Parallel Computing for FPGA

Denzel Chia Wen Xuan 4S2 (Leader)
David Lim Kang Wei 4S2

Table of contents

Introduction

1.1 Background Information

FPGA, short for Field Programmable Gate Array, is an integrated circuit that can be programmed in the field after manufacture[1]. While both microcontrollers and FPGA might seem similar in the fact that both run on code, FPGAs, unlike microcontrollers, are not pre-designed to perform certain tasks, and one can buy an FPGA and configure it to the design one needs, making it much more convenient in achieving certain processes as compared to microcontrollers.

The FPGA however suffers from high cost as a disadvantage. Nevertheless, for coders who wish to programme various different types of tasks without buying many different microcontrollers, FPGAs will turn out to be more cost efficient.

One of the most notable advantages FPGAs have over their counterpart is that they run on parallel processing unlike microcontrollers who processes code in series. Hence, FPGAs are able to achieve certain functions which would otherwise take much longer on a microcontroller, such as massive image or digital signal processing applications[2].

In FPGAs, one is coding the hardware, such as the actual wiring and logic gates, as compared to coding the software in microcontrollers. Thus, FPGAs run on hardware descriptive language, such as Verilog and VHDL, which are the two most common languages used in the field.

All in all, FPGAs are much more efficient in carrying out processes, especially large processes as compared to microcontrollers as they run on parallel processing. This makes them a better choice than microcontrollers if one is trying to achieve high speed processing in the nanoseconds range. Our project will compare the processing speeds of FPGAs and microcontrollers, and find out whether FPGAs really are faster than microcontrollers.

1.2 Objective

Our aim then is to compare the processing speeds of FPGAs against microcontrollers by exploring the parallel computing of the FPGA in floating point, which would be used to code the Fullconnect algorithm of the whole MCCN (Multi-tasked Cascaded Convolutional Networks) algorithm. It can identify faces of people in pictures or in live time, in other words a face recognition algorithm, and this will be done during the 6 week attachment with our DSO external mentors.

The speed comparison will be done by comparing the time taken for both the VHDL code and C++ code to carry out a process similar to that of Fullconnect - Convolution of a 2D array into a 1 D array by multiplying all values in a single row, followed by the summation of all the multiplied values to reach a single value, thus condensing a 2D array to a 1D array. We can thus get a grasp as to whether FPGAs or microcontrollers can run processes faster and more efficiently. Additionally, it should be noted that this process here will be used as a foundation for the actual Fullconnect algorithm which will be coded near the end of the year, especially during the 6 weeks attachment at DSO.

Literature Review

Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks

K., Zhang, Z., Zhang, Z., Li, & Q., Yu. (n.d.). *Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks* (Rep.). Retrieved March 24, 2019, from https://arxiv.org/ftp/arxiv/papers/1604/1604.02878.pdf.

This study looks into the current face detection and alignment tools such as state-of-the-art techniques on the challenging FDDB and WIDER FACE benchmark for face detection and AFLW benchmark for face alignment, and proposes a deep cascaded multi-task framework which exploits the inherent correlation between them to boost up their performance. In particular, the framework adopts a cascaded structure with three stages of carefully designed deep convolutional networks that predict face and landmark location in a coarse-to-fine manner. In addition, in the learning process, the authors have proposed a new online hard sample mining strategy that can improve the performance automatically without manual sample selection. Thus, they aim to achieve superior accuracy over the current modules while keeping real time performance.

The authors tackled this problem by exploiting a fully convolutional network. Given an image, they would initially resize it to different scales to build an image pyramid, which is the input of the three-stage cascaded network. The first stage is the Proposal network, which aims to obtain the candidate windows and their bounding box regression vectors to calibrate the candidates. Afterwards, the authors employed non-maximum suppression(NMS) to merge highly overlapped candidates. The second network is the Refine network, which further rejects a large number of false candidates, performs calibration with bounding box regression, and NMS candidate merge. The final stage, Output network, is similar to the second stage, but instead aims to describe the face in more details. In particular, the network would output five facial landmarks' position. In addition to multi-source training and online hard sample mining, experiments have shown that this strategy has yielded better performance.

In the article, the authors concluded that the proposed multi-task cascaded CNNs based framework for joint face detection and alignment have demonstrated the ability to consistently outperform the state-of-the-art methods across several challenging benchmarks, including FDDB and WIDER FACE benchmarks for face detection, and AFLW benchmark for face alignment, while keeping real time performance.

To ensure their conclusion is valid, the authors have carried out experiments on their proposed framework to ensure the effectiveness of their product. This is done by comparing their face detector and alignment against the state-of-the-art methods in Face Detection Data Set and Benchmark (FDDB), WIDER FACE, and Annotated Facial Landmarks in the Wild (AFLW) benchmark. Finally, they also evaluated the computational efficiency of their face detector. Mainly, the results they obtained from comparing their face detection technology were positive, where results have shows that their method consistently outperforms all the previous approaches by a large margin in both the benchmarks when comparing their method against the state-of-the-art methods [1, 5, 6, 11, 18, 19, 26, 27, 28, 29] in FDDB, and the state-of-the-art methods [1, 24, 11] in WIDER FACE. They also evaluate their approach on some challenge photos and has gained positive results. The cascade structure has also ensure that their method can achieve very fast speeds in joint face detection and alignment. It took 16fps on a 2.60GHz CPU and 99fps on GPU (Nvidia Titan Black), which is considered to be faster than normal.

The authors' work has provided many new insights to the face detection and alignment technology using convolutional networks, and have provided us with a basic framework to work with for this project as we improve on their methods. It can be noted that the authors' used a CPU to do this process, and hence our project aims to use an FPGA to carry out the exact same process, and compare the efficiency of our FPGA with their CPU and improve on the current network should it be tested and proven that the FPGA carries out this process faster.

