YAŞAR UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
MASTER THESIS

OPTIMIZATION OF CONVOLUTIONAL NEURAL NETWORKS VIA GRAPHIC CARDS FOR CENTRALIZED DATA

ERİNÇ CİBİL
THESIS ADVISOR: ASST. PROF. DR. İBRAHİM ZİNCİR

MASTER OF SCIENCE IN COMPUTER ENGINEERING

PRESENTATION DATE: 22.08.2019

BORNOVA / İZMİR
AUGUST 2019

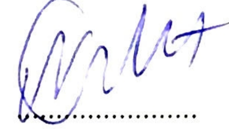
We certify that, as the jury, we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Jury Members: **Signature:**

Asst. Prof. Dr. İbrahim ZİNCİR
Yaşar University

....

Asst. Prof. Dr. Nalan ÖZKURT
Yaşar University


....

Asst. Prof. Dr. Samsun M. BAŞARICI
Adnan Menderes University

....

Prof. Dr. Cüneyt GÜZELİŞ
Director of the Graduate School

ABSTRACT

OPTIMIZATION OF CONVOLUTIONAL NEURAL NETWORKS VIA GPU FOR CENTERALIZED DATA

CİBİL, Erinç

Msc, Computer Engineering

Advisor: Asst. Prof. Dr. İbrahim ZİNCİR

August 2019

In this thesis, it is aimed to design a new approach optimized for systems that use multiple graphics processing units (GPU) in order to find highly discriminative attributes of digitized handwritten numbers obtained from MNIST dataset and their results.

In this study, the convolutional neural network (CNN) method and digitized handwriting classification method are discussed in three sections. In the first part, the classification is obtained by implanting the naive convolutional neural network into the graphic processing unit.

In the second stage, the process layers for graphic processing units are parallelized and the data is adjusted for parallel processing layers and the classification is aimed with optimized memory access pattern approach.

In the last stage, the method has been improved to work on more than one graphic processing unit. The aim of this stage is to improve the education time of convolutional neural network inversely proportional to the number of graphic processing units used.

Keywords : Handwritten digit classification, convolutional neural network (CNN), parallel processing, feature extraction, multiple gpu parallel processing, graphic processing unit (GPU).

ÖZ

EVRİŞİMSEL SİNİR AĞLARIN GRAFİK KARTLARI İLE VERİ MERKEZİ EN İYİLEMESİ

Cibil, Erinç

Yüksek Lisans, Bilgisayar Mühendisliği Tezli Yüksek Lisans Programı

Danışman: Dr. Öğr.Üyesi İbrahim ZİNCİR

Ağustos 2019

Bu tezde, MNIST veri setinde elde edilen dijitallestirilmiş el yazısı numaralar ve sonuçlarının, ayrıştırıcılığı yüksek özniteliklerinin bulunması için çoklu grafik işlem birimi (GİB) kullanan sistemler için optimize edilmiş yeni bir yaklaşım tasarımı hedeflenmiştir.

Bu çalışmada evrişimsel sinir ağı (ESA) yöntemi ile dijitalleştirilmiş el yazısı sınıflandırma yöntemi üç bölümde ele alınmıştır. İlk bölümde naif evrişimsel sinir ağının grafik işlem birimine uygulanması ile sınıflandırma elde edilmiştir.

İkinci aşamada grafik işlem birimleri için işlem katmanları paralelleştirilerek ve verinin paralel işlem katmanları için ayarlanıp eniyilenmiş bellek erişim şablonu yaklaşımla sınıflandırma hedeflenmiştir.

Son aşamada ise yöntemin birden fazla grafik işlem birimi üzerinde çalışması için yöntemde geliştirmeler yapılmıştır. Bu aşamada amaç, kullanılan grafik işlem birimi sayısı ile ters orantılı olarak evrişimsel sinir ağının eğitim süresinde gelişim sağlamaktır.

Anahtar Kelimeler: El yazısı numaraların sınıflandırılması, evrişimsel sinir ağı (ESA) , paralel işlem, öznitelik çıkarımı, çoklu grafik işlem birimi ile paralel işlem, grafik işlem birimi (GİB).

ACKNOWLEDGEMENTS

First of all, I would like to thank my consultant Dr. İbrahim Zincir who always supported me with his guidance in academic research, teaching on academic rules and techniques. I would not have completed this thesis without the support of his positive attitude when I was in despair. My supervisor's support for my orientation to the area I want to research without limiting my study area is the biggest factor in completing this study.

I would like to express my appreciation to my family for their limitless support during this thesis.

I would like a express my deepest thanks to my girlfriend Çağla Sarvan. Without her positive attitude and selfless behavior during the thesis period, this study would not have been successful.

I would like to thank Merve and Ezel Keç for their good friendship and support. Especially Merve's delicious coffees and positive attitude are important factors in keeping my morale high in this thesis.

The study would’t be successful without the names are given above.

Thank you.

Erinç CİBİL
İzmir, 2019

TEXT OF OATH

I declare and honestly confirm that my study, titled "OPTIMIZATION OF CONVOLUTIONAL NEURAL NETWORKS VIA GPU FOR CENTERALIZED DATA" and presented as a Master's Thesis, has been written without applying to any assistance inconsistent with scientific ethics and traditions. I declare, to the best of my knowledge and belief, that all content and ideas drawn directly or indirectly from external sources are indicated in the text and listed in the list of references.

Erinç CİBİL
Signature

....

September 7, 2019

TABLE OF CONTENTS

1. INTRODUCTION..1
2.HETEROGENOUS PARALLEL PROGRAMMING...5
2.1 PARALLEL PROGRAMMING..5
2.1.1 ANALYSIS OF PARALLEL ALGORITHMS...6
2.1.1.1 SPEEDUP...6
2.1.1.2 EFFICIENCY AND COST..6
2.1.1.3 SCALABILITY...6
2.1.1.4 COMPUTATION TO COMMUNICATION RATIO..7
2.1.2 CLASSIFICATION OF PARALLEL ARCHITECTURES..7
2.2 HETEROGENEOUS COMPUTING..8
2.2.1 MANY INTEGRATED CORES (MIC)...10
2.2.2 FIELD PROGRAMMABLE GATE ARRAY (FPGA)...10
2.2.3 GRAPHIC PROCESSING UNIT..10
2.3 COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA).......................................12
3. CONVOLUTIONAL NEURAL NETWORKS 19
3.1 CONVOLUTION LAYER..19
3.2 POOLING LAYER..22
3.3 FULLY CONNECTED LAYER...23
3.4 BACK PROPAGATION...24
4. DATA ORIENTED MULTI-GPU PROCESSING...27
4.1 NAIVE GPU IMPLEMENTATIONS...28
4.2 DATA ORIENTED GPU IMPLEMENTATION...32
4.3 DATA ORIENTED SCALABLE MULTI-GPU IMPLEMENTATION...........................39
5. CONCLUSIONS AND FUTURE RESEARCH...43

LIST OF FIGURES

FIGURE 1.1 : A) ARCHITECTURAL DESIGN OF THE FIRST MULTICORE CPU ARCHITECTURE FROM IBM
B) ILLUSTRATION OF INTELS HYPERTHREADING TECHNOLOGY. 2
FIGURE 1.2 : ILLUSTRATION OF PERCEPTRON FROM NEURAL NETWORKS AND LEARNING MACHINES...2
FIGURE 1.3 : IMPROVEMENT OF COMPUTER VISION WITH CNN FROM STANDFORD UNIVERSITY - CONVOLUTIONAL NEURAL NETWORK LECTURE MATERIALS 4
FIGURE 2.1: ILLUSTRATION OF A) SEQUENTIAL AND B) PARALLEL EXECUTION FROM PROFESSIONAL CUDA C PROGRAMMING...5
FIGURE 2.2 : ILLUSTRATION OF FLYNN TAXONOMY FROM PARALLEL PROGRAMMING: CONCEPTS AND PRACTICE...7
FIGURE 2.3 : CLASSICAL ARCHITECTURE OF HETEROGENEOUS COMPUTING.. .9
FIGURE 2.4 : A) MANY INTEGRATED CORE DEVICE. FIGURE. B) FIELD PROGRAMMABLE GATE ARRAY. C) GRAPHIC PROCESSING UNIT...........................9
FIGURE 2.5 : GPU BASED HPC ARCHITECTURE FROM PROFESSIONAL CUDA C PROGRAMMING..11
FIGURE 2.6 : MEMORY ACCESS PATTERN FOR GPU FROM PROFESSIONAL CUDA C PROGRAMMING...12
FIGURE 2.8 : THREAD ASSIGNMENT ON GPU WITH CUDA FROM PROFESSIONAL CUDA C PROGRAMMING...17
FIGURE 3.1 : LAYERS OF CONVOLUTIONAL NEURAL NETWORK..........................19
FIGURE 3.2 : ILLUSTRATION OF THE CONVOLUTION OPERATION ON 2-D MATRIX...20
FIGURE 3.3 : ILLUSTRATION OF THE CONVERGENCE DIFFERENCE BETWEEN RELU AND SIGMOID FUNCTION IN ALEXNET. SOLID LINE REPRESENTS RELU AND THE DASHED LINE REPRESENTS SIGMOID FUNCTION..................................21
FIGURE 3.4: LLUSTRATION OF MAX AND AVERAGE POOLING..............................22
FIGURE 3.5 : ILLUSTRATION OF BASIC FULLY CONNECTED NETWORK FROM NEURAL NETWORKS AND LEARNING MACHINES...23
FIGURE 3.6 : DISTRIBUTION OF LOSS WITH RESPECT TO OPERATORS.................24
FIGURE 4. 1 : MEMORY ACCESS PATERN OF NAVIE IMPLEMENTATION................32
FIGURE 4. 2 : DATA ORIENTATION OF INPUT IMAGE..33

FIGURE 4.3: EACH COLOR REPRESENTS INNER POSITION AND REPEATED COLOR PATTERNS REPRESENTS OUTPUT POSITIONS...35
FIGURE 4.4 : ILLUSTRATION OF THE DATA LAYOUT BEFORE MAX POOLING.....36
FIGURE 4.5: ILLUSTRATION OF OVERALL TASK DISTRIBUTION, THREAD CONCURRENCY AND DATA TRANSFER – COMPUTATION OVERLAP.....................38
FIGURE 4.6 : SPLITING INPUT IMAGE IS ILLUSTRATED...39
FIGURE 4.7: ILLUSTRATION OF OVERALL TASK DISTRIBUTION, THREAD CONCURRENCY AND DATA TRANSFER – COMPUTATION OVERLAP FOR MULTI-GPU..41
FIGURE 5.1: ELAPSED TIME WHILE COPYING DATA FROM HOST TO DEVICE MEMORY...44
FIGURE 5.2: ELAPSED TIME FOR THE CONVOLUTION OPERATION.......................44
FIGURE 5.3 : ELAPSED TIME FOR ACTIVATION LAYER. 45
FIGURE 5.4 : ELAPSED TIME FOR POOLING LAYER...45
FIGURE 5.5 : ELAPSED TIME FOR FULLY CONNECTED LAYER...............................46
FIGURE 5.6 : TOTAL DURATION OF FORWARD PASS OPERATION...........................46
FIGURE 5.7 : ACHIEVED SPEEDUPS..47

ABBREVIATIONS:

CNN	Convolutional Neural Networ
GPU	Graphic Process Unit
CPU	Centeral Process Unit
PCI-E	Peripheral Component Interconnect Express
SVM	Support Vector Machines
KT	Kernel Tricks
NVVP	NVIDIA Visual Profiler

CHAPTER 1
INTRODUCTION

Today, with the growth of problems, calculation of the solution needs more processing power. Problems are becoming more complex and the amount of data it needs in terms of the accuracy of problem solutions is increasing. In this context, the biggest share of the cake was taken by machine learning. Machine learning is essentially a paradigm using mathematical and statistical methods to process data, then make inferences from that data, and then make predictions on new data using these inferences. For example, machine learning, such as face recognition, spam detection in mails or social media advertisements, is encountered in many areas in current life. It may take years to obtain the required processing capability from a single processor system, depending on the size of the data and the complexity of the problem. The reason for this is that the processor which executes instructions in sequence cannot produce sufficient processing power.

With the advancement of technology, data processing techniques are accelerated at the hardware level. This increase can be examined in two important titles. The first reason for acceleration is the new hardware development techniques that emerged with the development of technology. These techniques have reduced the size of the processors. The shrinkage of the chip surfaces has also led to a shortening of the connection surfaces inside, as well as a reduction in Thermal Design Power (TDP) (Huck, 2011). In this way, the processors are able to operate in a stable manner by going to higher frequencies. The other part was not only to increase the number of cores due to the improvement in processor architectures, but also the idea of simultaneously processing multiple processes in the same core. The first multi-core processor was introduced by IBM in 1992 with IBM Power4 (Tendler et. al., 2002). IBM Power 4 has been designed as a structure that can scale and work together up to 32 process cores. The multi-core design is shown in picture 1-a. In order to run more than one job together in the same kernel, in 2002 Jon Stokes gave a detailed

description of the workpieces on the subject by dividing and processing as many processes as possible within the processor. Figure 1-b shows Stokes' diagram for hyper-threading.

