CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Accelerating a Machine Learning Algorithm on a Graphics Processing Unit

Master's thesis in Embedded Electronic System Design

PRASANNA KOTRAPPA
PRADEEP LOGANATHAN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master thesis 2021

Accelerating a Machine Learning Algorithm on a Graphics Processing Unit

PRASANNA KOTRAPPA
PRADEEP LOGANATHAN

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Accelerating a Machine Learning Algorithm on a Graphics Processing Unit

Supervisor: Per Stenström
Examiner: Per Larsson-Edefors

Master Thesis 2021
Department of Computer Science and Engineering
Division of Computer Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden 2021.

Accelerating a Machine Learning Algorithm on a Graphics Processing Unit
PRASANNA KOTRAPPA

PRADEEP LOGANATHAN
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Life long learning from zero(LL0) is a lifelong learning algorithm that has a dynamic neural network architecture. Many machine learning tools perform poorly on dynamic structures due to the overhead of growing computational maps with expanding networks. This thesis explores the possibility of delivering higher performance for the LL0 algorithm compared to the existing PyTorch implementation by developing a custom solution. This developed solution has a strongly coupled mapping of the LL0 algorithm with the GPU to achieve hardware acceleration. A set of benchmarks are defined to compare the performance of the between implementations.

Furthermore, the thesis develops a methodology to investigate potential bottlenecks and parallelism with the implementation mapped to a GPU. The thesis achieves a significant speedup of $\times 19.48$ on the number of feedforward per unit of time, compared with the similar PyTorch implementation, on an MNIST dataset.

Keywords: GPU, Hardware Acceleration, Machine Learning, Life Long Learning Algorithms, CUDA, Dynamic Architecture.

Glossary

GPU - Graphics Processing Unit
DNN - Deep Neural Network
LL0 - Lifelong Learning from Zero
NAS - Neural architecture search
NVP - NVIDIA Visual Profiler
CPU - Central processing unit
API - Application Programming Interface
GUI - Graphical User Interface
RAM - Random Access Memory
ROM - Read Only Memory
OOP - Object oriented programming
AutoML - Automated Machine Learning
CUDA - Compute Unified Device Architecture
ML - Machine Learning
GDDR - Graphics Double Data Rate
SP - Streaming Processors
SM - Streaming Mutliprocessor
SDRAM - Synchronous dynamic random-access memory
DRAM - Dynamic Random-Access Memory
GFLOPS - Giga floating-point operations per second

Acknowledgements

We would like to thank our supervisor Per Stenström for his guidance and support during our thesis and Industrial supervisor Niklas Engsner for his feedback and insights. A special thanks to Claes Strannegård for his feedback on the algorithms. We would like to thank our examiner Per Larsson-Edefors for his feedbacks.

Further we would like to thank our families for the support and continuous encouragement throughout our studies.

Prasanna Kotrappa and Pradeep Loganathan, Gothenburg, January 2021

Contents

List of Figures

List of Tables

1
Introduction

Historically programmers have written sequential programs, based on the model described in 1945 by Von Neumann [5]. Further they have relied on the advancement of microprocessors to make an application to run faster, develop new features, and improve the capabilities of the software. This development in hardware capabilities came to a staggering halt in 2003 due to limitations in aspects of physics such as heat-dissipation and energy consumption. These limitations curbed the increase of the clock frequency and stalled the growth in activities that can be performed in each clock cycle within a single CPU [6].

To solve this issue, the semiconductor industry has followed two main trajectories for designing microprocessors [7]. First one being, the multi-core (2 - 8 or so cores) trajectory which seeks to double the core count with each core sustaining the highest execution speed of sequential programs. The other path being the many-core (hundreds of core) trajectory, which concentrates on improving the throughput of the parallel applications. Both these paths have benefited immensely from the doubling effect of semiconductor process generation [8]. The characteristics of the application determine the performance it takes from the microprocessor. The sequential computation has benefits using multi-core architecture, whereas the applications with potential parallelism are advantageous to run on many-core architecture [9].

The development of many-core devices has opened a new avenue for programmers to write multi-threaded parallel code for applications which have inherent parallelism, and now can be exploited due to the existence of such devices. GPU is one such many-core device that has over the years developed in response to intense demand from the video game industry. The computational intensive workload of computer-generated graphics has resulted in the development of the modern GPU architecture, which is robust to handle applications that require high throughput, are computationally heavy or needs lower latency. Machine learning-based application is one such area where GPU is being used, as a hardware accelerator, and is showing astounding potential [10][11].

Machine learning is the field of study that utilizes algorithms to find patterns in data, which then allows a computer to learn automatically without human help or interventions. It is making significant advances to complex problems like image and speech recognition which were difficult to solve using traditional methods [12][13]. One of such algorithms is Life Long learning from Zero (LL0), a machine learning algorithm with a dynamic approach [14].

The LL0 is a patent-pending algorithm developed by Dynamic Topologies AB, Sweden, a machine learning-based startup based in Gothenburg. The algorithm has shown that it can outperform traditional machine learning algorithms when tested across numerous data sets of distinct types, on versatility and has shown fast learning with low energy consumption [15].

This algorithm uses the concept of neuroplasticity, which is the biological ability of a brain to change the neural structure to adapt to the environment [16]. LL0 employs a dynamic neural network structure which mimics animal behavioral learning such as forgetting, generalization, and expanding to achieve lifelong learning, unlike algorithms with static structure [15].

1.1 Problem Description

The implementation of LL0 is a prototype built to test the theory of the algorithm, and it uses PyTorch libraries for optimization on a GPU platform [17]. PyTorch is the most commonly used machine learning toolkit used in the community [18]. However, the PyTorch implementation of the LL0 algorithm scales poorly on large data sets due to two major factors. Firstly, the PyTorch implementation is incompatible because the graph-based structure was used to implement LL0 whereas PyTorch is an array-based programming model. Secondly, there is an increase in computational overhead for defining a complex dynamic structure in PyTorch. Besides, there are additional challenges to the implementation concerning the dynamic nature of LL0.

The paper [19] argues that the most common machine learning toolkit performs poorly on algorithms that have dynamic structures. This is due to the challenges and the complexity involved in developing robust systems that match the requirement of an evolving structure and also deliver substantial performance. This thesis explores a solution that will exploit the unique properties of LL0 and the inherent parallelism to overcome these limitations, as mentioned in [19], and to outperform the implementation in PyTorch.

1.2 Scope

The thesis aims to study the LL0 algorithm to identify potential points of hardware accelerations. This includes developing custom mapping of the algorithm with the GPU, setting up a methodology to eliminate bottlenecks and improve parallelism. Additionally, we will verify that the implementation works as the algorithm intends and define benchmarks to quantify the performance for comparison.

The thesis goal is accomplished by writing a customized software architecture, to capture the dynamic nature of the LL0, and writing dedicated Compute Unified Device Architecture (CUDA) kernel and data paths for GPU optimization. The functionality of the architecture is verified by testing on MNIST dataset [20], and IRIS dataset [21].

Hardware acceleration of algorithms can be limited by bottlenecks caused by data transfer, computation overhead, or underutilized parallelism. A methodology is framed to improve the performance of the implementation by profiling it for bottlenecks. The methodology standardizes the finding and fixing of bottlenecks into an iterative process that improves the performance of the implementation.

Tools like visual studio profiler and Nvidia profiler are used for profiling and analyzing potential sources of parallelism. Further to compare the performance between the two implementations, benchmarks are created which capture the essence of the LL0 algorithm and are used for comparison.

The rest of the thesis organization is as follows. Section 2 provides details on the background of the machine learning toolkit, GPU architecture, CUDA programming paradigm, and principals for performance gain. Section 3 introduces an overview of the LL0 algorithm. Section 4 explains the proposed software architecture for the implementation of LL0, and Section 5 provides the details of the LL0 implementation. Section 6 explains the experimental methodology. Section 7 presents the results from the thesis and the performance comparison with the python library implementation on a GPU platform, followed by discussions in Section 8 and conclusions of the thesis in Section 9.

2
Background

This section briefly describes about machine learning toolkits, graphics processing unit(GPU) architecture and Compute Unified Device Architecture(CUDA) platform and its framework, principles of energy efficient design, datasets used to test the performance gain and accuracy and a brief explanation of problem statement.

2.1 Machine Learning and Toolkit

2.1.1 Machine Learning

Machine learning has contributed successfully in the advancement in fields of image recognition, speech recognition [22][23], gaming [24], language recognition and analysis [25][26]. This is accomplished by relying on feature engineering [27]. Features are input data transformed into a numerical representation, which is easier to handle by the algorithms such as classifiers to produce better results. Feature engineering is about sections of the right features to be applied to data which results in algorithms giving the best prediction to problems. For example, in the case of identification of handwritten digits. The digits were predicted using features such as edges, angles of slopes, or enclosed holes. These features were estimated by filters.

Until around 2000, feature engineering was a manual process in which engineers handcrafted features for different data sets and repeated the process to find the best features. This was completely transformed with the concept of deep learning. Deep learning automated the process of feature selection which resulted in groundbreaking results and led to the current remarkable growth in the field of machine learning.

Deep learning identifies the best features automatically by iterating over the raw data. For example, in the above-mentioned identification of handwritten digits problem, the processes of deep learning start with a base filter for predictions. It iterates over the raw data multiple times, while incrementally changing the parameters of the filter, to achieve a better prediction after the next iteration. These incremental changes are governed by mathematical algorithms, such as gradient descent, which results in better predictions. At the same time, it avoids taking the big step sizes which help in achieving an optimal solution. This advancement helped to automate the painstaking process of creating handcrafted features.

2.1.2 Toolkit

The contribution to the advancement in machine learning field are primarily due to development of sophisticated learning models [12][28], large scale open source data sets [29][30] and powerful software toolkits which help researchers to put conceptual models into practice by utilizing computational resources for training models [31][32].

Deep learning practitioner investigates on experimenting with new models for novel research areas or examining new designs. However, before researching there are numerous additional challenges such as accelerating programs from a single device to running on data center clusters, to handle dataset preprocessing, statistical analysis, plotting, and more. These are nontrivial engineering challenges and impractical for a single practitioner to have skillsets to solve. Machine learning toolkits solve this problem by handling these engineering challenges in the backend and provide a platform for the deep learning practitioner to research on machine learning.

The most common programming paradigm used by various toolkits [33][34][35][36] is static declaration. Under this paradigm the solution is divided into two parts:

- **Definition of a computational architecture:** The front end part of toolkit where the deep learning practitioner describes the model, a structure or a neural network best suited for the algorithm. Toolkits provide a level of abstraction to the users which makes it simpler to express their models.

- **Execution of the computation:** The toolkit manages the hardware resources and makes optimisation necessary to get the acceleration from the underlying hardware.

Machine learning deals with computationally heavy workload, and the tools have to balance between providing users with a simple environment for development and manage the hardware resources effectively. This is achieved by with the help of *computational graphs*. A computational graph is a symbolic representation of complex numerical statements which consists of primarily computation operators. A compiler then maps it to the hardware which helps in maximum utilisation of underlying hardware [33].

The success of toolkit is in converting the user-provided abstractions into a computational graph. The user now can define a model and run it across different hardware architectures and also on a large number of devices in data centres with absolutely no knowledge of how the hardware works.

2.2 GPU Platform

GPUs are many-core computational devices that provide a platform for parallel computation. It is in contrast to CPU. Modern CPUs are multi-core and are efficient in terms of sequential processing. It supports tasks like operating systems, I/O devices, and software applications, which makes it more suitable to support computation for diverse applications. However, it is disadvantageous in the case of applications requires to perform complex repetitive operations. For example graphics processing. It is better to have multiple cores of small compute engines to perform the same task, instead of having one powerful compute engine to perform the heavy operation. This results in a speedup over sequential execution [1], and many such applications can be accelerated efficiently using GPU.

A GPU that is capable of handling many computations that were traditionally handled by CPU is known as general-purpose GPU (GPGPU). The GPU performance has provided a significant edge over the CPU concerning Giga floating-point operations per second (Gigaflops), computation speed. This provides an incentive for programmers to utilize the massively parallel computation offered by GPU and make the applications GPU-oriented.

Traditionally, GPUs were designed for rendering graphics pipelines influenced by the needs of computer graphics and accelerating 3-D and 2-D graphics. Thus, the processing capacity of the GPU was limited and inflexible for GPGPU. However, Machine Learning (ML) has recently emerged as a crucial application that is one of the guiding factors for the future of the design architecture of GPU. ML algorithms require high computational power and GPU has multiprocessing units which makes it a promising computing platform.

The GPU as a platform has assisted researchers in the fields of statistical physics, bioinformatics, computational finance, and many more. The GPGPU has become a prominent scientific, computing platform choice after NVIDIA Corporation introduced CUDA platform [37]. CUDA is NVIDIA's parallel programming paradigm and offers a software environment that enabled programmers to use high-level programming languages, for example, C/C++ to write instructions on massively parallel GPU traditionally shader-based framework was used to program the GPU [38]. This motivates further development of next-generation GPU algorithms [39].

2.2.1 GPU Architecture

The architecture of a CUDA-capable GPU is organized into multiple streaming multiprocessors (SM) that consist of multiple processors running the same instruction or threads as shown in Figure 2.1. A thread is an action or job that is performed on a processor. The amount of SMs varies from generation in GPU architecture. Each SM has an array of Streaming Processors(SPs) that share an instruction cache and control logic as shown in Figure 2.1. In an SM 32 threads that have the same instruction are executed together and this is called a warp. Each SM manages multiple warps using a warp scheduler. Modern GPUs come with gigabytes of Graphics Double Data Rate(GDDR) and Synchronous DRAM(SDRAM), referred to as global memory in Figure 2.1. The GPU has a separate RAM from the CPU, used for frame buffer memory which is needed for rendering graphics. The CUDA application transfers data between the CPU's system memory and GPU's global memory. During runtime, the application holds the data in the global memory and establishes a connection to system memory when data must be transferred between the GPU and CPU.

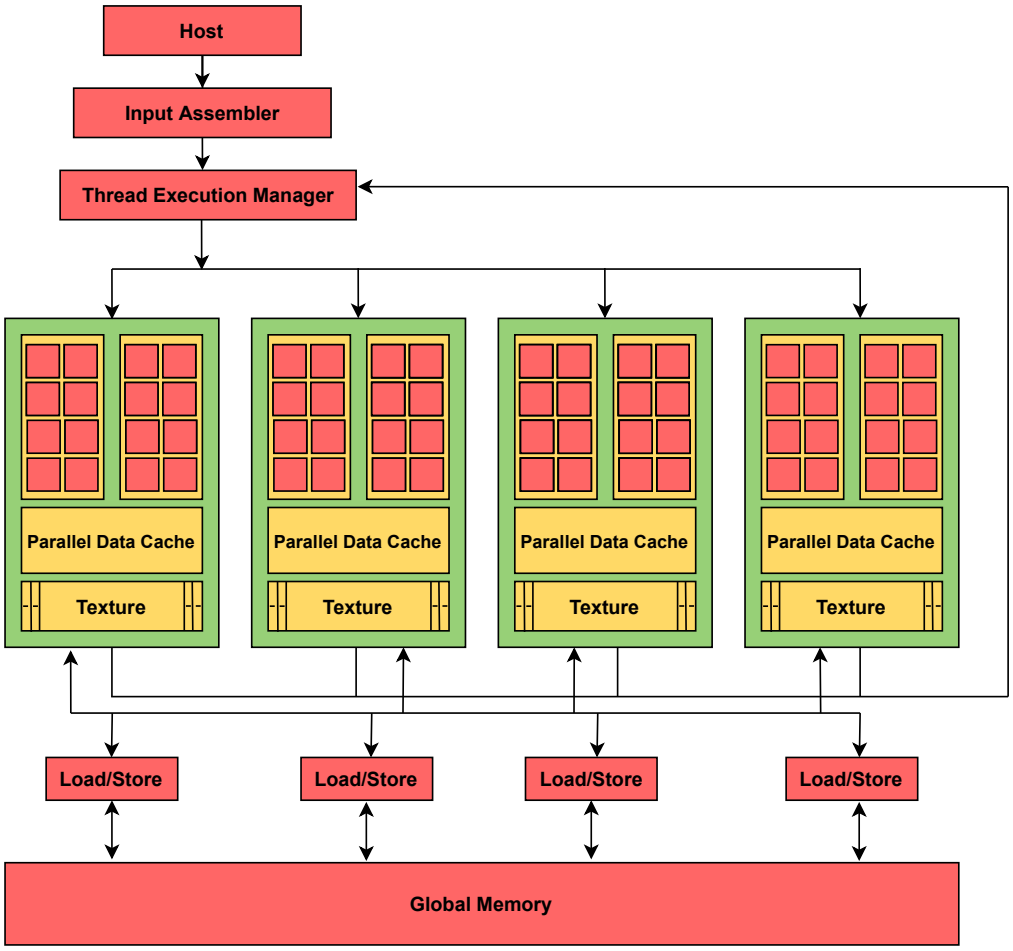

Figure 2.1: CUDA Capable GPU Architecture with gigabytes of Graphics Double Data Rate(GDDR) and Synchronous DRAM(SDRAM), known as Global memory [1]

2.2.2 Structure of a Streaming Multiprocessor

A GPU consists of an array of SM with several processors each, as shown in Figure 2.2, i.e N multiprocessor with M cores each. As illustrated in Figure 2.2 the processors share the same instruction unit, texture cache, and constant cache.