Overall, this report was the inspiration of this project and while this MCCN algorithm is one of the best in the current field, we hope that we can improve on this technology by utilising FPGAs to their full potential and carrying out the same process using the same framework in a shorter amount of time while maintaining a high accuracy.

Methodology

3.1 Software and Programming Language

This section will elaborate on the software and programming languages we have opted to use.

There are many FPGA softwares in the field available for use. Xilinx and Altera being the two most popular of them all, our project uses the Quartus Prime 18.1 software from Altera specifically. This software allows us to write our code and compile it to check for any errors before simulation. Also, Quartus Prime can only compile synthesized code, i.e the skeleton of our code which defines all the functions, logic gates, flip-flops etc. used in the algorithm, and should not contain any defined constant value.

The software which allows us to write a simulation of the code would be ModelSim - Intel FPGA Starter Edition 10.5b. After the compilation of code from Quartus Prime 18.1, we would write a separate code in ModelSim to simulate the synthesized code. This simulation can come in the form of input signals to defined clock periods, allowing for various types of simulation under many different conditions. ModelSim is also the software which would be used in the comparison afterwards as the result of the simulation would be displayed in ModelSim in the form of square waves, allowing for easier analysis.

As for the programming language used, FPGAs use hardware descriptive language(HDL) in particular, and of all the HDLs out there, the two most often used HDLs are Verilog and VHDL(VHSIC Hardware Description Language). For this project, we chose to use VHDL for a few reasons.

The main reason is due to VHDL being more strongly-typed as compared to Verilog. Hence for beginners like us doing our first FPGA project, it is more advantageous for us to use VHDL as it is harder for us to make mistakes because the compiler would not allow code which is invalid. On the other hand, while Verilog is more concise, leading to shorter lengths of code needing to be written, it is much easier to make mistakes as it allows coders to type invalid code.

Another reason that spurred us to use VHDL over Verilog is that VHDL is very deterministic, while Verilog is non-deterministic in certain circumstances. That is to say, while we might input the same signals on one end of the FPGA, the Verilog algorithm might exhibit different behaviours on different runs, making it unreliable when that situation occurs. This

increases the complexity of Verilog code even though it's meant to be much simpler, and it makes it confusing and maybe even unproductive to use Verilog at all. Hence, all reasons point to VHDL being an easier language to learn and begin with, thus the reason why we chose to use VHDL.

3.2 VHDL Code

This section shall explain how we convoluted a 2D array to a 1D array in VHDL to achieve a similar result to the actual Fullconnect function, explaining each process step-by-step.

In the case of our code, we intend to use the parallel processing of the FPGA to "condense" a 2D array into a single value. In order to achieve this function, we have decided to first multiply the columns into a single product before summing up the product of all the rows to get 1 single number. The array we would be working with is a 4 x 64 array of random integers from 1 to 10. The functions we used here are all instantiated by IP-cores, which are pre-written codes that we can use in our code to perform that exact function, hence accelerating design development and reducing the number of errors when compiling the code.

The IP-cores that we have used are the integer to floating point converter, the floating point multiplier and the floating point addition core. In order to use each core, we will be initiating and instantiating them as seen in the following pictures. Do note that in order to fit most of the code on one screen, we will be presenting them from Notepad++

```
11  ARCHITECTURE fc OF DSO_Fullconnect IS
12
13  --component IMGROM
22
23  component Int2fl
24      PORT
25      (
26          clock       : IN STD_LOGIC ;
27          dataa       : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
28          result      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
29      );
30  end component;
31
32  component fpmult
33      PORT
34      (
35          clock       : IN STD_LOGIC ;
36          dataa       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
37          datab       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
38          result      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
39      );
40  end component;
41
42  component fpadd
43      PORT
44      (
45          clock       : IN STD_LOGIC ;
46          dataa       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
47          datab       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
48          result      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
49      );
50  end component;
51
```

Fig 3.2.1

```
Int2flb_inst : Int2fl PORT MAP (
        clock    => clock,
        dataa    => datac,
        result   => result2
    );

Int2flc_inst : Int2fl PORT MAP (
        clock    => clock,
        dataa    => datad,
        result   => result3
    );

fpmult_inst : fpmult PORT MAP (
    clock    => clock,
    dataa    => dataa_m1,
    datab    => datab_m1,
    result   => result_m1
);

fpmult2_inst : fpmult PORT MAP (
        clock    => clock,
        dataa    => dataa_m2,
        datab => datab_m2,
        result   => result_m2
    );

fpmult3_inst : fpmult PORT MAP (
        clock    => clock,
        dataa    => result_m1,
        datab    => result_m2,
        result   => result_m3
);
```

