# Implementation Guidelines for Image Processing with Convolutional Neural Networks

Florian Bordes
University of Vienna
Währingerstr. 29
Vienna, Austria
florian.bordes@florens.fr

Erich Schikuta
University of Vienna
Währingerstr. 29
Vienna, Austria
erich.schikuta@univie.ac.at

## ABSTRACT

The domain of image processing technologies comprises many methods and algorithms for the analysis of signals, representing data sets, as photos or videos. In this paper we present a discussion and analysis, on the one hand, of classical image processing methods, as Fourier transformation, and, on the other hand, of neural networks. Specifically we focus on multi-layer and convolutional neural networks and give guidelines how images can be analyzed effectively and efficiently. To speed up the performance we identify various parallel software and hardware environments and evaluate, how parallelism can be used to improve performance of neural network operations. Based on our findings we derive several guidelines for applying different parallelization approaches on various sequential and parallel hardware infrastructure.

## Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

## Keywords

Image Processing, Convolutional Neural Networks, FFT, Sequential and Parallel Implementation

## 1. INTRODUCTION

In this paper we present a discussion and analysis of classical image processing methods, as Fourier transformation, and of neural networks. Specifically we focus on multi-layer and convolutional neural networks and show how effectively and efficiently images can be analyzed.

Hereby we explore the features of CUDA and OpenMP on multicore CPUs and GPUs for the parallelized simulation of a neural network based image processing. We analyze the application for different configurations of neural networks and give recommendations for their effective parallel simulation on both GPU and OpenMP versions.



*MoMM '16, November 28 - 30, 2016, Singapore, Singapore*



ISBN 978-1-4503-4806-5/16/11...$15.00

DOI: http://dx.doi.org/10.1145/3007120.3007165

## 2. IMAGE PROCESSING METHODS

### 2.1 Convolution

An image can be considered as a matrix consisting of values between 0 and 255. During a convolution, we apply a filter (also called kernel, which is a matrix too) on each pixel of this image. In fact a matrix product to compute a convolution is performed. The result of this operation is a new picture. For example, if we apply a Gaussian filter on the picture of Lena [1] Figure 1, we get Figure 2. We see, one goal of a convolution operation is to increase (sharpen) or decrease (blur) the details of a picture.

**Figure 1: Original Image of Lena [1]**

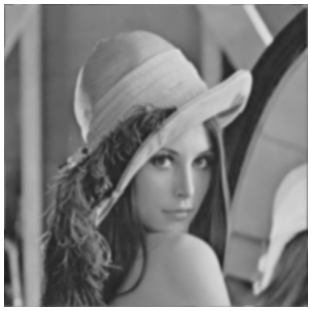

**Figure 2: Convolution Gaussian filter**

An algorithm realizing this method needs 4 loops, two for the picture and two for the kernel. If we have a picture with many layers like in a RGB picture, we must add another loop for each layer. However, in Algorithm 1, we consider the image has only one layer.

We implemented this algorithm in C++, which can be used with bitmap images of 8 bits or 24 bits. We performed some performance test on varying image sizes which are shown in Table 1.

COROLLARY 1. *So it can be concluded that the processing effort is acceptable for a small kernel, but with large pictures and large kernels, the execution times grow dramatically.*

### 2.2 Fourier Transformation

The Fourier transform allows to translate a function in a certain domain to the frequency domain, in fact, we can translate a numerical function into a frequency. The Fourier transform is given by the following formula

```
Data: img, kern, imgH, imgW, kernH, kernW
Result: img-conv
while i ≤ imgH do
    while j ≤ imgW do
        while k ≤ kernH do
            while l ≤ kernW do
                x = (imgW + i - kernW / 2 + k) %
                 imageW;
                y = (imgH + i - kernH / 2 + l) %
                 imageH;
                res = img[x][y] * kern[k][l];
            end
        end
        img-conv[i][j] = tmp;
    end
end
```

