

**Determination of the Usability of FPGA Technology
to Accelerate Option Pricing Algorithms**

Matthew James Carter
201371920

A DISSERTATION

Submitted to

The University of Liverpool

In partial fulfilment of the requirements
for the degree of
MASTER OF SCIENCE

20th September 2019

Abstract

High-performance computing is an increasingly important topic in the world of finance, particularly in the field of options pricing. As CPU performance plateaus, it is imperative that financial institutions look towards alternative hardware, such as Field Programmable Gate Arrays (FPGAs), to satisfy their demands. This thesis explores the feasibility of using FPGAs to accelerate options pricing libraries. Specifically, we illustrate efficient implementations of the Binomial Trees and Monte Carlo options pricing algorithms on an FPGA. We demonstrate how mathematical approximations such as the Taylor expansion can be used to further accelerate these algorithms. Furthermore, we compare FPGA implementations against a CPU implementation in order to determine whether the FPGA implementations produce an accurate result whilst minimising the energy use and runtime of the algorithm.


A proof of concept was developed on a Zynq-7020 FPGA and compared against a Cortex-A9 CPU. Following this, the algorithms were ported to an Alveo u200 data centre card and compared against an Intel Xeon E5649. We achieved a speedup of as much as 22x on the Zynq-7020 and as much as 20x on the Intel Xeon E5649. The FPGA implementations of the Binomial Trees algorithms were accurate to 4 decimal places, whereas the Monte Carlo algorithm was accurate to 5 decimal places. The energy comparisons were largely inaccurate, due to a lack of appropriate performance counters on these CPUs, and therefore this analysis was inconclusive.

Student Declaration

I hereby certify that this dissertation constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions or writings of another.

I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,

A handwritten signature in black ink on a light blue rectangular background. The signature appears to read "M Carter" in a cursive, slightly stylized script.

Matthew Carter

Abbreviations

ASIC - Application Specific Integrated Circuit

CPU – Central Processing Unit

FPGA – Field Programmable Gate Array

HDL – Hardware Descriptive Language

HFT – High-Frequency Trading

HLS – High-Level Synthesis

HPC – High-Performance Computing

IP – Intellectual Property

RTL - Register Transfer Level

TDP – Thermal Design Power

Table of Contents

1. Introduction.....	1
2. Background	2
2.1 Field Programmable Gate Arrays	2
2.2 Field Programmable Gate Arrays in Finance.....	2
2.3 Resource Requirements	4
3. Aims and Objectives	5
3.1 Statement of Hypotheses.....	5
3.1.1 Null Hypothesis	5
3.1.2 Alternative Hypothesis.....	5
3.2 Aims.....	5
3.2.1 Primary Aims	5
3.2.2 Secondary Aims	5
3.3 Objectives	5
4. Design	6
4.1 Project Management	6
4.2 Experimental Design.....	6
4.3 Experiments to be Performed.....	7
4.4 Analysis of Results	8
4.5 Changes from Original Design	8
5. Realisation.....	9
5.1 Zynq-7020 and Alveo u200 Implementation.....	9
5.2 Optimising Code for FPGA	11
5.3 Interesting Aspects of Implementation	13
5.3.1 Binomial Tree – European and American Options.....	13
5.3.2 Combined Binomial Tree.....	15
5.3.3 Monte Carlo – European Options	15
6. Results and Discussion	16
6.1 Accuracy	16
6.2 Time Performance of Implementations	17
6.2.1 Binomial Tree	17
6.2.2 Combined European and US Binomial Tree	20
6.2.3 European Monte Carlo	20
6.3 Energy Performance of Implementations	22
6.2.1 Cortex-A9 versus Pynq-Z2	22
6.2.2 Intel Xeon E5649 versus Alveo u200	22
6.4 Floating Point versus Fixed Point.....	23
6.5 Comparison of Algorithms	25

7. Evaluation	26
10. Conclusions.....	29
10.1 Summary	29
10.2 Future Work.....	29
11. References.....	30
12. Appendix 1 – Equations.....	34
12.1 Black Scholes Formula	34
12.2 Heston Formula.....	34
13. Appendix 2 - Pseudocode	35
13.1 Monte Carlo Black-Scholes	35
13.2 Tree-Based Solver Pseudocode.....	36
14. Appendix 3 – Original Project Design.....	37
14.1 Project Management	37
14.2 Experimental Design.....	37
14.3 Experiments to be Performed.....	37
14.4 Analysis of Results	38
14.5 FPGA Options Pricing Library	38
15. Appendix 4 - Trello Board	39
16. Appendix 5 – Zynq-7020 and Alveo u200 Design Flow	40
16.1 Zynq-7020.....	40
16.2 Alveo u200.....	42
17 Appendix 6 – Supporting Figures	44
17.1 European Binomial Tree	44
17.2 US Binomial Tree	45
17.3 Combined Binomial Tree.....	46
17.4 Monte Carlo	46
18. Appendix 7 – User Guide	47
18.1. Introduction.....	47
18.1.1 Overview.....	47
18.1.2 Requirements	47
18.2. Pynq-Z2.....	48
18.2.1 Binomial Tree	49
18.2.2 Monte Carlo	50
18.3. Alveo u200.....	51
18.3.1 Binomial Tree	53
18.3.2 Monte Carlo	54
18.4. References.....	55
19. Appendix 8 – Project Log	56

1. Introduction

The purpose of this project is to determine the usability of FPGAs to accelerate numerical libraries. Specifically, we will explore how FPGAs can be used to accelerate option pricing models. Over the last decade, a significant amount of research into the application of FPGAs in financial services has been conducted. This research has mainly focused on comparing an FPGA implementation against a CPU implementation of the same algorithm. As identified by [1], it is important to not only compare CPU implementations against FPGA implementations, but also compare different FPGA implementations against each other. By doing so, investment banks can make more informed decisions about the algorithms that they implement to ensure that they get accurate results as quickly as possible whilst minimising energy consumption.

Throughout this project, we implemented binomial tree algorithms to calculate the price of both European and American options. In addition, a Monte Carlo algorithm was implemented to calculate the price of European options. These algorithms were first implemented on a CPU following which the execution time, energy use and accuracy were measured. An initial proof of concept was then developed and executed on a Zynq-7020 FPGA. Finally, all algorithms were ported to run on an Alveo u200 FPGA. The execution time, energy use and accuracy of both FPGA implementations were recorded after this. Due to the low resource usage of the binomial trees algorithm, an additional IP was created that could price European and American options in parallel.

The FPGA implementations of binomial tree algorithms were accurate to three decimal places, whereas the Monte Carlo algorithm was accurate to four decimal places. The European binomial trees algorithm ran 7x faster on the Zynq-7020 in comparison to the Cortex-A9 and 3x faster on the Alveo u200 in comparison to the Intel Xeon E5649. The US binomial trees algorithm ran 20x faster on the Zynq-7020 in comparison to the Cortex-A9 and 1.6x faster on the Alveo u200 in comparison to the Intel Xeon E5649. The combined binomial trees algorithm ran 22x faster on the Alveo u200 than on the Intel Xeon E5649. The Monte Carlo algorithm ran 1.1x faster on the Zynq-7020 in comparison to the Cortex-A9 however, it ran 2x slower on the Alveo u200 in comparison to the Intel Xeon E5649. We have suggested several improvements to this algorithms design which optimise it further to run on an FPGA. Due to the lack of performance counters on these CPUs, the energy calculations were inaccurate and therefore our conclusions were inconclusive.

2. Background

2.1 Field Programmable Gate Arrays

An FPGA is a programmable logic device consisting of a matrix of configurable logic blocks connected via programmable interconnects [2, 3]. FPGAs are a form of reprogrammable hardware that can be configured by a user to perform a desired function [2, 3]. This makes them much more versatile when compared to similar hardware such as ASICs that are designed to perform specific functions.

An FPGA consists of three components: the programmable logic component which expresses a logic function; the programmable I/O component which provides an external interface, and the programmable interconnects which connects the different parts of the circuit together [4]. The power of FPGAs comes from their ability to simultaneously perform parallel functions and digital clock management functions that supply several high-speed clocks [5].

Due to their reprogrammable and parallel nature, FPGAs have become a topic of interest in several domains ranging from consumer electronics to HPC. A molecular dynamics OpenCL kernel was proposed in [6] that used 80% of the FPGAs resources and achieved a speed-up of 18x in comparison to a CPU implementation. Several machine learning kernels have been accelerated using FPGAs as shown in [7, 8, 9, 10, 11]. These kernels can achieve as much as a 61-times speed-up when compared to a CPU implementation. The field that we consider is reviewed in section 2.2.

The design flow for FPGAs varies drastically in comparison to a software design flow. FPGAs are typically programmed using a HDL such as Verilog [12]. These languages have a steep learning curve and take a long time to master [12]. An alternative is to write HLS which generates hardware modules from a high-level language such as C. In this case, the designer creates a module written in C/C++ and compiles it as a software program to verify it is working correctly. Following this, the designer can add directives to describe how sections of the code should behave on hardware. The HLS code can then be converted to RTL code using software such as Vivado HLS [4, 13]. A designer can significantly reduce development time by developing reusable modules called IP [4]. The hardware and software used throughout this thesis are discussed in section 2.3.

2.2 Field Programmable Gate Arrays in Finance

HPC is becoming an increasingly important topic within the world of finance. This is largely because, following the 2008 financial crisis, financial institutions are now required to deliver valuation and risk simulation results to regulatory bodies. As part of this, institutions are required to price exotic derivatives using appropriate market models [14]. For many of these models, no closed-form solution exists, and the solution must be approximated; this is a computationally expensive process and can last several days on state-of-the-art systems [14, 15]. HFT has added additional computational demands as investors aim to minimise the runtime of algorithms that are responsible for pricing assets and placing buy and sell orders [16]. As chip manufacturers find it increasingly hard to keep up with Moore's law, it is imperative that financial institutions look towards hardware to satisfy their demands [17].

In finance, an option is a contract that gives the owner the right to buy or sell an underlying asset within a specific timeframe. There are two types of option contract: a call option which gives the holder the right to buy the asset at a stated price; and, a put option which gives the holder the right to sell the asset at a specified price [18]. Several forms of options, such as European and American, also exist; each of these forms place different restrictions on when a holder can exercise their options and therefore changes how they are priced [18]. A European option is a point in time instrument as it can only be exercised on the expiration date. American options are a continuous time instrument, as they can be exercised at any point up to and including the expiration date.

There are several methods for pricing options, the most popular of which is the Black-Scholes pricing model. The Black-Scholes model is a differential equation that calculates the value of an option using the current stock price, expected dividends, the strike price, expected interest rates, time to maturity, and expected volatility [18, 19]. The Heston model is another mathematical model for pricing options. In this model, volatility is stochastic in nature whereas in the Black-Scholes model it is constant [20]. Both models are calculated at a point-in-time and therefore only apply to European options. Variants of these models exist for American options, but because of their continuous nature, the formulae are no longer expressed in closed form [21]. The formulae for both models are presented in *Appendix 1*.

There exist several methods that can be used to calculate the price of several different options. For example, the Monte Carlo method can be used to approximate solutions to the Black-Scholes and Heston models for both European and American options [22]. The Monte Carlo method is used to simulate stochastic processes such as the underlying asset price in the Black-Scholes model, and the underlying asset price and volatility in the Heston model [23]. The Black-Scholes Monte Carlo algorithm beings by generating n random paths which simulate the price movements of some underlying asset; these paths typically follow Geometric Brownian Motion. Each path is then discounted in order to determine its payoff. The price of the option is determined by calculating the average of all payoffs [24]. The pseudocode for the Black-Scholes Monte Carlo algorithm is presented in *Appendix 2*.

Tree-based pricing models also offer a solution for pricing complex options, such as American options. The most popular based tree-based method for pricing options is the binomial tree model as it is simple and efficient [25, 19]. This model works by discretising both time and the price of the underlying asset [25]. A step away from the root of the tree towards a leaf represents one time-step and each node represents the price of the underlying asset at that time-step. After generating the binomial tree, the value of the option at each time step can be calculated. The value of the option at the root of the node can then be calculated by combining each node's option value with the probability that an asset appreciates [25, 19]. The pseudocode for European and US binomial tree solvers is given in *Appendix 2*.

When calculating the value of a financial asset, it is important to be as precise as possible; a calculation that is out by even a few pence could result in significant losses as assets are typically traded in large volumes. Traditionally, option pricing calculations make use of floating-point arithmetic, which is costly and takes many clock cycles to complete. This is

because floating-point numbers require extra bits to express a number, making them costly in terms of storage. Moreover, extra operations are required to compute the differences between the exponential and mantissa values [26, 27]. These costs can be mitigated with the use of fixed-point arithmetic; however, this can reduce the accuracy of the solution [3, 27, 28]. The trade-offs between speed-up, energy usage and accuracy should be carefully considered when implementing fixed-point arithmetic.

A large amount of research into the usability of FPGAs to accelerate options pricing calculations has been conducted in recent years. [29] implemented a hybrid CPU-FPGA accelerated Monte Carlo options pricing algorithm using the Heston model. Their implementation ran 2x faster than a CPU-only implementation whilst using 89% less energy. A finite difference solver for the Black-Scholes options pricing method was presented in [30] with a speed-up of 8x. A Least-Squares Monte Carlo algorithm for pricing American options was presented in [31] which achieved a speed-up of 20x with significant energy savings. It should be noted that these results are affected by the CPU and FPGA used during the comparison.

2.3 Resource Requirements

The project began by developing a proof of concept, where we implemented algorithms on a Pynq-Z2 development board. The purpose of this was to quickly port the algorithms from CPU to FPGA and determine whether there was any benefit to running these algorithms on an FPGA. The Pynq-Z2 board brings the productivity benefits of Python to FPGA development, which allowed us to quickly develop the proof of concept. Moreover, the resources on the Pynq-Z2 board are more than enough for an initial proof of concept. This board has a 650-MHz dual core Cortex-A9 processor, 512MB DDR3 memory and a ZYNQ7020 FPGA. The FPGA consists of 53,200 look-up tables, 106,400 flip-flops and 220 DSP slices [32].

One of the intentions of this research is to inform investment banks of the benefits to using FPGAs to accelerate options pricing libraries. Because of this, after developing the proof of concept, we ran the algorithms on a larger FPGA that is more representative of the hardware that an investment bank would use. For this stage, we ran the algorithms on an Alveo u200 data centre card which was installed in a server (named livfpga) at the University of Liverpool. This card contains an XCU200 FPGA that consists of 892,000 look-up tables, 1,831,000 flip-flops and 5,867 DSP slices. A 2.53GHz six-core Intel Xeon E5649 processor was installed alongside the Alveo u200 [33].

The project also has several software requirements. Firstly, when developing for the Pynq-Z2, we used Vivado HLS to write our HLS code, generate RTL code, and create IPs. In addition, we used the Vivado Design Suite to generate block diagrams and create bitstreams for the Pynq-Z2. We also need the Pynq-Z2 board image which installs the development environment on the board. When developing for the Alveo u200, we used the SDAccel Development Environment which allowed us to write HLS code and generate bitstreams for this device [34, 35].

3. Aims and Objectives

3.1 Statement of Hypotheses

3.1.1 Null Hypothesis

1. FPGA implementations of options pricing algorithms will run slower than CPU implementations.
2. FPGA implementations of options pricing algorithms will be less energy efficient than CPU implementations.

3.1.2 Alternative Hypothesis

1. FPGA implementations of options pricing algorithms will run faster than CPU implementations.
2. FPGA implementations of options pricing algorithms will be more energy efficient than CPU implementations.

3.2 Aims

3.2.1 Primary Aims

1. Determine the speed-up, accuracy and energy efficiency of FPGA implementations of Monte Carlo and binomial tree algorithms to price European options in comparison to CPU implementations.
2. Compare the FPGA algorithms to determine which yields the greatest speed-up and accuracy whilst minimising energy use.
3. Analyse floating-point versus fixed-point arithmetic operations to determine the trade-off between speed-up, accuracy and energy usage.

3.2.2 Secondary Aims

1. Implement both Monte Carlo and binomial trees algorithms to price American options and perform the analysis described above.
2. Develop an FPGA accelerated options pricing library consisting of the algorithms explored throughout this project.
3. Modify the implemented algorithms to price options under the Heston model and perform the analysis described above.
4. Extend the analysis to incorporate exotic options, such as Asian options.

3.3 Objectives

1. Develop a CPU implementation of the Monte Carlo and binomial trees algorithms to approximate the solution of the Black-Scholes option pricing model for European options.
2. Develop an FPGA implementation of the Monte Carlo and binomial trees algorithms to approximate the solution of the Black-Scholes option pricing model for European options.
3. Measure and record the run time, accuracy and energy use of these algorithms for both the CPU and FPGA implementations.
4. Modify the algorithms to use fixed-point, rather than floating-point arithmetic and measure the speed-up, energy use and accuracy of these implementations.

4. Design

The original project design, as proposed in Assignment 1, can be found in *Appendix 3*. Any changes to the original design will be discussed in section 5.5.

4.1 Project Management

A Github repository [36] was setup to maintain a backup of all the code written throughout this project. This had the added benefit of ensuring that new parts of the system integrate seamlessly into the old system.

A backup of the Pynq-Z2 board was also regularly maintained, just in case the operating system had to be reinstalled. The contents of the board were regularly backed up to Google Drive. Similarly, all assignments and other important documents were also backed up to Google Drive.

Trello [37], an online project management tool, was used to manage the workload throughout this project. All tasks were entered on a Trello board and were labelled as either: to do, complete, in progress or on hold. A snapshot of this is given in *Appendix 4*.

4.2 Experimental Design

The project began by developing a proof of concept for the mentioned algorithms on a Pynq-Z2 development board. This involved implementing a Monte Carlo algorithm and binomial trees algorithm to approximate the solution to the Black-Scholes formula for pricing European options. The binomial trees algorithms run throughout this stage have a maximum height of 30,000 due to the limited BRAM available on the device. When performing Monte Carlo simulations, we only stored the simulated steps for each path on the device. This allowed us to perform large simulations, for example simulating 1,000,000 paths with 1,000 steps, during this stage.

Following this, the algorithms were ported to run on an Alveo u200 card. Our algorithms should run faster on the FPGA installed on this card due to it having more resources and a higher clock speed. Despite the Alveo u200 having more BRAM than the Pynq-Z2, we will keep the maximum tree height at 30,000. This is because there is little to no benefit to having a larger tree height as shown in section 6.1. Similarly, the size of our Monte Carlo simulations will remain the same. The output of each algorithm on the Pynq-Z2 and Alveo u200 was recorded and compared to ensure that the algorithms still worked correctly.

