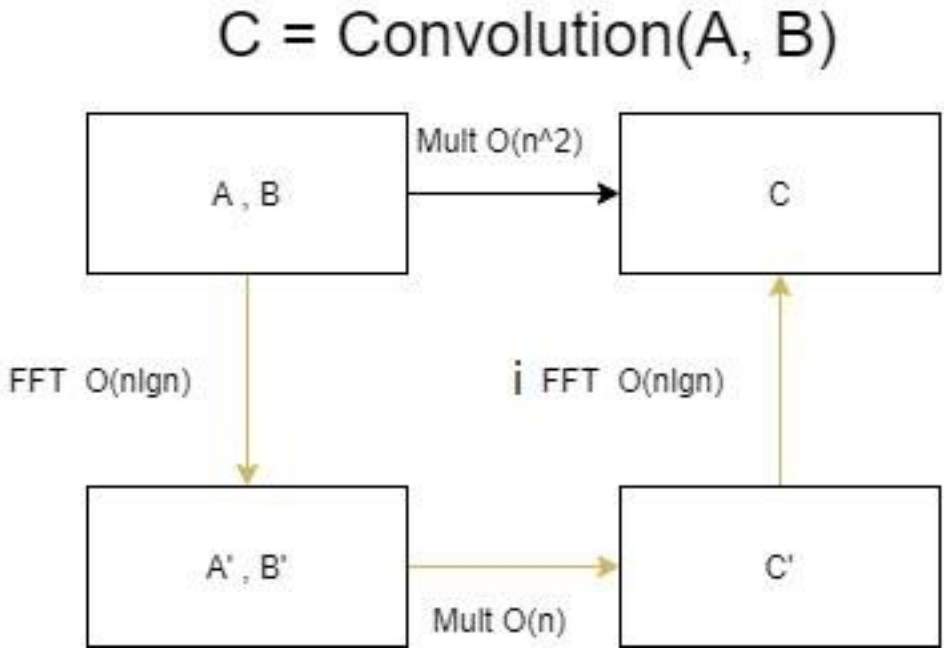
# 快速傅立葉轉換在二維捲積的硬體加速器實作

## 1. Problem Description

Convolution 因為大量重複 Multiply and Accumulate(MAC)的運算性質，使其非常適合設計硬體來加速。因而本次選擇此計算架構做為設計加速器的應用。

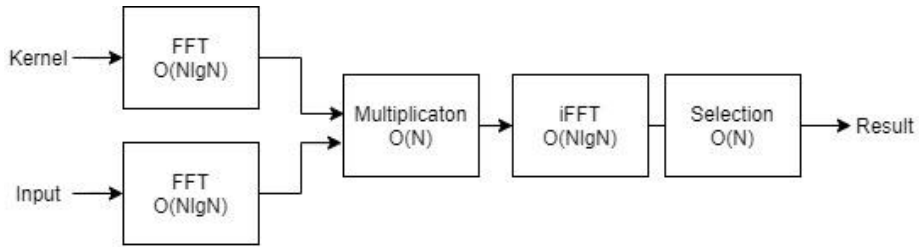



2D Convolution 在最簡單的計算會需要 O(N^2)的時間來計算，也就是上圖直接做乘法的黑色路徑。在這次實驗我會實作黃色路徑 O(NlgN)的快速傅立葉轉換捲積。

