Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

Samyuel Danyo
28/04/2018

FPGA Machine Learning Acceleration

Project supervisor: Professor Mark Zwolinski
Second examiner: Mr. Iain McNally

A project report submitted for the award of
MEng Electrical and Electronics Engineering

Abstract

With the tremendous advance of Machine Learning (ML) in the last ten years, its popularity has peaked. Artificial Intelligence's (AI) advantages over conventional programming are fascinating students, researchers and the big corporations all the same. The largest spotlight is taken by the applications of Convolutional Neural Networks (CNN) in natural speech recognition, computer vision and data classification. These ML methods are seen as a crucial driving force behind a connected smart world or in other words the Internet of Things (IoT).

However, problems arise if an affordable intelligent system needs to be implemented. The high-functioning Graphics Processing Units (GPUs) and Central Processing Units (CPUs), which are currently used for ML computation, are not power-efficient enough.

In this report I explore Field-Programmable Gate Array (FPGA) ML acceleration, using The High-Level Synthesis (HLS) tool - Intel FPGA SDK for OpenCL, as a possible solution to this problem.

Initially a Simple Linear Regression (SLR) model, implemented in multiple variations is studied. I test the performance and accuracy of the FPGA-accelerated system compared to a CPU-based benchmark. Based on the experiment's results I outline the benefits, drawbacks and best development practices for implementing Machine Learning applications in OpenCL for FPGAs.

Finally, I discuss the implementation of a four-layer Neural Network, classifying images of hand-written digits (MNIST dataset). I study the complexities of implementing a Neural Network in OpenCL for FPGA acceleration, the disadvantages of the approach and the possible benefits.

Contents

1. Introduction and Motivation

1.1 Why Machine Learning?

The Machine Learning (ML) field solves problems, too complex to be explicitly-programmed, by creating algorithms, which through the process of trial and error [1] learn models, mapping input data to a desired output [2].

Powerful example of a ML application is in the field of science dealing with how computers can understand images, called computer vision (CV). While for humans it is easy to distinguish a cat in a picture, this task is extremely hard for a computer. There are unlimited combinations of different cats in different surroundings. Even capturing the same cat in the same room, over time produces a vast set of different images. If this problem is approached with conventional programming methods or statistics, every possible picture of all cats needs to be labeled and explicitly coded for a computer to be able to make a decision if in a random image there is a cat or not. However, labeling all pictures is impossible and this is where the greatest advantage of ML comes in. Through a limited set of cat and not-cat images (training set) a ML algorithm is able to detect patterns and produce a model, which we as humans cannot understand but when a random image is presented, the computer is able to gain understanding through it and make a decision.

1.2 Why Machine Learning needs to be accelerated?

One type of ML algorithms is called Neural Networks (NN) [3]. They analyze gigantic amounts of input data, tuning the model on every iteration, until aimed accuracy is achieved [4]. During both: training and processing real, previously unseen, data (inference), NNs need huge amounts of computing power.

Convolutional Neural Networks (CNN) are a type of Deep Neural Networks (DNN) [5]. They are a family of ML algorithms, which is suited for solving some of the most complex problems like image classification (example above), natural speech recognition and cognitive computer functions. They implement long and complex data processing flow, called Deep Learning (DL) [6]. Currently, most of the research and commercial interest is focused on CNNs, because they show the greatest promise for solving extremely complex problems, which can make innovations like smart cities a reality. CNNs are already being used in the most advanced AI systems in: IoT, intelligent assistants, autonomous driving, androids, Fintech software etc.

In general, training and inference of DNNs are carried out on a state-of-the-art General-Purpose (GP) hardware like GPUs and CPUs. One thing is certain - ML usage will continue expanding exponentially, which is the reason why reliable, secure and power-efficient hardware platforms need to be developed simultaneously [7][6].

Current smart technologies, utilizing ML, communicate most of their data to cloud-computing data centers, where most of the processing happens. This results in long latencies and big bandwidth requirements. Nevertheless, some scientists believe that in the coming future, the advance in Artificial Intelligence would require lower latency and hence more local computing. Such data-processing, on the edge of the network, is called Edge Computing [8]. It leverages resources of the devices, connected to the network (for example smartphones, laptops, smart TV and any other smart technology in your home) to process data. [8][9]

I also share this belief, especially considering use-cases such as smart secure systems, AI assistants, autonomous driving or health-related applications. All of these systems need to convey instantaneous decision making, be highly reliable, keep data secure and learn in real time. Doing processing on the edge will lower latency, reduce the communications bandwidth and make systems more reliable as they do not depend on remote connections. Another important advantage is greater privacy and security as most of the data will not leave a certain perimeter (for example your home). Privacy and data handling will increasingly become more important pressing point as AI advances.

Edge Computing delivers another reason for the development of better ML and big-data processing systems. Utilizing heterogeneous computing API frameworks can be the key to efficient edge computing systems (gateways), which will be leveraging the advantages of a symbiosis between different types of processors.

1.3 Accelerator requirements.

Nevertheless, even with the tremendous progress in Processing Units (PU) and the ever-cheaper computing power, the price, as well as, environmental cost of implementing AI systems [10] are too high for mass market.

General Purpose Processing Units (GPPU) have the advantage of flexibility - being able to run various software, which is crucial for the fast innovation cycle of ML. However, that same advantage brings drawbacks, as the cost for being GP is subsystems that are not needed for every application but still draw power and increase latency. Furthermore, there is little visibility into the processing data flow, preventing developers from taking advantage of software specifics. The results are: lower speed and higher power usage. For bringing ML to complex, real-life uses, an extremely power-efficient solution with extended hardware-configuration control is needed. However, it still needs to be flexible, allowing the execution of various software, incorporating the fast innovation in ML.

1.4 Possible solutions.

While in the past, most of the technological innovation has been in hardware, in the last twenty years it has shifted to software, with ML being the brightest example. This is due to physical limitations, as well as, cost. Previously, if a person wanted to upgrade to the next generation of technology, they needed to buy the new, better device. This results in cumbersome hardware innovation cycle, slowing its propagation through society. In contrast, software provides better flexibility, bottlenecks are easier overcome and there not big overhead manufacturing costs or production time. Furthermore, all users can download the new application on their, already owned, device in a day.

Despite all of the points made above, usually, either GP hardware is used or software is designed for a particular hardware. I believe the opposite approach is more logical – designing hardware, customized for a particular application. This can drive faster innovation process, increase power-efficiency and make use of the software's specifications, avoiding unnecessary parts in the system.

One solution to the problem is designing Application-Specific Integrated Circuits (ASIC). However, while they deliver incomparable power-efficiency and data-flow control, ASICs introduce the above-mentioned hardware limitations. Taking into account the rate of innovation in ML, a flexible platform that can be easily updated to the next generation of AI systems is needed. The targeted platform needs to incorporate the advantages of both GPPU and ASICs. That is why, I believe a possible solution could be FPGA acceleration.

FPGAs deliver on both: greater power-efficiency and reconfigurability, giving flexibility to the implementation. Furthermore, with the recent advances in FPGA technology they perform better than ever in floating-point operations and can fit bigger designs. The efficient processing of ML applications requires a lot of data and task parallelism, as well as, vectorization of the data accesses and computation. FPGA architecture is extremely well suited for this, while operating with very high internal memory bandwidth. Last but not least, with the help of Open Computing Language (OpenCL) [11], FPGA implementations, give almost full control over the data flow. This delivers unprecedented opportunities of customization and optimization, allowing the subsystems to complement each other's strengths. Data flow visibility could significantly improve the power-efficiency of complex AI systems.

OpenCL makes hardware design faster by abstracting its low-level complexities, allowing the focus to be on functionality - similarly to software development. It bridges the gap between software and hardware, as well as, between different hardware platforms, making the implementations portable and more powerful. Intel FPGA SDK for OpenCL [12] brings another degree of freedom to the process, by automating optimization and synthesis of hardware and interfacing.

Based on the points made above the objectives of this project are:

- ❖ Implementing a ML algorithm in OpenCL for FPGA acceleration.
- ❖ Studying the OpenCL framework's function in heterogenous system of a host-CPU and a FPGA-accelerator.
- ❖ Testing if FPGA computing can provide a boost to ML applications' performance.

2. Background Theory

2.1 Machine learning

2.1.1 When & why is Machine Learning useful?

ML is a very powerful tool for solving problems, which either cannot/are very hard to be fully defined or simply have a vast number of possible variations of the desired outcome. Examples of such tasks are: face-detection – we as humans do not understand precisely what data in an image suggests there is a face in it; search – there are limitless combinations of strings (including different languages, mistakes, context) which are ever changing as also are the desired destinations (pages, websites, videos, pictures) on the web. These tasks cannot be tackled by conventional coding based on conditions. Consequently, ML is the best approach to solving such problems as it uses a limited set of data, which algorithms use to detect patterns. By making predictions based on their current model, ML algorithms calculate the error compared to the true, expected result and tune the model. This process continues until the useful patterns are picked-up and the model predicts with high accuracy from previously unseen input data.

ML is also used as part of big work-flows as it is able to simplify complex tasks, which are hard and time consuming for humans but easy for computers. It breaks-down large problems, enabling the automation of data-heavy decisions

Besides extremely complex tasks like computer vision, search, speech or cognitive decision making, ML can be extremely useful for seemingly simpler problems, previously tackled with human experience or statistics. Examples would be: probability of a disease based on a number of factors or DNA information; an experiment's output metrics' values based on input metrics' values; the net worth of an individual based on a number of factors; the occurrence of an error or fraud based on data etc.

The benefits of ML are:

1. ***Fitting & generalization***: the model is trained on a limited number of points, which can be placed in a part of the full possible input interval. The model can still generate an accurate representation, because the algorithm captures the patterns and will apply them on previously unseen points, even out of the training interval. This leads to the two main benefits over statistics: training data can be gathered from an interval, where it is easily accessible; ML will better model any interesting (unexpected) results.
2. ***ML transforms discrete data points to a continues model***.
   ML decision making is not based on conditional statements, requiring exact values but on a model, which can take any input value (continuous vector) and produce a continues output prediction. This means the targeted

experiment (decision) does not need to be conducted for all possible input values (all possible pictures labeled or all values of an input metric given a corresponding output). ML algorithms create a model based on the training set and, as mentioned above, generalization & fitting will deliver the best fit decision for all possible input values.

3. ***Automation***. Generalization and continuity mean that human intervention can be fully removed from the decision-making process. Once the system is set up, humans are not needed to make decisions, modify data or in any way facilitate the operation. Furthermore, even data scientists do not understand why particular values are assigned to the models or how they suggest useful patterns. This leads to an autonomous function of ML-based AI systems, where errors, bottlenecks or uncertainty is very low. The system can gather data automatically, learn based on it and make predictions. The more the ML-based systems are operating, the better they become.

2.1.2 Decisions

There are two main types of ML algorithms, based on the types of decisions they make: classification – when the model needs to decide to which class the input belongs; regression – when a continuous relationship between variables is modeled.

Classification

ML classifiers predict to which of a set of categories (classes) a data sample belongs. Based on their training, classifier models derive connections between patterns and classes. During inference, decision is made, depending on how much of these patterns are observed in the particular data sample. In the data space, the separation lines between different classes are called decision boundaries. The decision boundary between class 0 and class 1, is where there is equal (50%) probability that a data sample is either from class 0 or from class 1 [13]. Or in other words it exhibits in equal proportions patterns from class 0 and class 1. An example of a decision boundary between two classes, modeled by a binary classifier, implemented by me in Python can be seen in Fig 2.1.1

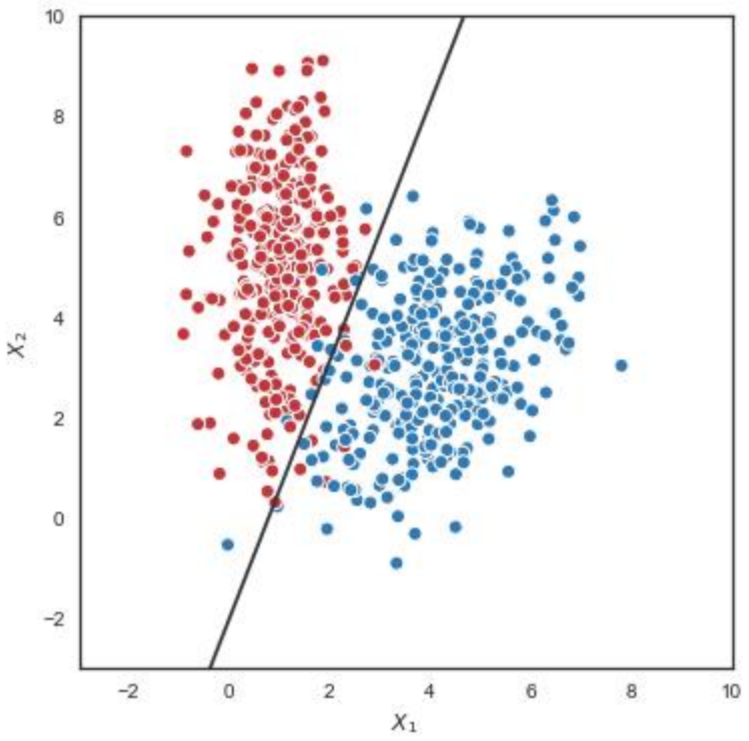


Figure 2.1.1: Decision boundary between two classes, modeled by a ML classifier.

Regression

Regression analysis is a statistical tool, which is used in ML to build models, representing continuous relationship between variables. In contrast with classification, regression modeling searches for a pattern of how inputs and targets map one to another, instead of pairing patterns and classes. Regression algorithms make decisions based on the pattern (function: f()) learnt by the model, where Y = f(X). An example of a Regression model, implemented by me in Python can be seen in Fig 2.1.2.

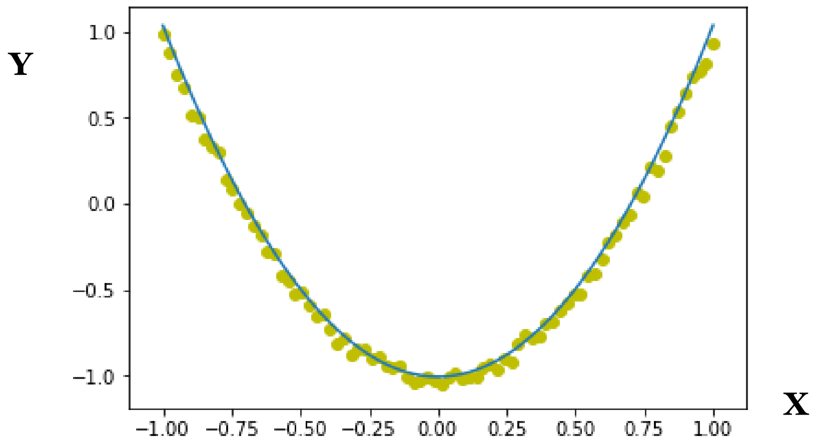


Figure 2.1.2: Regression model (blue line), learnt relationship between X and Y, where the true, expected values are the yellow dots.

2.1.3 Supervised Learning.

Supervised learning algorithms are trained by given example of inputs and their true target labels. In other words, the supervised learning training set is comprised of inputs (X) and their corresponding true targets (Y). The algorithm learns based on searching for patterns, mapping X to Y and tunes the model based on the errors between the prediction (P) and the target (Y) for a particular input (X).

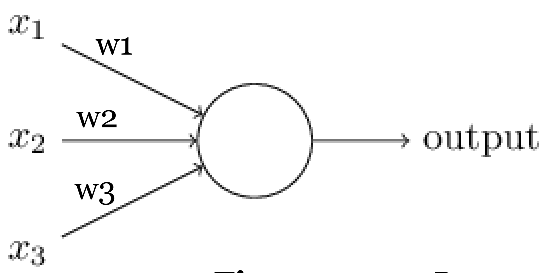
How Supervised Learning works.

1. The input data needs to be chosen. The accuracy of the learning depends on the appropriate choice of algorithm and its specifications in relation to the type of data.
2. Data-set needs to be gathered. Data samples of the input data and their corresponding true targets.
3. The input features representation (Design Matrix) of the input data samples need to be calculated. For example: if the input data is a set of values of a metric, for each (x), the input feature vector could be (1, x, x^2); if the input data is an image, each image can be represented by N pixels with values of [0…255], followed by normalizing them to [0…1].
4. The data-set is divided into three sub-sets: training (70%), validation (15%), test (15%).
5. The ML algorithm and its architecture needs to be chosen.
6. Train the model.
7. The validation set is used to evaluate the model and tune the hyperparameters (architectural and learning) of the algorithm for optimal performance.
8. The test set is used to determine the accuracy of the model on unseen data.

2.1.4 Perceptron

The perceptron is an artificial representation of a neuron, which was invented in 1957 by Frank Rosenblatt [14].

It is a supervised ML algorithm for binary linear classification. In other words, it can learn a single, straight line, decision boundary between two classes. It takes input features (X), assigns them weights ((W), the model) and takes the dot product (X·W), which is called the net activation (net) or the weighted sum of inputs. Finally, the perceptron outputs 1 for net > threshold, 0 otherwise. The full perceptron model can be seen in Fig 2.1.3.



$$output = \begin{cases} 0 \; if \; \sum_j w_j x_j \leq threshold \\ 1 \; if \; \sum_j w_j x_j > threshold \end{cases}$$

Figure 2.1.3: Perceptron Model and Equation. Adapted from [1].

The perceptron learns its model through updating weights in relation to the error of its prediction. The update algorithm is implemented as:

$$w_{ij} \longleftarrow w_{ij} - \eta(y_j - t_j) \cdot x_i \qquad (1)$$

where η is learning rate (how fast the model changes), (y) is prediction, (t) is target.

2.1.5 Activation functions.

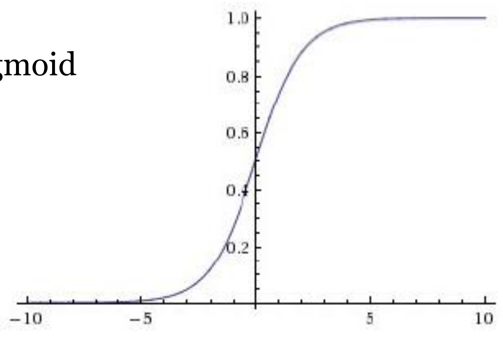
In the context of ML, activation functions [1], transform the net activation into an output by mapping the input data space in a different space, which provides better representation of the information (patterns). Activation functions are usually non-linear, which allows the capturing of more complex decision boundaries. The main activation functions, which usually yield the best results are: Sigmoid (logistic) [Fig 2.1.4:A], ReLU [Fig 2.1.4:B], tanh [Fig 2.1.4:C].

The Softmax activation function (2) is a

> *"generalization of the logistic function that "squashes" a K-dimensional vector z of arbitrary real values to a K-dimensional vector (z) of real values in the range [0,1] that add up to 1" [15].*

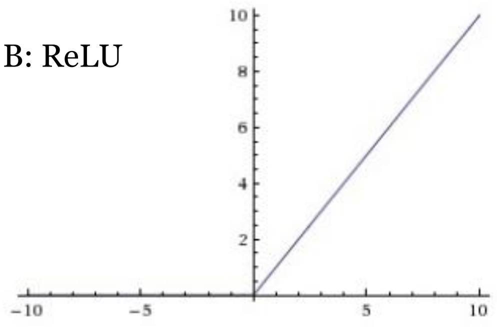
It is a very powerful tool for transforming multiple activations into a probabilistically-weighted representation of the prediction.

A: Sigmoid



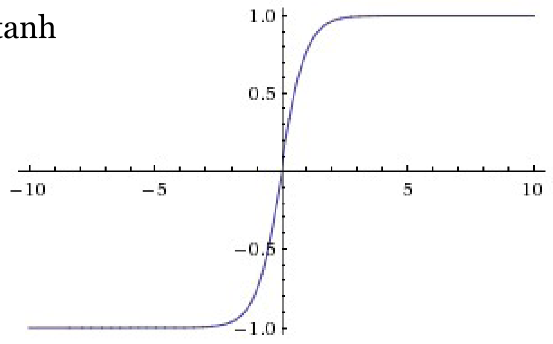
$$\sigma(z) = \frac{1}{1+e^{-z}}$$

B: ReLU



$$RelU(z) = max(0, z)$$

C: tanh



$$tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

Figure 2.1.4: Activation Functions. Adapted from [16].

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_j}} \quad \text{for j = 1, …., K} \tag{2}$$

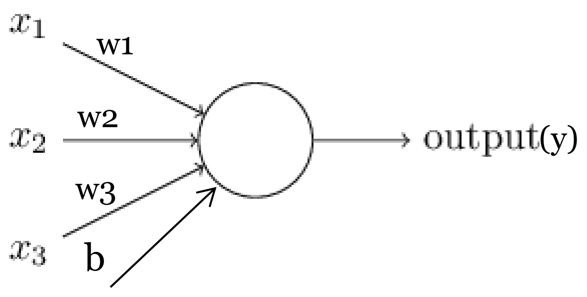
2.1.6 Artificial Neuron

In addition to the above-mentioned interpretation of ML, Tom M. Mitchell defines it in [17] as:

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."

As the name suggests Neural Networks (NN) [3] are comprised of connected artificial neurons. Neurons can be seen as filters, taking input, called features (X), which are being multiplied by numbers, called weights (W), which represent how important each feature is for determining the desired output, and adding a bias (B), controlling the impact of the neuron to the collective NN output.

The artificial neuron [Fig. 2.1.5] is a perceptron with a non-linear activation function and instead of a threshold, a bias is introduced, where (b = -threshold).



$$y = f(\mathbf{w} \cdot \mathbf{x} + b) = f(\sum_j w_j x_j + b)$$

Figure 2.1.5: Artificial Neuron Model and Equation. Adapted from [1].

2.1.7 Neural Networks

NNs are built from layers of parallel neurons. All neurons in a single layer, have the same input features but, they form different decision boundaries due to different weights and bias.

Deep Neural Networks' [Fig 2.1.6] structure contains four or more layers. The layers between the input and the output layer are called hidden. Their task is to extract patterns. In general, the higher the layer, the less neurons it has. The output-layer neurons combine the decision boundaries formed by all previous neurons to derive an accurate prediction.