In NVIDIA GPU architecture, each SM has multiple CUDA cores, and the number of cores depends on the generation of the architecture. A CUDA core is a specialized processor for integers and single point floating operations, which are used to run the threads for GPGPU computations [40].

Every SM is a collection of 32 threads having the same instructions executed at the same time. Multiple warps are managed and run concurrently by an SM using a warp scheduler. Any thread inside a warp having a different instruction leads to warp divergence, which leads to poor performance since the thread batch loses parallelisms and becomes sequential.

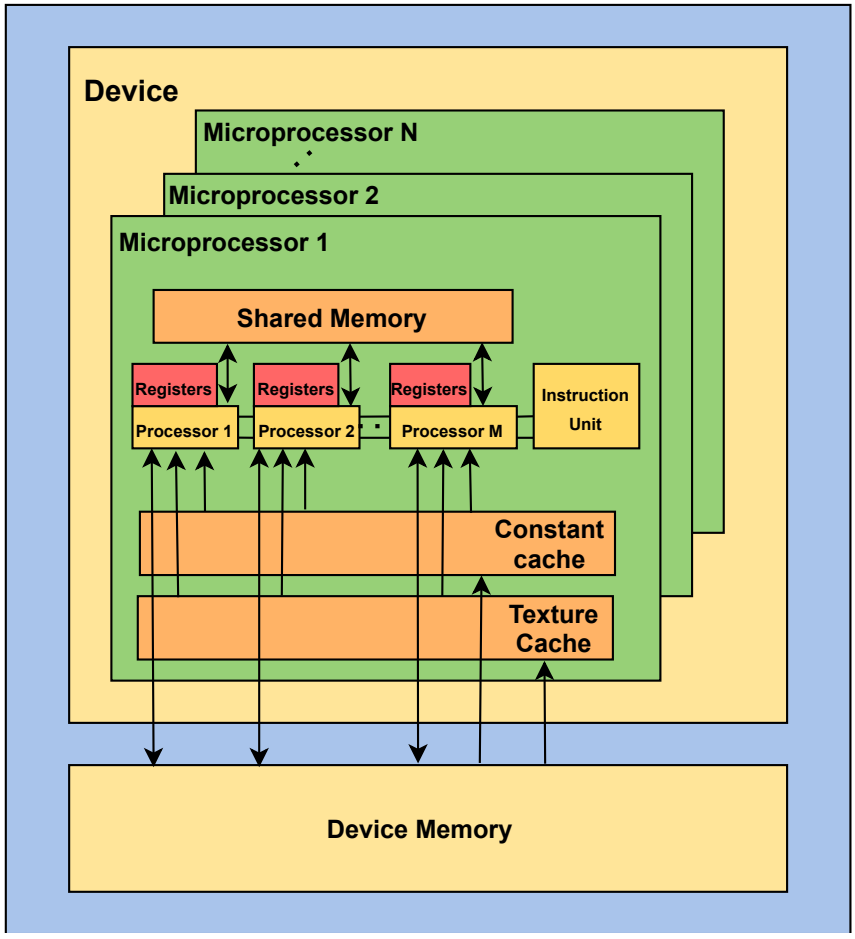

Figure 2.2: Multiprocessor Structure, Illustration of N multiprocessor with M cores each and details of every block of a core with the connection between the processor and the device memory [2].

2.2.3 Memory Hierarchy

Memory access is one of the biggest bottlenecks of computing [41], which is due to uneven growth in the performance of CPU and the lower performance of memory. This makes it expensive to complete memory access compared to CPU computation. Modern CPU manages this with large caches. A cache is a small chunk of memory that sits between the CPU and the main memory and offers faster access than main memory.

A GPU approaches the slower memory access problem using a different strategy: It uses massive parallelism to fill with computation while waiting for memory access. For example, an application like video processing, image processing, or simulations is well suited for GPU because the computational load is intensive enough to reduce the effect of memory access time. The memory hierarchy of GPU is designed to increase the throughput besides matching this strategy. Memory hierarchy has many layers as illustrated in Figure 2.5

- **Registers:** The fastest piece of the memory block which is located in each SM. Thus is used to store a local variable of the kernel and is limited in size.
- **Local memory:** It sits inside the global memory and acts as an additional memory space for SM. It is 150× slower than registers.
- **Shared memory:** A memory bank that lies inside an SM and has the same hardware structure as the L1 cache, is across blocks and is explained in detail in Section 2.3.4.
- **Global memory:** The vast majority of the memory and also the slowest. It is equivalent to RAM in CPU but differs in hardware structure.
- **L1\L2\L3 cache:** It behaves similarly to the cache of CPU and is used by compilers to cache memory blocks for GPU.
- **Constant memory:** It is a special memory block that is located in the global memory. It is used to store constants, which cannot be compiled into the program and values to be passed to kernels.

2.3 CUDA Programming Paradigm

Since 2003, attempts were made to use GPU's computational power for non-graphics applications, such as protein folding, quantitative trading, and software acceleration. However, to use GPUs as GPGPUs, the application developers had to use graphic-specific programs like Directx or OpenGL. To use such software application developers needed to have intricate details on how graphics work and how to use these graphical APIs.

To assist programmers in general-purpose computing on a GPU [42], NVIDIA developed CUDA, which is a parallel computing programming model. CUDA provides a platform for application developers to directly control the GPU hardware and use GPU's computational power to accelerate applications.

CUDA offers a scalable programming model with a low learning curve for developers with standard programming languages. The programming model consists of breaking the problem into small sections of independently-run parallel sub-programs. These subprograms are organized into blocks of thread, which is an independent line of code that can be run in parallel with similar independent lines of code. Application developers can control the execution of the threads and also the formations of the blocks. CUDA manages the mapping of these blocks on the GPU and manages memory. The CUDA programming paradigm has unlocked the potential of applications that can now use GPU as hardware accelerators.

2.3.1 CUDA Programming Framework

The CUDA program is a unified source code divided into two parts, host code, and device code. Generally, the code runs on the CPU and executes serially is known as host code. The code that exhibits a high degree of parallelism and is executed on the GPU is known as device code. Since CUDA is an extension of C, the host code of the program is written using C, and the device code is declared using CUDA C syntax. The applications can utilize the GPU features by calling the kernel written using the CUDA function. During compilation, the NVCC compiler [43] separates these two codes, the host code is compiled with the standard C compiler, whereas the device code is compiled with the NVCC compiler mapped to the GPU.

The CUDA program can be triggered by executing the kernel functions. During compilation, the kernel will allocate multiple threads to the GPU. There is a distinct syntax to call CUDA programs like *Global*, *Device* and *Host*. CPU uses the *Global* keyword to invoke a kernel on GPU and the *Host* keyword can be used to run the kernel on CPU. The GPU can invoke kernel using the *Device* keyword.

2.3.2 CUDA Thread Hierarchy

High performance of the processing system can be achieved by breaking the problem into small sections and process these individual sections parallelly. The GPU processes instructions parallelly by allocating multiple threads to each task. The threads generated by the kernel are called as *grid*, and grids are created for each invoked CUDA function. CUDA organizes these threads into logical *blocks*, where each thread is synchronized and has its ID to determine which data to work on, within the block that executes the single function. Thousands of threads together execute one function, referred to as kernel. However, data managed by each thread is different.

A group of one or more blocks is known as grids and blocks are made of threads as shown in Figure 2.3. These threads operate on individual cores of the GPU multiprocessor. The CUDA paradigm enables each core to execute the same instruction on different data and each thread execution is managed using its index $threadIdx$. There is a limit to the number of threads per block and a maximum of 1024 threads can be present in one block considering all the threads share the same limited memory resource of the block. Blocks in the grids can be managed using $blockIdx$.

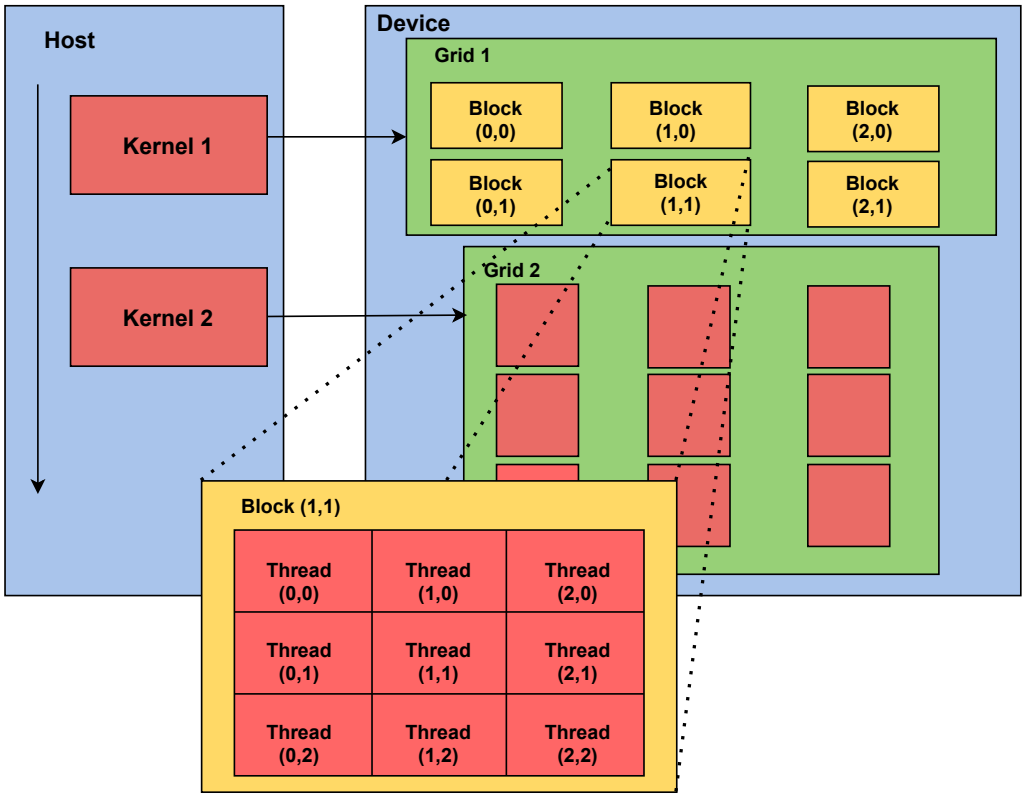

Figure 2.3: CUDA Thread Organization depicting how CUDA capable architecture, splits the device into grids, blocks, and threads in a hierarchical structure. Illustration of how Host(CPU) launches kernels along with its associated grids on the device(GPU) with a batch of thread blocks [3].

Problems with a high degree of parallelism can be solved using multidimensional arrays and CUDA offers built-in multidimensional thread indexing called *dim*3. If the application is multidimensional, the threads and blocks can be launched into three dimensions. The thread and block IDs can be 1D, 2D, or 3D and their index can be managed using *gridDim* and *blockDim* respectively. All the threads in the blocks execute the same function by efficiently sharing the data through a low latency shared memory, thus making it ideal for parallel processing. Additionally, there is no dependency between the two blocks of a grid i.e. two threads from two different blocks work independently without any synchronization.

2.3.3 CUDA Thread Execution

The CUDA thread execution is illustrated in Figure 2.4. The program starts with the execution of the host code (CPU code). When the kernel function is invoked, the device code will execute by launching a large number of threads to exploit data parallelism. As stated earlier several blocks form a grid and multiple threads in block execute an instruction parallelly by utilizing massive GPU processing power. The processor in the GPU multiprocessor facilitates the threads to collectively execute single instruction on different data. There are two kinds of thread execution in CUDA, *Synchronous* and *Asynchronous*. In synchronous execution, when the host invokes the device kernel, the host will suspend its execution until the GPU completes its tasks. However, in asynchronous execution, the host will continue to operate while the GPU is executing its set of instructions. When all the threads of the kernel complete its execution, the corresponding grid is terminated until the host invokes another kernel, resulting in the generation of another set of threads, *grid*, as shown in Figure 2.4.

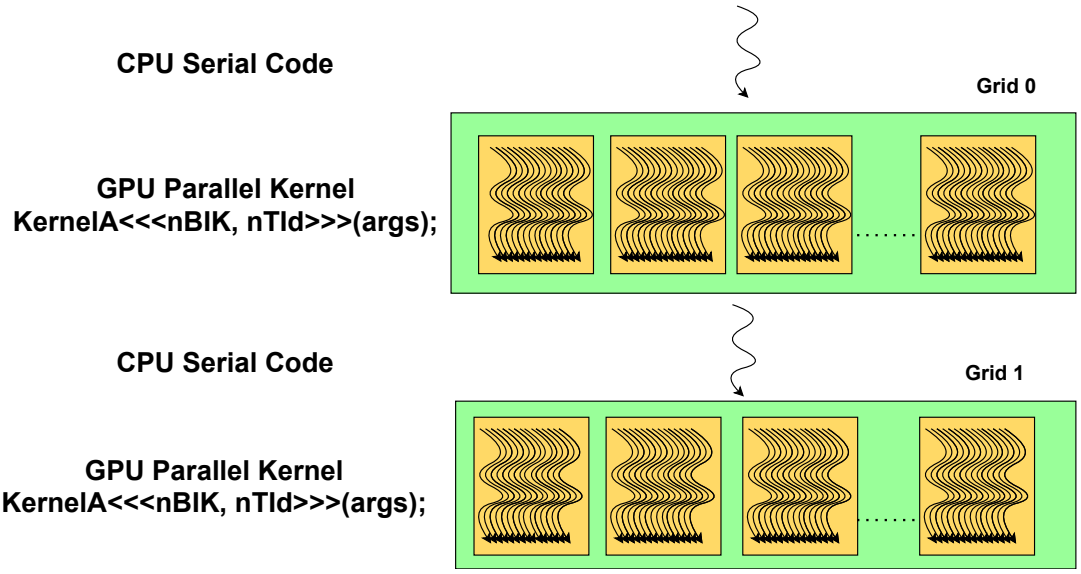

Figure 2.4: CUDA Thread Execution illustrating CPU serial code runs first followed by GPU parallel code where $nBIK$ and $nTid$ refers to several blocks and threads per kernel respectively. Once GPU completed its execution, the CPU starts its execution [3].

2.3.4 CUDA Memory Hierarchy

The memory hierarchy is an important component of the CUDA platform because memory access plays an important role in the computation time of the system. The memory structure of the CUDA is depicted in Figure 2.5. The physical components of memory are discussed in Section 2.2.3. The interaction of CUDA, the memory hierarchy and the control it provides to developers is the reason for the success of GPU as a hardware accelerator for general purpose applications.

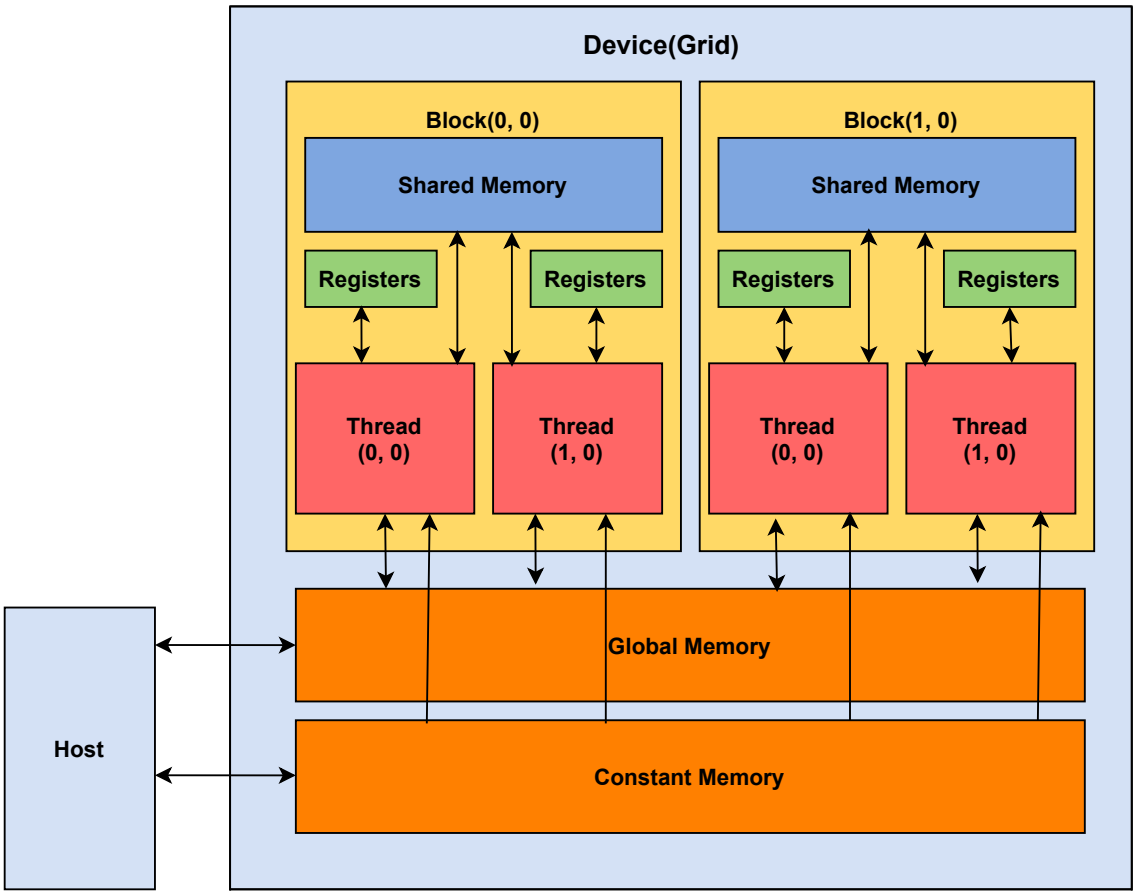

Figure 2.5: CUDA Memory Hierarchy depicting different types of memory in the CUDA device, Host transfer data to/from Global and constant memories on the device [3].