When conducting our experiments, each algorithm was run a total of 5 times with the same starting parameters. This allowed us to determine the average time it takes the algorithm to run, as well as identify the maximum and minimum run times. The experiments to determine accuracy were performed once. Due to the lack of appropriate performance calculators, the energy measurements were also only performed once.

4.3 Experiments to be Performed

Experiments were performed to determine the run time, accuracy and energy use of the mentioned algorithms. The total run time of our algorithms were measured and, where possible, times relating to data transfer and kernel runtime were also measured. On the Pynq-Z2 board, these times were measured using Python’s datetime function [38]. When working with the Alveo u200, the kernel runtime was measured using OpenCL’s event profiling [39]. Timings for the CPU implementation, and within the host code for the Alveo u200 were measured using the `gettimeofday()` function in C++ [30].

The accuracy of the FPGA implementations was determined by comparing their output against their respective CPU implementations. The outputs of both the Pynq-Z2 and Alveo u200 implementations were also compared to ensure that no accuracy was lost when moving between the two platforms. At the beginning of the project a program was developed to calculate the closed form solution to the Black-Scholes formula. To assess the quality of the solution from the European Monte Carlo and European binomial trees algorithms, we can compare them against the closed form solution. This will form part of our analysis when comparing the algorithms against one another. We cannot do the same for the US versions of these algorithms as no closed form solution exists.

Algorithm	Closed Form Solution Exists?
European Monte Carlo	Yes
European Binomial Tree	Yes
US Monte Carlo	No
US Binomial Tree	No

Table 1: Summary of Algorithms and their Solutions

A USB tester was used to measure the energy usage of the Pynq-Z2 board [41]. A downside to this is that we are measuring the energy usage of the whole board, not just the CPU or FPGA. Due to the age of the CPU installed in the livfpga server it was not possible to determine its energy usage during runtime. We therefore estimate the energy use of the Intel Xeon E5649 by considering the TDP of the processor. A script to determine the energy usage of the Alveo u200 was developed by the primary supervisor of this project. This works by polling the device once per millisecond and measuring its power usage. A Python script was then developed to integrate the power over time and determine the energy use. All energy measurements were made using optimised versions of the code; the binomial trees algorithms used a depth of 30,000 and the Monte Carlo simulations had 1,000,000 paths with 100 steps per path.

4.4 Analysis of Results

After developing the proof of concept, we compared the accuracy, runtime and energy usage of the implemented algorithms on the Cortex-A9 against the Zynq-7020 FPGA. The same comparisons were made when moving from the Zynq-7020 to the Alveo u200 however, we compared the results from the Alveo u200 against the Intel Xeon E5649. We also compared the accuracy of the algorithms on the Zynq-7020 against the accuracy of the algorithms on the Alveo u200. This analysis allowed us to either accept or reject the null hypothesis presented in section 4.1.1. Furthermore, this analysis fulfilled the requirements to satisfy the first primary aim of this project.

Following this, we compared the FPGA implementations against one another. The purpose of this analysis is to inform investment banks about which algorithm they should implement. To do this, we compared the output of both the European binomial tree and European Monte Carlo algorithms against the closed form solution from the Black-Scholes formula. In addition, we also considered the runtime and accuracy of these algorithms. This satisfies the second primary aim.

Finally, we analyse the accuracy, runtime and energy use of algorithms when using fixed-point arithmetic instead of floating-point arithmetic. This will provide insight into the trade-offs when using fixed-point arithmetic and satisfy the third primary aim of the project.

4.5 Changes from Original Design

The initial project design aimed to explore Monte Carlo algorithms for pricing both European and American options under Black-Scholes and Heston conditions. The initial project scope was too wide as it failed to account for the number of skills that had to be acquired and the amount of work that had to be done to optimise algorithms to run on an FPGA. Because of this, the scope of the project was narrowed. The updated design focuses on exploring both Monte Carlo and binomial trees algorithms for pricing European options under Black-Scholes conditions. Exploring these algorithms for pricing American options and under Heston conditions were made desirable aims. Similarly, combining these algorithms into a library was also made a desirable aim.

5. Realisation

5.1 Zynq-7020 and Alveo u200 Implementation

The Zynq-7020 FPGA used throughout this project was installed in a Pynq-Z2 board. Pynq is an open source project, developed by Xilinx, that provides the productivity benefits of Python to embedded systems designers. The designer first develops an IP in Vivado HLS, imports it into a block design in Vivado Design Suite and interacts with the generated bitstream file through a Jupyter notebook [41].

When designing an IP in Vivado HLS, we start by defining a top-level function, which is equivalent to the *main()* function in C. Following this, we must write several *interface* pragmas, which provide our IP with access to global memory. An example of this is shown below in figure 1.

```
#pragma HLS INTERFACE m_axi port=output_r offset=slave bundle=gmem0
#pragma HLS INTERFACE s_axilite port=output_r bundle=control
#pragma HLS INTERFACE m_axi port=S offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=S bundle=control
#pragma HLS INTERFACE m_axi port=K offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=K bundle=control
#pragma HLS INTERFACE m_axi port=T offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=T bundle=control
#pragma HLS INTERFACE m_axi port=D offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=D bundle=control
#pragma HLS INTERFACE m_axi port=r offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=r bundle=control
#pragma HLS INTERFACE m_axi port=v offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=v bundle=control
#pragma HLS INTERFACE m_axi port=type_r offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=type_r bundle=control
#pragma HLS INTERFACE m_axi port=height offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=height bundle=control
#pragma HLS INTERFACE s_axilite port=n_options bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control
```

Figure 1: Defining Interface Pragmas in Vivado HLS

In this case, we have both *m_axi* and *s_axilite* ports. The *m_axi* ports are used for data input and output, whereas the *s_axilite* ports are used for configuring the IP at runtime [42]. With the top-level function and memory interfaces defined, we can write the rest of our algorithm as a normal C/C++ program. Following this, the IP is compiled for the target device. Figure 2 below shows the design flow for the Zynq-7020 and Alveo u200. These are detailed further in appendix 5.

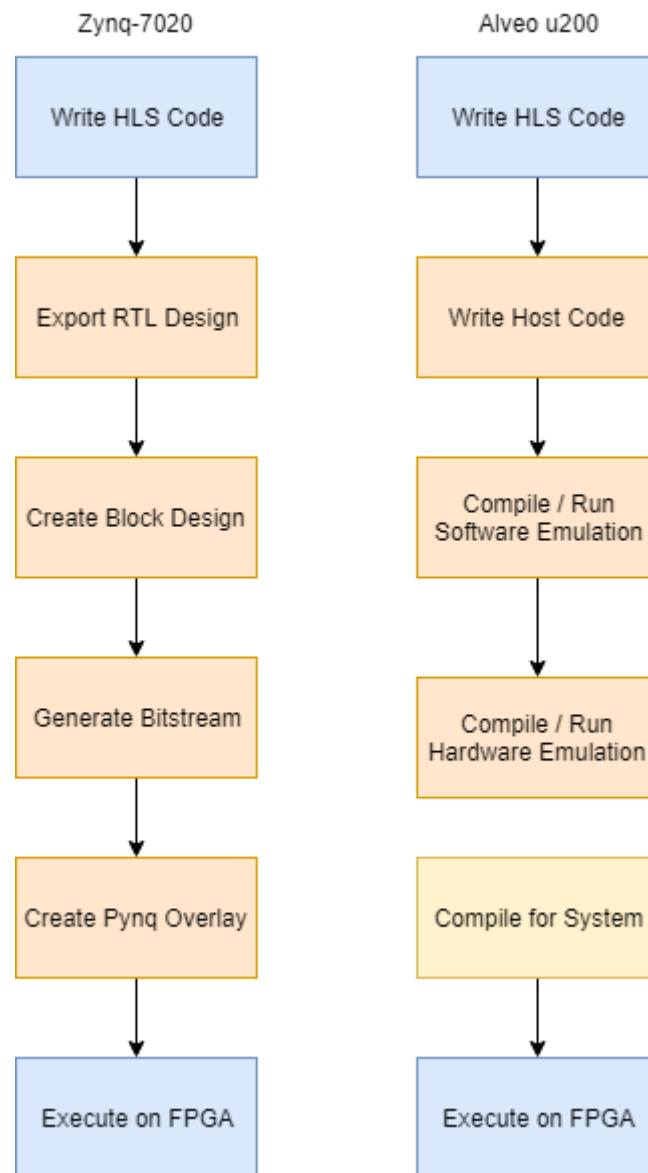


Figure 2: Design Flow of Zynq-7020 and Alveo u200

5.2 Optimising Code for FPGA

Both the Zynq-7020 and Alveo u200 have access to two types of memory: DDR and BRAM. BRAM memory is situated on the device itself, whereas DDR memory is not. To access DDR memory, the FPGA must communicate with it via a DDR channel. This can cause a significant bottleneck to performance if the IP must access DDR memory multiple times whilst it is executing. Therefore, it is best to use BRAM memory whilst the IP is executing [45].

An effective way to take advantage of this is by having a global read, computation and global write phase in the top-level function of the HLS code. We read all values from global memory and store them in BRAM during the global read phase. This reduces the number of times we have to access global memory during computation. As computation occurs, we store the computed values on BRAM. These values are then written back to global memory so they can be accessed by the host. A downside to this method is that the amount of BRAM memory available tends to be a lot less than the amount of DDR memory available; which can significantly limit the size of our algorithms. If this is the case, we can use a technique called buffer read-write. In this case, we set our buffer size to a constant value, say 256. We then read in 256 values from DDR memory, perform 256 computations and store the results in BRAM, finally we do a burst write to place these values in DDR memory. This can be further optimised by writing the computed set of values back to global memory as we are calculating new values. By doing this, we overcome the memory limitations of BRAM whilst overcoming the bottleneck of reading and writing from DDR memory [46].

Once an IP has been optimised for global memory accesses, we can further increase its performance by increasing the local memory bandwidth. In Vivado HLS, each array is assigned a single port which can be used to write data to, or read data from, the array. This can cause bottlenecks if the IP must perform lots of read and write operations on an array. We can use the *array_partition* pragma to split the array into several smaller arrays, which effectively increases the number of ports available for read and write operations. By increasing the number of ports available, we increase the bandwidth for operations on local memory [47].

Loop pipelining and loop unrolling can be used to improve the performance of IP by exploiting parallelism between loop iterations. When executing sequential code, any given iteration in a loop can only execute once the previous iteration is complete. By pipelining a for loop, operations in different iterations of the for loop can be performed concurrently, as shown in figure 3 below [48].

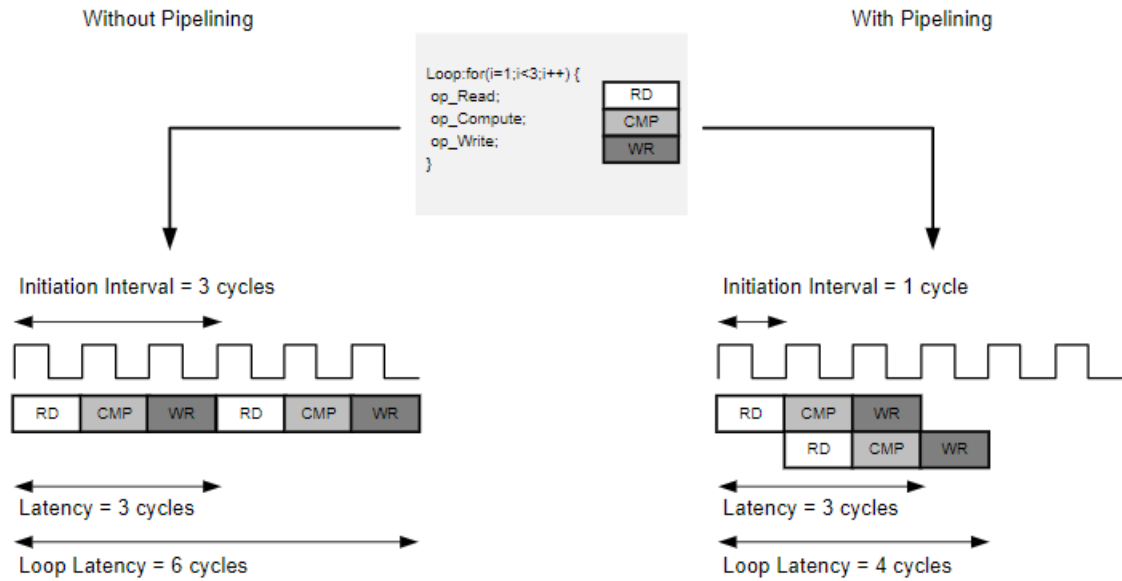


Figure 3: Loop Pipelining [48]

Here we can see that with loop pipelining the number of clock cycles between both read operations has been reduced from 3 to 1. When pipelining for loops, we should ensure that the initiation interval is as low as possible; by doing so, we reduce the number of clock cycles between the start of each consecutive loop iteration. When synthesising an IP design, Vivado HLS/SDAccel will always try to pipeline for loops with an initiation interval of one. Initiation interval can be limited by several factors such as loop carried dependencies and limited resources [48].

By unrolling a loop, we effectively create multiple copies of the loop in the RTL design, which can be executed in parallel. When specifying the unroll pragma, the loop will either be fully or partially unrolled. When the loop is fully unrolled, a copy of the loop is created for each iteration of the loop. If the loop is bound by a variable (i.e. the number of iterations is not defined during compilation), it is not possible to fully unroll the loop and it must be partially unrolled [48].

5.3 Interesting Aspects of Implementation

5.3.1 Binomial Tree – European and American Options

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	788	-
FIFO	-	-	-	-	-
Instance	340	420	85806	73374	0
Memory	0	-	1152	117	0
Multiplexer	-	-	-	655	-
Register	-	-	1762	-	-
Total	340	420	88720	74934	0
Available	280	220	106400	53200	0
Utilization (%)	121	190	83	140	0

Figure 4: Initial Resource Usage for the European Binomial Tree Algorithm

When developing IP for FPGAs, it is important to manage resource usage to ensure that the design fits on the FPGA. To begin, we implemented the binomial tree algorithm directly from the pseudocode shown in appendix 2. As shown in figure 4 above, the resource usage exceeded the limits of the device. This was due to the $exp(x)$ and $pow(a, b)$ functions used in the design.

```

1. for(int i = 0; i < height; i++) {
2.     St[i] = S * pow(u, (height-i)) * pow(d, i);
3. }

```

Figure 5: Binomial Tree Formation

To reduce the resource usage, we first removed the power functions, shown in figure 5 above, from our design. The first power calculation raises the constant u to the power $height$ down to 1 and the second raises the constants d from the power of 0 to $height-1$. A $pow(a, b)$ is called in each loop iteration, a lot of work is being repeated. This can be reduced by incorporating the power calculation into the for loop itself, as shown in figure 6 below.

```

1.     temp1 = pow(u, height);
2.     temp2 = 1;
3.
4.     // Initialise asset prices at maturity
5.     for(int i = 0; i < height; i++) {
6.         St[i] = S * temp1 * temp2;
7.
8.         temp1 /= u;
9.         temp2 *= d;
10.    }

```

Figure 6: Incorporating Power Calculations into the For Loop

Following synthesis, the for loop in figure 6 was scheduled with an initiation interval of 15, much higher than the target of 1. This was due to the floating-point division, which is computationally expensive. To fix this issue, before the loop, we calculate all values from a and store them in an array. The array is then read in reverse order to calculate the value of St . This reduced the initiation interval to a value of 8, this can be improved further by removing the loop carried dependency inside the for loop. A downside to this implementation is that the BRAM usage increased from 48% to 70% as more values are stored on the FPGA itself.

To further reduce the IPs resource usage, we approximate the value of the $\exp(a)$ functions with the following Taylor series [49]:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

To remove the factorials, we can rewrite this series as the following [50]:

$$e^x = 1 + \left(\frac{x}{1}\right) \left(1 + \left(\frac{x}{2}\right) \left(1 + \left(\frac{x}{3}\right) (\dots)\right)\right)$$

During our tests, when calculating $\exp(a)$, the value of a was always less than 0.1. This meant that we only needed to consider the first expansion when approximating the value of $\exp(a)$. Therefore, the following Taylor expansion was implemented into our binomial trees algorithm:

$$e^x = 1 + \left(\frac{x}{1}\right)$$

This is an extremely simple calculation and significantly reduced the amount of work being done in comparison to the work being done when calling the $\exp(a)$ function in C. If the value of a was higher, we would need to expand the equation to further to increase the accuracy of our approximation. The final resource usage after following these optimisation steps are given in figure 7 below.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	788	-
FIFO	-	-	-	-	-
Instance	196	33	9923	14567	0
Memory	0	-	1152	117	0
Multiplexer	-	-	-	655	-
Register	-	-	1762	-	-
Total	196	33	12837	16127	0
Available	280	220	106400	53200	0
Utilization (%)	70	15	12	30	0

Figure 7: Final Resource Usage for the European Binomial Tree Algorithm

5.3.2 Combined Binomial Tree

The LUT had highest resource usage, at 1.9% utilisation, for both the European and American binomial tree IPs on the Alveo u200. Because of this, we were able to combine the code for both algorithms into a single IP. This IP contains two functions – *eu_binomial_tree()* and *us_binomial_tree()* – which are both called in the top level function of the HLS code. The *dataflow* pragma is also placed in the top-level function to allow these functions to be executed simultaneously. This means that the IP concurrently calculates the European and American value of an option before returning them both to the host.

5.3.3 Monte Carlo – European Options

To perform a Monte Carlo simulation, we are required to generate random numbers from a normal distribution [51]. In the first CPU implementation, we used the *rand()* function in C to generate random numbers from a uniform distribution. A Box-Muller transform was then applied to transform these numbers from a uniform distribution to a normal distribution [52]. When porting the program to run on an FPGA, we found that the *rand()* function wasn't synthesisable in Vivado HLS. Because of this, a Mersenne-Twister algorithm was used to generate numbers from a uniform distribution instead of the *rand()* function [53].

