UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação
SSC5883 - Computação Reconfigurável (2025)

Explorando uma arquitetura simples de CNN em um *softcore* em FPGA

Aluno: Leonardo Zaniboni Silva
Nusp: 11801049
Docente: Prof. Dr. Vanderlei Bognato

Sumário

1 Introdução

Este trabalho tem como objetivo executar em um *softcore* uma CNN com uma arquitetura simples. O *softcore* escolhido foi o *Microblaze*. Para tal, as ferramentas utilizadas foram as seguintes:

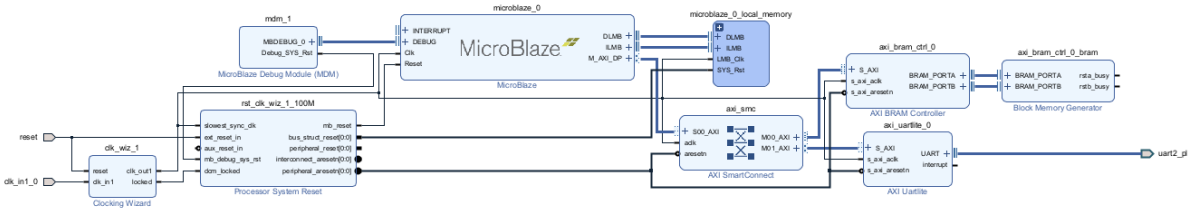
- Softwares: Vivado (confecção de *hardware* e simulação) e Vitis (confecção do software).

Observação: Inicialmente, eu tinha optado por desenvolver utilizando o *softcore neorv32*. Entretanto, obtive problemas para simular a operação do arquivo binário. Nesse contexto, migrei para o Microblaze devido as ferramentas do *Vivado*.

1.1 Criação do softcore

No software *Vivado*, criou-se o *bitstream* referente ao *softcore Microblaze*. A figura 1 apresenta o design realizado.

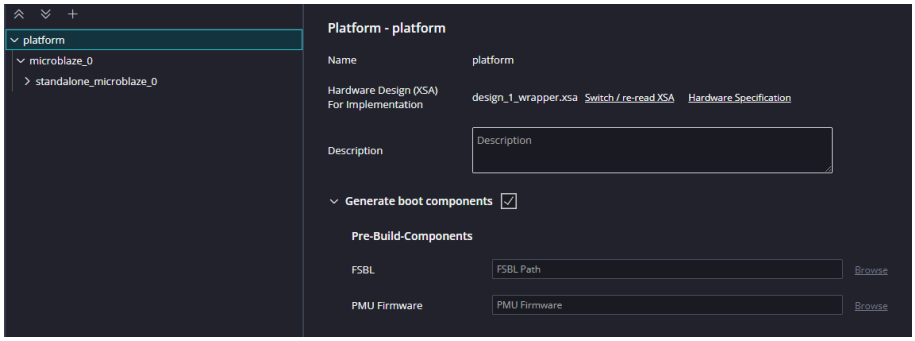
Figura 1: Microblaze IP.



Fonte: Elaborado pelo autor.

Uma vez finalizado o roteamento dos IPs, foi gerado o arquivo .xsa (Xilinx Support Archive). Com esse arquivo em mãos, foi possível criar uma plataforma no software Vitis para o desenvolvimento do código C para rodar a rede convolucional.

Figura 2: Plataforma Microblaze através do arquivo.



Fonte: Elaborado pelo autor.