The CUDA programming paradigm allows a program, with a sufficient amount of parallelism, to be broken down into several sub-components of computationally similar units which are then is mapped on a GPU for computation. A kernel is invoked to execute this sub-component. The CUDA executes the threads in the form of grids of blocks and maps individual blocks on an SM. Each block that is assigned to an individual SM is broken down into warps, which consists of 32 threads and executed parallelly on the SM.

To achieve high throughput each SM has a *register*, *shared memory* and L1 *cache*. The *register* is an extremely fast memory of a limited size where each thread can access for storing and retrieving data. The variables declared inside the kernels are stored here. *Shared memory* is a large block of memory having the same hardware architecture as that of L1*cache*. *Shared memory* is completely in control of users and the memory is shared by all the threads of a block. This control provides users to write optimized memory access using complete hardware-specific memory patterns. For example, the output could be built on shared memory before offloading it to global memory which saves a large number of clock cycles. L1*cache* is used by the compiler to bring the instruction and data from the large global memory.

A GPU is designed to have a high throughput and a large amount of its memory is off-chip called *global memory* which is separate from the GPU core. This memory is used to store data besides it is slow. L3 and L2 caches are used for transporting the data from global memory to SM and behave similarly to the caching mechanism of the CPU.

The flexibility provided by CUDA to control the memory hierarchy gives the developer an option to optimize the code and deliver better results than using a software library.

2.3.5 CUDA Runtime API

The intermediate layer between the application and CUDA driver referred to as CUDA runtime API is illustrated in Figure 2.6. The CUDA consists of a high-level runtime API and a low-level driver API. All the instructions are translated and processed by low-level API drivers. CUDA libraries can also invoke the CUDA runtime API. CUDA frameworks provide a controllable hierarchy to access the CUDA drivers.

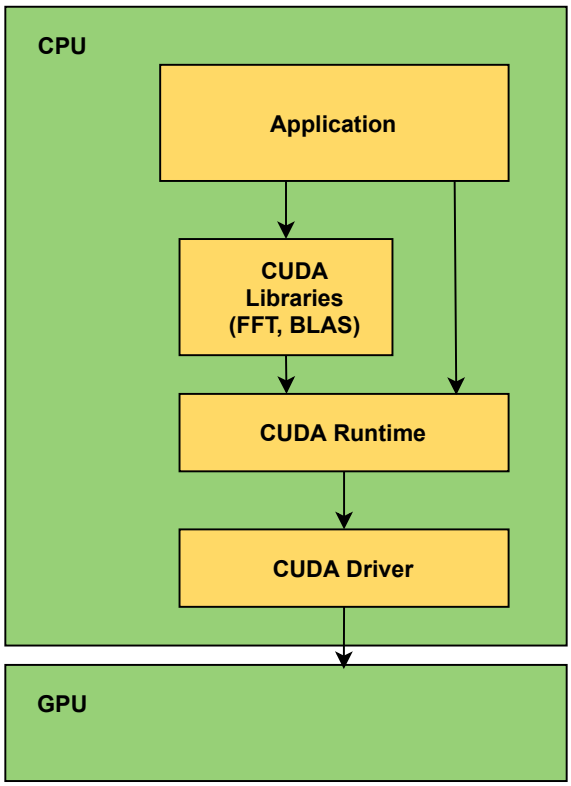

Figure 2.6: Illustration of CUDA API located between Application, Library, and driver to translate the high-level instruction into the low level to manage CUDA driver API [4].

2.4 Performance Evaluation in GPU

In ML applications, GPU is used as a coprocessor, which handles computationally heavy workloads whose neural structure is stored in the GPU RAM. CPU manages the control flow of logic, initiates function calls for the GPU and transfers necessary data required.

It is essential to evaluate whether the parts of the application are worth computing in GPU. Inefficient mapping of algorithms or applications will lead to performance throttling. Frequent data transfer with the CPU is also an expensive task that consumes multiple clock cycles causing slowdowns. Many useful general-purpose guides have been created to give an idea of how to better harness GPU power [44][45]. Evaluation helps to understand the extent of the concurrent processing power of the GPU being underutilized.

In the CPU world, a wide range of tools have been employed to investigate new concepts and designs such as binary instrumentation tools, simulators, instruction sampling tools, and profilers. These tools provide a variety of features and capabilities to the computer architects to develop new computer architecture [46]. On the other hand, GPU computing for scientific research has been steadily increasing

and it follows a different programming paradigm. The GPU manufacturers have developed similar tools leveraging hardware profiling and GPU debugging such as *NVIDIA's visual profiler* [47], *NVIDIA nsight systems* [48], and *nvprof*. nvprof helps to profile the application from the command line, diversely Nvidia visual profiler and nsight systems are GUI-based tools. But these tools are largely limited by the fixed amount of features and do not offer the user the flexibility provided by the CPU.

Simulators are used to gauge the performance of the GPU application. They are flexible and provide fine-grained details of execution. Further, they provide the most control over architecture under investigation and are fundamental for many types of detailed studies. However, they are time-consuming to develop and are moderately slow in simulation rates, which force the researcher to provide small-scale and tailor-made input so that the experiments execute within the time limit [46]. Thus for input-dependent applications, users cannot collect data for real workload execution conditions and its data sets. Also, due to the lack of interruptions and simultaneous multithreading(SMT) in GPU, it is difficult to measure the periodic sampling [49].

Profiling provides information on the performance of the application on GPU, which helps to evaluate the mapping of the functions on the device. Profiling enables the optimization of the performance of the application by providing insight into how the application uses the architecture. This includes memory access, stall times, and measuring the performance at the thread level [49]. Further, profiling helps to identify the inherent root cause of performance degradation of an application.

2.5 Profiling Tools

The performance of the LL0 implementation can be enhanced by eliminating all the potential performance bottlenecks without affecting the behavior of the system. These bottlenecks can be identified with the help of tools, referred to as *nvprof* and NVIDIA Visual Profiler (NVP). It is essential to identify and exploit the opportunities to optimize the application for high performance and understand how well the application is executed on the GPU, which design feature contributes to the performance degradation. To obtain detailed performance metrics, the application is evaluated on nvprof and NVP tools.

2.5.1 nvprof

CUDA is equipped with a powerful tool known as nvprof. Nvprof is a light-weight profiler and it provides information on how the CUDA kernels running on NVIDIA GPU. It can profile the applications written using different programming languages such as C++, python, etc. But, these applications should launch the kernel using the CUDA runtime API.

Data transfer is expensive in terms of energy consumption, where communication between the GPU and CPU through the bus is one of the bottlenecks which leads to a reduction in the performance of the system concerning time. Time spent on

transferring the data is the time lost on computing. Unoptimized and avoidable data transfer is introduced in the system while implementing the algorithms in the higher-level languages such as C++ and Python. To reduce the amount of data transfer, it is necessary to trace the CUDA calls and identify the specific portion of the program which is slow and improve the data path.

2.5.2 NVIDIA Visual Profiler(NVP)

The NVP provides profiling information using GUI, and it helps to assess each child process computation time i.e. the amount of time consumed by each child process in GPU and CPU, overall resources used by these child processes at a specific time, and the data migration between GPU to CPU.

In Figure 2.7, the application is loaded with MNIST dataset of 60000 images of size [28 × 28] pixel, observed the total number of runtime API calls between 140s and 380s. In the *thread section*, *data migration* which is the movement of data from GPU to CPU can be seen under the section $unified\ memory$. The overall data transfer between GPU and CPU can be observed under the section $CUDA$ where MemCpy(HtoD) denotes data being copied from CPU to GPU and MemCpy(DtoH) denotes data being copied from GPU to CPU, here H denotes Host (CPU) and D denotes Device(GPU). Overall computing resources consumed by each function can be viewed under the section compute.

For Example, the Gaussian activation forward function is consuming 13.4% GPU resource, and sigmoid activation forward consuming about 0.6% of GPU resources. These data help in analyzing the computation overhead and the application is accelerated by efficiently using these resources.

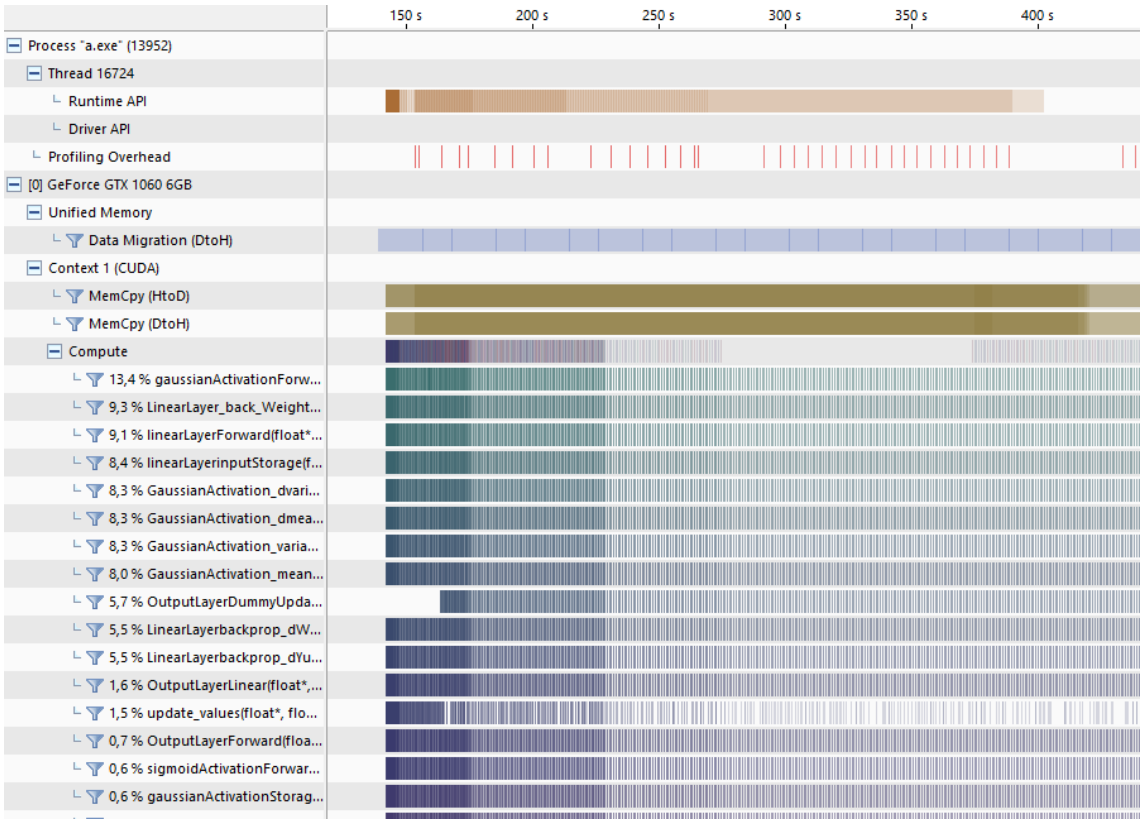

Figure 2.7: NVIDIA Visual Profiler depicting the number of computing resources consumed by each function, for example, the gaussian activation consumes 13.4%, the linear layer consumes 9.3%, etc. The data transfer between CPU and GPU is observed in section MemCpy(HtoD - DtoH). The runtime API calls when the application was loaded with an MNIST dataset of 60000 images of size [28 × 28] pixel.

2.6 Visual Studio Performance Profiler

Efficient memory management contributes to the overall performance of the system. The visual studio performance profiler is used to identify the memory consumption of each function. It is predominantly used for measuring stack and heap memories of CPU utilization with the help of an integrated development environment(IDE) Microsoft Visual Studio. The Visual studio facilitates users to insert breakpoints between the function calls, which results in the program execution halting. At that point, a snapshot can be taken that helps to determine the heap and stack memory consumption. Additionally, it provides the user to granularly analyze which process is responsible for high memory consumption or the memory leak.

In Figure 2.8, the process memory(MB) is the Random Access Memory (RAM) of the system, CPU(% of all processors) denotes the percentage of CPU being used. Memory usage denotes the amount of memory allocated and the change of heap memory.

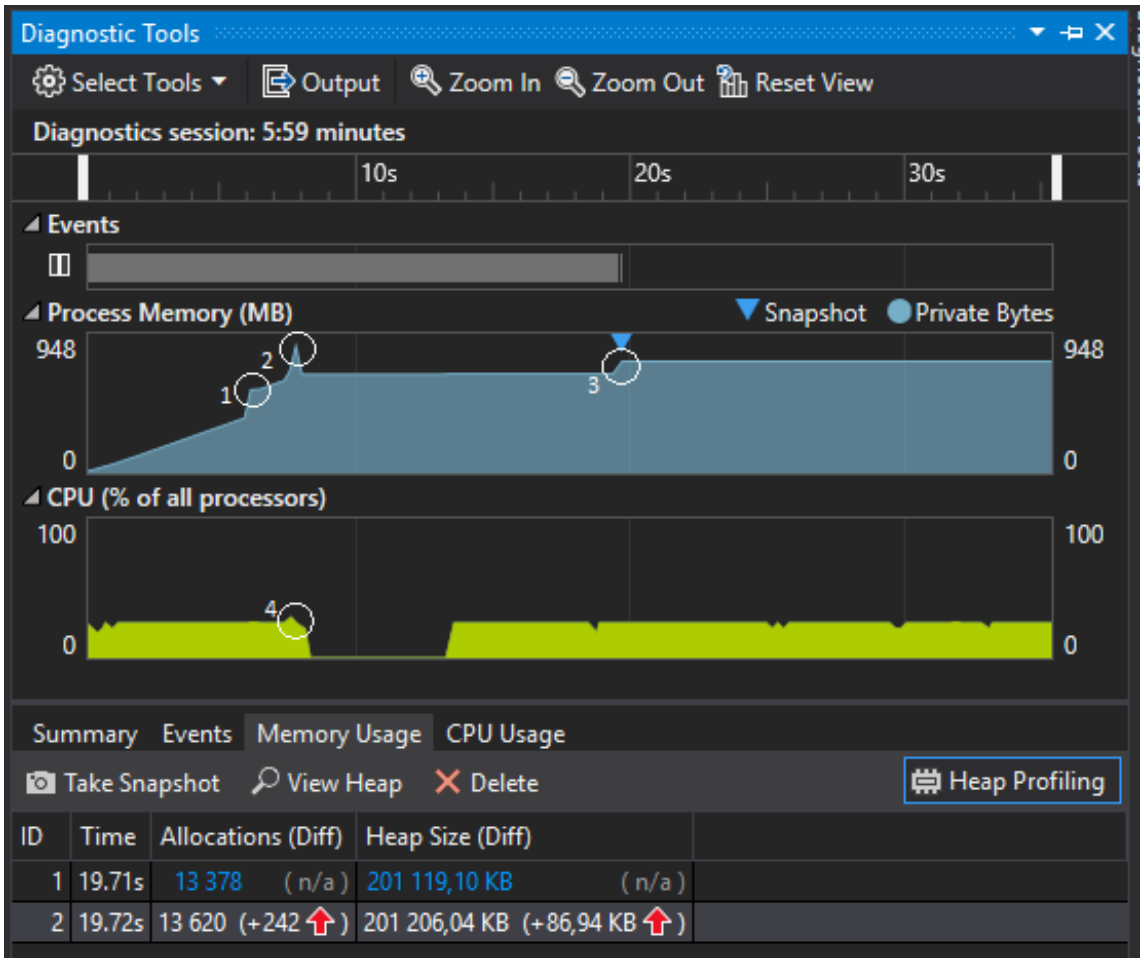

Figure 2.8: Visual Studio Performance Profiler depicting process memory behaviour on a memory scale of Megabyte(MB), Percentage of CPU usage and heap memory consumption when the application was loaded with MNIST dataset of 60000 images of size [28 × 28] pixel. To determine memory consumption, two snapshots have been taken at 19.71s and 19.72s. By looking closely on the memory usage, the heap memory usage has been increased by 86.94KB during the second snapshot at 19.72s.

In Figure 2.8, the tool visual studio profiler is used to observe the CPU memory and utilization. For this profiling section, a program is loaded with MNIST datasets of 60000 images of size [28 × 28] pixel each. There are four unique peaks observed which are explained below.

The initial linear rise in the memory is because of 60,000 training images and 10,000 test images being transferred from the Read Only Memory (ROM) to RAM followed by transferring of labels.

The first peak (marked as 1 in Figure 2.8) is due to the transfer of 60000 training images and their labels from RAM to GPU memory. The second peak(2) is due to the transfer of 10000 test images and their labels. The dip in process memory is because of freeing up of RAM memory. The third peak(3) denotes that a new layer in the neural network has been created. The reference point 4 indicates freeing up CPU memory since all the data has been transferred to GPU.