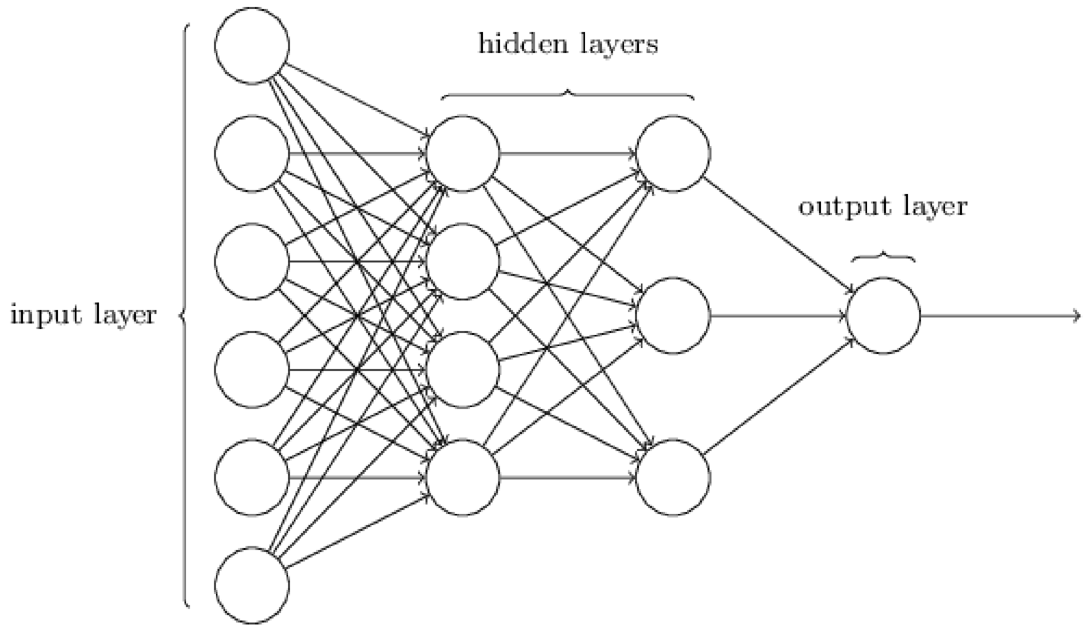


Figure 2.1.6 Deep Neural Network Model. Taken from [1].

Three Layer Neural Network Architecture

The one-hidden-layer NN is a powerful tool, because while yet simple it is able to model any continuous function, given appropriate number of neurons and non-linearities.

In it, the data travels only forward from input to output (feedforward) and each neuron in the hidden and output layers is connected to all neurons from the previous layer (fully-connected).

The input layer simply feeds the input features. The hidden layer applies one of the activation functions to the neurons' nets, usually Sigmoid or ReLU. The output layer has a single neuron for binary classification problems and N neurons for (N > 2) classification problems. In cases of multiple output neurons, Softmax delivers the best performance as an output activation function as it represents every neuron's output as a probability from 100%. An architecture schematic for the three-layer NN can be seen in Fig. 2.1.7.

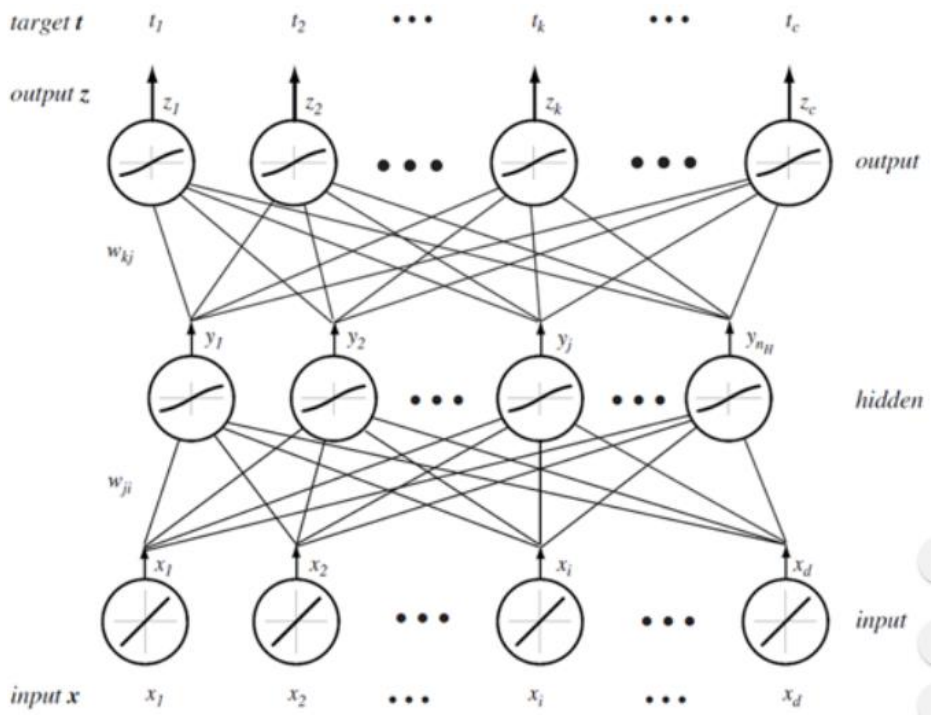


Figure 2.1.7 Three Layer Neural Network Model. Taken from [15].

2.1.8 Model Learning

Gradient Descent

As mentioned above, NNs learn a particular model by going through a great number of iterations with input vectors **X**n and target output vector **t**n. On each iteration a cost function **C**(w,b) (3) is calculated representing the "error" between the prediction and the target:

$$C(w,b) = \frac{1}{2N}\sum_{n=1}^{N} ||\mathbf{y}(\mathbf{x}_n) - \mathbf{t}_n||^2 \tag{3}$$

The cost function is used in an algorithm called Gradient Descent, which updates each weight and bias, in effort to minimize the error. Through many iterations (a full cycle through all training samples is called an epoch) the process aims to converge the cost function to a minimum, changing the model by a certain learning rate (*η*). The Gradient Descent weights and biases update equation:

$$w_k \rightarrow w'_k = w_k - \eta\frac{\partial C(w,b)}{\partial w_k} \quad b_l \rightarrow b'_l = b_l - \eta\frac{\partial C(w,b)}{\partial b_l} \tag{4}$$

Batch Gradient Descent (BGD).

In BGD, the gradients (costs) for the whole training set are calculated (full epoch), a sum or average of them is taken and only then an update to the model is done. BGD is very memory-heavy and slow. However, the learning is smoother and so a minimum error (full convergence) can be achieved.

Stochastic Gradient Descent (SGD).

In SGD, an update to the model is applied after calculating the error of each training sample. This makes it very computational-heavy but much faster. However, the faster (and bigger) changes of the model cause overshooting, which can make converging hard.

Mini Batch Gradient Descent (MBGD).

MBGD takes the best from BGD and SGD. An update is applied after each gradient calculation of a small batch of samples. This reduces the variance of the updates, leading to smoother convergence, while still being fast. Furthermore, it is able to utilize many matrix-manipulation optimizations, making it highly efficient.

Minimizing the Sum of Residuals. Moore–Penrose Pseudo-Inverse.

This approach to learning is specific to Regression. It aims to make the sum of prediction errors equal to zero, mapping the decision into a smooth trajectory, which results into a balanced model solution.

Represented by equation (5), residuals are equal to the difference between target and prediction, or in other words – the prediction error. The prediction is the dot multiplication of input features (design matrix **A**) and model (**w**).

$$\mathbf{r} = \mathbf{y} - \mathbf{A}\hat{\mathbf{w}} \tag{5}$$

Finding a residual vector (**r**), which lies in the nullspace of **A**:

$$\mathbf{A}^T\mathbf{r} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{6}$$

leads to the best-fit solution for the model (**w**):

$$\mathbf{A}^T(\mathbf{y} - \mathbf{A}\hat{\mathbf{w}}) = 0 \implies \mathbf{A}^T\mathbf{A}\hat{\mathbf{w}} = \mathbf{A}^T\mathbf{y} \implies \hat{\mathbf{w}} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{y}. \tag{7}$$

which is using the so-called Moore–Penrose pseudo-inverse of matrix **A:**

$$(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T \tag{8}$$

2.2 Linear Regression

Linear Regression is a statistical approach for mapping independent variables **X, Z...** to a target variable **Y**. It can be looked as creating a model, producing a set **Y** dependent on (**X, Z...**) by an approximation of a function (**Y** = f(**X, Z...**) + noise). When a single input variable **X** is used, the process is called Simple Linear Regression (SLR). Its aim is to predict the values of Y by analyzing previous relationship between **X** and **Y** (training). The accuracy of the model is measured by the resultant residuals. I have called the difference between a CPU prediction and a FPGA prediction, the calculation error, which will track the accuracy of FPGA floating point operations. An example of SLR, implemented by me in Python can be seen in Fig 2.2.1, where the red dots, represent the training targets, the yellow dots are the expected true testing labels and the blue line is the model's decision.

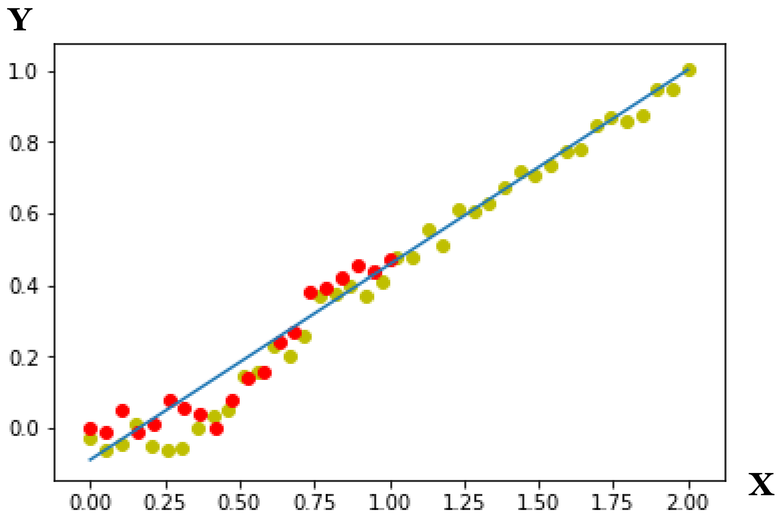


Figure 2.2.1: Simple Linear Regression Example.

2.3 OpenCL

OpenCL is an open framework standard, defined in [12] as:

unified programming model for accelerating algorithms on heterogeneous systems. OpenCL allows the use of a C-based programming language for developing code across ... CPUs, GPUs, DSPs and FPGAs.

Besides its portability, OpenCL's greatest advantage is the unified Application Programming Interface (API), allowing high level abstraction of communication protocols between different platforms.

A diagram of a not-accelerated and OpenCL-accelerated application can be seen in Fig. 2.3.1. It is important to note that the acceleration requires additional processing time for setup and communications, but it also presents the opportunity for a great reduction in execution time of the targeted part of the application.

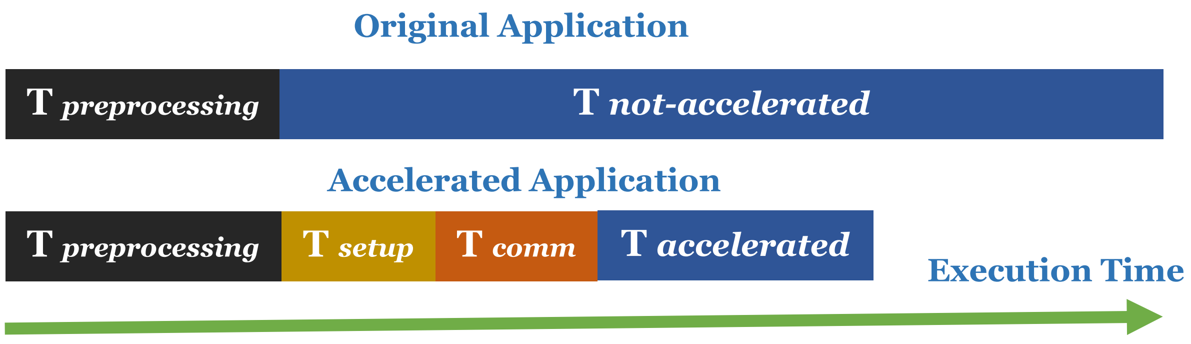


Figure 2.3.1: Comparison of a normal and OpenCL-accelerated application.

2.3.1 Execution Model

OpenCL applications consist of a host program and kernels.

The host program, C++ code with addition OpenCL standard libraries, acts as a controller, responsible for the platform & communications setup, data preprocessing, launching kernels, receiving the results, post-processing and freeing any occupied resources.

The kernels, written in the OpenCL, C99-based code, are parts of the application that are to be accelerated. In this project's example – they are synthesized as real hardware on a FPGA. When relating to a kernel, two different meanings can be inferred. The first one is the physical kernel, written in a separate (.cl) file and synthesized into hardware. The second one is: the API kernel objects, declared in the host program, and attributed the kernel arguments and specifications before launching calculations on the physical kernel hardware.

Context

OpenCL standard object, containing a description of the executing environment: number and type of devices, memories and execution flow.

Program

OpenCL standard object, containing the binary implementation of physical kernels, acting as a dynamic library during execution.

Buffers

OpenCL standard aligned memory objects, used to communicate data between host and kernels.

Command Queue

OpenCL standard object, which defines the order of API-calls execution. It supports three types of commands: buffer transfer, kernel launch and synchronization commands.

Work Item

The basic amount of work that would be executed individually on the acceleration device. Usually references a single execution of a physical kernel.

Kernel Execution Groupings

Multiple work items can be declared in a set, called work group. Each work group has N work items, executing the same code on different data.

Multiple work groups can be declared in a NDRange kernel execution space, where different work-groups are able to execute concurrently. The global size can be measured in total work-groups ($W_{x,y}$) or work items ($G_{x,y}$). Local size is the dimensions of a single work-group in work items ($L_{x,y}$). A schematic of the NDRange, can be seen in Fig. 2.3.2 and the OpenCL execution model in Fig. 2.3.3.

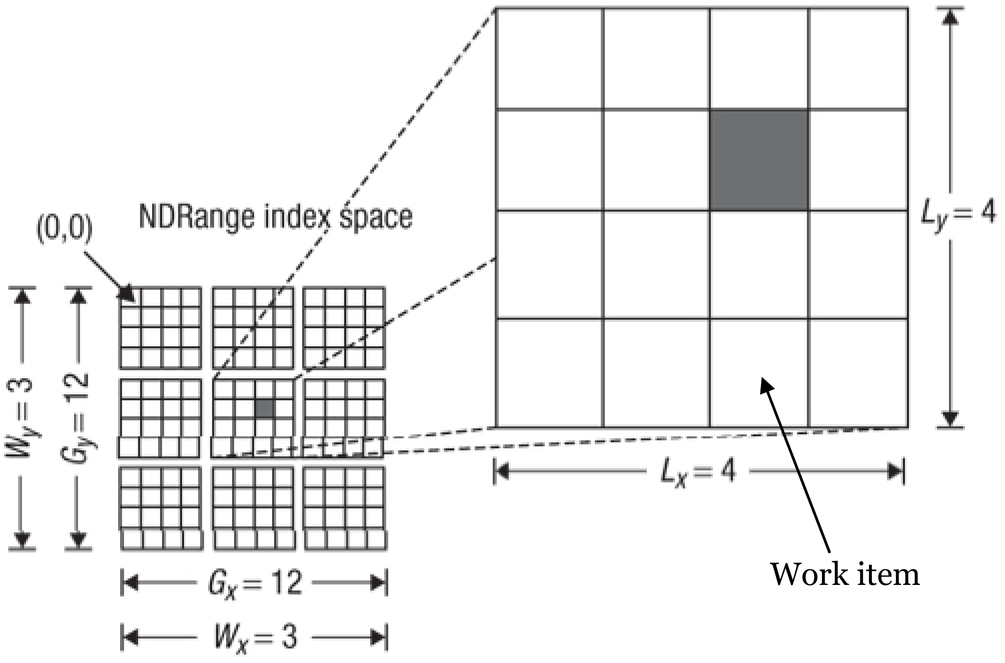


Figure 2.3.2: NDRange index space of work items. Adapted from [18].

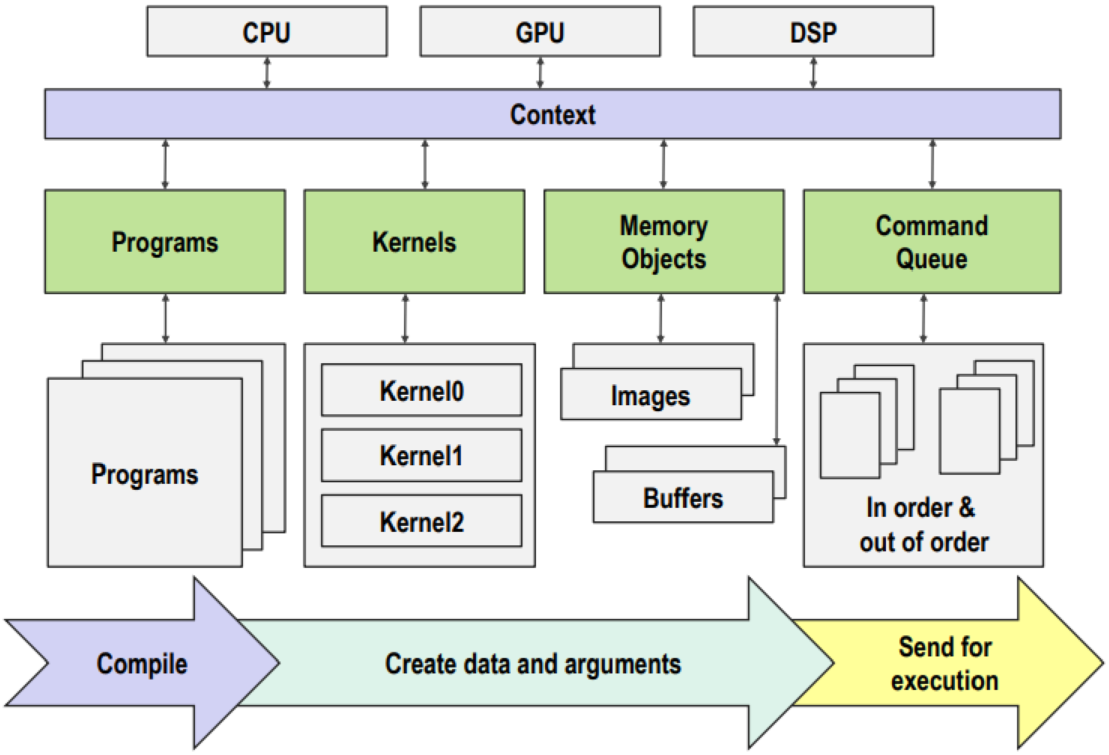


Figure 2.3.3: OpenCL Execution Model Schematic. Taken from [19].

2.3.2 Platform Model

In the OpenCL platform model the platform consists of a host, usually a CPU, and compute devices.

Compute device denotes each PU (FPGA, GPU...) connected to the host, where kernels will execute. Each compute device can process multiple work-groups or tasks in parallel.

A compute unit (CU) is each separate processing hardware on the compute device, which can execute a kernel concurrently (processor cores for GPPUs or hardware kernel for FPGAs). Each CU processes work-groups or tasks sequentially.

Each processing element, also called work item, is an instance of the kernel code. Tasks, as well as, work items, which are part of the same work group, execute sequentially on a single CU and concurrently across multiple CUs.

Schematic of the OpenCL platform model can be seen in Fig. 2.3.4.

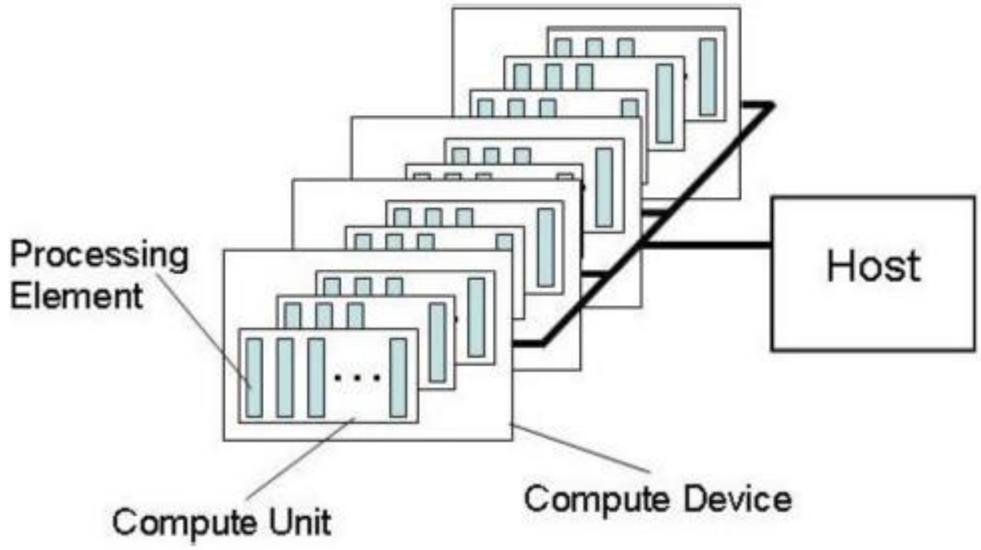


Figure 2.3.4: OpenCL Platform Model. Taken from [20].

2.3.3 Memory Model

The OpenCL memory model defines five memory types [11]:

• **Host memory**: accessible only by the host.

• **Global memory**: permits read/write access to the host and all work-items on a compute device. It is memory on the ***compute device***, which is accessible to the host through the OpenCL API. It is the largest and slowest kernel-accessible memory and can store only 1-D arrays.

• **Constant memory**: region of the global memory that remains constant during kernel execution. It is copied to on-chip cache before launching kernels using it. Due to this, constant memory is much faster, but also ***very limited***!

• **Local memory**: accessible by all work items in a work-group. Mainly used for communication between work items or storage of kernel-calculation data needed by all work items in a single group. Usually, it is embedded on the compute device, delivering low latency and high bandwidth.

• **Private memory**: restricted to a single work-item. Usually, implemented via on-chip registers

Schematic of the OpenCL memory model can be seen in Fig. 2.3.5.