1.2 Rede CNN

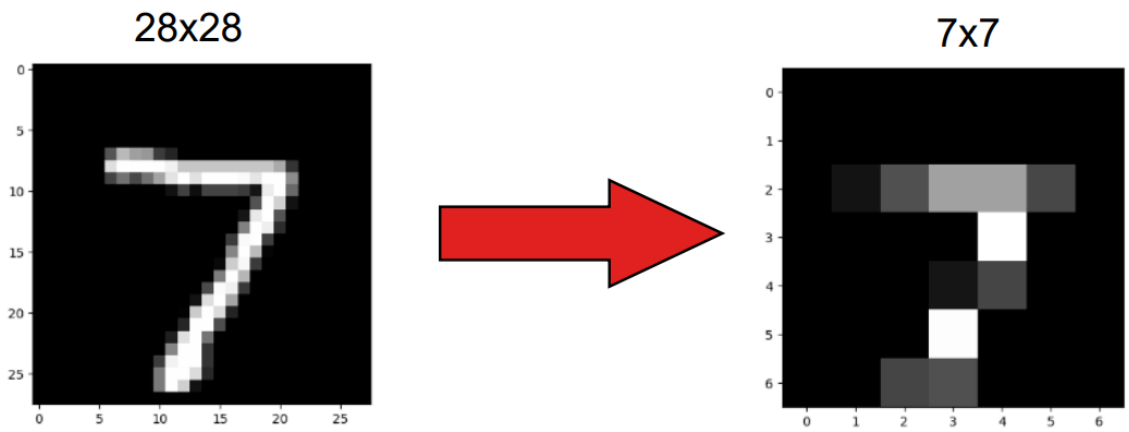
Um dos gargalos observados foi o tempo demorado da simulação. Ao sintetizar uma rede com várias camadas, o tempo para rodar a simulação completa RTL explodia muito rápido. Logo, para fins comparativos, optou-se por fazer o modelo mais básico de todos, composto por uma única camada CNN, seguida de um *max-pooling*, de um achatamento e da camada densa na saída para classificação.

```
1
2 model = tf.keras.models.Sequential([
3     tf.keras.layers.Conv2D(1, (3, 3), activation='relu', input_shape=(7,
      7, 1)),
4     tf.keras.layers.MaxPooling2D(2, 2),
5     tf.keras.layers.Flatten(),
6     tf.keras.layers.Dense(10, activation='softmax')
7 ])
```

Listing 1: Implementação da CNN básica.

Ademais, optou-se por comprimir o *dataset* para agilizar ainda mais o processo de simulação. Para tal, a imagem de entrada originalmente de 28x28 foi comprimida para 7x7.

Figura 3: Compressão das imagens do *dataset*.



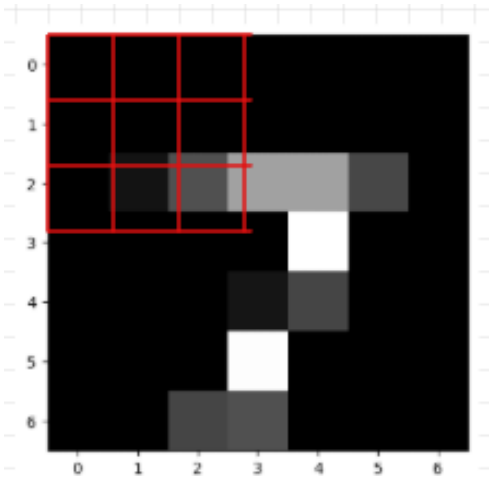
Fonte: Elaborado pelo autor.

Tendo em mãos a rede já treinada, um ponto crucial para a implementação da rede neural é realizar a análise das dimensões dos vetores de entrada e saída de cada camada, tendo em vista que precisamos definir esses elementos no código C.

1.2.1 Embarcando a Rede Neural em C

O processo de convolução, por si só, reduz as dimensões da imagem de entrada 7x7, tendo em vista que não foi realizado nenhum *padding*. Logo, com base na figura abaixo, ao analisar o centro do kernel 3 por 3 e ver por onde o kernel 'desliza' durante a convolução, é possível observar que a saída dessa camada terá 5x5 de tamanho.

Figura 4: Kernel em cima da imagem.



Fonte: Elaborado pelo autor.

```c
//A saida e o resultado da convolucao (5x5)
//A entrada e a imagem disposta na figura acima (7x7).
//O kernel possui tamanho (3x3)
//O peso da polarizacao e unitario (1)
void conv(float output[5][5], uint8_t input[7][7], float kernel[3][3], float bias) {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++)  {
            float acc = 0.0;

            for (int kernal_pos_x = 0; kernal_pos_x < 3; kernal_pos_x++) {
                for (int kernal_pos_y = 0; kernal_pos_y < 3; kernal_pos_y++) {
                    acc += input[i + kernal_pos_x][j + kernal_pos_y] * kernel[kernal_pos_x][kernal_pos_y];
                }
            }

            output[i][j] = acc + bias;
        }
    }
}
```

Listing 2: Camada Convolucional.

Em seguida, para minimizar ainda mais o tamanho da *feature map* gerada, a saída da camada convolucional foi passada para a entrada de *max pooling*, que tem como objetivo reduzir ainda mais o tamanho das sub-amostras geradas. Observe que nesta etapa, perde-se bastante informação da imagem inicial, pois nosso mapa de características é extremamente pequeno. Devido a esse max-pooling, o modelo apresentou uma grande dificuldade em dife-