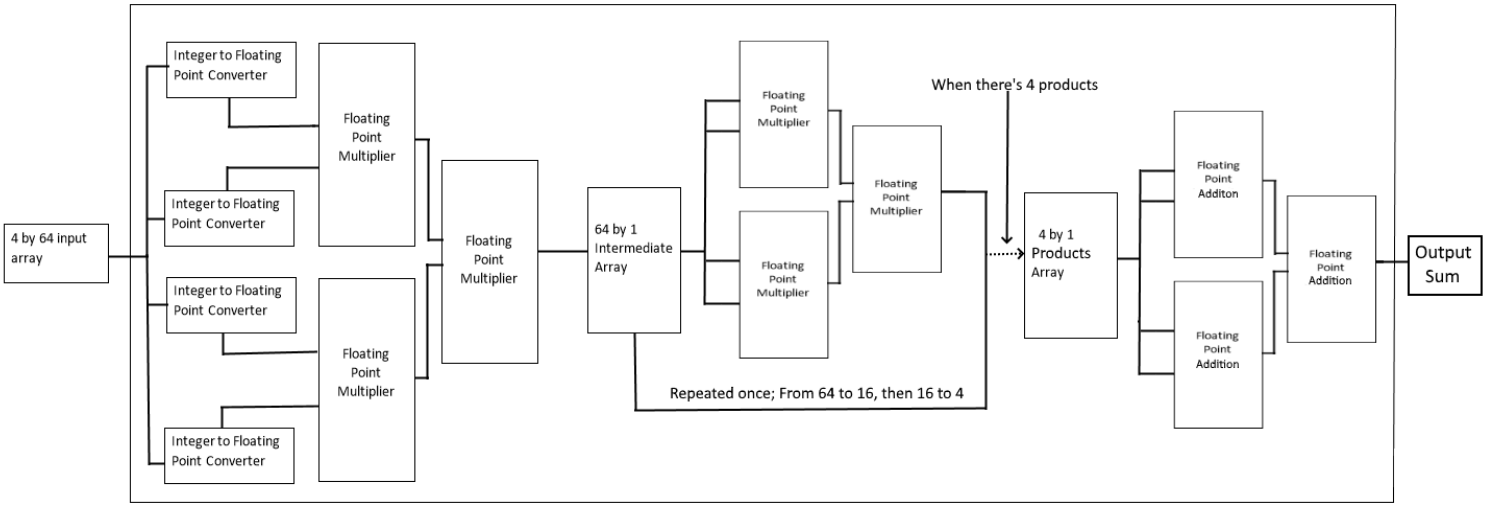
Fig 3.2.2

Having initialised and instantiated the IP cores and configuring their port maps to signals, we will be able to directly access them later on in the code.

However, before we continue to explain the code, there are a few components of the FPGA which will be brought up in a while that needs a brief explanation of their function. The most important function would be the clock in an FPGA. FPGAs run on synchronous synthesizing, which means that all the processes run based on the FPGAs local clock cycles. The lowest clock period which one can set for an FPGA is 5 nanoseconds. While we are using a clock running on 1 nanosecond for the simulation, we would multiply the time taken by 5 to determine how long our code would run on an actual fpga.

Additionally, due to the use of a clock in VHDL, the code would run similar to a never ending while loop, continuously running its processes every clock cycle. Hence, we have made use of if statements in our code to change the code that would be processed through each time. For instance, once the multiplication of values are done, an if statement would be used to change the code being processed for the addition phase.

The following will be an overview of the code. Each IP Core component used and the code in general will be explained along the way in the following sections.


4 by 64 input array
Integer to Floating Point Converter
Integer to Floating Point Converter
Integer to Floating Point Converter
Integer to Floating Point Converter
Floating Point Multiplier
Floating Point Multiplier
Floating Point Multiplier
64 by 1 Intermediate Array
Floating Point Multiplier
Floating Point Multiplier
Floating Point Multiplier
When there's 4 products
4 by 1 Products Array
Floating Point Additon
Floating Point Addition
Floating Point Addition
Output Sum
Repeated once; From 64 to 16, then 16 to 4

3.2.1: Integer to Floating-Point Converter

The first part of the algorithm is to input the values read from the array into an integer to floating converter IP core. The basic function of this core is to convert an 8-bit integer value into a 32-bit single precision decimal value. This is to increase the accuracy of our values and the overall accuracy of our algorithm. To understand how this function works, we first need to understand the way floating point representation works.

Floating-point representation has a complex encoding scheme with three basic components: mantissa, exponent and sign. Usage of binary numeration and powers of 2 resulted in floating point numbers being as single precision (32-bit) and double precision (64-bit) floating-point numbers. Both single and double precision numbers are defined by the IEEE 754 standard. According to the standard presentation, a single precision number has one sign bit, 8 exponent bits and 23 mantissa bits where as a double precision number comprises of one sign bit, 11 exponent bits and 52 mantissa bits[3].

When converting an integer to a floating point number, the binary representation of the integer is just shifted until the mantissa is within the right range i.e $1 < m < 2$, and the exponent is just how many steps it shifts. The rest of the 23-bit mantissa would be filled up with zeroes should there be enough space.

The segment of the code used during the conversion of integer to floating point will be in the next section of this report, as it involves the first round of multiplication as well.

3.2.2: Floating Point Multiplier

After the input values have gone through the integer to floating point converter, the now 32-bit values are then multiplied together in a floating point multiplier IP core. This IP core can only multiply two floating values at a time, hence in order to get the most out of the FPGAs parallel processing abilities, we have decided to use 3 floating point multipliers at the same time.

In short, the below processes explains how floating point multipliers work[4].

1. Obtaining the sign; i.e. S1 xor S2.
2. Adding the exponents and subtracting the bias value; i.e. (E1 + E2 – Bias).
3. Multiplying the significand; i.e. (1.M1*1.M2).
4. Placing the decimal point in the significand result.
5. Normalizing the result; i.e. obtaining 1 at the MSB of the results significand.
6. Rounding the result to fit in the available bits.
7. Checking for underflow/overflow occurrence.

Underflow and overflow are errors which might occur during simulation of code due to incorrect synchronous synthesizing, which refers to the incorrect assignment of clock cycles to the IP cores. Underflow occurs when the number of clock cycles used for each floating point multiplier is overestimated, causing the IP core to carry out its function even though there are no input values. This will cause the output to have errors. On the other hand, overflow occurs when one underestimates the number of clock cycles required for an IP core to carry out its function, and this results in errors when the number of values input into the IP core exceeds the limit. Hence, while using IP cores, it is important to synchronize the clock along with the IP cores in order to prevent these errors from occuring.

