Parallel Programming for FPGAs

ECE 8893 - FPG

Spring 2022

Project Report: CRR Option Pricing

By Mourad Musallam

1. Introduction:

There are many different types of financial instruments used within the stock market such as buying and selling individual stocks, option contracts, futures trading, commodities trading etc. Most if not all of these types of trades are done in real time and within a stock exchange. Within this sector there are many different types of computations that trading firms use in order to predict profitability or find possible edge in the market. FPGAs have been used in various ways for running financial computations. One such computation is option contract pricing; option contracts are traded by the millions each and every day. Because of this option price modeling is constantly being implemented in various ways. One method that is common for pricing and predicating possible pricing is the Cox-Cross- Rubinstein binomial tree model for option pricing or simply stated as binomial option pricing model (BOPM) [1].

In this project I will be investigating how to implement this binomial tree model using C/C++ and HLS Vitis. In a general sense I will need to implement a simple version of the of this within C/C++ with limited external library usage since HLS Vitis does not allow many third-party libraries. Then I will synthesize the non-optimized program to make sure it works properly with HLS Vitis. Once the C/C++ version is implemented I tend to optimize the program and experiment with how I could achieve parallelization, low latency, and proper utilization.

2. Problem Description:

Option price solvers are very important when it comes to pricing options in the stock market. Trading firms spend billions of dollars executing trades within the stock market. In doing so it's important to leverage every aspect of the market to leverage profitability and pricing on their trading techniques. Moreover, trading firms spend millions of dollars on computing hardware such as FPGAs, GPUs, ASICs, or third-party computing clusters.

In regards to FPGAs, one has to consider how to design the FPGA for their desired computation(s) and more importantly how to best accelerate their computations. In this case we are exploring with implementing the CRR binomial pricing model on an FPGA. Not only do we want to implement this on the FPGA but how well can we utilize parallel computing and low latency implementation to best accelerate the computation. C/C++ and HLS Vitis make this exploration much more viable when

compared to traditional FPGA design. This type of implementation is important in several aspects. Let's say the designer is trying to implement various tree solvers (not just binomial) on their FPGA. Implementing each one in C/C++ and HLS Vitis allows them to explore what the most suitable FPGA design is for those algorithms as a whole. Once the designer knows which FPGA is most suitable for their application, they can spend more time optimizing and accelerating. Thus, improving development time and optimization.

3. Related Work:

The application for FPGAs in the finance sectors is quite common and popular. Especially when it comes to stock trading within the stock exchange. Trading systems execute trades at extreme speeds, large volumes, and in real time thus making FPGAs the preferred choice for such applications [2]. Not only is speed important but also reconfigurability and programmability. When comparing to ASICs, FPGAs are much more suitable [2].

FPGAs have been used in various solvers for option pricing, some methods used are finite difference solvers for American options [3], Monte Carlo method for European options [3], tree-based solvers for American options [3], and least squares method implemented with Monte Carlo method on American options [3]. Moreover, FPGAs also bring a high level of parallelism in regards to computations. For example, using OpenCL and C/C++ programming to best optimize computations across heterogenous platforms including FPGAs. This type of approach has been done on asset option pricing using Intel FPGAs [4]. Xilinx FPGAs have also been used to leverage high levels of parallelism through HLS Vitis and C/C++ by utilizing dataflow techniques to achieve better performance on credit default swaps (another trading instrument) engine [5].

Overall, this shows that there are several applications for using FPGAs traditionally or for implementing high levels of parallelism through HLS (Xilinx or Intel) and OpenCL.

4. Proposed Approach:

So, at a high level the implementation of the CRR is generally the same step wise. Although one could use various inputs or parameters which can influence the result in different ways. In this case, I will build the CRR model using a series of input parameters and then with those parameters my program will output the 4 options prices. Now there are different types of option contracts and slightly different ways they are executed depending on which country you are in.

Generally speaking, there are two types of option contracts calls and puts. A call option is purchased on the idea that a specific stock or asset will increase in value if the stock (or asset) value goes up, the option price value is increased. A put option is purchased on the idea of the stock or asset is going down in price i.e., if the stock goes down in price the put option increases in value. Again, how these are priced can vary on several parameters in regards to this implementation I will use a specific set of parameters for my option pricing (discussed below). Moreover, computing option prices

can also vary by country or region. American options and European option have slightly different rules on how they are executed. The main difference between the two options (American and European) is that an American option can be exercised earlier then the expiration date vs the European option can only be exercised on the expiration date. This does influence the option price (American option) which will be incorporated in my model. Overall based on my input parameters in table 1 my CRR program will compute 4 option price tree(arrays), American call option (AC), American put option (AP), European call option (EC), and European put option (EC). Additionally, since my output will be 4 arrays, I will only output the option values at the initial time step (n=1) although you could display the option values at different instances in time.

Table 1: Input Parameters

Number of Steps: n	50
Stock Spot Price: Spot	100
Stock Strike Price: K	100
Expiration Date: T	1 year
Dividend Yield: q	0.05
Volatility: v	0.2
Risk Free Rate: r	0.05
Δt	T/n
Up factor: u	$e^{v*\sqrt{\Delta t}}$
Down factor: d	1/u
Probability: p	$\frac{e^{(r-q)(\Delta t)} - d}{u - d}$
Output Results	
American call	AC
American put	AP
European call	EC
European put	EP

So, my program will follow three general steps for computing the option prices based on the input parameters in table 1. Step 1 create a binomial price tree, step 2 compute the option value at each final node i.e., the expiration of the option contract, and step 3 compute the option value at each previous node starting from the second to last node. Below is a more detailed step by step process.