# 2. Design Concept

## (1) Algorithm flow chart

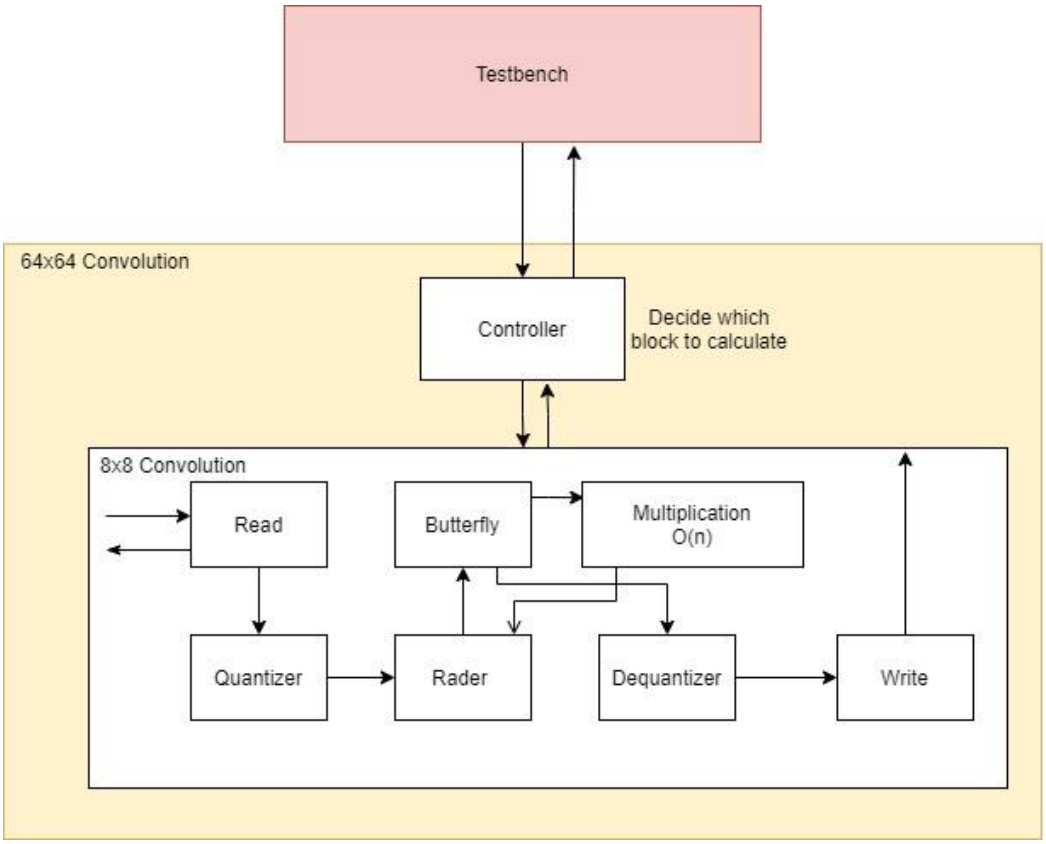
FFT Convolution 的流程圖如 Problem description 提到，會先對 Input, Kernel 做 FFT，再對兩個 frequency domain 做內積但每個乘出來的值獨立存放不累加，最後對乘出來的陣列做 Invert FFT，資料選擇後即可拿到需要的輸出。



# 3. Verilog Design

在架構設計中，我將 Input, Weight 的 SRAM 讀取，以及 Output SRAM 的寫入整合進 testbench，加速器要自行決定資料位址並與 testbench 傳輸。

Top module 為預設 64x64 的 convolution module,它可以透過改變參數來實現 128x128 甚至更大的計算硬體, 但考慮模擬時間與實驗的計算複雜度需求，最終選擇了 64x64 的大小。




由於根據需求(64x64, 128x128, 或是更大)來實作對應大小的 FFT 並不實際，所以這個 module 主要分為兩部分。

如上圖，第一部分為 controller, 它負責 data tiling, 將

64x64 的 convolution 切成 121 組 8x8 convolution，重複使用 8x8 Convolution 硬體來計算。

在 Controller 的部分，因為 Data tiling 會將 64x64 -> 62x62 的 convolution 切成很多份 8x8 -> 6x6 的小 part，也因為長寬不一定整除 6，譬如 64x64 就沒辦法，因此我切成分成四部分來做計算以滿足各種長寬的需求。