2.7 Principles for an Energy Efficient Design

The ML computation workload involves a large number of repetitive operations of matrix multiplication, additions, and floating-point operations. Repetitious operations of such kind are energy inefficient in CPU because the cores are not designed for these kinds of operations. Modern hardware accelerators resolve these strains with high efficiency. However, the intense energy consumption of ML workload is not due to the operation themselves rather because of the data movement. The excessive energy consumption is primarily due to constant data movement from off-chip to on-chip or from on-chip to cache[50][51]. ML algorithms need to process large amounts of data to work, where the energy consumption for the transfer of data from the data storage to hardware accelerators is an order of magnitude higher than the computation of data itself.

The key to having an energy-efficient design is to have less data movement. There are two major design philosophies to reduce the data movement in multi-level storage systems. One is the creation of data paths, and the other is exploiting system parallelism.

- **Data Path:** Creation of the data path leads to the management of the data flow. It assists in the scheduling for data reuse and data locality. This results in high energy efficiency design as the data remains in the closest level of storage hierarchy and prevents costly access to higher levels of storage.
- **Parallelism:** The exploitation of the inherent parallelism of the machine learning systems increases the throughput of data being sent to a hardware accelerator. This increases the energy efficiency of the system because a large amount of data can be processed. Increased throughput of data results in data spending lesser time in memory subsystems.

2.8 Dataset

Machine learning algorithms, in general, are tested against datasets to determine the accuracy, energy consumption, and performance of an algorithm. Benchmarking the ML algorithm with classic datasets gives an advantage of comparing with a large set of algorithms. There are different kinds of datasets available depending on the purpose of the ML applications such as image and speech recognition etc. Machine learning algorithms are normally of two types *Classification* and *Regression*.

Regression predicts the values and has independent and dependent variables. Predicting the value of an independent variable by varying dependent variables is known as regression, for example, predicting the forest fire by providing RH - relative humidity, temperature, and rain, etc. details as input data [52][21].

Classification refers to categorizing the new data based on previously trained data during training. For example, age of abalone can be predicted by providing physical measurements like length, height, diameter, and the number of rings, etc as input to the training [21].

In this thesis, multiple datasets were used during benchmarking and proof of verification. Some of them are explained below.

2.8.1 IRIS Dataset

The IRIS dataset contains a set of Iris flower data known as Fisher's data. Typically this is used for statistical classification purposes. It contains a total of 150 data points information of the Iris setosa, Iris virginica, and Iris versicolour with 50 samples each. Classification in the iris dataset is performed based on four properties or features of flowers such as length and width of the sepals and Petals(measured in centimeters) and species [21].

Iris is a fairly simple dataset to test on a machine learning algorithm. In this thesis, the iris dataset has been used to test the classifying capability of machine learning during the initial stage of the algorithm development. This dataset is useful to examine the accuracy of the algorithm and after achieving acceptable results the algorithm is tested against large and difficult datasets like MNIST which is explained in the next section 2.8.2.

2.8.2 MNIST Dataset

Modified National Institute of Standards and Technology(MNIST) is a large handwritten dataset consist of 60000 image samples for training and 10000 image sample for testing purposes. MNIST database is derived from a large National Institute of Standards and Technology(NIST) database as NIST is not suitable for machine learning applications. MNIST is one of the commonly used datasets in the field of machine learning like image processing, and pattern recognition because the data is

readily available and doesn't require formatting and preprocessing.

The MNSIT database is a set of images of $[28 \times 28]$ pixel each with grayscale containing digits information bound between [0, 9]. So this is used in supervised learning to classify numbers between [0, 9]. Each image in MNIST is annotated with labels denoting the correct number. Since the MNIST has achieved success in terms of benchmarking dataset, similar MNIST like datasets have been created for a different purpose such as Fashion-MNIST, Kannada MNIST, etc.

In this thesis, MNIST dataset have been used to measure the accuracy of the system and determine the performance bottleneck by providing all 60000 images as an input to the system.

2.9 Problem Statement

ML toolkits assist deep learning practitioners with their research. The most popular programming model for the tools is static declaration[19]. However, these kinds of models perform poorly on the machine algorithms that have the following unique features[19]:

- **Variably sized inputs:** A common occurrence in natural language processing algorithm is the differences in speech pattern sizes.

- **Variably structured inputs:** Tree[53] and graph structure[54] algorithm where structure of computation varies with input.

- **Variably structured outputs:** Unique scenario when the algorithm search space is restrictive to popular points [55].

LL0, the dynamic machine learning algorithm which is the focus of this thesis shares the majority of the above-mentioned traits. This makes it unique and the common toolkit will perform poorly for these algorithms.
The tool-kits [19][56] have provisions to handle the above mentioned unique properties. This is achieved at a cost by implementing a dynamic declaration paradigm, which increases the computational cost.

PyTorch [18], which is inspired by [19][56], is the most famous toolkit in the machine learning community for researching models. However, when the implementation of LL0 was built it faced scalability and performance issues. Similarly, [17] implementation of LL0, built on NumPy, faced an identical issue.

This thesis has identified two reasons for these issues, firstly, the implementation of LL0 was graph-based whereas PyTorch is an array-based programming modeland thus it encountered a compatibility issue. Secondly, PyTorch is designed for flexibility to describe models. But, LL0 has unique features described in Section 5. The very high level of granular control which is provided by PyTorch adds complexity in

describing LL0 which increases the overhead of control flow and data flow descriptions.

The solution is to invent a unique software architecture, which defines LLO and has granular control to deliver high performance when compared to the implementation described in PyTorch. The paper [19] best describes the following challenges involved with building unique solutions:

- **Difficulty in expressing complex flow-control logic:** When an algorithm needs a complex data structure to define itself or require a nontrivial control flow. It is difficult to implement them correctly in a higher-level language and implementing them directly on a computational graph toolkit is more complicated and needs significant efforts from the user.

- **Complexity of the computation graph implementation:** The result of a custom data structure implemented on a preexisting computation graph doesn't support such structures. This increases the complexity of implementation and reduces the possibility of optimization resulting in a nontrivial challenge.

- **Difficulty in debugging:** The best location for catching an error is during the time of declaration. However, in the case of an algorithm that is dynamic, logical errors might occur during the execution. This is because the structure might change from that of its original declarations, which makes it difficult to identify the location of crashes thus making the debugging process inherently demanding.

This thesis explores a software architecture, described in Section 4.1, which provides a basic platform for users to build dynamic structures. The proposed software architecture has overcome the above-mentioned challenges by introducing a complex data structure called layers which captures the essence of the LL0 algorithm.

The usage of concepts of layers instead of a computational graph helps in defining the structures. This defines the LL0 algorithm better than the fine-grain control provided by the computation graphs. The complex control flow logic and the optimization are done in the backend, which is invisible to the users. The burden of debugging is also reduced because tests and checks for logic errors can be built directly upon the layers at the time of the declaration itself.

The proposed architecture is then optimized for performance and energy efficiency.

3

Overview of LL0 Algorithm

LL0 is a machine learning algorithm that focuses on learning continuously from the environment, and this process is known as life-long learning [57]. To accomplish this, it leverages deep learning to build and train a network. LL0 predominantly focuses on forming an efficient network for individual data set with high accuracy. This is achieved by using network-modification mechanisms for making predictions with precision. The LL0 network starts from a blank state and develops its structure dynamically according to the algorithm described below.

3.1 LL0 Algorithm

The LL0 has the dynamic capability to grow and reduce the size of the network, by adding or removing neural nodes, continuously during training. The network is developed to predict an outcome by recognizing a pattern from the available training data. The main loop of the algorithm is as shown in Algorithm 1[15].

The neural network starts with no neural nodes and develops its structure based on the input, and recognizes a pattern from the available training data. The network predicts a result by computing the incoming data. If the prediction matches with actual output, a fine-tuning of the learning parameters is performed known as backpropagation, explained in detail in Section 5.4. Or else the new nodes are created which is known as the extension, which is explained in detail in Section 5.2. LL0 creates a complex dynamic network during training and these structure changes based on the input data. Generalization is performed to generalize the data to get better accuracy. The LL0 needs to update the current classifier by accommodating new continuous streams of data without affecting the previously acquired knowledge. This process is continued until every data point is fed to the system. Learning from input data results in memory expansion which affects the performance of the system. Thus it would be better to discard the relatively less used nodes and these obsolete nodes are removed by executing forgetting as illustrated in Algorithm 1.

Algorithm 1: Main Loop of LL0

```
receive the first data point (x, y) ;
form |x| input nodes and |y| output nodes ;
while true do
    compute network output ŷ produced by input x;
    if prediction (ŷ) ≠ y then
        generalization;
        Extension;
    else
        backpropagation;
    end
    forgetting;
    receives new data points (x, y);
end
```

Unlike traditional Deep Neural Network (DNN), LL0 uses four different forms of neuroplasticity [15]:
(i) *backpropagation*, which fine-tunes the parameters;
(ii) *extension*, which builds the network by adding new nodes;
(iii) *forgetting*, which removes the least used nodes;
(iv) *generalization*, which adjusts the node parameter that potentially prevents creating extra nodes due to extension.

LL0 has four types of nodes where each node constitute distinct activation functions: *input node*, which has identity activation function. *Output node*, which has softmax activation function. *value node*, which has Gaussian activation function. *Concept node*, which has sigmoid activation function. These activation functions are explained in the next section.

3.2 Mathematical Activation Function

The activation function is an abstraction that represents whether or not the node is activated. The activation function facilitates the output to stay between the acceptable interval. Nonlinearity is introduced in the form of the activation function where nonlinearity is a threshold-based function. To explain it better we can consider an example: If the value of the node is above the threshold then the node is said to be activated. When more than one node has a non-zero value, the activation function decides to fire the correct classified nodes whose value is above the threshold.

3.2.1 Gaussian Activation Function

The Gaussian activation function constitutes the value node in LL0, which is rare among neural networks because it has two parameters that are expensive to compute. The output of the Gaussian activation function is bound between [0, 1] and the

Gaussian function is defined in equation 3.1.

$$g(x) = \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{3.1}$$

where μ denotes the peak of the activation function. The slope of the curve is controlled by σ. For example in Figure 3.1 the value of both μ and σ is 0.5.

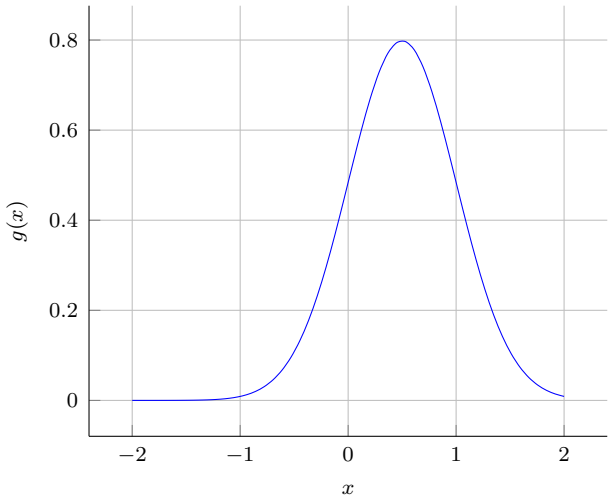

Figure 3.1: Gaussian activation functions $g(x) = \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$, where $\mu = 0.5$, $\sigma = 0.5$.

3.2.2 Sigmoid Activation Function

The sigmoid activation function is an activation function of the concept node and is a regularly used activation function in a neural network because of its nonlinear nature. The sigmoid function is differentiable, which has a non-negative derivative at each point [58]. The output of the sigmoid function is always bounded within the range [0,1] which makes it easier to make a distinct prediction. A small change in the value of x has a significant effect on $f(x)$ when the x value is close to zero. Besides, $f(x)$ responds slowly at the far end because of a small gradient shown in Figure 3.2. Sigmoid function can be expressed as equation 3.2.

$$f(x) = \frac{1}{1+e^{-x}} \tag{3.2}$$

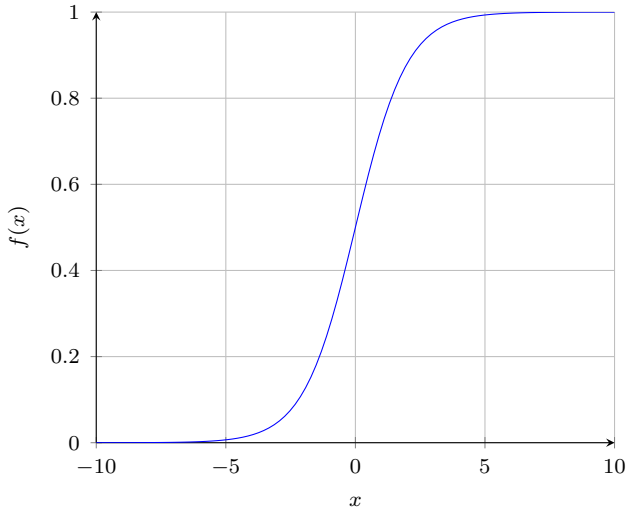

Figure 3.2: Sigmoid activation functions $f(x) = \frac{1}{1+e^{-x}}$, where x defines the slope of the curve.

3.2.3 Softmax Activation Function

The softmax function is an activation function of the output node, which is frequently used for classification purposes. Softmax function accepts the set of inputs and outputs the probability of the predicted neuron. Softmax function can be defined as equation 3.3.

$$Softmax({z_j}^{(L)}) = \frac{e^{{z_i}^{(L)}}}{\Sigma_j e^{{z_j}^{(L)}}} \tag{3.3}$$

where $e^{{z_i}^{(L)}}$ stands for weighted sum of the i^{th} softmax neuron and $\Sigma_j e^{{z_j}^{(L)}}$ is the sum of all j softmax neurons.

4
Software Architecture

This section describes the proposed software architecture for the implementation of LL0. It includes a description of data storage, dynamic networks, and the key concept of layers.

4.1 Software Architecture

The LL0 algorithm has a dynamic neural network structure. The paper [19] argues that the machine learning toolkits will face performance issues on the dynamic structure. One such toolkit is PyTorch[18] , which is derived from [19][56]. When LL0 was implemented on PyTorch, it encountered a scalability issue [17]. The paper [17] suggested a need for a specialized solution. This thesis conceives a software architecture for LL0 which delivers better results than the implementation in PyTorch.

The most common machine learning toolkit [33][34][35] uses the computational graph as a central design principle for software architecture. A computational graph is a symbolic representation of a numerical equation that is used to optimize the workload of machine learning algorithms. The use of a computational graph provides fine-grain control over the implementation of the algorithm. However, complex control flow and the data flow logic are needed to use this fine-grain control. The computational overhead of the toolkit increases proportionally to the complexity in defining the structure of an algorithm.

The shape of the LL0 neural network grows with the data. When implementing the algorithm in a toolkit, the computational graph will grow proportionally to the LL0 neural network. This adds overhead and the constant recomputation will make performance diminish over time. Hence the core principle of the proposed software architecture is to limit this growing complexity and overhead of an expanding LL0 neural network.

The design principle of the proposed architecture is inspired by Caffe [59]. It is a machine learning toolkit specialized for images that use the concept of *layers* over computational graphs. *Layers*, in essence, resembles a neural network layer where it accepts input and drives output after computation. *Layers* are well suited to resolve the problem of the growing complexity of expanding LL0 neural network.

In *layers*, the logic and data flow of expansion can be built inside. Therefore, when the network grows, only new *layers* have to be added. Adding new *layers* only increases the computation cost of data. It also saves the overhead of complex data paths and recomputation of graphs. The growing complexity is limited by having them pre-computed and built-in layers.

Every time the LL0 neural network changes, the computational graph have to be recomputed. However, in the case of *layers*, the logic of expansion is inbuilt, which needs no recomputation. Additionally, it reduces the overall overhead of the network. There are few disadvantages as well, unlike the computational graph, the layers need manual optimization and support to build the structural shape of LL0.

4.1.1 Architecture

There are two major challenges for this software architecture to overcome. Firstly, it has to provide the users with an abstraction. This abstraction has to be simple enough for users to define algorithm and various dynamic neural networks. It also has to capture the dynamic nature of the LL0 and hide underlying systems and hardware dependencies. Secondly, it has to provide performance. The goal of this thesis is to provide better performance than existing solutions. The abstraction has to have a tightly coupled mapping with the GPU.

The proposed software solves these challenges by utilizing the concept of *layers*, shown in Figure 4.1. The abstraction was built using the concepts of *Runtime polymorphism* and *Object oriented programming (OOP)*. These are used to hide the inner working from the users. It also provides an abstraction that is used for building LL0. The performance challenge was solved by breaking *layers* into multiple subunits, shown in Figure 4.2. Each subunit holds optimized GPU kernel invocations for computations. It ensures better mapping of algorithms and easier to profile the code.

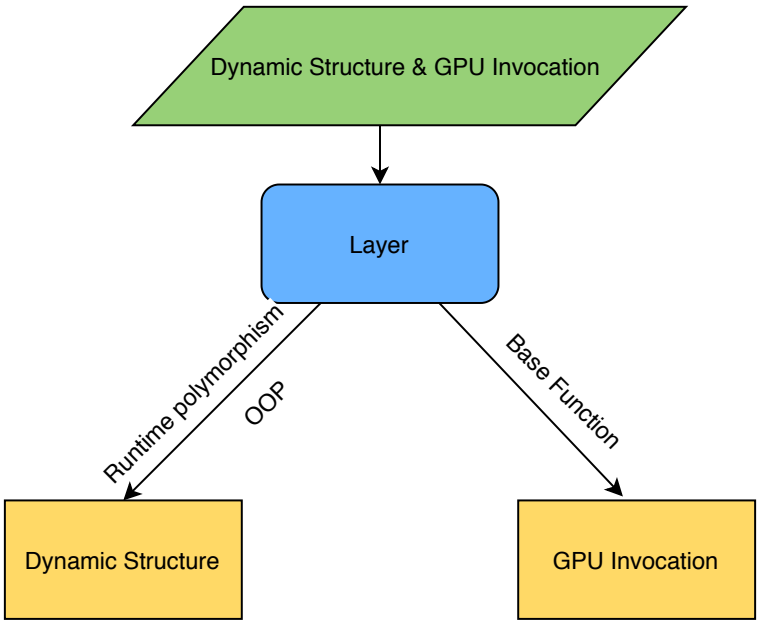

Figure 4.1: The solution to the challenges of LL0 implementation.