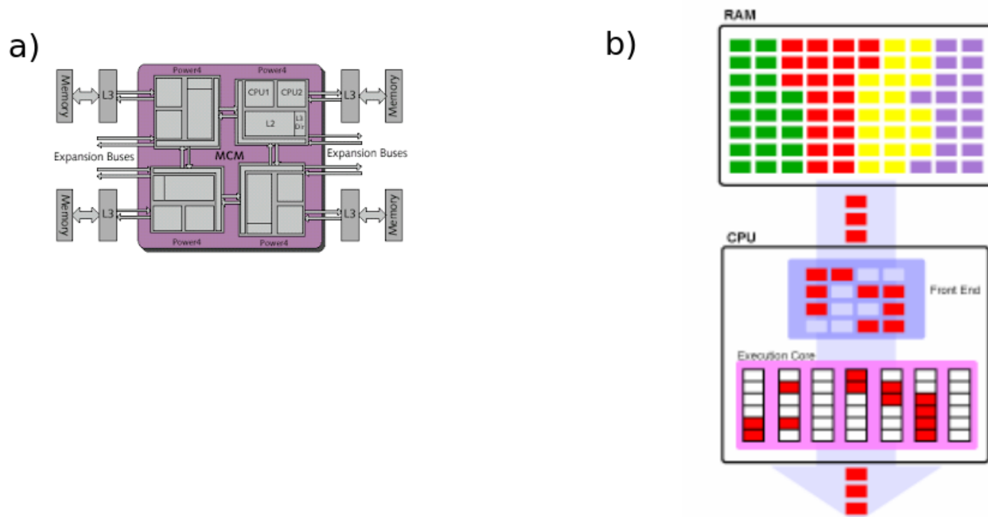


Figure 1.1 : a) Architectural design of the first multicore CPU architecture from IBM (Tendler et. al., 2002)
b) Illustration of Intels HyperThreading technology. (Strokes, 2002)

Development speed in both hardware and software is increasing day by day and the error rate is gradually decreasing. Thanks to these technological developments, more innovative solutions were produced and more efficient results were obtained.

The fact that people are fascinated by the ability to make smart machines is not new. The first seeds of machine learning were described by Rosenblatt in his article “The Perceptron” (1958). In general he described the basic structure of an intelligent system. In Figure 2, the structure of a perceptron is shown in simple form.

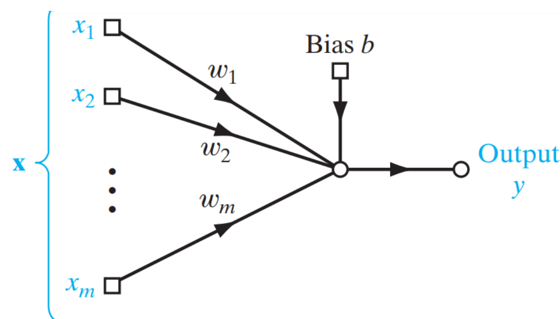


Figure 1.2 : Illustration of perceptron from Neural Networks and Learning Machines. (Haykin, 2009)

Artificial neural networks became popular again in the late 1980s with the introduction of back propagation and demonstration of good results, but this popularity didn't last long. When support vector machines (SVM) and kernel tricks (KT) appeared in the early 90s, artificial neural networks were drawn back into the darkness. SVMs gave better results. Increasing the number of layers in artificial neural networks did not improve results and back propagation does not work well on some models. The reasons for the failure in the 90s can be briefly attributed to the following, data sets were too small, computers had very little processing power, false initialization and false non-linear activation functions.

Since 2009 artificial neural networks have been experiencing the 3rd spring. First, in 2009, Hinton and his students developed a new training method for the first speech recognition problem. In that study, they used bot supervised and unsupervised layers together (Mohamed, Dahl & Hinton , 2009).

Then Hinton and his doctoral student Alex Krizhevsky, trained a 7 layer convolutional neural network using the same method that proposed in 2009. Today, this architecture is known as AlexNet (Krizhenvsky, Ilya and Hinton, 2012). In the competition Krizhenvsky participated with AlexNet and he won the first place with a 16.4% error rate. The second method was the combination of the best computer vision algorithms discovered until 2012 and the error rate was 25.8%.

The same competition gained popularity in the following years and deep learning methods have reduced the error rate to less than 5% today. In the same data set, the human error rate was around 5,1%.

In the following figure 1.3 shows the development of computer vision.

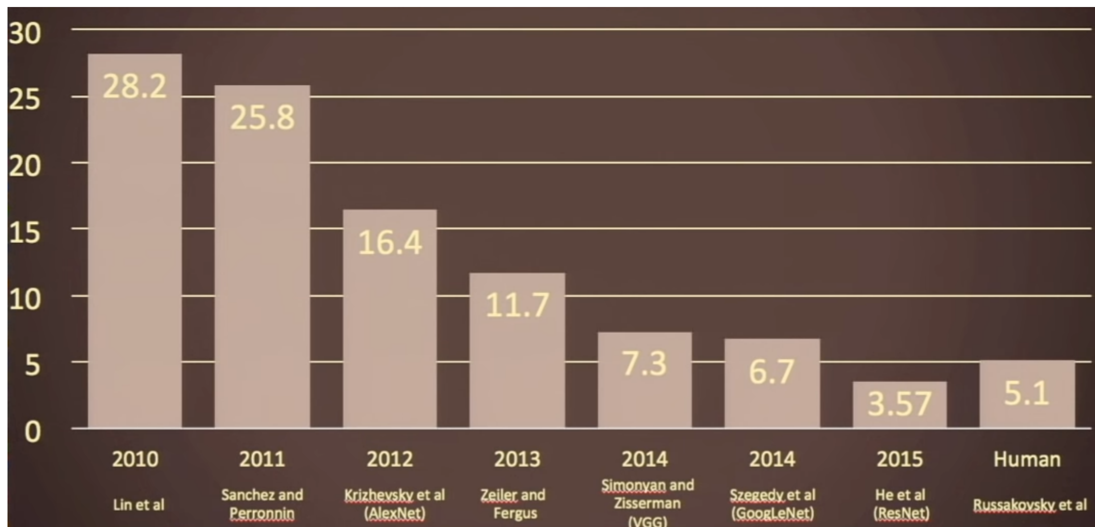
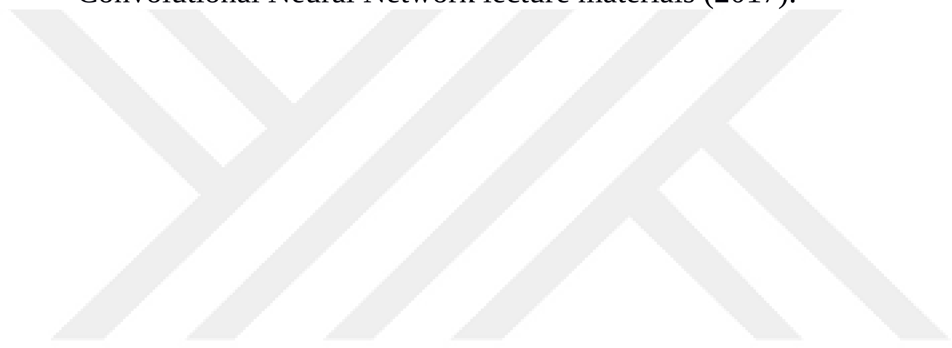


Figure 1.3 : Improvement of computer vision with cnn from Standford University - Convolutional Neural Network lecture materials (2017).



CHAPTER 2
HETEROGENEOUS PARALLEL COMPUTING

2.1 PARALLEL COMPUTING

As we described in the previous chapter, interests in parallel computation has been increased. One of the main reasons for the exponential increase in the computational time of the problems is that high quality algorithms related to the problem not be developed yet.

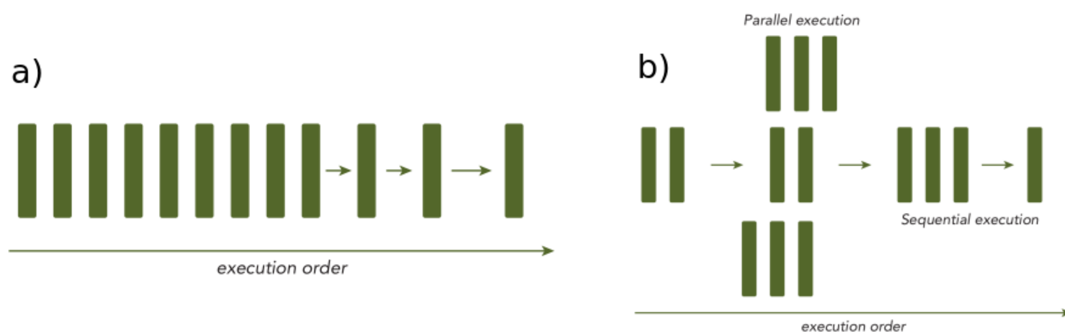


Figure 2.1: Illustration of a) sequential and b) parallel execution from Professional CUDA C Programming (See p.3 and p.4)

Parallel computing can be described as a type of computation which has many calculations can be carried out concurrently. Also main principle behind that is if the larger problem can be divided into smaller problems, those small problems can be solved simultaneously. Figure 2.1-a and 2.1-b shows the differences of sequential and parallel executions respectively and also in figure 2.1-b shows that the problem divided 3 subproblems which can be solved in parallel.

There are some basic concepts and terminologies in parallel programming. The concepts and terminologies are important for understanding and analyzing parallel computing.

2.1.1 ANALYSIS OF PARALLEL ALGORITHMS

According to Schmid et al. (2017) basic measurements of parallel algorithms are speedup, efficiency and cost, scalability and computation to communication ratio.

2.1.1.1 SPEEDUP

When a parallel algorithm is designed, new algorithm is expected to run faster then the sequential one. In parallel algorithms, speedup is defined as the comparison metric in the equation 1. In the equation speedup is denoted as **S,** computation time on sequential approach is denoted as **T(1)** and computation on parallel algorithm approach is called as **T(p).**

$$S=\frac{T(1)}{T(p)} \tag{1}$$

(Parallel programming concepts and practice p.2)

2.1.1.2 EFFICIENCY AND COST

Term of efficiency shows that how well the algorithm fits on more then one processor. When designing a parallel algorithm the goal is to catch 100% efficiency. 100% efficiency means that the algorithm is speeding up with the inverse ratio of processor number **p**. It’s also called as linear speedup (Some exceptions in the literature are increasing speed up more than p, those are called as super-linear speedups.). In equation 2 efficiency called as **E** is a ratio of speedup S over number of processors.

$$E=\frac{S}{p}=T\frac{(1)}{(T(p)\times p)} \tag{2}$$

(Parallel programming concepts and practice p.2)

2.1.1.3 SCALABILITY

Because of the algorithm can run on different type and number of processors the calculation of efficiency may not be enough to analyze. In those cases the algorithm tested on the various number of processors in the same system to understand how behaves the algorithm in parallel work load. That called scalability analysis. There

are two different type of scalability analysis, those are strong and weak scalability. When the number of the processor is changed but the data size is fixed, that is called strong scalability. On the other hand when number of processing power changed and also the data size varies that called weak scalability.

2.1.1.4 COMPUTATION TO COMMUNICATION RATIO

This metric analyses the algorithm for communication or computation overheads. The metric is calculated by the time spent for calculation over the time spent for communicating between processors. The higher ratios achieved when both speedup and efficiency gains.

2.1.2 CLASSIFICATION OF PARALLEL ARCHITECTURES

In general there are a lot of ways to classify parallel architectures but in 1972 Micheal J. Flynn published his taxonomy and it became the most used one. According to Flynn's Taxonomy parallel architectures are divided into 4 subgroups.

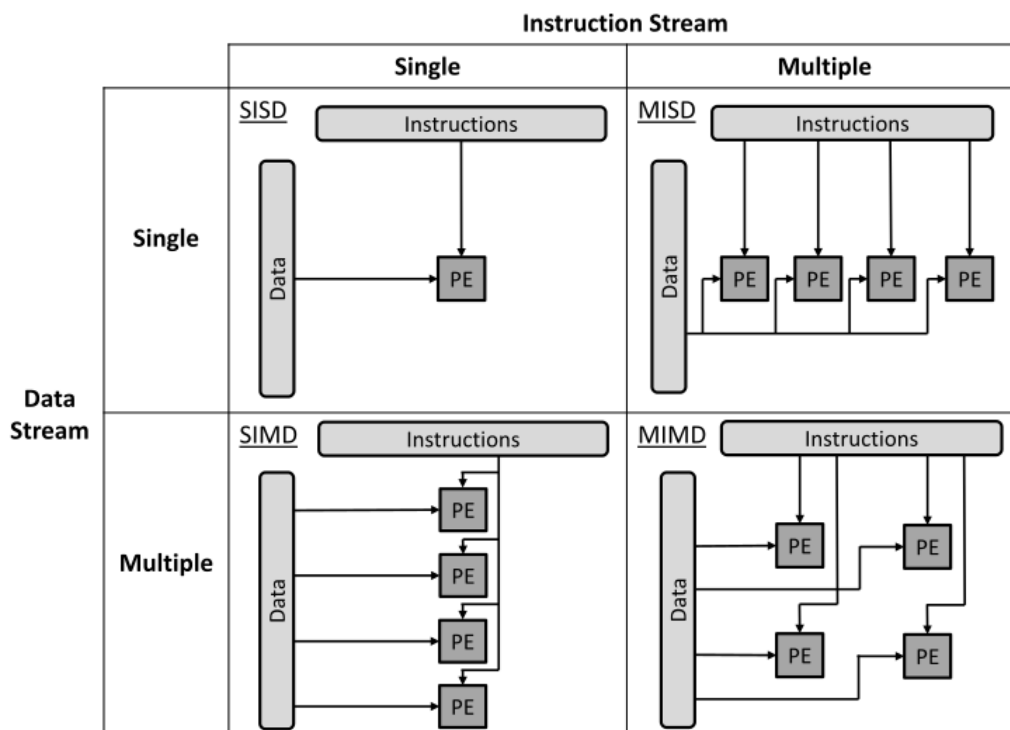


Figure 2.2 : Illustration of Flynn taxonomy from Parallel programming: concepts and practice (See p.59)

* Single Instruction Single Data (SISD) states to classical von Neumann architecture. In this architecture there is only one processor processing only one data. All of the instructions are executes sequentially.

* Single Instruction Multiple Data (SIMD) represents an architecture that can process multiple data with the same instruction.

* Multiple Instruction Single Data (MISD) refers to same data is processes more then one instructions. This type of parallelism is not commonly used except in pipelined architectures.