**Algorithm 1:** Convolutional algorithm

**Table 1: Execution time for classical convolution**

| Exp# | Pict.size | Bits | Size kernel | Time |
|---|---|---|---|---|
| Exp 1 | 512x512 | 8 | 7x7 | 3.89s |
| Exp 2 | 512x512 | 24 | 7x7 | 11.25s |
| Exp 3 | 512x512 | 8 | 49x49 | 29.12s |
| Exp 4 | 1024x1024 | 8 | 49x49 | 349.10s |

$$\mathcal{F}(f) : v \Rightarrow (F)(v) = \int_{-\infty}^{+\infty} f(t)e^{-i2\pi vt}dt \tag{1}$$

In this case, we have a function in the temporal domain with the variable $t$ and we translate this function in a frequency domain with the variable $v$. When we use the Fourier transformation on an image, we obtain a frequency picture. However, we need to use a different formula to take into consideration the two dimensions of a picture

$$\mathcal{F}(u,v) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} f(i,j)e^{-2i\pi(\frac{ui}{N}+\frac{vj}{N})} \tag{2}$$

The Cooley-Tukey FFT algorithm algorithm became popular in 1965 [3]. One property of the FFT is to allow to be separated. First, we cut the input data into two parts by separating the even indices and the odd indices

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2i\pi}{N}nk} \tag{3}$$

After the separation we get

$$X_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{2i\pi}{N}2mk} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{2i\pi}{N}(2m+1)k} \tag{4}$$

If we isolate $e^{-\frac{2i\pi}{N}}$ in the second part and separate the sums, we get

$$E_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{2i\pi}{N}2mk} \tag{5}$$

$$O_k = \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{2i\pi}{N}2mk} \tag{6}$$

$$X_k = E_k + e^{-\frac{2i\pi}{N}}O_k \tag{7}$$

However, with this operations we perform the FFT only on the $\frac{N}{2}$ inputs, but with the periodicity property of the DFT, we can simply compute the second part of the input $X_k$ with $0 < k < \frac{N}{2}$ by

$$E_{k+\frac{N}{2}} = E_k \tag{8}$$

$$O_{k+\frac{N}{2}} = O_k \tag{9}$$

We know, the Twiddle factor can be

$$e^{-\frac{2i\pi}{N}(k+\frac{N}{2})} = -e^{-\frac{2i\pi}{N}} \tag{10}$$

And we have for $0 < k < \frac{N}{2}$

$$X_{k+\frac{N}{2}} = E_k - e^{-\frac{2i\pi}{N}}O_k \tag{11}$$

Based on these equations we can define Algorithm 2.

```
Data: x, N
if N = 1 then
    return x_0
end
else
    E = DFT(x, N/2);
    O = DFT(x+1, N/2);
    while k < N/2 - 1 do
        tmp = O[k] ∗ e^(i∗PI/N);
        x[k] = E[k] + tmp;
        x[k + N/2] = E[k] - tmp;
    end
end
```

**Algorithm 2:** Cooley Tukey FFT Algorithm: Spatial to frequency domain translation

# 3. SEQUENTIAL IMPLEMENTATION

For the implementation in C++, we need to add some operations as padding to set the image size and kernel size to $2^n$. To apply the FFT on a matrix, we must apply the FFT on each line of the matrix first and secondly on each column. We used the transposition to avoid rewriting the FFT function. After having converted the image and the kernel in the frequency domain, we used a simple multiplication on frequencies to perform convolution operations. Then we applied the inverse FFT operation to get the new image in the spatial domain and create a new bitmap with these data.

The performance results obtained by the implementation of the FFT are given in Table 2. The time to compute a convolution operation on small kernels and small pictures are relatively high. Comparing these execution times to table 1, we see that the classical convolution operation shows better performance. However, in the case of a large picture with large kernel the FFT shows very good performance. Compare the experiment 4 of the classical convolution to experiment 5 of the FFT where we gain more than one minute. With experiments 2 and 3 or 5 and 6, we can observe that changing the size of the kernel does not impact the execution time for the convolution operation. In fact, it is clear that using FFT we need to scale the image and the kernel in a same size. Also the kernel size is not important compared to the image size for a FFT.