The following part contains the description of the proposed architecture.

4.1.1.1 Data Storage

The software architecture utilizes a single type of structure for data storage known as the mat, which is a multidimensional array with multiple CPU and GPU invocation inbuilt functions. The mat can be used to store input data, parameters, or placeholders. This is used by layers as a memory storage unit.

The mat includes functions for the tasks of, 'reserving memory in the GPU Dynamic Random-Access Memory (DRAM)', 'releasing memory', and 'pointers to access data stored in it'. Usage of the mat has three main advantages over generic data structures. The first one is the reduction in development time by providing a storage structure compatible for LL0. The second one is an increase in performance because the data storage is in GPU DRAM and lastly the mat uses an array-based storage system, which is better equipped to handle matrix calculations.

4.1.1.2 Layers

The concept of *layers* in Caffe[59] serves as a neural network. However, in the proposed software architecture, the *layers* is modular and behaves as a neural network or as an activation function. The behavior depends on the combination of its underlying subunit. The *layers* also contains detailed data flow logic which dictates the computation.

LL0 is comprised of the input, value, concept, and output nodes, described in detail in Section 5.1. These nodes have a unique property and are distinct in functionality. A specific combination of these nodes forms the LL0 network as shown in Figure 4.3. Their rules of combination are discussed in section 5.2.2.

The software architecture uses *layers* to build the LL0 network. *Layers* are modular and swapping the underlying subunits can represent the nodes. A specific combination of subunits can resemble all the unique and distinct nodes, which are described in the below sections. To create the LL0 network with the software architecture the *layers* are combined in the fashion described in section 5.2.2.

Layers consist of three subunits called base function, neural layer, and neural network as shown in Figure 4.2:

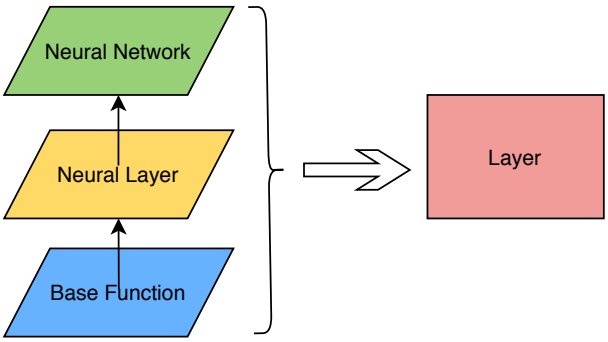

Figure 4.2: Components of the layer: base function, neural layer and neural networking merging to form layer. Layer is the building block of implementation of LL0.

Base Function: The base function contains all the declarations and the GPU invocation needed for a particular node type.
Neural Layer: The neural layer is a template for combining all different kinds of base functions together.
Neural Network: Neural Network is a template designed to hold multiple layers which together form the LL0 network.

Figure 4.3 shows a pictorial description of the LL0 algorithm. It consists of a combination of input, value, concept, and output nodes. To recreate the identical structure in software, the architecture uses *layers*. By changing the *base function*, it can represent all the above nodes. To combine the different *layers*, users can use the *neural network*. Figure 4.4 is the visual representation of the algorithm implemented in the thesis software architecture.

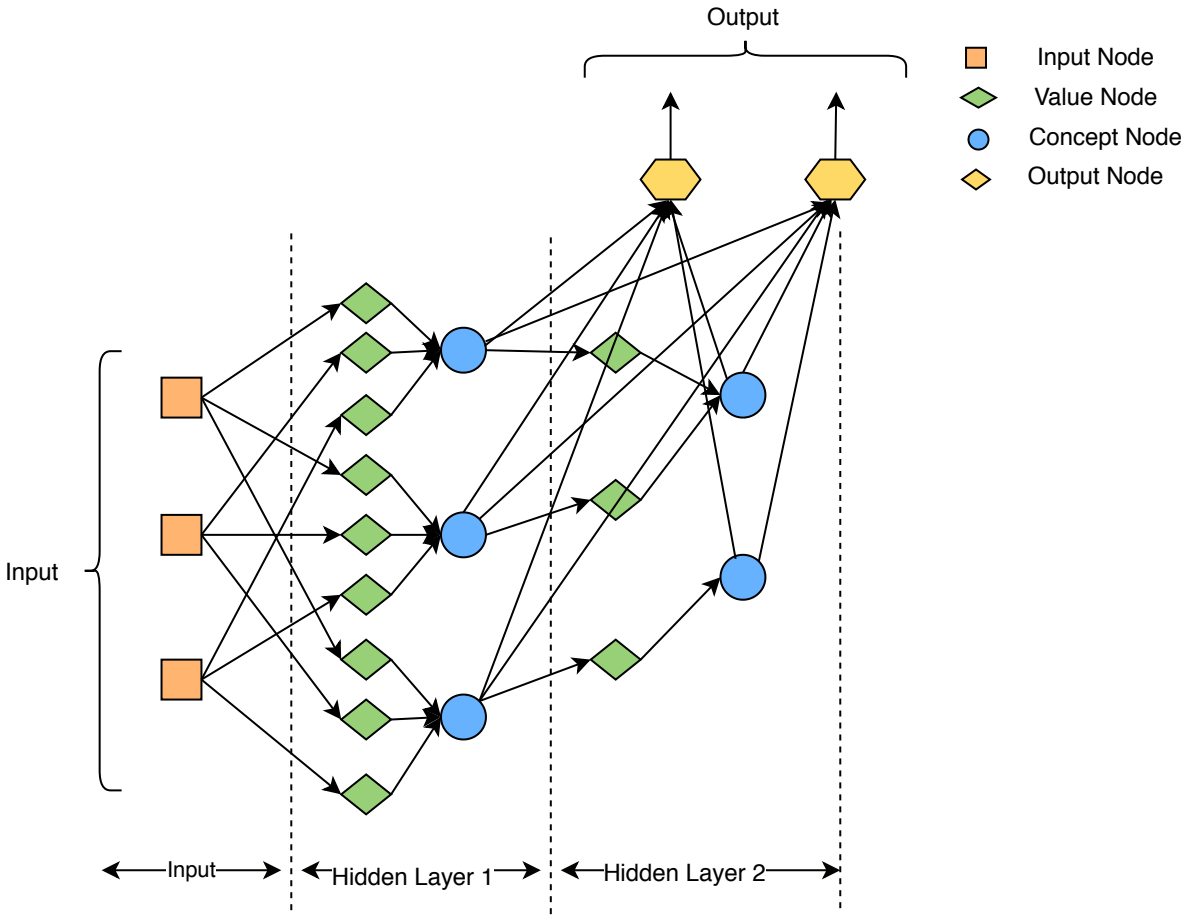

Figure 4.3: A representation of LL0 network with hidden layer and nodes.

There are some limitations of this software architecture. As the software architecture is centered around the LL0 algorithm, the performance of the architecture on other machine learning algorithms is unknown. Also, this software architecture is committed to a CPU-GPU combination and a single machine implementation. Therefore it offers no support for different hardware accelerators or to multiple devices.

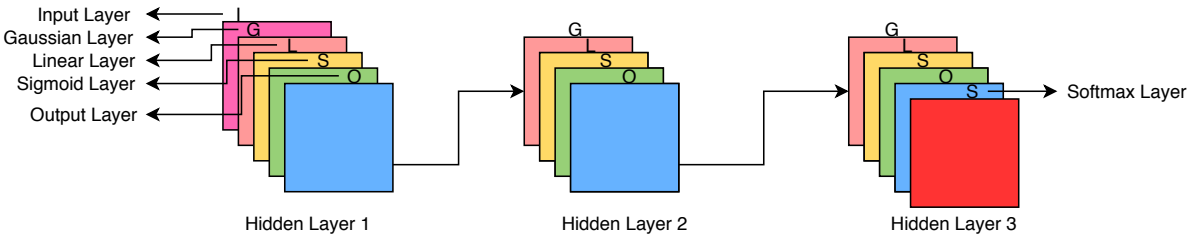

Figure 4.4: Software architecture of the implementation of LL0 depicting the data path and the hidden layer compartmentalisation.

5

The LL0 Implementation

This section aims to provide information about the implementation of LL0 based on the proposed software architecture and its key concepts in detail. Further this section provides a detailed description of each node, extension rules, and backpropagation. This is inspired by the theory of neuroplasticity. Additionally, it outlines the impact of hyperparameters.

5.1 Basic Units of LL0 Algorithm

LL0 is a machine learning algorithm with a dynamic structure as presented in Section 3.1. Besides, this is a unique ML algorithm when compared to other ML algorithms as it doesn't have a neural structure in the beginning. LL0 constructs its structure as and when data is supplied to the system.

LL0 is comprised of four types of nodes, namely *Input node*, *Value node*, *Concept node* and *Output node*. In the beginning, LL0 consists of only an input node and an output node. When the network makes the incorrect prediction, a new value node and the concept nodes are created to rectify this error. This method of adding new nodes to the network is called Extension, where the extension is bound with stringent rules to add new nodes to the network.

As described in Section 3.1 each node constitutes distinct activation function. The four node types are:

- *Input Node:* It has an identity activation function and it always broadcasts its value to the value nodes. Input nodes have multiple outgoing edges connected to every value nodes as shown in Figure 5.1.

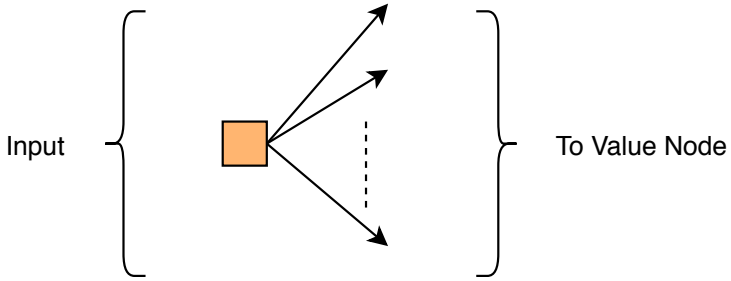

Figure 5.1: Input Node depicting multiple outgoing edges to the value nodes.

- *Value Node:* It has a Gaussian activation function that always receives data from the input node. The middle of the Gaussian activation function defines the value of the node to be retained. The width of the slope is determined by standard deviation, σ. The value node is an expensive function to compute as it has two parameters μ and σ. The value node always has one incoming and outgoing edge. The outgoing edge is always connected to concept nodes as shown in Figure 5.2.

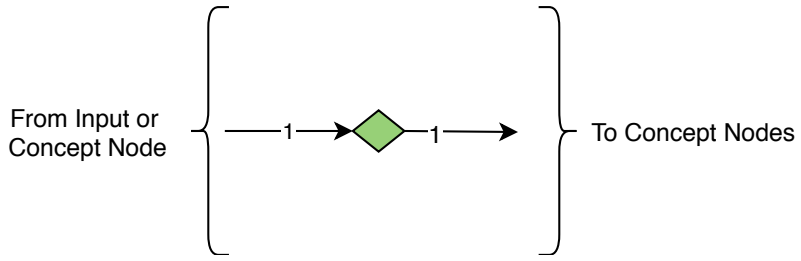

Figure 5.2: Value Node depicting one incoming edge either from Input Node or Concept Node and outgoing edge to the concept node.

- *Concept Node:* It has a sigmoid activation function that always receives data from the value nodes. Concept nodes can receive data from multiple value nodes. It computes the sigmoid activation function based on the incoming data. The outgoing edge of the concept nodes are connected to both value nodes and every output node as shown in Figure 5.3.

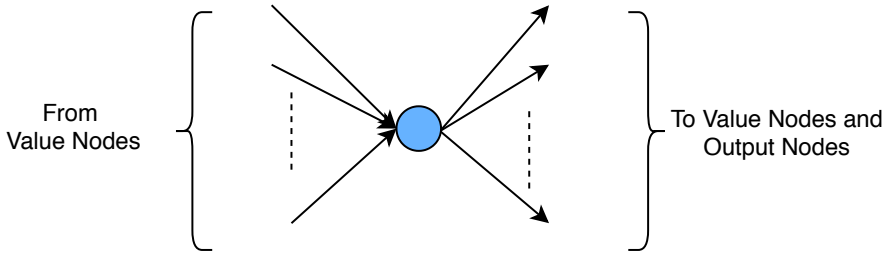

Figure 5.3: Concept Node depicting incoming edges from value node and outgoing edges to value node and output nodes.

- *Output Node:* It has a softmax activation function. Each output node has incoming edges from all the concept nodes in the neural network. The outgoing edge is the output of the softmax function shown in Figure 5.4.

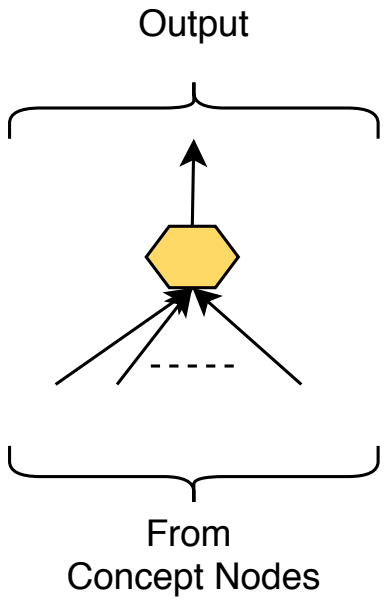

Figure 5.4: Output node depicting incoming edges from every concept nodes and outgoing edge is the output of the softmax activation function.

As specified earlier, LL0 develops its neural network depending on the principles of neuroplasticity, which is transformed into mathematical forms knows as *Extension*, *Backpropagation*, *Generalisation* and *Forgetting*. These rules constitute the LL0 algorithm illustrated in Algorithm 2. When the system is provided with the data (x, y) where |x| is data with its label |y|, the prediction denoted by $\hat{y}$ is evaluated first by performing feedforward and backpropagation, which is explained in Section 5.4. The predicted output is then compared with the actual output. If the prediction is incorrect, new nodes are created by executing an extension, where the extension is restricted by a set of extension rules. Further, feedforward is performed to determine the correctness of the data. When the new prediction matches with actual data, new data points (x, y) are received. This process continues until all the data points are supplied to the network,

Algorithm 2: Main Loop of LL0

Receives data points (x, y) with data |x| and label |y| ;
The predicted output of x is denoted by $\hat{y}$;
while *true* **do**
  Feedforward;
  Backpropagation;
  if *prediction* $\hat{y} \neq y$ **then**
    Extension set*;*
    Extension rule*;*
    Feedforward*;*
  end
  Receives new data points (x, y)*;*
end

5.2 Extension

The strength of the LL0 algorithm is to add new nodes to the network, known as extension. This gives LL0 a dynamic approach not only by adding new nodes to the existing network but consequently making the network expand on its own. When the input data is fed to the network, if the network's prediction is incorrect the new nodes are created to correct the error. However, the extension process has stringent rules to add additional new nodes to the network as shown in Algorithm 3. The rules for extensions may differ depending on the position in the network where the extension is triggered. These rules are explained in Section 5.2.2. However, the control on extension is decided by the machine learning practitioner for fine-tuning the model for distinct data sets.

5.2.1 Extension Set

During feedforward, when the network makes an incorrect prediction, where it doesn't recognize the pattern i.e. predicted output does not correspond with actual output, then the extension is triggered. Firstly the algorithm scan for the extension set, which is created based on the value of concept node x. The value of the concept node x must be higher than the value of predetermined threshold t ($x \geq t$). Additionally, it should not have a child node. The concept node meeting these two criteria gets activated and is therefore referred to as the parent node. Two or more concept nodes that have values greater than threshold t constitute an extension set. These parent nodes get eliminated from the extension set as a result of child nodes being formed from them. It is should be noted that the extension set can be formed either by the concept nodes or input nodes.

5.2.2 Extension Rules

When the extension is triggered, the extension sets are created. It is essential to follow some protocol to determine where to add new nodes in the network that can correct the error and improve the overall result. The rules are described below.

As illustrated in Figure 5.5, when the extension is triggered and if only one concept node is activated, then the network must be extended from the input nodes. If the network makes an incorrect prediction, a new node (3) is created from the input nodes (0) and (1). This is because only one concept node (2) exists in the network. As per the extension policy, at least two or more nodes in the network have to be activated in-order to extend from the concept nodes.