* Multiple Instruction Multiple Data (MIMD) parallelism is used when more than one data is processed by more than one instructions. While this operation executes, the whole processors and data streams work independently.

In figure 2.2 the above substances are respectively on the top left, bottom left, top right and bottom right.

2.2 HETEROGENEOUS COMPUTING

Actually, everyone wants an application to run faster. Related to this, for more than 25 years the 'Heterogeneous System Architecture (HSA) Foundation' has been working. The real proposition of the heterogeneous computing architecture is; not only to make it possible to run the program on multiple processors, as in parallel programming, but also that different types of processors can work together. (Gaster, Howes, Kaeli, Mistry & Schaa, 2012). In general, the structure of heterogeneous programming architectures is shown in Figure 2.3.

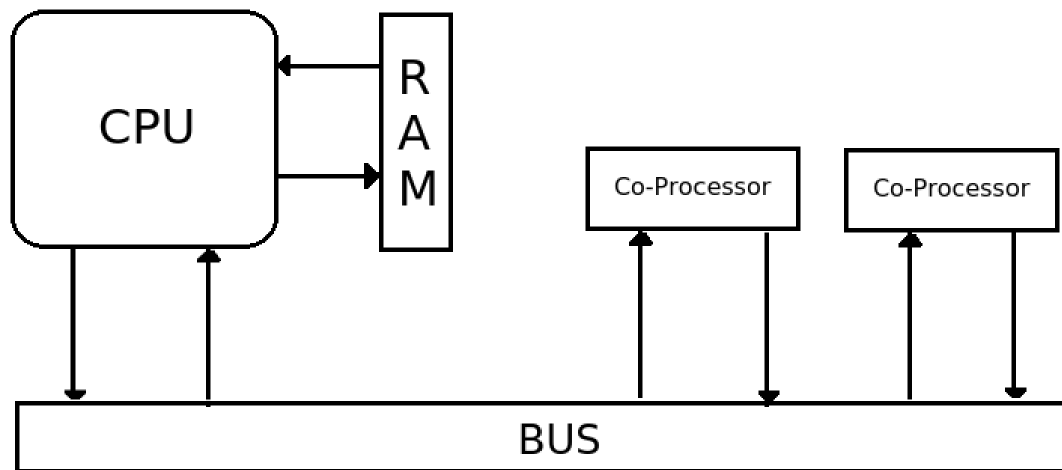


Figure 2.3 : Classical architecture of heterogeneous computing.

The structure basically works with the processor that performs the main job coordination and system tasks and the system memory reserved for that processor. The purpose of the central processing unit is to control the system as well as to assign tasks for the side processors. The side processors are responsible for fulfilling the task particles assigned by the central processing unit. The communication between the central processing unit and the peripheral processing units is carried out by the bus. Today, heterogeneous programming is most commonly seen in 3 different types. Figure 2.4 shows the three basic types of auxiliary processing units used in different heterogeneous programming structures. The use scenarios are different from each other, although they look identical to each other in appearance. The possible usage scenarios are mentioned below and the graphic processing unit (GPU) is detailed in the thesis subject.

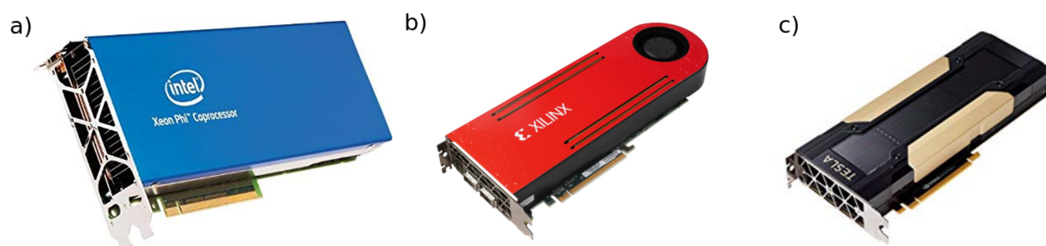


Figure 2.4 : a) Many integrated core device. Figure. b) Field programmable gate array. c) Graphic processing unit.

2.2.1 MANY INTEGRATED CORES (MIC)

MIC units are generally produced in the architecture of central processing units in computer processors. It is a combination of multiple cores with the same processing capacity as the CPU. Generally, these structures are used to solve problems requiring high processing power and complex directives (Atanassov et. al., 2017). Since the structures of the MICs and the central processing units are very close to each other, the C or C ++ language is preferred for such hardware.

2.2.2 FIELD PROGRAMMABLE GATE ARRAY (FPGA)

Simply “Field programmable gate array“ can be described as reconfigurable embedded circuits. The logic blocks in these circuits and the ability to configure the connections of these blocks have made it very easy to develop. In the general processing units that we use today, it accomplishes this task with a combination of multiple instructions to do a job. However, in FPGAs, since the business logic is defined directly to the hardware at the logic level, this process can be performed in very low cycles instead of communicating with various parts of the processing unit. FPGAs were generally used by design engineers. The hardware design was designed on FPGA and applied from the tests and then the prepared design was turned into a circuit board. With technological advances in the last decade, FPGA is now being used not only by design engineers but also for the rapid solution of major problems (Clive, M., 2004). Verilog or VHDL languages have been developed to make logical design with FPGAs.

2.2.3 GRAPHIC PROCESSING UNIT

For more than 10 years mainstream computers in HPC users prefer GPUs as a co-processor. The development of hardware technology over time has also made great progress in GPU technology. Thanks to the progressive GPU architectures, processing capacities are further increased and high energy efficiency is achieved. This development has increased the use of GPUs as a general purpose co-processor to solve parallel tasks. Although it is designed for parallel processing of graphics, it is in perfect harmony with SIMD structure due to its hardware architecture. HPCs with GPUs share the same architecture as other HPC types. In figure 2.5 GPU based

HPC architecture shows us that the system has memory and processor. In GPU systems GPUs also have their own memory and todays GPUs have more then 4000 cores. Most of the cores are just capable of multiplication and addition operations. In addition to that, there are some special function units are exists. The GPU and CPUs are connected via PCI-Express bus.

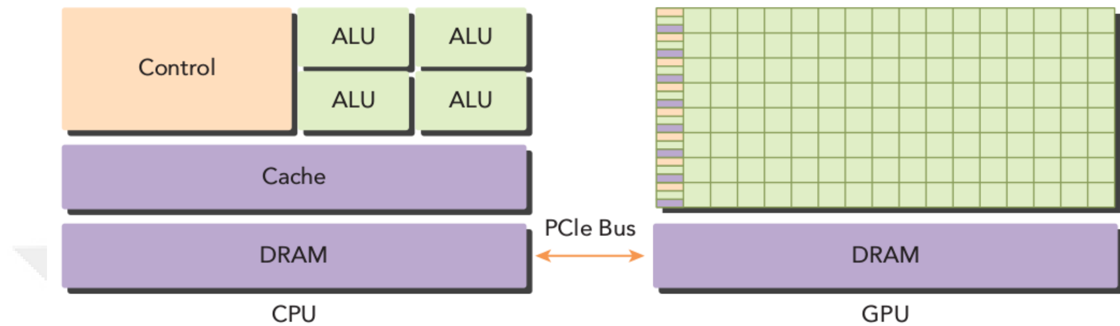


Figure 2.5 : GPU based HPC architecture from Professional CUDA C Programming (See p.9)

In heterogeneous computing programs have 2 separate parts. The CPU system called as host and GPUs are called as device. The host code runs on CPU and mostly it assigns tasks to the co-processors and does some organizations between parallel tasks. On the other hand device code runs on GPU. The tasks that assigned to GPU by CPU are computationally intensive and mostly those tasks exhibit huge amounts of data parallelism. For those kind of tasks GPUs are used to accelerate executions. Within this perspective if a task will process small amounts of data and has sophisticated control mechanisms, it is a good choice to process in CPU. Else if the task can be divided into small and simple tasks and the same operations are performed repeatedly for many data, it is a good choice to do so on the GPUs. The critical point is when working with GPUs data must be on device memory. Data is transferred from host memory to device memory via PCI-Express.

In GPU architecture there are thousands of small cores. Those cores are called streaming processors (SP). Streaming processors are not capable of doing complex tasks. Therefore tasks are assigned to groups of SPs and those are called streaming multi-processor (SM). In addition to global memory, each SM has its own L1 cash. Those L1 cashes are on the same chip with processors and also each of the SP has its

own registers. L1 cashes and registers are smaller than global memory. When data comes to GPU, task related data is transferred and cashed into SM, then SPs are read data from L1 to registers to do tasks. As a SP, reading data from L1 (aproximately 6-8 cycle) is so much faster then reading from global memory (aproximately 700 - 800 cycle). Therefore L1 called as on-chip memory and global memory called as off-chip memory.

In streming multi-processors data are read by batches and each streaming processor should work on consecutive part of the data. The data rate read and processed is called memory occupancy. In figure 2.6 memory occupancy is illustrated.

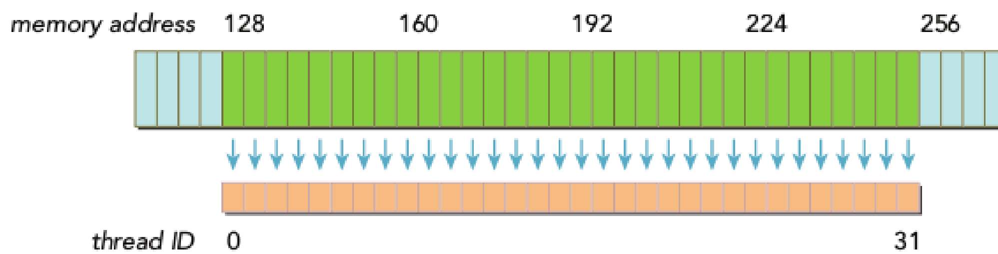


Figure 2.6 : Memory Access Pattern for GPU from Professional CUDA C Programming (See p.189)

Most common platforms for GPU based HPCs are Open Computing Language (OpenCL)(Guillon, A. J., 2015) and Compute Unified Device Architecture (CUDA). In 2008 OpenCL was invented by Apple and Khronos Group as an open source. In rest of the thesis CUDA platform is used due to the weakly support of OpenCLs on modern high end GPUs.

2.3 COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA)

End of 2006 Nvidia introduced its general purpose parallel computing platform that is called CUDA. The aim of the platform is solving complex computationally intensive problems on Nvidia gpus. It was first declared as a C and C++ extension, so that the software environment could be used in high-level programming languages. As a result of that, it overcomes low learning rate.

In CUDA C++ extension the code of the application has 2 parts. Host codes are standart C/C++ files and with respect to programming language those files have an

extention like “.h .c / .hpp .cpp”. On the other hand CUDA codes are stored in “.cu and .cuh” files. Host codes can be compiled by strandart C or C++ compiler, but device codes are compiled by Nvidia cuda compiler called nvcc.

Typical CUDA programs have three steps. First step is copy data from host to device, then process the data on the device. At the end of the process copy data back from device to host. Instead of memcpy in C or C++, CUDA uses cudaMemcpy command with an additional 4th parameter. The last parameter of the function declares the direction of the data. That direction can be one of the following types:

- cudaMemcpyHostToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

If the operation succesfully done, “cudaSuccess” message is returned.

In CUDA file device function is called as kernel. Instead of standart functions, kernels have an extra keyword right before the return type. Those keywords before return type restrict kernel behavior.

- “__global__ “means kernel callable from the host or devices with compute capability 3 and executes on device. Those kernels must have void return type. This keyword is mostly used as an entry point of device part.
- “__host__” means kernel can only be called by host and executes on host.
- “__device__” means kernel can only be called by device and executes on device. This keyword mostly used for dynamic parallelism and recursive algorithms.

The other important point on CUDA execution model is when launching a kernel there is a triple chevron notation.

“ kernelName<<<BlockNumber, ThreadNumberPerBlock>>>(args**) ”

All of the threads when launching a single kernel collectively are named as grid. Grids contains lots of thread blocks. Each of those blocks are assigned to SMs. The

block is a group of threads that can execute together. To navigate between threads there are three pre-initialized variables. These variables are threadIdx, blockIdx and blockDim. “threadIdx” is the number of threads in the block, “blockIdx is the number of blocks in the grid and “blockDim” is the number of thread spawned by the kernel for each block. Number of thread per block is limited and it can be maximum 1024. Thread id is calculated by the formula given below.

“ int t_id = blockIdx.x * blockDim.x + threadIdx.x ”

Block number and thread number per block are calculated by given data and number of SMs on device before launching the kernel. These two parameters directly effect the application performance. For example if there is a 32 data element for calculation, the elements can be grouped by 8 on each block and launch by four blocks in parallel. (Ex. kernel_name<<<4,8>>>(pointer of the elements)). The assigment is illustrated on figure 2.7.

Figure 2.7 : Data assignment for threads and blocks from Professional CUDA C Programming (See p.37)

Kernel launch parameters may vary depending on the GPU. GPUs are categorized according to SM technologies and hardware limits. This classification is called compute capability. The abbreviated table 1 below shows the classification of the hardware capacities .