The first few clock cycles will utilise two floating point multipliers to multiply 4 floating point values at the same time. Afterwards, the results from these 2 multipliers will then be input into the third floating point multiplier in order to get the multiplication value of 4 floating point values.
These values will then be stored 64 x 1 array for the second round of multiplication.

The code used in the conversion from integer to floating point, followed by the first round of multiplication is as follows:

```
process (clock)
begin
if rising_edge(clock) then
        if i < 4 then
            if j < 15 then
                dataa <= s_mem(i,4*j);
                datab <= s_mem(i,4*j+1);
                datac <= s_mem(i,4*j+2);
                datad <= s_mem(i,4*j+3);
                dataa_m1 <= result;
                datab_m1 <= result1;
                dataa_m2 <= result2;
                datab_m2 <= result3;
                if (cnt > 29) then
                    s_mem1(cnt-30) <= result_m3;
                end if;
                j <= j+1;
                cnt <= cnt + 1;
            else
                dataa <= s_mem(i,4*j);
                datab <= s_mem(i,4*j+1);
                datac <= s_mem(i,4*j+2);
                datad <= s_mem(i,4*j+3);
                dataa_m1 <= result;
                datab_m1 <= result1;
                dataa_m2 <= result2;
                datab_m2 <= result3;
                if (cnt > 29) then
                    s_mem1(cnt-30) <= result_m3;
                end if;
                cnt <= cnt + 1;
                i <= i+1;
                j <= 0;
            end if;
```

Fig 3.3.2.1

As seen in the code, for every value of i which represents the number of columns of the array, the code will run 64/4 = 16 times per column. During every run, it will take in 4 integer values and convert them into floating points. After which, it will take the results and input them into the 3 multipliers used. The first and second multiplier would receive the four values, and their outputs would feed into the third multiplier. The result of the third multiplier would then be inputted into the array. In order to prevent underflow or overflow, a signal, cnt, is used to determine when the third multiplier is done, which would input the result into the 64 by 1 array, otherwise known as s_mem1. In this code, we used the condition j < 15 so that when j reaches 15, i would increase by 1 while j is set back to 0, thereby completing the inputting of values for one column of the 4 by 64 array.

```
elsif m < 30 then
    j <= 65;
    i <= 5;
    if m < 7 then
        dataa_m1 <= result;
        datab_m1 <= result1;
        dataa_m2 <= result2;
        datab_m2 <= result3;
    end if;
    s_mem1(m+34) <= result_m3; --remaining 30 values
    m <= m+1;
```

Fig 3.3.2.2

Of course, since the values are only inputted when cnt is at 30 and above from Fig 3.3.2.1, there would still be 30 values remaining. The 30 values would be obtained and inputted into the array. Additionally, as there will be 7 more values obtained from the integer to floating point converters, we have inputted them into the floating point multiplier as shown.

The second round of multiplication will use another 3 floating point multipliers, repeating the process as mentioned in the above paragraph. The values obtained after this second round of multiplication will be input back into the 64 x 1 array and this process is looped to achieve a final single value for each row. Simply put, after the first round of multiplication, the 64 values will boil down to 16 values, and the second and third round of multiplication will be conducted using the second set of floating point multipliers to obtain 1 final value for each row. The code used will be as follows:

```
elsif h < 16 then
    j <= 65;
    m <= 31;
    i <= 5;
    dataa_m3 <= s_mem1(4*h);
    datab_m3 <= s_mem1(4*h+1);
    dataa_m4 <= s_mem1(4*h+2);
    datab_m4 <= s_mem1(4*h+3);
    h <= h+1;
elsif h < 23 then
    j <= 65;
    m <= 31;
    i <= 5;
    h <= h+1;
elsif o < 16 then
    j <= 65;
    m <= 31;
    i <= 5;
    h <= 24;
    s_mem1(o) <= result_m6;
    o <= o+1;
```

Fig 3.2.2.3

In this code, we set a separate signal, h, to send the 64 values from the intermediate 64 by 1 array into 3 other floating point multipliers. The fourth and fifth floating point multipliers would obtain the data from the intermediate s_mem1 array, with their results wired as the inputs of the sixth floating point multiplier. After a specific delay of 23 clock cycles, we would then take the result_m6, the result of the 6th floating point multiplier and input it back into the s_mem1. Now, we would end up with 16 products. The third round of multiplication would be as follows:

```
elsif l < 4 then
    j <= 65;
    k <= 13;
    m <= 31;
    i <= 5;
    h <= 17;
    o <= 17;
    dataa_m3 <= s_mem1(4*l);
    datab_m3 <= s_mem1(4*l+1);
    dataa_m4 <= s_mem1(4*l+2);
    datab_m4 <= s_mem1(4*l+3);
    l <= l+1;
elsif l < 8 then
    j <= 65;
    k <= 13;
    m <= 31;
    i <= 5;
    h <= 17;
    o <= 17;
    if l > 3 then s_products(l-4) <= result_m6;
    end if;
    l <= l+1;
```

Fig 3.2.2.4

As seen in the code, the fourth, fifth and sixth floating point multipliers would be reused, taking the remaining 16 values in s_mem1 and multiplying them in the same fashion to obtain 4 products of each row. As the values used in multiplying are consecutive to one another, the 4 products would be the respective products of each column in the 4 x 64 array. The 4 products would then be inputted into a separate s_products array after a specific delay.