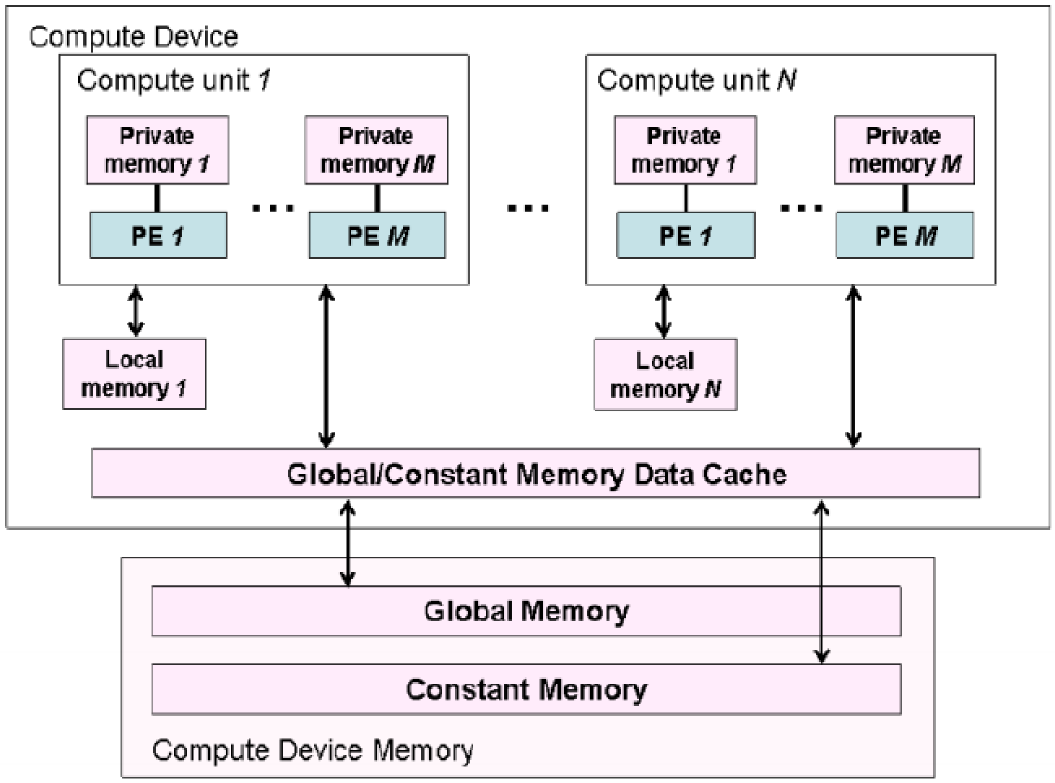


Figure 2.3.5: OpenCL Memory Model. Taken from [20].

2.3.4 Programming Model

The OpenCL standard supports two main types of parallel execution: data parallelism and task parallelism.

Data Parallelism

In data parallelism, the same piece of code is concurrently executed on different data. There are two main types: Single Program Multiple Data (SPMD) and Single Instruction Multiple Data (SIMD). On GPPU with multiple cores, data parallelism between work-groups is by default, while for FPGAs, OpenCL offers a few approaches to accomplish the same. Some of these techniques can be used on GPPUs for better optimization and will be discussed in the "OpenCL Throughput Optimizations for FPGA" section.

Task Parallelism

In task parallelism, different tasks are concurrently executed on data. OpenCL compilers may optimize kernel code to form pipelines, executing different operations in the same clock cycle.

Kernel objects can be launched either as a task (easier task parallelism optimization), consisting of a single work item, or as a NDRange (possible data parallelism) of work items with the corresponding (API) commands:

- **clEnqueueTask()**
- **clEnqueueNDRangeKernel()**

2.4 Intel FPGA SDK for OpenCL

Intel FPGA SDK for OpenCL is a High-Level Synthesis (HLS) tool, which abstracts away the complexities of low level FPGA programming. It optimizes the OpenCL implementation into a highly customized architecture, tailored to the specifics of the system.
Intel FPGA SDK for OpenCL encapsulates the OpenCL framework, additional Intel FPGA libraries, the Altera Offline Compiler (AOC), Quartus and virtual FPGA emulator. Step-by-step guide on how to run OpenCL code on a FPGA using the SDK, as well as, recommendations and solutions to very specific errors can be seen in Appendix 1.

2.4.1 Compilation Flow

The AOC compilation flow can be seen in Fig. 2.4.1. It translates the OpenCL (.cl) kernel code in HDL (Verilog) implementation. During this process the code is analyzed, using the target device's Board Support Package (BSP). From the BSP, the compiler extracts information about the target device's resources, topology and communication protocols. Last, the generated HDL design is mapped to a FPGA image by Quartus. The SDK automatically optimizes the kernel code for a good FPGA performance. Reports are outputted, where resource utilization and design topology can be studied.
The host program is compiled using a GCC toolchain (standard or cross-compiler, based on the host) with supplementary libraries and directives.

2.4.2 Emulation

The SDK also offers emulation capabilities, which allow fast syntax and functionality debugging, compared to the hours-long full compilation. When emulated, the application will functionally behave identically as on a FPGA but any optimizations, run time, accuracy or other device-specific metrics will not be accurate.

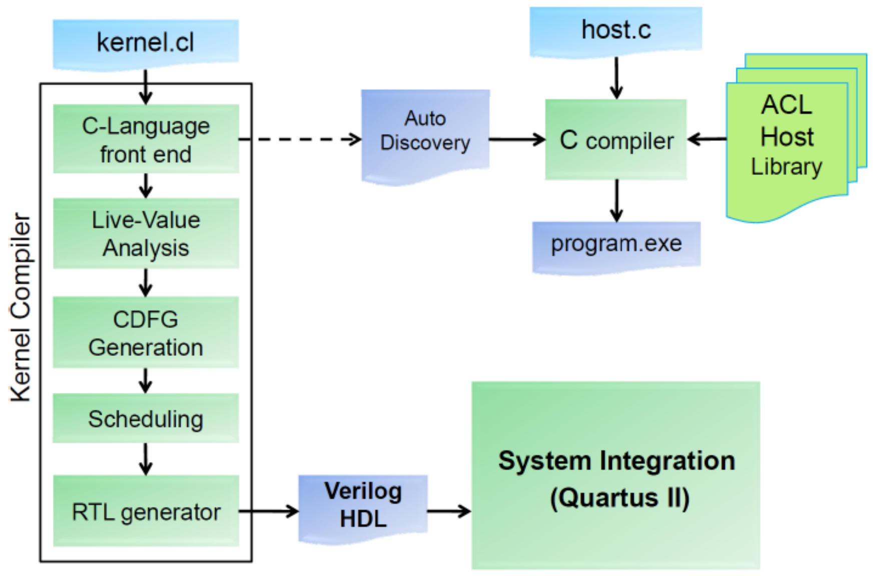


Figure 2.4.1: OpenCL Application Compilation. Taken from [21].

2.5 OpenCL Throughput Optimizations for FPGAs

Besides, data and task parallelism, OpenCL for FPGAs also provides the opportunity for loop parallelism. The AOC can optimize kernel code to implement loop parallelism by synthesizing additional hardware, which allows the execution of next loop iterations, as soon as, any data dependencies are resolved, instead of waiting for a full completion of the current iteration. Loop parallelism schematic can be seen in Fig. 2.5.1, where it is compared with data parallelism. In particular cases optimizing loops can prove more efficient then data parallelism, as an additional work item in the system below, would result in one additional clock cycle for loop parallelism compared to five more for data parallelism.

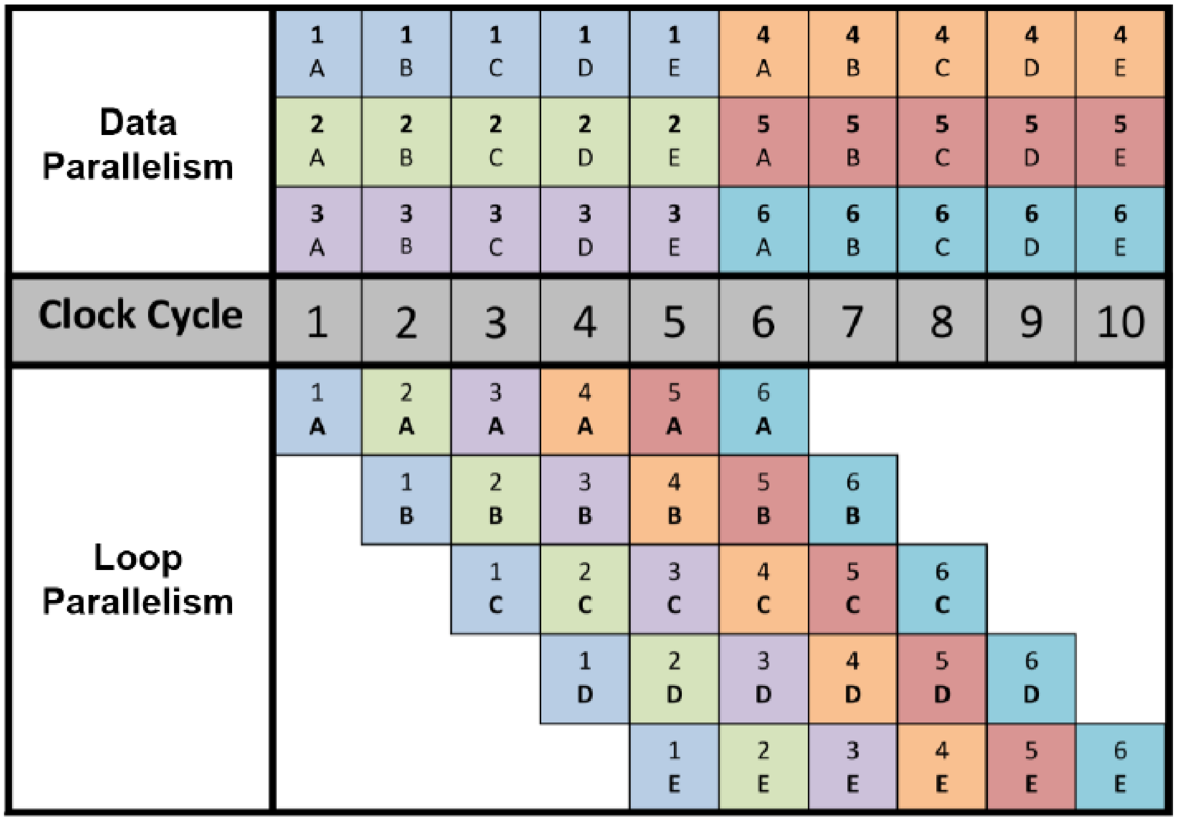


Figure 2.5.1: Comparison of Loop and Data Parallelism. Adapted from [22].

2.5.1 Parallelism Optimizations

Kernel Vectorization

Converts read, write and arithmetic operations from scalar type to a SIMD mode. Executions will target a chunk (vector) of data instead of single addresses. Kernel Vectorization also ensures contiguous memory access pattern and reduces the number of memory accesses, improving memory usage efficiency. Comparison between SIMD and Compute Unit Replication FPGA kernel implementations can be seen in Fig. 2.5.2.

- ***Automatic***
  Declared with <*num_simd_work_teims(2^n)*> and <*reqd_work_group_size((2^n)*a)*> *kernel attributes*. SIMD size must be a power of two and the work group size needs to be divisible by it.
- ***Manual***
  Manual vectorization can be achieved by using vector accesses: <*ARRAY[G_idx*4+(0…3)]*>. This approach would require ¼ of the original work items.
- ***Vector Data Types***
  The same effect can be achieved by using vector data types like: <*float4*>.
- ***Vectorization Drawbacks***
  Bigger memory bandwidth is required, as well as, more data per cycle. Some functions may be left not parallelized, the performance benefit gets saturated with increasing SIMD size as the system gets memory-limitations bound.

Loop Unrolling

Implemented by the compiler if <*#pragma unroll*> or < *__attribute__((opencl_unroll_hint))*> is used before the loop. The compiler optimizes the hardware to use loop parallelism.

- ***Drawbacks***
  A lot of hardware resource are needed and dramatically increases compilation time.

Compute Unit Replication

Data parallelism can be achieved by making multiple copies of a kernel pipeline. Compute unit replication is declared using <*num_compute_units(N)*> kernel attribute. It is important to note that compute units will be utilized only if multiple work groups are declared. The scheduler will dispatch a work group to the first available compute unit. Comparison between SIMD and Compute Unit Replication FPGA kernel implementations can be seen in Fig. 2.5.2.

- ***Drawbacks***
  Linearly increases the amount of hardware resources needed. Furthermore, it requires more resource than SIMD, as control logic blocks will also be replicated. Multiple CUs share common global memory bandwidth, which can cause memory access contention. As CUs are replicas, they may access the same memory locations, using the same global communication paths, which causes bottlenecks.

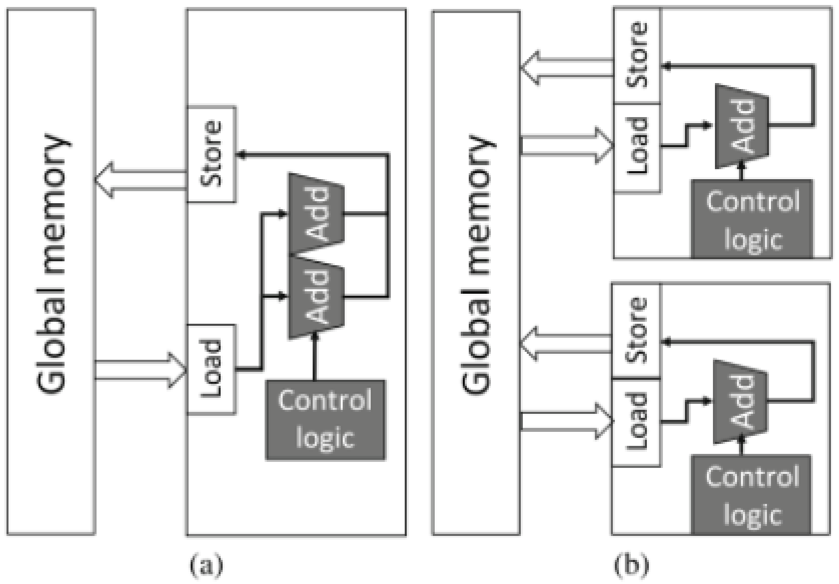


Figure 2.5.2: Comparison of SIMD (a) and CU Replication (b). Taken from [23].

2.5.2 Communication Optimizations

OpenCL applications' performance is highly dependable on communications, where low rate of data, contention or bottlenecks can hinder speed-ups.

Memory Alignment

Enables Direct Memory Access (DMA) transfer between host and FPGA, which significantly improves communication efficiency. The optimization can be implemented by declaring aligned data objects, using <*posix_memalign*> in Linux or <*aligned_malloc*> function in Windows.

Local Memory Caching

Use of local memory can improve kernel performance by reducing global memory accesses.

Coalescing Memory Access

Utilizes SIMD read and writes. As explained above, vectorization can deliver great performance improvement.

Channels

GPPU OpenCL implementations require global memory accesses for communication between different kernels (pipelining multiple physical kernels), which is slow and can cause pipeline stalls. AOC offers a better solution for FPGAs, where channels between kernels are implemented using

FIFO buffers. Channels facilitate kernel parallelism (similar to loop parallelism), where kernels can run concurrently with the "consumer" kernel processing intermediate data, as soon as, it is available. Channels also act as a synchronization between kernels, as the "consumer" will wait for the "producer" to send the right data, something which needs more complex implementation, using global memory. Diagram of a kernel pipeline, utilizing channels, can be seen in Fig. 2.5.3.

- ***Drawbacks***

  The implementation cannot use automatic SIMD vectorization, as the required arbitration is not supported as of writing this report. However, manual vectorization can be implemented. Caution needs to exercised, if CU replication is used!
- ***Warnings***
  - "Producer" and "consumer" kernels need to respectively write and read the channel equal times or deadlock will occur.
  - In order to prevent a deadlock, kernel objects need to be submitted in different queues from the host or the kernels will execute sequentially and so deadlock.
  - Multiple kernels cannot read or write the same channel simultaneously.
  - Channels should be read/write-only from the same kernel. Kernels reading and writing to the same channel will have bad performance.

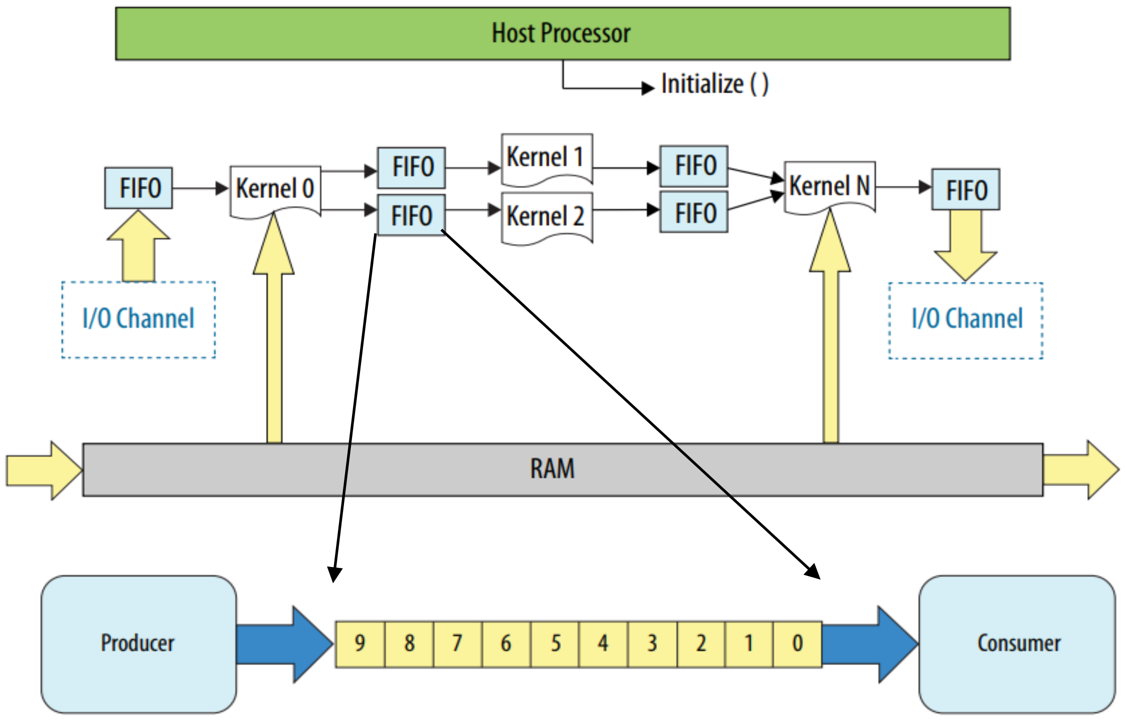


Figure 2.5.3: FPGA kernel pipeline using channels. Adapted from [24].

3. Related Work

While the Khronos Group has done a good job on documentation [11], there is not a clear, practical guide to the different optimization approaches for FPGAs and their impact on performance. Furthermore, Intel FPGA have outlined a number of optimizations and examples in their guides [24] [25], but once more there is not a unified and simple guide to data and task parallelism, as well as, memory usage and communications. Best practices and implementation recommendations need be derived, based on a study of the above-mentioned specifications' impact on performance.

In [26] insightful overview on the specifics, opportunities and advantages of FPGA architecture compared to GPUs is provided. The proposed optimizations of the kernel code delivered an average of 5.3 times speed-up compared to non-optimized code. However, while the work on kernel data flow is impressive, host-based context and data-transfer problems are not explored.

Wang et al present the DLAU – a scalable Deep Learning Accelerator Unit In [27], innovative architecture, implemented on an FPGA, customized for ML, which tackles the problems of the huge size of DNN, their evolving nature and power consumption, achieving up to 36 times speed-up with roughly the same power. However, OpenCL, its interface and data flow control capabilities are not studied. Furthermore, there is not insight on the way different specifications impact the performance, which is crucial for the fast-evolving and diverse field of ML.

In [28] a CNN inference implementation, using Intel FPGA SDK for OpenCL is presented. The dissertation gives detailed analysis of FPGA parallelism, as well as, data communication. ML based optimization of kernel code and affirmation of the power-efficiency benefits of such design are shown. However, it is lacking an in-depth study on host-level interfacing parallelism, especially in a system running multiple applications, which is extremely important for AI systems.

4. System Implementation and Experiment

I implemented a SLR algorithm and a four-layer NN in Python and ran it using Anaconda and Jupyter Notebook. I used these implementations to train models, which I later used for testing.

My C++ programs were compiled with GCC for emulation (executed on x86) and arm-linux-gnueabihf-gcc cross-compiler, part of the Intel SoC FPGA Embedded Development Suite v16.1, for testing (embedded ARM).

OpenCL kernel code was compiled using Intel FPGA SDK for OpenCL v16.1. and DE1-SoC OpenCL BSP V1.1.

Testing was done on a DE1-SOC board: CPU: Dual-core ARM Cortex-A9, 1GB DDR3 SDRAM; FPGA: Cyclone V SoC 5CSEMA5F31C6, 64MB SDRAM;

Comparison in timings were done for T not-accelerated (benchmark) vs T setup +T comm + T accelerated [Fig. 2.3.1]. The timings were taken in three metrics: real elapsed time, CPU processing time and CPU processing time in clock cycles.

4.1 Simple Linear Regression

4.1.1 Why I chose to study SLR inference?

SLR is the simplest ML algorithm. This allowed me to focus my efforts on in-depth study of the OpenCL framework and how it can be utilized for ML applications instead of spending most of my time on functional implementation of the algorithm.

The whole SLR algorithm is using matrix multiplications, which is the best candidate for accelerating on FPGAs. Furthermore, matrix multiplication is the basis of all ML algorithms and the most computationally-heavy part for most of them. Through studying SLR, I am able to deduct best practices and get performance measurements, which are relevant to the whole ML field.

I chose to implement inference rather than the whole algorithm, because inference is the most viable candidate to initially move from cloud to edge computing. Inference is based on matrix multiplication and it is similar to all ML algorithms.

4.1.2 Simple Linear Regression Python implementation.

I started by writing a full SLR implementation, where I set the problem, create a training set, learn the model and validate it. The full code with explanations of every step is given in Appendix 2.1. The problem and implementation results can be seen in Fig 4.1.