COROLLARY 2. *For small images or small kernels we should use the classical method. However, if we want to use larger kernels, the FFT method is preferable.*

**Table 2: Execution Time for FFT and Convolution**

| Exp# | Taille image | Bits | Taille kernel | Time |
|---|---|---|---|---|
| Exp 1 | 512x512 | 8 | 7x7 | 21.62s |
| Exp 2 | 512x512 | 24 | 7x7 | 64.10s |
| Exp 3 | 512x512 | 24 | 49x49 | 66.39s |
| Exp 4 | 512x512 | 8 | 49x49 | 21.44s |
| Exp 5 | 1024x1024 | 8 | 49x49 | 286.72s |
| Exp 6 | 1024x1024 | 8 | 7x7 | 286.79s |

An important concept of Convolutional Neural Networks (CNN) is pooling, which is a form of non-linear down-sampling.

Written in C++, the convolutional neural network is a simple update of the Multi layer backpropagation (MLP) algorithm to perform convolutional operation. Two new layers are added: a convolutional layer and a max pooling layer. To compute the convolution operation, we use the classical convolution. In fact for the MNIST database [2], we use a small image and small kernel, so it is not necessary to use FFT in this case. For the test, we applied the following network:

Input ⇒ CL ⇒ Max ⇒ CL ⇒ Max ⇒ Cl ⇒ Max ⇒ FL ⇒ Output.

CL means Convolutional Layer, Max Max Pooling layers and FC Fully connected layer that is used for the perceptron. Our implementation leads to the following finding:

COROLLARY 3. *It takes more time to use a CNN than a MLP. By the way, the error rate is not better.*

# 4. PARALLELIZATION APPROACHES

In the following we will analyse different parallelization approaches for our image processing problem applying various techniques. Based on our findings we will deduce problem specific implementation recommendations.

## 4.1 Parallel Convolutional Operations

### 4.1.1 OpenMP

#### *Fast Fourier Transformation.*

The FFT is relativelly easy to parallelize because there are many parts of code that can be run independently. At first, we apply parallelism to each layer of the picture (in this example we used RGB pictures). We tried to improve performance by modifying the FFT functions directly to perform parallelism. To apply FFT we need to perform the following operations:

- 1. Apply a padding to the picture to have a size of $2^n$.
- 2. Apply a padding of the kernel to have the same size.
- 3. Apply the FFT operation upon the kernel.
- 4. Apply the FFT operation upon the picture.
- 5. Apply a rotation on the kernel.

For parallelization we need to identify operations which are not dependent. In fact, we can gather kernel operations together. However, these operations must be execute in sequential order, so we can not apply the FFT before padding. We decided to associate a thread with the kernel operations and another thread with the pictures operations. The program waits until these operations terminate to perform the convolution operation. In order to achieve this it was necessary to use the pragma functionality of OpenMP. The results are shown in Table 3. We gain a factor 2 with this improvement. The FFT operations are executed on each line of the matrix. However, these operations are not dependent, so we launched a thread per line to perform the FFT operation. We followed the same approach on the columns too. The results are shown in Table 4. In this case we instantiate one thread for each line, so with 1024 lines we create 1024 threads, which is too much. As result, when we use parallelism on a CPU, we have to limit the number of threads to keep good performances.

| Exp# | API | Pict.size | Bits | Kernel | Time |
|---|---|---|---|---|---|
| Exp 1 | OpenMP | 512x512 | 24 | 7x7 | 3.481s |
| Exp 2 | OpenMP | 512x512 | 24 | 49x49 | 3.563s |