renciar dígitos semelhantes (como 7 e 1; 8 e 0; 4 e 9). Se caso fosse necessário ter um modelo com uma alta acurácia, essa etapa teria que ser retirada (acabei mantendo-a porque aumenta a velocidade de simulação no Vivado, tendo em vista que diminui o número de operações em seguida).

```
//entrada 5x5, saida da primeira camada
//saida 2x2, reduzida pela metade, como 5 e impar, optei por reduzir para 2 e nao para 3.
void maxpool2d_5_to_2(float output[2][2], float input[5][5]) {
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            float max_values = input[i * 2][j * 2];
            if (input[i * 2][j * 2 + 1] > max_values)
            {
                max_values = input[i * 2][j * 2 + 1];
            }
            if (input[i * 2 + 1][j * 2] > max_values)
            {
                max_values = input[i * 2 + 1][j * 2];
            }
            if (input[i * 2 + 1][j * 2 + 1] > max_values)
            {
                max_values = input[i * 2 + 1][j * 2 + 1];
            }
            output[i][j] = max_values;
        }
    }
}
```

Listing 3: Max pooling.

Em seguida, essa *feature map* 2x2 é achatada para virar um vetor 1x4.

```
//entrada 2x2
//saida 1x4
void flatten(float output[4], float input[2][2]) {
    int ind = 0;

    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            output[ind++] = input[i][j];
        }
    }
}
```

Listing 4: Flatten.

E, por último, esse vetor de quatro posições resultante é passado como entrada na camada densa de classificação.

```
//entrada 1x4
//saida 10 [softmax]
void dense_result_final(float input[4], float output[10], float weights[4][10], float bias[10])
{
    for (int i = 0; i < 10; i++)
     {
        output[i] = bias[i];

        for (int j = 0; j < 4; j++)
        {
```

```
            output[i] += input[j] * weights[j][i];
        }
    }
}
```

Listing 5: Densa final.

Desse modo, no vetor de saída (output), obtém-se a inferência. Para saber a classe resultante, basta pegar o índice do maior valor da última camada.

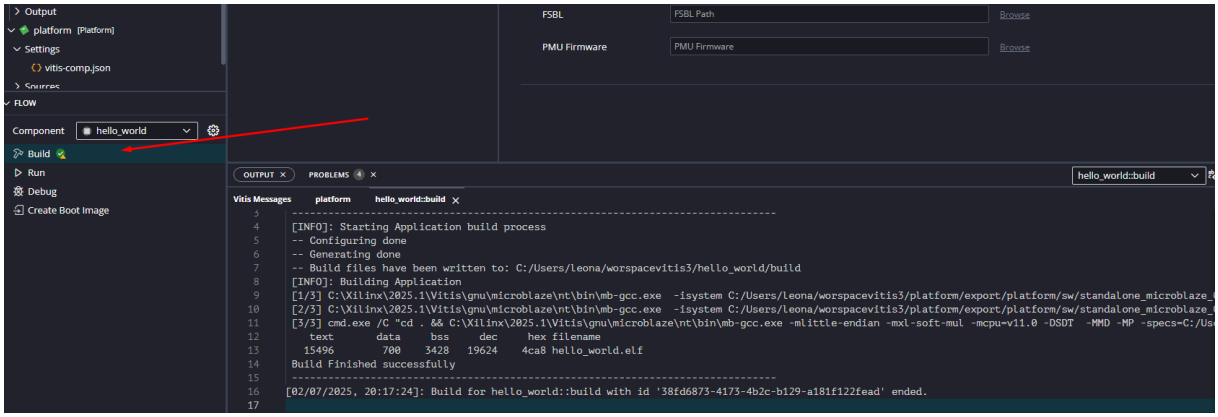
```
int main()
{
    init_platform();

    float conv_output[5][5];
    float pooled_min[2][2];
    float flatted[4];
    float output[10];

    uint8_t inf_result = 0;

    xil_printf("I"); //print para pegar o inicio
    conv(conv_output, test_image, conv1_weights, conv1_bias);
    maxpool2d_5_to_2(pooled_min, conv_output);
    flatten(flatted, pooled_min);
    dense_result_final(flatted, output, dense_weights, dense_bias);
    xil_printf("F"); //print para pegar o fim

    for (int i = 0; i < 10; i++) {
        if(output[i] > inf_result)
        {
            inf_result = i;
        }
    }

    //classe resultante
    xil_printf("%d", (int)(inf_result));

    cleanup_platform();
    return 0;
}

}
```

Listing 6: Fluxo principal.

1.2.2 Build do programa

Através do *template* padrão do *helloword*, a CNN foi implementada e o *.elf* foi gerado.

Figura 5: Build através da interface do Vitis



Fonte: Elaborado pelo autor.