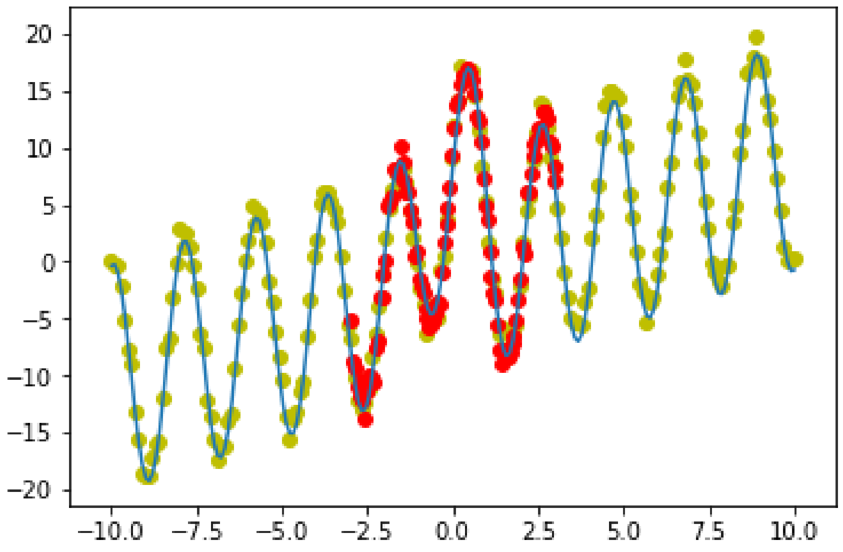


Figure 4.1: Python SLR implementation results. Prediction in BLUE, true labels in YELLOW, training targets in RED.

The Python implementation trains the model, which I later use for the OpenCL inference implementation. The part of SLR (features initialization & inference), which I will be accelerating on a FPGA, can be seen in Listing 4.1.

```
XptsPed;     # Size of the inference set
XlinPred;    # Inference interval
W;           # The model (weights & bias)
# Initializing the inference features (inference Design Matrix)
APred = np.stack((np.ones(XptsPred), XlinPred, np.sin(3*XlinPred),
        np.exp(-1*XlinPred**2))).T
# Doing the inference (making a prediction, based on the model)
Ypred = APred.dot(W)
```

Listing 4.1: SLR features initialization and inference in Python code.

4.1.3 Simple Linear Regression C++ implementation

The second stage of the study was to implement the SLR initialization and inference in C++, which serves as a benchmark, against which the OpenCL system's performance will be compared. The code, which will be accelerated on a FPGA, can be seen in Listing 4.2. The full run() function can be seen in Appendix 2.2

```
N;  // Size of the inference set
weights; // The model (weights & bias)
// Initializing the inference features (inference Design Matrix)
for(unsigned j = 0; j < N; ++j) {
      features[4 * j] = 1;
      features[4 * j + 1] = input_X[j];
      features[4 * j + 2] = sin(3 * input_X[j]);
      features[4 * j + 3] = exp(-1 * input_X[j] * input_X[j]);
      // Doing the inference (making a prediction, based on the model)
 prediction_Y[j] = features[4 * j] * weights[0]  +
                   features[4 * j + 1] * weights[1] +
                   features[4 * j + 2] * weights[2] +
                   features[4 * j + 3] * weights[3];
}
```

Listing 4.2: SLR features initialization and inference C++ code.

4.1.4 Simple Linear Regression OpenCL implementation

I started with an implementation, which accelerates both feature initialization and inference. It is comprised of two parts: the host program, designed to run on a CPU, is acting as a controller – initializing the OpenCL API, preprocessing data, sending the input and receiving the output, the host program's run() function can be seen in Appendix 2.3; the kernel, synthesized as hardware on the FPGA, acts as the accelerator - initializing the features, executing the inference and sending back the result [Listing 4.3].

```
__kernel void simple_linear_regression(__constant float *restrict x,
                                     __global float *restrict y) {
      // The model (weights & bias)
      __local float weights[4];
      weights[0] = -0.55395846;
      weights[1] = 1;
      weights[2] = 10;
      weights[3] = 8.94836441;

      // Get index of the work item. Which data point is processed.
      int index = get_global_id(0);

      // Initializing the inference features
      __local float features[4];
      features[0] = 1;
      features[1] = x[index];
      features[2] = sin(3 * x[index]);
      features[3] = exp(-1 * x[index] * x[index]);

      // Inference (making a prediction, based on the model
      y[index] = features[0] * weights [0] +
                 features[1] * weights [1] +
                 features[2] * weights [2] +
                 features[3] * weights [3];
}
```

Listing 4.1: SLR features initialization and inference OpenCL kernel

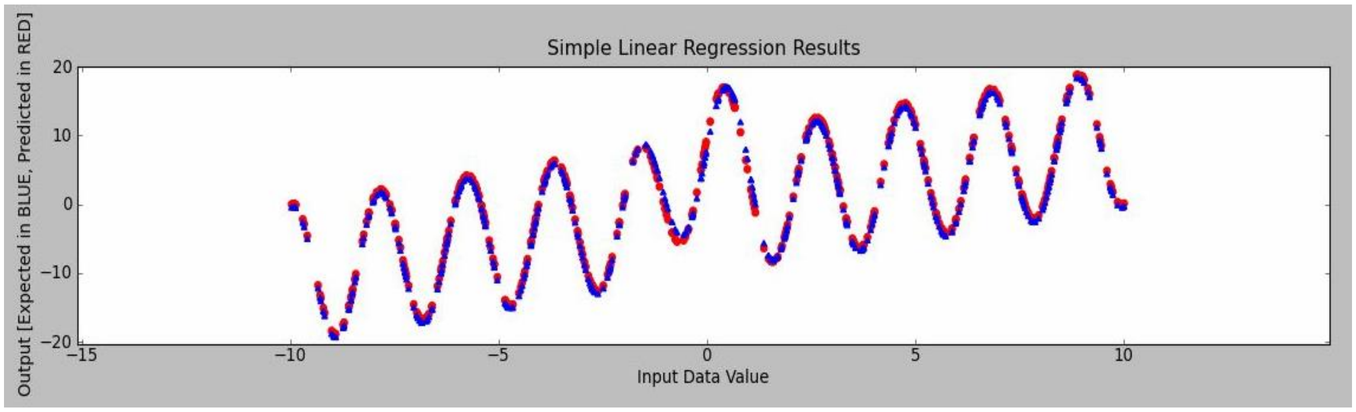


Figure 4.2: OpenCL SLR inference results. Prediction (RED) vs True Values

For a better representation of the accuracy of my implementation and evaluation of the FPGA floating-point calculations, I included verification functionality. The CPU calculates the expected prediction results, as well as, the expected truthful labels and compares them to the kernel output. Any error bigger than a set threshold is logged in a file. All input and output data, expected values, residuals and errors are written in a (.dat) file. As an extension I wrote a Python script [Appendix 2.4], reading the data file and producing plots of the results [Fig. 4.2], errors [Fig. 4.3].

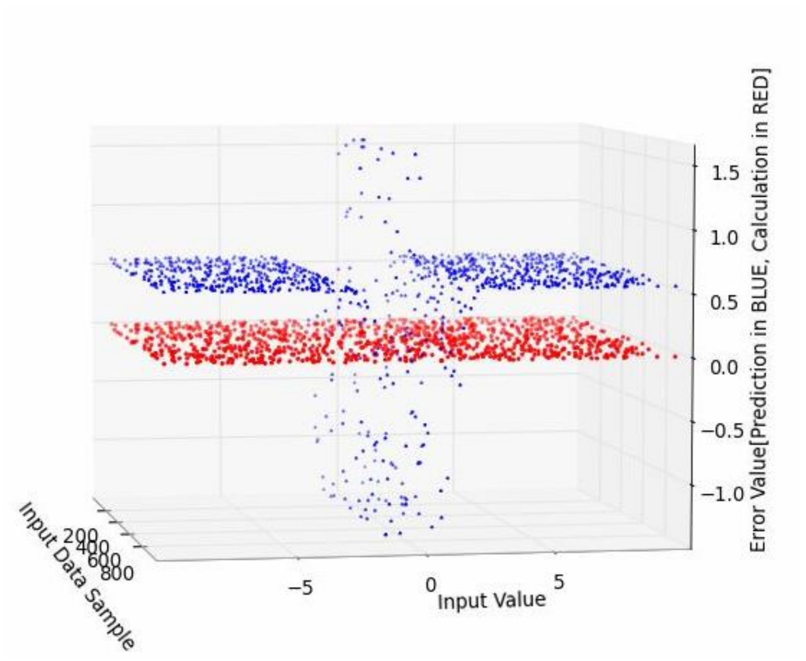


Figure 4.3: OpenCL SLR inference results. Errors (RED), Residuals

Specifications study

In effort to devise best practices for OpenCL implementation of ML applications, based on targets as performance or logic complexity and size, I have developed multiple variations of the original SLR implementation [Appendix 2.5].

While memory alignment is used in all implementations, different possible optimizations are explored. The variations are used for studying the execution-time and logic size (linked to power consumption) impact of: communications, types of operations, types of memory, as well as, C++ and OpenCL optimizations. The full implementations can be seen in the included design archive [Appendix 6].

1. **Study of FPGA Memory**
   SLR_glb studies the effect of using <__*global*> memory instead of <__*local*>.

2. Study of OpenCL communications and FPGA computation.

The two variations explore the performance impact of data transmission, local memory caching, FPGA computation and transfer bottlenecks.

- ***Partial local memory caching.***
  In the first (SLR_w), both - the input data and weights are transferred from the host to the FPGA.
- ***Reduced FPGA computation.***
  In the second (SLR_f) - the features are calculated on the host. Weights and features are sent, leaving only the matrix multiplication (inference) to the FPGA.

3. Study of host-controlled parallelism.

As mentioned in the introduction, the goal of this project is to not only study the possible performance efficiency improvement, but also the flexibility of data-flow control with focus on parallelism. I developed six variations of the original SLR OpenCL implementation, grouped by pairs, exploring host-side parallelism.

- ***Kernel parallelism. Logic & memory allocation.***
  The first pair studies the impact of adding < *reqd_work_group_size*()> kernel attribute in addition to using <*local_work_size*> when launching a kernel with <*clEnqueueNDRangeKernel()*>. I explore if multiple concurrent kernel executions will be launched and the effects on performance.
  - In SLR_g instead of sending an NDRange kernel with a NULL argument for local workgroup size, a size of <global_work_size / 10> is declared.
  - SLR_k_g differs only in its kernel code by a declaration of <*reqd_work_group_size*(global_work_size / 10, 1, 1)>.
- ***Host communications & queues scheduling.***
  With the second pair I aim to study what is the effect of sending NDRange kernels from two different queues on kernel parallelism. I would like to find out if kernel-level parallelism can be achieved with a single kernel implementation. Also, if two different physical kernels execute concurrently from different queues.
  - In the first variation (SLR_q), the original SLR kernel is submitted in two queues.
  - In SLR_k_q, I added a copy of the physical FPGA kernel and submitted them in two different queues.
- ***Host communications & threads scheduling.***
  In the last pair, I explore the effect of sending NDRange kernels from two different host threads.

- In the first variation (SLR_t) a single FPGA kernel is launched from two different threads.
- In SLR_k_t – an additional physical FPGA kernel (copy) is declared, sending both from different threads.

4. **Study of work-group size.**
   SLR_k_g_b studies the effect of using reduced work-group size. Similarly, to SLR_k_g, the system uses two physical kernels with optimized hardware for the <*reqd_work_group_size*>.
5. **Study of FPGA-controlled data parallelism (FPGA Performance Optimization).**

In effort to investigate how data parallelism is handled on the FPGA side, facilitated by the OpenCL standard, I created four additional variations of the SLR implementation. All of them use a single queue & thread, with defined <local_work_size> and <reqd_work_group_size>, transferring input only. I aim to study the efficiency of the kernel optimizations in the form speed boost and logic size, as well as, limitations.

❖ ***Data parallelism via Coalescing Memory Access (SIMD).***
SLR_k_g_SIMD utilizes the <num_of_simd_work_items(4)> kernel attribute, which enforces automatic vectorization of scalar operations like loading/storing/addition and others. I study its impact on performance, in combination with partial task parallelism (prompted by the defined work-group size) optimized by the AOC.

❖ ***Data parallelism via Compute Unit Replication.***
SLR_k_g_CU utilizes the <num_compute_units(4)> kernel attribute, which urges the AOC to create 4 replications of the kernel pipeline. This in combination with <reqd_work_group_size> enforces data parallelism between work groups, with each, processing in a separate CU. I study the impact of full data parallelism on performance and logic size, compared to SIMD, with focus on possible bottlenecks and control logic delays.

❖ ***Data parallelism via mixed approach (SIMD & CU Replication).***
SLR_k_g_SIMD_CU utilizes <num_of_simd_work_items(2)> & <num_compute_units(2)> kernel attributes. I study the tradeoff between full parallelism and better kernel optimization and its impact on performance. I seek the best balance between SIMD (reduced control logic & concurrent data access) and pipeline replication.

❖ ***Data parallelism via vector data types.***
SLR_k_g_V_D utilizes < float4> vector data type. Its use allows the operation on a vector of four data addresses concurrently. I study its impact on performance, especially in comparison with automatic SIMD.

4.2 Four-Layer Neural Network: Classifying Hand-Written Digits (MNIST)

The Modified National Institute of Standards and Technology (MNIST) database [29] consists of square 28x28 pixels images of handwritten digits [Fig. 4.2.1]. The data set is 70 000 samples in total, which I divided into: [42 000:14 000:14 000] subsets for training, validation and testing, respectively. I chose the MNIST dataset, because it presents me with the opportunity of implementing a more complex, real-life system, while still not making it too convoluted on the ML side, which lets me study OpenCL.

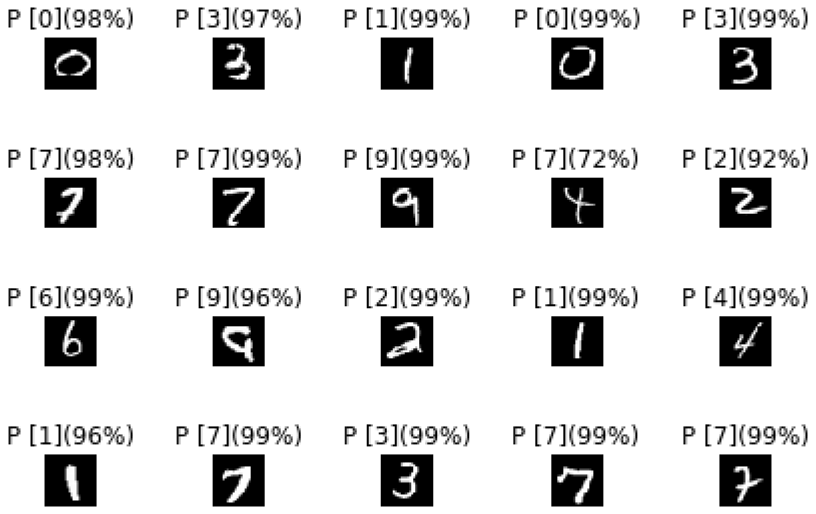


Figure 4.2.1: MNIST hand-written digits.

My Four-Layer Neural Network MNIST Classifier Architecture

- Input layer – 784 (28x28) input features with values [0-255].
- Hidden Layer 1 – 500 neurons, ReLU activation function.
- Hidden Layer 2 – 150 neurons, Sigmoid activation function.
- Output Layer – 10 neurons, Softmax activation function.
- Feedforward and fully connected.

For training, I used Mini-Batch Gradient Descent with batch size of 500 and learning rate of 0.1.

4.2.1 Four-Layer NN Python Implementation.

I used the Python implementation for training the model, which I later used for the OpenCL study. The model took 63 epochs to converge, with test accuracy of [96.67%] and average test certainty of [96.74%]. The cost function convergence can be seen in Fig. 4.2.2. Increase in accuracy over training epochs can be seen in Fig. 4.2.3, certainty [Fig. 4.2.4]. The confusion table of accurate and erroneous predictions can be seen in Fig. 4.2.5.

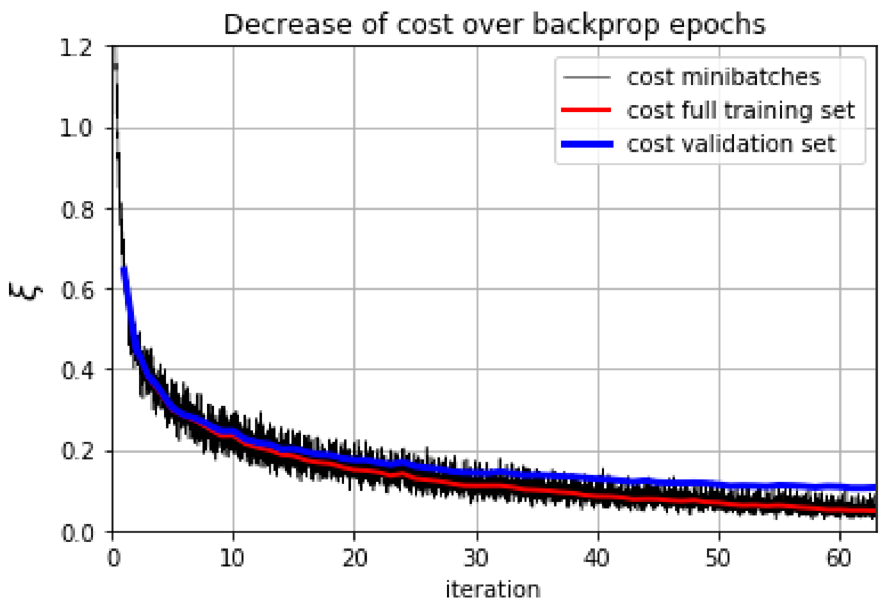


Figure 4.2.2: Cost function convergence to a minimum over training epochs.

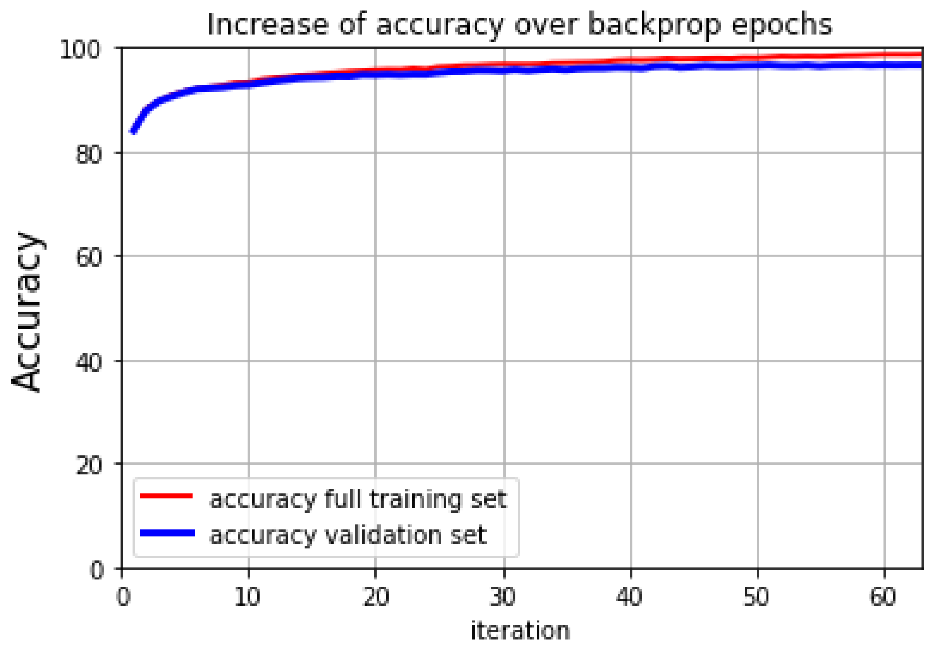


Figure 4.2.3: Increase of accuracy over training epochs.

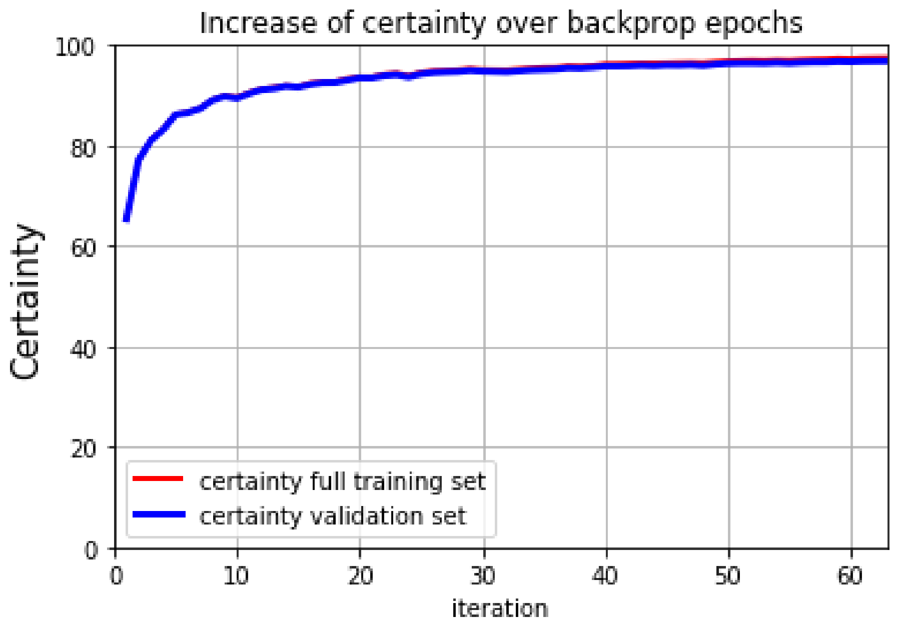
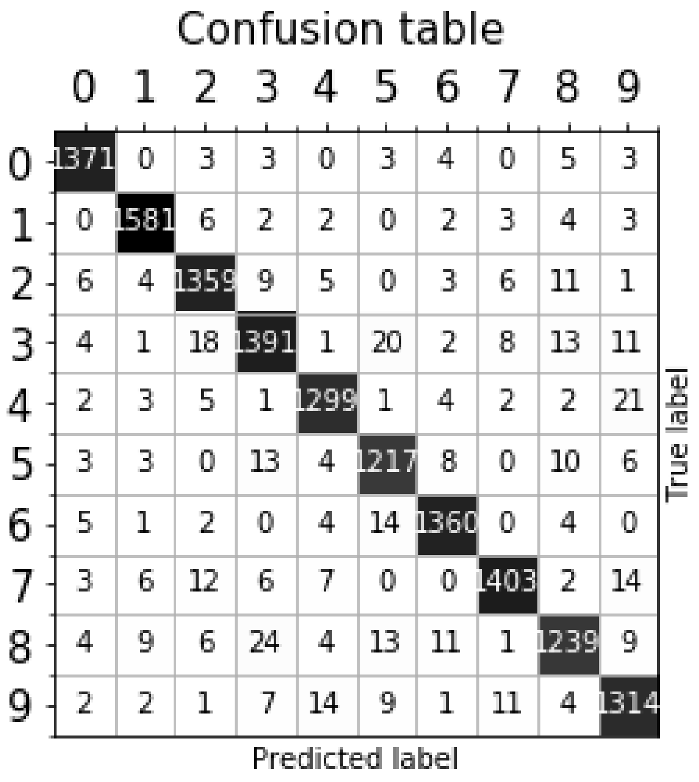


Figure 4.2.4: Increase of certainty over training epochs.