**Table 3: Execution Time of Fast Fourier Transformation with OpenMp upon Layers and Separate Threads for Kernel and Picture**

| Exp# | API | Pict.size | Bits | Kernel | Time |
|---|---|---|---|---|---|
| Exp 1 | OpenMP | 512x512 | 24 | 7x7 | 4.644s |
| Exp 2 | OpenMP | 512x512 | 24 | 49x49 | 4.636s |

**Table 4: Execution Time for Fast Fourier transform with Threads for each Line and Matrix Column**

#### *Traditional Convolution.*

In a first approach we used parallelism over each layer of the picture with a *pragma omp prallel for* statement. In Table 5 we observe that the time of execution is divided by 3. In a second try, we apply the parallelism on each line of the picture's matrix. However, the performance was below expectations as can be seen in Table 6. The parallel approach showed worse results than the sequential one. This is obviously clear because when we define a thread to work on each line of the matrix, we create 512 different threads and our multicore processor can execute only 16 threads in parallel. The management of threads (create and destroy) takes a lot of time. In this special case, it is degrading the performances dramatically. Thus, when we use this approach upon a matrix, we need to limit the thread number. So we tried again but with the option *num_threads(10)* after the

| Exp# | API | Pict.size | Bits | Kernel | Time |
|---|---|---|---|---|---|
| Exp 1 | No | 512x512 | 24 | 7x7 | 0.324s |
| Exp 2 | OpenMP | 512x512 | 24 | 7x7 | 0m0.125s |
| Exp 3 | No | 512x512 | 24 | 49x49 | 15.122s |
| Exp 4 | OpenMP | 512x512 | 24 | 49x49 | 5.503s |

**Table 5: Execution Time for Traditional Convolution with OpenMP on each Layer**

| Exp# | API | Pict.size | Bits | Kernel | Time |
|---|---|---|---|---|---|
| Exp 1 | No | 512x512 | 24 | 7x7 | 0.324s |
| Exp 2 | OpenMP | 512x512 | 24 | 7x7 | 0m2.001s |
| Exp 3 | OpenMP | 512x512 | 24 | 49x49 | 3m8.437s |

**Table 6: Execution Time for Traditional Convolution with OpenMP on each Matrix and Columns Lines.**

pragma to limit the thread number to 10. The result are shown in Table 7.

| Exp# | API | Pict.size | Bits | Kernel | Time |
|---|---|---|---|---|---|
| Exp 1 | No | 512x512 | 24 | 7x7 | 0.324s |
| Exp 2 | OpenMP | 512x512 | 24 | 7x7 | 0.149s |
| Exp 3 | OpenMP | 512x512 | 24 | 49x49 | 1m20.641s |

**Table 7: Execution Time for Traditional Convolution with OpenMP upon a matrix with limited amount of threads**

We faced an improvement, however, the performance is below our first parallelization optimization, where we applied the pragma on the different layers. This convolution method is quite difficult to parallelize. However, as result we recommend the parallelism operation for each layer that gives good performance results.

COROLLARY 4. *The performance improvement obtained with OpenMP is remarkable, particularly on pictures with multiple layers, where we associate a thread per layer. We realized that we have to keep the numbers of thread low, because the cost for management of threads exceeds the performance gain.*

### 4.1.2 Pthreads

By curiosity, we investigated if Pthreads were more efficient than OpenMP API, so we did the same tests using Pthreads.

#### *Fast Fourier Transformation.*

Analogous to the test with OpenMP, we execute a thread for each layer of the picture: The obtained results are shown in Table 8. The execution times are cut by three as in the

| Exp# | API | Pict.size | Bits | Kernel | Time |
|---|---|---|---|---|---|
| Exp 1 | No | 512x512 | 24 | 7x7 | 12.876s |
| Exp 2 | Pthread | 512x512 | 24 | 7x7 | 4.628s |
| Exp 3 | Pthread | 512x512 | 24 | 49x49 | 4.629s |