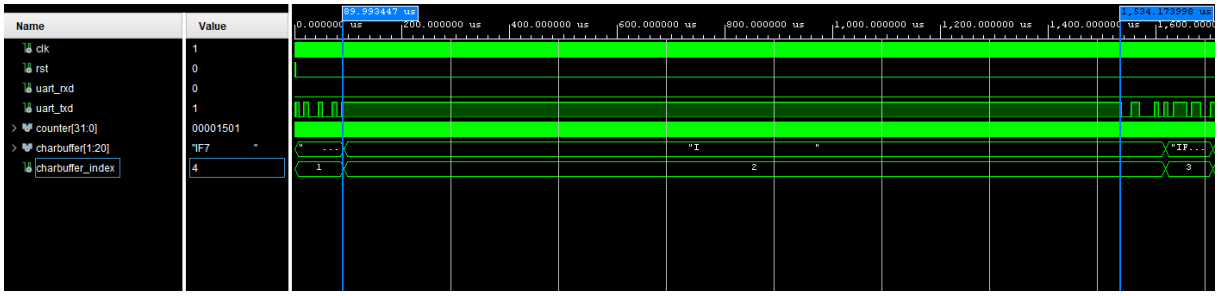
1.3 Simulação

Para gerar a simulação, o arquivo de testbench presente em `https://github.com/Neelam96/MicroBlaze-RTL-Simulation/tree/master/VIVADO` foi alterado para bater com os sinais presentes no design realizado na figura 1. Em seguida, associou-se à simulação o arquivo binário gerado pelo *Vitis*.

1.3.1 Sem FPU

O tempo para rodar a rede neural foi calculado através da diferença entre os *prints* dos *carácteres I* e *F* presentes no *listening* 6. Esse tempo de execução compreende o intervalo entre o último bit da uart_txd do primeiro caractere e o bit de início (start bit) do próximo caractere (ambos marcados em azul na figura 6).

Figura 6: Simulação - waveform sem FPU.



Fonte: Elaborado pelo autor.

Foram rodadas essa mesma rede para 5 imagens diferentes de entrada, incluindo aquela disposta na figura 3. A tabela abaixo resume os valores obtidos.