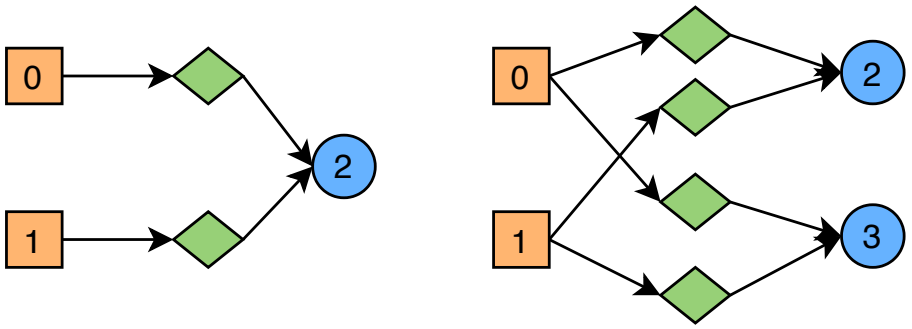

Figure 5.5: Illustration of how the new node is added into the neural network from the Input Node

When an extension is triggered and if one or more concept nodes are activated, then the network can be expanded from the concept nodes whose value is above a certain threshold. When the network executes an incorrect prediction and a new node(4) is created from the concept node (2) and (3) if these nodes have activation above threshold as shown in Figure 5.6.

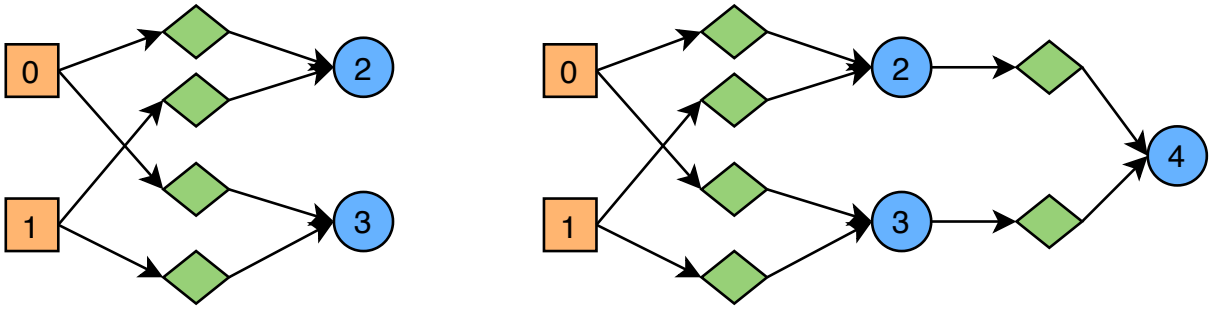

Figure 5.6: Illustration of how the new node is added into the neural network from the concept nodes

As presented in Figure 5.5 and Figure 5.6, the value node is created between the concept and input node. In addition, this can also be created between the two concept nodes. The outgoing edge of the concept node is connected to all output nodes.

Updating the extension policy and rules, now the algorithm can be described as shown in Algorithm 3.

Algorithm 3: Main Loop of LL0 with Extension Rule

```
Receives data points (x, y) with data |x| and label |y| ;
The predicted output of x is denoted by ŷ ;
while true do
    Feedforward;
    Backpropagation;
    if prediction ŷ ≠ y then
        Extension Triggered ;
        Scan for Extension Set;
        if Extension Set Found then
            Extend from concept node;
            Extension rule;
            Feedforward;
        else
            Extend from Input node;
            Extension rule;
            Feedforward;
        end
    end
    Receives new data points (x, y);
end
```

5.3 Hyper Parameters

The initial parameter values of the nodes affect the accuracy of the algorithm. It is necessary to have accurate values for all the nodes because the determination of pattern recognition is dependent on these values. When the data is supplied to the system, if the parameter values are incorrect, unnecessary nodes are created that results in low accuracy and performance. The activation function recognizes data patterns if these parameters have the correct value. Further, the newly created nodes will also have the correct value. The node values are decided by parameters such as μ, σ in the Gaussian function, bias b in the sigmoid function, and weights w in the linear function as specified in Section 3.2. The initial values for these parameters are determined by using hyper-parameter such as a_v and b_c, where a_v controls the slope of the Gaussian activation function and b_c is responsible for the slope of the curve in sigmoid function.

The equation used to determine the parameter a_v for the Gaussian equation is presented as:

$$g(x) = \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) = 0.5, x = \mu + a_v \tag{5.1}$$

By setting $x = \mu + a_v$, the result of 0.5 can be achieved by moving a_v away from the center. In the above equation 5.1, μ denote the center of the activation function. When the value node is created, the activation value of the parent node is assigned to the value of the center. The hyper-parameter a_v denotes the distance away from the parent activation. It is necessary to move a_v away from the center to achieve the 50% of the input. This is because the incoming weights to the value nodes are constant 1 and they do not affect the value node's behavior.

Standard deviation, σ, can be found by solving the above equation and it is noted that the initial value of the σ is dependent on hyper-parameter a_v as shown in the equation below,

$$\sigma = \pm\frac{a_v}{\sqrt{2}\sqrt{ln(2)}} \tag{5.2}$$

For Sigmoid activation function, below equations are used,

$$f(nw + b) = \frac{1}{1 + e^{-(nw+b)}} = 0.9 \tag{5.3}$$

$$f(a_c nw - b_c nw + b) = \frac{1}{1 + e^{-(a_c nw - b_c nw + b)}} = 0.5 \tag{5.4}$$

where a_c is always equal to 1 and b_c is the ratio of the value nodes, to get the output of sigmoid as 0.5. b is the bias, w is the incoming weights to the concept nodes and n is the total number of parents to the concept node. The initial value of the concept node is determined by solving the above equation. It is observed that these initial values are dependent on parameters a_c and b_c as shown below,

$$w \approx \frac{-2.19722}{n(a_c - b_c - 1)}, n(a_c - b_c - 1) \neq 0 \tag{5.5}$$

$$b = logit(0.9) - \frac{2.19722}{-a_c + b_c + 1}, (-a_c + b_c + 1) \neq 0 \tag{5.6}$$

where $logit(x)$ represents inverse of the sigmoid function. After the tuning parameters a_c and b_c are set, equations 5.5 and 5.6 are used to get the initial values for the concept node. Since each concept node receives data from a distinct parent node, setting up the correct initial value to achieve good prediction is difficult. Since the network assigns these values dynamically during extension, it is important to set the initial parameter to the correct value otherwise unnecessary nodes get created. For example, if a wrong concept node is activated and it doesn't recognize the pattern, the new node will get created based on the values of activated nodes(parent node). Since these values are incorrect, this affects the accuracy and performance of the network. If the initial parameter value is slightly inaccurate, and if the new

nodes are created from these nodes. Their values can be adjusted by executing backpropagation which is explained in Section 5.4.

5.4 Backpropagation

LL0 exercises backpropagation to adjust the learned parameters to achieve a better result. Once feedforward is completed, the backpropagation is triggered and the following parameters of bias ($b_i^{(l)}$), mean ($\mu_i^{(l)}$), standard deviation ($\sigma_i^{(l)}$), weights ($w_i^{(l)}$) and ($W_i^{(l)}$) are fine-tuned or updated after the backpropagation task is completed.

The backpropagation in LL0 is calculated backwards, i.e. the calculation starts from the output node and continues until input nodes. Additionally, the cross-entropy error function is used in LL0.

$$E = \sum_{d=0}^{D} t_d * y_d^{(L)} \tag{5.7}$$

Where D is the output dimension of the problem, t_d is the target label and $y_d^{(L)}$ is the output of the softmax activation function which is the output of the network. The derivative of the error concerning the parameters is calculated to determine the update rule for each parameter.

The partial derivative of the error with respect to the weighted sum of the output node is given below,

$$\frac{\partial E}{\partial z_i^{(L)}} = y_i^{(L)} - t_i \tag{5.8}$$

Equation 5.8 denotes the derivative of the error through the softmax function to the weighted sum. This equation is used in all the concept nodes considering all of them are connected to every output node. As stated, backpropagation starts from output nodes and continues until the input nodes. For each concept nodes, derivative for bias ($b_i^{(l)}$), and weights ($W_i^{(l)}$) between concept nodes and output nodes are calculated and are presented as follows,

$$\frac{\partial E}{\partial W_{i,j}} = \frac{\partial E}{\partial z_j^{(L)}} * y_i^{(l)} \tag{5.9}$$

$$\frac{\partial E}{\partial y_i^{(l)}} = \sum_{o_j^{(L)} \in OutputNodes} (\frac{\partial E}{\partial z_j^{(L)}} * W_{i,j} * y_i^{(l)}) + \sum_{v_j^{(l+1)} \in children(c_i^{(l)})} (\frac{\partial E}{\partial z_j^{(l+1)}}) \tag{5.10}$$

$$\frac{\partial E}{\partial z_i^{(l)}} = sigmoid(z_i^{(l)})(1 - sigmoid(z_i^{(l)})) * \frac{\partial E}{\partial y_i^{(l)}} \tag{5.11}$$

$$\frac{\partial E}{\partial b_i^{(l)}} = \frac{\partial E}{\partial z_i^{(l)}} \tag{5.12}$$

Equation 5.9 is obtained upon performing backpropagation from output nodes to concept nodes by applying the chain rule. Similarly, equation 5.10 is obtained which defines the incoming derivatives. It is necessary to compute concept nodes derivatives carefully as they receive derivatives from both output nodes and value nodes. The first sum in Equation 5.10 denotes the incoming values from the output nodes and the second sum computes the incoming derivative from the value nodes. $c_i^{(l)}$ denotes the set of children node of the type value node. The deepest(rightmost side) nodes in the network don't have children nodes, so the incoming derivatives from the value nodes are zero because they are only connected to output nodes.

For value nodes, derivative of the mean ($\mu_i^{(l)}$), standard deviation ($\sigma_i^{(l)}$) and weights ($w_i^{(l)}$) are calculated and are given as below,

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial z_j^{(l)}} y_i^{(l-1)} \tag{5.13}$$

$$\frac{\partial E}{\partial y_i^{(l-1)}} = \frac{\partial E}{\partial z_j^{(l)}} w_{i,j}, j = child\_id(v_i^{(l-1)}) \tag{5.14}$$

$$\frac{\partial E}{\partial \mu_i^{(l-1)}} = \frac{\partial E}{\partial y_i^{(l-1)}} \frac{(z_i^{(l-1)} - \mu_i^{(l-1)}) \exp[-\frac{(z_i^{(l-1)} - \mu_i^{(l-1)})^2}{2(\sigma_i^{(l-1)})^2}]}{(\sigma_i^{(l-1)})^2}$$

$$\frac{\partial E}{\partial \sigma_i^{(l-1)}} = \frac{\partial E}{\partial y_i^{(l-1)}} \frac{(z_i^{(l-1)} - \mu_i^{(l-1)})^2 \exp[-\frac{(z_i^{(l-1)} - \mu_i^{(l-1)})^2}{2(\sigma_i^{(l-1)})^2}]}{(\sigma_i^{(l-1)})^3}$$

$$\frac{\partial E}{\partial z_i^{(l-1)}} = \frac{\partial E}{\partial y_i^{(l-1)}} * -\frac{(z_i^{(l-1)} - \mu_i^{(l-1)}) \exp[-\frac{(z_i^{(l-1)} - \mu_i^{(l-1)})^2}{2(\sigma_i^{(l-1)})^2}]}{(\sigma_i^{(l-1)})^2}$$

Since value nodes have one incoming and outgoing edge, where the incoming edge is always '1' and the outgoing edge is trainable which is derived in Equation 5.13. $\frac{\partial E}{\partial z_i^{(l-1)}}$ is used to find the derivatives of the recursive nodes which can be noticed in Equation 5.10.

These equations are needed to find the derivatives of both the value and concept nodes. This procedure is continued until backpropagation has reached to input nodes. Once backpropagation is completed, the parameters are updated such that the network error is minimized. The parameters are updated using the gradient descent as shown below,

$$p_i \longleftarrow p_i - \delta * \frac{\partial E}{\partial p_i} \tag{5.15}$$

Where δ is the learning rate. The backpropagation propagates through all the nodes in the network until it reaches input nodes and adjusts all the parameters to get good accuracy.

6

Experimental Methodology

This section presents the experimental methodology used to improve the performance of the thesis implementation. Additionally, it presents how to quantify the performance by setting up the benchmark and strategies for profiling, to increase the performance.

6.1 Methodology Approach

The goal of this thesis is to create a new implementation of the LL0 algorithm that performs better than the existing PyTorch implementation. To achieve this objective, the LL0 is implemented using C++, and its performance has improved by following the methodology.

The methodology is a framework for improving the performance of the system. It involves defining and measuring performance and analyzing results. Performance is evaluated as a set of benchmarks. The flow includes the use of tools such as nvprof and visual studio profiling tools for analysis. Hereafter, this framework for improving the performance is referred to as methodology in this report.

The hardware platform is established before the execution of the below steps. This is because of two main reasons. Firstly, the benchmark can be evaluated over the same platform. This eliminates any unfair hardware advantages. Secondly, the fine-tuning for performance is dependent on the comprehensive utilization of the GPU.

The below-mentioned steps form the methodology and are explained briefly below. For this section, the thesis implementation is referred to as C++ implementation.

1. Verification
2. Executing benchmarks on C++ and PyTorch implementation.
3. Profiling.
4. Identifying bottleneck.
5. Modify software structure to relieve the bottleneck.
6. If no new bottleneck is found exit the step or else iterate.

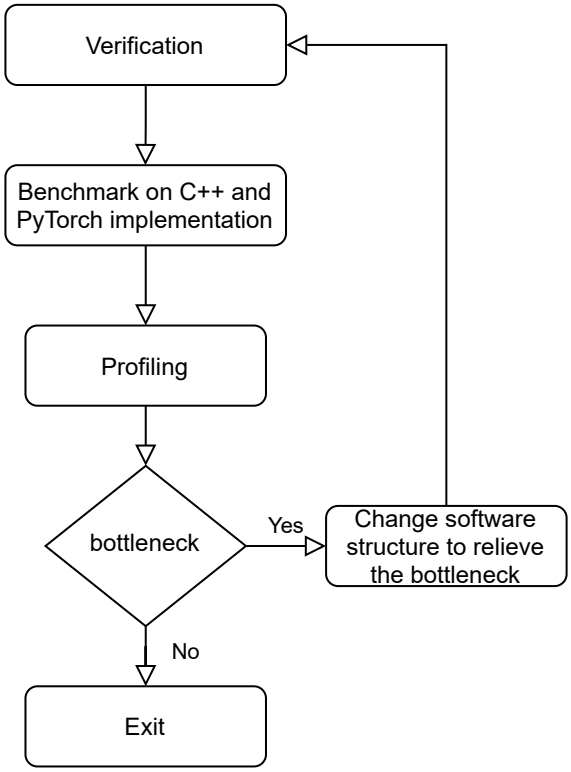

Figure 6.1: Methodology iteration flow.

6.1.1 Verification

Modifying software architecture opens the possibility of affecting the functionality of the implementation. Since the C++ implementation is a custom-made solution, modifying it for performance might lead to deviation from the working of LL0 algorithm. Verifying the functionality of implementation prevents deviation from the algorithm.

6.1.2 Executing Benchmarks

The next step of the methodology is to execute the benchmarking with both the implementations. Benchmarks execution provides valuable information in regards to variations in performance. The impact of improvement in performance on the removal of potential bottlenecks in the C++ implementation is studied.

6.1.3 Profiling

Profiling tools are run while monitoring the C++ implementation executing datasets. The information from the tools is used in further steps for identifying bottlenecks. The explanation of the profiling strategy is in the following subsections. To avoid fine-tuning the performance for a specific dataset, multiple datasets are used for analysis.

6.1.4 Identifying Bottlenecks

Data from profiling tools and benchmarking results are analyzed. Potential bottlenecks and datapaths for parallelism are identified for performance improvement.

6.1.5 Modify Software Architecture

After identifying the potential bottlenecks, modification is performed on the software architecture which opens the possibility of the implementation to deviate from LL0. Some optimization has more effect than others. For example, modifications for mapping of implementation with the streaming multiprocessors do not affect functionality. However, changing the data path of computations affects the functionality of the systems, which need to be tested.

6.1.6 Exit Strategy

Since the methodology is iterative, it is exited when no potential bottlenecks are found.

6.2 Hardware Resource

To compare two different implementations of an algorithm, there is a need to set up a common platform that will provide a common ground to compare the implementation. This helps to eliminate any advantages which might occur if the implementation were running on a separate device.

The development environment of this thesis consists of Intel core i5-7600 CPU @ 3.50GHz processor, 32 GB RAM, and NVIDIA GeForce GTX 1060 6 GB, with 10 streaming multiprocessor(SM), with 1280 CUDA Cores, 6144 MB GDDR5 memory (192.19 GB/s bandwidth) connected in a PCI-Express x16 Gen 3 slot [60].

6.3 Setting up of Benchmarks

Benchmark is defined as standardized monitoring of performance, which is repeated for comparisons between the systems. There are mainly two types of benchmarking namely micro and macro-benchmark. Microbenchmarks are for measuring components and units where as macro benchmarks are for the entire system.

For the thesis, the focus is on microbenchmarks because the goal is to achieve hardware acceleration of the algorithm over the building of machine learning toolchains.

In this thesis, the comparisons are between custom-built C++ and PyTorch. Here the PyTorch implementation is a treated black box whose inner working of the system is unknown, which adds a challenge in finding a benchmark for comparisons. Quantitative properties like time and memory consumption can be evaluated, unlike

qualitative properties like the number of floating-point operations.

Total time spent and memory consumption can be the potential benchmark. However, it falls short as these metrics do not provide valuable performance information, for example, the effect on performance when scaling up of implementation on larger datasets. The below-described benchmarks are used for comparisons.