Due to small memory sizes per SM, it is important to divide problems into small pieces and process that pieces on different SM parallely. When the kernel launched with the block numbers and threads per block parameters, kernel is assigned to separate SM with respect to block numbers. As shown in table 1 each SM can execute maximum 1024 consecutive thread if there are enough registers available. More than 1024 threads should be divide by different blocks. Equation 3 below shows how to calculate thread number and blocks.

$$NumberofBlocks = \lceil \frac{Numberofcalculations}{1024} \rceil \quad (3)$$

The basic formulation shown below in equation 4 calculates the number of blocks to be launched while executing kernel. Then the formula below uses the number of blocks for divide and distribute threads evenly.

$$Numberofthreads = \lceil \frac{Numberofcalculations}{NumberofBlocks} \rceil \quad (4)$$

Table 1: Cuda Compute Capability Classification from CUDA Programming Guide 10.1, 2019

	Compute Capability											
Technical Specifications	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.5**
Maximum number of resident grids per device (Concurrent Kernel Execution)	16	4	32				16	128	32	16	128	
Maximum dimensionality of grid of thread blocks	3											
Maximum x-dimension of a grid of thread blocks	$2^{31}-1$											
Maximum y- or z-dimension of a grid of thread blocks	65535											
Maximum dimensionality of thread block	3											
Maximum x- or y-dimension of a block	1024											
Maximum z-dimension of a block	64											
Maximum number of threads per block	1024											
Warp size	32											
Maximum number of resident blocks per multiprocessor	16				32							16
Maximum number of resident warps per multiprocessor	64											32
Maximum number of resident threads per multiprocessor	2048											1024
Number of 32-bit registers per multiprocessor	64 K			128 K	64 K							
Maximum number of 32-bit registers per thread block	64 K	32 K	64 K				32 K	64 K		32 K	64 K	
Maximum number of 32-bit registers per thread	63	255										
Maximum amount of shared memory per multiprocessor	48 KB			112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB	64 KB
Maximum amount of shared memory per thread block [26]	48 KB										96 KB	64 KB
Number of shared memory banks	32											
Amount of local memory per thread	512 KB											
Constant memory size	64 KB											

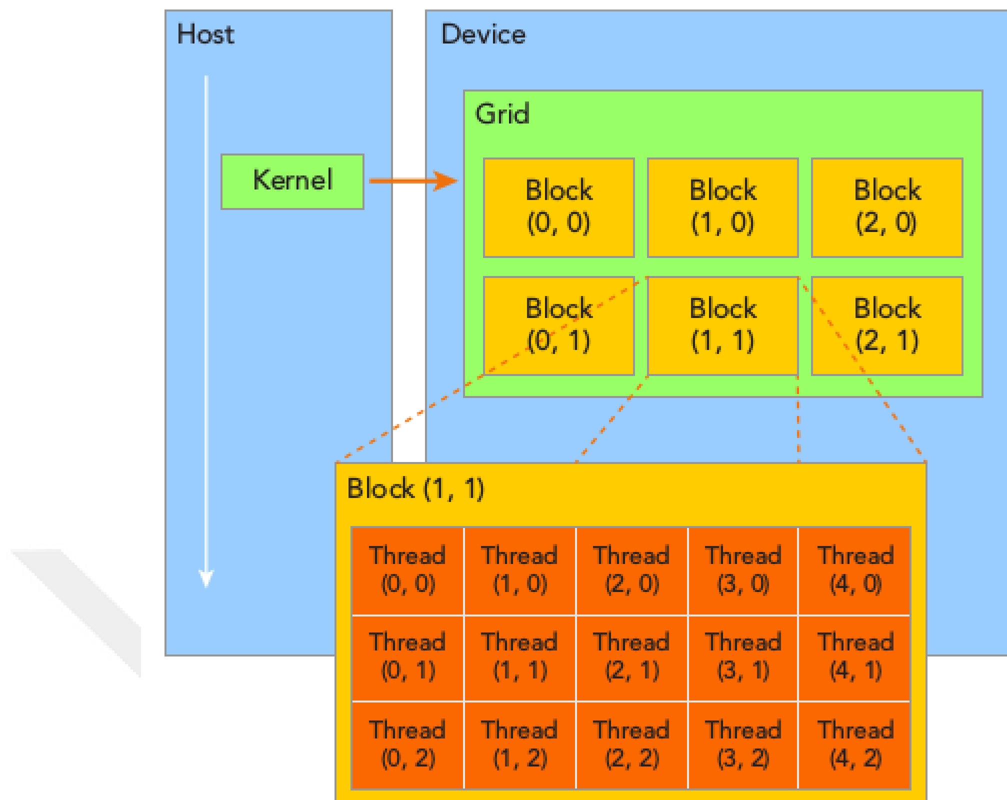


Figure 2.8 : Thread assignment on GPU with CUDA from Professional CUDA C Programming (See p.31)

While up to 1 thread can be execute on singe SM but only 32 different insttruction can be executed at the same time. The execution of the 32 thread called as warp. The thread assignment on the GPU shown in Figure 2.8.

CHAPTER 3

CONVOLUTIONAL NEURAL NETWORKS

In the last decade, the increadible development of artificial intelligence is gradually closing the gap between mechine and man. Thanks to the researchers and computer enthusiasts, the applications based on Computer Vision are using everyday routines when unlocking the phone with retina or tagging friends on photo in social media. Convolutional Neural Networks are one of the most important building blocks behind these achievements.

Basically convolutional neural networks are a combination of the convolution layer, pooling layer, activation layer and fully connected layer. Connection of the layers in CNN is shown in figure 3.1. In this chapter layers of the convolutional neural networks and connection between these layers are explained. The form of data will change while passing data from one layer to another layer. Convolutional neural networks can be used with every set of data which can be represented as single or multi dimensional matrices.

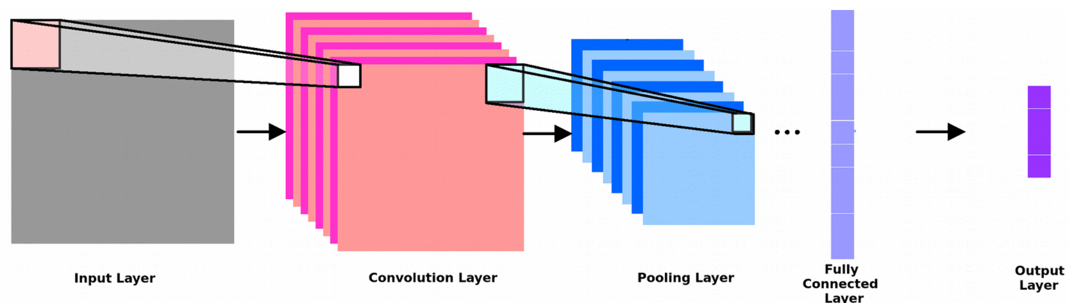


Figure 3.1 : Layers of convolutional Neural Network

3.1 CONVOLUTION LAYER

Real-life signals can be expressed as linear combinations of basic signals or cosine and sine functions. This representation of signals can be defined through Fourier Theory. The concept of convolution is mainly based on Fourier Theory. In a linear system, an output signal is obtained by applying a function to the input signal. This

output signal is expressed as the convolution of the function applied by the input signal. This structure is represented by the summation symbol in discrete-time systems, and by integral in continuous-time systems. Nowadays many applications and systems, use this basic convolution concept. For instance, the digital High Pass Filter or Low Pass Filter coefficients are shifted over the noisy audio signal along the time axis to reduce the noise which is called as convolutional sum of filter function and signal.

In figure 3.2 convolutional sum operation over input matrix 4 by 4 and filter matrix 2 by 2 is illustrated. Production of the convolutional sum is 3 by 3.

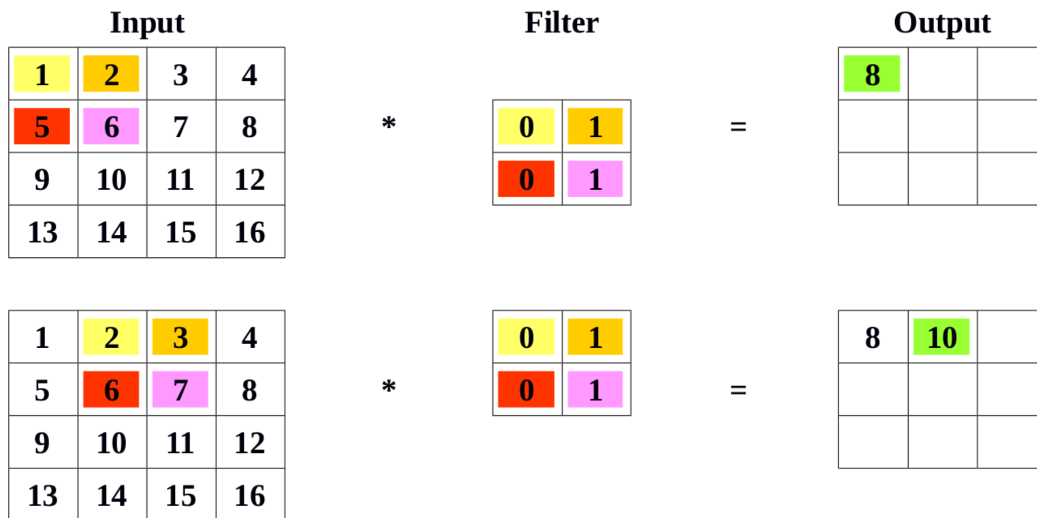


Figure 3.2 : Illustration of the convolution operation on 2-D matrix

In mathematical language, convolution of input I with the filter f is denoted by equation 5. Product of the convolution operation is denoted as G and the size of G depends on input matrix row and column size and also the stride s. Stride is the key point of number of elements shifted in input matrix for each element of the product G. Size of the G matrix is calculated by equation 6 given below.

$$G=[m,n]=(I*f)[m,n]=\sum_j \sum_k f[j,k]\, I[m-j,n-k] \qquad (5)$$

$$G_{output}=\left\lfloor \frac{(G_{input}-f_{input})}{s}+1 \right\rfloor \qquad (6)$$

Convolution process is important in terms of being the most basic process used for filtering and obtaining features from images and signals in digital image/signal

processing techniques. Features obtained from signals or images are used in classification algorithms. In the literature artificial neural networks are the most widely used classification algorithms which test the discriminative of features. CNN has emerged as an architecture that enables the use of the features obtained by convolution in artificial neural networks directly.

Basic multiplication and addition operations are applied in layers. The features obtained as a result of these operations show an exponential increase. To get rid of the linearity, activation functions are placed between layers. Nonlinear results are obtained between activation functions to reduce the slope of the gradient difference in the subsequent layers.

Due to gathered information from the article of Krizhevsky et. al., in this study rectified linear unit functions used as an activation function. The convergence of the error rate is dramatically increased against sigmoid function which is traditionally used in neural networks.

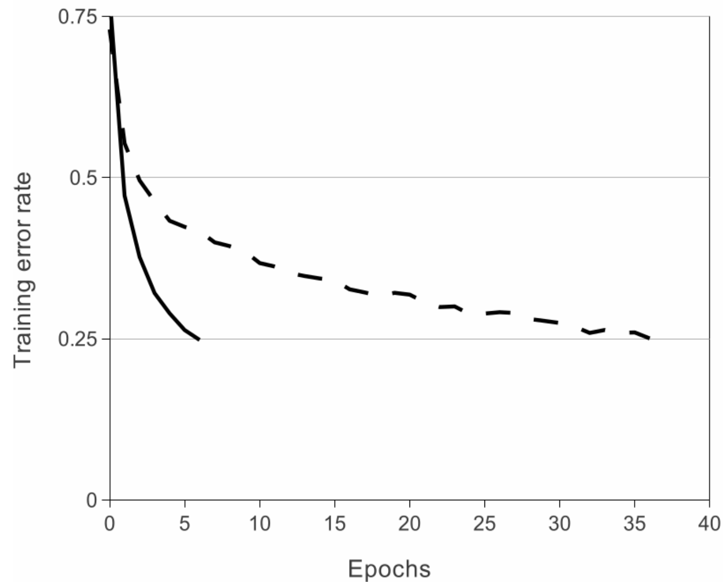


Figure 3.3 : Illustration of the convergence difference between ReLU and Sigmoid function in AlexNet. Solid line represents ReLU and the dashed line represents Sigmoid function. (Krizhevsky et. al. , 2012)

The formulation of the ReLU is shown in equation 7. Simply ReLU takes x as an input and compares with 0. If the number is bigger then zero, it returns the number x else returns zero.

$$y = max(0, x) \qquad (7)$$

3.2 POOLING LAYER

As long as in the convolution layer, pooling layer is also reduce the spatial size of the gethered future from previous layer. There are 2 reasons for using the pooling layer. The first reason is to reveal the dominant attribute within the field and the second reason is to reduce the cost of computation.

Only two types of pooling operations exist.

- Max pooling, returns the maximum value in the given area of the feature set.
- Average pooling, returns the average value in the given area of the feature set.

The formulation of the max pooling given equation 8. That formulation can be used both max and average pooling via changing operation type.

$$h_{i,j} = operation\{x_i + k - 1, j + l - 1, \forall 1 \leqslant k \leqslant m \wedge 1 \leq l \leq m\} \quad (8)$$

Mostly in CNN architectures max pooling is used due to it performs as a noise suppressant property. So can be said that mostly max pooling performs better than average. In figure 3.4 both of the max and average pooling operations are illustrated.

11	6	8	21	17	18
6	5	3	0	9	4
3	26	11	6	2	5
3	4	9	2	4	1
18	2	4	5	6	9
5	7	3	8	42	3

Pooling with 2x2 and stride by 2

Max

11	21	18
26	11	5
18	8	42

7	8	12
9	7	3
8	5	15

Average

Figure 3.4: llustration of Max and Average pooling

The layers of CNN described before can be combined to get more detailed features from raw image. In Zeiler and Fergus article, the effect of multiple consecutive convolution and pooling layers on the detail levels is described. Therefore, if deeper

details are desired, a deeper set of attributes can be extracted by replicating these first two layers.