Confusion table

True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	1371	0	3	3	0	3	4	0	5	3
1	0	1581	6	2	2	0	2	3	4	3
2	6	4	1359	9	5	0	3	6	11	1
3	4	1	18	1391	1	20	2	8	13	11
4	2	3	5	1	1299	1	4	2	2	21
5	3	3	0	13	4	1217	8	0	10	6
6	5	1	2	0	4	14	1360	0	4	0
7	3	6	12	6	7	0	0	1403	2	14
8	4	9	6	24	4	13	11	1	1239	9
9	2	2	1	7	14	9	1	11	4	1314

Figure 4.2.5: Confusion table of predictions.

4.2.2 Four-Layer NN Inference C++ Implementation

The C++ implementation serves as a benchmark, against which the OpenCL system's performance will be compared. The model and test data are included to the program through header files. The full implementation can be seen in the included design archive [Appendix 6].

4.2.3 Four-Layer NN Inference OpenCL Implementation

The OpenCL implementation is comprised from two main parts: host program, where I setup the environment, declare and initialize data structures and post-process the results (including verification and writing to a data file); the kernels code [Appendix 3]: implementation of the NN's architecture into three kernels: hidden layer 1 (HL_1), hidden layer 2 (HL_2) and output layer (OL). The model, test samples and their corresponding targets are included to the host program through header files.

The three kernels are launched from separate queues as NDRanges of a single work group with 14 000 work items (each processing a single image). The use of data and task parallelism inside kernels is limited, as SIMD is not allowed while using channels and reqd_work_group_size() attribute is not specified, due to limitations in hardware resources. Kernel parallelism is achieved through the use of channels between HL_1, HL_2 and OL. Kernel pipeline schematic can be seen in Fig. 5.5. Due to its high-accuracy requirements and complex operations like finding the maximum element in a vector or exponentials of very small numbers, I choose to implement the Softmax activation on the host side.

I am interested in studying the FPGA performance of a more complex ML application with focus on accuracy and channels pipelining.

I believe the results and insight, gather from the above-mentioned studies, will give clarity on the best-performing host-side parallelism methods, communications approach, FPGA kernel, data & task parallelism, optimizations and code structures.

5. Results and Discussion.

The results from the experiment show great promise for future FPGA acceleration of ML applications. FPGAs are able to vastly speed up integer or lower-precision floating point calculations, with peak testing speed-up of 1022%, which, based on average FPGA power consumption, implies even greater power-efficiency gain compared to a CPU implementation. However, some drawbacks are also occurrent. High-precision floating point calculations prove difficult and system sizes are limited. Nevertheless, having in mind this experiment used Cyclone V FPGA, which is an older technology, especially compared to the current state-of-the-art FPGAs likes the Arria 10, I believe many of the problems have already been solved. FPGA technology develops fast, with a focus on logic size and a better floating-point precision.

Overall, my results prove that FPGAs can boost ML applications' performance. Through the vast number of implementations and versions, I proved that FPGAs are extremely flexible in their reconfigurability. These facts prove my thesis that FPGAs show great promise in accelerating ML. Moreover, they are a good candidate for facilitating Edge ML Computing.

5.1 Simple Linear Regression Results

Results from the SLR studies can be seen in Table 5.1 and Table 5.2. Each test was carried out three times in identical conditions and the average was taken. As seen from Fig. 4.3, the FPGA calculates without any noticeable errors.

5.2 Four-Layer Neural Network Results

Despite the NN inference working impeccably in emulation and compiling without any problems, it loses accuracy when running on the FPGA. I believe the cause is the lack of precision when calculating very small floating-point numbers due to the older technology of the Cyclone V FPGA. Especially having in mind the NN inference requires over 94 000 floating-point operations per output, compared to SLR's 27. I used multiple approaches in effort to fixing the issue, including incorporating double precision and utilizing FPGA memory, without any effect. Results can be seen in Table 5.3. The kernel-system's schematic and pipeline data-flow can be seen in Fig. 5.5. Each image is loaded from the global memory in HL_1, where its activations are calculated and fed into the channel. Next, HL_2 reads the channel and stores the features into local memory, from where they are loaded for HL_2's activations computation. Similarly, the OL calculates the outputs and writes them back to global memory.

Table 5.1: SLR implementations logic size and performance testing results. Part 1.

Board: DE1-SOC **System:** Cyclone V: 5CSEMA5F31C6N **Host:** Dual-core ARM Cortex-A9	**Dataset:** {IN: vector<float> [10 000 000] ∈ {-80.0:80.0}; OUT: vector<float> [10 000 000]; (27 FLO* per each OUT[x])								
Implementation **Characteristic**	**SLR_host_c**	**SLR**	**SLR_glb**	**SLR_w**	**SLR_f**	**SLR_g**	**SLR_k_g**	**SLR_k_g_b**	**SLR_q**
FPGA Logic	**0%**	**21%**	**21%**	**21%**	**16%**	**21%**	**21%**	**21%**	**21%**
ALUTs	**0%**	**10788 (10%)**	**10711 (10%)**	**10710 (10%)**	**7689 (7%)**	**10788 (10%)**	**10788 (10%)**	**10788 (10%)**	**10788 (10%)**
Board interface	0	2160	2160	2160	2160	2160	2160	2160	2160
Constant cache interconnect	0	554	554	554	554	554	554	554	554
Global interconnect	0	555	555	555	555	555	555	555	555
SLR kernel	**0%**	**7519 (7%)**	**7442 (7%)**	**7441 (7%)**	**4420 (4%)**	**7519 (7%)**	**7519 (7%)**	**7519 (7%)**	**7519 (7%)**
FFs	**0%**	**24683 (11%)**	**24551 (11%)**	**24881 (11%)**	**20854 (10%)**	**24683 (11%)**	**24683 (11%)**	**24683 (11%)**	**24683 (11%)**
Board interface	0	1908	1908	1908	1908	1908	1908	1908	1908
Constant cache interconnect	0	7440	7440	7440	7440	7440	7440	7440	7440
Global interconnect	0	5381	5381	5381	5381	5381	5381	5381	5381
SLR kernel	**0%**	**9954 (5%)**	**9822 (4%)**	**10152 (5%)**	**6125 (3%)**	**9954 (5%)**	**9954 (5%)**	**9954 (5%)**	**9954 (5%)**
RAMs	**0%**	**77 (15%)**	**78 (15%)**	**77(15%)**	**68 (13%)**	**77 (15%)**	**77 (15%)**	**77(15%)**	**77(15%)**
Board interface	0	20	20	20	20	20	20	20	20
Constant cache interconnect	0	29	29	29	29	29	29	29	29
Global interconnect	0	0	0	0	0	0	0	0	0
SLR kernel	**0%**	**28 (5%)**	**29 (5%)**	**28 (5%)**	**19 (4%)**	**28 (5%)**	**28 (5%)**	**28 (5%)**	**28 (5%)**
DSPs	**0%**	**18 (16%)**	**19 (16%)**	**20 (18%)**	**4 (4%)**	**18 (16%)**	**18 (16%)**	**18(16%)**	**18(16%)**
SLR kernel	**0%**	**18 (16%)**	**18(16%)**	**20 (18%)**	**4 (4%)**	**18 (16%)**	**18 (16%)**	**18 (16%)**	**18 (16%)**
Performance					**				
Real Time (ms)	**7675.22**	**975.144**	**985.489**	**1338.999**	**10252.367**	**981.534**	**965.954**	**963.845**	**981.907**
Processing Time (ms)	**7676.67**	**670**	**680**	**1036.67**	**9903.3**	**673.3**	**656.667**	**660**	**673.33**
Kernel Time (ms)	**0**	**305.7**	**305.99**	**305.69**	**289.03**	**305.68**	**305.68**	**305.71**	**152.99 + 152.97**
Speed Up	**100%**	**787%**	**779%**	**573%**	**75%**	**782%**	**795%**	**796%**	**782%**
Logic Efficiency. Speed Up/Logic Size	**NA**	**37.48**	**37.1**	**27.29**	**4.79**	**37.24**	**37.86**	**37.9**	**37.24**
Speed-Weighted Logic Efficiency. **Speed Up*100*(e^Speed Up/e^10)/Logic Size**	**NA**	**4.454**	**4.07**	**0.382**	**0.0005**	**4.21**	**4.874**	**4.928**	**4.21**

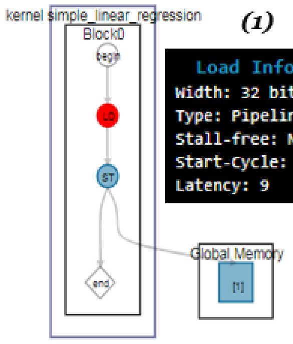
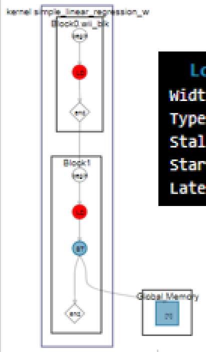
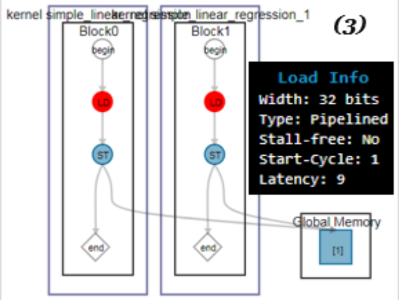
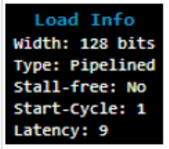
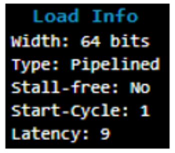
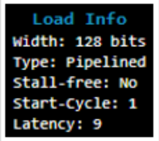
~~Implementation~~ Characteristic	SLR_host_c	SLR	SLR_glb	SLR_w	SLR_f	SLR_g	SLR_k_g	SLR_k_g_b	SLR_q
Kernel System Schematic		 kernel simple_linear_regression Block0 begin LD ST end Global Memory [1]	*(1)* Load Info Width: 32 bits Type: Pipelined Stall-free: No Start-Cycle: 1 Latency: 9	 kernel simple_linear_regression_w Block1 Global Memory	*(2)* Load Info Width: 32 bits Type: Pipelined Stall-free: No Start-Cycle: 1 Latency: 9	*(1)*	*(1)*	*(1)*	*(1)*
* *As the features take too much memory on the host, the CPU runs out of memory, the experiment was run with 1 000 000 elements. The values scaled, (multiplied by 9.6 for kernel time & by 10 for other timings) by appropriate multiples, deduced by observing the proportions of timings between different input sizes.	* Floating Point Operations. Only: Addition, Subtraction, Compare (w=1);Division, Multiply (w=4);Sine, Exponential (w=8).								

Table 5.2: SLR implementations logic size and performance testing results. Part 2.

Implementation / Characteristic	SLR_k_q	SLR_t	SLR_k_t	SLR_k_g_CU	SLR_k_g_SIMD	SLR_k_g_SIMD_CU	SLR_k_g_V_D
FPGA Logic	35%	21%	35%	60%	40%	46%	40%
ALUTs	21318 (19%)	10788 (10%)	21318 (19%)	40135 (37%)	24220 (22%)	29079 (27%)	24145 (22%)
Board interface	2160	2160	2160	2160	2160	2160	2160
Constant cache interconnect	554	554	554	894	554	554	554
Global interconnect	3566	555	3566	9588	555	3566	555
SLR kernel	2 x 7519(7%)	7519 (7%)	2 x 7519(7%)	27493 (25%)	20951 (19%)	22799 (21%)	20876 (19%)
FFs	36404 (17%)	24683 (11%)	36404 (17%)	58462 (27%)	42327 (19%)	46606 (21%)	42190 (19%)
Board interface	1908	1908	1908	1908	1908	1908	1908
Constant cache interconnect	7440	7440	7440	9500	7440	7440	7440
Global interconnect	7148	5381	7148	10682	5381	7148	5381
SLR kernel	2 x 9954 (5%)	9954 (5%)	2 x 9954 (5%)	36372 (17%)	27598 (13%)	30110 (14%)	27461 (13%)
RAMs	105 (20%)	77(15%)	105 (20%)	176 (34%)	104 (20%)	123 (24%)	104 (20%)
Board interface	20	20	20	20	20	20	20
Constant cache interconnect	29	29	29	44	29	29	29
Global interconnect	0	0	0	0	0	0	0
SLR kernel	2 x 28 (5%)	28 (5%)	2 x 28 (5%)	112 (22%)	55 (11%)	74 (14%)	55 (11%)
DSPs	36 (32%)	18(16%)	36 (32%)	72 (64%)	72 (64%)	72 (64%)	72 (64%)
SLR kernel	2 x 18(16%)	18 (16%)	2 x 18(16%)	72 (64%)	72 (64%)	72 (64%)	72 (64%)
Performance							
Real Time (ms)	839.08	625.6 + 650.6	848.7 + 737.4	794.695	750.66	772.991	1026.949
Processing Time (ms)	656.67	803.3 + 806.7	813.3 + 520	663.33	656.67	676.67	666.67
Kernel Time (ms)	180.89 + 180.82	160.7 + 155.5	180.8 + 182.5	127.702	91.925	100.423	359.9
Speed Up	915%	793%	968%	966%	1022%	993%	747%
Logic Efficiency. Speed Up/Logic Size	26.14	37.8	27.66	16.1	25.55	21.59	18.68
Speed-Weighted Logic Efficiency. Speed Up*100*(e^Speed Up/e^10)/Logic Size	11.173	4.77	20.085	11.46	31.837	20.13	1.488
Kernel System Schematic	*(3)*  kernel simple_linear_regression kernel simple_linear_regression_1 Load Info Width: 32 bits Type: Pipelined Stall-free: No Start-Cycle: 1 Latency: 9	*(1)*	*(3)*	*(1)*	*(1)*  Load Info Width: 128 bits Type: Pipelined Stall-free: No Start-Cycle: 1 Latency: 9	*(1)*  Load Info Width: 64 bits Type: Pipelined Stall-free: No Start-Cycle: 1 Latency: 9	*(1)*  Load Info Width: 128 bits Type: Pipelined Stall-free: No Start-Cycle: 1 Latency: 9

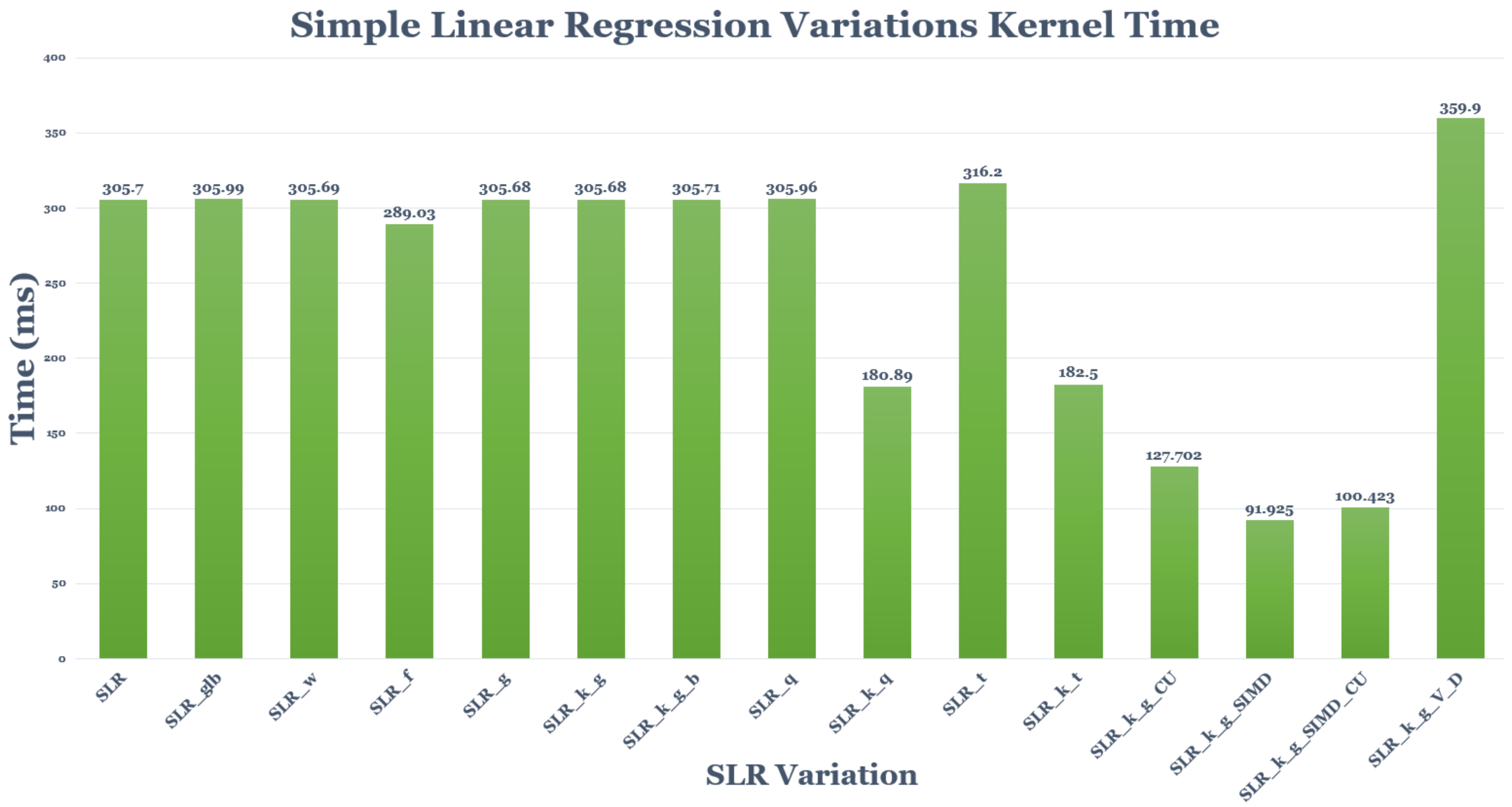


Fig. 5.1: SLR implementations kernel times.

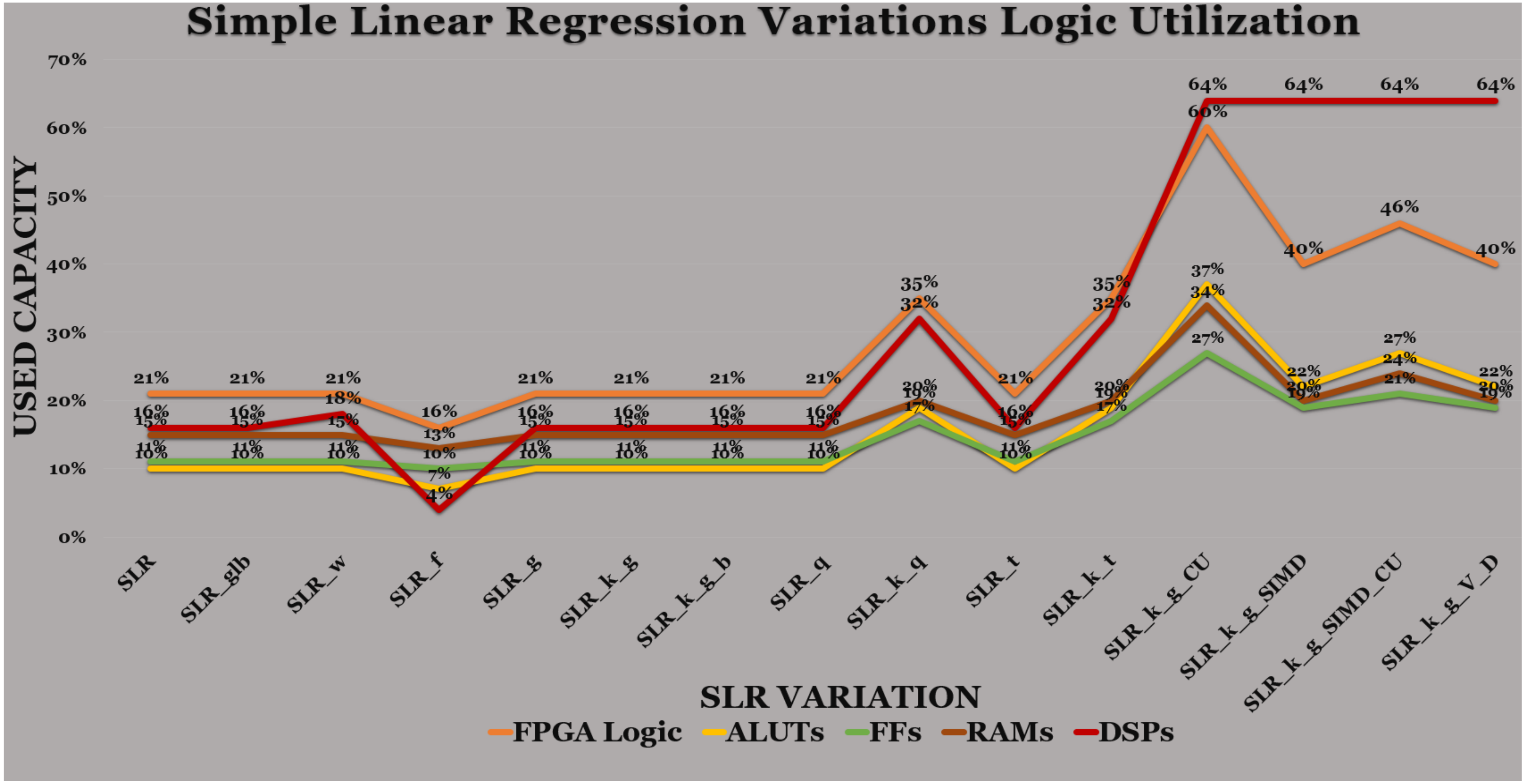


Fig. 5.2: SLR implementations logic size.