3.2.3: Floating Point Addition

After obtaining the four values of the multiplication of each row, these values are stored in a third array which spans 4 x 1. The final step is to sum these values up to condense the 2D array into a 1D array. Just like floating point multipliers, the IP cores of floating point addition can only add up two floating values every single time, hence we used 3 floating point addition cores, first taking the sum of the first 2 and last 2 values separately, then input the 2 values obtained into the final floating point addition core to obtain the final 1D result of a 4 x 64 array. The code will be as follows:

```
                i <= i+1;
            else
                dataa_a1 <= s_products(0);
                datab_a1 <= s_products(1);
                dataa_a2 <= s_products(2);
                datab_a2 <= s_products(3);

                sum1 <= result_a3;
            end if;
        --end if;
    end if;
end process;
```

Fig 3.2.3

In the code, the products would be fed into 3 floating point addition cores, each wired similar to the floating point multipliers. After running it a few times, the sum, result_a3, would be written into a signal known as sum1, which is the value obtained at the end of the code.

Regarding the Floating Point Addition core, it has quite a complicated process of conditional statements to go through before the summation is carried out. Usually, there would be 2 cases to take note of, the first being a summation of 2 values with the same sign, and the other being a summation of 2 values with opposite signs. In our case, our integers are all positive integers, hence only the process for 1 case will be presented. The following are the steps involved in floating point addition[5].

Step 1:- Enter two numbers N1 and N2. E1, S1 and E1, S2 represent exponent and significand of N1 and N2 respectively.

Step 2:-Is E1 or E2 =‟0‟. If yes; set hidden bit of N1 or N2 is zero. If not; then check if E2 > E1, if yes swap N1 and N2 and if E1 > E2; contents of N1 and N2 need not to be swapped.
Step 3:- Calculate difference in exponents d=E1-E2. If d= “0” then there is no need of shifting the significand. If d is more than “0” say “y” then shift S2 to the right by an amount „y‟ and fill the left most bits with zero. Shifting is done with hidden bit.
Step 4:- Amount of shifting i.e. “y” is added to the exponent of the N2 value. New exponent value of E2= (previous E2) + “y”. Now result is in normalize form because E1 = E2.
Step 5:- Check if N1 and N2 have different sign
Step 6:- Add the significands of 24 bits each including hidden bit S=S1+S2. Step 7:- Check if there is carry out in significand addition. If yes; then add „1‟ to the exponent value of either E1 or new E2. After addition, shift the overall result of significand addition to the right by one by making MSB of S as „1‟ and dropping LSB of significand.
Step 8:- If there is no carry out in step 6, then previous exponent is the real exponent.
Step 9:- Sign of the result i.e. MSB = MSB of either N1 or N2.
Step 10:- Assemble result into 32 bit format excluding 24th bit of significand i.e. hidden bit.

3.2.4: Conclusion

This will mark the end of the VHDL code. The algorithm is made up of an overall of 4 integer to floating point conversion, 6 floating point multiplier and 3 floating point addition IP cores. The whole process runs at a clock speed of 5 nanoseconds per clock cycle, and use 3 arrays - a 4 x 64 initial array, a 64 x 1 intermediate array to store intermediate values from first round multiplication and a 4 x 1 array for addition.

3.3 C++ Code

This will be a brief explanation on how the C++ code will mirror the VHDL code so as to provide a fair ground for comparison between the efficiency of FPGAs and microcontrollers.

The 256 integers are again stored in an array and for the multiplication of values, loops will be used to multiply the values from 256 values to 64 to 16 to 4. The multiplication process is the same as VHDL coding where 4 values are multiplied in a single process to obtain the multiplicand of the 4 values, and the process is repeated. Finally, the 4 final products will be stored in an array before the addition of the 4 values are carried out. The final result will match that of the VHDL code, making this a fair comparison.

One slight difference in the C++ code from the VHDL code is that the C++ code carries out multiplication of integer values read straight from the array unlike the VHDL code which converts the values into floating points beforehand so as to accumulate enough bits for multiplication to occur. This process might seem to favour the C++ code as it takes lesser steps to carry out the same process. That being said, the efficiency of the FPGA is not jeopardised as while FPGAs might seem to need more steps to meet the requirements for the same process, they make it up with their superior processing speed.

The following page is the image of our C++ Code, presented on Notepad++:

change.log | C++ Code (Notepad++)

```
1   #include <bits/stdc++.h>
2   using namespace std;
3
4   int main()
5   {
6       clock_t tStart = clock();
7       double i=1;
8       int j;
9       int k;
10      double arr[255];
11      if(true){ //just to close the 256 lines for easier reading
274     for(j=0;j<64;j++){
275         arr[j]=arr[4*j]*arr[4*j+1]*arr[4*j+2]*arr[4*j+3];
276         //cout << arr[j] << " " << j << endl;
277     }
278     cout << endl;
279     for(j=0;j<16;j++){
280         arr[j]=arr[4*j]*arr[4*j+1]*arr[4*j+2]*arr[4*j+3];
281         //cout << arr[j] << " " << j << endl;
282     }
283     //cout << endl;
284      for(j=0;j<4;j++){
285         arr[j]=arr[4*j]*arr[4*j+1]*arr[4*j+2]*arr[4*j+3];
286         //cout << arr[j] << " " << j << endl;
287     }
288     //cout << endl;
289     arr[0]=arr[0]+arr[1];
290     arr[1]=arr[2]+arr[3];
291    // cout << arr[0] << endl << arr[1] << endl;
292     cout << arr[0] + arr[1] << endl;
293     printf("Time taken: %.9fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);
294     return 0;
295 }
```

Fig 3.3.1

This marks the end of the explanation of both codes. Before sharing our results and conclusion, we will present how our jobs are distributed and our timeline thus far.