Step 1: Create a binomial price tree based off the stock price at n = 0.

- Let S_0 be the price of the stock at initial time.
- Let u and d be the specific factor that the next time period the price will be up or down. Such that $u \geq 0 \; and \; 0 < d \leq 1$. Giving $S_{up} = S \cdot u \; or \; S_{down} = S \cdot d$.

Step 2: Compute option value at each final node (expiration date).

- $Max[(S_n - K), 0]\ for\ call\ option.$
- $Max[(K - S_n), 0]\ for\ put\ option\ option.$
- K is the strike price and S_n is the spot price of the option at the n^{th}period.

Step 3: Compute the option value at each previous node starting from second to last node.

- Binomial Value$= [p \times option\ up + (1 - p) \times option\ down] \times e^{-r\Delta t}$

Overall, these three steps were my starting point and essentially how I built my C/C++ model. For a general overview of what the price tree should look like see figure 1 and figure 2. Figure 2 gives the first couple of steps of my tree using although the input parameters are slightly different.

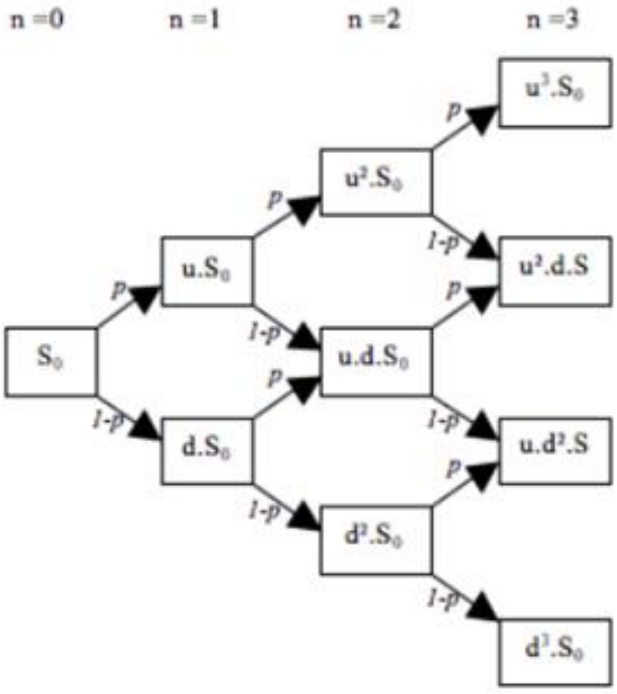


Figure 1 Price Tree Steps

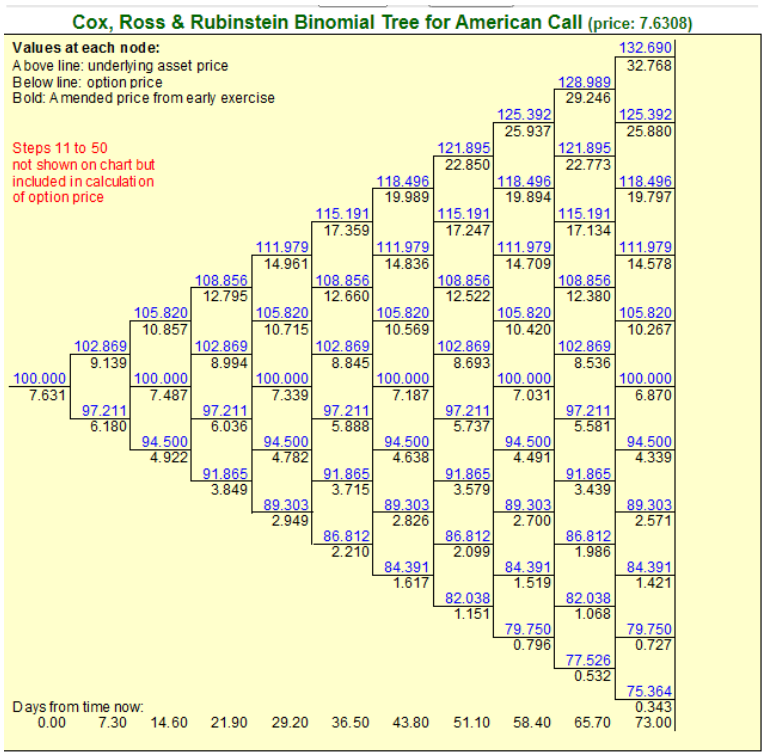


Figure 2 Example of Price Tree American Option

5. C++ Implementation:

In order to implement my CRR model I used a total of 3 files main.cpp, top.cpp, and utils.h. This is the general setup when one is going to run their program using HLS Vitis. First, I decided to pre-define my input parameters in the utils.h header file (figure 3). This will be used as part of the computation in my top.cpp.

```
1  #ifndef __UTILS_H__
2  #define __UTILS_H__
3
4  //#include <vector>
5  #include <stdio.h>
6  #include <math.h>
7  #include <iostream>
8  #include <string>
9
10
11
12 #define n 50  //number of steps
13
14 #define Spot 100              // Spot Price
15
16 #define K 100                 // Strike Price
17
18 #define T 1                   // Years to maturity
19
20 #define q 0.05                //dividend yield
21
22 #define r 0.05                    // Risk Free Rate
23
24 #define v 0.20                    // Volatility
25
26 #define dt (T) /(n)  //DetlaT 0.01
27
28 #define u 1.0286880693018583 //Up factor
29
30 #define d 1/u  //down factor
31
32 #define p 0.49292940355494647  //probability
33
34 #endif
```