6.3.1 Benchmark: Number of Feedforward in the unit of Time

The movement of data from input to output in a ML neural network is called feedforward or inference. This process is used by the neural network to make a prediction from a given set of input data. Feedforward is statistically the most reoccurring process in ML algorithms both in the training and the testing phase. The bulk of the work is measured by averaging the number of feedforward in a unit of time. This benchmark enables us to comment on the scalability of the implementations.

6.3.2 Benchmark: Time Consumed for Multiple Epoch

In the training phase of LL0 algorithm, the first epoch is used for training and building the network. The subsequent epochs are identical to each other and they are only used for training the network. The time consumed for the first epoch is stochastic as the network is growing. This randomness will vary between implementations and data sets.

In the subsequent epochs after the first epoch, the stochasticity is reduced as the network stops its expansion. Hence, the average time consumed for the subsequent epochs is a good measurement of performance.

6.4 Profiling Strategy

In this thesis, the profiling strategy is performed by tracing the GPU activities and monitoring memory traffic. The Nvidia tool kit *nvprof*, a command-line profiling tool, and NVCC compiler are used to generate a detailed report for each kernel at the thread level.

To identify the bottlenecks in a mapping of algorithm in GPU, it is essential to keep track of CUDA launch, memory transfer between the CPU & GPU, and also the amount of Application Programming Interface(API) calls. Nvprof and NVP provides detailed information about these traits through the command line and Graphical User Interface (GUI) respectively.

Additionally, detailed thread-level information is analyzed to identify sources of parallelism in the kernel code, which is then modified to increase the bandwidth. Similarly, memory consumption is traced using a visual studio performance profiler by placing the breaking point. This assists in analyzing both the stack and heap memory. Combing results from these tools provide information about memory management.

A secondary profiling strategy is employed to obtain detailed performance metrics by analyzing API calls and Kernel execution. This is crucial to identify the possibilities to enhance the application performance and determine better mapping to GPU. This profiling method helps in reviewing GPU utilization for each section of code.

The following are indebt profiling and analysis strategies employed to fine-tune performance.

6.4.1 API Calls Analysis

CUDA provides API for tasks such as transferring data between CPU and GPU, allocating memory on GPU, internal data movement, and general computations on GPU.

Analyzing the reports generated from *nvprof* helps to determine the number of an API function invocation, the overall percentage of time of execution, and the average time of execution with minimum and maximum time consumed. Monitoring API calls provide vital information about GPU utilization. By optimizing parameters such as the number of API invocations, the timing of API invocation, and the order of API invocation maximizes GPU utilization.

6.4.2 Application Profiling

The GPU performance analysis is carried out using application profiling while considering the GPU idle time, in addition to combining the result of kernel profiling and wrap profiling.

6.4.2.1 Kernel Execution Analysis

In CUDA, a portion of code running on GPU is called a kernel. Further kernel is broken into threads. Threads are single instructions that run individually in the streaming multiprocessor. Managing the threads in the kernel is crucial to capitalize on the parallel processing of the GPU.

In the GPU environment, a large number of threads are created concurrently. It leads to potential race conditions and time-intensive synchronization. Detailed reports of kernel execution are generated using the NVCC compiler to identify and correct this problem.

The compiler manages the hardware resources in GPU and the user is responsible for the mapping of the kernels. Efficient mapping leads to high parallelism and higher GPU utilization. Thus, the emphasis is given to the efficient breakdown of data and associate it with a relevant number of threads, blocks, and grids.

6.4.2.2 Wrap Profiling

The application execution time and the GPU activities are measured at the thread level. These measurements are averaged and provided as the quantitative analysis at the wrap level: Kernel Execution Time, Communicating, Computing, Reading, Writing, Device Synchronization, and Wasted Computation.

The quantitative analysis at the wrap level is described as,

- **Kernel Execution Time:** Overall time of execution of each kernel is determined. Further, the number of times each kernel function is invoked, and the minimum and maximum time taken by each kernel to compute is observed. Also, the percentage of each kernel engaged GPU time is calculated.
- **Idle:** Provides the information about the amount of the time GPU is idle or waiting for another execution to complete is determined.
- **Communicating:** No processing takes place during data transfer between the CPU and the GPU. This time is determined.
- **Reading:** The amount of the time taken to read data from the GPU memory and time spent to load instructions are observed.
- **Writing:** The amount of time taken to write to GPU memory and time spent to store instructions are calculated.
- **Device Synchronization:** Time taken to synchronize threads in the device after kernel function execution is determined. Multiple threads from the same wrap might be executing the same function but have divergent execution paths and the compiler needs to synchronize these threads after completing the execution.
- **Wasted Computation:** Emphasis is given to avoid threads from a wrap to diverge and execute a portion of code segments. The GPU architecture provides conditional execution of instructions which perhaps an efficient approach. The instructions are executed if the boolean variable *predicate* is true, restricting branching and allowing threads in a wrap to execute together. If the predicate is false, the instruction execution time is termed as wasted computation.

7
Results

In this thesis, a customized implementation of the LL0 algorithm was developed. This implementation was found to perform better than the existing solutions and its results are discussed below. A methodology has been followed to achieve higher performance as described in Section 6.1. This methodology was used for identifying the bottlenecks, improve parallelism, and have a better mapping with GPU. The performance difference with PyTorch implementation was compared against benchmarks.

This section further presents the result achieved, which includes the creation of a custom solution for hardware acceleration, verification of the solution, and the performance gain achieved when compared with PyTorch implementation. In this section, the thesis implementation will be referred to as C++ implementation.

7.1 Reflection of Software Architecture

Several challenges were faced while creating the software architecture during this thesis. The challenge was between providing flexibility to capture the dynamic nature of the LL0 algorithm and achieving higher performance, which is described in detail in Section 2.9.

Multiple architectures such as [18][19][56] were studied for concepts of software architecture. After a design space exploration, software architecture was developed, which was inspired by Caffe [59].

The software architecture was confirmed to have better performance than the PyTorch implementation and it provides a platform for deep learning practitioners to explore the LL0 algorithm.

7.2 Verifying the implementation

The PyTorch and C++ implementation are of the same LL0 algorithm. The difference was in the implementation's handling of data and computation paths, which affected the performance but not the output behavior. It is important to note that, verifying the output behavior of the C++ implementation matching with that of PyTorch was a necessary step before comparison of performances.

To ensure the output behavior of the C++ implementation matches with that of the PyTorch implementation, both the implementations were run on the same device which makes it a fair comparison. The device used was, NVIDIA GTX 1060, and information on the hardware architecture can be found in Section 6.2.

To analyze the degree of similarity between the implementations. Both are run with identical datasets and their prediction is captured.

For this thesis, the MNIST training dataset, 60 thousand data, and testing dataset, 10 thousand data, is used. The prediction outputs from the implementations are plotted for 15 epochs shown in Figure 7.1 and 7.2. A high degree of correlation of 0.9660 and 0.9103 is found between the implementations. This confirms that the implementations are similar in their output behaviors.

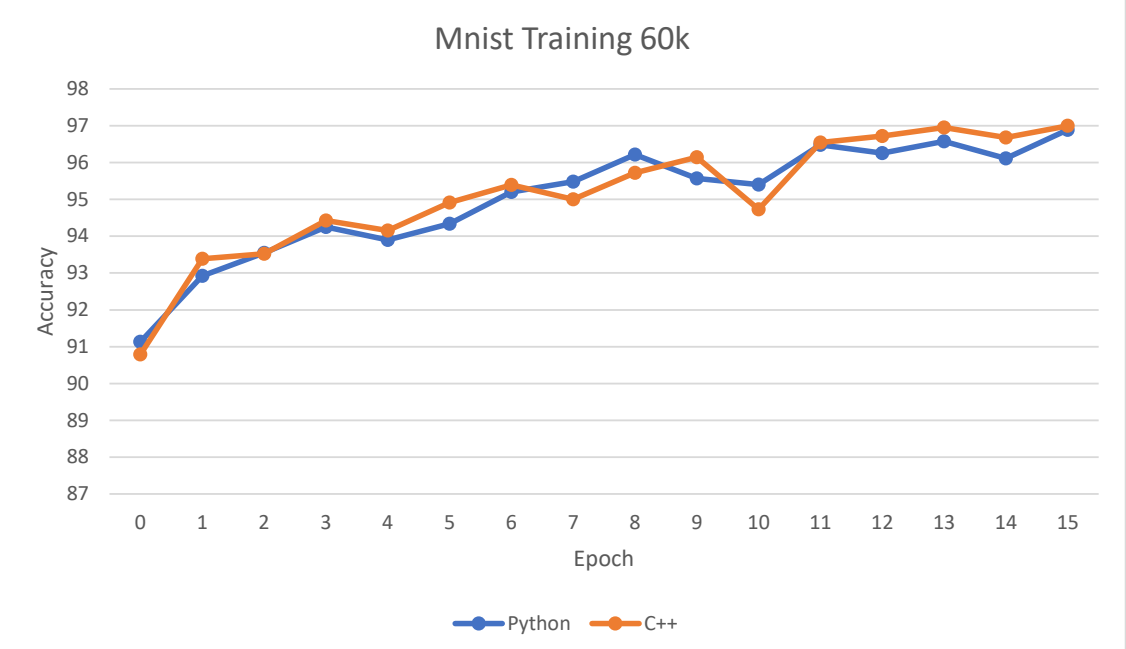

Figure 7.1: Accuracy of C++ and PyTorch implementation on MNIST 60,000 training data and having a correlation of 0.9660.

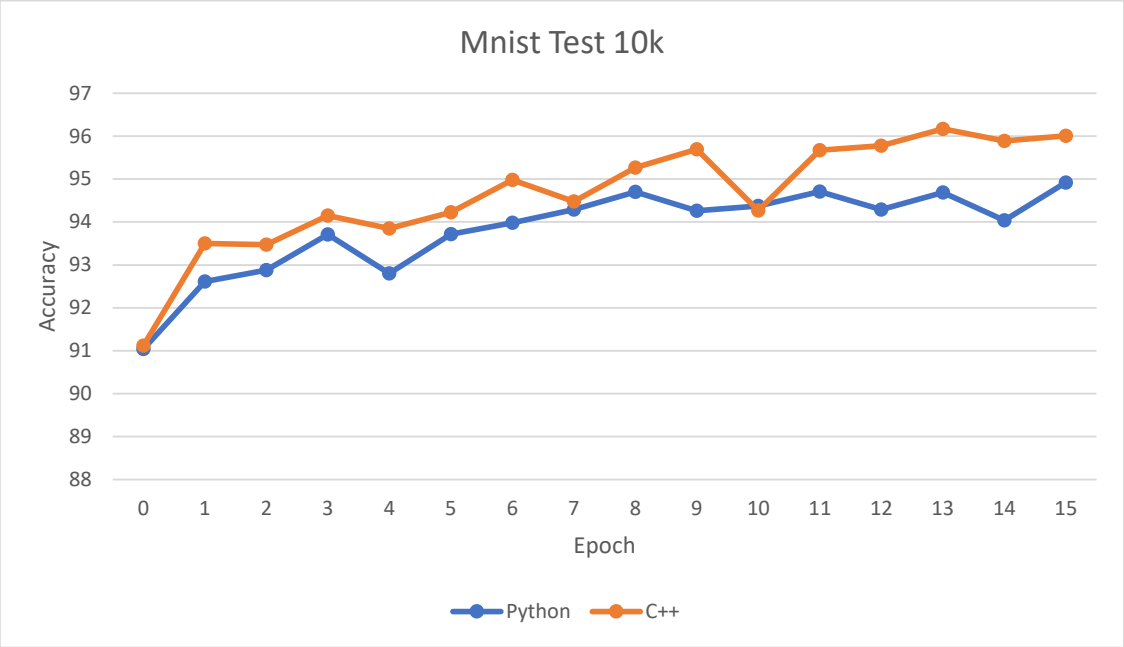

Figure 7.2: Accuracy of C++ and Python implementation on MNIST 10,000 testing data and having a correlation of 0.9103.

7.3 Benchmark of Performance: Number of Feedforward in units of Time

The benchmark was evaluated for the C++ and Python implementation against the MNIST dataset for different batch sizes. The result of the feedforward of 60000 training data points is illustrated in Figure 7.3. Further, the result of 10000 testing data points is depicted in Figure 7.4. Here, batch size refers to the number of data points sent simultaneously/parallel to the system. Since the difference of time is in the order of magnitude, the result is better represented in a logarithmic scale than a linear scale for time.

The C++ implementation consistently outperforms the PyTorch implementation with a maximum speed up of $\times 19.489$ in the training dataset and $\times 17.892$ in the testing dataset as shown in table 7.1 and 7.2. It is also observed from the table that the acceleration decreases when moving from a batch size of 1 to 500 for both implementations.

Mnist Train 60k			
Batch Size	Python Avg(sec)	C++ Avg(sec)	Speed up
1	109.534	5.6202	×19.489
10	11.314	1.3258	×8.534
20	7.424	1.164	×6.378
100	4.8	0.9944	×4.827
200	4.502	1.0264	×4.386
500	4.366	1.0252	×4.259

Table 7.1: Time taken for Mnist 60,000 Feed forwards for different batch sizes by Python and C++ implementation followed by speed up of C++ over the Python implementation

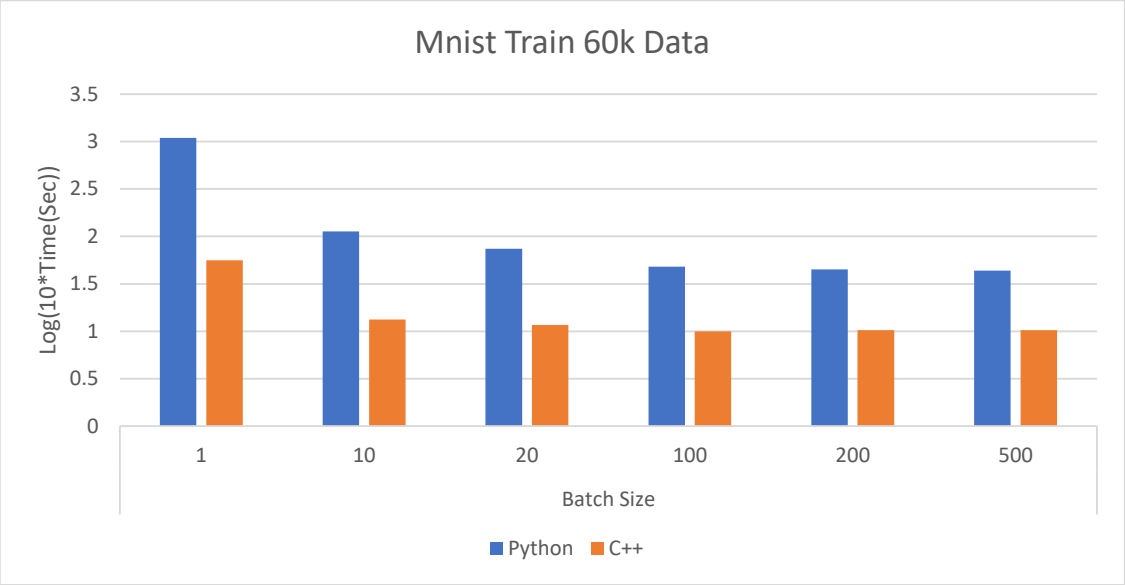

Figure 7.3: The total time consumed, in logarithmic scale, for sending 60,000 training data point of MNIST data set with respect to different batch sizes.

Mnist Test 10k			
Batch Size	Python Avg(sec)	C++ Avg(sec)	Speed up
1	17.824	0.9962	×17.892
10	1.812	0.2364	×7.665
20	1.206	0.208	×5.798
100	0.848	0.1792	×4.732
200	0.77	0.186	×4.140
500	0.742	0.188	×3.947

Table 7.2: Time taken for MNIST 10,000 Feed forwards for different batch sizes by Python and C++ implementation. The last column depicts the speed up of C++ over the Python implementation.

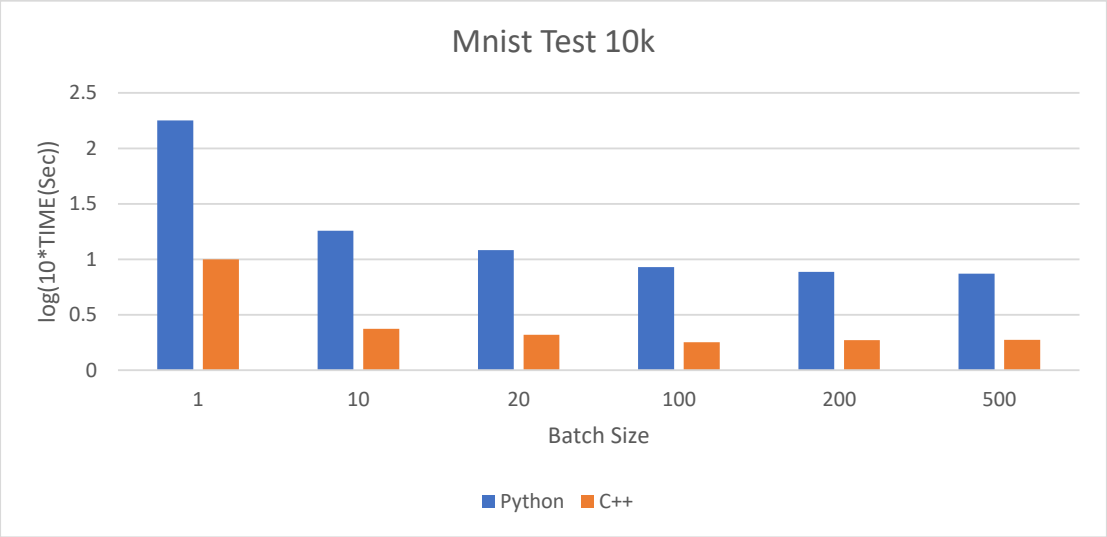

Figure 7.4: The total time consumed, in logarithmic scale, for sending 10,000 testing data point of MNIST data set with respect to different batch sizes.