**Table 8: Execution Time for Fast Fourier Transformation and Convolution with Pthreads on each Pictures Layer**

OpenMP implementation. In a second step we apply the same improvements separating the processing operations on the kernel and the pictures. As shown in Table 9, the results are very similar to the OpenMP implementation. However with Pthreads, we had to modify a large part of the source code. In fact, we created a structure to contain the arguments and three additional functions: one to perform parallel computation on the layers, one for the kernel and another one for the picture.

| Exp# | API | Pict.size | Bits | Kernel | Time |
|---|---|---|---|---|---|
| Exp 1 | Pthread | 512x512 | 24 | 7x7 | 3.548s |
| Exp 2 | Pthread | 512x512 | 24 | 49x49 | 3.546s |

**Table 9: Execution Time for Fast Fourier Transform and Convolution with Pthreads and Separate Threads for Kernel and Picture**

#### *Traditional Convolution.*

By applying parallelization on each layer with traditional convolution and Pthread we obtain results shown in Table 10. The results are too pretty similar to OpenMP. I have

| Exp# | API | Pict.size | Bits | Kernel | Time |
|---|---|---|---|---|---|
| Exp 1 | Pthread | 512x512 | 24 | 7x7 | 0.117s |
| Exp 2 | Pthread | 512x512 | 24 | 49x49 | 5.284s |

**Table 10: Execution Time for Traditional Convolution with Pthreads**

not tested to apply the parallelism on the line and columns directly on the matrix this time because we have already seen that this kind of optimization has bad performance.

COROLLARY 5. *The performances observed with Pthreads are pretty similar to OpenMP. However, the programming efforts to use Pthreads are high. For the Pthread implementation, we realized three different structures to get the function arguments and seven new functions to use Pthreads. With OpenMp, we only had to add five or six pragma code lines. Achieving the same performance, this last solution is the best for our case of use. So we recommend for a fast development process and/or not experienced developer to focus on OpenMP, specifically for our problem.*

### 4.1.3 CUDA

To conclude our tests, we use CUDA. To make a comparison with the parallelization on CPU, we compile and use a program that is available in the CUDA SDK sample folder named ConvolutionSeparable. This program uses many optimizations like Separable Filter and memory coalescence to achieve better performance. Using the same image of the previous tests we get the results as shown in Table 11. The results are impressive.

| Exp# | API | Pict.size | Bits | Kernel | Time |
|---|---|---|---|---|---|
| Exp 1 | CUDA | 512x512 | 24 | 7x7 | 0.00018s |
| Exp 2 | CUDA | 512x512 | 24 | 49x49 | 0.00060s |

**Table 11: Execution Time for Convolution with CUDA**

## 4.2 Parallel Neural Network Execution

### 4.2.1 Multilayer Perceptron Parallelization

For the multilayer perceptron we use the same source code that was used in the first part of this report. The main difference is the use of OpenMP for parallelization and the use of a batch learning mode. Each worker executes forward and backward operations computing the different deltas. Only updating of the synaptic weights is done sequentially. The first test is done with digit 3 and 4 of the MNIST database,

a learning rate of 0.01, a momentum of 0.9 and 3 layers of 30 neurons. As can be seen in Table 12 increasing the number of threads the execution time is reduced. gives results as shown in Table 13. Similar to the previous tests, if we increase the thread number, we reduce the computation time.

| Exp# | OpenMP | Train | Error | Time |
|---|---|---|---|---|
| Exp 1 | No | 5 | 1.35% | 89.85s |
| Exp 2 | Yes (4 threads) | 5 | 0.80% | 41.076s |
| Exp 3 | Yes (8 threads) | 5 | 4.01% | 37.162s |
| Exp 4 | Yes (16 threads) | 5 | 2,35% | 23.399s |

**Table 12: Execution Times of Multilayer Perceptron with Parallelized Batch Training on the 3 and 4 Digit of MNIST Dataset**

| Exp# | OpenMP | Train | Error | Time |
|---|---|---|---|---|
| Exp 1 | No | 1 | 56.34% | 21m44.262s |
| Exp 2 | Yes (4 threads) | 1 | 27.97% | 7m10.854s |
| Exp 3 | Yes (8 threads) | 1 | 27.34% | 5m17.810s |
| Exp 4 | Yes (16 threads) | 1 | 31.73% | 4m58.916s |
| Exp 5 | Yes (16 threads) | 5 | 20.46% | 21m57.382s |