Figure 3 utils.h

Next, I have my main.cpp file which is where I initialize 5 2D arrays of size n+1 with float data type. The 5 arrays are, the price tree array(S), European call option array (EC), European put option array (EP), American call option array (AC), and American put option array (AP). These 5 arrays will be passed into my top function which where the CRR model computation is implemented. Once complete the 4 option values will be printed on screen at the initialize time step figure 4 shows my main.cpp setup.

```cpp
#include "utils.h"
using namespace std;

void top(float AC_DR[n+1][n+1],float AP_DR[n+1][n+1],float EC_DR[n+1][n+1],float EP_DR[
float S_DR[n+1][n+1]);


int main()
{

    float S[n+1][n+1];
    float EC[n+1][n+1];
    float EP[n+1][n+1];
    float AC[n+1][n+1];
    float AP[n+1][n+1];

    top(AC,AP,EC,EP,S);

// Output of prices of calls and puts

    cout << "The Cox Ross Rubinstein prices using " << n << " steps are... " << endl;

    cout << "European Call " << EC[0][0] << endl;

    cout << "European Put  " << EP[0][0] << endl;

    cout << "American Call " << AC[0][0] << endl;

    cout << "American Put  " << AP[0][0] << endl;

    cout << endl;
    return 0;

}
```

Figure 4 main.cpp

Lastly is the top.cpp file, which contains all my computations for my CRR model. The top.cpp file is made up of various functions such as my initial top function, moveTobram, buildtree, terminalPayoff, backwardRecursion, then lastly, I have a nested for-loop to move back to DRAM. Note that in my project I used a the Pynq-Z2 board (xc7z020clg400-1). In regards to that, I did use 5 different ports (bundles) one for each array and ran my tests doing this. Although I realized rather late that the actual Pynq-Z2 board only has 4 so for my HLS Vitis implementation I kept it at 5 (see #pragma interface bundle in figure 5). Since changing this late into my project would require drastic changes to my optimizations.

```cpp
void top(float AC_DR[n+1][n+1],float AP_DR[n+1][n+1],float EC_DR[n+1][n+1],float EP_DR[n+1][n+1],
float S_DR[n+1][n+1])
{
    #pragma HLS interface m_axi depth=51*51 port=AC_DR offset=slave bundle=memAC
    #pragma HLS interface m_axi depth=51*51 port=AP_DR offset=slave bundle=memAP
    #pragma HLS interface m_axi depth=51*51 port=EC_DR offset=slave bundle=memEC
    #pragma HLS interface m_axi depth=51*51 port=EP_DR offset=slave bundle=memEP
    #pragma HLS interface m_axi depth=51*51 port=S_DR offset=slave bundle=memS
    #pragma HLS interface s_axilite port=return

    float AC_B[n+1][n+1]; float AP_B[n+1][n+1];float EC_B[n+1][n+1];float EP_B[n+1][n+1];float S_B[n+1][n+1];

    moveTobram(AC_B,AC_DR);
    moveTobram(AP_B,AP_DR);
    moveTobram(EC_B,EC_DR);
    moveTobram(EP_B,EP_DR);
    moveTobram(S_B,S_DR);

    buildtree(S_B);
    terminalPayoff(AC_B,AP_B,EC_B,EP_B,S_B);
    backwardRecursion(AC_B,AP_B,EC_B,EP_B,S_B);

    for (int i = 0; i<(n+1);i++)
    {

        for (int j = 0; j <(n+1);j++)
        {
            AC_DR[i][j] = AC_B[i][j];
            AP_DR[i][j] = AP_B[i][j];
            EC_DR[i][j] = EC_B[i][j];
            EP_DR[i][j] = EP_B[i][j];
            S_DR[i][j]  = S_B[i][j];
        }
    }
}

```

Figure 5 top function

Now I give an overview of the structure of my computations in my top.cpp file. My top.cpp file has a movetobram function which simply moves all the initial values from DRAM to BRAM of the 5 2D arrays. Then the first computation that is executed is the buildtree function. This function simply builds my binomial price tree array using array S along with the initial up and down factor see figure 6 for code structure.

Once my price tree is built the terminalPayoff function is called. The function computes the option value at each final node as shown in figure 7. Here my 4 option arrays are passed in along with my price tree array (S array). Additionally, I have built a max function (figure 8) which is used within my terminalPayoff function. After this computation my 4 option arrays (AC, AP, EC, EP) have been partially completed.

Once the terminalPayoff function is complete. Now the backwardRecursion function is called by passing the 5 arrays. This function computes the option values for each previous nodes in the tree starting from the 2nd to last node. Figure 9 shows how the function is implemented for each option price tree. Note that since American options can be exercised early, we have to use the max function within each iteration as this influences the option value. Once these computations are complete, I use a nested for-loop to move back to DRAM and print out my option values at the initial steps n = 0 although you could change the indexes and you will get option values for other n steps.

