

Parallelized Convolutional Layer

Group 12

彭敬樺

Computer Science

NCTU

0416106

wilbert.phen@gmail.com

方君安

Computer Science

NCTU

0416076

kjw472800@gmail.com

林彥傑

Computer Science

NCTU

0416213

jay101630@gmail.com

Github Link for source code : https://github.com/WarClans612/pp_final

1. Abstract

Over the last several years, machine learning techniques, particularly when applied to neural networks, have played an increasingly important role in the design of pattern recognition systems. In fact, it could be argued that the availability of learning techniques has been a crucial factor in the recent success of pattern recognition applications such as continuous speech recognition and handwriting recognition.

Deep learning (also known as deep structured learning or hierarchical learning) is part of a broader family of machine learning methods based on learning data representations, as opposed to task-specific algorithms. Learning can be supervised, semi-supervised or unsupervised. In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery.

CNNs use a variation of multilayer perceptrons designed to require minimal pre-processing. This type of Neural Network is usually used in image classification related problems. A CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolutional layers, pooling layers, fully connected layers and normalization layers.

Convolutional networks were inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage.

They have applications in image and video recognition, recommender systems, image classification, medical image analysis, and natural language processing.

Due to the increasing amount of data and calculation power needed, parallelized parts of the layer could greatly improve overall time consumption for data training.

However, there are problems, which due to the full connectivity between nodes they suffer from the curse of dimensionality, and thus do not scale well to higher resolution images.

2. Introduction

Convolutional layers apply a convolution operation to the input, passing the result to the next layer. The convolution emulates the response of an individual neuron to visual stimuli. The convolutional layer is the core building block of a CNN.

Convolution is the process of adding each element of the image to its local neighbors, weighted by the kernel. This is related to a form of mathematical convolution. It should be noted that the matrix operation being performed - convolution - is not traditional matrix multiplication, despite being similarly denoted by $*$.

For example, if we have two three-by-three matrices, the first a kernel, and the second an image piece, convolution is the process of flipping both the rows and columns of the kernel and then multiplying locally similar entries and summing. The element at coordinates $[2, 2]$ (that is, the central element) of the resulting image would be a weighted combination of all the entries of the image matrix, with weights given by the kernel:

$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \right) [2, 2] \\ = (i \cdot 1) + (h \cdot 2) + (g \cdot 3) + (f \cdot 4) + (e \cdot 5) + (d \cdot 6) + (c \cdot 7) + (b \cdot 8) + (a \cdot 9).$$

Figure 1. Simple matrix multiplication

The other entries would be similarly weighted, where we position the center of the kernel on each of the boundary points of the image, and compute a weighted sum.

The values of a given pixel in the output image are calculated by multiplying each kernel value by the corresponding input image pixel values. This can be described algorithmically with the following pseudo-code:

```

for each image row in input image:
  for each pixel in image row:
    set accumulator to zero
    for each kernel row in kernel:
      for each element in kernel row:
        if element position corresponding* to pixel position
        then
          multiply element value corresponding* to pixel value
          add result to accumulator
        endif
    set output image pixel to accumulator

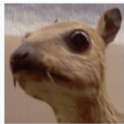

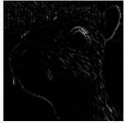

```

*corresponding input image pixels are found relative to the kernel's origin.

If the kernel is symmetric then place the center (origin) of kernel on the current pixel. Then kernel will be overlapped with neighboring pixels too. Now multiply each kernel element with the pixel value it overlapped with and add all the obtained values. Resultant value will be the value for the current pixel that is overlapped with the center of the kernel.

If the kernel is not symmetric, it has to be flipped both around its horizontal and vertical axis before calculating the convolution as above.

Depending on the element values, a kernel can cause a wide range of effects.

Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	

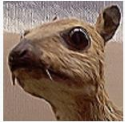

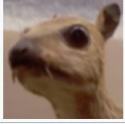

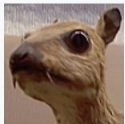
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur 3 × 3 (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	
Gaussian blur 5 × 5 (approximation)	$\frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	
Unsharp masking 5 × 5 Based on Gaussian blur with amount as 1 and threshold as 0 (with no image mask)	$\frac{-1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & -476 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	

Figure 2. Different kernel value and its result

3. Statements of Problem

It can be seen from the process above that convolving an image with a kernel filter involved massive number of kernel multiplication(convolution). With the increasing number of picture and filter resolution, the computation power needed to be used increased.