(i) Main

計算(row, col): (0,0) ~ (60, 60)的 100 個小 part。如下圖左上角

(ii) Rim-row

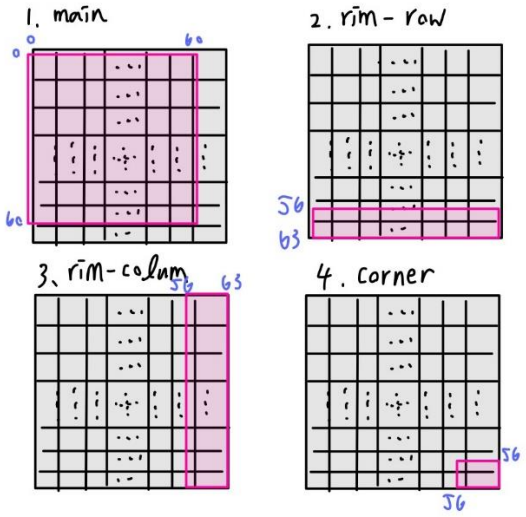
計算(row, col): (56,0) ~ (60, 60)的 10 個小 part。如下圖右上角

(iii)Rim-Colum

計算(row, col): (0,56) ~ (60, 60)的 10 個小 part。如下圖右上角

(iv) Corner

計算最右下角的 1 個 block，(row, col):(56,56)



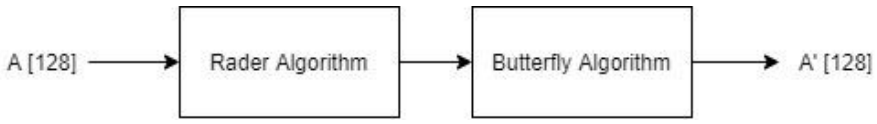

Controller 便會根據以上四種狀態去決定計算 block 的 base address(每個 block 左上角的絕對位址)，並會在讀寫資料時將 base address 加到 8x8 Convolution module 需求的相對位址。

```
always @(*) begin

    input_addr = (SM_input_addr_i + addr_base_i) * input_W + addr_base_j + SM_input_addr_j;
    weight_addr = SM_input_Weight_addr_i* kernel_W + SM_input_Weight_addr_j;
    output_addr = (SM_output_addr_i + addr_base_i) * output_W + addr_base_j + SM_output_addr_j;

end
```

FFT 的演算法的部分，我先使用 Rader Algorithm 來把 Input 重新排列成 Butterfly 演算法需求的 input 格式再做計算。




但由於 Butterfly 合併操作演算法原版是使用高精度浮點數以及三角函數來計算，為了在不支援浮點數運算的

Verilog 內實作，我使用了 Quantization 來將所有運算轉換至整數型態運算，這部分我是先在 C++ golden model 內實作出來並計算 error rate，調整到可接受範圍後才用 Verilog 實作。

在 golden model 內我先將所有 input，包含三角函數的輸出值往左 shift 21 bits，搭配使用 64-bits 整數才做到極小誤差，大約萬分之一。並且我將 quantized 後的三角函數根據 butterfly 演算法內已知的 input(h, on)直接輸出到 Verilog 內，以達到最高的精確度，不是有選擇經常會使用的將三角函數切成 512 份再做計算的較大誤差版本。

```
module COS(
    input wire on,
    input wire [10:0] h,
    output reg signed[`BITS - 1:0] value
);
    always @(*) begin
        if(on == 1) begin//on = 1 in C++
            case(h)
                2       : value = -2097152;
                4       : value = 0;
                8       : value = 1482910;
                16      : value = 1937515;
                32      : value = 2056855;
                64      : value = 2087053;
                128     : value = 2094625;
                default : value = 0;
            endcase
        end
        else begin // on = -1 in C++
            case(h)
                2       : value = -2097152;
                4       : value = 0;
                8       : value = 1482910;
                16      : value = 1937515;
                32      : value = 2056855;
                64      : value = 2087053;
                128     : value = 2094625;
                default : value = 0;
            endcase
        end
    end

endmodule
```

```
module SIN(
    input wire on,
    input wire [10:0] h,
    output reg signed[`BITS - 1:0] value
);
    always @(*) begin
        if(on == 1) begin//on = 1 in C++
            case(h)
                2       : value = 0;
                4       : value = -2097152;
                8       : value = -1482910;
                16      : value = -802545;
                32      : value = -409134;
                64      : value = -205556;
                128     : value = -102902;
                default : value = 0;
            endcase
        end
        else begin // on = -1 in C++
            case(h)
                2       : value = 0;
                4       : value = 2097152;
                8       : value = 1482910;
                16      : value = 802545;
                32      : value = 409134;
                64      : value = 205556;
                128     : value = 102902;
                default : value = 0;
            endcase
        end
    end

endmodule
```