3.3 FULLY CONNECTED LAYER

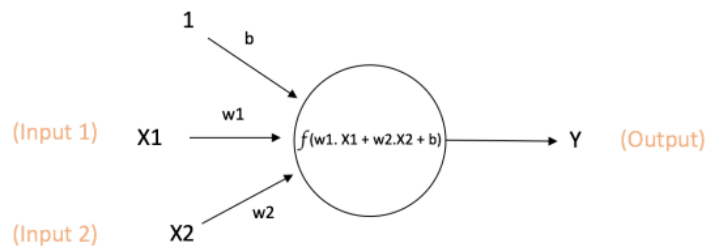
Fully connected layer is simply multi layer perceptron. Perceptrons have an input and output. In CNN fully connected layer, after flatten the production of the pooling layer, it used as an input of the fully connected layer. The input of the fully connected layer denoted as x in Equation 9 from Neural Networks and Learning Macines, Haykin (See p.27). Perceptron calculates the output with the set of weights w and adds bias b. Bias is important for early levels of the training. The output of the layer feeds the next layer as an input until output layer of the fully connected layers reached.

$$y_i = \left(\sum_j w_i \times x_j \right) + b \qquad (9)$$

The computationally efficient way of gathering non linear combination of the features in convolutional neural network is using fully connected network layer. When the output is obtained, the results are used for the classification after using logistic function. Logistic function is given equation 10.

$$out_{h1} = \frac{1}{1 + e^{-net_{h1}}} \qquad (10)$$

After calculations are obtained, the results of logistic function y can be used for classification the input raw image and calculation of error.



Output of neuron = Y= ƒ(w1. X1 + w2.X2 + b)

Figure 3.5 : Illustration of basic fully connected network from Neural Networks and Learning Machines. (Haykin, 2009)

3.4 BACK PROPAGATION

After doing all of the calculations, the results of the forward pass are obtained. In early levels of the training most of the time the results are incorrect. Reason of that, the network should be feed by better weights and bias. To get better multiplicative and additive parameters the network trained via back propagation algorithm. When the output results obtained by fully connected network, error rates of each output should be calculated (Equation 11). The total error rate of the network is sum of the errors (Equation 12).

$$E_i = \frac{1}{2} \times (target_i - output_i)^2 \qquad (11)$$

$$E_{total} = \sum_{i}^{j} \left(\frac{1}{2}\right) \times (target_i - output_i)^2 \qquad (12)$$

After that total error is obtained, the error rate should be distribute over previous layers to calculate Δw *and* Δx by applying the chain rule (Equation 13).

$$\frac{\partial E_{total}}{\partial w} = \frac{\partial E_{total}}{\partial out_o} \times \frac{\partial out_o}{\partial net_o} \times \frac{\partial net_o}{\partial w} \qquad (13)$$

By formula 13 negative gradient can be calculated and that gradient can be use for to train the network with respect to calculated error *E*. When back propagating distribution of the errors mathematical operations are important (see figure 3.6).

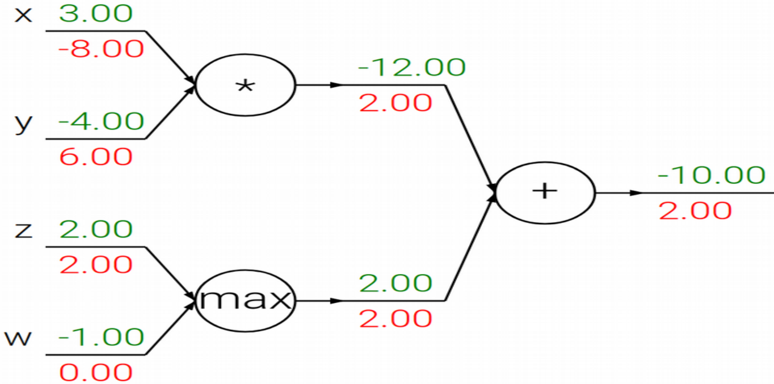


Figure 3.6 : Distribution of loss with respect to operators from Stanford Lecture Notes 2017

The mathematical operators changes the behavior while back propagation. Behaviors of the operations that used in figure 3.6 are explained below.

Add gate : While back propagating regardless off what their value is distributed evenly. As shown in figure 3.6, partial derivative of the loss value 2.00 distributed evenly.

Multiply gate: While back propagating multiplication gate distribution of the loss is calculated for first operand by multiplication of the loss with the second operand. As shown in the figure 3.6 distribution of the loss x is equal to y times loss and loss of y is equal to x times loss.

Max gate : Distribution over max gate is simply distribute the loss value just over the maximum number

CHAPTER 4
DATA ORIENTED MULTI-GPU PROCESSING

As known, convolutional neural networks are highly parallelizable structures, but first of all, all parameters of CNN should be defined for reproducibility and consistency in all experiments. In the table blow parameters of CNN are defined.

Table 2 : CNN parameters.

Parameter	Value
Image width and height	28, 28
Total data count	70000
Train data count	60000
Test data count	10000
Filter width and height	5, 5
Filter count	6
Filter initialization	Random, seed = 0, floating point between 0-1
Convolution stride	1
Pooling size	4,4
Activation Function	Rectified linear units (relu)
Fully connected layer	1
Output number of fully connected later	10
Fully connected layer weight initialization	Random, seed = 0, floating point between 0-1
Number of weights in fully connected layer	216
Mini batch size	2048
Stream batch size	128
Learning type	Backpropagation
Error function	Mini-batch gradient descent
Weight update	Average of the delta gradients

4.1 NAIVE GPU IMPLEMENTATIONS

Due to the nature of convolution operation, all parts can be calculated differently from each other. As mentioned in chapter 3, standard convolutional neural network has four steps. Algorithm 1 shows thet psuedo code of the CNN.

Since the optimization algorithm of CNN uses mini batch gradient descent, the network should be feed as much as the mini batch amount. This value is shown as m in algorithm 1. The native implementation refers while reach to the termination criteria, loop through the mini batch data. In mini batch data convolution operation has to be calculate. Calculation of the convolution operation can be parallel inside the mini batch. For every filter the convolution operation must be done repeatedly. Number of filter is denoted by f in the algorithm below.

Step	Task	Layer
1	Define input[m][iR][iC]	
2	Define o_conv[m][f][iR-fR+1][iC-fC+1]	
3	Define o_activation[m][f][iR-fR+1][iC-fC+1]	
4	Define o_pooling[m][f][(iR-fR+1)/pH][(iR-fR+1)/pW]	
5	Define o_fullyC[m][o]	
6	Define gradient_loss[o]	
7	**While** not termination criteria satisfied	
8	i ← 0	
9	**While** i < m	
10	j ← 0	
11	**While** j < f	
12	k ← 0	
13	**While** k < iR – fR + 1	
14	l ← 0	
15	**While** l ← iC – fC + 1	**Convolution Layer**
16	convolve(input[i][k][l], filter[j], o_conv), l = l + 1	
17	**End** while, k = k + 1	
18	**End** while, j = j +1	
19	**End** while	
20	j ← 0	
21	**While** j < f	**Activation Function**
22	k ← 0	
23	**While** k < iR – fR + 1	

```
24                l ← 0
25                While l ← iC – fC + 1
26                    relu(o_conv[i][j][k][l], o_activation), l = l + 1
27                End while, k = k + 1
28            End while, j = j + 1
29        End while
30        j ← 0
31        While j < f
32            k ← 0
33            While k <  (iR-fR+1) / pH
34                l ← 0
35                While l < (iC-rC+1) / pW
36                    maxPool(o_activation[m][f][k][l],o_pooling), l = l + 1
37                End while, k = k + 1
38            End while,  j = J + 1
39        End while
40        j ← 0
41        While j < o
42            k ← 0
43            While k < f
44                l ← 0
45                While l < (iR-fR+1) / pH
46                    x← 0
47                    While x < (iC-fC+1) / pW
48                        applyWeights (o_pooling[i][k][l][x], o_fC[m][o])
49                        x = x + 1
50                    End while
51                End while
52            End while
53        End while
54    End while
55    calculateLoss(o_fullyC)
56    i ← 0
57    While  i < m
58        calc_dC(calc_dA(calc_dP(calc_df(gradient_loss)))), i = i + 1
59    End while
60 End while
```

Pooling Layer

Fully Connected Layer

Back Propagation

Input parameters of convolve function *i* represents the ith image in the input batch. Filters are stored in arrays with size of f with column size fC and row size fR. The convolve operation can be done in parallel with the execution of kernel parameters MINIBATCH_SIZE blocks and CONVOLUTION_LAYER_OUTPUT_COLUMN_SIZE * CONVOLUTION_LAYER_OUTPUT_ROW_SIZE threads.

```
basicConvolve<<<MINIBATCH_SIZE, CONVOLUTION_LAYER_OUTPUT_COLUMN_SIZE *
CONVOLUTION_LAYER_OUTPUT_ROW_SIZE>>>(d_miniBatchInputData, d_filters,
d_miniBatchConvolutionOutput);
```

In kernel each thread calculates output matrix via given algorithm below.

```
__global__ void basicConvolve(float *input, float *filter, float *output){
	int innerPosition = threadIdx.x;
	int innerPositionRow = innerPosition / CONVOLUTION_LAYER_OUTPUT_ROW_SIZE;
	int innerPositionCol = innerPosition %
	CONVOLUTION_LAYER_OUTPUT_COLUMN_SIZE;
	int imageNumber = blockIdx.x;
	int imageStartPosition = imageNumber * INPUT_ROW_SIZE * INPUT_COLUMN_SIZE;
	int generalPosition = imageNumber * blockDim.x + innerPosition;
	float result = 0.0f;
	for (int k = 0; k < CONVOLUTION_FILTER_COUNT; ++k) {
		for (int i = 0; i < CONVOLUTION_FILTER_ROW_SIZE; ++i) {
			for (int j = 0; j < CONVOLUTION_FILTER_COLUMN_SIZE; ++j) {
				result += input[imageStartPosition
						+ innerPositionRow * INPUT_ROW_SIZE
						+ innerPositionCol * INPUT_COLUMN_SIZE
						+ i * INPUT_COLUMN_SIZE + j]
						* filter[k *
CONVOLUTION_FILTER_COLUMN_SIZE * CONVOLUTION_FILTER_ROW_SIZE + i *
CONVOLUTION_FILTER_COLUMN_SIZE + j];
			}
			output[(imageNumber * CONVOLUTION_FILTER_COUNT + k)
					* CONVOLUTION_LAYER_OUTPUT_ROW_SIZE
			*CONVOLUTION_LAYER_OUTPUT_COLUMN_SIZE
			+ InnerPositionRow
			* CONVOLUTION_LAYER_OUTPUT_COLUMN_SIZE
			+ innerPositionCol] = result;
		}
	}
}
```

In activation function basically loop through with every element of output vector of convolution layer and apply max(0, o_conv) function. Because of the each Sm can process maximum 1024 thread, thread number and block number must be re calculate. The number of elements in the o_conv array is 24*24*2048*6 = 7077888. therefore kernel must be launch with parameters 2048 blocks of 576 threads.

In pooling layer loop through for each image, every 4 by 4 area should be traversed and element with maximum value should be extracted. The output array of the pooling layer should be 2048 by 6 by 6 by 6.

Kernel is launch with given parameters below

```
maxPooling<<<MINIBATCH_SIZE, (CONVOLUTION_LAYER_OUTPUT_COLUMN_SIZE /
POOLING_COLUMN_SIZE) * (CONVOLUTION_LAYER_OUTPUT_ROW_SIZE /
POOLING_ROW_SIZE)>>>(d_miniBatchActivationOutput, d_miniBatchPoolingOutput);
```

Therefore kernel is launch with 2048 blocks of 36 threads and kernel implementation is given below.

```
__global__ void maxPooling(float *input, float *output){
	int innerPosition = threadIdx.x;
	int row = (innerPosition / 6);
	int col = innerPosition - row * 6;

	float temp = 0.0f;
	float max = 0.0f;
	for (int i = 0; i < POOLING_ROW_SIZE; ++i) {
		for (int j = 0; j < POOLING_COLUMN_SIZE; ++j) {
			temp = input[blockIdx.x * blockDim.x + row * 24 + col * 4 + i * 24 + j];
			if (temp > max){
				max = temp;
			}
		}
	}
	output[blockIdx.x * 216 + row * 6 + col] = max;

}
```

In the fully connected layer the kernel can be launched with parameters 2048 block and 216 threads. Basically we can assign all the connection of an image to a SM. Than every connection in fc can be done calculate parallel.

Fully connected kernel launched with give code below.

```
fullyConnected<<<MINIBATCH_SIZE, CONVOLUTION_FILTER_COUNT *
CONVOLUTION_LAYER_OUTPUT_COLUMN_SIZE / POOLING_COLUMN_SIZE *
CONVOLUTION_LAYER_OUTPUT_ROW_SIZE /
POOLING_ROW_SIZE>>>(d_miniBatchPoolingOutput, d_fcFilters, d_fcOutput);
```

As mentioned equation 9 in chapter 3.3 fully connected kernel can be easyly implemented.

```
__global__ void fullyConnected(float *input, float *weight, float *output){
	int point = blockIdx.x * blockDim.x + threadIdx.x;
	for (int i = 0; i < CLASS_COUNT; ++i) {
		output[blockIdx.x * CLASS_COUNT + i] = input [point] * weight[threadIdx.x *
			CLASS_COUNT + i];
	} }
```

4.2 DATA ORIENTED GPU IMPLEMENTATION

In proposed data oriented approach instead of reading batch of image data one by one. Data should be combined column by column. One of the problem in the naive approach is while reading the data. The memory access pattern in the naive approach access the data is not accessing repeated data for each repeated thread. For each thread 128b data stored on cash and 32 sequential data access by each thread in warp. Memory access pattern of naive approach is illustrated in figure 4.1 below. In the figure, every color in thread and data shows the assignment.

t	0	1	2	3	4	5	6	7	8	…	...	...	...	...	23				
d																			
r0	0	1	2	3	4	5	6	7	8						23	24	25	26	27
r1	0	1	2	3	4	5	6	7	8			...			23	24	25	26	27
r2	0	1	2	3	4	5	6	7	8						23	24	25	26	27
r3	0	1	2	3	4	5	6	7	8						23	24	25	26	27
r4	0	1	2	3	4	5	6	7	8						23	24	25	26	27

Figure 4. 1 : Memory access patern of navie implementation

When the thread access the data more then data number 128, thread should access the off-chip memory. The cost of accessing off-chip memory is between 600 to 800 cycle depending on status of the buffer.