As a result of this my initial latency and utilization is shown in figure 10 and figure 11. Additionally, the code provides the correct result when I run my C simulation (figure 12) as I used a simple python program to make sure my outputs are correct.

```
void buildtree(float mat[n+1][n+1])
{

    for (int j = 0; j<= n; j++)
    {
        for (int i = 0; i <= j; i++)
        {
            mat[i][j] = Spot*powf(u, j - i)*powf(d, i);
        }
    }
}
```

Figure 6 buildtree function in top.cpp

```
void terminalPayoff(float matAC[n+1][n+1],float matAP[n+1][n+1],float matEC[n+1][n+1],
float matEP[n+1][n+1], float matS[n+1][n+1])
{

    for (int i = 0; i <= n; i++)
    {
        matEC[i][n] = max(matS[i][n] - K, 0.0);
        matAC[i][n] = max(matS[i][n] - K, 0.0);
        matEP[i][n] = max(K - matS[i][n], 0.0);
        matAP[i][n] = max(K - matS[i][n], 0.0);
    }
}
```

Figure 7 terminalPayoff function

```
float max(float a, float b)
{
    return (a>b) ? a : b;
}
```

Figure 8 max function

```
void backwardRecursion(float matAC[n+1][n+1],float matAP[n+1][n+1],float matEC[n+1][n+1],float matEP[n+1][n+1], float matS[n+1][n+1])
{
    for (int j = n - 1; j >= 0; j--)
    {


     for (int i = 0; i <= j; i++)
        {
            matEC[i][j] =expf(-r*dt)*(p*matEC[i][j + 1] + (1 - p)*matEC[i + 1][j + 1]);

            matEP[i][j] =expf(-r*dt)*(p*matEP[i][j + 1] + (1 - p)*matEP[i + 1][j + 1]);

            matAC[i][j] = max(matS[i][j] - K,expf(-r*dt)*(p*matAC[i][j + 1] + (1 - p)*matAC[i + 1][j + 1]));

            matAP[i][j] = max(K - matS[i][j],expf(-r*dt)*(p*matAP[i][j + 1] + (1 - p)*matAP[i + 1][j + 1]));

        }

    }
}
```

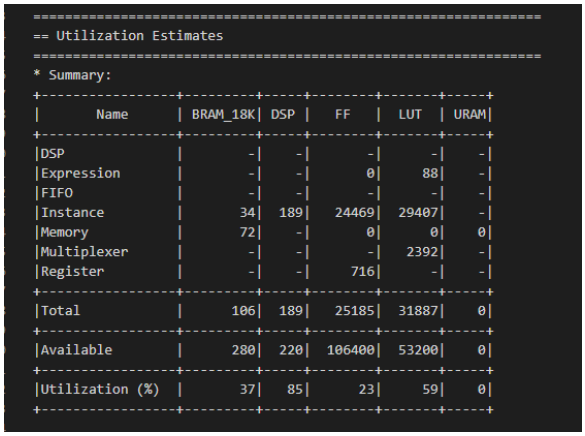
Figure 9 backwardRecursion function

```
+ Latency:
    * Summary:
    +---------+---------+-----------+-----------+------+-------+---------+
    |  Latency (cycles) |  Latency (absolute)   |   Interval    | Pipeline|
    |   min   |   max   |    min    |    max    | min  |  max  |   Type  |
    +---------+---------+-----------+-----------+------+-------+---------+
    |     8246|    14897|  82.460 us|  0.149 ms|  8247|  14898|       no|
    +---------+---------+-----------+-----------+------+-------+---------+
```

Figure 10 Initial Latency

```
The Cox Ross Rubinstein prices using 50 steps are...
European Call 7.53931
European Put  7.53927
American Call 7.63081
American Put  7.63077
```

Figure 11 Output from C simulation



```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-----+--------+-------+-----+
|       Name      | BRAM_18K| DSP |   FF   |  LUT  | URAM|
+-----------------+---------+-----+--------+-------+-----+
|DSP              |        -|    -|       -|      -|    -|
|Expression       |        -|    -|       0|     88|    -|
|FIFO             |        -|    -|       -|      -|    -|
|Instance         |       34|  189|   24469|  29407|    -|
|Memory           |       72|    -|       0|      0|    0|
|Multiplexer      |        -|    -|       -|   2392|    -|
|Register         |        -|    -|     716|      -|    -|
+-----------------+---------+-----+--------+-------+-----+
|Total            |      106|  189|   25185|  31887|    0|
+-----------------+---------+-----+--------+-------+-----+
|Available        |      280|  220|  106400|  53200|    0|
+-----------------+---------+-----+--------+-------+-----+
|Utilization (%)  |       37|   85|      23|     59|    0|
+-----------------+---------+-----+--------+-------+-----+
```

Figure 12 Initial Utilization

6. Optimization:

Build tree function: The first optimization I decided to go with was to try and unroll and pipeline my buildtree function since that is the first computation that is executed although while synthesizing my program it would result in a hard error and not complete at all. I realized that the way I was building my tree structure array was causing the synthesizing to fail. Note that when building a tree in an array the first-dimension index is the second loop and is conditioned on the upper loop. So here loop indexing and loop re ordering had to be done see below code structure. After re ordering the loops/index my utilization stayed relatively the same but my latency improved went from 0.149 ms to 0.122 ms.

```
//loop reordering and indexing for buildtree function

for (int i = 0; i <= n; i++)

    {
        for (int j = 0; j <= n; j++)
        {
            mat[j][i] = Spot*powf(u, i - j)*powf(d, j);
        }
    }
+ Latency:
    * Summary:
    +---------+---------+----------+----------+------+-------+---------+
    |  Latency (cycles) |  Latency (absolute)  |   Interval   | Pipeline|
    |   min   |   max   |    min   |    max   |  min |  max  |   Type  |
    +---------+---------+----------+----------+------+-------+---------+
    |     8146|    12196|  81.460 us|  0.122 ms|  8147|  12197|       no|
    +---------+---------+----------+----------+------+-------+---------+
```