此外，由於 C++只能使用固定 bits 數的整數(例如 32bits, 64bits)，我在實驗時發現在 Verilog design 使用

65-bits 能達到比 golden 還低一些的 error rate, 平均錯誤率大約為萬分之一。下圖為 Error rate 計算公式:

```
double error = abs(abs(Result) - abs(Golden)) / abs(Golden);
```

而蝴蝶合併操作會需要複數運算，但因為做過 quantization，在複數乘法時就需要 dequantized，由於我的 bits 開到 65，所以在不會 overflow 的情況下我選擇全部算完再往右 shift 21 bits，這樣便可以**保留較多**的資料降低誤差。下圖為複數乘法實根相加部分。

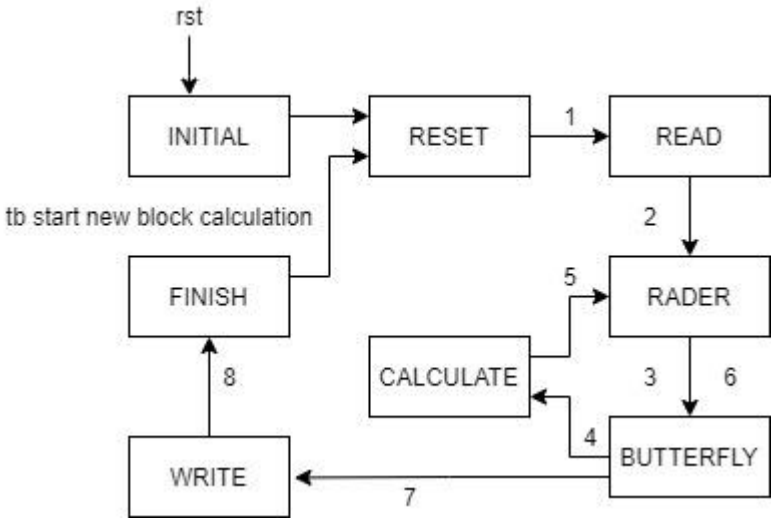
```
C_r = (A_r*B_r - A_i*B_i) <<< 21
```

而不是選擇先做 partial dequantized 再累加避免 overflow 的實作方法。圖為複數乘法實根相加部分。

```
C_r = ((A_r*B_r)<<< 21) - ((A_i*B_i) <<< 21)
```

最後是下圖的 state machine，系統在 reset 的 INITIAL 後會進到 RESET，歸 0 所有 buffer 後進到 READ 讀 Input 和 kernel，之後便會進到 FFT 計算的 RADER 和 BUTTERFLY 來做快速傅立葉轉換，在經過 CALCULATE O(N)的相乘後會透過改變旋轉因子的參數，使用同一組 FFT 電路 RADER、BUTTERFLY 來做逆向 FFT，最終於 WRITE 寫出後便會進到

FINISH 等待 controller reset 開始新一輪計算。



# 4. Achievement

Quantization Error

Quantized FFT 因為原本是使用高精度浮點數計算，轉換成整數運算後無法避免誤差的發生，但是仔細調整 Quantization 參數後，最終結果的誤差小到幾乎可忽略。平均錯誤率僅有萬分之一。

- Average error rate: **0.016845%**
- Max error rate: **0.277778%**

# 5. Reference

FFT 演算法 Golden Model 修改自：

https://www.itread01.com/content/1546333932.html