3.4 Job Distribution

The following table is the job distribution of this project.

Member	**Roles**
Denzel Chia Wen Xuan (4S2)	Coding Written Report
David Lim Kang Wei (4S2)	Project slides and documentation Coding

3.5 Timeline

Our timeline is as follows:

Date	**Overview**	**Outcome**
~1st Mar	● Looked through DSO project listings	● Shortlisted a number of projects we were interested in, applied for projects we were interested in
21 Mar	● Interview for project - FPGA implementation for imaging	● Understood more project details from the mentors, shared our own knowledge in this field
22nd Mar	● Accepted to take up project - FPGA implementation for imaging	● Marks the start of our project
27th Mar	● First meeting with DSO mentors	● Discussed details of the project, such as what software to use. Brief exposure on VHDL language
7th Apr	● Proposal Evaluation	● Information dissemination to school judges on what our project is about, and how we are going to go about doing the project. Passed the proposal round.
15th Apr ~ 30th May	● Learning of VHDL code, as well as doing mini projects to test out certain concepts	● Helped us get comfortable with VHDL coding
30 May	● Met with DSO mentors	● Compilation of VHDL coding knowledge with mentors, discussed how to carry out the project going forward
24 Jun	● Met with DSO mentors	● Discussed about the use of floating point multipliers and implemented them in our code
1st July	● Met with DSO mentors	● Discussed what we were going to present for mid-term evaluation, resolved certain errors in the code
4th July	● Mid-Term Evaluation	● Presentation and explanation of code up till the multiplication process
17th July	● Met with DSO mentors	● Discuss about what to present for project

		finals, as well as how to go about doing it
7th Aug	● Project finals	● Presentation of completed school project.
17 Nov - 29 Dec	● 6 week attachment with DSO	● Improvements on code and field-testing on actual FPGA ● Completion of DSO project at the end of the year

Results and Discussion

4.1 C++ Code

The following are the results we have obtained from the C++ Code.

From the code itself, upon running it on an online compiler, in which OnlineGDB beta was used, it took 0.000093 seconds, which translates into 9.3 * 10^-5 seconds, or otherwise 93 microseconds, as shown in Figure 4.1.1.

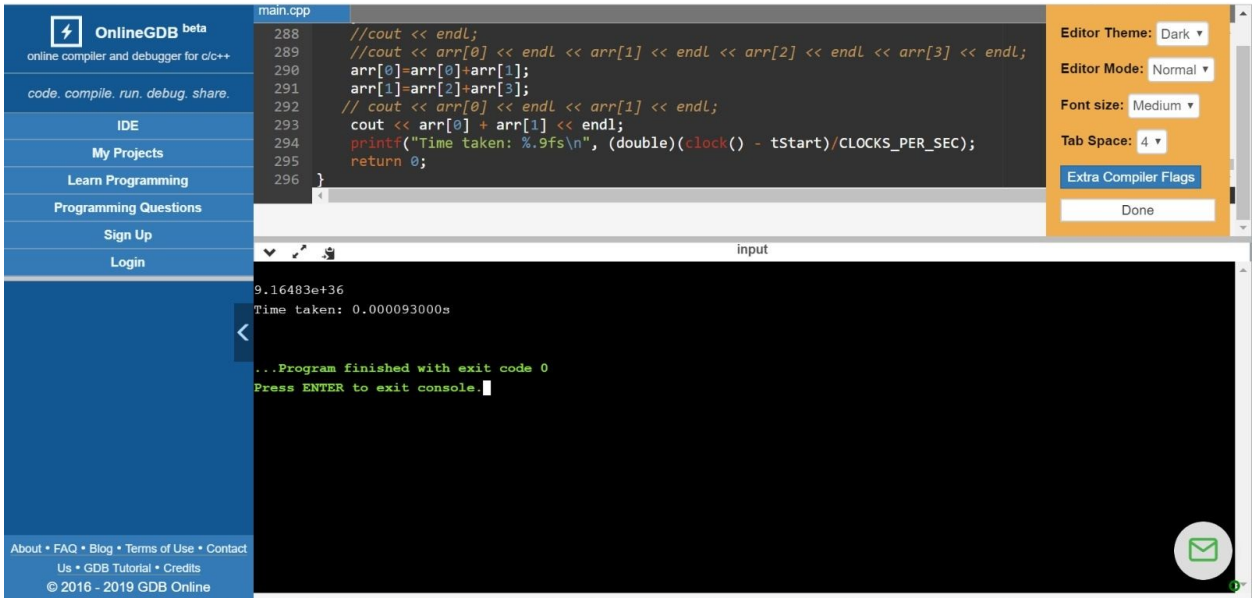


Fig 4.1.1

Additionally, we ran the code another time, this time with the product values of each of the 64 numbers. This is to compare with the results obtained on ModelSim from the VHDL code as proof that the code is functional. The results are as follows:

