional Neural Network layer

A Convolutional Neural Network layer consists of matrix multiplication, activation, and optional optimizations such as max-pool or batch-norm. For the matrix multiplication there is a kernel of fixed size that accounts for the weights of the neural network. Ideally these weights are trained by back-propagation on a GPU and saved. The kernel is multiplied by each section of the input matrix as it slides across the matrix piece by piece. For the purpose of this simulation, a stride of 1 is selected, that is, the kernel slides by 1 number each time. The values of all the products are summed and assigned to the respective resultant matrix cell.

In the implementation for the Convolutional Neural Network layer there are 3 functions that carry out the task:

1. matmul(): This function does element-wise multiplication for each kernel by the input matrix, sums up all the values and assigns it to the respective resultant matrix cell. There are 4 loops nested here in total: 2 responsible for sliding the kernel over the input matrix and 2 for the multiplication itself.
2. initialize_matrices(): loads in the values for the matrices and the kernel. This is again optimized by using OpenMP for.
3. ReLU(): This is an activation function that simply checks if the values are greater than 0. If they are below 0, then they are assigned 0.

There are 3 bottlenecks induced here:

a. I/O overhead
b. Non-collapsing
c. Inefficient Algorithm

Usually the models are not going to be trained on a CPU. Thus, model weights are saved into a file and loaded when inference is needed to be done. This can be a huge overhead as there are often millions of parameters in novel architectures. Thus loading them is a potential bottleneck that can be optimized.
In the element-wise multiplication it is necessary to utilize the collapse functionality efficiently. It is easy to write code such that the inner loop has variables depending on the outer loops. The behavior of the collapse() functionality is unspecified. Thus, it might not be correct. This implementation includes an inefficient as well as efficient way to write the loops such that their independence contributes to the efficiency of the program.

4 Experimental Setup

All the experiments have been conducted and tested remotely on the NYU crunchy1 computer. Specifications include: Four AMD Opteron 6272 (2.1 GHz) (64 cores) CPU, 256 GB memory, OS: CentOS 7.

For compiling the program, terminal command:

```
gcc -g -pg -Wall -fopenmp -o neural neural.c -lm
```

5 Results and Analysis

5.1 Matrix Multiplication (dot product)

By Implementing the dot product efficiently and optimally we saw significant improvement in the speedup as the number of parallel threads increases and as much Amdahl's Law permits. The speed went up to 29.36 in case of 64 threads.

For the critical region we observed a significant decrease in performance. The the code with the critical region was slower than the sequential implementation.

For the file I/O overhead induced version, we observed that the file I/O is indeed a significant bottleneck. The bar graph in fig.2 shows that as the number of threads increases, the time spent on actual multiplication decreases. This implies that more and more percentage of time was spent on getting the input from the file.

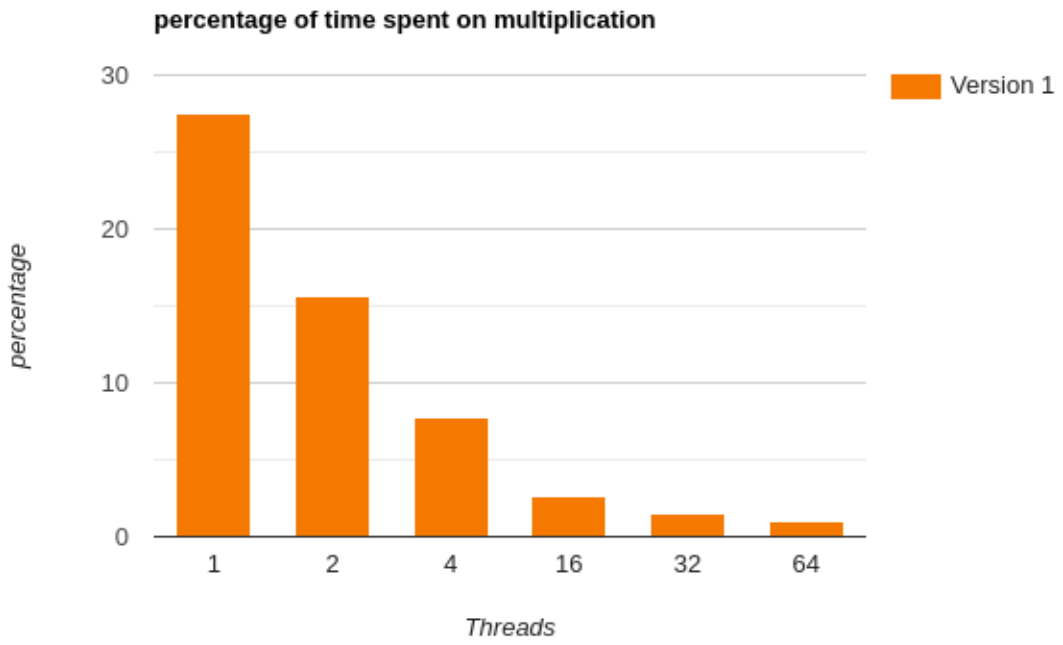

fig.2

From fig.2, we observed that I/O impacts the overall execution time. Thus for highly scaled parallel systems, I/O poses as a huge bottleneck for tasks such as matrix multiplication