Now that I was able to get a slight improvement, I tried to implement pipeline in the top loop and unroll in the second loop in buildtree function. Unfortunately, this led to over utilization of DSPs and LUTs as seen below.

```
for (int i = 0; i <= n; i++)
    {
        #pragma HLS pipeline
        for (int j = 0; j <= n; j++)
        {
            #pragma HLS unroll
            mat[j][i] = Spot*powf(u, i - j)*powf(d, j);
        }
    }
}
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-----+--------+--------+-----+
|       Name      | BRAM_18K| DSP |   FF   |   LUT  | URAM|
+-----------------+---------+-----+--------+--------+-----+
|DSP              |        -|    -|       -|       -|    -|
|Expression       |        -|    -|       0|      50|    -|
|FIFO             |        -|    -|       -|       -|    -|
|Instance         |       71|  763|  114393|  194847|    -|
|Memory           |      115|    -|       0|       0|    0|
|Multiplexer      |        -|    -|       -|    6576|    -|
|Register         |        -|    -|    2328|       -|    -|
+-----------------+---------+-----+--------+--------+-----+
|Total            |      186|  763|  116721|  201473|    0|
+-----------------+---------+-----+--------+--------+-----+
|Available        |      280|  220|  106400|   53200|    0|
+-----------------+---------+-----+--------+--------+-----+
|Utilization (%)  |       66|  346|     109|     378|    0|
+-----------------+---------+-----+--------+--------+-----+
```

My last attempt for improving this is trying to partition all 5 arrays within my top function. More specifically I used a cyclic partition with dimension 1 and with a factor of 1 as seen below. Doing so improved my latency 0.122 ms to 0.109ms and I had slight reduction in overall utilization.

```
    #pragma HLS ARRAY_PARTITION variable=S_B dim = 1 factor = 2 cyclic
    #pragma HLS ARRAY_PARTITION variable=AC_B dim = 1 factor = 2 cyclic
    #pragma HLS ARRAY_PARTITION variable=AP_B dim = 1 factor = 2 cyclic
    #pragma HLS ARRAY_PARTITION variable=EC_B dim = 1 factor = 2 cyclic
    #pragma HLS ARRAY_PARTITION variable=EP_B dim = 1 factor = 2 cyclic
    for (int i = 0; i <= n; i++)
    {
        #pragma HLS pipeline
        for (int j = 0; j <= n; j++)
        {
            #pragma HLS unroll
            mat[j][i] = Spot*powf(u, i - j)*powf(d, j);
        }
    }
```

```
+ Latency:
    * Summary:
    +---------+---------+----------+----------+------+-------+---------+
    |  Latency (cycles) |  Latency (absolute)  |   Interval   | Pipeline|
    |   min   |   max   |    min   |    max   | min |  max  |   Type  |
    +---------+---------+----------+----------+------+-------+---------+
    |     6852|    10902|  68.520 us|  0.109 ms|  6853|  10903|       no|
    +---------+---------+----------+----------+------+-------+---------+
  ================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-----+--------+-------+-----+
|       Name      | BRAM_18K| DSP |   FF   |  LUT  | URAM|
+-----------------+---------+-----+--------+-------+-----+
|DSP              |        -|    -|       -|      -|    -|
|Expression       |        -|    -|       0|     50|    -|
|FIFO             |        -|    -|       -|      -|    -|
|Instance         |       22|  179|   18718|  27260|    -|
|Memory           |       72|    -|       0|      0|    0|
|Multiplexer      |        -|    -|       -|   2392|    -|
|Register         |        -|    -|     693|      -|    -|
+-----------------+---------+-----+--------+-------+-----+
|Total            |       94|  179|   19411|  29702|    0|
+-----------------+---------+-----+--------+-------+-----+
|Available        |      280|  220|  106400|  53200|    0|
+-----------------+---------+-----+--------+-------+-----+
```

```
|Utilization (%)  |       33|   81|      18|     55|    0|
+-----------------+---------+-----+--------+-------+-----+
```

Terminal Payoff function: Now I started to focus on my terminal payoff function. This function was acting rather weird as I tried to fully unroll it but it led to no change in latency (increased latency by 0.001). Additionally, I also used pragma unroll factor doing so only led to increase in latency as seen below when I used factor = 10

```
void terminalPayoff(float matAC[n+1][n+1],float matAP[n+1][n+1],float
matEC[n+1][n+1],
float matEP[n+1][n+1], float matS[n+1][n+1])
{

    for (int i = 0; i <= n; i++)
    {
        #pragma HLS unroll factor = 10
        matEC[i][n] = max(matS[i][n] - K, 0.0);
        matAC[i][n] = max(matS[i][n] - K, 0.0);
        matEP[i][n] = max(K - matS[i][n], 0.0);
        matAP[i][n] = max(K - matS[i][n], 0.0);
    }
}
+ Latency:
    * Summary:
    +---------+---------+-----------+----------+------+-------+---------+
    |  Latency (cycles) |  Latency (absolute)  |   Interval   | Pipeline|
    |   min   |   max   |    min    |    max   |  min |  max  |   Type  |
    +---------+---------+-----------+----------+------+-------+---------+
    |     6830|    13380|  68.300 us|  0.134 ms|  6831|  13381|       no|
    +---------+---------+-----------+----------+------+-------+---------+
```