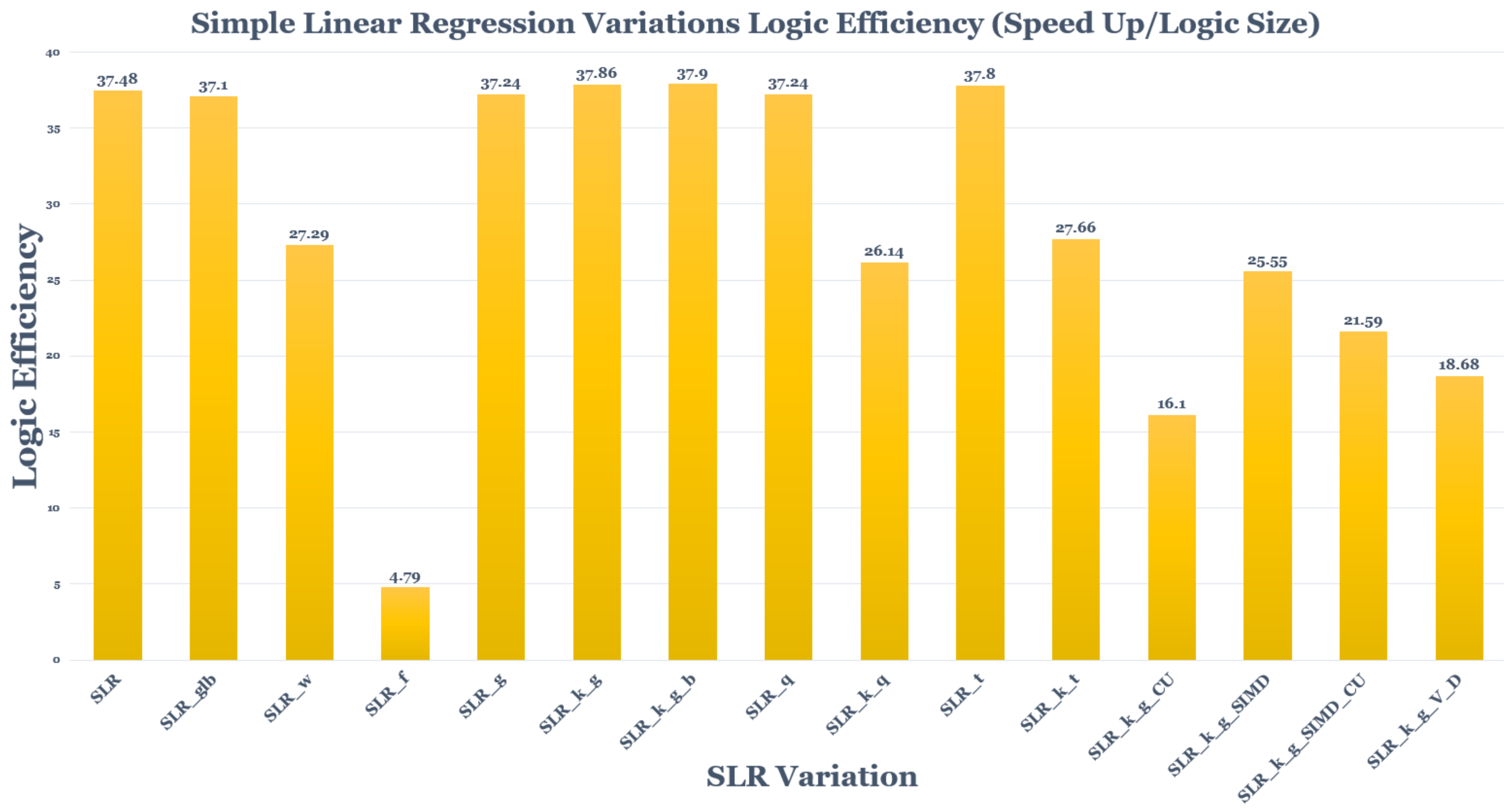


Fig. 5.3: SLR implementations logic efficiency.

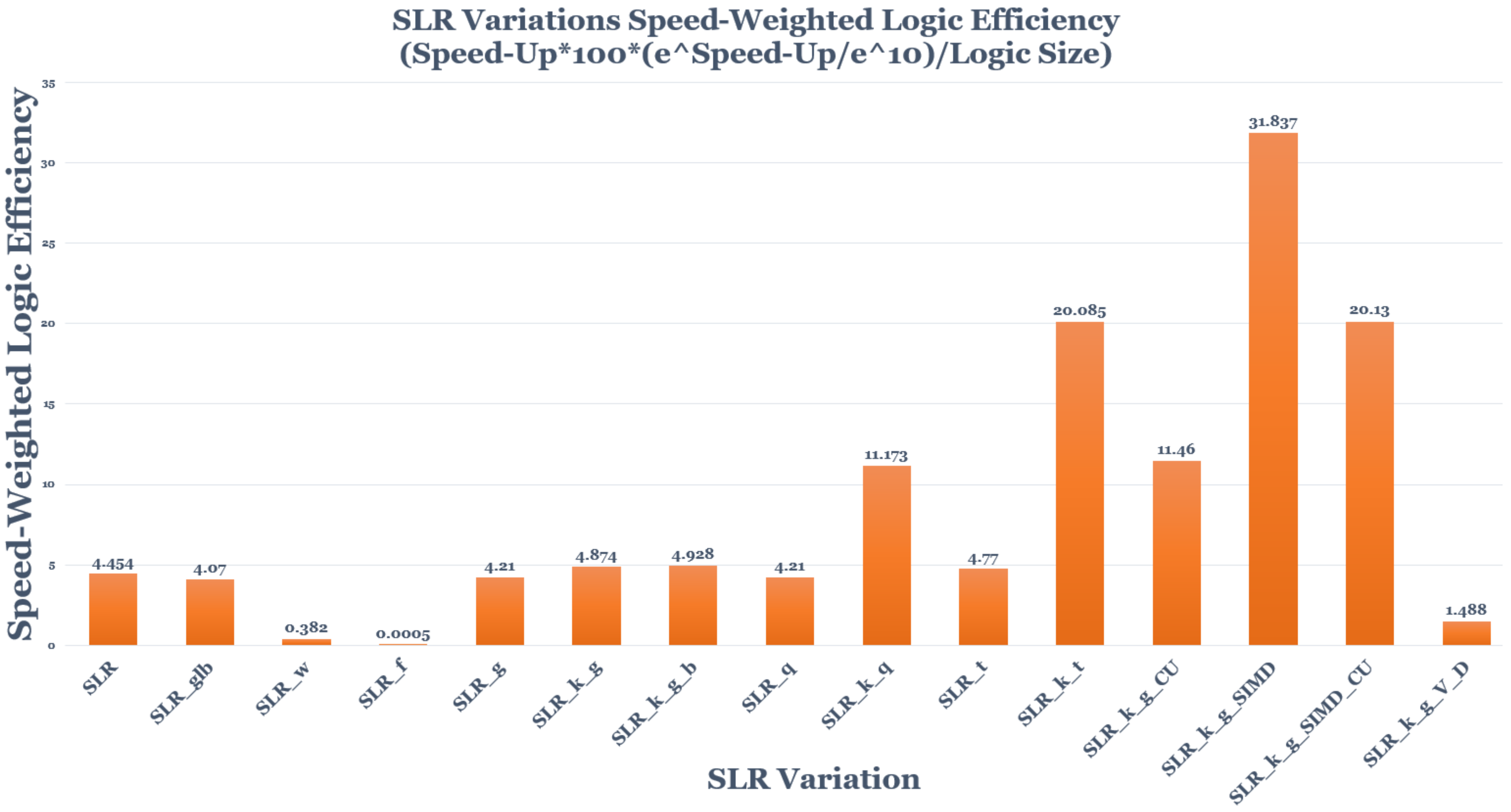


Fig. 5.4: SLR implementations speed-weighted logic efficiency.

Table 5.3: 4-Layer NN logic size and performance testing results.

Dataset: {IN: const float X_test[14000][784];
MODEL: W_HL_1[784][500], B_HL_1[500]; W_HL_2[500][150], B_HL_2[150]; W_OL[150][10], B_OL[10];
OUT: float Y[14000][10];**}**

94109 FLO* per each OUT[x][y]

Implementation / **Characteristic**	**4_Layer_NN_host_c**	**4_Layer_NN_host**
FPGA Logic	**0%**	**51%**
ALUTs	**0%**	**31211 (28%)**
Board interface	0	2160
Constant cache interconnect	0	894
Global interconnect	0	11636
HL_1	**0%**	**5089 (5%)**
HL_2	**0%**	**6212 (6%)**
OL	**0%**	**6050 (6%)**
FFs	**0%**	**53185 (24%)**
Board interface	0	1908
Constant cache interconnect	0	9500
Global interconnect	0	12730
HL_1	**0%**	**11544 (5%)**
HL_2	**0%**	**12243 (6%)**
OL	**0%**	**14696 (7%)**
RAMs	**0%**	**191 (37%)**
Board interface	0	20
Constant cache interconnect	0	40
Global interconnect	0	0
HL_1	**0%**	**48 (9%)**
HL_2	**0%**	**64 (12%)**
OL	**0%**	**59 (11%)**
DSPs	**0%**	**19 (17%)**
HL_1	**0%**	**1 (1%)**
HL_2	**0%**	**16 (14%)**
OL	**0%**	**2 (2%)**
Performance		
Real Time (ms)	**149116.6703**	**161822.591**
Processing Time (ms)	**149026.67**	**480**
Processing Cycles (n)	**149026666.7**	**480**
Kernel Time (ms)	**0**	**161348.1 + 161348.7 + 161349.5**
Accuracy	**95.6929%**	**10%**
Certainty	**95.27%**	**59.19%**
Speed Up	**100%**	**92.14%**

* **F**loating **P**oint **O**perations. Addition, Subtraction, Compare (w=1); Division, Multiply (w=4); Sine, Exponential (w=8).

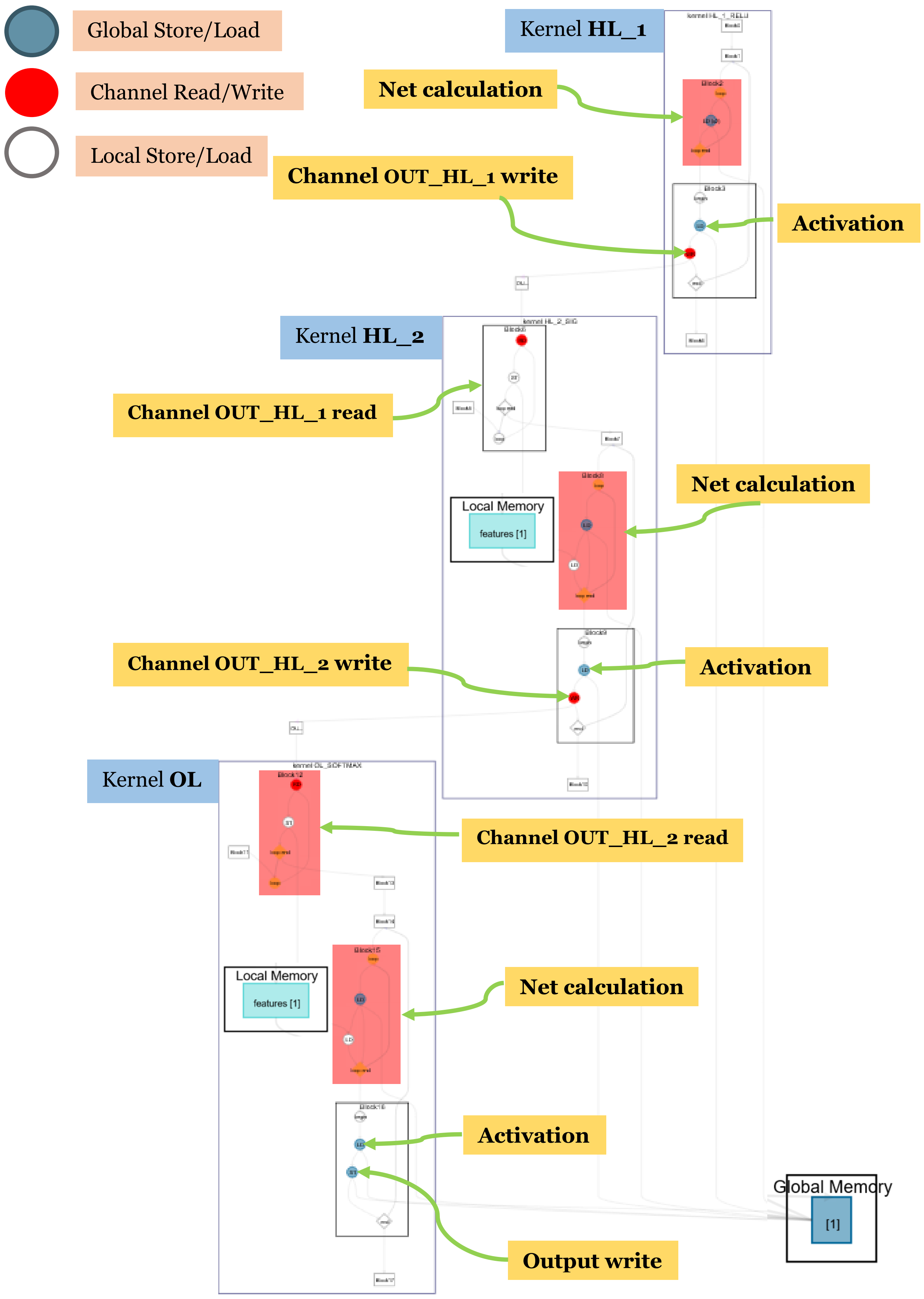


Fig. 5.5: 4-Layer NN Kernel System Schematic.

5.3 Discussion

Based on run-time measurements and implementation logic reports, I calculated three evaluation metrics. Speed up measures the run-time improvement against the C++ benchmark, logic efficiency – the speed up, compared to the logic size and speed-weighted logic efficiency, which assigns higher priority on lower run times. I decided to take logic size as an important evaluation metric, because it has straight relationship with power consumption and cost. Nevertheless, having in mind the big variation in run-time improvement, I decided to put an exponential weighting on speed up, as the higher it is, the more important it gets, compared to logic size.

Study of FPGA Memory

When the speed-ups of SLR and SLR_glb are compared, it can be seen that local memory utilization delivers performance improvement over global memory. From Table 1 it can be seen local memory is more logic-expensive. However, based on the logic efficiencies [Fig. 5.3-4], local memory should be prioritized over global (full local memory caching).

Study of OpenCL communications and FPGA computation.

Based on the SLR_w test, communicating constant, highly-used data, especially of small size, greatly damages performance, as kernel-time stays the same but host-side processing time increases significantly.

As seen in Table 1, SLR_f, delivers worse performance than SLR_host_c, due to excessive host processing time. This is the case because on top of calculating features, the host needs to communicate them to the FPGA.

Overall, as much as possible computationally-heavy functions, need to be performed on the host. The less communication - the better, with on-FPGA priority for often-used data.

Study of host-controlled data & task parallelism.

- ❖ Based on SLR_g and SLR_k_g test data, in contrast with GPU OpenCL implementations, launching multiple work-groups does not cause kernel parallelism, as there is only one physical CU on the FPGA. Simply launching multiple work-groups, slightly hinders performance as they need to be scheduled. Adding a <*reqd_work_group_size*> improves performance as the AOC optimizes the kernel.
- ❖ Similarly, to the previous test, launching kernel objects from different queues with a single physical CU does not grant speed-up as the two queues need to execute sequentially. However, adding a second CU, enables a concurrent execution. Launching two kernel copies from two separate queues, proves an efficient manual way to achieve kernel parallelism.
- ❖ Using threaded host program, boosts performance as the pre-processing and setup can be done in parallel. However, for FPGA kernel parallelism, a second copy of the kernel needs to be declared.

Overall, the best host-controlled parallelism technique is using multiple CUs with multiple threads. This forces both – host-side parallelism, as well as, FPGA kernel parallelism. Specifying *<reqd_work_group_size>* is recommended, as it lets the compiler optimize the kernel pipeline. Threads will only run in parallel if there are enough free CPU cores. Their sequential execution can result in a worse performance, then a single-threaded application.

Study of work-group size

In my experiment, using smaller work group size did not have an effect on performance. However, it could prove a good approach if, the smaller size leaves space for task parallelism or loop unrolling in the kernel pipeline.

Study of FPGA-controlled data parallelism (FPGA Performance Optimization).

Looking at the four kernel optimization techniques test data, it can be seen that SIMD performs best. Four element-wide SIMD implementation takes less than twice the original SLR logic, while delivering additional 235% speed up, compared to the benchmark code. It also speeds up the kernel execution with 332.5% [Fig. 5.1], which would lead to even greater outperformance of the SIMD method for bigger problems.

The second-best performing implementation is the mix between 2 element-wide SIMD and 2 CU replication. The 4 CU replication approach also performs well but on much higher logic-size cost. Vector data type vectorization strikes the worse performance.

On the speed-up weighted logic efficiency [Fig 5.4], SIMD proves vastly superior than CU replication because of two main drawbacks of replication: slower processing - separate CUs memory accesses are not coalesced, while using same memory paths, causing bottlenecks and data access contention; bigger size (more power) - as seen from the data and system schematics in Table 2, replication results in more control and communication logic.

I believe the significantly worse performance of *<float4>* is due to removing task parallelism optimization, which otherwise the AOC applies.

5.4 Best Practices for Optimizing FPGA OpenCL Code

Communications Optimization

1. **Memory Alignment:** should always be used if possible.
2. **Local Memory Caching:** local memory should be utilized as much as possible with priority for often-used data.
3. **Coalescing Memory Access:** the most efficient parallelism technique. Should be utilized either automatically (SIMD declaration) or manually.
4. **Channels:** should be used for big designs with multiple kernels, which can run concurrently and need to communicate a lot of data between them. While automatic SIMD cannot be used, manual should be implemented.

Parallelism Optimizations

1. **Kernel Vectorization:** as mentioned above, the most efficient (and fast) parallelism approach. Needs to always take priority. Manual implementation can grant the best results [23] with automatic closely following. Usage of vector data types is very dependent on the implementation. Nevertheless, it needs to be noted that as seen in Fig 5.2, some of the FPGA logic like RAM or DSPs does not scale proportionally to overall logic, which might cause limitations on parallelism.
2. **Loop Unrolling:** improves throughput linearly but with the associated cost of bigger logic. Should be implemented only if SIMD allows space.
3. **Computer Unit Replication:** should be looked at last, if there are resources left from the first two approaches. Except in cases where SIMD (power of 2) cannot ultimately utilize resources (no space for a 16 SIMD but enough space for 3 replicas of a 4 SIMD kernel). CU copies will only be utilized if there are enough dispatched work-groups.

Kernel Specification

1. Task kernels need to be used with computational-heavy, small-data problems, which can greatly benefit from task and loop parallelism.
2. NDRange kernels need to be used with big-data problems, which can greatly benefit from data and kernel parallelism.
3. Minimum data should be transferred between host and FPGA, in big chunks, rather than small. Maximum FPGA computation should be utilized with communications between kernels facilitated through channels.

6. Project Management

I was able to finish the project well ahead of the deadline, leaving plenty of time for refining my work. Despite the big size of the project, many software problems and health issues, due to my risk management, I was able to do more work than initially planned. I succeeded in achieving all the goals set in my Project Brief [Appendix 4] and Progress Report of:

- Implementing a ML application on a FPGA;
- Proving FPGAs can boost its performance;
- Studying interfacing & host-side optimization strategies for OpenCL on FPGAs.

Furthermore, I decided to expand my research by studying:

- FPGA-based parallelism;
- Memories & work-group size effect on performance;
- Implementing a more complex ML algorithm in the form of a NN.

I had regularly scheduled meetings with Prof. Zwolinski, as well as, two meetings with Mr. McNally. Project management overview can be seen in Appendix 5: Gantt Chart of the project schedule [A. 5.1], Gantt Chart of the actual project flow [A. 5.2], Risk Assessment [A. 5.3], Skills Audit [A. 5.4].

7. Conclusion

In this report, I present the development of SLR and a four-layer NN implementations, as well as, studying their performance using Intel FPGA SDK for OpenCL.

I start with a description of the SLR Python implementation, followed by the C++ benchmark, incorporating initialization and inference of the model. I finish the SLR commentary by presenting the OpenCL version and the family of variations, studying different aspects of OpenCL's FPGA framework.

Next, the four-layer NN for classifying the MNIST dataset is introduced. Sequentially, its architecture, Python code and training results, C++ and OpenCL implementations are discussed.

I present and analyze the systems' test results, with insights on the advantages, disadvantages and opportunities of OpenCL implementations for FPGA.

The report finishes with the best practices for OpenCL systems-development for FPGAs, which I derived through the experiment.

I was able to achieve the project's objectives of: implementing ML applications in OpenCL, studying their behavior in multiple specifications and devising a strategy for OpenCL ML systems-development. Furthermore, I proved that computational-heavy ML operations can be accelerated with OpenCL on FPGAs, creating a more power-efficient platform. As I have discussed in 4.1.1 and 4.2, the studied problems are applicable to all ML methods, making the above-displayed findings relevant to general ML acceleration.

7.1 Future Work

I would like to use the best practices and insight's relevance to OpenCL FPGA development of ML applications in implementing a smart surveillance AI system. It would use several FPGAs for accelerating multiple ML algorithms/models for image, sound and sensor-data processing. Such system could be used in smart homes, autonomous cars or as a standalone security platform.

The project will not only broaden the research area, by studying acceleration of various ML methods, but also utilize the newest FPGA and OpenCL technology.

References

[1] M. Nielsen. Neural Networks and Deep Learning http://neuralnetworksanddeeplearning.com/index.html, Online book, Accessed May, 2017.