In figure 4.2 shows the orientation of the data. When data is re constructed like in figure 4.2 each part of the each filter can be executed in-line with sequential threads in warp. Unlike CPUs, GPUs have very well optimized context switching mechanism on warps.

Input [3][4][4]

d						
		3	1	2	3	4
	2	1	2	3	4	8
1	1	2	3	4	8	12
	5	6	7	8	12	16
	9	1	11	12	16	
	13	14	15	16		

Input [4][12]

d	1	2	3	1	2	3	1	2	3	1	2	3
	1	1	1	2	2	2	3	3	3	4	4	4
	5	5	5	6	6	6	7	7	7	8	8	8
	9	9	9	10	10	10	11	11	11	12	12	12
	13	13	13	14	14	14	15	15	15	16	16	16

* Each color represents the part of different image.

Figure 4. 2 : Data orientation of input image

With this approach data is accessed with the %100 memory occupancy. Since every sequential thread accessing data in sequence in a warp the unnecessary global memory accesses count becomes zero.

The other optimization approach is streaming data and the computation. CUDA streams are simple FIFO structure like queue. The main difference is cuda default stream is defined by blocking queue due to protect data against common data race conditions. Instead of default stream, CUDA can support explicitly defined non-blocking streatms. With the minimum sixteen non-blocking streams, the data transfer and the computation can be overlapped. While half of the data transfers the other half of the data can be process by the SMs.

When using this approach all of the input and output arrays of the network should be recalculate. For example with the data for total 2048 image in 16 streams with 28 rows and 28 columns new input array should be 28 rows and 128 * 28 = 3584 columns. The array size is increased to 28 * 1792 = 100352 times 16 elements. For an output array 128*24*24 = 73728 times 16 elements. While maximum thread number is 1024, the kernel should be launched with 72 blocks and 1024 thread for each stream. While this stream processes data, the other stream can be copy next 64 image from host memory to device memory. Due to cudaMemcpy overhead less then 0.01 MB of transfers reduces the application performance.

To launch convolution kernel loops are used for traversing filter position, changing streams. Kernel launch approach is given below. With that approach basically every

position of the filter elements and input elements are multiplied. Instead of creating 25 different result array, adding each identical matrices gives the result of the convolution operation. By that reason that can be placed inside the kernel for saving GPU memory. In addition to that, for every stream an event is recorded and it helps for scheduling. With that events, streams are paused until their data transfered.

```
for (int k = 0; k < NUMBER_OF_STREAMS; k++) {
	for (int i = 0; i < CONVOLUTION_FILTER_ROW_SIZE; i++) {
		for (int j = 0; j < CONVOLUTION_FILTER_COLUMN_SIZE; j++) {
			if (!transfered[k]){
				loadMiniStreamBatchToGPU(miniBatchCounter, k,
					streams[k], d_miniBatchInputData);
				cudaEventRecord(events[0][k], streams[k]);
					transfered[k]=true;
			}
			cudaStreamWaitEvent(streams[k], events[0][k],0);
			convolve<<<72, 1024, CONVOLUTION_FILTER_COUNT*sizeof(float),
streams[k]>>>(d_miniBatchInputData + (k * streamLength) + (i * streamRowLengthForInput) + (j *
MINIBATCH_SIZE), d_filters+(i * CONVOLUTION_FILTER_COLUMN_SIZE + j),
d_miniBatchConvolutionOutput + (k * streamConvolutionOutputLength) + (i *
streamRowLengthForConvolution) + (j * MINIBATCH_SIZE), streamConvolutionOutputLength);
		}
	}
}
```

As seen in the kernel launch parameter shared variable array used for data access. Every thread should be accessed same position of a filter for all filters. Therefore shared variable is defined with the size of 6 * sizeof(float). In convolution kernel shared variable array is filled and then all threads are synchronized to make sure of every thread gets the filter element. After that computation of each elements can be calculated by the kernel given below.

```
__global__ void convolve(float *input, float *filterX, float *output, int
	streamConvolutionOutputLength){
	int rowNumber = blockIdx.x / 3;
	int position = rowNumber*INPUT_COLUMN_SIZE*INPUT_ROW_SIZE + threadIdx.x;
	if (input[position] != 0.0f) {
		extern __shared__ float fx[CONVOLUTION_FILTER_COUNT];
		for (int i = 0; i < CONVOLUTION_FILTER_COUNT; ++i) {
			fx[i] = filterX[i];
		}
		__syncthreads();
		for (int i = 0; i < CONVOLUTION_FILTER_COUNT; ++i) {
			atomicAdd(&output[position + i * streamConvolutionOutputLength] ,
output[position + i * streamConvolutionOutputLength] + input[position] * fx[i]);
		}
	}
}
```

After every layer operation gpu streams should be synchronize.

```
for (int i = 0; i < NUMBER_OF_STREAMS; ++i) {
		cudaStreamSynchronize(streams[i]);
}
```

This study is based on maximize memory occupancy, but max pooling operation not fits very well on that. Because of that while activation operation not only ReLU applied but also data layout redesigned for pooling operation. ReLU kernel launched with the code below.

```
RELU<<<activationBlockSize,activationThreadSize, 0, streams[i]>>>
(d_miniBatchConvolutionOutput + (i * streamReluOutputLength), d_reluOutput, i,
streamReluOutputLength, streamReluArrayLength);
```

If the output length of each stream is a multiplication of maximum thread per block, block size is calculated by output length over 1024 else output length over 1024 plus one. Also if output length is larger than 1024, thread number is seted 1024 else thread number remains as output length.

In kernel each position of the input matrix is addressed to a position both inner area and outer area. That means inner area position gives the position of the elements which are assigned to new data layout and output area position gives the position of the elements which are compared together. In figure 4.3 inner and outer positions are illustrated.

	I1E1	I2E1	I3E1	I1E2	I2E2	I3E2	I1E3	I2E3	I3E3	I1E4	I2E4	I3E4	
	I1E5	I2E5	I3E5	I1E6	I2E6	I3E6	I1E7	I2E7	I3E7	I1E8	I2E8	I3E8	
	I1E9	I2E9	I3E9	I1E10	I2E10	I3E10	I1E11	I2E11	I3E11	I1E12	I2E12	I3E12	

Figure 4.3: Each color represents inner position and repeated color patterns represents output positions.

With respect to given information above kernel code is implemented below.

```
__global__ void RELU(float *input, float *output, int streamNumber, int streamOfset,
	int innerArrayOfset){
	int element = blockIdx.x * blockDim.x + threadIdx.x;
	if(input[element] > 0.0f){
		int threadOuterRow = element / maxPoolingOuterRowLength;
		int temp = element - (threadOuterRow * maxPoolingOuterRowLength);
		int threadInnerRow = temp / maxPoolingInnerRowLength;
		temp = temp - (threadInnerRow * maxPoolingInnerRowLength);
		int threadOuterCol = temp / maxPoolingOuterColumnLength;
		temp = temp - (threadOuterCol * maxPoolingOuterColumnLength);
		int threadInnerCol = temp / maxPoolingInnerColumnLength;
		temp = temp - (threadInnerCol * maxPoolingInnerColumnLength);
		int arrayNum = threadInnerRow * POOLING_COLUMN_SIZE + threadInnerCol;
		int position = (threadOuterRow * maxPoolingOuterRowPosition + threadOuterCol)
* MINIBATCH_SIZE + temp;
		output[streamNumber * streamOfset + arrayNum * innerArrayOfset + position] =
input[element];
	}
}
```

	I1E1	I2E1	I3E1	I1E3	I2E3	I3E3	I1E2	I2E2	I3E2	I1E4	I2E4	I3E4	
	I1E5	I2E5	I3E5	I1E7	I2E7	I3E7	I1E6	I2E6	I3E6	I1E8	I2E8	I3E8	
	I1E9	I2E9	I3E9	I1E11	I2E11	I3E11	I1E10	I2E10	I3E10	I1E12	I2E12	I3E12	

Figure 4.4 : Illustration of the data layout before max pooling.

With respect to figure 4.4, now it can be easily compared each row with next row like reduction technique. After comparision the results gives the maximum elements of each pooling area as a vector with new layout. Implementation of the max pooling is given below and the kernel launch parameters are calculated with the same way in ReLU launch parameters.

```
__global__ void maxPooling (float *input, float *output, int streamNumber, int streamSize,
        int poolArraySize){
        int element = blockIdx.x * blockDim.x + threadIdx.x;
        const int total = POOLING_COLUMN_SIZE * POOLING_ROW_SIZE;
        float compOperands[total];
        for (int i = 0; i < total; i++){
                compOperands[i] = input[i * poolArraySize + element];
        }
        int div = total;
        for (int i = 1; i <= 4; i++){
                div = div / 2;
                for(int j = 0; j < div; j++ ){
                        compOperands[j] = compOperands[2*j] > compOperands[2*j+1] ?
compOperands[2*j] : compOperands[2*j+1];
                }
        }
        output[streamNumber * streamSize + element] = compOperands[0];
}
```

After pooling operation, fully connected layer remains. Due to nature of the fully connected layer it directly fits for maximum memory occupancy. But the filters does not fits well. For that reason shared memory is allocated with the size of CLASS_COUNT times sizeof (float) in GPU. Then calculation done with respect to given code below.

```
__global__ void fullyConnected(float *input, float *fcfilter, float *output, int streamNumber,
int streamSize){
        int element = blockIdx.x * blockDim.x + threadIdx.x;
        if(input[element] > 0.0f) {
                extern __shared__ float fxFc[CLASS_COUNT];
                for (int i = 0; i < CLASS_COUNT; ++i) {
                        fxFc[i] = fcfilter[blockIdx.x * CLASS_COUNT + i];
                }
```

```
			__syncthreads();
			for (int i = 0; i < CLASS_COUNT; ++i) {
					output[streamNumber*streamSize + i * blockDim.x +
						threadIdx.x] = input[element] * fxFc [i];
			}
		}
}
```

With that approach thread concurrency and data transfer – calculation overlap is achieved. To visualize that results NVIDIA Visual Profiler (NVVP) tool is used. Illustration of the task distribution and concurrency shown in figure 4.5.

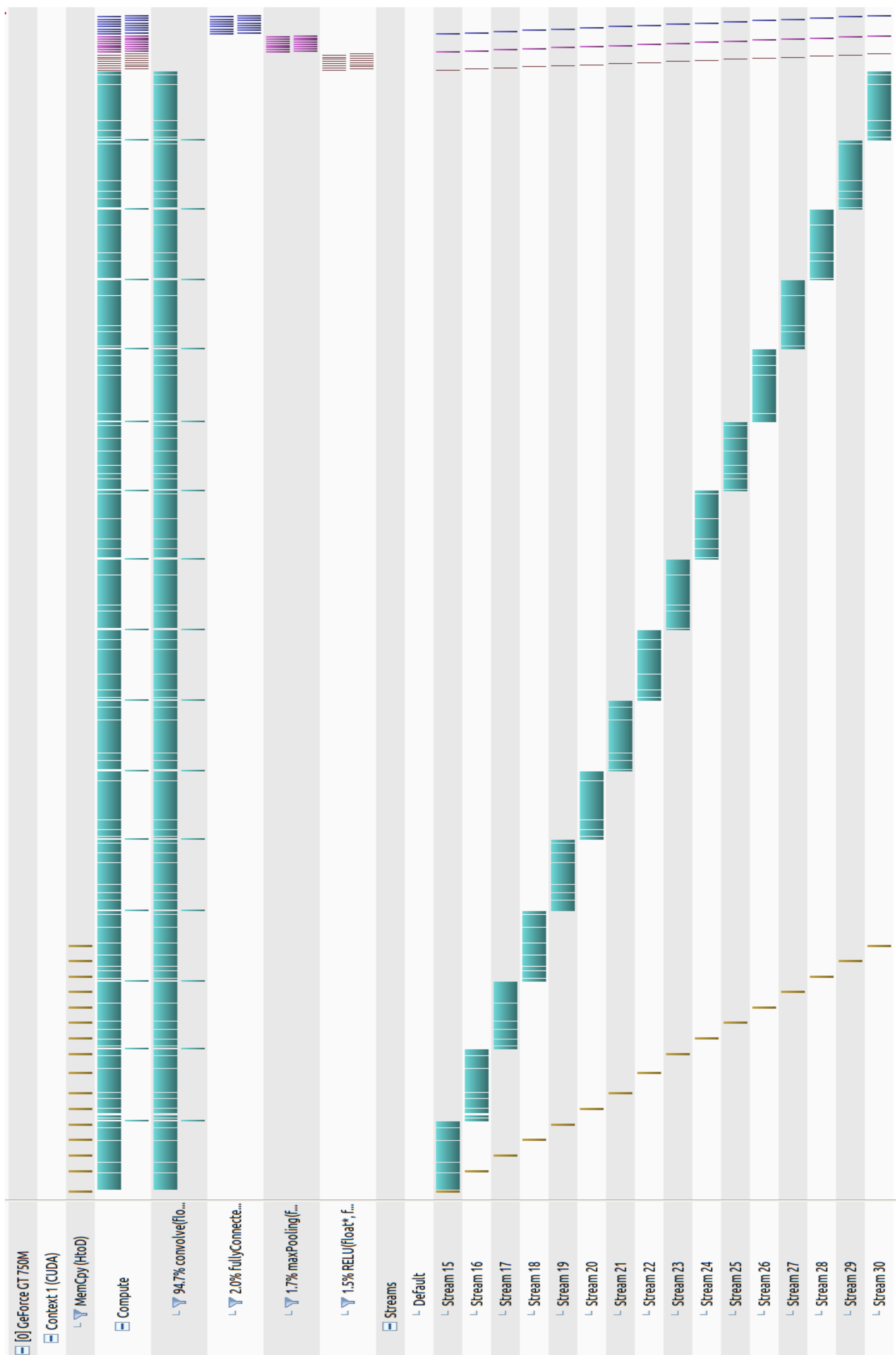


Figure 4.5: Illustration of overall task distribution, thread concurrency and data transfer – computation overlap.