The performance gain achieved in this thesis is discussed in Section 8.

8

Discussion

In this section, we present the discussion about the results presented in the previous Section 7. In addition, the shortcoming of the system, future work, and suggestions are presented.

8.1 Verifying the implementation

To improve implementation performance, a methodology has been followed to identify the source of bottlenecks and fine-tune implementation to eliminate these bottlenecks. This modification has the potential to alter the functionality of the implementation from the LL0 algorithm. Therefore, it is essential to verify that each revision to the implementation only leads to improved of performance and does not change the functionality of the system.

A two-fold system of checks is performed to verify the implementation. Firstly, a stringent checking of each functionality of the implementation is compared with the LL0 algorithm. The algorithm has a distinct neural network and dynamic capability. This changing real-time network structure makes the manual verification process complex.

Secondly, the output behavior of PyTorch and C++ implementations is compared to check if they are correlated. This method verifies the output behavior of C++ implementation by comparing it with the PyTorch implementation whose functionality is identical to LL0.

In ML algorithm, the output behavior may be similar but not identical because the algorithms are inherently stochastic. To prove that the output behavior of the implementations is similar, both were run with identical data-sets and their respective predictions were noted for several iterations. These predictions were then used to calculate the degree of correlation between the implementations.

The MNIST dataset is used to train the system and the resulting predictions are compared to the PyTorch implementation shown in Figure 7.1 and 7.2. The implementation of the thesis and the PyTorch implementation achieved a quality of correlation with the MNIST dataset as represented in Figure 7.1 and 7.2. The accuracy of C++ closely follows that of the PyTorch implementation and has a high degree of correlation of 0.9103 for 10,000 test data and a correlation of 0.9660 for 60,000 training data. This verifies that the output behavior of the implementation is equivalent.

8.2 Benchmarking of Performance

In this thesis, the benchmarks are used to compare the implementation. It is also used to track the performance improvement when following the methodology. This thesis has defined two benchmarks, the number of feedforward in units of time and total time consumed for multiple epochs.

The benchmarking of performance, number of feedforward in units of time, is a good estimator for overall system performance because the function of feedforward/inference is statistically the most used functionality of machine learning algorithm.

Table 7.1 compares C++ implementation with PyTorch implementation for the number of feedforward in units of time. It demonstrates the number of data points sent to the system and reports the average time to execute 60,000 data points. The output of the number of feedforwards is measured on two different data types namely training, and test data as presented in Figure 7.3 and Figure 7.4 respectively. In both instances, thesis implementation performs significantly better than PyTorch implementation.

The number of feedforward for batch size 1 produced a speedup of up to 19 times and up to 4 times in case of batch size 100 to 500. The increase in speed is primarily due to the exclusion of bottlenecks. The memory throughput of the system is significantly improved by writing custom kernel code. This increase in throughput allowed exploiting the parallelism of the hardware. By analyzing memory access patterns and profiling, the application has achieved an improvement in the bandwidth. Profiling strategies are described in section 6.4.

The Speedup for batch sizes 100, 200, and 500 have resulted in 4 times boost. However, the time consumed decreases only about 70 - 30 milliseconds for batch sizes from 100 to 500 as shown in Table 7.1. There is a significant increase in data transfers when batch size is increased from 100 to 500, but the improvement in time is negligible. This needs additional research as it is a bottleneck and has the potential for gain in future performance. One hypothesis is the system is reaching the theoretical upper limit of the memory bandwidth of the GPU. However, calculation and research are needed before reaching any conclusions.

The benchmark of time consumption for multiple epoch was not evaluated because the development of the performance version of the backpropagation faltered. To get the high performance the differentiation was manually calculated and hardcoded into the implementation. However, this became extremely complex in batch mode. Furthermore, debugging became even more complicated.

PyTorch uses the AutoGrad tool to calculate complex differentiation. Integrating AutoGrad with C++ implementation will require significant effort because AutoGrad requires computation graphs while C++ implementation was not built to support computation graphs for performance reasons.

8.3 Shortcomings of the System

There are three major shortcoming of the systems which are discussed below.

Firstly, the thesis implementation's software architecture only supports Nvidia GPUs. It will require a significant change in software structure to support other hardware architectures. This is the opposite of the PyTorch implementation. Implementation based on the PyTorch framework makes it possible to scale across different underlying hardware architectures including CPUs, GPUs, TPUs(Tensor Processing Units), and future potential hardware without modifying the software database.

Secondly, in order to achieve high performance, the thesis implementation took a trade-off between rapid prototyping vs performance. Every new machine learning dataset that has a unique structure will require custom input and output kernels in C++ implementation. The effects of this problem will diminish with more datasets being supported by the architecture. However, it will not match the agility of drag and drop and rapid prototyping of PyTorch implementation.

Finally, the methodology applied for improving the performance of the implementation involves analyzing and observing the custom written kernel CUDA blocks and data paths using the Nvidia toolchains. This opens up a possibility of performance to be fine-tuned towards the microarchitecture of the underlying GPU. Since the software implementation is written in CUDA API, it is possible to run on another class of Nvidia GPU. However, the performance may change in different microarchitectures and this would require fine-tuning for data parallelism and bandwidth improvement.

8.4 Future Work

There are a few possible enhancements that can be explored. Overcoming the shortcoming of the system is also a potential enhancement. For example, support for multiple microarchitectures. Currently, the system supports only GPU, especially NVIDIA CUDA cores. Since the software architecture is written in CUDA, there is a possibility for supporting advanced hardware such as NVIDIA TESLA cores. These new cores are built for heavy workloads, available for server-grade GPU, and deliver higher performance.

The current system is designed to deliver performance utilizing hardware acceleration. There is an option to convert it into a toolchain which deep learning practitioners can use to study LL0. This would require utilizing the current system as the engine and build tools. Features can be added for example, custom functions for multiple datasets, multiple architectures, and support of multiple GPU and servers.

Furthermore, there is a possibility to have a higher performance system with a GPU-FPGA combination. Currently, in the case of GPUs, the dynamic nature is controlled by the software. However, the dynamic nature can be mapped by partial reconfiguration of the FPGA systems. As LL0 grows rapidly during the initial phase of training and then slows down. This phenomenon can be exploited by GPU and FPGA systems. First, GPU trains the LL0 during the initial phase and after an optimum point hands over to the FPGA for training. It would be interesting to study the performance gains in such systems. The optimum point of handover can be calculated on the cost of partial reconfiguration and the time it takes.

Also it is important to study the system performance and effects under the massive datasets. As currently the system is tested only with datasets with sizes that fit into the GPU memory resulting in faster access to data and decrease in the stall time of hardware. However, massive datasets of size in the order of Terabytes are theoretically supported by the system which open up another dimension of optimization in the fields of CPU-GPU memory space which needs to be researched upon.

Since the thesis implementation was completely written in C++ and CUDA programming paradigm, the software can run in microcontrollers. Nvidia Jetson Xavier and Jetson Nano are some of the Single Computer Boards (SCB) which are ideal candidates to run the LL0. Porting of the LL0 application to the above-mentioned boards is simple as it is written in a model supported by boards. LL0 running on these boards would provide an opportunity to study machine-learning-based solutions for signal processing, IoT data analytics, and many more.

Since the thesis implementation has achieved a boost up in speed compared to PyTorch implementation, a study on real-time processing of data is possible. This is helpful for real-time machine learning-based prediction projects which can be a value add to the industry.

8.4.1 Suggestion

During the development and verification phase of the software architecture, it was observed that the LL0 network would grow very swiftly and reach its limits at the beginning of the training. This phenomenon was not limited to specific datasets and was narrowed down to the condition for growing the network. In the LL0 algorithm, if the network predicts correctly it performs backpropagation, the learning step, else the network grows. The backpropagation step, described in Section 5.4, helps the network remember the input. It is observed that when the network predicts wrong output even with a small margin of error, the network grows. During the initial phase of training, the network predominantly predicts the output with a small margin of error. The accumulation of wrongful predictions forces the network to grow rapidly.

To overcome the rapid expansion of the LL0 structure, an additional step of backpropagation before the condition check for network expansion needs to be carried out. In this step during the training phase, if the network makes a wrong prediction, a backpropagation is performed followed by another prediction. If the prediction is wrong again then the network expands by adding a new node. This additional step of backpropagation before the condition check has shown to stabilize the network expansion.

There is also a possibility to have a high energy efficient implementation if the rules of LL0 algorithm is modified. Currently, in the algorithm, for every wrong prediction, the network expands. For example, if there are 32 wrong predictions, the network expands 32 times. From a hardware perspective, this is a best-case scenario because each streaming multiprocessor supports 32 threads and the above case is a highly parallel and energy-efficient scenario. In contrast, if there are 33 wrong predictions, the network expands 33 times. This is the worst-case scenario from the hardware perspective because two streaming multiprocessors will be required and hardware has to support 64 threads. But out of 64, only 33 are utilized, which is a high energy-inefficient scenario. This thesis suggests tweaking the rule for expansion in LL0 algorithm from a single wrong prediction to 32 wrong predictions. It will improve hardware utilization and save energy. An optimum strategy can also be researched.

9
Conclusion

The thesis aimed to study and analyze the possibility to create a performance version of the LL0 algorithm. Software architecture is built and its performance is improved by following a methodology. The study successfully determined that the thesis version is significantly faster compared to the PyTorch version of the LL0. Maximum acceleration of $\times 17.89$ and $\times 19.48$ for the MNSIT test and training data sets was achieved for a batch size of one respectively, which can be seen in results section in Table 7.1 and Table 7.2.

The benchmark for total time for multiple epochs is not evaluated because the performance version of backpropagation in batch mode was not built. This is due to the complication in the creation of manual differentiation of gradients. There is a possibility to solve this, by integrating the tool AutoGrad into the thesis implementation.

In addition to hardware acceleration, the modular nature of the software architecture in combination with CUDA solves the device portability issue, which supports all NVIDIA GPU with CUDA cores. This allows the implementation to run on different GPUs without modification. However, reasonable fine-tuning will be required for achieving high performance in various GPU microarchitectures.

Bibliography

[1] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[2] A. Lippert, "Nvidia gpu architecture for general purpose computing," 2009.

[3] D. Patterson, "In praise of programming massively parallel processors: A hands-on approach."

[4] X. Lou, "Acceleration of distance-to-default with gpu," Ph.D. dissertation, Master-Thesis, School of Information & Communication Technology Royal . . . , 2012.

[5] J. von Neumann, "First draft of a report on the edvac," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[6] H. Sutter, "The free lunch is over a fundamental turn toward concurrency in software," 2013.

[7] W. Hwu, K. Keutzer, and T. G. Mattson, "The concurrency challenge," *IEEE Design Test of Computers*, vol. 25, no. 4, pp. 312–320, July 2008.

[8] R. R. Schaller, "Moore's law: past, present and future," *IEEE Spectrum*, vol. 34, no. 6, pp. 52–59, June 1997.

[9] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[10] K.-S. Oh and K. Jung, "Gpu implementation of neural networks," *Pattern Recognition*, vol. 37, no. 6, pp. 1311 – 1314, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0031320304000524

[11] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 873–880.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[13] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[14] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural networks : the official journal of the International Neural Network Society*, vol. 113, p. 54—71, May 2019. [Online]. Available: https://doi.org/10.1016/j.neunet.2019.01.012

[15] C. Strannegård, H. Carlström, N. Engsner, F. Mäkeläinen, F. Slottner Seholm, and M. Haghir Chehreghani, "Lifelong learning starting from zero," in *Artificial General Intelligence*, P. Hammer, P. Agrawal, B. Goertzel, and M. Iklé, Eds. Cham: Springer International Publishing, 2019, pp. 188–197.

[16] J. D. Power and B. L. Schlaggar, "Neural plasticity across the lifespan," *Wiley Interdisciplinary Reviews: Developmental Biology*, vol. 6, no. 1, p. e216, 2017.

[17] H. Carlström and F. Slottner Seholm, "Supervised learning with dynamic network architectures," 2019.

[18] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[19] G. Neubig, C. Dyer, Y. Goldberg, A. Matthews, W. Ammar, A. Anastasopoulos, M. Ballesteros, D. Chiang, D. Clothiaux, T. Cohn, K. Duh, M. Faruqui, C. Gan, D. Garrette, Y. Ji, L. Kong, A. Kuncoro, G. Kumar, C. Malaviya, P. Michel, Y. Oda, M. Richardson, N. Saphra, S. Swayamdipta, and P. Yin, "Dynet: The dynamic neural network toolkit," 2017.

[20] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[21] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[22] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[23] H. Ze, A. Senior, and M. Schuster, "Statistical parametric speech synthesis using deep neural networks," in *2013 ieee international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 7962–7966.

[24] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[25] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[26] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of machine learning research*, vol. 12, no. Aug, pp. 2493–2537, 2011.

[27] A. Zheng and A. Casari, *Feature engineering for machine learning: principles and techniques for data scientists*. " O'Reilly Media, Inc.", 2018.

[28] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[29] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson, "One billion word benchmark for measuring progress in statistical language modeling," 2013.

[30] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," 2014.

[31] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 571–582.

[32] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[33] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: A cpu and gpu math compiler in python," in *Proc. 9th Python in Science Conf*, vol. 1, 2010, pp. 3–10.

[34] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2016.

[35] D. Yu, A. Eversole, M. Seltzer, K. Yao, O. Kuchaiev, Y. Zhang, F. Seide, Z. Huang, B. Guenter, H. Wang, J. Droppo, G. Zweig, C. Rossbach, J. Gao, A. Stolcke, J. Currey, M. Slaney, G. Chen, A. Agarwal, C. Basoglu, M. Padmilac, A. Kamenev, V. Ivanov, S. Cypher, H. Parthasarathi, B. Mitra, B. Peng, and X. Huang, "An introduction to computational networks and the computational network toolkit," Tech. Rep. MSR-TR-2014-112, October 2014. [Online]. Available: https://www.microsoft.com/en-us/research/publication/an-introduction-to-computational-networks-and-the-computational-network-toolkit/

[36] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[37] D. Schaa and D. Kaeli, "Exploring the multiple-gpu design space," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–12.

[38] J. Allard and B. Raffin, "A shader-based parallel rendering framework," *VIS 05. IEEE Visualization, 2005.*, pp. 127–134, 2005.

[39] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Computer graphics forum*, vol. 26, no. 1. Wiley Online Library, 2007, pp. 80–113.

[40] Zhe Fan, Feng Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, 2004, pp. 47–47.

[41] U. Drepper, "What every programmer should know about memory," *Red Hat, Inc*, vol. 11, p. 2007, 2007.

[42] C. Nvidia, "Nvidia cuda programming guide (version 1.0)," *NVIDIA: Santa Clara, CA*, 2007.

[43] NVIDIA, "Nvidia cuda compiler," https://en.wikipedia.org/wiki/Nvidia_CUDA_Compiler.

[44] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 73–82.

[45] NVIDIA, "Cuda c++ programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[46] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of gpu architectures," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 185–197.

[47] NVIDIA, "Nvidia profiler user's guide," https://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf.

[48] "NVIDIA Nsight Systems User Guide, author = NVIDIA, note = https://docs.nvidia.com/nsight-systems, note = Accessed: 27-5-2020,."

[49] B. R. Coutinho, G. L. M. Teodoro, R. S. Oliveira, D. O. G. Neto, and R. A. C. Ferreira, "Profiling general purpose gpu applications," in *2009 21st International Symposium on Computer Architecture and High Performance Computing*, 2009, pp. 11–18.

[50] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, pp. 10–14.

[51] B. Dally, "Power, programmability, and granularity: The challenges of exascale computing," in *2011 IEEE International Test Conference*. IEEE, 2011, pp. 12–12.

[52] P. Cortez and A. Morais, "A data mining approach to predict forest fires using meteorological data," 01 2007.

[53] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, Jul. 2015, pp. 1556–1566. [Online]. Available: https://www.aclweb.org/anthology/P15-1150

[54] X. Liang, X. Shen, J. Feng, L. Lin, and S. Yan, "Semantic object parsing with graph lstm," *Lecture Notes in Computer Science*, p. 125–143, 2016. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-46448-0_8

[55] J. Buckman, M. Ballesteros, and C. Dyer, "Transition-based dependency parsing with heuristic backtracking," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing; 2016 Nov 1-5; Austin, Texas, USA. Stroudsburg (USA): Association for Computational Linguistics (ACL); 2016. p. 2313-18.* ACL (Association for Computational Linguistics), 2016.

[56] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a next-generation open source framework for deep learning," in *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, vol. 5, 2015, pp. 1–6.

[57] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural Networks*, 2019.

[58] J. Han and C. Moraga, "The influence of the sigmoid function parameters on the speed of backpropagation learning," in *From Natural to Artificial Neural Computation*, J. Mira and F. Sandoval, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 195–201.

[59] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[60] NVIDIA, "Nvidia gtx 1060 architecture," https://www.nvidia.com/en-us/geforce/news/geforce-gtx-1660-ti-advanced-shaders-streaming-multiprocessor/.