From here I decided to go with only using HLS pipeline within the loop. As a result, the latency had not improved from our initial improvement of 0.109 ms from the buildtree function. Also, the utilization was unchanged as well. Below shows the code structure and output utilization and latency.

```
    for (int i = 0; i <= n; i++)
    {
        #pragma HLS PIPELINE
        matEC[i][n] = max(matS[i][n] - K, 0.0);
        matAC[i][n] = max(matS[i][n] - K, 0.0);
        matEP[i][n] = max(K - matS[i][n], 0.0);
        matAP[i][n] = max(K - matS[i][n], 0.0);
    }
    * Summary:
```

```
    +---------+---------+----------+----------+------+-------+---------+
    |  Latency (cycles) |  Latency (absolute)  |   Interval   | Pipeline|
    |   min   |   max   |    min   |    max   |  min |  max  |   Type  |
    +---------+---------+----------+----------+------+-------+---------+
    |     6852|    10902|  68.520 us|  0.109 ms|  6853|  10903|       no|
    +---------+---------+----------+----------+------+-------+---------+
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-----+--------+-------+-----+
|       Name      | BRAM_18K| DSP |   FF   |  LUT  | URAM|
+-----------------+---------+-----+--------+-------+-----+
|DSP              |        -|    -|       -|      -|    -|
|Expression       |        -|    -|       0|     50|    -|
|FIFO             |        -|    -|       -|      -|    -|
|Instance         |       22|  179|   18718|  27260|    -|
|Memory           |       72|    -|       0|      0|    0|
|Multiplexer      |        -|    -|       -|   2392|    -|
|Register         |        -|    -|     693|      -|    -|
+-----------------+---------+-----+--------+-------+-----+
|Total            |       94|  179|   19411|  29702|    0|
+-----------------+---------+-----+--------+-------+-----+
|Available        |      280|  220|  106400|  53200|    0|
+-----------------+---------+-----+--------+-------+-----+
|Utilization (%)  |       33|   81|      18|     55|    0|
+-----------------+---------+-----+--------+-------+-----+
```

One more improvement I tried for my terminal payoff function was to add HLS data flow just above the loop then add HLS pipeline. Unfortunately, this had no change in latency as it stayed the same at 0.109ms and a slight increase in utilization as seen below. I did attempt a few other optimizations for this function but I either got no change or drastic increase in latency. Overall my terminal pay off function was left unchanged.

```
    #pragma HLS DATAFLOW
    for (int i = 0; i <= n; i++)
    {
        #pragma HLS PIPELINE
        matEC[i][n] = max(matS[i][n] - K, 0.0);
        matAC[i][n] = max(matS[i][n] - K, 0.0);
        matEP[i][n] = max(K - matS[i][n], 0.0);
        matAP[i][n] = max(K - matS[i][n], 0.0);
    }
== Utilization Estimates
================================================================
* Summary:
```

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	54	-
FIFO	-	-	-	-	-
Instance	22	183	19160	28074	-
Memory	72	-	0	0	0
Multiplexer	-	-	-	2196	-
Register	-	-	695	-	-
Total	94	183	19855	30324	0
Available	280	220	106400	53200	0
Utilization (%)	33	83	18	57	0

Backward Recursion function: The last function I have is the backward recursion function. This function computes the option values for each node in the tree working backwards and starting at the second to last node. More specifically the 4 option arrays I am trying to compute. Here I noticed that my inner loop index was conditioned on the upper loop index. I decided to change the index to a constant value. Doing this improved my latency from 0.109 ms to 0.09186 ms (91.86 us) and my overall utilization decreased slightly. Nonetheless this was a good improvement.

```
void backwardRecursion(float matAC[n+1][n+1],float matAP[n+1][n+1],
float matEC[n+1][n+1],
float matEP[n+1][n+1], float matS[n+1][n+1])
{
    for (int j = n - 1; j >= 0; j--)
    {

      for (int i = 0; i <= n-1; i++)
        {
            matEC[i][j] =expf(-r*dt)*(p*matEC[i][j + 1] + (1 - p)*matEC[i + 1][j
+ 1]);

            matEP[i][j] =expf(-r*dt)*(p*matEP[i][j + 1] + (1 - p)*matEP[i + 1][j
+ 1]);

            matAC[i][j] = max(matS[i][j] - K,expf(-r*dt)*(p*matAC[i][j + 1] + (1
- p)*matAC[i + 1][j + 1]));

            matAP[i][j] = max(K - matS[i][j],expf(-r*dt)*(p*matAP[i][j + 1] + (1
- p)*matAP[i + 1][j + 1]));

        }

    }
}
```

```
+ Latency:
    * Summary:
    +---------+---------+----------+----------+------+------+---------+
    |  Latency (cycles) |   Latency (absolute)  |   Interval  | Pipeline|
    |   min   |   max   |    min   |    max   | min  |  max |   Type  |
    +---------+---------+----------+----------+------+------+---------+
    |     9186|     9186|  91.860 us|  91.860 us|  9187|  9187|       no|
    +---------+---------+----------+----------+------+------+---------+
================================================================
== Utilization Estimates
================================================================
* Summary:
```

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	8	-
FIFO	-	-	-	-	-
Instance	22	179	18750	27364	-
Memory	72	-	0	0	0
Multiplexer	-	-	-	2374	-
Register	-	-	662	-	-
Total	94	179	19412	29746	0
Available	280	220	106400	53200	0
Utilization (%)	33	81	18	55	0