4.3 DATA ORIENTED SCALABLE MULTI-GPU IMPLEMENTATION

While scaling the data oriented approach in multiple GPUs, data reconstructed like in single GPU approach with the small difference. Due to the convolution operation data can not be split in the middle. When split 28 by 28 data first the output layer of the convolution layer must be calculated. When 28 by 28 image convolved with 5 by 5 filter the output is 24 by 24. When dividing data 2 piece due to memory access pattern optimization, it should be split via rows. Equation 14 calculates the exact position to the last row number for input image.

$$LastRowNumber = \frac{inputRowSize - filterRowSize + 1}{2} + filterRowSize - 1 \quad (14)$$

And the other part of the input first row number calculated by equation 15

$$FirstRowNumber = LeastRowNumber - filterRowSize + 1 \quad (15)$$

After that division complete, rows between first row of the image part 2 and last row of the image part 1 copied both side due to convolution operation. Split operation of the input image is illustrated in figure 4.6.

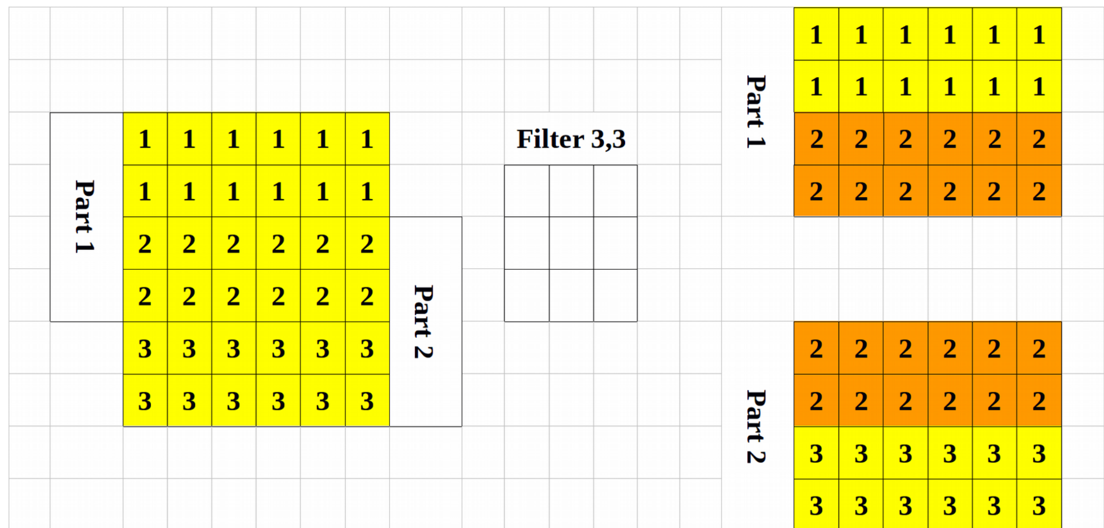


Figure 4.6 : Spliting input image is illustrated

After pooling layer in multi GPU approach each GPU holds half of the extracted futures of the entire network. Before the calculation of error rates, both GPUs should be synchronized. While synchronization of the data instead of classical method writing data to host memory after that copy data to other GPU, with the direct access

property of the GPUs, data can be transferred via PCI-Express with using more PCI-Express lanes. With direct GPU access 12 GBps transfer speed is reached. When the recursive calculations of the back propagation process ends, simply summation of the all delta matrices with each other results the correct value of the delta weights of the entire network.

Last important point of multi GPU process is the distribution of the tasks to GPUs with breadth first approach. With breadth first task assignment GPUs can hide kernel launch overheads between computation time.

Illustration of the task distribution, concurrency and data tranfer – computation overlap for multi-GPU shown in figure 4.7.

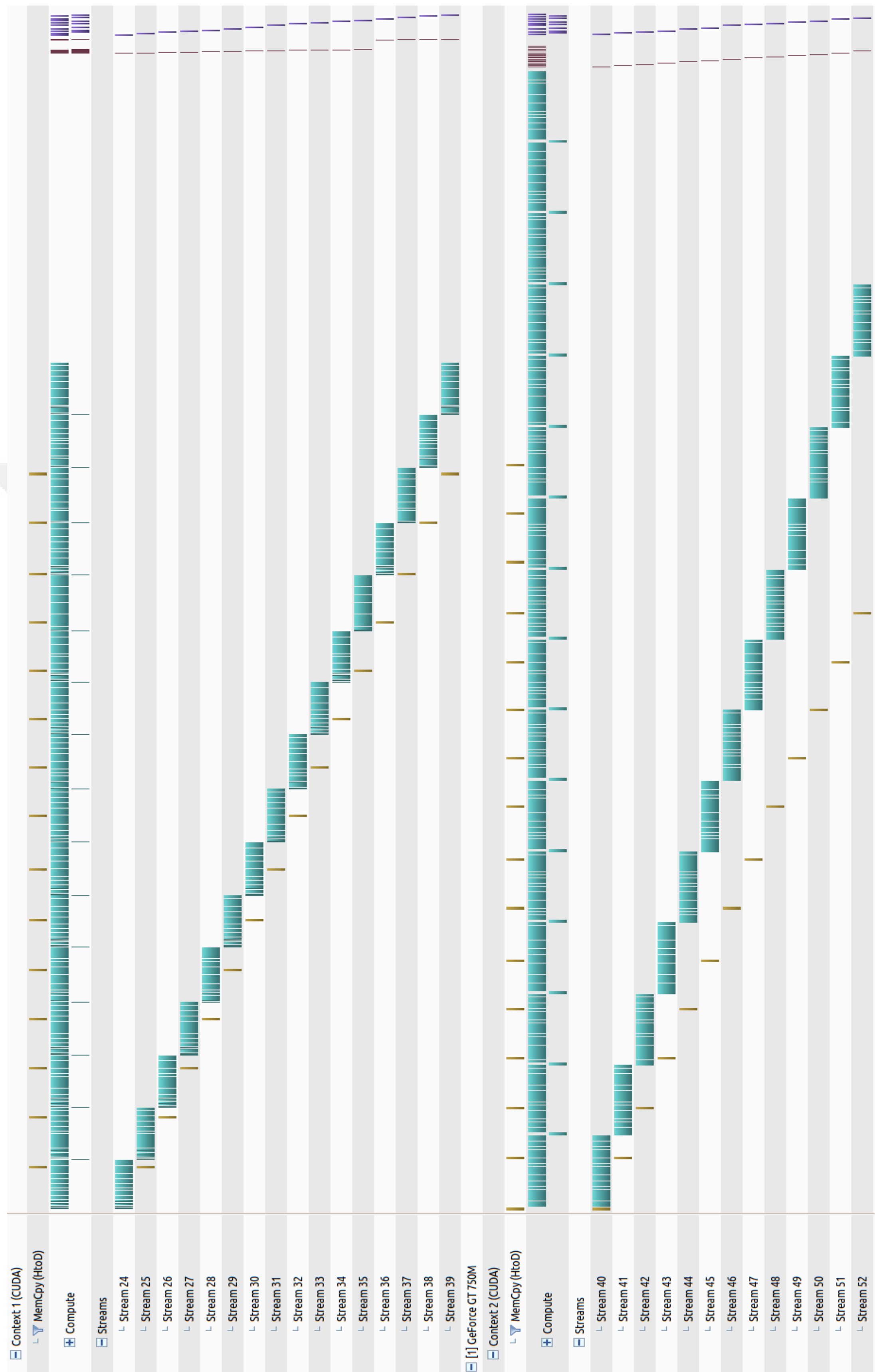


Figure 4.7: Illustration of overall task distribution, thread concurrency and data transfer – computation overlap for multi-gpu.

CHAPTER 5
CONCLUSIONS AND FUTURE RESEARCH

As conclusion the convolutional neural network algorithm is optimized for memory access pattern , device memory hierarchy, thread occupancy and concurrency obtained. With that improvement in total 4.29 times speedup gained. Table 3 shows the speedup for each layer of the convolutional neural network and memory copies.

Operation	**Naive**	**Optimized**	**Multi-GPU Optimized**
Memory copies	1 ms.	0.06 ms.	0.018 ms.
Convolution	131.3 ms.	30.47 ms.	15.96 ms.
RELU	1.78 ms.	0.33 ms.	0.17 ms.
Max Pooling	0.36 ms.	0.55 ms.	0.3 ms.
Fully Connected Network	2.65 ms.	0.58 ms.	0.3 ms.
Total	137.09 ms.	31.99 ms.	16.748 ms.
Speed Up	X 1	X 4.29	X 8.19

The configuration of the computer used for testing are Intel i7 4700MQ, 8GB ram and 2 x Nvidia GT 750M 2GB. But in tests only one GPU is used.

For the multi gpu efficiency is calculated with the formula given chapter 2.

$$Efficiency=\frac{T_{(multi-GPU)}}{Number\ of\ GPU\times T_{(single-GPU)}}=\frac{8.19}{(2\times 4.29)}=0.96$$

Only 4% loss is occurred while computing 2 GPU and achieved efficiency is 96%.

With respect to approaches described in chapter for, following results are gained. Results of the both naive, optimized and multi-gpu optimized methods are detaily given figures below

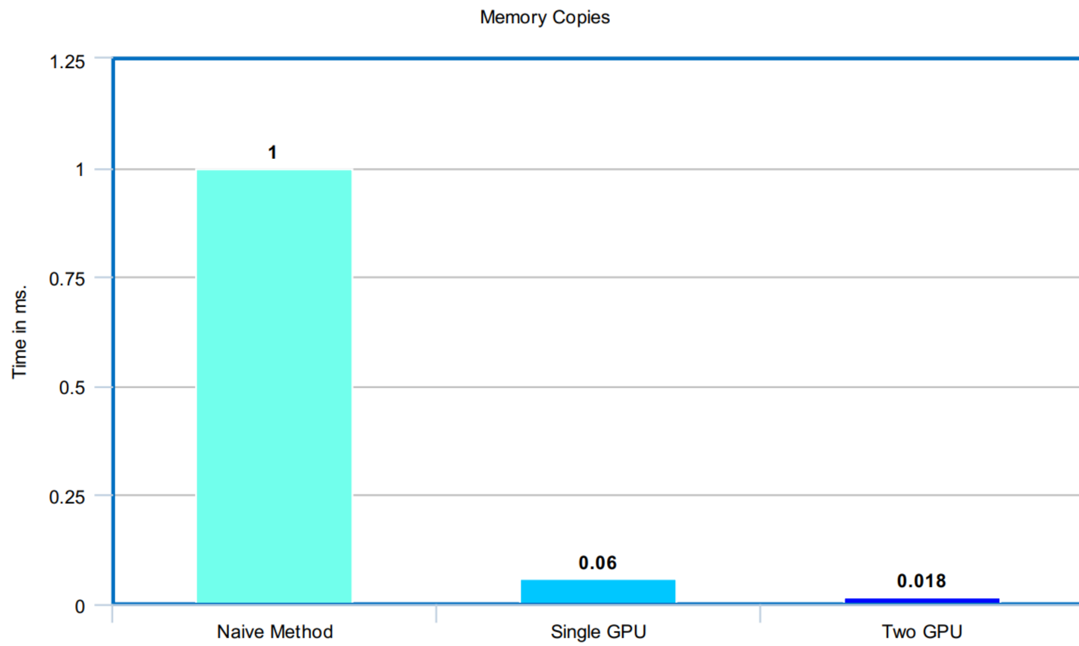


Figure 5.1: Elapsed time while copying data from host to device memory.

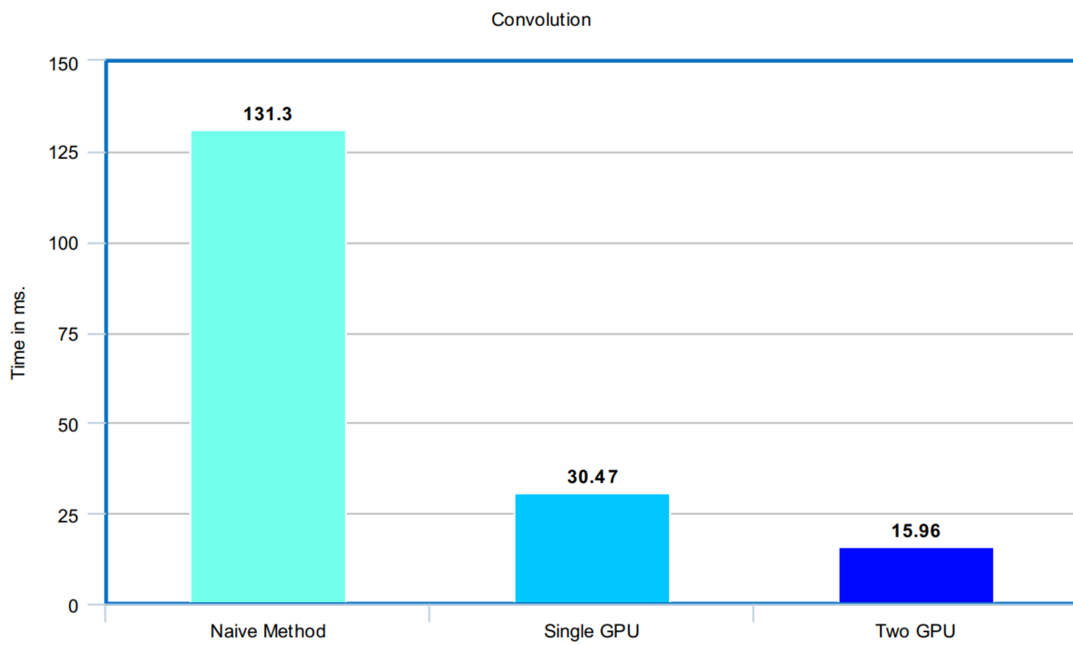


Figure 5.2: Elapsed time for the convolution operation.