Because convolution is such a central operation in the algorithm, much work need to be invested in making matrix multiplication algorithms efficient. Applications of convolution in computational problems are found in many fields including scientific computing and pattern recognition. Many different algorithms have been designed for multiplying matrices on different types of hardware, including parallel and distributed systems, where the computational work is spread over multiple processors (perhaps over a network).

Directly applying the mathematical definition of matrix multiplication gives an algorithm that takes time on the order of n^2 to convolve $n \times n$ kernel matrix for each pixel in the picture ($\Theta(n^2)$ in big O notation). Convolution computation is needed for each time the filter is moved throughout the picture space. And would need approximately $m \times n \times n$ (m as number of pixel (and considering sufficient padding and stride equal to 1).

This calculation increased faster as the size of filter increase. As each pixel in the picture need its own convolution job to

be done. And so the time needed for computation will increase.

4. Proposed Solution

The design of many core processors is strongly driven by demands for greater performance at reasonable cost. To make effective use of the available parallelism in such systems, the parallel programming model is of great importance. There are several parallel programming models, runtime libraries, and APIs that help developers to move from sequential to parallel programming. For the purposes of this project, we have chosen 2 parallel programming models that suit the problem: OpenMP and CUDA.

In convolutional layer, primitive data has to multiply by matrix filters, in order to get the feature map. In this process, numerous number of matrix multiplication is calculated. To reduce the heavy workload, we can accelerate the process by the parallelizing the computation into several CPU cores or several CUDA cores.

When using multi-core CPU to accelerate the work, firstly, we designate the total number of work needed to be done by each core. The separation of workload is hoped to be done as fairly as possible by considering the number of cores that the current machine has and by considering the number matrix multiplication operations that need to be utilized.

Next, each CPU cores will be assigned its designated start and end requirements of its calculation to specify its portion of works and to specify the location of stored result, without relying on other parallelized parts of the code. This method is hoped to be able to decrease dependency between spawned thread and diminishing the need to use any locking mechanism to protect result data.

When using CUDA cores to accelerate the work, we first maintain the number of matrix multiplications that can be done. By calculating massive number of matrix calculation using CUDA cores, the timing is sure to be improved due to the advantage of CUDA cores in processing matrix related workload.

4.1. OpenMP

OpenMP is the de-facto standard for shared memory programming, and is based on a set of compiler directives or pragmas, combined with a programming API to specify parallel regions, data scope, synchronization, and so on. It also supports runtime configuration through the use of runtime environment variables, e.g. `OMP_NUM_THREADS` to specify the number of threads at runtime. OpenMP is a portable parallel programming approach and is supported on C, C++, and Fortran. It has

been historically used for loop-level and regular parallelism through its compiler directives.

Since OpenMP is a language enhancement, every new construct requires compiler support. Therefore, its functionality is not as extensive as library-based models. Moreover, race condition is a serious problem in OpenMP. Although it provides the user with a high level of abstraction, the programmer still has to avoid race condition.

4.2. CUDA

The graphics processing unit (GPU), as a specialized computer processor, addresses the demands of real-time high-resolution 3D graphics compute-intensive tasks. By 2012, GPUs had evolved into highly parallel multi-core systems allowing very efficient manipulation of large blocks of data. This design is more effective than general-purpose central processing unit (CPUs) for algorithms in situations where processing large blocks of data is done in parallel, such as: fast sort algorithms of large lists, two-dimensional fast wavelet transform, molecular dynamics simulations, push-relabel maximum flow algorithm, or image processing.

CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing — an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. The CUDA platform is designed to work with programming languages such as C, C++, and Fortran. CUDA was an acronym for Compute Unified Device Architecture, but Nvidia subsequently dropped the use of the acronym.

4.3. Serial Approach

To parallelize the convolution layer, we need to understand what convolution layer does. The main purpose of the convolution layer is to extract some features from the input image, like the outline of the object, and then pass them to the next layer to do other job, for example, detect what is that object. The process of extract the feature from the image is called **convolution operation**.