[2] A. L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development,* 3(3):210–229, 1959.

[3] Anderson, J. (1995). *An introduction to neural networks*. Cambridge, Mass.: MIT Press.

[4] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[5] Goodfellow, I., Bengio, Y. and Courville, A. (2017). *Deep learning*. Cambridge, Mass: The MIT Press, pp.326-366.

[6] Goodfellow, I., Bengio, Y. and Courville, A. (2017). *Deep learning*. Cambridge, Mass: The MIT Press

[7] Bandyopadhyay, D. & Sen, J. Wireless Pers Commun (2011) 58: 49. https://doi.org/10.1007/s11277-011-0288-5

[8] M. Satyanarayanan, "The Emergence of Edge Computing," in *Computer*, vol. 50, no. 1, pp. 30-39, Jan. 2017. doi: 10.1109/MC.2017.9, URL: ieeexplore

[9] Satyanarayanan, M. (2011). Mobile computing: the Next Decade. *ACM SIGMOBILE Mobile Computing and Communications Review*, 15(2), p.2.

[10] Russell, Stuart J.; Norvig, Peter (2003), Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, ISBN: 013-790395-2.

[11] The Khronos Group. (2017). *OpenCL - The open standard for parallel programming of heterogeneous systems*. [online] Available at: https://www.khronos.org/opencl/ [Accessed 9 Dec. 2017].

[12] Altera.com. (2017). *Intel FPGA SDK for OpenCL*. [online] Available at: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html [Accessed 9 Dec. 2017].

[13] Lee, Peter M. (2012). "Chapter 1". *Bayesian Statistics*. Wiley. ISBN 978-1-1183-3257-3.

[14] Rosenblatt, Frank (1957), The Perceptron--a perceiving and recognizing automaton. Report 85-460-1, Cornell Aeronautical Laboratory.

[15] Farrahi, K., Artificial Neural Networks, https://secure.ecs.soton.ac.uk/notes/comp3206/neuralnetworks.pdf

[16] Stanford. CS231n Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io, Accessed April, 2018.

[17] T. M. Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, March 1997.

[18] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL Programming Guide*. Addison-Wesley, July 2011.

[19] Khronos.org. (2012). *OpenCL Details*. [online] Available at: https://www.khronoss.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf [Accessed 21 Apr. 2018].

[20] Khronos OpenCL Working Group. The OpenCL Specification, Version 1.0,2009. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.0.pdf, Accessed December, 2017.

[21] T. S. Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, and J. Freeman.
OpenCL for FPGAs: Prototyping a Compiler. *Int'l Conf. Reconfigurable Systems and Algorithms, ERSA'12*, 2012.

[22] OpenCL on FPGAs for GPU Programmers, http://design.altera.com/openclforward, Acceleware Corp., 2014, [Online; Accessed Apr, 2018].

[23] Waidyasooriya, H., Hariyama, M. and Uchiyama, K. (2018). *Design of FPGA-Based Computing Systems with OpenCL*. Cham: Springer International Publishing.

[24] Altera.com. (2017). *Intel FPGA SDK for OpenCL Programming Guide*. [online] Available at: https://www.altera.com/documentation/mwh1391807965224.html [Accessed 9 Dec.2017]

[25] Altera.com. (2017). *Intel FPGA SDK for OpenCL Best Practices Guide*. [online] Available at: https://www.altera.com/documentation/mwh1391807516407.html [Accessed 9 Dec.2017].

[26] Jia, Q., Zhou, H. *Tuning Stencil Codes in OpenCL for FPGAs*. 2-5 Oct. 2016, Scottsdale, AZ, USA, IEE, DOI**:** 10.1109/ICCD.2016.7753287

[27] Wang, C., Gong, L., Yu, Q., Li, X., Xie, Y. and Zhou, X. (2016). DLAU: A Scalable Deep Learning Accelerator Unit on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp.1-1.

[28] Li, Huyuan, "Acceleration of Deep Learning on FPGA" (2017). *Electronic Theses and Dissertations*. 5947. http://scholar.uwindsor.ca/etd/5947

[29] Yann.lecun.com. (2018). *MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges*. [online] Available at: http://yann.lecun.com/exdb/mnist/ [Accessed 27 Apr. 2018].

Appendix

A. 1 Step-by-Step Guide for Running OpenCL Applications on Intel FPGAs.

Example: DE1-SoC Board, Windows Compilation

Follow Instructions:

From page 11 in (or for your target FPGA):
https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_c5soc_getting_started.pdf

From page 10 in (or for your target Development Board):
http://www.terasic.com.tw/attachment/archive/836/DE1SOC_OpenCL_v02.pdf

1. Download Standard Edition OpenCL SDK v16.1 (V. BSP supports)
   a. Quartus + AOC.
2. Download Intel SoC FPGA Embedded Design Suite v16.1 (V. BSP supports)
   a. Cross-Compiler for host applications.
3. Download DE1-SoC (your board) OpenCL Board Support Package (BSP)
4. Install the SDK (after unpacking, run setup.bat)
5. Set Up the Environment Variables
6. Install the EDS (make sure Arm DS-5 is installed at the end)
7. Install the BSP
8. Set up additional Environmental Variables (license, BSP, PATH)
9. Restart Computer!!!
10. In your working cmd.exe, execute the SDKROOT\init_opencl.bat (might need to add Visual Studio (link.exe) directory to PATH)
11. If the compilation does not progress for hours or displays: "Error: aoc: Can't find VisualStudio linker LINK.EXE.", YOU SHOULD NOT RUN init_opencl.bat, instead: make sure you have included link.exe parent directory to the PATH (make sure it is for the correct system x64/x86) Ex: <*Microsoft Visual\VC\Tools\MSVC\14.13.26128\bin\Hostx64\x64*> and also it is before any other <LINK.exe> Ex: "C:\msys64\usr\bin" in the PATH. Might need to include any dependable libs, run <*Microsoft Visual Studio ROOT\VC\Auxiliary\Build\vcvars64.bat*>! More info: *https://alteraforum.com/forum/archive/index.php/t-46646.html*
12. If you have downloaded examples newer than v16.1 (your supported BSP version): or for Win/Linux OS, you will need to replace the Makefile with one for Arm32 cross (or use a control variable to switch between compilations) (add <*-lacl_emulator_kernel_rt*> to <*AOCL_LINK_CONFIG*> to remove warning). You will need to replace the <*findPlatform(Intel FPGA)*> with *(Altera)* for v.17 and newer examples.

13. If you get an error that a ".so" lib file cannot be read/recognized, make sure that <*AOCL_BOARD_PACKAGE_ROOT*> environmental variable is set correctly (*s5_ref* for emulation, *c5soc/de1soc/…* for your SOC).
14. You are ready to compile the kernel code with < *aoc (-v) device/kernel.cl -o bin/kernel.aocx --board de1soc_sharedonly (your BSP)*> in cmd.exe and cross-compile the host code with *EDSROOT\Embedded_Command_Shell.bat* <make>. If you get "*Compiler Error, not able to generate hardware*", look at the report, probably your design is too big for the FPGA!

Follow instruction in (or for your Development Board):
ftp://ftp.altera.com/up/pub/Altera_Material/16.0/Tutorials/Linux.pdf

15. Download the Linux image for the DE1-SoC board from: https://altera.com/support/training/university/materials-software.html
16. Write the image to a microSD (make sure the MSEL is configured).
17. Connect the board to power, UART to USB & Ethernet & put in the card
18. Start Putty, go to serial and open a connection. Start the board.
19. <ipconfig> your windows machine and <ifconfig> the board. If the first 3 numbers of the IPs are not the same: <ifconfig> eth0 xxx.xxx.xxx.yy, where x's are the same as your WIN IP and yy are of your choice.
20. Download OpenSSH (if you do not have it) on you Window Machine: https://openssh.en.softonic.com/?ex=REG-60.1
21. **!!!!!**Before installing OpenSSH, make sure to save your PATH as it might wipe it**!!!!!!!** You can permanently set Environment Variables with <*setx NEWVAR %PATH%*>, make sure you run cmd.exe as administrator!
22. Install OpenSSH & open the README to see how to create keys.
23. Now on cmd.exe you can connect through SSH to the board with <*ssh root@board_IP*> & copy files with <*scp [-r] file root@board_IP:file*> (-r for directories). Password: <*password*>. Even if it cannot create /home/.ssh, the connection still works.
24. You need to send the "*arm32/*" and "*driver/*" directories from the <*terasic/de1soc*> BSP folders to the board (or make sure the libs and driver on the embedded Linux are the same as yours BSP).
25. (Back up the old dirs.!) Replace the <*opencl_arm32_rte/host/arm32*> directory that came with the Linux image (it is 14.0) with the new one (16.0). Replace the < *opencl_arm32_rte/board/c5soc/driver*> (14.0) with the new one (16.0), replace the new driver's Makefile with the old's and <make> to install the driver.
26. <*~/source init_opencl.sh*> & you can program the kernel with <*aocl program /dev/acl0 kernel.aocx*> & the host with <*chmod +x host*> (to make it executable) and <*./host –options*>
27. If you get the error : "*Wrote only 0 out of 2315488 bytes into /tmp/fileRtscVI for writingCannot read RBF from the FPGA programming section into tmp file /tmp/fileRtscVI. aocl program: Program failed.*" You

28. have run out of space in */tmp*. Execute: <*df*> to check the disk space and <*mount -o remount,size=10M /tmp/*> to increase the memory in */tmp*.
29. To connect to the Linux GUI, download VNC Viewer: https://www.realvnc.com/en/connect/download/viewer/
30. Create a new connection with <*board_ip: :5901*> & connect.

Specific Compilation/Run Errors for OpenCL FPGA Development:

When dealing with big arrays of data (models):
Include them through header files:
For C++ host:
In the header <array.h>:
File guards: #ifndef ARRAY_H
#define ARRAY_H
#endif
Declaration: extern const float ARRAY[][];
In the source <array.cpp>:
#include "array.h"
Initialization: const float ARRAY[][] = {{...},{...}}

For OpenCL kernels:
<array.cl>
File guards: #ifndef ARRAY_H
#define ARRAY_H
#endif
Declaration/Initialization: __constant (__global) float ARRAY[][] = {{...},{...}};
Be cautious, as *__constant* memory is faster but also limited of size, if you have big arrays, they need to be in *__global*.

If they need to be in the *main.cpp* source: Declare them as **global** variables or allocate memory dynamically (malloc, vectors) or the program stack overflows!

If you get the error: <*cc1plus.exe: out of memory allocating 65536 bytes*> Probably your version of the compiler has a bug, which causes high resources allocation. I advise getting the compiler on a system with high memory resources.

Do not initialize variables inside the problem class declaration, instead in a dedicated *class::init()*.

Declare an autonomous *cleanup()* function, which is called by the *AOCUTILS lib*.

Make sure to use "*\n*" at the end of each <*printf();*>s as otherwise the function might deadlock!

If you are using cannels to pipeline the kernels, make sure to launch each kernel in a separate queue, otherwise as queues are sequential, the kernels will probably deadlock.

NB!!! NDRange kernels need to be used with caution when channel pipelining is implemented. The kernels most-probably will be unable to used SIMD or multiple CU replications as this would deadlock/cause race conditions!

Keep in mind currently only 1-d arrays can be send through the clEnqueue OpenCL facilities.

CARE, OpenCL does not support 2-D arrays in global memory, meaning 1D arrays need to be used and appropriate accesses implemented or images incorporated!

A.2 Simple Linear Regression

A.2.1 Python Implementation

```
from __future__ import division, print_function, unicode_literals
%matplotlib inline
import os
import numpy as np
import matplotlib.pyplot as plt
# Used to create the training set. Training Y relationship to input X.
def somefun(x):
#The model approximate this pattern, learning from X and Y.
 return 10*np.exp(-2*x**2) +np.sin(3*x)*10 +x

Xpts = 100                                      # Training points:100
Xlin = np.linspace(-3,3,Xpts)                   # Training interval [-
3:3]
Y = somefun(X1lin)                              # Training targets, our
Ys
Y += 0.7*np.random.normal(0,1,Xpts)             # Noise is added, as in
real life, there is always distortion
XptsPred = 200                                  # Inference points:200
(the # of points, the learnt model will be tested on, fitting)
XlinPred = np.linspace(-10,10,XptsPred)         # Inference interval [-
10:10] Presentation of the power of generalization
YTrue = somefun(XlinPred)                       # True labels for the
inference interval
YTrue += 0.7*np.random.normal(0,1,XptsPred)     # Will be used to
validate the prediction
A = np.stack((np.ones(Xpts), Xlin,              # Constructing the
training features (training Design Matrix)
  np.sin(3*Xlin), np.exp(-1*Xlin**2))).T
APred = np.stack((np.ones(XptsPred), XlinPred,# Constructing the
inference features (inference Design Matrix)
  np.sin(3*XlinPred), np.exp(-1*XlinPred**2))).T
W = np.linalg.pinv(A).dot(Y)                    # Training the model
(weights & bias)
Ypred = APred.dot(W)                            # Doing inference
(making a prediction, based on the model)

plt.plot(XlinPred,Ypred)                        # Displaying the
prediction (regression) in BLUE
plt.scatter(XlinPred,YTrue, color='y')          # Displaying the true
labels in YELLOW
plt.scatter(Xlin,Y, color='r')                  # Displaying the
training targets in RED
```

Listing 2.1: slr.py

A.2.2 C++ SLR Implementation: run()

```
///////////////////////////////////////////////////////////////////
////Created by Samyuel Danyo on 06/11/2017
////////////////////////////////////////////////////////////////////
////This is C++ Benchmark program implementing a Simple Linear Regression
//// run()
void Simple_Linear_Regression::run() {
     int status;
 std::clock_t start_p;
 std::clock_t end_p;
      unsigned duration;
 std::clock_t clocks;
 double start_r;
 double end_r;
 double time_r;
      start_p = std::clock();
 start_r = aocl_utils::getCurrentTimestamp();

      weights[0] = -0.55395846;
      weights[1] = 1;
      weights[2] = 10;
      weights[3] = 8.94836441;

      printf("Launching calculation (%d elements)\n",N);

      for(unsigned j = 0; j < N; ++j) {
            features[4 * j] = 1;
            features[4 * j + 1] = input_X[j];
            features[4 * j + 2] = sin(3 * input_X[j]);
            features[4 * j + 3] = exp(-1 * input_X[j] * input_X[j]);

            prediction_Y[j] = weights[0] * features[4 * j]  + features[4
            * j + 1] * weights[1] + features[4 * j + 2] * weights[2] +
            features[4 * j + 3] * weights[3];
      }

 end_p = std::clock();
 end_r = aocl_utils::getCurrentTimestamp();
 clocks = end_p - start_p;
      duration = clocks / CLOCKS_PER_MS;
 time_r = (end_r - start_r) * 1e3;

      // Wall-clock time taken.
      printf("\nReal Elapsed Time: %0.3f ms\n", time_r);
      printf("\nProcessing Time in Clock Cycles: %d \n", (int) clocks);
      printf("\nProcessing Time: %f ms\n",(float) duration);
```

```
        // Verify results.
        pass_pred = true;
        FILE *file0 = fopen(LOG_F_NAME, "w");
        assert(file0);
        for(unsigned j = 0; j < N; ++j) {
              if(fabsf(ref_output_pred[j] - prediction_Y[j]) > (fabsf(0.05
              * ref_output_pred[j]) + 0.00001)) {
                    fprintf(file0, "Failed prediction verification of index
                    %d\nInput: %f\nOutput: %f\nReference: %f\n", j,
                    input_X[j], prediction_Y[j], ref_output_pred[j]);
            pass_pred = false;
         }
        }
        if (pass_pred == 0){
              fprintf(file0, "FAILED PREDICTION");
        }
        fclose(file0);

        FILE *file1 = fopen(OUT_F_NAME, "w");
        assert(file1);
        fprintf(file1, "X\tInput\t\tTrue Label\tCPU Model Out\tFPGA
        Prediction\tCalc Error\tResidiual\n");
        for(unsigned j = 0; j < N; ++j) {
              fprintf(file1,
              "%d\t%f\t%f\t%f\t%f\n",j,input_X[j],ref_output_pred[j],predic
              tion_Y[j],(ref_output_pred[j]-prediction_Y[j]));
        }
        fclose(file1);
}
```

Listing 2.2: main_c.cpp: run()

A.2.3 Original OpenCL SLR Implementation Host Program: run()

```
////////////////////////////////////////////////////////////////////
////Created by Samyuel Danyo on 06/11/2017
//Based on Intel FPGA example [Multithread Vector Operation] main.cpp
/////////////////////////////////////////////////////////////////////
////This is OpenCL SLR implementation host program: run()
////Written for Intel FPGA, synthesized with Intel FOGA SDK for OpenCL
void Simple_Linear_Regression::run() {
      cl_int status;
      std::clock_t start;
      std::clock_t end;
      unsigned duration;
      std::clock_t clocks;
      const double start_time = getCurrentTimestamp();
      start = std::clock();
      // Launch the problem for each device.
      scoped_array<cl_event> kernel_event(num_devices);
      scoped_array<cl_event> finish_event(num_devices);

      for(unsigned i = 0; i < num_devices; ++i) {
      // Transfer inputs to devices. Each of the host buffers supplied to
      // clEnqueueWriteBuffer is already aligned to ensure that DMA is used
      // for the host-to-device transfer.
            cl_event write_event[1];
            status = clEnqueueWriteBuffer(queue[i], input_X_buf[i],
            CL_FALSE,0, n_per_device[i] * sizeof(float), input_X[i], 0,
            NULL, &write_event[0]);
            checkError(status, "Failed to transfer the input X dataset");
  // Set kernel arguments.
            unsigned argi = 0;

            status = clSetKernelArg(kernel[i], argi++, sizeof(cl_mem),
            &input_X_buf[i]);
            checkError(status, "Failed to set argument %d", argi - 1);

            status = clSetKernelArg(kernel[i], argi++, sizeof(cl_mem),
            &prediction_Y_buf[i]);
            checkError(status, "Failed to set argument %d", argi - 1);

            // Enqueue kernel.
            // Use a global work size corresponding to the number of
            // elements for this device.
            // We don't specify a local work size, the runtime chooses
            // (it'll choose one work-group with the same size as the
            // global work-size).
            // Events are used to ensure that the kernel is not launched
            // until the writes to the input buffers have completed.
            const size_t global_work_size = n_per_device[i];
            printf("Launching for device %d (%d elements)\n", i,
            global_work_size);

            status = clEnqueueNDRangeKernel(queue[i], kernel[i], 1, NULL,
        &global_work_size, NULL, 1, write_event, &kernel_event[i]);
            checkError(status, "Failed to launch kernel");

            // Read the result. This the final operation.
            status = clEnqueueReadBuffer(queue[i], prediction_Y_buf[i],
            CL_FALSE,0, n_per_device[i] * sizeof(float), prediction_Y[i],
            1, &kernel_event[i], &finish_event[i]);
            checkError(status, "Failed to read buffer");

            // Release local events.
            clReleaseEvent(write_event[0]);
      }
      // Wait for all devices to finish.
      clWaitForEvents(num_devices, finish_event);

      const double end_time = getCurrentTimestamp();
      end = std::clock();
      clocks = end - start;
      duration = clocks / CLOCKS_PER_MS;
      // Wall-clock time taken.
      printf("\nReal Elapsed Time: %0.3f ms\n", (end_time - start_time) *
      1e3);
      printf("\nTime in Clock Cycles: %d \n", (int) clocks);
      printf("\nProcessing Time: %0.3f ms\n",(float) duration);

      // Get kernel times using the OpenCL event profiling API.
      for(unsigned i = 0; i < num_devices; ++i) {
            cl_ulong time_ns = getStartEndTime(kernel_event[i]);
            printf("Kernel time (device %d): %0.3f ms\n", i,
            double(time_ns) * 1e-6);
  }


 // Release all events.
      for(unsigned i = 0; i < num_devices; ++i) {
            clReleaseEvent(kernel_event[i]);
            clReleaseEvent(finish_event[i]);
      }
      // Verify results.
      pass_calc = true;
      pass_pred = true;
      FILE *file0 = fopen(LOG_F_NAME, "w");
      assert(file0);
	for(unsigned i = 0; i < num_devices; ++i) {
	   for(unsigned j = 0; j < n_per_device[i]; ++j) {
	      if(fabsf(ref_output_calc[i][j] - prediction_Y[i][j]) >
	      (fabsf(0.005 * ref_output_calc[i][j]) + 0.00001)) {
			fprintf(file0, "Failed calculation verification of
			device %d, index %d\nInput: %f\nOutput:
			%f\nReference: %f\n", i, j,
			input_X[i][j],prediction_Y[i][j],
			ref_output_calc[i][j]);
			pass_calc = false;
	      }
	      if(fabsf(ref_output_pred[i][j] - prediction_Y[i][j]) >
	      fabsf(1.5 / abs(input_X[i][j]) + 0.05 *
	      abs(ref_output_pred[i][j]) + 0.7 /
	      abs(ref_output_pred[i][j]) + 0.15)) {
			fprintf(file0, "Failed prediction verification of
			device %d, index %d\nInput: %f\nOutput:
			%f\nReference: %f\n", i, j, input_X[i][j],
			prediction_Y[i][j], ref_output_pred[i][j]);
			pass_pred = false;
	      }
	   }
	}
	if ((pass_pred == 0) && (pass_calc == 0))
	      fprintf(file0, "FAILED PREDICTION & CALCULATION");
	fclose(file0);

	FILE *file1 = fopen(OUT_F_NAME, "w");
	assert(file1);
	fprintf(file1, "X\tInput\t\tTrue Label\tCPU Model Out\tFPGA
	Prediction\tCalc Error\tResidiual\n");
	for(unsigned i = 0; i < num_devices; ++i) {
	      for(unsigned j = 0; j < n_per_device[i]; ++j) {
			fprintf(file1, "%d\t%f\t%f\t%f\t%f\t%f\t%f\n", (j +
			i * n_per_device[i-
			1]),input_X[i][j],ref_output_pred[i][j],ref_output_c
			alc[i][j],prediction_Y[i][j],(ref_output_calc[i][j]-
			prediction_Y[i][j]),(ref_output_pred[i][j]-
			prediction_Y[i][j]));
	      }
	}
	fclose(file1);
}
```