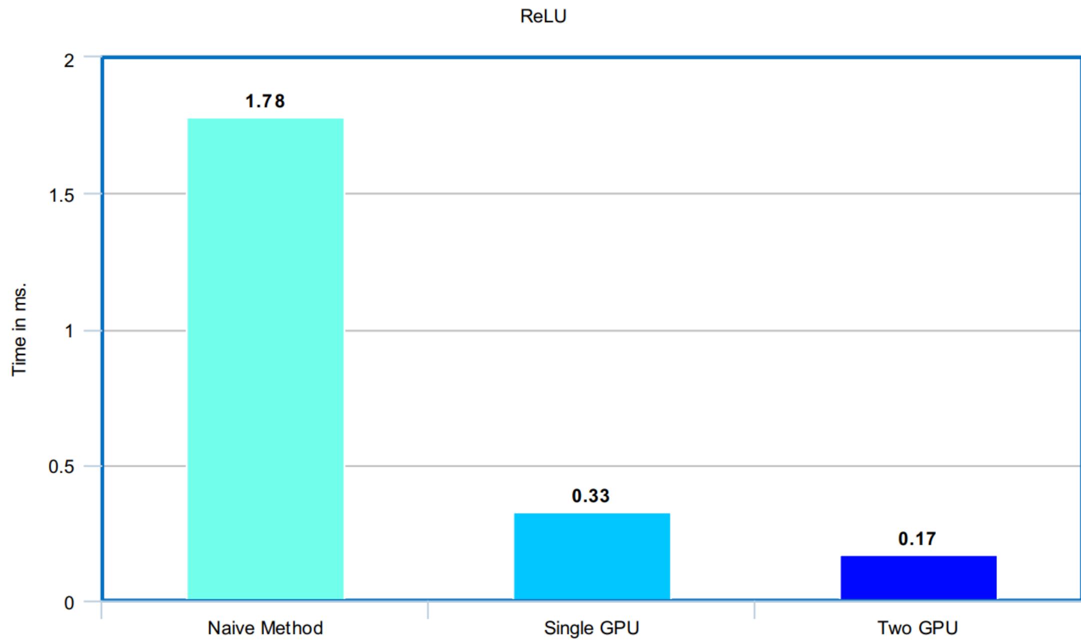


Figure 5.3 : Elapsed time for activation layer.

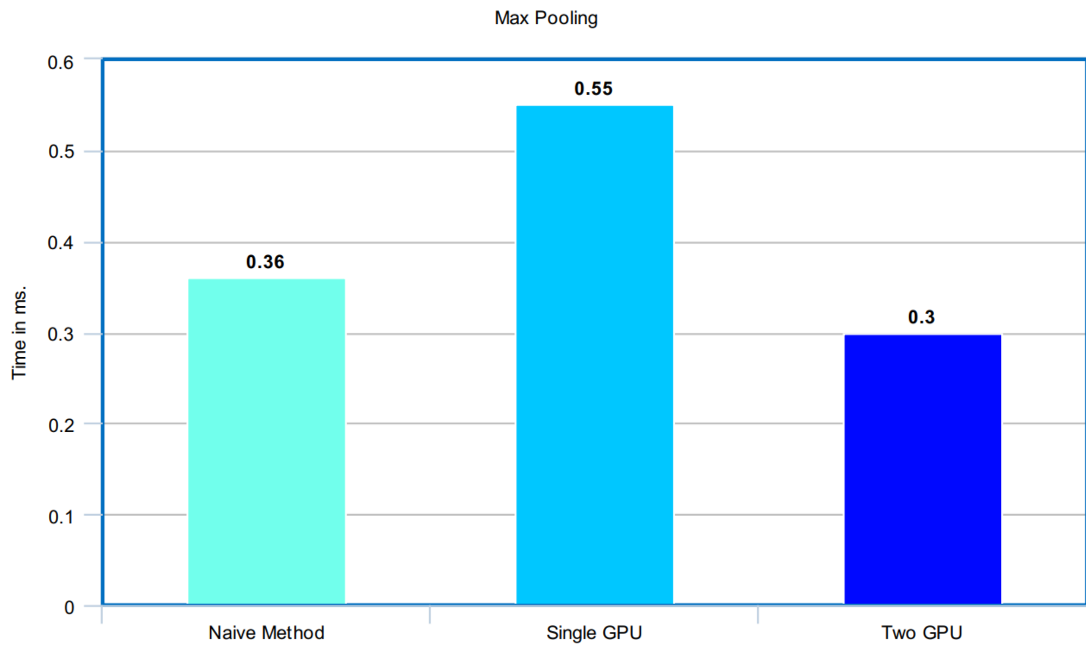


Figure 5.4 : Elapsed time for pooling layer.

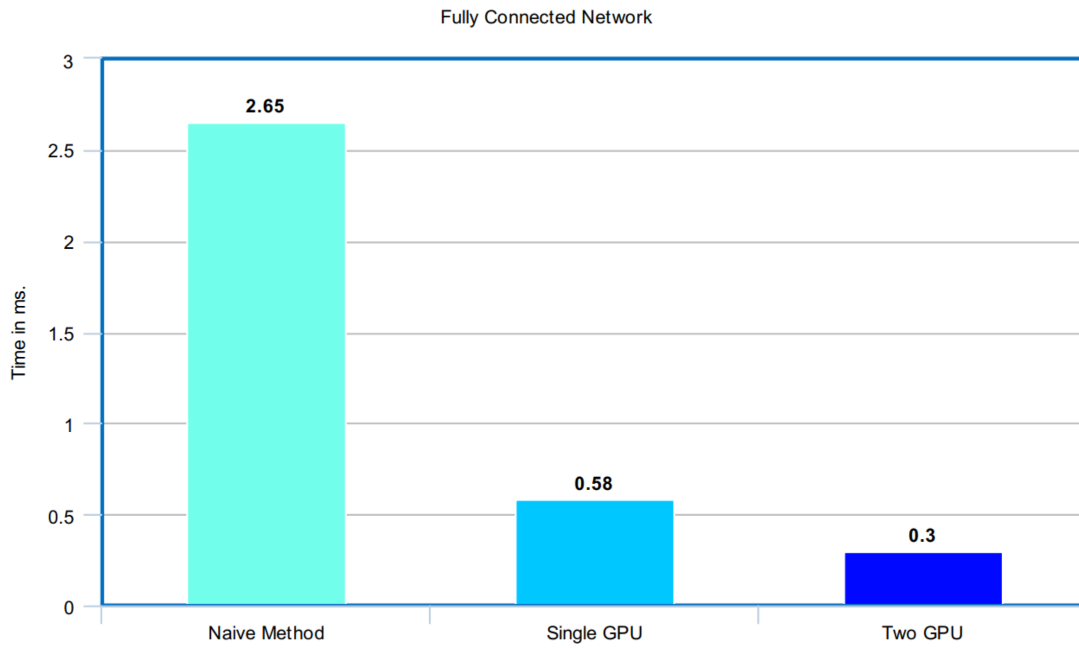


Figure 5.5 : Elapsed time for fully connected layer.

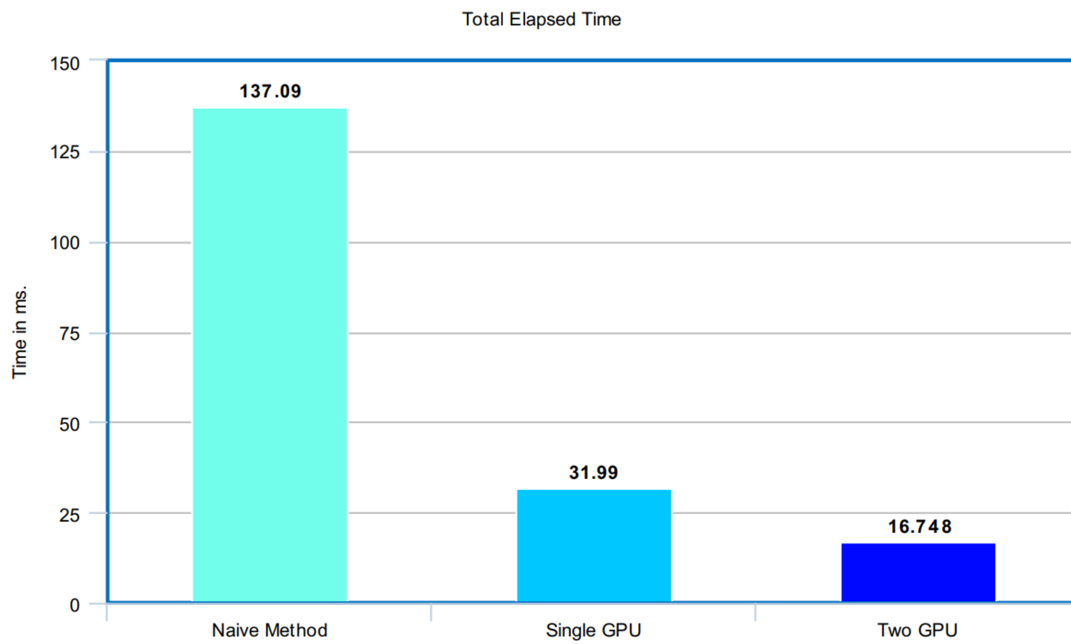


Figure 5.6 : Total duration of forward pass operation.

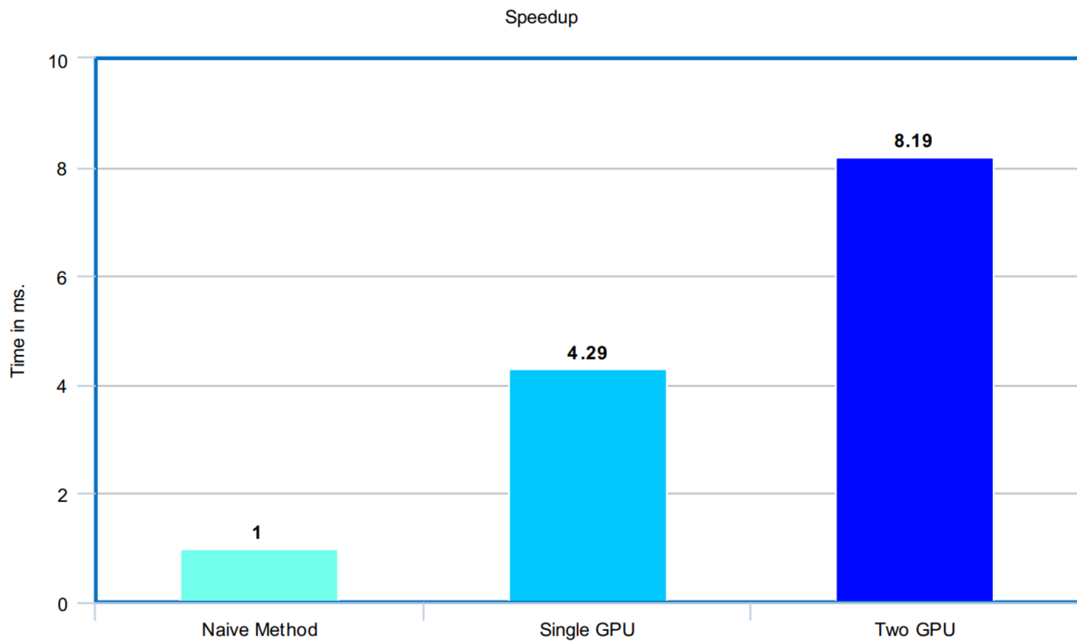


Figure 5.7 : Achieved speedups.

In this study there are a lot of rooms for improvement. For each GPU and GPU streams can be assign to different thread to reduce the kernel and asynchronous copy launch overhead. Result of that approach approximately 10% speedup expected.

For the future works this approach will be implemented on tiling problem. Within this approach not only the data but also parameters of the CNN will be distributed over multiple GPUs. This leads dividing the deeper neural networks to smaller parts then next small parts of the networks can be processed discretely.

Our first goal is generalizing the technique that studied in this thesis. After that generalization, this technique will be combined with evolutionary learning. With that knowledge next study will be related to real time ECG arrhythmia detection.

REFERENCES

Huck, S. (2011). Measuring processor power tdp vs. acp. *White Paper, Revision*, *1*.

Tendler, J. M., Dodson, J. S., Fields, J. S., Le, H., & Sinharoy, B. (2002). POWER4 system microarchitecture. *IBM Journal of Research and Development*, *46*(1), 5-25.

Stokes, J. (2005). Introduction to multithreading, superthreading and hyperthreading, Oct. 2002.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), 386.

Gaster, B., Howes, L., Kaeli, D. R., Mistry, P., & Schaa, D. (2012). *Heterogeneous computing with openCL: revised openCL 1*. Newnes.

Atanassov, E., Barth, M., Byckling, M., Codreanu, V., Ilieva, N., Karasek, T., ... & Stachon, M. (2017). Best Practive Guide Intel Xeon Phi v2. 0.

Maxfield, C. (2004). *The design warrior's guide to FPGAs: devices, tools and flows*. Elsevier.

Schmidt, B., Gonzalez-Dominguez, J., Hundt, C., & Schlarb, M. (2017). *Parallel programming: concepts and practice*. Morgan Kaufmann.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE transactions on computers*, *100*(9), 948-960.

Guillon, A. J. (2015). An Introduction to OpenCL C++. *Khronos Group*.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, *1*(4), 541-551.

Nvidia, C. U. D. A. (2019). Programming guide.

Cheng, J., Grossman, M., & McKercher, T. (2014). *Professional Cuda C Programming*. John Wiley & Sons.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).

Haykin, S. S. (2009). *Neural networks and learning machines/Simon Haykin*. New York: Prentice Hall,.

Mohamed, A. R., Dahl, G., & Hinton, G. (2009, December). Deep belief networks for phone recognition. In Nips workshop on deep learning for speech recognition and related applications (Vol. 1, No. 9, p. 39).

Li, F., Johnson, J. & Yeung, S (2017). Standford University – Convolutional Neural Network lecture materials.