Imagine we have an image represented as a 5x5 matrix with integer values, and we have a 3x3 matrix, called **filter**, or **filter matrix**. We take this filter matrix and slide it as 3x3 window around the image. At each position the filter visits, the filter will multiply the values of the 3x3 matrix by the values in the image that currently being covered by the

window. Figure 1 demonstrate how the convolution operation is done. First, the 3x3 window (the yellow area) covers the left top of the image. Then the filter multiply the values of itself by the values of the 3x3 window, and then we sum each product and get the final output value. This computation is like the dot product on 2D vector.

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Figure 3. Example 5x5 image matrix & 3x3 filter matrix

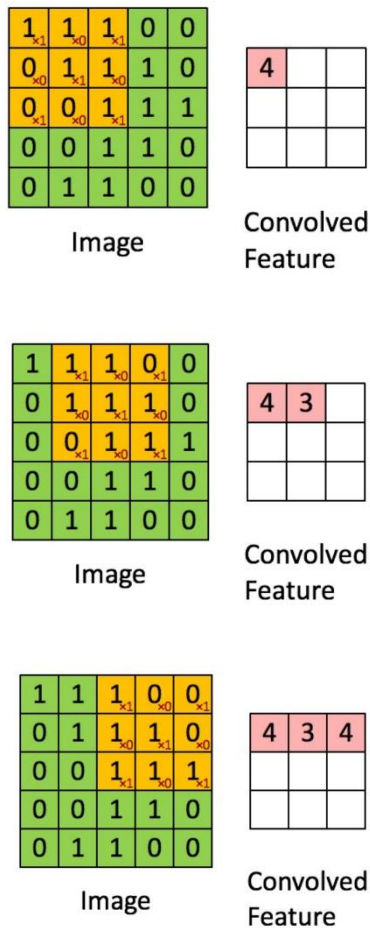


Figure 4. Example of Convolution Operation

After the convolution operation, we can get a matrix which is small than the input image, we call it **feature map**. The feature map contains the features those are extracted from the input image by the filter matrix. The type of the feature is determine by the filter matrix. For example, the following filter matrix can extract the outline of the objects in the input image:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

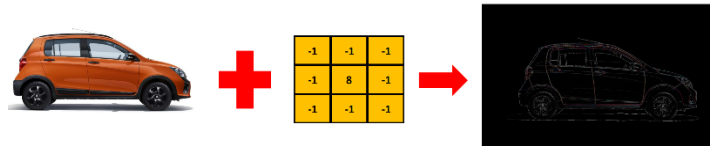


Figure 5. Example of filter matrix

As you can see in Figure 1, the size of the feature map is smaller than the original input image. In this project, we want to keep the size of the feature map to be the same as the input image. In order to archive this, before doing the convolution operation to the input image, we can do the **zero padding** operation. That is, fill zero around the input image. The formula of zero padding is:

$$p = \frac{k - 1}{2}$$

While k is the dimension of the filter matrix and p is the size we should fill around the image. E.g., the image is 5x5 and the filter matrix is 3x3, then p is equal to 1, so we need to padding one row (column) of zero to the around the image.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

↓

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 6. Example of zero padding operation

Following is the pseudo code of the serial approach:

```
def main():
    open the input image
    load filter matrices from the file
    for each filter matrix in filter matrices do
        output[i] = conv_layer(input_image, filter)
    done

def conv_layer(input_image, filter, channel):
    padding input_image
```

```

allocate the memory for the output
for i in output_height do
  for j in output_width do
    output[i][j] = dot_product(input_image, i, j, filter)
  done
done

def dot_product(input_image, start_i, start_j, filter):
  sum = 0
  for x in filter_height do
    for y in filter_width do
      sum +=
input_image[start_i+x][start_j+y]*filter[x][y]
    done
  done
  return sum

```

We use OpenCV to help us to open the image, and we load multiple different filter matrices from the file. For each filter matrix, we first need to pad the image depended on the size of the filter. Then we can do the convolution operation and get the output. Notice that in general, an image has three channels: R, G, B, so we need to do three convolution operation for each filter matrix.