Listing 2.3: main.cpp: run()

A.2.4 Python Script for Plotting the SLR Results.

```python
#! /usr/bin/python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from numpy import array
file_name = 'bin/data.dat'
def get_col (col):
    column = []
    with open(file_name) as f:
      next(f)
      for line in f:
        column.append(float(line.split('\t')[col-1]))
    return column
fig1 = plt.figure()
fig3 = plt.figure()
ax1 = Axes3D(fig1)
ax3 = fig3.add_subplot(2, 1, 1)
for c, m, z in [('r', 'o', 3), ('b', '^', 5)]:
    xs = get_col(1)
    del xs[0]
    ys = get_col(2)
    del ys[0]
    zs = get_col(z)
    del zs[0]
    ax1.scatter(xs, ys, zs, color=c, marker=m)
    ax3.scatter(ys, zs, color=c, marker=m)
ax1.set_xlabel('Input Data Sample')
ax1.set_ylabel('Input Value')
ax1.set_zlabel('Output [Expected in BLUE, Predicted in RED]')
ax1.set_title('Simple Linear Regression Results')
ax1.legend()
ax3.set_xlabel('Input Data Value')
ax3.set_ylabel('Output [Expected in BLUE, Predicted in RED]')
ax3.set_title('Simple Linear Regression Results')
fig2 = plt.figure()
ax2 = Axes3D(fig2)
for c, m, z in [('r', 'o', 6), ('b', '^', 7)]:
    xs = get_col(1)
    del xs[0]
    ys = get_col(2)
    del ys[0]
    zs = get_col(z)
    del zs[0]
    ax2.scatter(xs, ys, zs, color=c, marker=m)
ax2.set_xlabel('Input Data Sample')
ax2.set_ylabel('Input Value')
ax2.set_zlabel('Error Value[Prediction in BLUE, Calculation in RED]')
ax2.set_title('Simple Linear Regression Error Results')
ax2.legend()
plt.show()
```

Listing 2.4: plot.py

A.2.5 OpenCL FPGA Kernel Implementations

```
//////////////////////////////////////////////////////////////////
//Created by Samyuel Danyo on 06/11/2017
//SLR Original Version.
__kernel void simple_linear_regression(__constant float *restrict x,
__global float *restrict y){
        // Weights vector
        __local float weights[4];
        weights[0] = -0.55395846;
        weights[1] = 1;
        weights[2] = 10;
        weights[3] = 8.94836441;
        // Get index of the work item
        int index = get_global_id(0);
        // Creating the design vector
        __local float features[4];
        features[0] = 1;
        features[1] = x[index];
        features[2] = sin(3 * x[index]);
        features[3] = exp(-1 * x[index] * x[index]);
        // Executiong our inference
        y[index] = features[0] * weights [0] + features[1] * weights
[1] + features[2] * weights [2] + features[3] * weights [3];}

//SLR_w receiving Input Data and Weights.
__kernel void simple_linear_regression_w(__constant float *restrict x,
__constant float *restrict weights,__global float *restrict y){

        // Get index of the work item
        int index = get_global_id(0);
        // Creating the design vector
        __local float features[4];
        features[0] = 1;
        features[1] = x[index];
        features[2] = sin(3 * x[index]);
        features[3] = exp(-1 * x[index] * x[index]);
        // Executiong our inference
        y[index] = features[0] * weights [0] + features[1] * weights
[1] + features[2] * weights [2] + features[3] * weights [3];}

//SLR_f receiving Features Matrix and Weights.
__kernel void simple_linear_regression_f(__constant float *restrict
weights,__constant float *restrict features,__global float *restrict
y){
        // Get index of the work item
        int index = get_global_id(0);
        // Executiong our inference
        y[index] = features[4 * index] * weights[0] + features[4 *
index + 1] * weights[1] + features[4 * index + 2] * weights[2] +
features[4 * index + 3] * weights [3];}
```

```
//SLR_glb using global instead of local memory.
__kernel void simple_linear_regression(__constant float *restrict x,
__global float *restrict y)
{
 // Weights vector
 float weights[4];
 weights[0] = -0.55395846;
 weights[1] = 1;
 weights[2] = 10;
    weights[3] = 8.94836441;
 // Get index of the work item
 int index = get_global_id(0);

 // Creating the design vector
 float features[4];
 features[0] = 1;
 features[1] = x[index];
 features[2] = sin(3 * x[index]);
    features[3] = exp(-1 * x[index] * x[index]);

 // Executiong the inference
 y[index] = features[0] * weights [0] + features[1] * weights [1] +
features[2] * weights [2] + features[3] * weights [3];}

//SLR_k_g defining a required work group size.
_attribute__((reqd_work_group_size(10000, 1, 1)))
__kernel void simple_linear_regression(__constant float *restrict x,
                         __global float *restrict y){
 // Weights vector
 __local float weights[4];
 weights[0] = -0.55395846;
 weights[1] = 1;
 weights[2] = 10;
        weights[3] = 8.94836441;
 // Get index of the work item
 int index = get_global_id(0);

 // Creating the design vector
 __local float features[4];
 features[0] = 1;
 features[1] = x[index];
 features[2] = sin(3 * x[index]);
        features[3] = exp(-1 * x[index] * x[index]);

 // Executiong our inference
 y[index] = features[0] * weights [0] + features[1] * weights [1] +
features[2] * weights [2] + features[3] * weights [3];}

//SLR_k_g_b defining a reduced required work group size.
__attribute__((reqd_work_group_size(4, 1, 1)))
__kernel void simple_linear_regression(__constant float *restrict x,
                         __global float *restrict y)
{-||-}
```

```
//SLR_k_g_CU implementing compute unit replication.
__attribute__((reqd_work_group_size(10000, 1, 1)))
__attribute__((num_compute_units(4)))
__kernel void simple_linear_regression(__constant float *restrict x,
                         __global float *restrict y)
{
 // Weights vector
 __local float weights[4];
 weights[0] = -0.55395846;
 weights[1] = 1;
 weights[2] = 10;
        weights[3] = 8.94836441;
 // Get index of the work item
 int index = get_global_id(0);

 // Creating the design vector
 __local float features[4];
 features[0] = 1;
 features[1] = x[index];
 features[2] = sin(3 * x[index]);
        features[3] = exp(-1 * x[index] * x[index]);

 // Executiong our inference
 y[index] = features[0] * weights [0] + features[1] * weights [1] +
features[2] * weights [2] + features[3] * weights [3];}

//SLR_k_g_SIMD implementing SIMD kernel vectorization.
__attribute__((reqd_work_group_size(10000, 1, 1)))
__attribute__((num_simd_work_items(4)))
__kernel void simple_linear_regression(__constant float *restrict x,
                         __global float *restrict y)
{
 // Weights vector
 __local float weights[4];     //Maybe those will need to be in vector
data type float16
 weights[0] = -0.55395846;
 weights[1] = 1;
 weights[2] = 10;
        weights[3] = 8.94836441;
 // Get index of the work item
 int index = get_global_id(0);

 // Creating the design vector
 __local float features[4];     //Maybe those will need to be in
vector data type float16
 features[0] = 1;
 features[1] = x[index];
 features[2] = sin(3 * x[index]);
        features[3] = exp(-1 * x[index] * x[index]);

 // Executiong our inference
 y[index] = features[0] * weights [0] + features[1] * weights [1] +
features[2] * weights [2] + features[3] * weights [3];}
```

```
//SLR_k_g_SIMD_CU implementing compute unit replication and SIMD.
__attribute__((reqd_work_group_size(10000, 1, 1)))
__attribute__((num_compute_units(2)))
__attribute__((num_simd_work_items(2)))
__kernel void simple_linear_regression(__constant float *restrict x,
                                       __global float *restrict y){
 // Weights vector
 __local float weights[4];
 weights[0] = -0.55395846;
 weights[1] = 1;
 weights[2] = 10;
        weights[3] = 8.94836441;
 int index = get_global_id(0);  // Get index of the work item
 // Creating the design vector
 __local float features[4];
 features[0] = 1;
 features[1] = x[index];
 features[2] = sin(3 * x[index]);
        features[3] = exp(-1 * x[index] * x[index]);
 // Executiong our inference
 y[index] = features[0] * weights [0] + features[1] * weights [1] +
features[2] * weights [2] + features[3] * weights [3];}
//SLR_k_g_V_D implementing vector data type kernel vectorization.
__attribute__((reqd_work_group_size(10000, 1, 1)))
__kernel void simple_linear_regression(__constant float4 * restrict x,
                                       __global float4 * restrict y){
 // Weights vector
 __local float4  weights;
 weights = (float4) (-0.55395846, 1, 10, 8.94836441);
 // Get index of the work item
 int index = get_global_id(0);
 // Creating the design vector
 __local float4 features[4];
 features[0] = (float4) (1, x[index].s0, sin(3 * x[index].s0), exp(-1
* x[index].s0 * x[index].s0));
 features[1] = (float4) (1, x[index].s1, sin(3 * x[index].s1), exp(-1
* x[index].s1 * x[index].s1));
 features[2] = (float4) (1, x[index].s2, sin(3 * x[index].s2), exp(-1
* x[index].s2 * x[index].s2));
    features[3] = (float4) (1, x[index].s3, sin(3 * x[index].s3),
exp(-1 * x[index].s3 * x[index].s3));
 // Executiong the inference
 y[index].s0 = features[0].s0 * weights.s0 + features[0].s1 *
weights.s1 + features[0].s2 * weights.s2 + features[0].s3 *
weights.s3;
 y[index].s1 = features[1].s0 * weights.s0 + features[1].s1 *
weights.s1 + features[1].s2 * weights.s2 + features[1].s3 *
weights.s3;
 y[index].s2 = features[2].s0 * weights.s0 + features[2].s1 *
weights.s1 + features[2].s2 * weights.s2 + features[2].s3 *
weights.s3;
 y[index].s3 = features[3].s0 * weights.s0 + features[3].s1 *
weights.s1 + features[3].s2 * weights.s2 + features[3].s3 *
weights.s3;}
```

Listing 2.5: simple_linear_regression (SLR_*).cl OpenCL kernel code.

A.3 Four-Layer Neural Network FPGA Kernel Code

```
////////////////////////////////////////////////////////////////////
//Created by Samyuel Danyo on 14/03/2018
#ifndef four_layer_nn_host
#define four_layer_nn_host
#define cl_khr_fp64
#endif

#ifdef cl_khr_fp64
    #pragma OPENCL EXTENSION cl_khr_fp64 : enable
#elif defined(cl_amd_fp64)
    #pragma OPENCL EXTENSION cl_amd_fp64 : enable
#else
    #error "Double precision floating point not supported by OpenCL
implementation."
#endif

//Channels can be used only with altera FPGA OpenCL SDK, for other
SDKs
//channels need to be replaced with pipes
//Channeled kernels cannot be vectorized with SIMD, as that may result
in
//Multiple CU trying to write the same channel.

channel float OUT_HL_1;
channel float OUT_HL_2;

#define INPUT_N 784
#define HL_1_N 500
#define HL_2_N 150
#define OL_N 10
#define RELU(x) (x > 0 ? x : 0)
#define SIGMOID(x) (1.0f / (1 + exp(-x)))
#define Y(x,y) Y[x*OL_N + y]

__kernel
void HL_1_RELU(__constant float *restrict X_test,
     __constant float *restrict W_HL_1,
     __constant float *restrict B_HL_1)
{
 // Get index of the work item (offset within the NDRange)
 int img = get_global_id(0);

 __local float activation;
 for(int n = 0; n < HL_1_N; n++)
 {
  activation = 0.0f;
  for(int f = 0; f < INPUT_N; f++)
   activation += X_test[img*INPUT_N + f] * W_HL_1[f*HL_1_N + n];
  activation += B_HL_1[n];
  activation = RELU(activation);
  write_channel_altera(OUT_HL_1, activation);}
}
```

```
__kernel
void HL_2_SIG(__constant float *restrict W_HL_2,
    __constant float *restrict B_HL_2)
{
 // The offset within the NDRange (which image is processed is implied
via the channel)
 __local float activation;
 __local float features[HL_1_N];
 // n for neuron index, f for feature index
 for(int f = 0; f < HL_1_N; f++)
  features[f] = read_channel_altera(OUT_HL_1);
 for(int n = 0; n < HL_2_N; n++)
 {
  activation = 0.0f;
  //#pragma unroll
  for(int f = 0; f < HL_1_N; f++)
  {
   activation += features[f] * W_HL_2[f*HL_2_N + n];
  }
  activation += B_HL_2[n];
  activation = SIGMOID(activation);
  write_channel_altera(OUT_HL_2, activation);}
}

__kernel
void OL_SOFTMAX(__constant float *restrict W_OL,
      __constant float *restrict B_OL,
      __global float *restrict Y)
{
 int img = get_global_id(0);
 __local float activation;
 __local float features[HL_2_N];
 // n for neuron index, f for feature index
 for(int f = 0; f < HL_2_N; f++)
  features[f] = read_channel_altera(OUT_HL_2);
 for(int n = 0; n < OL_N; n++)
 {
  activation = 0.0f;
  for(int f = 0; f < HL_2_N; f++)
  {
   activation += features[f] * W_OL[f*OL_N + n];
  }
  activation += B_OL[n];
  Y(img,n) = activation;}
}
```

Listing 3.1: Neural Network OpenCL kernel code.

A.4 FPGA Machine Learning Accelerator

Project Brief (COMP3200: Part III Individual Project)

Samyuel Danyo

Supervisor: Prof. Mark Zwolinski

We used to live in a hardware driven world where most of the innovation work was focused on better physical properties of technology. However, for a long time now this has changed and the major innovation progress happens in software. This is due to physical limitations, as well as, cost and customer incentive. In the past, if a person had bought a device, the only way to get the next generation was to buy the new, better device. Obviously, this slows down the innovation process, both - in research and design, as well as, propagation through society. All of this results in cumbersome hardware innovation cycle. On the other hand, with software - limitations are much harder defined and easier overcome, there is not manufacturing time or big physical costs and literally all users can download the new program on their, already owned, device in a day.

Another thing, that Steve Jobs already mentions in the distant 1995, is that exactly due to the already mentioned facts, while for most things, hardware included, the difference between the best and average(bad) may be in the range of ten times, in software it is a hundred to one.

Nevertheless, despite all of the points made above, in general, we still make either general purpose hardware or software for a particular hardware. I believe the opposite approach is more logical – making hardware, customized for a particular software. This can drive much faster innovation, computing speed and power efficiency, because there won't be unnecessary parts in the system.

One solution to the problem is designing ASIC (application-specific integrated circuit). However, they still introduce the above-mentioned hardware limitations. That is why, I believe a better solution could be using FPGAs, which can be specialized for a particular software but also are programmable in near real time, meaning that they can easier evolve with the application.

My project is focused particularly on Machine Learning acceleration using FPGAs, which I believe can be an amazing match, because of a few points:

First, current innovation in neural networks is extremely fast, making the design of customized hardware hard. FPGAs can give the best solution to that, because of their reconfigurability.

Second, Machine Learning and Big Data Analysis as a whole is all about data processing, which requires a lot of parallel and vector-based computing and FPGAs are very good at that thanks to their architecture and very high internal memory bandwidth.

Third, the high-speed internal data processing and the lack of unnecessary hardware can give an increased power efficiency, compared to regular GPUs an CPUs.

Last but not least, the specialized hardware can give us the tools to build a system, much better representing our own brains. It will have separate, highly efficient, parts for handling different types of tasks, ultimately delivering parallel functionality and low power usage. Another benefit of outsourcing machine learning in a system can give us a level of abstraction between the main CPU, which would be connected to the Internet and so vulnerable and most of the data processing, resulting in better security.

The aim of the project is to implement machine learning algorithms on a FPGA and compare their speed/power efficiency to running on a CPU/GPU. Next point would be to build interface software for communication between the FPGA and a host computer in a step to building the system, described above.

My target for the first semester is a working prototype algorithm implemented on a FPGA, including test results on actual speed/power efficiency improvement against CPU/GPU, as well as, research on best matches between algorithms and data types. In the second semester I would expect to finish implementation of one or a group of algorithms on the FPGA and create an interface software for offloading the machine learning to it.

A.5 Project Management.

A.5.1 Gantt Project Schedule Chart

Table A.5.1: Gantt Chart illustrating the planned project's schedule. The milestones of implementation, testing and project completion are marked with stars.

Month	**Oct**				**Nov**				**Dec**					**Jan**				**Feb**				**Mar**				**Apr**					**May**
Week	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
Project Intro	▓	▓	▓	▓					▓	▓																					
Introduction	░	░																													
Project Management			░	░					░	░																					
Supervisor Meetings	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓				▓	▓	▓	▓	▓	▓	▓	▓	▓	▓					▓	▓	
Background			▓	▓	▓	▓	▓	▓						▓	▓	▓	▓	▓													
Related Work			░	░	░	░								░	░	░															
Theory			░	░	░	░	░	░						░	░	░	░	░													
Architecture Design			▓	▓	▓	▓								▓	▓																
Implementation					▓	▓	▓	▓						▓	▓	▓	▓	▓	▓	▓	▓★										
SLR					░	░	░	░																							
4-Layer-NN														░	░	░	░	░	░	░	░										
Verification							▓	▓									▓	▓	▓	▓	▓										
Testing															▓	▓	▓					▓	▓	▓★							
Report Writing									▓	▓																▓	▓	▓	▓	▓★	

A.5.2 Gantt Chart of The Actual Project Flow

Table A.5.2: Gantt Chart illustrating the actual project flow. The milestones of implementation, testing and project completion are marked with stars.

Month	**Oct**				**Nov**				**Dec**					**Jan**				**Feb**				**Mar**					**Apr**				**May**
Week	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
Project Intro	🟩	🟩	🟩	🟩					🟩	🟩								🟩	🟩						🟩						
Introduction	▨	▨																													
Project Management			▨	▨					▨	▨								▨	▨						▨						
Supervisor Meetings	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩				🟩								🟩					🟩	🟩		
Background			🟩	🟩	🟩	🟩	🟩	🟩										🟩	🟩	🟩				🟩	🟩						
Related Work			▨	▨	▨	▨												▨	▨												
Theory			▨	▨	▨	▨	▨	▨										▨	▨	▨				▨	▨						
Architecture Design			🟩	🟩	🟩	🟩												🟩									🟩				
Implementation					🟩	🟩	🟩	🟩										🟩	🟩	🟩	🟩					🟩	🟩 ★				
SLR					▨	▨	▨	▨												▨	▨										
4-Layer-NN																		▨	▨	▨	▨					▨	▨				
Verification							🟩	🟩											🟩	🟩	🟩	🟩	🟩				🟩	🟩	🟩		
Testing																								🟩			🟩	🟩	🟩 ★		
Report Writing									🟩	🟩								🟩		🟩				🟩	🟩				🟩	🟩 ★	

A.5.3 Risk Assessment

Table A.5.3: Project Risks Assessment.

Problem	Loss	Prob.	Risk	Plan
Insufficient Literature Found	3	2	**6**	Field-related literature will be searched. Focus will shift to experimenting and analysis.
Design Barriers	4	2	**8**	Thorough research of both – theory and related project will be done. The design will be aided by in-depth analysis and goals setting.
Insufficient Software Development Resources	5	1	**5**	Based on research and design goals, the needed resources will be stated in advance. In critical cases suitable replacements will be chosen**.**
Insufficient Hardware Resources	4	3	**12**	Based on research and design goals, the needed resources will be stated in advance. In critical cases suitable replacements will be chosen**.**
Software Development Bottlenecks	4	2	**8**	With the help of frequent supervisor meetings and Project Management reassessment, either solutions will be found or the goals of the project will be changed accordingly.
Unreliable Testing Approach	5	2	**10**	Based on thorough research, the optimal solution will be chosen. If it is not available, a sufficient solution and error handling procedures will be incorporated.
Working Machine Failure.	5	2	**10**	Project back-up will be done. With the help of frequent supervisor meetings, a solution to any setback will be found.
Illness	3	3	**9**	Healthy work-life balance will be kept. Sufficient time "cushions" will be planned into the project's schedule.

A.5.4 Skills Audit

Ratings are based on initial research and previous background. Focused training is done to handle any inadequate knowledge.

Table A.5.4: Skills Audit.

Skill	Rating (1-5) [Based on Project Requirements]
Technical Skills	
C-based Programming	4
OpenCL Proficiency	3
UNIX	5
Shell Scripting	5
Python Coding	4
Intel FPGA SDK Proficiency	2
FPGA	3
Software Verification	4
Testing Approaches	2
Machine Learning	4
Communication & Analysis Skills	
Research	5
Writing	3
Analytical Thinking	5
Project Management	4
Risk Management	4
Evaluation	3
Design	5

A.6 Design Archive Content

Table A.6: Design Archive Contents.

Design Archive Contents	
File	**Description**
SLR	**Simple Linear Regression Algorithm**
Source	**Source Code**
SLR_*	**Variations**
main.cpp	Host Code
slr*.cl	Kernel Code
report.html	Kernel Implementation Report
lib	**Report Library**
Emulation	Emulation Binaries
SLR_*	**Variations**
host_e	Host Code Binary (Linux, x86)
*.aocx	Kernel Binary (emul)
FPGA	**Emulation Binaries**
SLR_*	**Variations**
host_e	Host Code Binary (ARM)
*.aocx	Kernel Binary (emul)
Developer	**Helper Files**
Emulation.sh	Emulation Wrapper
plot.py	Plotting Script
Makefile_main	Makefile for Host Code Compilation
Makefile_main_c	Makefile for C++ Benchmark Compilation
README	Development Instructions
4-Layer-NN	**4-Layer-NN MNIST Classifier**
Source	**Source Code**
DATA_H	**Model Data & Test Dataset**
main.cpp	Host Code
4_layer_nn.cl	Kernel Code
report.html	Kernel Implementation Report
lib	**Report Library**
Emulation	**Emulation Binaries**
host_e	Host Code Binary (Linux, x86)
4_layer_nn_e.aocx	Kernel Binary (emul)
FPGA	**Emulation Binaries**
host_e	Host Code Binary (ARM)
4_layer_nn.aocx	Kernel Binary (emul)
Developer	**Helper Files**
Makefile_main	Makefile for Host Code Compilation
Makefile_main_c	Makefile for C++ Benchmark Compilation
README	Development Instructions