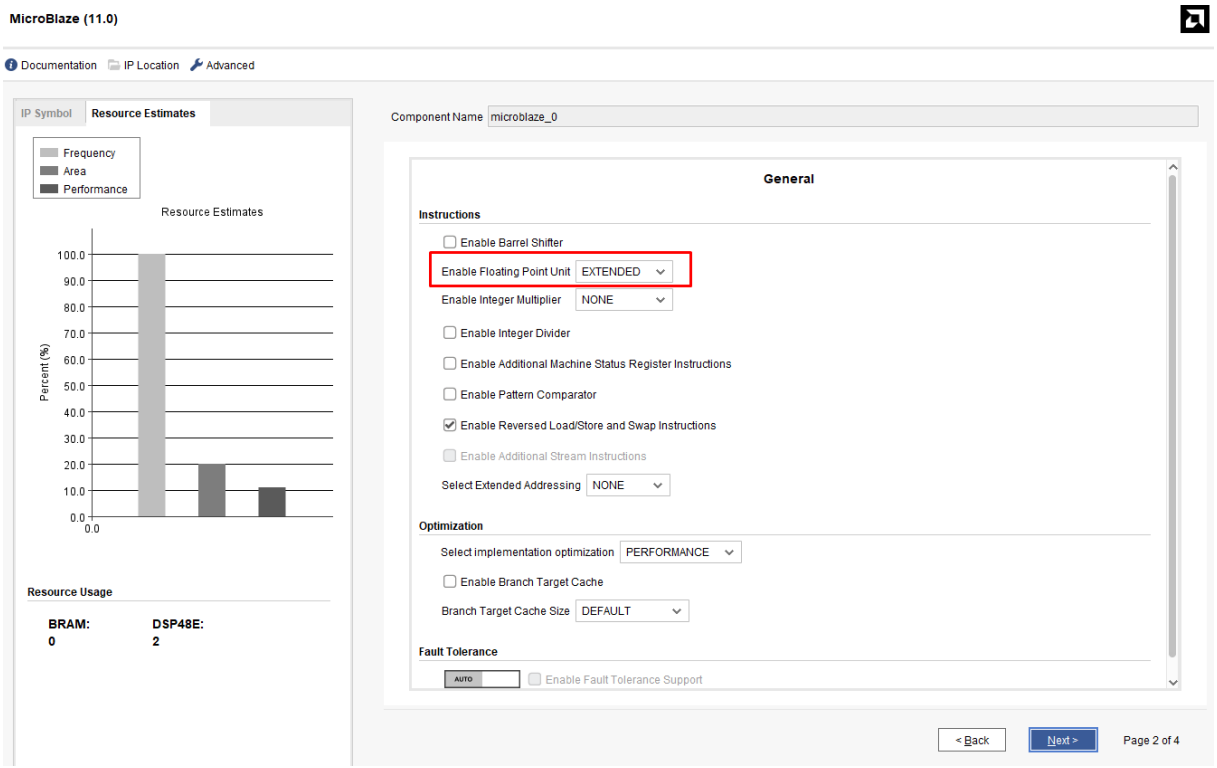
Tabela 1: Tempo de execução sem FPU

Amostra de teste	Inicio [us]	Fim [us]	Tempo demorado [us]
7	89.993	1534.17	1444.177
0	87.375	1410.85	1323.475
1	89.325	1602.900	1513.575
1	87.187	1691.662	1604.475
6	89.375	1591.662	1502.287

1.4 Com FPU

Para habilitar a FPU no Vivado, é necessário acessar a Figura 1 novamente e ativar a funcionalidade diretamente no IP do Microblaze.

Figura 7: Implementando FPU.



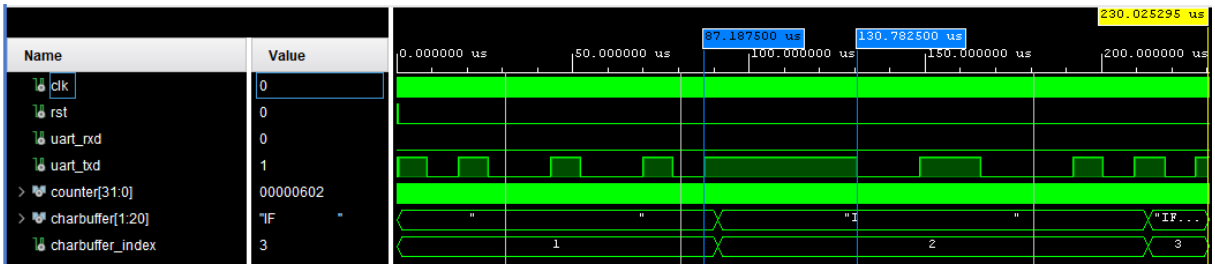
Fonte: Elaborado pelo autor.

Novamente, o arquivo XSA foi gerado junto com o bitstream. Em seguida, utilizando o Vitis, uma nova plataforma foi criada com base nesse arquivo. O interessante é que o Vitis reconhece que o novo hardware contém uma unidade de ponto flutuante (FPU) e já adiciona a flag -mhard-float na compilação do código C por padrão.

Ao introduzir a mesma imagem de entrada mostrada na figura 6, obteve-se o resultado disposto pela figura 8. Observe que a simulação com FPU se tornou tão eficiente que fica difícil medi-la pelos bits da serial UART, uma vez que o tempo de processamento necessário

para rodar a rede é parecido com o tempo do inverso do baud rate da comunicação.

Figura 8: Simulação - waveform com FPU.



Fonte: Elaborado pelo autor.

Resumindo, ao adicionar a *FPU*, o tempo de executar a rede passou para a ordem de alguns microssegundos. Para efeitos comparativos, a tabela 1 foi refeita utilizando FPU.

Tabela 2: Tempo de execução com FPU

Amostra de teste	Inicio [us]	Fim [us]	Tempo demorado [us]
7	87.187	130.782	43.595
0	87.187	139.502	52.315
1	87.187	130.782	43.595
1	87.187	130.782	43.595
6	87.187	130.782	43.595

2 Conclusão

Com a integração da FPU, a operação demonstrou uma diminuição percentual de aproximadamente 96,16% no tempo de execução, o que corresponde a um fator de aceleração de cerca de 26,07 vezes. Por fim, pode-se concluir que a presença da FPU introduz um impacto notório na velocidade de execução das operações em ponto flutuante. Isso ocorre porque a CPU não precisa emulá-las via software, resultando em um ganho significativo de performance.

3 Outras tentativas

Tentei rodar uma rede mais pesada através da biblioteca do *tensorflowlite*. Entretanto, tive problemas de dependências com a biblioteca em questão e não consegui compilar o programa, mesmo depois de muitas tentativas. Tendo em vista os problemas que tive no meio

do caminho, optei apenas por implementar a CNN no *softcore* e realizar a comparação com e sem FPU.

4 Anexo

O colab segue disposto nesse link: `https://colab.research.google.com/drive/1R1ILZI2Ssmk_ajNxlPJYYAoyPpiplk1_?usp=sharing`