4.4.OpenMP Approach

In the serial approach, we can find that each convolution operation will produce large number of matrix dot product computation and each dot product computation is independent from each other. In this case, if we do not use any parallelization technology, we will spend too much time in matrix dot product. For example, an rgb image in size of 2048x1157 cost almost 69 seconds to done the convolution operation with twenty 15x15 filter matrices at our experiment. To optimize the efficiency of the convolution operation, we introduce OpenMP to our project.

As mentioned above, each matrix dot product computation is independent from each other, so we can parallelize these computations to each CPU core. E.g., the three computation at Figure 1 can be parallelized because the three parts is independent from each other.

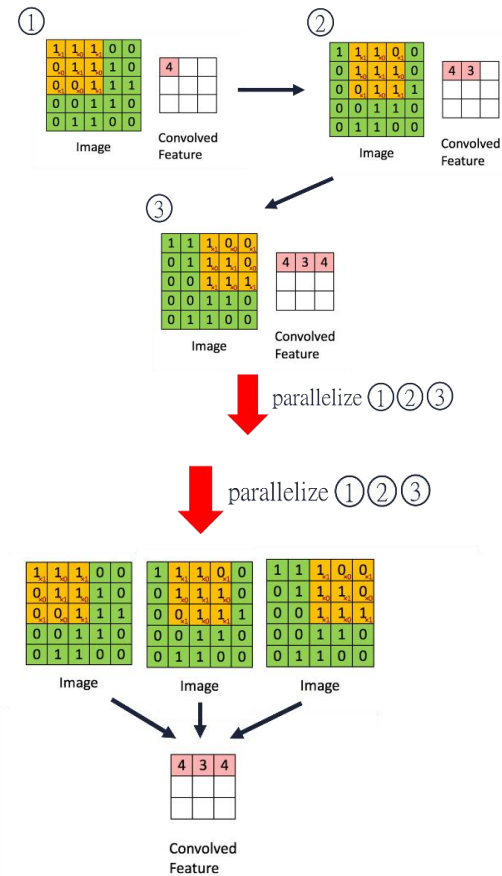


Figure 7. Visualization of OpenMP Approach

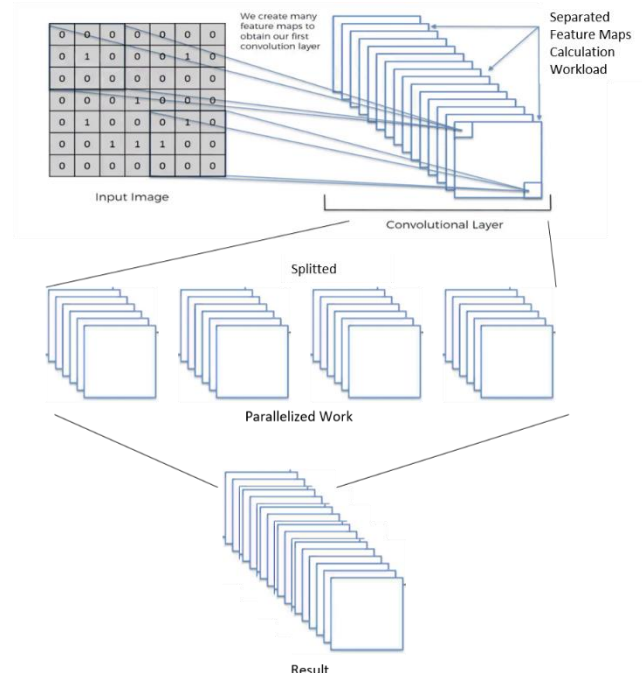


Figure 8. Simple graph visualization of the system

4.5.CUDA Approach

GPU acceleration is one of the mainstream ways to speedup program. With multicores in the device, the program can implement data parallelism computing at high efficiency. Despite of some limitations of GPU, it is still able to perform well.

To parallelize convolutional layer, there are two methods. One is to parallelize program based on filters. In other words, programmers assign one thread with one filter to do computing and generate one feature map resulting after one layer. The other way is mapping each thread to one elements in feature feature maps. In order to utilize max computing power of the GPU, we choose latter to implement.

Parallelizing program to the number of output matrix's size has some advantages over parallelizing program to the number of filters. One is that the program parallelized to more threads. In the GPU, we have thousands of cores. Larger number of threads facilitate program to use resource wholly. Another advantage is that computing with more threads is suitable to GPU, since when computing in the GPU, it is necessary for program to move data from host to device. Only with the immense throughput can program compensate the overhead of moving data.