**Table 13: Execution Times of Multilayer Perceptron with Parallelized Batch Training on all MNIST Dataset**

### 4.2.2 Convolutional Neural Network Parallelization

Analog to our MLP tests we parallelize the batch training over different threads. For digits 3 and 4 of the MNIST database we obtain the results as shown in Table 14. We

| Exp# | API | Train | Error | Time |
|---|---|---|---|---|
| Exp 1 | OpenMP | 1 | 15.21% | 4m25.469s |
| Exp 2 | OpenMP | 2 | 6.72% | 6m47.457s |

**Table 14: Execution Times of Convolutional Neural Network witch Batch Training on digit 3 and 4 of MNIST dataset**

observe that the CNN is harder to train than the MLP. However, there are more layers and more computation is required.

In a second attempt we use CUDA to perform the convolution operations. We realized the computation of the kernel by using one threads for one pixel. The results are shown in Table 15 for digit 3 and 4 of the MNIST database.

COROLLARY 6. *Parallelization of a neural network is a sensible task. We have to take care about the parameters for the neural network and also how to manage the memory. Both approaches delivered good results. Parallelism on CPU with OpenMP for MLP delivers good results. The results for the CNN are also good and even better than the MLP.*

We analyzed the parallelization of neural networks heavily in the past. In our former work we developed a number of rules how to approach the parallel implementation of neural network execution in general, independently of the specific parallel hardware. Starting with developing very general frameworks for neural network parallelization [6], we covered also the practical parallelization on classical supercomputers [7] and cluster systems [8, 5], and explored multicore CPU and GPU systems [4].

| Exp# | API | Train | Error | Time |
|---|---|---|---|---|
| Exp 1 | Cuda | 1 | 6.73% | 15m51.660s |

**Table 15: Execution Times of Convolutional Neural Network with CUDA on digit 3 and 4 of MNIST dataset**

# 5. CONCLUSION

In this paper we presented a study on the applicability of convolutional neural networks for image processing and compared its usage to classical approaches, as Fourier transformation. Hereby a specific focus was laid on performance issues of the various sequential and parallel implementations. We derived several recommendations based on our findings to ease and guide the implementation approach on different sequential and parallel infrastructures.

# 6. ACKNOWLEDGMENTS

This work was partially supported by the ERASMUS Programme of the European Union.

# 7. REFERENCES

[1] The Lena story. http://www.cs.cmu.edu/chuck/lennapg/lenna.shtml. Accessed: 2015-06-11.

[2] The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/. Accessed: 2016-01-29.

[3] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[4] A. A. Huqqani, E. Schikuta, S. Ye, and P. Chen. Multicore and gpu parallelization of neural networks for face recognition. *Procedia Computer Science*, 18:349–358, 2013.

[5] H. Schabauer, E. Schikuta, and T. Weishäupl. Solving very large traveling salesman problems by som parallelization on cluster architectures. In *Parallel and Distributed Computing, Applications and Technologies, 2005. PDCAT 2005. Sixth International Conference on*, pages 954–958. IEEE, 2005.

[6] E. Schikuta, H. Wanek, and T. Fuerle. Structural data parallel simulation of neural networks. *Journal of Systems Research and Information Science*, 9(1):149–172, 2000.

[7] E. Schikuta and C. Weidmann. Data parallel simulation of self-organizing maps on hypercube architectures. *Proceedings of WSOM*, 97:4–6, 1997.

[8] T. Weishäupl and E. Schikuta. Parallelization of cellular neural networks for image processing on cluster architectures. In *Parallel Processing Workshops, 2003. Proceedings. 2003 International Conference on*, pages 191–196. IEEE, 2003.