Before developing the FPGA implementation, we analysed the runtime of each section of the Monte Carlo algorithm: initialisation, path generation, price calculation, standard error calculation. Path generation took the longest time, so we decided to concentrate on using the FPGA to accelerate this portion of the algorithm. We start by transferring the initial constants to the FPGA and then generate M paths with N steps per path. When pricing European options, we are only concerned about the price of the option at the final step of each path. Therefore, we only wrote this result back to the host. Figure 8 below shows how we used a burst-write as shown in [46] to more efficiently write data back to the host.

```
1. DATA_TYPE burst_buffer[BURSTBUFFERSIZE];
2.
3. // Simulate BURSTBUFFERSIZE paths and burst write to memory
4. burst_buffer: for(int i = 0; i < M; i += BURSTBUFFERSIZE) {
5.     int chunk_size = BURSTBUFFERSIZE;
6.
7.     if((i + BURSTBUFFERSIZE) > M) {
8.         chunk_size = M - i;
9.     }
10.
11. // Generate M paths
12. generate_paths: for(int j = 0; j < chunk_size; j++) {
13.     burst_buffer[j] = generate_path(&rand, S[0], deltaT, nudt, vdt, N);
14. }
15.
16. // Burst write to memory
17. write_paths: for(int j = 0; j < chunk_size; j++) {
18.     output_r[i + j] = burst_buffer[j];
19. }
20. }
```

Figure 8: Burst-write to Memory

In the above code, *BURSTBUFFERSIZE* is set to 256. So, we simulate 256 paths at a time and burst write the results to memory. This process is repeated until we have simulated a total of M paths.

6. Results and Discussion

A spot price of 50, strike price of 50, time to maturity of 1 year, dividend yield of 0%, volatility of 25% and risk-free rate of 5% were used when collecting the results presented in this section. These values were chosen arbitrarily and do not represent a typical option.

Throughout section 6.2, we will analyse the runtime of unoptimised and optimised CPU and FPGA implementations. The unoptimised CPU implementations were compiled with the -O0 flag which turns off compiler optimisation. The optimised version was compiled with the -O3 flag which optimises for code size and execution time but does not activate auto-parallelisation. For the FPGA implementations, the unoptimised version had no HLS pragmas added to the code. The optimised version did have HLS pragmas added. When compiling for FPGA, enabling compiler optimisation significantly increases the compilation time; as such, we disabled it with the -O0 flag. Due to the time constraints of the project we were unable to analyse a parallel CPU implementation against an FPGA implementation.

A list of supporting figures are supplied in appendix 6.

6.1 Accuracy

Algorithm	Platform		
	CPU	Zynq-7020	Alveo u200
EU Binomial Tree	2.78541565	2.78521299	2.78521300
US Binomial Tree	8.01245689	8.01226807	8.01227000
EU Monte Carlo	2.76810241	2.76814928	2.76810290

Table 2: Output of Algorithms on CPU, Zynq-7020 and Alveo u200

The output of the CPU, Zynq-7020 and Alveo u200 implementations of the implemented algorithms are shown in table 2 above. Both binomial trees algorithms were run with a depth of 30,000 and the Monte Carlo algorithm had 1,000,000 paths with 100 steps per path. Both binomial trees algorithms are accurate to three decimal places on both FPGAs whereas the Monte Carlo algorithm is accurate to four. Due to the secretive nature of investment banking, we do not know what decimal place accuracy these firms aim for. However, stock screeners such as Yahoo! Finance display information to 4 decimal places. Based on this, the accuracy of our Monte Carlo algorithm is acceptable. The accuracy of the binomial trees algorithm would ideally be increased to 4 decimal places.

As we are approximating the values of the exponential function in the FPGA implementation of our binomial tree algorithms; we expect the output of the CPU and FPGA implementations to differ slightly. This does not however explain the difference in output for the Monte Carlo algorithm, as we are not approximating any values in this algorithm. The slight differences between the Zynq-7020 and Alveo u200 are likely due to the loops being pipelined and unrolled differently on each device.

6.2 Time Performance of Implementations

6.2.1 Binomial Tree

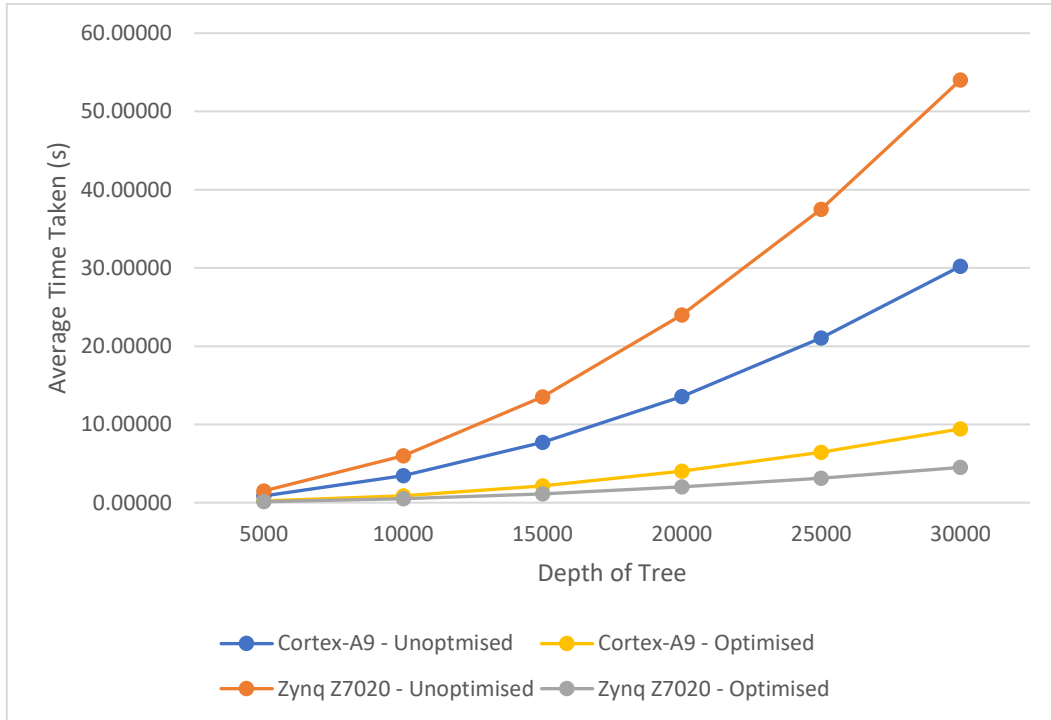


Figure 9: Runtime of EU Binomial Tree for Cortex-A9 and Zynq-7020

As we can see in figure 9 above, the unoptimised binomial trees algorithm ran slower on the Zynq-7020 than on the Cortex-A9. This is expected given that the clock speed of the Zynq-7020 is slower than the clock speed of the Cortex-A9. After optimising the HLS code through the methods discussed in section 6, we were able to achieve a 7x speedup over the unoptimised CPU version. After enabling compiler optimisation on the CPU with the -O3 compiler flag the speedup is reduced to 2x.

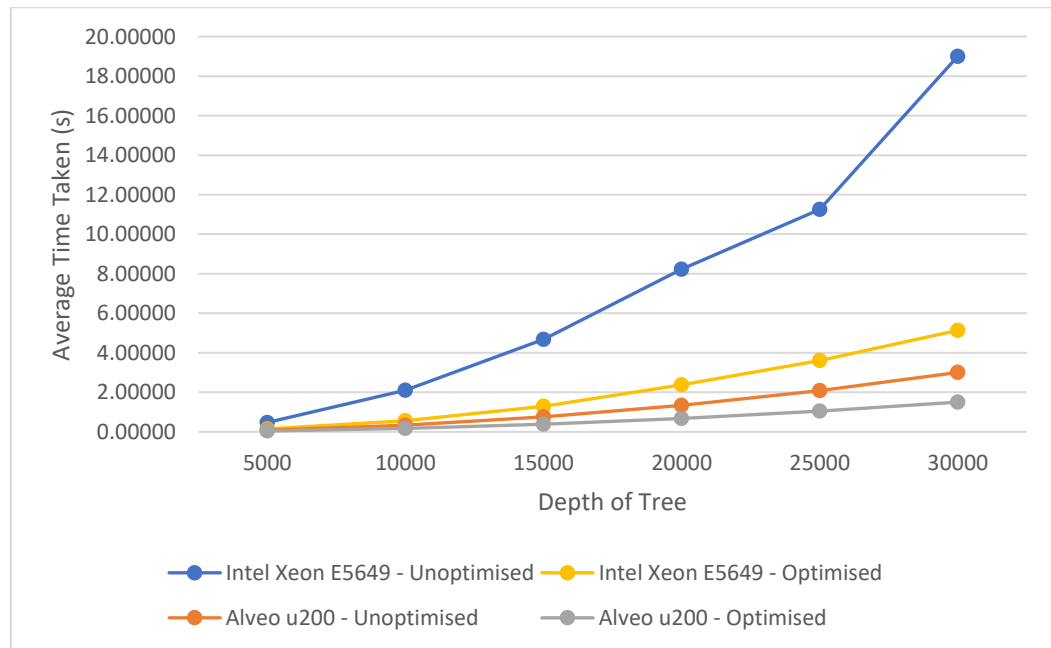


Figure 10: Runtime of EU Binomial Tree for Intel Xeon E5649 and Alveo u200

In comparison to the Zynq-7020, the Alveo u200 unoptimised implementation runs faster than both the unoptimised and optimised Intel Xeon 5649 implementation. This is because the SDAccel compiler pipelined some of the for loops before the addition of HLS pragmas, as indicted by the compiler log. The optimised version achieved a speedup of 12x in comparison to the unoptimised CPU implementation. This speedup was reduced to 3x when compiling the CPU implementation with the -O3 flag.

Similar research was carried out by [54] who compared the runtime of a European binomial tree algorithm on an Intel Xeon E5-1650 c3 and Alveo u200. They achieved a 70x speedup against a parallel CPU implementation using 12 threads. Despite the substantial speedup achieved in their paper, the binomial tree model used yields inaccurate results. Using the starting constants listed at the start of this chapter, and a depth of 1024, the output is 3.174 where the closed form solution is 2.787. This level of inaccuracy would lead to substantial losses if the algorithm was employed by an investment bank. Whilst our implementation gives 3x (rather than 70x) speed-up it does give accurate results which would protect the finances of an investment bank.

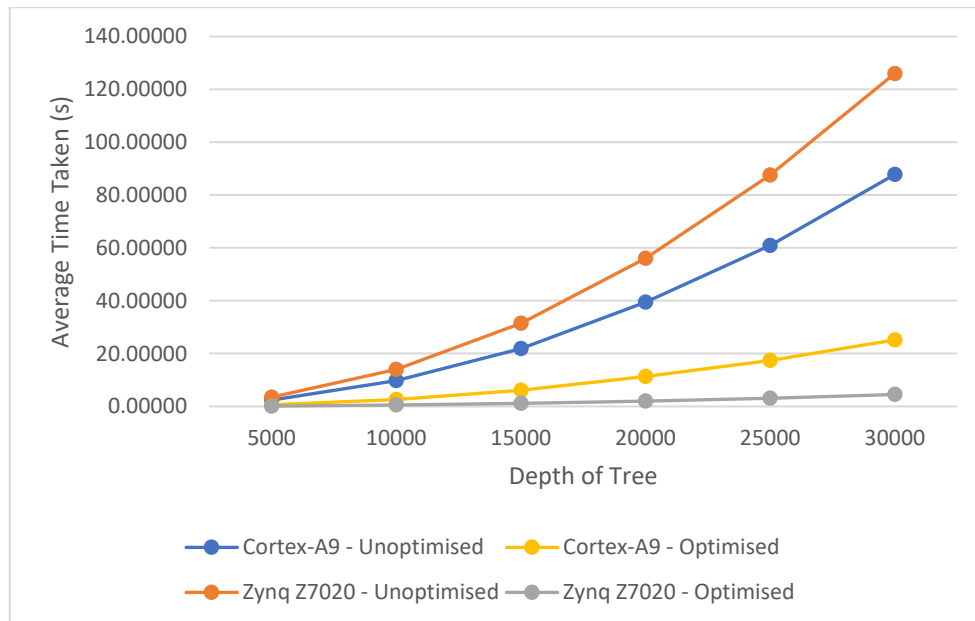


Figure 11: Runtime of US Binomial Tree on Cortex-A9 and Zynq-7020

The CPU implementation of the US binomial tree algorithm took substantially longer to run on the Cortex-A9 than the European implementation. This is because the mathematics applied throughout the traversal stage of the algorithm differs between these two implementations. As with the European algorithm, the unoptimised implementation of the US algorithm also ran slower on the Zynq-7020 than both Cortex-A9 implementations. The optimised version of the code on the Zynq-7020 ran just as fast as the European algorithm. For the US binomial tree algorithm, a substantial speedup of 20x was achieved in comparison to the unoptimised CPU implementation. This speedup was reduced to 5x when compared to the optimised CPU implementation.

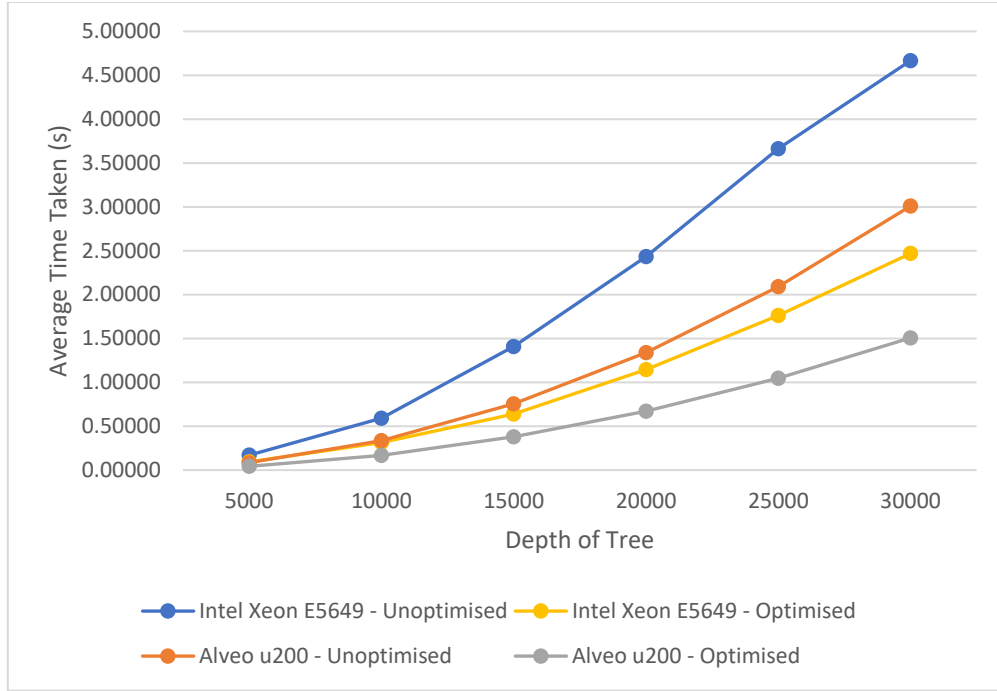


Figure 12: Runtime of US Binomial Tree on Intel Xeon E5469 and Alveo U200

Unlike the Cortex-A9, the CPU implementation of the US binomial tree algorithm ran 4x faster than the European algorithm on the Intel E5649. A few reasons exist which could explain this difference. Firstly, the Intel chip may be better equipped to deal with the mathematical calculations applied during the traversal stage. A more compelling reason is that the Intel chip has higher amounts of cache memory, meaning that it must access global memory a fewer number of times. These hypotheses need to be analysed further, however that research is outside the scope of this project. Due to the decrease in runtime of the CPU implementation, we achieved a speedup of 3x against the unoptimised CPU implementation and 1.6x against the optimised CPU implementation.

Both the European and US binomial trees algorithms are significantly bottlenecked by the *power_calculation* and *loop_traversal* for loops. Both for loops contain a loop carried dependency, and as such the compiler is unable to maintain an initiation interval of 1 which significantly increasing the latency of these loops. The only way to reduce the initiation interval is by removing the dependency from these for loops.

6.2.2 Combined European and US Binomial Tree

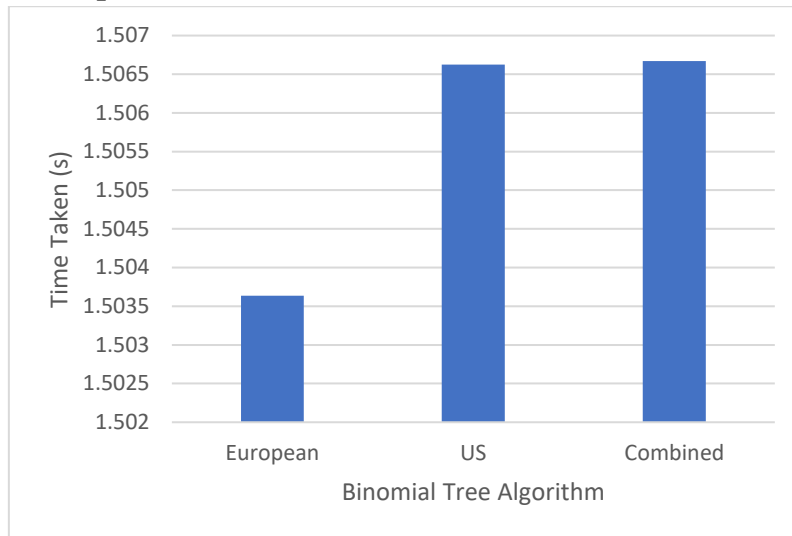


Figure 13: Runtime of Binomial Trees Algorithms

Figure 13 above shows the runtime of the European, US and combined binomial trees algorithms with a depth of 30,000. Figure 13 shows that the European and US implementations both ran concurrently, and at the same speed as running them individually, meaning that the total time to solve both algorithms is the time taken to solve the slowest algorithm. Whereas the time taken to execute both algorithms on the CPU is the time taken to run them one after another, as we have not optimised to reuse any data. This provides a significant speedup of 15x over the CPU implementation where both algorithms are executed sequentially.

6.2.3 European Monte Carlo

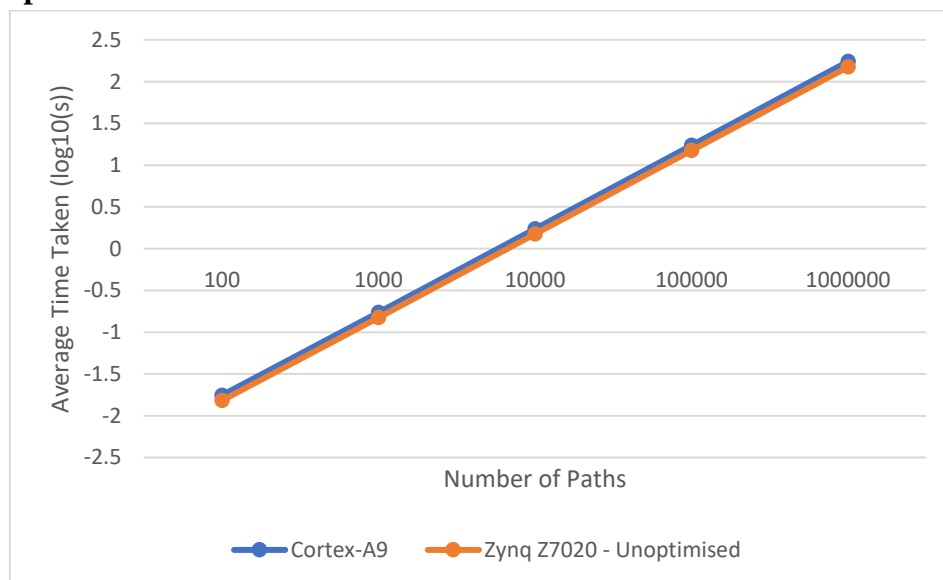


Figure 14: Monte Carlo Runtime on Cortex-A9 versus Zynq-7020



Figure 15: Monte Carlo Runtime on Intel Xeon E5649 versus Alveo u200

As shown in figure 14, the unoptimised version of the Monte Carlo algorithm ran slightly faster on the Zynq-7020 than the Cortex-A9, achieving a speedup of 1.1x. Due to the high resource usage we were unable to optimise the algorithm further without running out of resources on the device. In contrast to this, figure 15 shows that the unoptimised Monte Carlo algorithm ran roughly 3x slower on the Alveo u200 than the Intel Cortex E5649. The algorithm slowed down further after following the optimisation steps stated in section 6.3. The HLS reports show that the latency of the *generate_paths* function increased significantly when optimising the code, which lead to the increased execution time. The current design could be further improved to run on an FPGA: currently, we have to wait for numbers to be generated and transformed during each iteration of the *path_generation* loop but this delay could be removed by utilising HLS streams [55]. By utilising HLS streams, we would better optimise the algorithm for dataflow. The use of HLS streams will be explored in further works.

Algorithm	Speedup			
	Zynq-7020 (unoptimised)	Zynq-7020 (optimised)	Alveo u200 (unoptimised)	Alveo u200 (optimised)
European Binomial Tree	7x	2x	12x	3x
US Binomial Tree	20x	5x	3x	1.6x
European Monte Carlo	1.1x	-	0.12x	0.09x

Table 3: Summary of Timing Results

6.3 Energy Performance of Implementations

6.2.1 Cortex-A9 versus Pynq-Z2

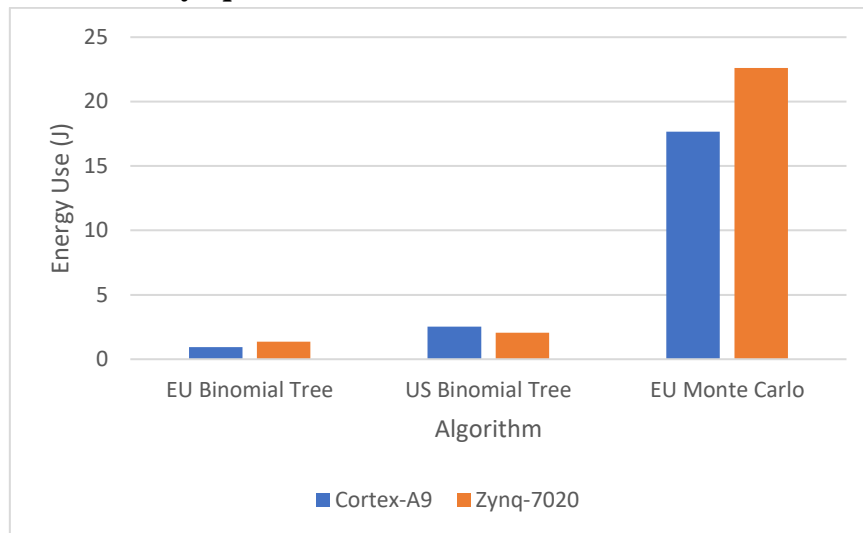


Figure 16: Energy Use on Cortex-A9 versus Zynq-7020

Two measurements were made: one when the board was idle, and another when the optimised version of each algorithm was executing. The difference between the two was then taken in order to calculate energy consuming by running the algorithm. Figure 16 shows that the Cortex-A9 tends to use less energy than the Zynq-7020. The US binomial trees algorithm saw a significant speedup when implemented on the Zynq-7020 which is why the results are opposite for this algorithm. As we were measuring the energy use of the whole board, and not specific components, these results do not provide a true insight into the energy use of the Cortex-A9 and Zynq-7020. We were working in a Jupyter notebook environment whilst working with the Zynq-7020 so the CPU would have been in use at the same time as the FPGA. This would explain why we are seeing higher energy use for the Zynq-7020 when we hypothesised that it would be lower.

6.2.2 Intel Xeon E5649 versus Alveo u200

The Intel Xeon E5649 is an old CPU and lacks the performance counters required to measure its power consumption. As mentioned in section 4.3, the TDP was used instead. The TDP of the Intel CPU is 80W and is defined as “the average power, in watts, the processor dissipates when operating at base frequency with all cores active under an Intel defined, high-complexity workload” [56]. As the CPU has 6 cores, we will assume that the minimum power that the CPU would use is a sixth of this value.

Algorithm	Energy Use (J)		
	Intel Min (13W)	Intel Max (80W)	Alveo
EU BT	66.77	410.89	419.49
US BT	32.24	198.40	322.64
EU MC	262.24	1613.76	8294.26
Combined	95.29	586.39	482.47

Table 4: Energy Use of Intel Xeon E5649 versus Alveo u200

As in the previous section, these results show that the CPU implementation tends to use less energy than the FPGA implementation. The script used to measure power use on the Alveo also measures idle usage for 5 seconds before and after running the kernel, so actual energy use would be lower. Furthermore, the estimation of energy use for the Intel fails to account for the energy use of memory, which is included in the Alveo measurements. As the Monte Carlo script is slower on the FPGA, we expect the energy use to be higher than the CPU implementation. It would be interesting to revisit this analysis in the future when we have access to more reliable tools to measure the energy use of both the Intel processor and Alveo u200.

6.4 Floating Point versus Fixed Point

The algorithms implemented thus far have all used floating-point data types. Most algorithms do not need the level of accuracy provided by a floating-point design and would perform equally as well with a fixed-point design.

Moving from a floating-point to fixed-point design can benefit developers in several ways. First and foremost, the developer should see a significant reduction in the utilisation of FPGA resources. This is because fewer DSP blocks, look-up tables and flip-flops are needed when working with fixed-point data types. This reduction in FPGA resources should also lead to a reduction in power consumption. Furthermore, the developer should also see latency improvements as the number of resources are reduced.

Xilinx offer a header file, named *ap_fixed.h*, which allows us to convert our IP to a fixed-point design. We can define a fixed-point type by typing *ap_fixed<W, I>* where W is the total number of bits required and I is the number of bits for the integer part; any remaining bits are used for the decimal part. For example, *ap_fixed<10, 5>* would create a fixed-point type with 5 integer bits and 5 decimal bits. Due to the power calculations in this algorithm, the maximum depth of the tree was limited to 5,000. If the depth was set at 30,000, we would require 56 bits to represent just the integer part of our number and would see little benefit from converting to a fixed-point design.

To determine the number of bits required for the integer part, we calculated the value of $\log_2 n$ where n is the range of numbers we wish to represent. The smallest number we need to represent is 0 and the largest is just under 75 million. Therefore, we require $\log_2(75\text{million})$ or 26 bits to represent the integer part of our number. To be within 0.5 of the output from the floating-point implementation, we require 7 decimal places of accuracy. This meant that we needed $\log_2(10^7)$ or 23 bits to represent the decimal part of our number. The final definition of the fixed-point type was therefore *ap_fixed<49, 26>*.

Name	LUT	LUTMem	REG	BRAM	URAM	DSP
Platform	153438	17736	222047	320	0	7
▼ User Budget	1028802	574104	2142433	1840	960	6833
Used Resources	13864	657	15110	151	0	192
Unused Resources	1014938	573447	2127323	1689	960	6641
▼ binomial_tree_binomial_tree (1)	13864	657	15110	151	0	192
binomial_tree_1	13864	657	15110	151	0	192

Figure 17: Fixed-Type Resource Usage

The resource usage of all components dropped after switching to a fixed-point design, however the change wasn't that significant. This is because of the high number of bits required to represent the values required for our algorithm.

Type	Depth	Output
Floating Point	1000	2.78835110
Floating Point	5000	2.78508570
Fixed Point	1000	2.78941320
Fixed Point	5000	3.04024650

Table 5: Floating-Point and Fixed-Point Output

With a depth of 1,000 the output from the fixed-point design was accurate to two decimal places. The differences in these answers is due to the lower precision for the fixed-point design. Overflow and rounding are also handled differently in the fixed-point design which could also lead to different answers. With a depth of 5,000 the answer is much more inaccurate. This suggests that we need higher decimal place accuracy for trees of this size.

Type	Depth	Average	Min	Max	Speedup
Floating Point	1000	0.0022755	0.002185	0.002356	-
Floating Point	5000	0.0426866	0.042637	0.042729	-
Fixed Point	1000	0.0022027	0.002147	0.002254	1.033077
Fixed Point	5000	0.0424855	0.042477	0.042495	1.004733

Table 6: Time Taken for Floating-Point and Fixed-Point

As we can see from table 6 above, we yield a slight performance increase when moving to a fixed-point design. An energy saving of 15% was observed as the energy usage fell from 335J in the floating-point design down to 284-Joules in the fixed-point design. To improve these numbers, we would have to reduce the resource usage even further.

The energy and timing values could be improved by further reducing the resource usage on the device. Such a large range isn't required by all variables in the binomial trees algorithm. We could therefore reduce the number used for these variables, which would reduce the resource utilisation on the device.

6.5 Comparison of Algorithms

The closed form solution to the Black-Scholes model using the values mentioned at the start of this section is 2.78676301. The output of our binomial trees algorithm for varying depth, and Monte Carlo algorithm for varying height, are given in table 7 and 8 respectively. The number of steps in the Monte Carlo algorithm was fixed at 100.

EU Binomial Tree			
Depth	CPU	Pynq-Z2	Alveo
1000	2.78714871	2.78835106	2.78835110
5000	2.78578687	2.78508568	2.78508570
10000	2.78542614	2.78532600	2.78532600
15000	2.78543425	2.78546715	2.78546710
20000	2.78374958	2.78534555	2.78534560
25000	2.77286482	2.77295160	2.77295160
30000	2.78541565	2.78521299	2.78521300

Table 7: Output of Binomial Tree Algorithm with Varying Depth

EU Monte Carlo			
# Paths	CPU	Pynq-Z2	Alveo
100	2.47350740	2.47350873	2.47350840
1000	2.95371366	2.95371251	2.95371220
10000	2.76025653	2.76025878	2.76025580
100000	2.76002216	2.76002427	2.76002430
1000000	2.76810241	2.76814928	2.76810290

Table 8: Output of Monte Carlo Algorithm with Varying Number of Paths

As we can see from table 7 above, the binomial trees algorithm provides a solution that is much closer to the closed form solution than the Monte Carlo algorithm. Moreover, the binomial trees algorithm can give a close approximation with a depth of just 1,000 whereas the Monte Carlo algorithm requires at least 10,000 paths before it can give a reasonable approximation. From this, and the analysis carried out in sections 7.2 and 7.3, we see the binomial trees algorithm can provide a much better approximation, in shorter amount of time and with lower energy use. This supports the findings by [1] who stated that the Monte Carlo algorithm should only be used as a last resort.

7. Evaluation

Since redefining the scope of the project, as discussed in section 5.5, the project has progressed at a reasonable pace, despite hitting several roadblocks along the way. The first roadblock was figuring out how to generate random numbers on the FPGA after discovering that the *rand()* function in C is not synthesisable when placed in HLS code. After some research, we found that the Mersenne-Twister algorithm combined with the Box-Muller transform algorithm was widely used within finance research. The Box-Muller transform generates pairs of normally distributed numbers, which aren't required when pricing under Black-Scholes conditions; however, they are required when pricing under Heston conditions. Another stumbling block came when the binomial trees algorithms used too many resources and would not fit on the Zynq-7020 FPGA. The implementation of the Taylor expansion, as well as integrating power calculations into the for loop, minimised the resource usage and allowed our design to fit on the Zynq-7020.

We have seen good speedup results for both binomial trees algorithms on both the Zynq-7020 and Alveo u200. These results were further improved when combining the European and American binomial trees algorithms into one IP and running them concurrently. Unfortunately, our current implementation of the Monte Carlo algorithm on the Alveo u200 is slower than the CPU implementation. This is slightly disappointing as we hoped all FPGA implementations would be faster than their respective CPU implementations at this stage of the project. It was also disappointing that we were unable to make a detailed evaluation into the energy use on an FPGA in comparison to a CPU. The analysis for the fixed-point design was also limited due to the high number of bits required to represent the range of numbers in the algorithm. Despite this, we did see a slight increase in speed and energy efficiency when moving to a fixed-point design.

In general, CPUs perform well whilst using both the *exp(x)* and *pow(a, b)* functions. However, to get good performance on FPGAs we had to use approximations such as the Taylor series for calculating the value of *exp(x)*. This makes FPGAs perform well for specific situations, such as when *x* is less than 0.1 as mentioned in section 5.3.1. However, for larger values we must add an extra expansion term to maintain the same accuracy when approximating the value of *exp(x)*. This shows the importance of the device being reconfigurable so we can load bitstreams that are optimised for the input data.

8. Learning Points

Developing software to run on an FPGA is something that I have never explored before. As such, the project had a steep learning curve and allowed me to gain a lot of new skills and knowledge. First and foremost, I learnt how to develop algorithms that take advantage of FPGA architecture. I also learnt how to write HLS code, and use the relevant software, specifically Vivado Design Suite and SDAccel, that allow us to interface with FPGAs. I also gained an appreciation of how to use OpenCL to interact with a device from a host. Finally, I learnt about generating pseudorandom numbers and mathematical approximations such as the Taylor expansion.

Redefining the project scope towards the beginning of the project was crucial to its overall success. Without narrowing the scope, I would not have been able to meet all the aims I set out to achieve. The implementation of the Taylor expansion and integration of power calculations into the for loops were also crucial to the success of this project. Without them, the Zynq-7020 implementations would not have run faster than the Cortex-A9 whether the HLS code was optimised or not.

Having to redefine the scope of the project at the beginning was a significant drawback for this project. This stressed the importance of ensuring that the scope of a project is suitable for the time available; something which we will aid us when scoping projects in the future

9. Professional Issues

9.1 Professional Competence and Integrity

At the beginning of the project, it was necessary to review previous literature relating to FPGAs and options pricing in finance. This review revealed the knowledge and skills required for this project. As such, I was able to develop my knowledge and skills such that I did not claim a level of competence that I did not possess.

9.2 Duty to Relevant Authority

A user manual has been included in the Github repository as well as the data collected throughout the experiments [36]. This ensures that no data are misrepresented or withheld. As we are not dealing with any confidential information, there were no concerns regarding how the results are disclosed.

9.3 Duty to the Profession

It is important to encourage and support our peers in their professional development. To do this, the code developed throughout this project along with the results are available in a public Github repository. A copy of this thesis will also be uploaded following the marking process. This repository will stay live following the project.

9.4 Ethical use of Data

No data was required to conduct the research throughout this project. Before conducting the research, arbitrary values were selected for the initial stock price, strike price, dividend yield, risk-free rate, volatility and time to maturity. Whilst conducting the experiments data relating to the run time, energy usage and accuracy of the algorithms were collected. These data were then analysed to test our hypothesis. If all the results are reported accurately, there will be no concerns regarding the ethical use of data.

10. Conclusions

10.1 Summary

Based on the work done throughout this thesis, we can at least partially reject the first null hypothesis (presented in section 3.1.1); that all FPGA implementations of the algorithms covered will be slower than the respective CPU implementations. The Monte Carlo algorithm needs to be optimised further to fully reject the null hypothesis. Due to the lack of reliable energy measurements, we were unable to reject the second null hypothesis (presented in section 3.1.2) that the FPGA implementations will be more energy efficient than the CPU implementations.

The algorithms that were built during this project demonstrate how FPGAs can be used to accelerate options pricing libraries, whilst maintaining a similar accuracy to the CPU implementations. A summary of the timings results is presented in table 3 at the end of section 6.2. We also demonstrate how a fixed-point design can be employed to reduce the resource usage of an FPGA and in turn reduce the energy consumption.

10.2 Future Work

Given the limited timeframe of this project, it was not possible to gain a full appreciation of the techniques used when designing algorithms for FPGAs. Therefore, it would be useful to revisit this work after gaining more knowledge on how to optimise designs for FPGAs and apply this knowledge to the algorithms covered throughout this project. In addition, it would be useful to revisit the energy use analysis when we have access to more profiling tools for recording power consumption. It would also be useful to use a more modern CPU.

Not all variables in the binomial tree algorithm have a large range of values. This means we could assign a lower number of bits to store these values, thus reducing the resource usage and providing a performance increase. We could also analyse the performance of these algorithms in a multithreaded or multiprocessor environment as well as on other accelerators such as GPUs. Lastly, the research can be expanded in order to satisfy all desirable aims.

11. References

- [1] Jin, Q., Luk, W. & Thomas, D. (2011) On Comparing Financial Option Price Solvers on FPGA, *IEEE International Symposium on Field-Programmable Custom Computing Machines*. Available online: <http://www.doc.ic.ac.uk/~wl/papers/11/fccm11qj.pdf> [Accessed: 12/06/2019].
- [2] Xilinx (n.d.) What is an FPGA? Field Programmable Gate Array. Available online: <https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html> [Accessed: 10/06/2019].
- [3] Moore, A. & Wilson, R. (2017) *FPGAs For Dummies*, 2nd Edition. Hoboken: John Wiley & Sons, Inc.
- [4] Amano, H. (2018) *Principles and Structures of FPGAs*. Singapore: Springer.
- [5] Sadrozinski, H. & Wu, J. (2011) *Applications of Field-Programmable Gate Arrays in Scientific Research*. Boca Raton: Taylor & Francis Group. Available online: <https://www.taylorfrancis.com/books/9780429063466> [Accessed: 11/06/2019].
- [6] Waidyasooriya, H., Hariyama, M. & Kasahara, K. (2016) Architecture of an FPGA accelerator for molecular dynamics simulation using OpenCL. *IEEE International Conference on Computer Science and Information Science (ICIS)*. Japan, 26-29 June 2016. Available online: <https://ieeexplore.ieee.org/abstract/document/7550743> [Accessed: 01/07/2019].
- [7] Hussain, H., Benkrid, K. & Seker, H. (2016) Novel dynamic partial reconfiguration implementations of the support vector machine classifier on FPGA. Available online: <https://journals.tubitak.gov.tr/elektrik/abstract.htm?id=18984> [Accessed: 01/07/2019].
- [8] Rabieah, M. & Bouganis, C. (2015) FPGA based nonlinear Support Vector Machine training using ensemble training. *IEE International Conference on Field Programmable Logic and Applications (FPL)*. London, 2-4 September 2015. Available online: <https://ieeexplore.ieee.org/abstract/document/7293972> [Accessed: 01/07/2019].
- [9] Kara, K., Alistarh, D., Alonso, G., Mutlu, O. & Zhang, C. (2017) FPGA-Accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-Off. *IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Napa, CA, 30 April – 2 May 2017. Available online: <https://ieeexplore.ieee.org/abstract/document/7966672> [Accessed: 01/07/2019].
- [10] Wang, C., Gong, L., Yu, Q., Li, C., Xie, Y. & Zhou, X. (2016) DLAU: A Scalable Deep Learning Accelerator Unit on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3). Available online: <https://ieeexplore.ieee.org/abstract/document/7505926> [Accessed: 30/06/2019].
- [11] Cai, R., Ren, A., Liu, N., Ding, C., Wang, L., Qian, X., Pedram, M. & Wang, Y. (2018) VIBNN: Hardware Acceleration of Bayesian Neural Networks. Available online: <https://arxiv.org/pdf/1802.00822.pdf> [Accessed: 30/06/2019].
- [12] Pang, A. & Membrey, P. (2017) *Beginning FPGA: Programming Metal*. Berkley: Springer.
- [13] Xilinx (n.d.) Vivado High-Level Synthesis. Available online: <https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html> [Accessed: 18/06/2019].

- [14] Schryver, C. (2015) FPGA Based Accelerators for Financial Applications. Germany: Springer. Available online: <https://link.springer.com/book/10.1007%2F978-3-319-15407-7#about> [Accessed: 25/06/2019].
- [15] Tian, X., Benkrid, K. & Gu, X. (2008) High Performance Monte-Carlo Based Option Pricing on FPGAs. Available online: <https://pdfs.semanticscholar.org/3331/301c0f4b8f8cda35898e58a2406e7e6ee26a.pdf> [Accessed: 24/06/2019].
- [16] Aswani, S. (2016) High-Frequency Trading Need High-Intelligence Computing. *HPCWire*, Internet Edition. 4th April. Available online: https://www.hpcwire.com/solution_content/hpe/financial-services/high-frequency-trading-need-high-intelligence-computing/ [Accessed: 24/06/2019].
- [17] Theis, T. & Wong, H. (2017) The End of Moore's Law: A New Beginning for Information Technology. Available online: <https://aip.scitation.org/doi/abs/10.1109/MCSE.2017.29> [Accessed: 16/09/2019].
- [18] Brealey, R., Myers, S. & Allen, F. (2016) Principles of Corporate Finance. McGraw-Hill Education.
- [19] Bodie, Z., Kane, A. & Marcus, A. (2013) Essentials of Investments: Global Edition. Europe: McGraw-Hill Education.
- [20] Chen, J. (2018) Heston Model. Available online: <https://www.investopedia.com/terms/h/heston-model.asp> [Accessed: 30/06/2019].
- [21] Clewlow, L., Llanos, J. & Strickland, C. (1994) Pricing Exotic Options in a Black-Scholes World. Available online: <https://warwick.ac.uk/fac/soc/wbs/subjects/finance/research/wpaperseries/1994/94-54.pdf> [Accessed: 18/06/2019].
- [22] University of Oxford (2018) Pricing American Options with Monte Carlo Methods. Available online: https://www.maths.ox.ac.uk/system/files/attachments/TT18_dissertation_1000246.pdf [Accessed: 19/06/2019].
- [23] Malesevic, G. (2017) Use of the Monte Carlo Simulation in Valuation of European and American Call Options. Available online: <https://publications.lakeforest.edu/cgi/viewcontent.cgi?article=1115&context=seniortheses> [Accessed: 22/06/2019].
- [24] Mohammed, S. & Singh, S. (n.d.) Pricing Options Using Monte-Carlo Methods. Available online: https://www.cmi.ac.in/~shariq/Shariq%20files/option_pricing.pdf [Accessed: 30/06/2019].
- [25] Jin, Q., Thomas, D., Luk, W. & Cope, B. (2009) Exploring Reconfigurable Architectures for Tree-Based Option Pricing Models. Available online: <http://www.doc.ic.ac.uk/~wl/papers/09/trets09qj.pdf> [Accessed: 29/06/2019].
- [26] Thornton, S. (2017) Fixed point vs Floating point. Available online: <https://www.microcontrollertips.com/difference-between-fixed-and-floating-point/> [Accessed: 22/06/2019].
- [27] Overton, M. (2001) Numerical Computing with IEEE Floating Point Arithmetic. US: Society for Industrial & Applied Mathematics. Available online:

http://sistemas.fciencias.unam.mx/~gcontreras/joomla15/tmp/Numerical_Computing_with_IEEE_Floating_Point_Arithmetic.pdf [Accessed: 22/06/2019].

[28] Xilinx (2019) SDAccel Programmers Guide. Available online: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1277-sdaccel-programmers-guide.pdf [Accessed: 23/06/2019].

[29] Schryver, C., Shcherbakov, I., Kienle, F., Wehn, N., Marxen, H., Kostiuk, A. & Korn, R. (2011) An Energy Efficient FPGA Accelerator for Monte Carlo Option Pricing with the Heston Model. Available online: <https://ieeexplore.ieee.org/abstract/document/6128621> [Accessed: 15/06/2019].

[30] Chatziparaskevas, G., Brokalakis, A. & Papaeofstathiou, I. (2012) An FPGA-Based Parallel Processor for Black-Scholes Option Pricing Using Finite Differences Schemes. Available online: <https://dl.acm.org/citation.cfm?id=2492887> [Accessed: 15/06/2019].

[31] Tian, X. & Benkrid, K. (2009) American Option Pricing on Reconfigurable Hardware Using Least-Squares Monte Carlo Method. Available online: <https://ieeexplore.ieee.org/document/5377662> [Accessed: 17/07/2019].

[32] Xilinx (2018) Zynq-7000 SoC Data Sheet: Overview. Available online: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf [Accessed: 01/07/2019].

[33] Xilinx (2019) Alveo u200 and u250 Data Centre Cards Data Sheet. Available online: https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf [Accessed: 01/07/2019].

[34] Xilinx (n.d.) Vivado Design Suite. Available online: <https://www.xilinx.com/products/design-tools/vivado.html> [Accessed: 02/07/2019].

[35] Xilinx (n.d.) SDAccel Development Environment. Available online: <https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html> [Accessed: 02/07/2019].

[36] Carter, M. (2019) COMP702 FPGA MSc Project. Available online: <https://github.com/mjcarter95/COMP702-FPGA-MSc-Project>

[37] Trello (n.d.) Trello. Available online: <https://trello.com> [Accessed: 18/06/2019].

[38] Python (n.d.) Basic date and time types. Available online: <https://docs.python.org/2/library/datetime.html> [Accessed: 12/08/2019].

[39] Khronos (n.d.) clGetEventProfilingInfo. Available online: <https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clGetEventProfilingInfo.html> [Accessed: 12/08/2019].

[40] GeeksforGeeks (n.d.) Measure execution time with high precision in C/C++. Available online: <https://www.geeksforgeeks.org/measure-execution-time-with-high-precision-in-c-c/> [Accessed: 12/08/2019].

[41] YOTINO (n.d.) YOTINO USB Tester Voltmeter Ammeter Digital LCD Voltage Monitor Current Meter Capacity Tester with Backlight. Available online: <https://www.amazon.co.uk/gp/product/B078SRP64S/> [Accessed: 02/07/2019].

[42] Xilinx (n.d.) PYNQ – Python Productivity for Zynq. Available online: <http://www.pynq.io/> [Accessed: 05/06/2019].

- [43] Niroshan, C. (2018) Xilinx Vivado HLS Beginners Tutorial: Integrating IP Core into Vivado Design. Available online: <https://medium.com/@chathura.abeyrathne.lk/xilinx-vivado-hls-beginners-tutorial-integrating-ip-core-into-vivado-design-7ddea40c9b7e> [Accessed: 15/07/2019].
- [44] Xilinx (2019) SDAccel Examples. Available online: https://github.com/Xilinx/SDAccel_Examples [Accessed: 28/07/2019].
- [45] Xilinx (2019) Matrix Multiply with Local Memory. Available online: https://github.com/Xilinx/SDAccel_Examples/tree/master/getting_started/cpu_to_fpga/02_local_mem_c [Accessed: 24/07/2019].
- [46] Xilinx (2019) Matrix Multiplication Burst Read Write. Available online: https://github.com/Xilinx/SDAccel_Examples/tree/master/getting_started/cpu_to_fpga/03_burst_rw_c [Accessed: 02/09/2019].
- [47] Xilinx (n.d.) pragma HLS array_partition. Available online: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/gle1504034361378.html [Accessed: 26/07/2019].
- [48] Xilinx (n.d.) Loop Pipelining and Loop Unrolling. Available online: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_pipelining_loop_unrolling.html [Accessed: 16/07/2019].
- [49] Efunda (n.d.) Taylor Expansions of Exponential Functions. Available online: https://www.efunda.com/math/taylor_series/exponential.cfm [Accessed: 09/08/2019].
- [50] GeeksforGeeks (n.d.) Efficient Program to Calculate e^x . Available online: <https://www.geeksforgeeks.org/program-to-efficiently-calculate-ex/> [Accessed: 09/08/2019].
- [51] unknown (n.d.) Monte Carlo Simulation, Chapter 4.
- [52] Weisstein, E. (2019) Box-Muller Transformation. Available online: <http://mathworld.wolfram.com/Box-MullerTransformation.html> [Accessed: 18/07/2019].
- [53] Sultanik, E. (2013) A pure C implementation of the Mersenne twister is a pseudo-random number generation algorithm. Available online: <https://github.com/ESultanik/mtwister> [Accessed: 18/07/2019].
- [54] Xilinx (2019) Binomial Model. Available online: <https://github.com/Xilinx/BinomialModel> [Accessed: 15/08/2019].
- [55] Xilinx (2018) pragma HLS stream. Available online: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/ylh1504034366220.html [Accessed: 24/07/2019].
- [56] Intel () Intel Xeon Processor E5649. Available online: <https://ark.intel.com/content/www/us/en/ark/products/52581/intel-xeon-processor-e5649-12m-cache-2-53-ghz-5-86-gt-s-intel-qpi.html> [Accessed: 09/09/2019].
- [57] unknown (n.d.) The Binomial Method, Chapter 2.

12. Appendix 1 – Equations

12.1 Black Scholes Formula

This formula was taken from [18].

$$C_0 = S_0 e^{-\delta T} N(d_1) - X e^{-rT} N(d_2)$$

Where

$$d_1 = \frac{\left(\ln \left(\frac{S_0}{X} \right) + \left(r - \delta + \frac{\sigma^2}{2} \right) T \right)}{\sigma \sqrt{T}}$$

$$d_2 = d_1 - \sigma \sqrt{T}$$

And where

C_0 = Current call option value

S_0 = Current stock price

$N(d)$ = Probability that a draw from a standard normal distribution will be less than d

X = Exercise price

δ = Annual dividend yield of underlying stock

r = Risk – free rate

T = Time remaining until expiration (in years)

σ = Standard deviation of the annualised continuously compounded rate of return of the stock

12.2 Heston Formula

This formula was taken from [19].

$$\begin{aligned} dS_t &= rS_t dt + \sqrt{V_t} S_t dW_{1t} \\ dV_t &= k(\theta - V_t)dt + \sigma \sqrt{V_t} dW_{2t} \end{aligned}$$

Where

S_t = the asset price at time t

r = risk free rate

$\sqrt{V_t}$ = Volatility (standard deviation) of the asset price

σ = volatility of the volatility $\sqrt{V_t}$

θ = long – term price variance

k = rate of reversion to the long – term price variance

dt = indefinitely small positive time increment

W_{1t} = Brownian motion of the asset price

W_{2t} = Brownian motion of the asset's price variance

ρ = correlation coefficient for W_{1t} and W_{2t}

13. Appendix 2 - Pseudocode

13.1 Monte Carlo Black-Scholes

This is the pseudocode for Monte Carlo simulation of the Black-Scholes model for a European Call Option, taken from [50].

```
initialise_parameter{K, T, S, sig, r, div, N, M}

# Compute constants
dt = T/N
nudt = (r - div - 0.5 * sig^2)dt
sigdt = sig * sqrt(dt)

sum_CT = 0
sum_CT2 = 0

for i = 0 to N do {for each simulation}
    St = S
next i

for j = 1 to M do {for each simulation}

    for i = 1 to N do {for each time step}
        ε = standard_normal_sample
        St = St * exp(nudt + sigdt * ε) {evolve the stock price}
    next i

    ST = exp(lnSt)
    CT = max(0, ST - K)
    sum_CT = sum_CT + CT
    sum_CT2 = sum_CT2 + CT * CT

next j

call_value = sum_CT / M * exp(-rT)
SD = sqrt((sum_CT2 - sum_CT * sum_CT / M) * exp(-2rT) / (M - 1))
SE = SD / sqrt(M)
```

13.2 Tree-Based Solver Pseudocode

This is the pseudocode for pricing a European call option using a tree-based solver, taken from [56].

```
initialise_parameter{K, T, S, sig, r, div, height}

# Compute constants
dt = T / height
u = exp(((r - div) * dt) + (v * sqrt(dt)))
d = exp(((r - div) * dt) - (v * sqrt(dt)))
pu = (exp((r - div) * dt) - d) / (u - d)
pd = 1 - pu
disc = exp(-r * dt)

# Initialise asset prices at maturity
for i = 0 to height
    St[i] = S * pow(u, height-i) * pow(d, i)
next i

# Initialise option values at maturity
for i = 0 to height
    C[i] = max(K - St[i], 0)
next i

# Traverse tree
for i = height-1 to 0
    for j = 0 to i
        C[j] = (dpu * C[j]) + (dpd * C[j+1])
    next j
next i
```

When pricing American options, the traversal loop becomes

```
# Traverse tree
for i = height-1 to 0
    for j = 0 to i
        C[j] = (dpu * C[j]) + (dpd * C[j+1])
        St[j] = St[j] / u
        C[j] = max(C[j], max(St[j] - K, 0))
    next j
next i
```

14. Appendix 3 – Original Project Design

14.1 Project Management

We will implement version control to maintain a backup of all code written and ensure that new parts of the system integrate seamlessly into the old system. A Github repository has been setup for this purpose. In addition, the contents of the PYNQ-Z2 board will be regularly backed up to Google Drive. Similarly, we will also backup all assignments and other documents to Google Drive.

We have decided to use Trello in order to manage the project workload. The tasks will be divided into four categories: to be completed, completed, in progress and on hold. This will allow us to manage these tasks and ensure that the project proceeds at an acceptable rate.

14.2 Experimental Design

This project begins by implementing a proof of concept on the PYNQ-Z2 development board. During this stage, we will implement a Monte Carlo algorithm to approximate the solutions to the Black-Scholes and Heston option pricing models. The Monte Carlo simulations performed at this stage will be small using only 100 paths.

Following this, we aim to scale our simulations and run them on the Alveo FPGA. In this case, we will run Monte Carlo simulations that have 2,500 paths or greater. As we move from the PYNQ-Z2 board to the Alveo FPGA, unit tests will be applied to ensure that the algorithms still work correctly. When performing these tests, we will use fixed starting parameters and number of simulated paths to ensure fair comparison. The answers from the Alveo FPGA implementation will then be compared to the PYNQ-Z2 implementation. As we are running Monte Carlo simulations, no two runs will yield the same answer; however, the variance of these answers should be minimal.

When conducting our experiments, we will run our algorithm multiple times with the same starting parameters. The exact number of times we run the algorithm will depend on its runtime. This is because, in the case of the Nimbix platform, we want to ensure that we account for outliers, but we don't want to run out of compute time. We will then take the average of the results over all runs to account for anomalous results.

14.3 Experiments to be Performed

We will perform a number of experiments to determine the run time, accuracy and energy use of the mentioned algorithms. We will measure the total run time of our algorithms, as well as measuring times relating to data transfers and kernel runtimes. These measurements will be taken by using Python's time function and C/C++ time function whilst working on the PYNQ board. Similarly, when working on the Nimbix cloud or Barkla, these measurements will be taken by using C/C++ time function and the OpenCL profiling functions.

When approximating the price of European options, we can determine the accuracy of our solution by comparing it directly with the closed-form solutions. This is not possible for American options as there is no closed-form solution. In this case, we will calculate the estimated standard error to determine the accuracy of our Monte Carlo solution. This is done by dividing the sample standard deviation by the square root of the number of paths.

We will use a USB tester such as to measure the energy usage of our PYNQ board. This will tell us the energy use of the whole board, rather than just the FPGA, but should

provide some insight into the efficiency of a hybrid CPU-FPGA implementation. On the Nimbix cloud or Barkla, we will use API queries to determine the energy usage. The specific API queries are still to be determined.

14.4 Analysis of Results

In the first stage of analysis, we will compare CPU implementations of options pricing algorithms against hybrid CPU-FPGA implementations of the same algorithms. We will analyse the run time and accuracy of these algorithms on both implementations as well as the energy used to run them. This analysis will allow us to either accept or reject the null hypothesis presented in section 4.1.1. Furthermore, by performing this analysis, we will satisfy the first primary aim of this project.

Following this, we will compare the hybrid CPU-FPGA implementations against each other. The purpose of this comparison is to help inform investors about which algorithm is best to implement. Throughout this analysis, we will consider the quality of solution from each algorithm, how quickly the algorithm executes and how much energy is used during execution.

Finally, we will compare the performance of these algorithms when using floating-point and fixed-point numbers. We will then analyse the trade-off between speed-up, accuracy and energy efficiency when using either floating or fixed-point numbers.

14.5 FPGA Options Pricing Library

The options library will provide users with access to the algorithms developed throughout this project. The host code will be written in C++, so we can follow an object-oriented design. In addition, we will use OpenCL to allow communication between the host and the device. *Figure 1* below outlines the architecture for this system.

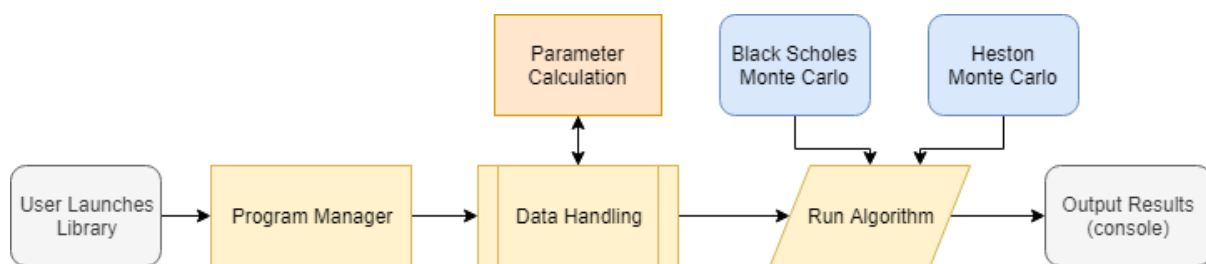


Figure 1: System Architecture

After launching the library, the user will be prompted by the ‘program manager’ to specify the algorithm they wish to use, and what type of option they wish to price. Following this, they will be prompted to enter the required parameters for the chosen algorithm. These parameters will be calculated from a historical dataset should one be provided. If one is not provided, the user must enter the parameters. Following this, the chosen algorithm will run. If enabled, both a CPU and hybrid CPU-FPGA implementation will run, otherwise just the hybrid implementation will run. Once the algorithm has finished running, the results of the run will be output to the console.

15. Appendix 4 - Trello Board

The Trello board is organized into four columns:

- In Progress**: Empty column with an "Add a card" button.
- On Hold**: Contains one card: "HLS Streams for Monte Carlo". Includes an "Add another card" button.
- To Do**: Contains seven cards:
 - FPGA Options Pricing Library
 - CPU US Monte Carlo Implementation
 - EU Binomial Tree UnderHeston Conditions
 - US Binomial Tree Under Heston Conditions
 - EU Monte Carlo Under Heston Conditions
 - US Monte Carlo Under Heston Conditions
 - Extend analysis for Asian optionsIncludes an "Add another card" button.
- Complete**: Contains eight cards:
 - CPU EU Monte Carlo Implementation
 - CPU US Binomial Tree Implementation
 - CPU EU Binomial Tree Implementation
 - First FPGA EU/US Binomial Tree Implementation
 - First FPGA Monte Carlo Implementation
 - Box-Muller Transform
 - Mersenne-Twister Algorithm
 - Taylor Expansion
 - Integrate Power Calculations into ForIncludes an "Add another card" button.

16. Appendix 5 – Zynq-7020 and Alveo u200 Design Flow

16.1 Zynq-7020

The Zynq-7020 FPGA used throughout this project was installed in a Pynq-Z2 board. Pynq is an open source project, developed by Xilinx, that provides the productivity benefits of Python to embedded systems designers. The designer first develops an IP in Vivado HLS, imports it into a block design in Vivado Design Suite and interacts with the generated bitstream file through a Jupyter notebook [42].

When designing an IP in Vivado HLS, we start by defining a top-level function, which is equivalent to the *main()* function in C. Following this, we must write a number of *interface* pragmas, which provide our IP with access to global memory. An example of this is shown below in *Figure 1*.

```
#pragma HLS INTERFACE m_axi port=output_r offset=slave bundle=gmem0
#pragma HLS INTERFACE s_axilite port=output_r bundle=control
#pragma HLS INTERFACE m_axi port=S offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=S bundle=control
#pragma HLS INTERFACE m_axi port=K offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=K bundle=control
#pragma HLS INTERFACE m_axi port=T offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=T bundle=control
#pragma HLS INTERFACE m_axi port=D offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=D bundle=control
#pragma HLS INTERFACE m_axi port=r offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=r bundle=control
#pragma HLS INTERFACE m_axi port=v offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=v bundle=control
#pragma HLS INTERFACE m_axi port=type_r offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=type_r bundle=control
#pragma HLS INTERFACE m_axi port=height offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=height bundle=control
#pragma HLS INTERFACE s_axilite port=n_options bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control
```

Figure 1: Defining Interface Pragmas in Vivado HLS

In this case, we have both *m_axi* and *s_axilite* ports. The *m_axi* ports are used for data input and output, whereas the *s_axilite* ports are used for configuring the IP at runtime [42]. With the top-level function and memory interfaces defined, we can write the rest of our algorithm as a normal C/C++ program. After writing our algorithm, the final stages are to synthesise our code and then export it as RTL. The synthesis stage ensures the correctness of our C/C++ program. When exporting the program as RTL, our C/C++ code gets converted into an HDL such as Verilog.

After exporting our design as RTL, we are ready to import the IP into the Vivado Design Suite. An example of an imported IP is shown in figure 1 below, where we can see the *m_axi* and *s_axilite* ports which were defined in our HLS code. This IP is then integrated into a block design, shown in figure 2, which describes how the different components on the Pynq-Z2 board interact with each other and is used when generating a bitstream.

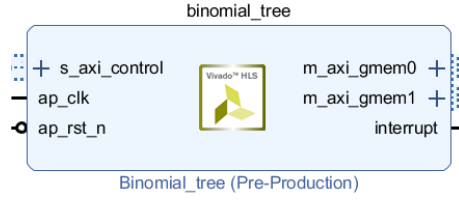


Figure 1: Binomial Tree IP in Vivado Design Suite

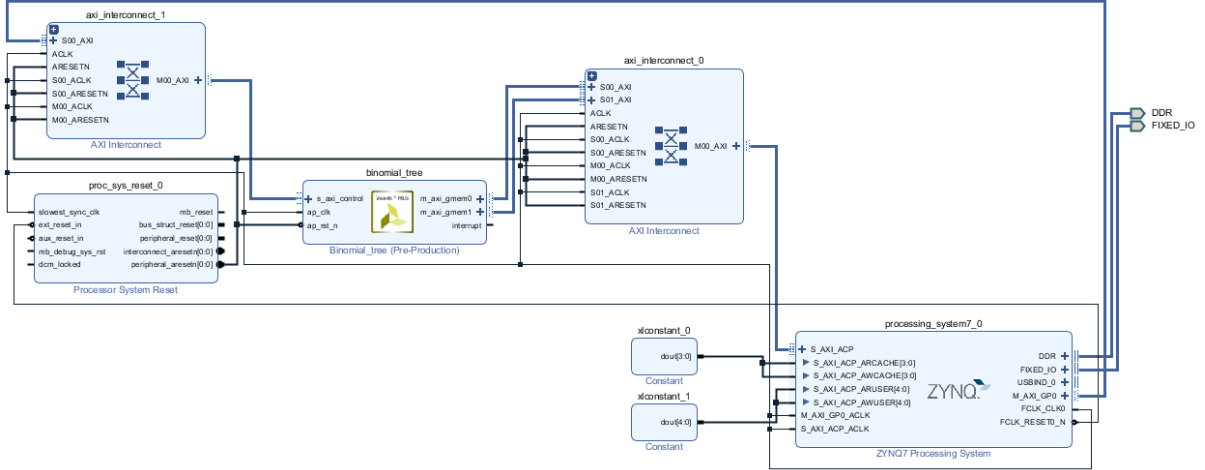


Figure 2: Binomial Tree Block Design in Vivado Design Suite

The final block designs for all algorithms implemented on the Zynq-7020 are similar in appearance to figure 2. In the above design, two AXI interconnects are used to allow our IP to communicate with the master and slave ports on the ZYNQ7 Processing System IP. The constant IPs that can be seen in the design are used to enforce cache coherency at a hardware level on the Zynq programmable logic [43]. Once the block design is complete, we generate a bitstream file which is then used to program the FPGA at runtime.

Once the block design is complete, we generate a bitstream file which is then used to program the FPGA at runtime. Now that the bitstream has been generated we are ready to work in the Pynq development environment. This requires a driver class for our IP, shown in figure 3 below, that defines read and write functions to interact with the register addresses found in the header file after synthesis.

```

class BinomialTreeDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

        bindto = ['xilinx.com:hls:binomial_tree:1.0']

        @property
        def status(self):
            return self.read(0x00)

        @status.setter
        def status(self, value):
            self.write(0x00, value)

        @property
        def output(self):
            return self.read(0x10)

        @output.setter
        def output(self, value):
            self.write(0x10, value)

        @property
        def spot_price(self):
            return self.read(0x18)

        @spot_price.setter
        def spot_price(self, value):
            self.write(0x18, value)

```

Figure 3: Binomial Tree Driver Class

```

if (BinomialTree.status == ap_idle) or (BinomialTree.status == ap_ready):
    BinomialTree.status = ap_start

while(status != ap_idle):
    status = BinomialTree.status

```

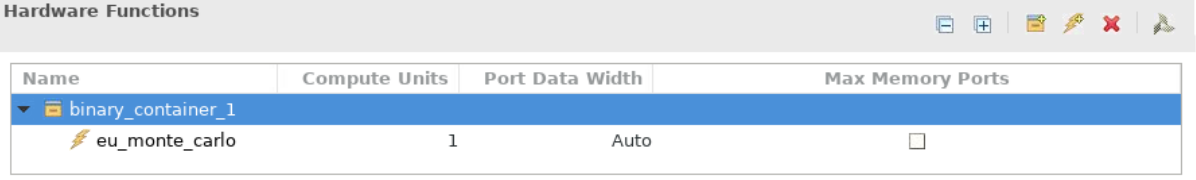
Figure 4: Running Algorithm on Zynq-7020

After using these functions to transfer data to the FPGA, we are ready to run our IP as shown in figure 4. We first check that the *status* register is set to *ap_idle*, if this is the case, we can set the register to *ap_start* which tells the IP to run. We then enter a while loop that polls the *status* register to determine when the algorithm has finished running. Once finished, we can read the output register and print the result to the client.

16.2 Alveo u200

When porting our algorithms to run on the Alveo u200, we were able to reuse the HLS code developed in section 6.1. The only thing that changed was how we interacted with the device from our host. In this case, OpenCL was used to allow the host to communicate with the device. The host files used were based upon Xilinx's SDAccel Examples Github repository [44].

First, we create an SDAccel Application Project which the HLS files and host code are imported into. Following this, we have to add a binary container to the project and define the top-level function as shown in figure 5 below.



The screenshot shows the 'Hardware Functions' window in SDAccel. It contains a table with the following data:

Name	Compute Units	Port Data Width	Max Memory Ports
binary_container_1			
eu_monte_carlo	1	Auto	<input type="checkbox"/>

Figure 5: Binary Container and Top-level Hardware Function in SDAccel

We are now ready to build our project. The SDAccel environment provides three build modes: software emulation, hardware emulation and hardware. Software emulation allows users to verify the correctness of their code whereas hardware emulation allows users to verify that their design is fit for hardware. These tools allow users to quickly develop and verify their designs. After running and verifying the software and hardware emulation, we are ready to build our design for hardware. This is a lengthy process that tends to take two hours or more. After compilation the design can be ran on the FPGA from the SDAccel environment or command line.

17 Appendix 6 – Supporting Figures

17.1 European Binomial Tree

Zynq-7020 vs Cortex A9 (1 option, varying depth)												
Platform	# of Options	Depth	Time Taken (s)							Speedup		
			1	2	3	4	5	Average	Min	Max	Unoptimised	Optimised
Cortex-A9 - Unoptimised	1	5000	0.86577	0.86536	0.86550	0.86559	0.86551	0.86554	0.86536	0.86577	-	-
Cortex-A9 - Unoptimised	1	10000	3.45882	3.43635	3.45593	3.45835	3.41056	3.44400	3.41056	3.45882	-	-
Cortex-A9 - Unoptimised	1	15000	7.69119	7.70050	7.69971	7.69650	7.69909	7.69740	7.69119	7.70050	-	-
Cortex-A9 - Unoptimised	1	20000	13.59668	13.59142	13.51107	13.55888	13.55763	13.56313	13.51107	13.59668	-	-
Cortex-A9 - Unoptimised	1	25000	20.98550	21.09110	21.05971	21.05091	21.05874	21.04919	20.98550	21.09110	-	-
Cortex-A9 - Unoptimised	1	30000	30.20419	30.17912	30.19275	30.24043	30.20333	30.20396	30.17912	30.24043	-	-
Cortex-A9 - Optimised	1	5000	0.21916	0.21920	0.21911	0.21916	0.21921	0.21917	0.21911	0.21921	-	-
Cortex-A9 - Optimised	1	10000	0.90563	0.90082	0.88586	0.90561	0.88142	0.89587	0.88142	0.90563	-	-
Cortex-A9 - Optimised	1	15000	2.13841	2.13336	2.13214	2.13318	2.13135	2.13369	2.13135	2.13841	-	-
Cortex-A9 - Optimised	1	20000	4.03617	4.00152	3.99445	4.00744	3.99995	4.00790	3.99445	4.03617	-	-
Cortex-A9 - Optimised	1	25000	6.44649	6.44253	6.42342	6.47566	6.42562	6.44274	6.42342	6.47566	-	-
Cortex-A9 - Optimised	1	30000	9.43090	9.44510	9.42491	9.41304	9.43111	9.42901	9.41304	9.44510	-	-
Zynq Z7020 - Unoptimised	1	5000	1.50351	1.50351	1.50352	1.50345	1.50359	1.50352	1.50345	1.50359	0.575679976	0.145789369
Zynq Z7020 - Unoptimised	1	10000	6.00553	6.00587	6.00562	6.00556	6.00553	6.00562	6.00553	6.00587	0.573462799	0.146768062
Zynq Z7020 - Unoptimised	1	15000	13.50739	13.50744	13.50742	13.50767	13.50764	13.50751	13.50739	13.50767	0.569860447	0.15778854
Zynq Z7020 - Unoptimised	1	20000	24.00951	24.00945	24.00949	24.00943	24.00941	24.00946	24.00941	24.00951	0.564907945	0.166599089
Zynq Z7020 - Unoptimised	1	25000	37.51151	37.51134	37.51138	37.51127	37.51134	37.51137	37.51127	37.51151	0.561141649	0.171297973
Zynq Z7020 - Unoptimised	1	30000	54.01423	54.01406	54.01380	54.01357	54.01354	54.01384	54.01354	54.01423	0.559189322	0.174606327
Zynq Z7020 - Optimised	1	5000	0.12615	0.12617	0.12613	0.12612	0.12612	0.12614	0.12612	0.12617	6.861939003	1.737542513
Zynq Z7020 - Optimised	1	10000	0.50206	0.50205	0.50206	0.50206	0.50206	0.50206	0.50205	0.50206	6.859759741	1.784388031
Zynq Z7020 - Optimised	1	15000	1.12801	1.12797	1.12797	1.12796	1.12797	1.12798	1.12796	1.12801	6.82406383	1.891603598
Zynq Z7020 - Optimised	1	20000	2.00390	2.00391	2.00391	2.00390	2.00392	2.00391	2.00390	2.00392	6.768342413	2.000044214
Zynq Z7020 - Optimised	1	25000	3.12982	3.12982	3.12984	3.12982	3.12983	3.12983	3.12982	3.12984	6.725355815	2.058498403
Zynq Z7020 - Optimised	1	30000	4.50576	4.50575	4.50575	4.50576	4.50575	4.50575	4.50575	4.50576	6.703419147	2.092659592

Alveo u200 vs Intel Cortex E5649 (1 Option, Varying Steps)												
Platform	# of Options	Depth	Time Taken (s)							Speedup		
			1	2	3	4	5	Average	Min	Max	Unoptimised	Optimised
Intel Xeon E5649 - Unoptimised	1	5000	0.45785	0.47561	0.47625	0.47908	0.47830	0.47342	0.45785	0.47908	-	-
Intel Xeon E5649 - Unoptimised	1	10000	2.10237	2.09911	2.13623	2.10161	2.10728	2.10932	2.09911	2.13623	-	-
Intel Xeon E5649 - Unoptimised	1	15000	4.98641	4.60739	4.60381	4.60465	4.60351	4.68115	4.60351	4.98641	-	-
Intel Xeon E5649 - Unoptimised	1	20000	8.23715	8.23070	8.23159	8.22647	8.23239	8.23166	8.22647	8.23715	-	-
Intel Xeon E5649 - Unoptimised	1	25000	12.96190	12.99067	13.32074	13.40638	3.60544	11.25702	3.60544	13.40638	-	-
Intel Xeon E5649 - Unoptimised	1	30000	19.22006	18.89323	18.90218	18.89998	19.11759	19.00661	18.89323	19.22006	-	-
Intel Xeon E5649 - Optimised	1	5000	0.14412	0.14561	0.14910	0.14947	0.14616	0.14689	0.14412	0.14947	-	-
Intel Xeon E5649 - Optimised	1	10000	0.56578	0.56566	0.56175	0.56744	0.56504	0.56513	0.56175	0.56744	-	-
Intel Xeon E5649 - Optimised	1	15000	1.28321	1.31524	1.31524	1.28854	1.28204	1.29685	1.28204	1.31524	-	-
Intel Xeon E5649 - Optimised	1	20000	2.37080	2.37347	2.36926	2.36819	2.36555	2.36946	2.36555	2.37347	-	-
Intel Xeon E5649 - Optimised	1	25000	3.60544	3.60434	3.59596	3.60404	3.60511	3.60298	3.59596	3.60544	-	-
Intel Xeon E5649 - Optimised	1	30000	5.13804	5.13518	5.13684	5.13448	5.13593	5.13609	5.13448	5.13804	-	-
Alveo u200 - Unoptimised	1	5000	0.08474	0.08469	0.08473	0.08468	0.08471	0.08471	0.08468	0.08474	5.588716832	1.734065604
Alveo u200 - Unoptimised	1	10000	0.33559	0.33557	0.33555	0.33556	0.33567	0.33559	0.33555	0.33567	6.285462828	1.684014834
Alveo u200 - Unoptimised	1	15000	0.75320	0.75309	0.75307	0.75314	0.75313	0.75313	0.75307	0.75320	6.215629796	1.721958199
Alveo u200 - Unoptimised	1	20000	1.33729	1.33733	1.33731	1.33729	1.33729	1.33730	1.33729	1.33733	6.155425044	1.771818405
Alveo u200 - Unoptimised	1	25000	2.08821	2.08810	2.08816	2.08821	2.08812	2.08816	2.08810	2.08821	5.390883141	1.725431234
Alveo u200 - Unoptimised	1	30000	3.00565	3.00565	3.00559	3.00557	3.00567	3.00563	3.00557	3.00567	6.323677351	1.708826374
Alveo u200 - Optimised	1	5000	0.04273	0.04268	0.04264	0.04273	0.04266	0.04269	0.04264	0.04273	11.09057917	3.441182019
Alveo u200 - Optimised	1	10000	0.16823	0.16824	0.16822	0.16832	0.16824	0.16825	0.16822	0.16832	12.53690594	3.358915031
Alveo u200 - Optimised	1	15000	0.37703	0.37705	0.37712	0.37711	0.37711	0.37708	0.37703	0.37712	12.41407084	3.43915448
Alveo u200 - Optimised	1	20000	0.66925	0.66928	0.66923	0.66935	0.66927	0.66928	0.66923	0.66935	12.29936758	3.540331608
Alveo u200 - Optimised	1	25000	1.04483	1.04481	1.04480	1.04483	1.04479	1.04481	1.04479	1.04483	10.77423626	3.448452373
Alveo u200 - Optimised	1	30000	1.50364	1.50363	1.50361	1.50367	1.50364	1.50364	1.50361	1.50367	12.64042429	3.415779967

17.2 US Binomial Tree

Zynq-7020 vs Cortex A9 (1 option, varying depth)												
Platform	# of Options	# of Steps	Time Taken (s)					Average	Min	Max	Speedup	
			1	2	3	4	5				Unoptimised	Optimised
Cortex-A9 - Unoptimised	1	5000	2.40563	2.40311	2.40412	2.40290	2.40421	2.40399	2.40290	2.40563	-	-
Cortex-A9 - Unoptimised	1	10000	9.70573	9.69801	9.69911	9.69877	9.70611	9.70155	9.69801	9.70611	-	-
Cortex-A9 - Unoptimised	1	15000	21.88975	21.88464	21.88628	21.88752	21.88243	21.88613	21.88243	21.88975	-	-
Cortex-A9 - Unoptimised	1	20000	39.41257	39.41329	39.41614	39.40258	39.40512	39.40994	39.40258	39.41614	-	-
Cortex-A9 - Unoptimised	1	25000	60.90524	60.91793	60.91740	60.90192	60.91643	60.91178	60.90192	60.91793	-	-
Cortex-A9 - Unoptimised	1	30000	87.74519	87.77076	87.76148	87.78271	87.81878	87.77578	87.74519	87.81878	-	-
Cortex-A9 - Optimised	1	5000	0.63707	0.62877	0.61802	0.62407	0.61363	0.62431	0.61363	0.63707	-	-
Cortex-A9 - Optimised	1	10000	2.66539	2.66577	2.66054	2.66438	2.66108	2.66343	2.66054	2.66577	-	-
Cortex-A9 - Optimised	1	15000	6.17753	6.16331	6.16534	6.16981	6.17506	6.17021	6.16331	6.17753	-	-
Cortex-A9 - Optimised	1	20000	11.30451	11.29616	11.31415	11.29508	11.29923	11.30182	11.29508	11.31415	-	-
Cortex-A9 - Optimised	1	25000	17.42825	17.43536	17.43826	17.43578	17.43370	17.43427	17.42825	17.43826	-	-
Cortex-A9 - Optimised	1	30000	25.20698	25.17235	25.17132	25.17961	25.18185	25.18242	25.17132	25.20698	-	-
Zynq Z7020 - Unoptimised	1	5000	3.50232	3.50232	3.50233	3.50230	3.50231	3.50231	3.50230	3.50233	0.68640	0.17826
Zynq Z7020 - Unoptimised	1	10000	14.00442	14.00441	14.00441	14.00441	14.00442	14.00441	14.00441	14.00442	0.69275	0.19019
Zynq Z7020 - Unoptimised	1	15000	31.50666	31.50667	31.50665	31.50663	31.50664	31.50665	31.50663	31.50667	0.69465	0.19584
Zynq Z7020 - Unoptimised	1	20000	56.00888	56.00886	56.00885	56.00884	56.00885	56.00886	56.00884	56.00888	0.70364	0.20179
Zynq Z7020 - Unoptimised	1	25000	87.51088	87.51122	87.51119	87.51109	87.51111	87.51110	87.51088	87.51122	0.69605	0.19922
Zynq Z7020 - Unoptimised	1	30000	126.01346	126.01304	126.01304	126.01307	126.01308	126.01314	126.01304	126.01346	0.69656	0.19984
Zynq Z7020 - Optimised	1	5000	0.12696	0.12696	0.12696	0.12696	0.12695	0.12696	0.12695	0.12696	18.93534	4.91747
Zynq Z7020 - Optimised	1	10000	0.50376	0.50369	0.50369	0.50370	0.50370	0.50371	0.50369	0.50376	19.26021	5.28763
Zynq Z7020 - Optimised	1	15000	1.13047	1.13046	1.13046	1.13049	1.13046	1.13047	1.13046	1.13049	19.36026	5.45811
Zynq Z7020 - Optimised	1	20000	2.00720	2.00720	2.00721	2.00720	2.00722	2.00720	2.00720	2.00722	19.63424	5.63063
Zynq Z7020 - Optimised	1	25000	3.13397	3.13397	3.13397	3.13397	3.13397	3.13397	3.13397	3.13398	19.43596	5.56299
Zynq Z7020 - Optimised	1	30000	4.51071	4.51071	4.51070	4.51072	4.51072	4.51071	4.51070	4.51072	19.45941	5.58280

ALVEO vs INTEL (1 Option, Varying Depth)												
Platform	# of Options	# of Steps	Time Taken (s)					Average	Min	Max	Speedup	
			1	2	3	4	5				Unoptimised	Optimised
Intel Xeon E5649 - Unoptimised	1	5000	0.17384	0.17485	0.17552	0.17273	0.15328	0.17004	0.15328	0.17552	-	-
Intel Xeon E5649 - Unoptimised	1	10000	0.58573	0.58967	0.59563	0.59041	0.59782	0.59185	0.58573	0.59782	-	-
Intel Xeon E5649 - Unoptimised	1	15000	1.41807	1.40876	1.40820	1.38936	1.41578	1.40803	1.38936	1.41807	-	-
Intel Xeon E5649 - Unoptimised	1	20000	2.43171	2.43589	2.42802	2.43685	2.43126	2.43275	2.42802	2.43685	-	-
Intel Xeon E5649 - Unoptimised	1	25000	3.59513	3.81899	3.71960	3.58705	3.59277	3.66271	3.58705	3.81899	-	-
Intel Xeon E5649 - Unoptimised	1	30000	4.67090	4.66391	4.66401	4.66852	4.66649	4.66676	4.66391	4.67090	-	-
Intel Xeon E5649 - Optimised	1	5000	0.07401	0.09790	0.09689	0.09935	0.10329	0.09429	0.07401	0.10329	-	-
Intel Xeon E5649 - Optimised	1	10000	0.31249	0.31557	0.31162	0.30978	0.31255	0.31240	0.30978	0.31557	-	-
Intel Xeon E5649 - Optimised	1	15000	0.62801	0.65257	0.63024	0.63019	0.65926	0.64005	0.62801	0.65926	-	-
Intel Xeon E5649 - Optimised	1	20000	1.14645	1.14910	1.13631	1.14882	1.14695	1.14552	1.13631	1.14910	-	-
Intel Xeon E5649 - Optimised	1	25000	1.77241	1.77076	1.77105	1.73972	1.75497	1.76178	1.73972	1.77241	-	-
Intel Xeon E5649 - Optimised	1	30000	2.45401	2.48675	2.44058	2.48333	2.47994	2.46892	2.44058	2.48675	-	-
Alveo u200 - Unoptimised	1	5000	0.08525	0.08521	0.08518	0.08533	0.08523	0.08524	0.08518	0.08533	1.994857257	1.10612
Alveo u200 - Unoptimised	1	10000	0.33660	0.33672	0.33658	0.33660	0.33664	0.33663	0.33658	0.33672	1.758165889	0.92803
Alveo u200 - Unoptimised	1	15000	0.75469	0.75466	0.75468	0.75465	0.75470	0.75467	0.75465	0.75470	1.865749386	0.84812
Alveo u200 - Unoptimised	1	20000	1.33944	1.33939	1.33949	1.33944	1.33938	1.33943	1.33938	1.33949	1.816261579	0.85523
Alveo u200 - Unoptimised	1	25000	2.09075	2.09076	2.09077	2.09080	2.09081	2.09078	2.09075	2.09081	1.751838535	0.84264
Alveo u200 - Unoptimised	1	30000	3.00883	3.00884	3.00881	3.00889	3.00873	3.00882	3.00873	3.00889	1.551027946	0.82056
Alveo u200 - Optimised	1	5000	0.04315	0.04324	0.04314	0.04315	0.04327	0.04319	0.04314	0.04327	3.937352388	2.18320
Alveo u200 - Optimised	1	10000	0.16930	0.16917	0.16922	0.16926	0.16927	0.16925	0.16917	0.16930	3.49699076	1.84586
Alveo u200 - Optimised	1	15000	0.37862	0.37855	0.37858	0.37859	0.37854	0.37858	0.37854	0.37862	3.719275983	1.69068
Alveo u200 - Optimised	1	20000	0.67124	0.67124	0.67126	0.67131	0.67122	0.67125	0.67122	0.67131	3.624181904	1.70654
Alveo u200 - Optimised	1	25000	1.04728	1.04723	1.04725	1.04730	1.04724	1.04726	1.04723	1.04730	3.497419552	1.68228
Alveo u200 - Optimised	1	30000	1.50667	1.50653	1.50666	1.50664	1.50663	1.50663	1.50653	1.50667	3.097493738	1.63871

17.3 Combined Binomial Tree

ALVEO vs INTEL (1 Option, Varying Steps)												
Platform	# of Options	# of Steps	Time Taken (s)							Speedup		
			1	2	3	4	5	Average	Min	Max	Unoptimised	Optimised
Intel Xeon E5649 - Unoptimised	1	5000	0.61711	0.61818	0.61921	0.61958	0.61603	0.61802	0.61603	0.61958	-	-
Intel Xeon E5649 - Unoptimised	1	10000	2.59083	2.58903	2.59041	2.59619	2.58976	2.59124	2.58903	2.59619	-	-
Intel Xeon E5649 - Unoptimised	1	15000	5.78608	5.78828	5.81405	5.79085	5.79234	5.79432	5.78608	5.81405	-	-
Intel Xeon E5649 - Unoptimised	1	20000	10.28546	10.33180	10.33496	10.29659	10.29351	10.30846	10.28546	10.33496	-	-
Intel Xeon E5649 - Unoptimised	1	25000	16.65781	16.25471	16.24904	16.21849	16.25177	16.32636	16.21849	16.65781	-	-
Intel Xeon E5649 - Unoptimised	1	30000	23.23902	23.24938	23.31878	23.25646	23.58140	23.32901	23.23902	23.58140	-	-
Intel Xeon E5649 - Optimised	1	5000	0.19206	0.21562	0.21773	0.21569	0.22160	0.21254	0.19206	0.22160	-	-
Intel Xeon E5649 - Optimised	1	10000	0.84481	0.84219	0.84625	0.84616	0.84601	0.84508	0.84219	0.84625	-	-
Intel Xeon E5649 - Optimised	1	15000	1.93885	1.93670	1.94058	1.93904	1.93978	1.93899	1.93670	1.94058	-	-
Intel Xeon E5649 - Optimised	1	20000	3.32838	3.27759	3.32542	3.30298	3.32656	3.31218	3.27759	3.32838	-	-
Intel Xeon E5649 - Optimised	1	25000	5.10219	5.06108	5.07458	5.08841	5.09957	5.08517	5.06108	5.10219	-	-
Intel Xeon E5649 - Optimised	1	30000	7.31885	7.51423	7.26228	7.29699	7.25717	7.32990	7.25717	7.51423	-	-
Alveo u200	1	5000	0.04318	0.04321	0.04326	0.04324	0.04319	0.04322	0.04318	0.04326	14.30048972	4.9179729
Alveo u200	1	10000	0.16921	0.16920	0.16930	0.16927	0.16929	0.16925	0.16920	0.16930	15.30982948	4.9930065
Alveo u200	1	15000	0.37858	0.37856	0.37862	0.37861	0.37860	0.37860	0.37856	0.37862	15.30478432	5.1215352
Alveo u200	1	20000	0.67130	0.67130	0.67127	0.67129	0.67129	0.67129	0.67127	0.67130	15.35612482	4.9340352
Alveo u200	1	25000	1.04727	1.04732	1.04729	1.04733	1.04728	1.04730	1.04727	1.04733	15.58906565	4.8555195
Alveo u200	1	30000	1.50665	1.50665	1.50666	1.50667	1.50667	1.50667	1.50665	1.50673	15.48380788	4.8649658

17.4 Monte Carlo

PYNQ vs ARM (Varying # paths, fixed steps)												
Platform	# of Paths	# of Steps	Time Taken (s)							Speedup		
			1	2	3	4	5	Average	Min	Max	Unoptimised	Optimised
Cortex-A9	100	100	0.01757	0.01762	0.01761	0.01754	0.01750	0.01757	0.01750	0.01762	-	-
Cortex-A9	1000	100	0.17144	0.17319	0.17300	0.17290	0.171576	0.17242	0.17144	0.17319	-	-
Cortex-A9	10000	100	1.72717	1.71261	1.72999	1.72745	1.72855	1.72515	1.71261	1.72999	-	-
Cortex-A9	100000	100	17.41424	17.25345	17.20711	17.27068	17.23626	17.27635	17.20711	17.41424	-	-
Cortex-A9	1000000	100	175.24926	173.03333	174.12351	173.69360	172.68856	173.75765	172.68856	175.24926	-	-
Zynq Z7020 - Unoptimised	100	100	0.01519	0.01521	0.01527	0.01518	0.01518	0.01521	0.01518	0.01527	1.155259489	-
Zynq Z7020 - Unoptimised	1000	100	0.14980	0.14978	0.14979	0.14980	0.14980	0.14979	0.14978	0.14980	1.151085784	-
Zynq Z7020 - Unoptimised	10000	100	1.49582	1.49578	1.49578	1.49578	1.49576	1.49578	1.49576	1.49582	1.153342647	-
Zynq Z7020 - Unoptimised	100000	100	14.95554	14.95517	14.95525	14.95528	14.95533	14.95531	14.95517	14.95554	1.1551978	-
Zynq Z7020 - Unoptimised	1000000	100	149.55240	149.55015	149.54889	149.55042	149.55122	149.55062	149.54889	149.55240	1.161865182	-

ALVEO vs INTEL (1 Option, Varying Depth)												
Platform	# of Paths	# of Steps	Time Taken (s)							Speedup		
			1	2	3	4	5	Average	Min	Max	Unoptimised	Optimised
Intel Xeon E5649 - Unoptimised	100	100	0.00502	0.00504	0.00503	0.00502	0.00507	0.00504	0.00502	0.00507	-	-
Intel Xeon E5649 - Unoptimised	1000	100	0.04726	0.04738	0.04744	0.04762	0.045409	0.04702	0.04541	0.04762	-	-
Intel Xeon E5649 - Unoptimised	10000	100	0.34018	0.33613	0.33949	0.31743	0.33926	0.33450	0.31743	0.34018	-	-
Intel Xeon E5649 - Unoptimised	100000	100	2.96234	2.97870	2.97359	2.99220	3.17460	3.01628	2.96234	3.17460	-	-
Intel Xeon E5649 - Unoptimised	1000000	100	27.06692	26.65196	26.94075	27.15231	26.86714	26.93582	26.65196	27.15231	-	-
Intel Xeon E5649 - Optimised	100	100	0.00379	0.00378	0.00370	0.00380	0.00380	0.00378	0.00370	0.00380	-	-
Intel Xeon E5649 - Optimised	1000	100	0.03556	0.03617	0.03623	0.03620	0.03382	0.03560	0.03382	0.03623	-	-
Intel Xeon E5649 - Optimised	10000	100	0.26291	0.26067	0.25837	0.25495	0.26783	0.26094	0.25495	0.26783	-	-
Intel Xeon E5649 - Optimised	100000	100	2.25671	2.30523	2.29048	2.29649	2.30113	2.29001	2.25671	2.30523	-	-
Intel Xeon E5649 - Optimised	1000000	100	20.19618	20.14883	20.15809	20.15820	20.19883	20.17203	20.14883	20.19883	-	-
Alveo u200 - Unoptimised	100	100	0.00916	0.00921	0.00928	0.00920	0.00923	0.00922	0.00916	0.00928	0.54649	0.40972
Alveo u200 - Unoptimised	1000	100	0.08757	0.08761	0.08758	0.08757	0.08761	0.08759	0.08757	0.08761	0.53684	0.40639
Alveo u200 - Unoptimised	10000	100	0.87092	0.87098	0.87097	0.87102	0.87093	0.87096	0.87092	0.87102	0.38406	0.29960
Alveo u200 - Unoptimised	100000	100	8.70440	8.70439	8.70440	8.70435	8.70438	8.70438	8.70435	8.70440	0.34652	0.26309
Alveo u200 - Unoptimised	1000000	100	87.03890	87.03886	87.03888	87.03881	87.03888	87.03887	87.03881	87.03890	0.30947	0.23176
Alveo u200 - Optimised	100	100	0.02256	0.02260	0.02255	0.02254	0.02249	0.02255	0.02249	0.02260	0.22335	0.16745
Alveo u200 - Optimised	1000	100	0.22107	0.22123	0.22111	0.22115	0.22114	0.22114	0.22107	0.22123	0.21263	0.16096
Alveo u200 - Optimised	10000	100	2.20649	2.20652	2.20651	2.20652	2.20658	2.20652	2.20649	2.20658	0.15159	0.11826
Alveo u200 - Optimised	100000	100	22.06053	22.06057	22.06052	22.06055	22.06061	22.06056	22.06052	22.06061	0.13673	0.10381
Alveo u200 - Optimised	1000000	100	220.60091	220.60072	220.60085	220.60088	220.60085	220.60084	220.60072	220.60091	0.12210	0.09144

18. Appendix 7 – User Guide

18.1. Introduction

18.1.1 Overview

This user guide sets out to inform users how to build and execute the code listed in [1]. This collection of code was developed as part of a Master’s thesis at the University of Liverpool. The supporting dissertation and results are also listed in [1].

18.1.2 Requirements

Firstly, users must clone the Github repository listed in [1]. Following this, users must have the following:

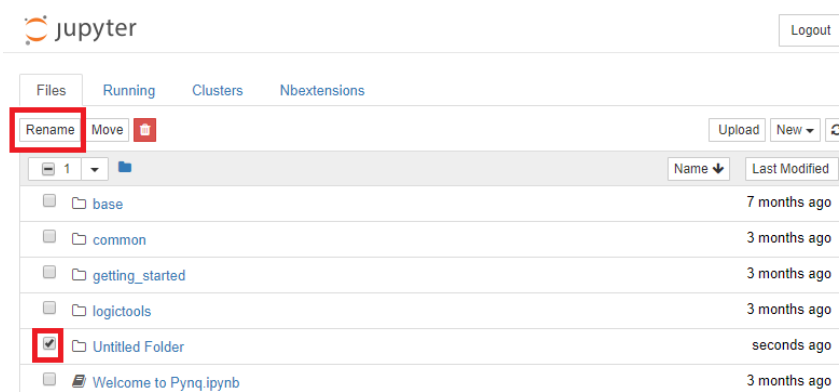
- A Pynq-Z2 board with the relevant board image installed. Instructions on how to install this board image can be found at [2].
- An Alveo u200 mounted and setup for your system. Installation instructions for the Alveo u200 card can be found at [3].

18.2. Pynq-Z2

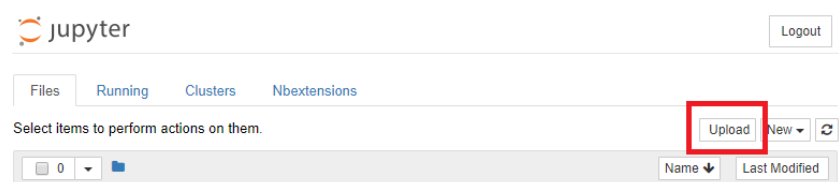
After installing the Pynq-Z2 board image, navigate to <http://pynq> in your web browser which will open the Jupyter environment. From here, click the “new” button towards the top right-hand side of the environment and then select “folder”.



We can rename this folder by selecting the check box to the left of it and then clicking “rename” towards the top left-hand side of the environment.



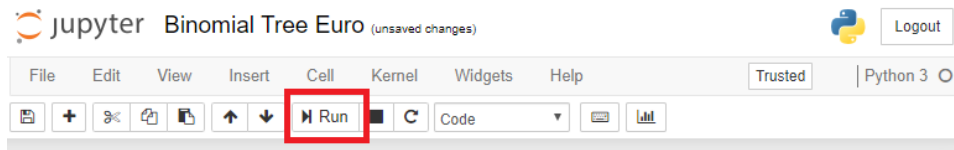
Rename the folder to the name of the algorithm you are going to use, e.g. European Binomial Tree. Navigate into the folder by clicking it. You can then upload the algorithm to the Pynq-Z2 by clicking on the “upload” button towards the top right-hand side of the environment.



Navigate to the cloned Github repository on your computer and navigate into the folder of the algorithm you wish to use. Navigate to the “Pynq-Z2” folder and then upload **all** of the contents in this folder.

You can now run the algorithm through the Jupyter environment or through the command line. To run in the Jupyter environment, click on the `.ipynb` file that you uploaded. This will

launch the Jupyter notebook, the algorithm can then be run by clicking the “Run” button towards the top of the notebook.



You will have to click run for each cell in the notebook, alternatively you can press *shift + enter* to execute each cell in the notebook. To execute via command line, first navigate to the algorithms folder. From here, you can execute the *.py* file that you uploaded, for example:

sudo python3 eubinomialtree.py

The Python file **must** be executed with the “suffix” prefix. If a number of zeroes appear in the output, simply execute the file again as the cache coherency on the Pynq-Z2 will have failed.

18.2.1 Binomial Tree

A file called *option_data.txt* is included for the Binomial Trees algorithm. Here, you can enter data for up to 25 options. The price for all options will then be returned at the end in the form of an array.

```

1 # Option pricing input file
2 # -----
3 # You may enter data for up to 25 options, any extra will be ignored
4 #
5 # Info
6 # S - Stock Price      (float) Underlying stock price
7 # K - Strike Price     (float) Options strike price
8 # T - Time to Maturity (float) Time (in years) until the option expires
9 # D - Dividend Yield   (float) Eg for 5% enter "5" and not "0.05"
10 # r - Risk-free Rate   (float) Eg for 5% enter "5" and not "0.05"
11 # v - Volatility        (float) Eg for 20% enter "20" and not "0.2"
12 # type                 (int)  The type of option, enter "0" for call and "1" for put
13 # height               (int)  The height of the tree
14 #
15 # Structure input data
16 # S,K,T,D,r,v,type,height
17 50,50,1,0,0.05,0.20,1,30000

```

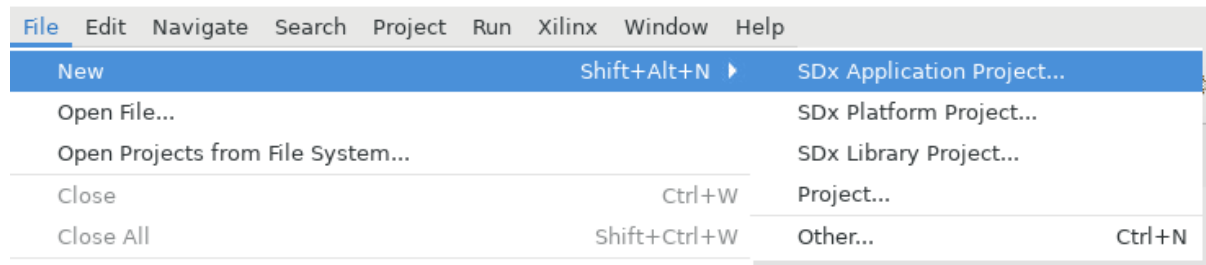
18.2.2 Monte Carlo

For the Monte Carlo simulation, the stock data and simulation data are set in the notebook/Python file themselves. The data is set in cell 5 of the notebook or between lines 121 and 130 of the Python file. These values must be set before you execute the notebook or Python file.

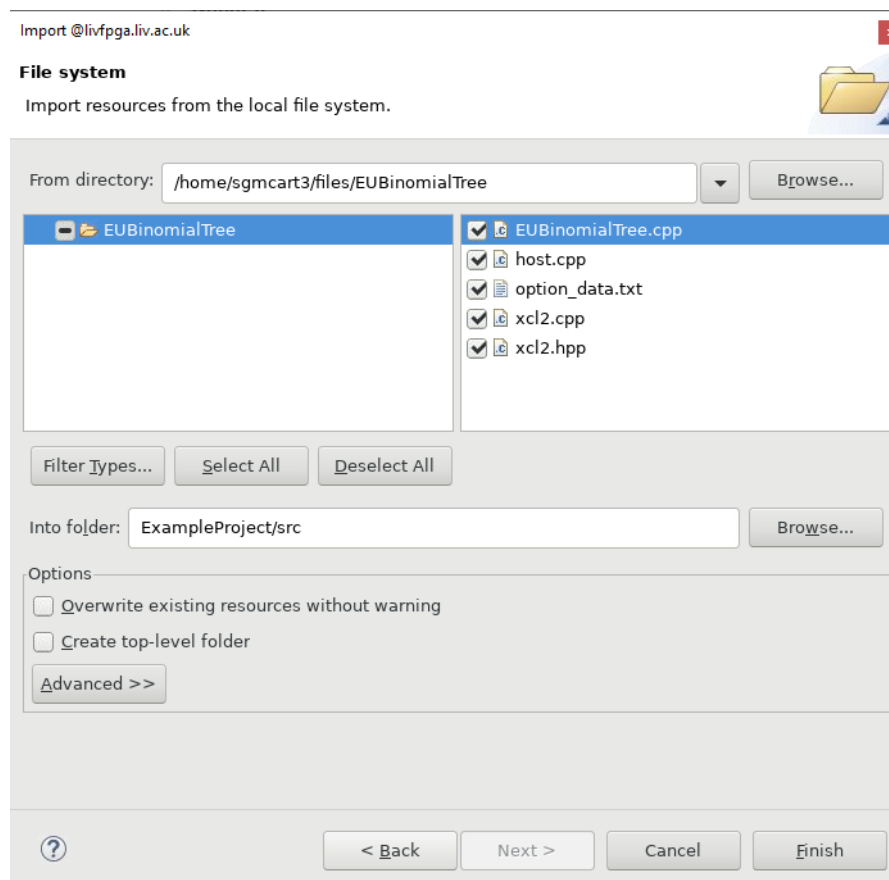
```
S[0] = 50  
K[0] = 50  
T[0] = 1  
D[0] = 0  
r[0] = 0.05  
v[0] = 0.20  
type_r = 1  
M = 1000000  
N = 100
```


18.3. Alveo u200

Start by creating a new SDAccel Application Project, this is done by clicking file -> new -> SDAccel Application Project. Ensure that you select the Alveo u200 platform and create an empty application.

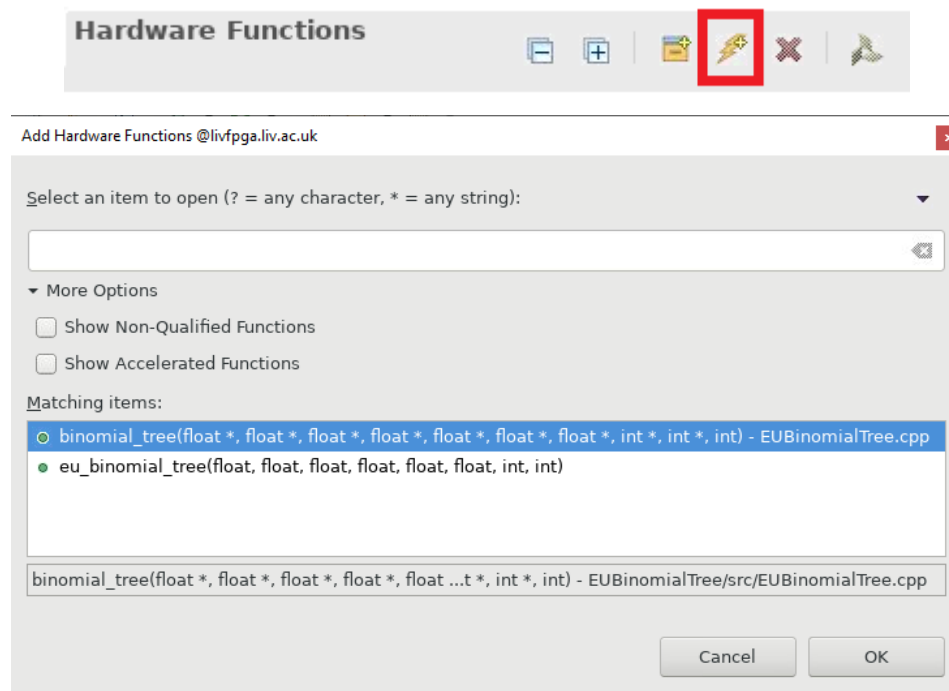


Following this, we are ready to import the source files into the project. If you are working on a remote server, you must first upload the relevant SDAccel and HLS files from the Github repository (NOTE: for the Monte Carlo algorithm there is a separate HLS file for the Pynq-Z2 and Alveo u200 implementation). Following this, right click on the “src” folder of the project and select “import”. Then select “General -> File System” and navigate to where the source files are on your system. Highlight them and click “finish”.

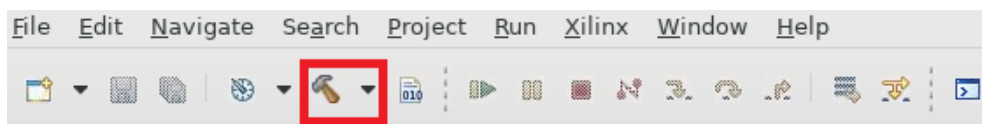


Now add a hardware function to the project. This is done by clicking the lightning icon next to the “Hardware Functions” title. Select the name of the top level function for the algorithm

– this is always the name of the type of algorithm you are running, for example if running a Binomial Tree algorithm, the top-level function is “binomial_tree”.



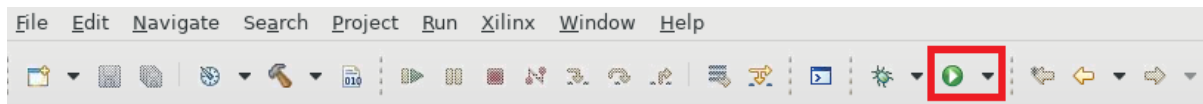
The project is now ready to be built. The C code and hardware design can be verified by selecting either the *sw_emu* or *hw_emu* build configuration. To build for the FPGA, select *System* (NOTE: building for the FPGA takes a minimum of 3 hours for the Binomial Tree algorithms and 10 hours for the Monte Carlo algorithms). After selecting the build configuration, click on the hammer icon in the toolbar.



After the project is built, select the “Run” menu on the toolbar and then select “Run Configurations”. Select the Arguments tab and ensure that “Automatically add binary container(s) to arguments” is selected.



Following this, the build project can be executed by clicking the play button on the menu bar.



The built program can also be executed from the command line. If executing a *sw_emu* build, you must first type the following:

```
export XCL_EMULATION_MODE=sw_emu
```

If executing a *hw_emu* build, you must type the following:

```
export XCL_EMULATION_MODE=hw_emu
```

The build project can then be executed by typing, for example:

```
./EUBinomialTree.exe ./binary_container_1.xclbin
```

18.3.1 Binomial Tree

A file called *option_data.txt* is included for the Binomial Trees algorithm. Here, you can enter data for up to 25 options. The price for all options will then be returned at the end in the form of an array.

```

1 # Option pricing input file
2 # -----
3 # You may enter data for up to 25 options, any extra will be ignored
4 #
5 # Info
6 # S - Stock Price      (float) Underlying stock price
7 # K - Strike Price     (float) Options strike price
8 # T - Time to Maturity (float) Time (in years) until the option expires
9 # D - Dividend Yield   (float) Eg for 5% enter "5" and not "0.05"
10 # r - Risk-free Rate   (float) Eg for 5% enter "5" and not "0.05"
11 # v - Volatility       (float) Eg for 20% enter "20" and not "0.2"
12 # type                (int)  The type of option, enter "0" for call and "1" for put
13 # height              (int)  The height of the tree
14 #
15 # Structure input data
16 # S,K,T,D,r,v,type,height
17 50,50,1,0,0.05,0.20,1,30000

```

NOTE: If executing a Binomial Tree algorithm from the SDAccel environment, the *option_data.txt* file **MUST** be placed in the relevant build folder. For example, if running for *System* build, place the option data file in the “System” folder. If executing via command line the *option_data.txt* file **MUST** be placed in the root of the projects folder.

18.3.2 Monte Carlo

For the Monte Carlo simulation, the option data is set in the “host.cpp” file. The data is set between lines 31 and 39. If these data are changed, the host file will be recompiled once you run the project. Assuming no changes have been made to the HLS code, only the host file will be recompiled, not the entire project.

18.4. References

- [1] Carter, M. (2019) COMP702 FPGA MSc Project. Available online: <https://github.com/mjcarter95/COMP702-FPGA-MSc-Project> [Accessed:].
- [2] Xilinx (2018) PYNQ-Z2 Setup Guide – Python productivity for Zynq (Pynq) v1.0. Available online: https://pynq.readthedocs.io/en/latest/getting_started/pynq_z2_setup.html [Accessed:].
- [3] Xilinx (2019) Getting Started with Alveo Data Center Accelerator Cards. Available online: https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/ug1301-getting-started-guide-alveo-accelerator-cards.pdf [Accessed: 16/09/2019].

19. Appendix 8 – Project Log

Date	Task
11/06/2019	Weekly review meeting
18/06/2019	Weekly review meeting
25/06/2019	Weekly review meeting
02/07/2019	Weekly review meeting
04/07/2019	Submitted project report
09/07/2019	Implemented Black-Scholes formula on Pynq-Z2, transferring floats to and from Zynq-7020
09/07/2019	Weekly review meeting
15/07/2019	Redefined project scope
16/07/2019	Weekly review meeting
18/07/2019	Added Mersenne-Twister and Box-Muller algorithm for generating random numbers
23/07/2019	Weekly review meeting
27/07/2019	Initial implementation of European Binomial Tree algorithm on CPU
30/07/2019	Weekly review meeting
01/08/2019	Initial implementation of American Binomial Tree algorithm on CPU
05/08/2019	Implemented European and American Binomial Tree algorithm on Zynq-7020
06/08/2019	Weekly review meeting
08/08/2019	Implemented Taylor series to approximate the value of exponential function, combined power calculation into for loop on Zynq-7020
11/08/2019	Porting European and American Binomial Tree algorithm to Alveo u200
13/08/2019	Weekly review meeting
15/08/2019	Removing unnecessary reads/writes from Binomial Tree algorithm, combined for loops to reduce work
19/08/2019	Finalised Binomial Tree algorithms on Zynq-7020 and Alveo u200
20/08/2019	Weekly review meeting
21/08/2019	Converted Box-Muller algorithm to not have a for loop, significantly reduced the depth of unrolling
25/08/2019	Implemented Monte Carlo algorithm on Pynq-Z2 and Alveo u200
27/08/2019	Weekly review meeting

29/09/2019	Implemented combined Binomial Tree algorithm on Alveo u200
02/09/2019	Implemented burst buffer write for Monte Carlo algorithm
03/09/2019	Weekly review meeting
09/09/2019	Finished collecting results
10/09/2019	Weekly review meeting
18/09/2019	Weekly review meeting
19/09/2019	Submitted dissertation