When implementing, we invoke a kernel and create the same number of threads as output matrix's size. Each threads access the data of part image and all filter to compute the one result element in the output matrix. We can parallelize program by this methods is due to no dependence between each threads. The size of blocks did not determined until we did serval test and observed which dividing way have highest performance.

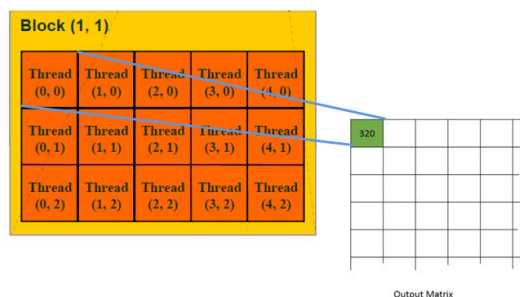


Figure 9. One thread accounts for one element

4.6.CUDA Improvement

To improve the CUDA code's performance, using different memory space is an ideal method. Except for global memory, a GPU still has other type of memory. They have their own attributes. For example, although constant

memory is read only, it can be access in local time by threads in same grids. The shared memory is also able to be access in local time. However the shared memory space is only shared by threads in the same blocks. If we can take advantage of these features properly, we can make our program have high performance.

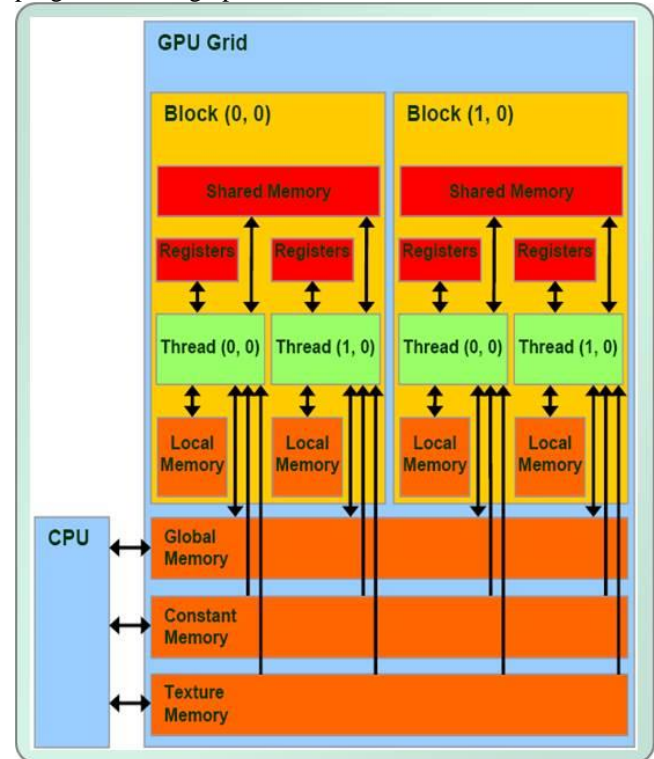


Figure 10. Figure for CUDA memory model

When CUDA cores compute for the result of output matrix, we notice that data of filter is repeatedly access by cores. If we put those filter in the global memory, the program is destined to assume large latency. The filter's data is read only. It will not be modified by threads. Therefore, we can put filter in constant memory. By doing so, every thread can get the filter in local time and the program can get rid of latency of getting data from global memory.



Figure 11. Constant memory share by grid

The other way to strengthen the program is by using shared memory. We notice the compute to off-chip memory access

ratio is close to 1:1. It is not very high, so it still have space for us to improve. Hence, we put input image in shared memory and expect it can reduce the latency of frequent getting data from global memory. Each blocks has to put part of the input image which they are responsible for in the shared memory and share by threads in block.

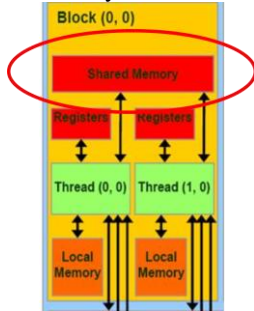


Figure 12. Constant memory share by block

However, there are some difficulty. The number of elements is not equal to the size of the block. To be more detailed, we can not ask each threads to put one elements in shared memory. Instead, in different case, different threads have to put different number of element in shared memory. To do so, we have to add some if statement in CUDA code, which will increase the overhead. Therefore, there are some uncontrolled extra factors make us not evaluate the performance. We have to test to figure out the extent of improvement.