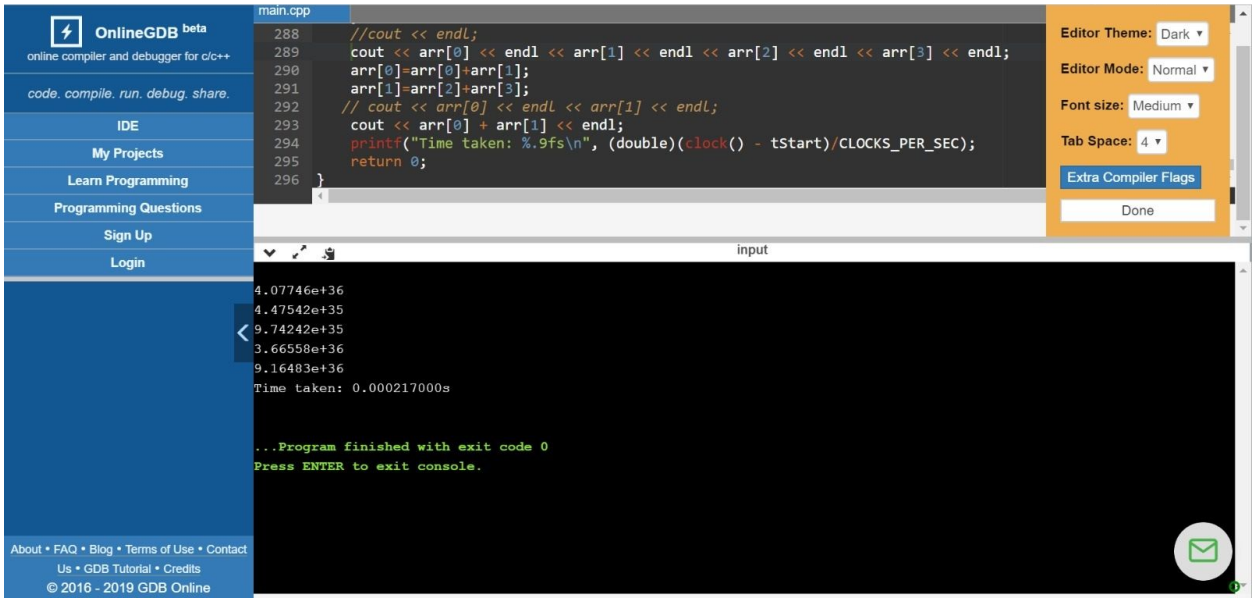


Fig 4.1.2

As shown, the products for values of array[0] to array[63], array[64] to array[127], array[128] to array[191], array[192] to array[255] are 4.07746*10^36, 4.47542*10^35, 9.74242*10^35 and 3.66558*10^36 respectively, with the total sum of 9.16483*10^36 as shown in Fig 4.1.2.

4.2 VHDL Code

The following will be the results we have obtained from the VHDL code upon simulating it through ModelSim. As ModelSim's simulation comes in the form of input signals under defined clock periods, the results will be in the form of multiple waveforms of values stored in the original 4 by 64 input array, the 64 by 1 array used, the 4 by 1 product array, the values in the IP cores among others. Since we are using 7-bit integers and 32-bit floating points, the values will be in the form of '1's and '0's, as shown:

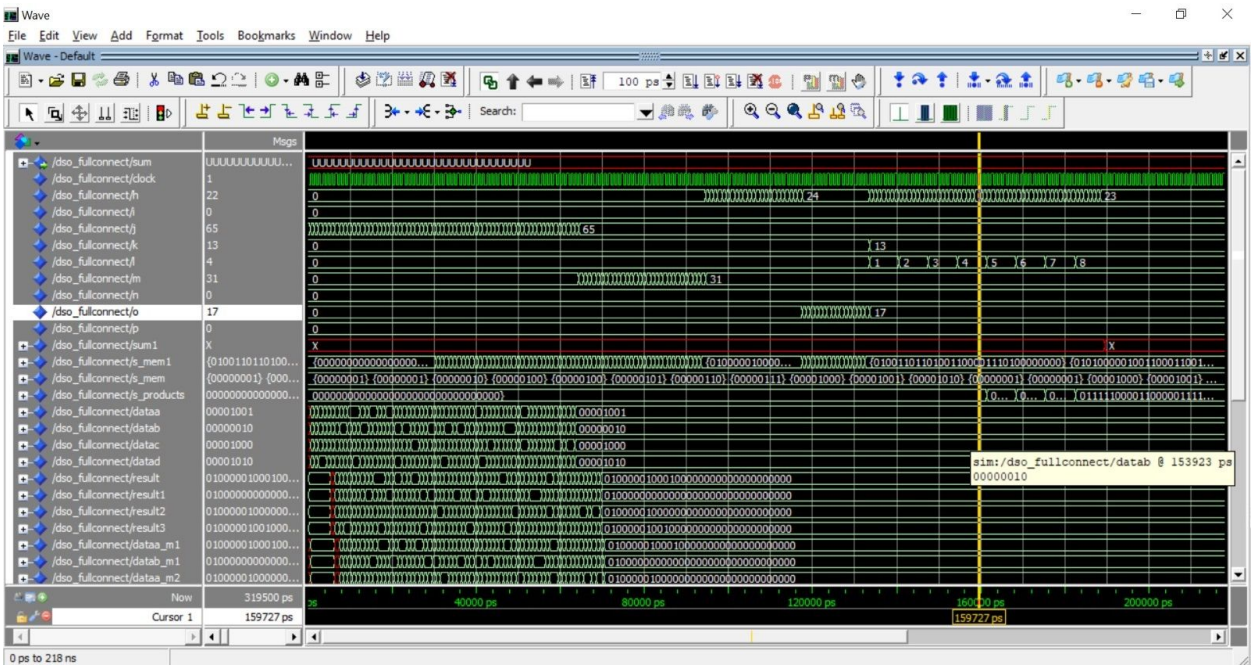


Fig 4.2.1

The binary values seen in most of the signals in the figure above are of the final value. Regarding the values of the 4 products as well as the sum, we have changed their display from binary to fp32, which displays the results in their scientific notation. The following are the results.