After that I went ahead and tried to unroll the second loop within my backward recursion function. Although I was somewhat skeptical as it took a little longer than normal for it to finish synthesizing. As a result, the latency got slightly worse back to 0.109 ms and my DSPs, FFs, and LUTs were all over utilized as shown below.

```
void backwardRecursion(float matAC[n+1][n+1],float matAP[n+1][n+1],
float matEC[n+1][n+1],
float matEP[n+1][n+1], float matS[n+1][n+1])
{
    for (int j = n - 1; j >= 0; j--)
    {

       #pragma HLS unroll
      for (int i = 0; i <= n-1; i++)
        {
            matEC[i][j] =expf(-r*dt)*(p*matEC[i][j + 1] + (1 - p)*matEC[i + 1][j + 1]);

            matEP[i][j] =expf(-r*dt)*(p*matEP[i][j + 1] + (1 - p)*matEP[i + 1][j + 1]);

            matAC[i][j] = max(matS[i][j] - K,expf(-r*dt)*(p*matAC[i][j + 1] + (1 - p)*matAC[i + 1][j + 1]));

            matAP[i][j] = max(K - matS[i][j],expf(-r*dt)*(p*matAP[i][j + 1] + (1 - p)*matAP[i + 1][j + 1]));

        }

    }
}
```

Latency:
* Summary:

Latency (cycles)		Latency (absolute)		Interval		Pipeline
min	max	min	max	min	max	Type
10902	10902	0.109 ms	0.109 ms	10903	10903	no

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-----+--------+-------+-----+
|       Name      | BRAM_18K| DSP |   FF   |  LUT  | URAM|
+-----------------+---------+-----+--------+-------+-----+
|DSP              |        -|    -|       -|      -|    -|
|Expression       |        -|    -|       0|      8|    -|
|FIFO             |        -|    -|       -|      -|    -|
|Instance         |       22|  232|  145316|  69291|    -|
|Memory           |       72|    -|       0|      0|    0|
|Multiplexer      |        -|    -|       -|   2178|    -|
|Register         |        -|    -|     662|      -|    -|
+-----------------+---------+-----+--------+-------+-----+
|Total            |       94|  232|  145978|  71477|    0|
+-----------------+---------+-----+--------+-------+-----+
|Available        |      280|  220|  106400|  53200|    0|
+-----------------+---------+-----+--------+-------+-----+
|Utilization (%)  |       33|  105|     137|    134|    0|
+-----------------+---------+-----+--------+-------+-----+
```

Although I didn't think HLS unroll would help at all I did decide to use HLS unroll factor=10 on the inner loop just to see if it made a difference. Doing so took a very long time for the synthesizing to complete so I decided to terminate the program. Moreover, instead of HLS unroll I tried to go with just HLS pipeline instead. I applied HLS pipeline to the inner loop. Which again resulted in a very long synthesizing time in Vitis. So, after about 5 min I decided to terminate the program. Another optimization I tried is creating a subfunction for my inner for loop. Here my inner for-loop is in a function within my backward recursion functions. I also added HLS dataflow just before my subfunction call although when I did this, it showed that my inner loop was pipelined but my latency and utilization did not change at all. So reverted back to the nested loop below is an example using the subfunction. Overall, this is where I stopped for optimizing my backward recursion function.

```
void subfunction(float matAC[n+1][n+1],float matAP[n+1][n+1],
float matEC[n+1][n+1],float matEP[n+1][n+1], float matS[n+1][n+1],int k)
{
    for(int i = 0;i<=n-1;i++)
    {

        matEC[i][k] =expf(-r*dt)*(p*matEC[i][k + 1] + (1 - p)*matEC[i + 1][k + 1]);
        matEP[i][k] =expf(-r*dt)*(p*matEP[i][k + 1] + (1 - p)*matEP[i + 1][k + 1]);
        matAC[i][k] = max(matS[i][k] - K,expf(-r*dt)*(p*matAC[i][k + 1] + (1 - p)*matAC[i + 1][k + 1]));
        matAP[i][k] = max(K - matS[i][k],expf(-r*dt)*(p*matAP[i][k + 1] + (1 - p)*matAP[i + 1][k + 1]));
    }
}


void backwardRecursion(float matAC[n+1][n+1],float matAP[n+1][n+1],float matEC[n+1][n+1],
float matEP[n+1][n+1], float matS[n+1][n+1])
{
    for (int j = n - 1; j >= 0; j--)
    {
        #pragma HLS dataflow
        subfunction(matAC,matAP,matEC,matEP,matS,j);
    }
}
```

7. Results/Future works:

After attempting various optimizations within my top.cpp file my overall latency was reduced down to 0.09186 ms (91.860 us) from 0.149ms. Which resulted in a ~1.62 speed up. Although I implemented several optimizations within my top function such as HLS pipeline, HLS unroll, loop-reordering, and array partition. Additionally, my utilization was not to bad with DSP utilization being the highest after optimizations at 81%. BRAM, FF, and LUTs were relatively unaffected. But my speed generally speaking was not that great either. This leads me to believe that the structure of my program could probably be coded in a different manner. Maybe if I used a struct to build my binomial tree instead of a traditional 2D array. HLS Vitis does have some different pragmas one could use on structs.

I think if implemented an entirely different code structure when building a binomial tree, I will be able to further optimize this CRR model. Furthermore, I also used float data type, I'm certain that if used FIXED-POINT data type my optimization would be much easier to improve. Another thing to point out is that I did this implementation on a Pynq-Z2 board which is not a very powerful board but it was worth the challenge to see how far one could get. Lastly, I did implement this model in Vivado and generated bitstream but I had to change my bundle(ports) for my interface because there is only 4 available on the actual board therefore the results are not the same but it is doable.

As time permits, I may try and redo the code structure and test out the differences between my original implementation and compare to the new implementation.

$$Speed - up = \frac{14897\ cycles}{9186\ cycles} = 1.62$$

+ Latency:
    * Summary:

Latency (cycles)		Latency (absolute)		Interval		Pipeline
min	max	min	max	min	max	Type
9186	9186	91.860 us	91.860 us	9187	9187	no

Figure 13 Final latency

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-----+--------+-------+-----+
|       Name      | BRAM_18K| DSP |   FF   |  LUT  | URAM|
+-----------------+---------+-----+--------+-------+-----+
|DSP              |        -|    -|       -|      -|    -|
|Expression       |        -|    -|       0|      8|    -|
|FIFO             |        -|    -|       -|      -|    -|
|Instance         |       22|  179|   18750|  27364|    -|
|Memory           |       72|    -|       0|      0|    0|
|Multiplexer      |        -|    -|       -|   2374|    -|
|Register         |        -|    -|     662|      -|    -|
+-----------------+---------+-----+--------+-------+-----+
|Total            |       94|  179|   19412|  29746|    0|
+-----------------+---------+-----+--------+-------+-----+
|Available        |      280|  220|  106400|  53200|    0|
+-----------------+---------+-----+--------+-------+-----+
|Utilization (%)  |       33|   81|      18|     55|    0|
+-----------------+---------+-----+--------+-------+-----+
```

Figure 14 Final Utilization

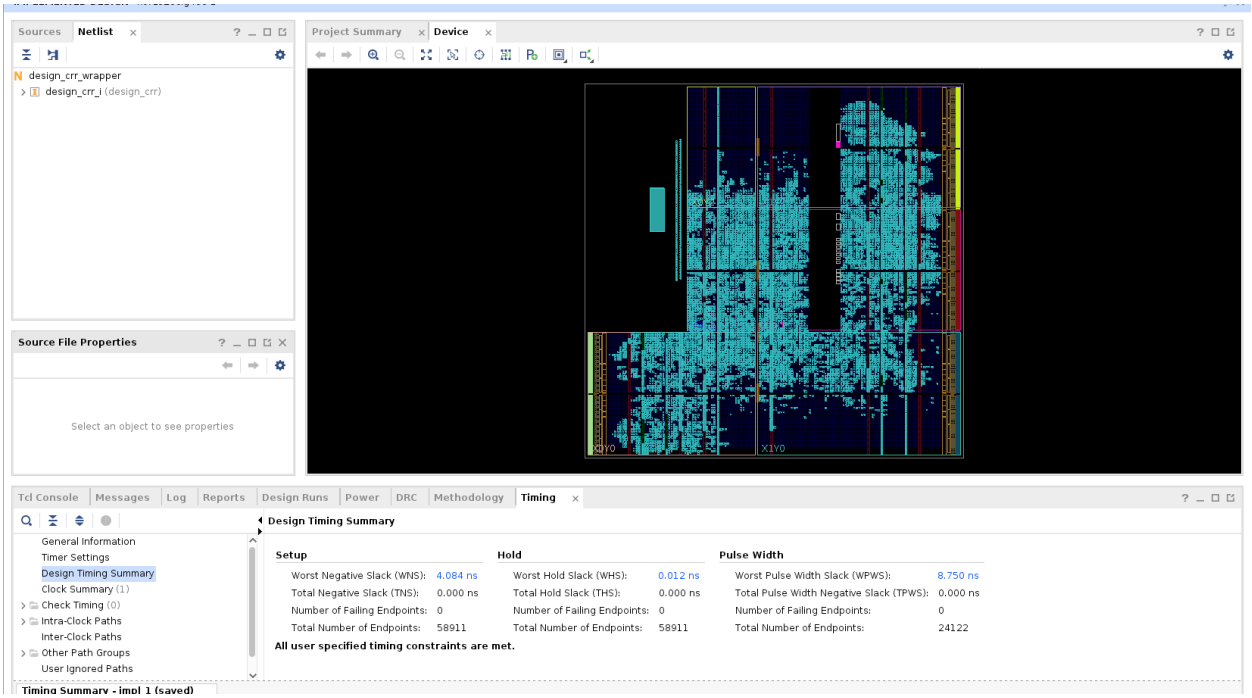


Figure 15 Vivado

REFERENCES

[1] [1]Wikipedia Contributors, “Binomial options pricing model,” Wikipedia, Jun. 06, 2019. https://en.wikipedia.org/wiki/Binomial_options_pricing_model.

[2] [2]“FPGAs in the Financial Services Industry,” EnterpriseAI, Nov. 06, 2017. https://www.enterpriseai.news/2017/11/06/fpgas-financial-services-industry/ (accessed Mar. 21, 2022).

[3] [3]Q. Jin, W. Luk, and D. B. Thomas, “On Comparing Financial Option Price Solvers on FPGA,” IEEE Xplore, May 01, 2011. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5771255.

[4] [4]A. O. Mahony, G. Zeidan, B. Hanzon, and E. Popovici, “A Parallel and Pipelined Implementation of a Pascal-Simplex Based Two Asset Option Pricer on FPGA using OpenCL,” IEEE Xplore, Oct. 01, 2020. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9264992.

[5] [5]N. Brown, M. Klaisoongnoen, and O. T. Brown, “Optimisation of an FPGA Credit Default Swap engine by embracing dataflow techniques,” IEEE Xplore, Sep. 01, 2021. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9556017 (accessed Mar. 21, 2022).