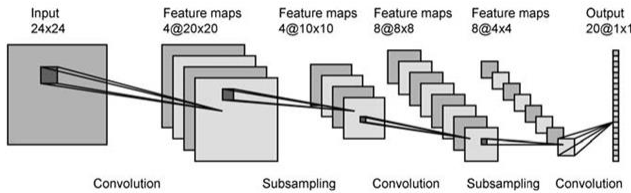


Figure 13. Simple figure of Convolutional Neural Network

When using shared memory, we also have to notice whether there is bank conflict or not. If there is bank conflict occurring, it take times the access latency to complete. To detect whether there is conflict or not, we use nvprof which is a toolkit offered by nvidia. We use parameter `shared_load_transactions_per_request` to get the information. If `shared_load_transactions_per_request` ≤ 1 , it means there is no bank conflict since the cores can get data by one transaction. Fortunately, there is no bank conflicts.

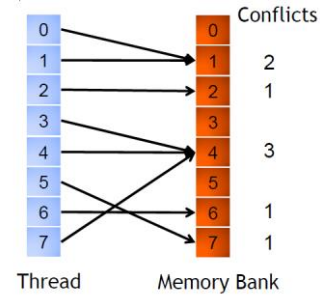


Figure 14. Bank Conflict

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "GeForce GTX 950M (b)"					
Kernel: sharedMatrixMultiply(intx, int, int, int, intx)					
6	shared_load_transactions_per_request	Shared Memory Load Transactions Per Request	1.000000	1.000000	1.000000

Figure 15. Bank conflict detection

5. Testing Environments

All experiments is tested on the same environment. Using Ubuntu 16.04 on device with spec:

CPU : i7 8700k
RAM : 64 GB
GPU : RTX 2080 8GB

6. Experiments

The focus of this section is on the performance of the three implementations of the approach towards image convolution. Below is the 2 testing image used to provide visualization of the results.



Figure 16. test1.jpg



Figure 17. test2.jpg

-1	-1	-1	1	0	-1
-1	8	-1	2	0	-2
-1	-1	-1	1	0	-1
-1	-2	-1	-1	0	1
0	0	0	-2	0	2
-1	-2	-1	-1	0	1