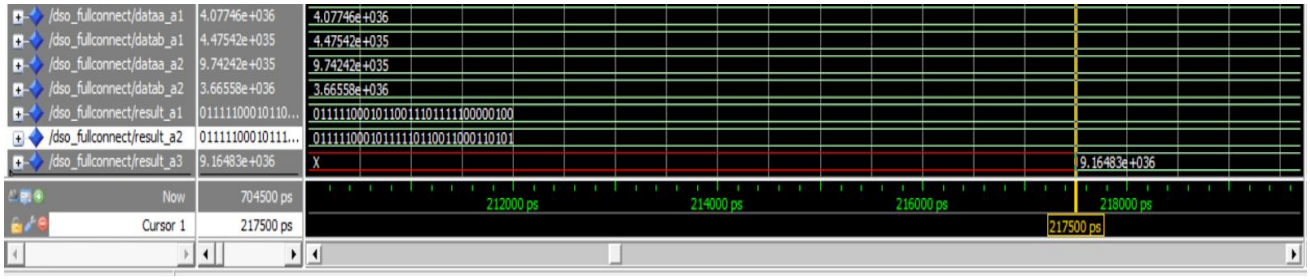


Fig 4.2.2

As shown, the values of the four products from the (0,0) to (0,63), (1,0) to (1,63), (2,0) to (2,63), (3,0) to (3,63) are 4.07746*10^36, 4.47542*10^35, 9.74242*10^35 and 3.66558*10^36, as seen in signals dataa_a1, datab_a1, dataa_a2 and datab_a2 respectively. It should be noted that signals result_a1 and result_a2 are results of the additions of dataa_a1 and datab_a1, dataa_a2 and datab_a2 respectively. Lastly, the sum of all 4 products can be seen in signal result_a3, which is 9.16483*10^36. As these values match up with those in the C++ Code, this shows that our VHDL code is accurate and has the right values.

Furthermore, the time taken for the VHDL code to output the sum is a mere 217.5 nanoseconds, equivalent to 0.218 microseconds. Since this was under a clock cycle of 1 nanosecond, under the standard 5 nanoseconds, it would take 1.09 microseconds. Comparing this to the C++ Code of 93 microseconds, the VHDL code is around 85 times faster than the C++ code. This clearly presents the superiority in processing speed of FPGAs over microcontrollers, while still maintaining accuracy through the different processes.

Conclusion and Future Plans

In conclusion, we have successfully managed to code a process similar to that of Fullconnect and managed to convolute a 2D array to a 1D array in VHDL. The code is then put to the test by comparing it with a C++ code, and we have achieved convincing results based on simulation results which showed that FPGAs have the potential to process things hundreds of times faster than microcontrollers. This confirms the claim that FPGAs, while being more costly than microcontrollers, have the advantage of high efficiency in completing processes.

Our next step in our project would be building on the foundations of this simple 4 x 64 array code, and code in the actual Fullconnect algorithm while reading from an S memory on a much larger scale, which is estimated to be a 1152 x 256 2D array. This will call for a much more IP cores to be used, such as floating point multipliers, in order to further improve the efficiency. The future code will also be field tested, meaning that we would be using an actual FPGA to test it out, instead of limiting the code to simulation using ModelSim. This will factor in real-life conditions which might cause discrepancies to arise in the code which did not surface in an ideal environment such as ModelSim, thus we would need to revise our code multiple times in order to successfully run it in an FPGA.

Personal Reflections

David:

Personally, this Infocomm project was amazing and overall, pretty fun. Despite the pains of debugging, the stress of learning a new code and learning how to use two different tools (namely Quartus and ModelSim), in the end, we were able to complete it successfully, and seeing that I could achieve such a feat on our own, completely justifies the fun journey I had gone through. Nonetheless, despite this project receiving a closure during Finals, our project would still continue with our attachment with the DSO mentors, wherein we will code the fullconnect portion of the algorithm. In conclusion, this was one real fun project to do, and I look forward with great interest on the attachment with my DSO mentors in the November and December Holidays.

Denzel:

This project has expanded my horizon and has let me gain exposure to the rarely heard of hardware descriptive language and the coding of hardware. Through the course of this project, I have picked up the basics of the VHDL language and have successfully managed to mirror the fullconnect function in the current code, and have managed to build a solid foundation of knowledge on the VHDL language, which would certainly enable me to code the actual fullconnect algorithm more smoothly. I have also learnt that communication with our mentors are very important as this being part of a DSO project, it is not as convenient to meet up with our mentors unlike our school mentor which we can easily arrange a time to meet up. Hence, it taught me to use the best of our meetings together and to work on the project efficiently, and constantly updating the mentors through Whatsapp or Email to keep them in the loop, preventing any forms of miscommunication.

Bibliographies

(1) What is field-programmable gate array (FPGA)? - Definition from WhatIs.com. (n.d.). Retrieved from https://whatis.techtarget.com/definition/field-programmable-gate-array-FPGA

(2) FPGA vs Microcontroller - Advantages of Using An FPGA. (2019, April 02). Retrieved from https://duotechservices.com/fpga-vs-microcontroller-advantages-of-using-fpga

(3) Hollasch, S. (n.d.). IEEE Standard 754 Floating Point Numbers. Retrieved from https://steve.hollasch.net/cgindex/coding/ieeefloat.html

(4) V. C., Chary, & Rao, V. V., Prasad. (2015). An Optimized Implementation of IEEE-754 Floating Point Multiplier for DSP Applications. *International Journal of Research in Advanced Engineering Technologies,5*(1), 304-309. Retrieved June 25, 2019, from http://www.ijraet.com/pdf17/42.pdf

(5) N., Grover, & M. K., Soni. (2014). Design of FPGA based 32-bit Floating Point Arithmetic Unit and verification of its VHDL code using MATLAB. *International Journal of Information Engineering and Electronic Business(IJIEEB),6*(1), 1-14. doi:10.5815/ijieeb.2014.01.0