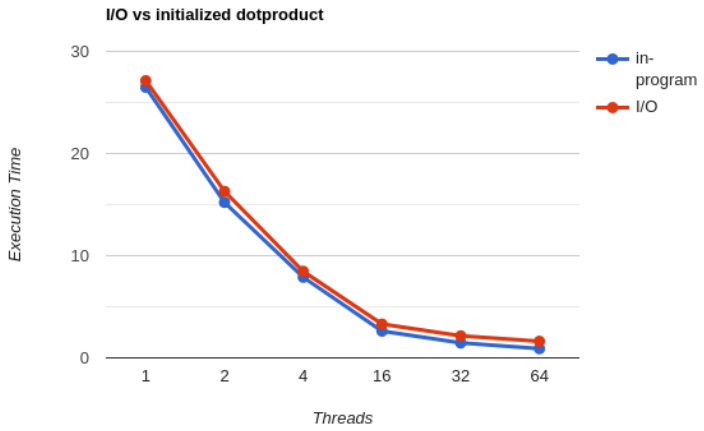

fig.3

5.2 Element-wise Matrix multiplication

By implementing element wise multiplication optimally we observed a good speedup as the number of threads increased. The speedup goes up to 11.43 in case of 64 threads and increases consistently with the number of threads, and as the problem size increases. This implies that the program was well parallelized and can scale well.

For the induced bottleneck, which is removing the collapse clause, we can observe that the speedup stops after 4 threads. This is because after 10 threads (number of iterations of the outermost loop) the remaining threads are not able to contribute anymore.

Thus, by using the collapse clause, we observed that in the case of a narrow input matrix (in our case: 10x10000000) we cannot see significant speedups. Therefore, writing code that does not distribute over all nested loops can be a significant bottleneck in case of element-wise matrix multiplication.

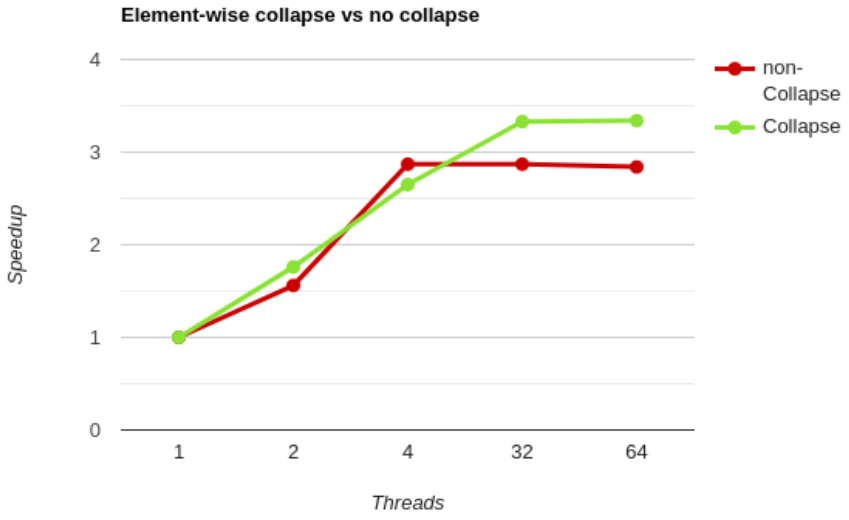

fig.4

5.3 Convolutional Neural Network layer

By implementing the optimal code, we see significant improvements in performance as the number of threads increase and as the size of the problem increases.

For element-wise piece by piece multiplication and the sum of all products it is important to note that as the size of the kernel k increases, the number of iterations of the outer loops decreases. Thus, not having the collapse clause acts as a severe bottleneck for large values of k.

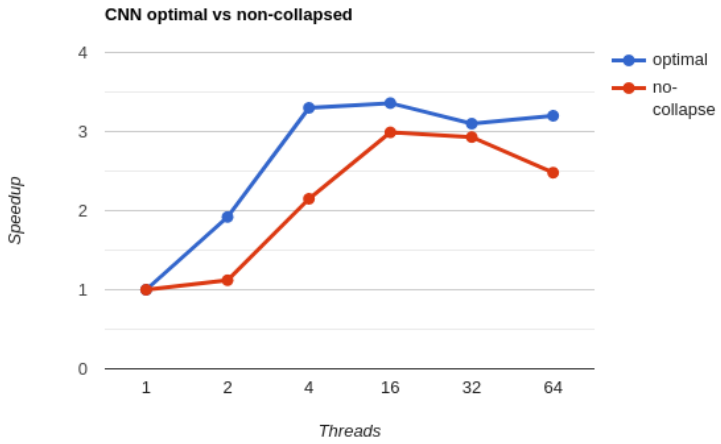

Fig.5

For most applications of CNNs on CPUs the training is going to be done on a GPU and the weights are going to be loaded on to the CPU for inference only. Thus, it makes it important to note that having the I/O from file has a significant impact on the performance of the neural network.

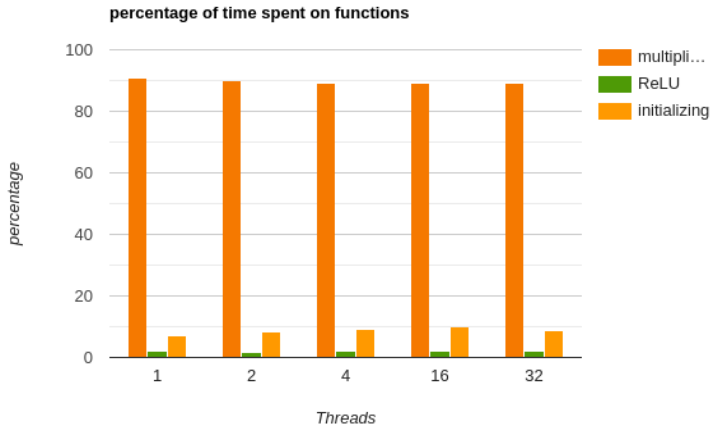

Fig.5

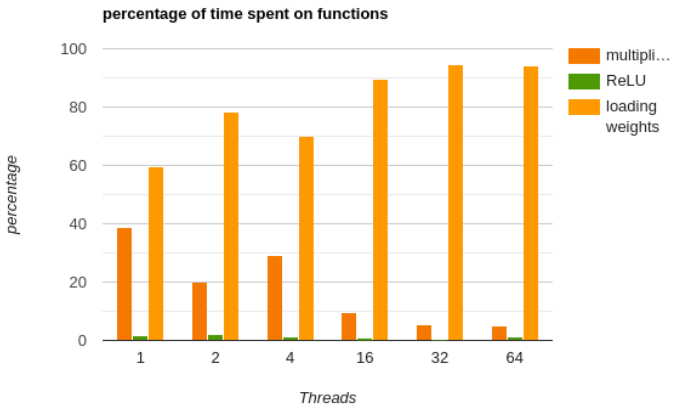

Fig.6

Fig .5 is the percentage of time spent on different functions when the weights were initialized inside the program. Fig.6 shows the percentage of time spent on different functions when the weights were loaded from an outside file.

We can observe that in case of the I/O overhead induced file, the percentage of time spent on matrix multiplication and activation function decreases. However, the percentage of time spent on the I/O increases. Thus, for highly scalable programs, this can pose a bottleneck factor.

6 Conclusions

Based on the experiments in this report, we can successfully conclude the following:

- I/O pose as a huge bottle-necking factor. If optimized, can make CNN inference much more efficient
- When dealing with narrow matrices, thread distribution over all loops can be a bottleneck. This can be solved by efficiently using the collapse clause
- Critical regions have a significant overhead, and can heavily slow down the program. Thus, critical sections should be circumvented.

References

1. J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," 2011 38th Annual International Symposium on Computer Architecture (ISCA), 2011, pp. 353-364.

2. Susan L. Graham Peter B. Kessler Marshall K. McKusick, “gprof: a Call Graph Execution Profiler”, 1982, Association for Computing Machinery Volume 17, 6.

3. Rajput, Vibha & Katiyar, Alok. (2013). Proactive bottleneck performance analysis in parallel computing using openMP. 2.

4. Joao, Jos\'{e} A. and Suleman, M. Aater and Mutlu, Onur and Patt, Yale N. “Bottleneck Identification and Scheduling in Multithreaded Applications”, April 2012, Association for Computing Machinery.

5. S. Eyerman, K. Du Bois and L. Eeckhout, "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," 2012 IEEE International Symposium on Performance Analysis of Systems & Software, 2012, pp. 145-155, doi: 10.1109/ISPASS.2012.6189221.

6. M. Musaev and M. Rakhimov, "Accelerated Training for Convolutional Neural Networks," 2020 International Conference on Information Science and Communications Technologies (ICISCT), 2020, pp. 1-5.