Figure 18. Four matrix filter used on the testing image



Figure 19. test1.jpg applied with 1st filter



Figure 20. test1.jpg applied with 2nd filter



Figure 21. test1.jpg applied with 3rd filter



Figure 22. test1.jpg applied with 4th filter

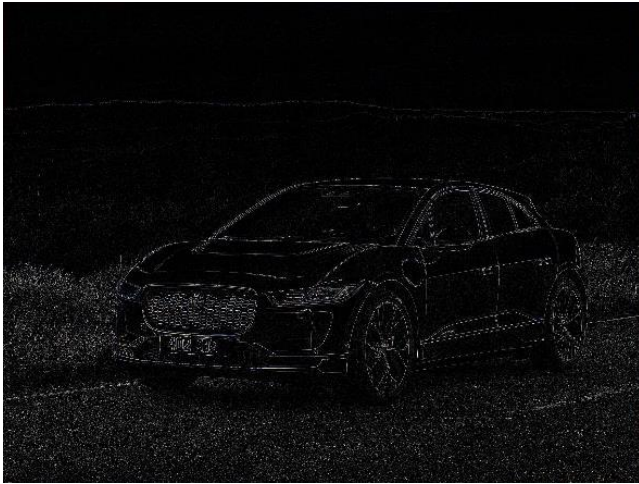


Figure 23. test2.jpg applied with 1st filter



Figure 26. test2.jpg applied with 4th filter



Figure 24. test2.jpg applied with 2nd filter

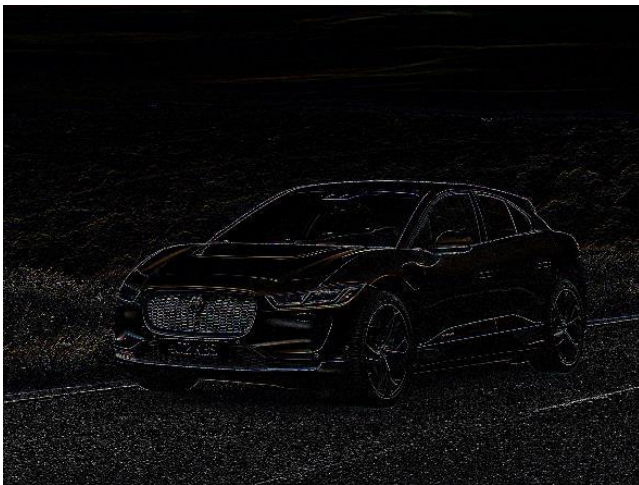


Figure 25. test2.jpg applied with 3rd filter

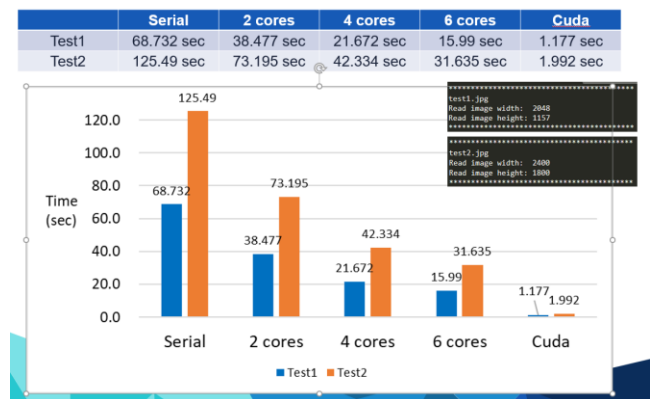


Figure 27. Comparison of performance for different approaches

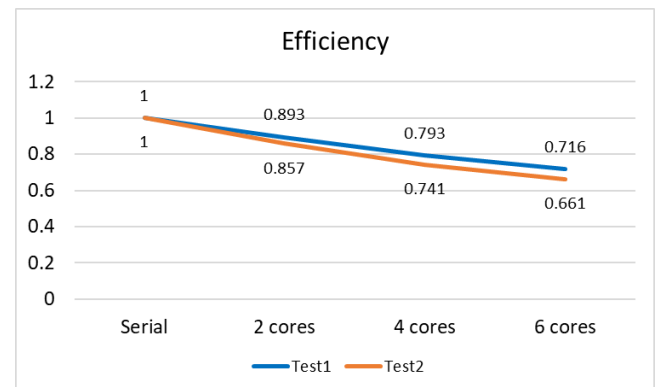


Figure 28. Figure of efficiency between approaches

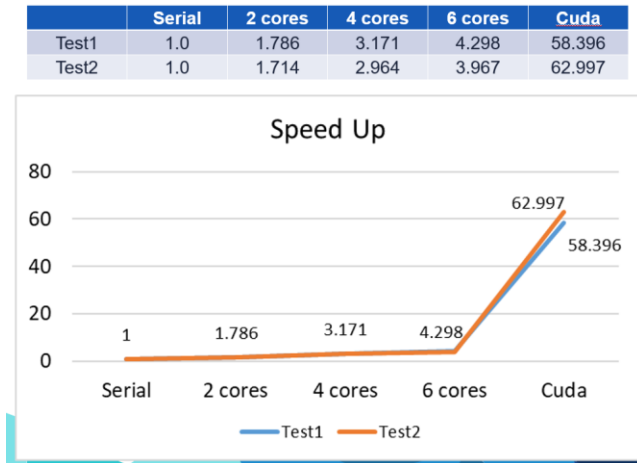


Figure 29. Figure of speedup between approaches

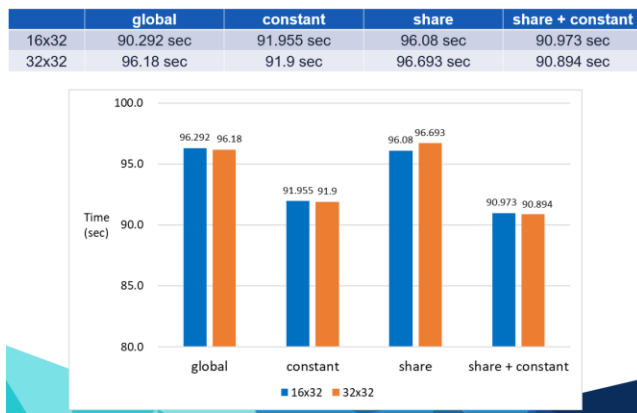


Figure 30. Comparison of performance for CUDA improvements

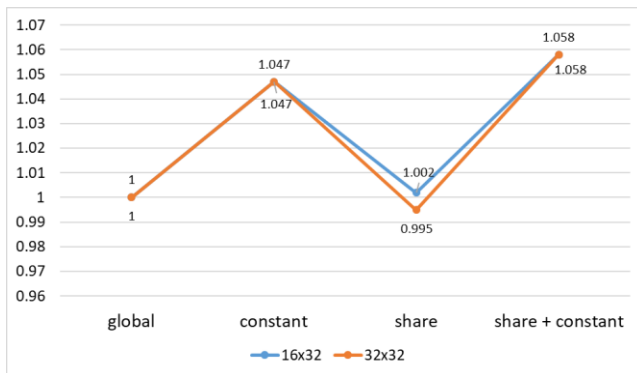


Figure 31. Figure of speedup between CUDA improvements

7. Conclusion

Optimally, the speedup from parallelization would be linear—doubling the number of processing elements should halve the runtime, and doubling it a second time should again halve the runtime. However, very few parallel algorithms achieve optimal speedup. Most of them have a

near-linear speedup for small numbers of processing elements, which flattens out into a constant value for large numbers of processing elements.

Understanding data dependencies is fundamental in implementing parallel algorithms. No program can run more quickly than the longest chain of dependent calculations (known as the critical path), since calculations that depend upon prior calculations in the chain must be executed in order. However, most algorithms do not consist of just a long chain of dependent calculations; there are usually opportunities to execute independent calculations in parallel.

Speedup improvement from using GPU rather than CPU is massive. And although the massive improvements, we could only achieve 20% of GPU utilization. Another way to increase the speedup in the graph further is by using larger picture input and increase filter size. However, we are determined to provide real life expected results when using our approach as solution.

When using CPU thread to the max, we achieve quite a linear improvement as the number of cores used increase. However, when using the entire CPU cores at once, it has some disadvantage compared to using partial (20% approximately in our case) of the GPU. Full utilization of CPU could result in unresponsive or sluggish system.

8. Reference

- [1] Jilin Zhang, Junfeng Xiao, Jian Wan, Jianhua Yang, Yongjian Ren, Huayou Si, Li Zhou, and Hangdi Tu, A Parallel Strategy for Convolutional Neural Network Based on Heterogeneous Cluster for Mobile Information System, <https://www.hindawi.com/journals/misy/2017/3824765/>
- [2] Dang Ha The Hien, A guide to receptive field arithmetic for Convolutional Neural Networks, <https://medium.com/mlreview/a-guide-to-receptive-field-arithmetic-for-convolutional-neural-networks-e0f514068807>
- [3] Keiron O'Shea and Ryan Nash, An Introduction to Convolutional Neural Networks, https://www.researchgate.net/publication/285164623_An_Introduction_to_Convolutional_Neural_Networks
- [4] Parallelizing Convolutional Neural Network Training, <http://bcliu.github.io/parallel-cnn/>
- [5] Parallel Deep Convolutional Neural Network Training by Exploiting the Overlapping of Computation and Communication – HiPC 2017
- [6] Filter Shaping for Convolutional Neural Networks – ICLR 2017
- [7] Deep Residual Learning for Image Recognition - CVPR 2015
- [8] What do Deep Networks Like to See - CVPR 2018
- [9] Convolution-based Operations, <http://www.mif.vu.lt/atpazinimas/dip/FIP/fip-Convolut-2.html>
- [10] Matrix multiplication algorithm, https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm
- [11] OpenMP, <https://en.wikipedia.org/wiki/OpenMP>
- [12] CUDA, <https://en.wikipedia.org/wiki/CUDA>
- [13] Kernel (image processing), [https://en.wikipedia.org/wiki/Kernel_\(image_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing))