Relatório do trabalho final de SSC0740

Allan Garcia Cavalcante e Silva - 13731222
Eduarda Fritzen Neumann - 12556973

4 de julho de 2024

1 Introdução

Projetou-se e treinou-se uma rede neural convolucional para classificação de imagens do *dataset* MNIST, que representam dígitos escritos a mão. Configurou-se um processador NIOS II no ambiente *Platform Designer* para execução dessa rede durante a classificação de imagens do *dataset* de teste do MNIST. Houveram tentativas de tornar a classificação *multi-thread* e instanciar múltiplos processador NIOS II para a execução dessa, porém não houve sucesso.

Os códigos discutidos nesse relatório podem ser encontrados nesse repositório do GitHub.

2 Rede neural convolucional

2.1 Arquitetura

O *dataset* MNIST é composto por imagens quadradas de 28 *pixels* em escala de cinza, logo a entrada da rede possui dimensões $28 \times 28 \times 1$. Cada imagem representa um dígito, logo existem 10 possíveis categorias, de modo que a saída da rede possui dimensão 10×1, onde cada elemento representa a probabilidade de que a imagem de entrada corresponda ao dígito daquele índice.

Definiu-se 3 camadas convolucionais, todas com 1 filtro de *kernel* de dimensão 3×3. Ou seja, a primeira camada intermediária tem dimensão 26×26, a segunda 24×24 e a terceira 22×22. Essa última é então achatada para dimensão 484×1. Após, uma camada *fully-connected* de dimensão 484×10 é aplicada para gerar a saída da rede.

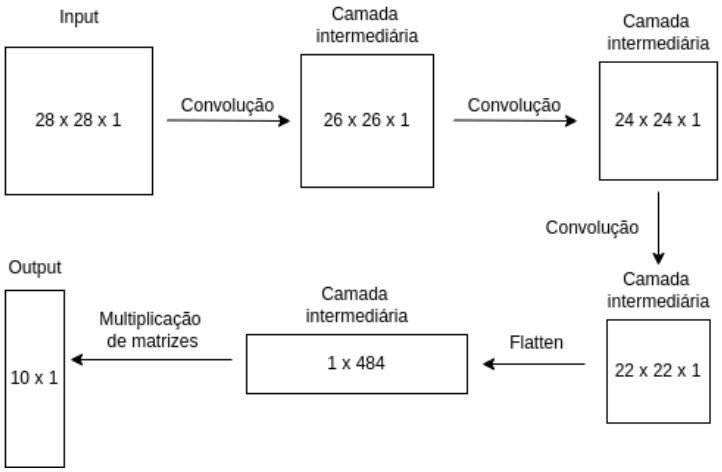
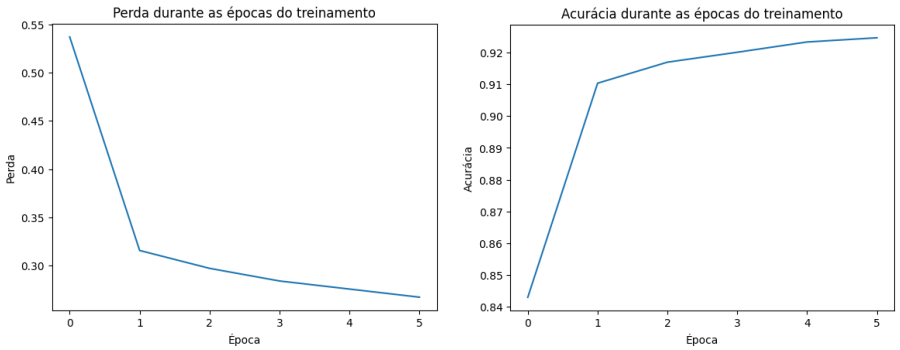


Figura 1: Diagrama da arquitetura da rede neural convolucional

A função de ativação das camadas convolucionais é ReLU (*rectified linear unit*) e da camada *fully-connected* é *softmax*. Desse modo, a soma dos valores da camada de saída é 1, garantido que esses elementos possam ser interpretados como porcentagens e que a rede seja treinada para que uma porcentagem maior indique mais certeza que a imagem representa o dígito correspondente.

2.2 Treinamento

Utilizou-se a biblioteca TensorFlow para treinamento da rede, com as 1875 imagens de treinamento do *dataset* durante 6 épocas. Ao final dessas, a rede treinada obteve uma acurácia de aproximadamente 93% nas imagens de teste.



(a) Perda (b) Acurácia

Figura 2: Métricas durante as épocas do treinamento

3 Execução da rede em C

Foi codificada uma aplicação na linguagem C para classificação das imagens utilizando a rede treinada, de modo que o programa compilado possa ser executado no processador NIOS II.

3.1 Importação dos pesos

Os pesos dos kernels das camadas convolucionais e da camada fully-connected, obtidos utilizando Python, além das imagens com valores de cinza entre 0 e 1, foram exportados para um arquivo de texto, que foi incluído no programa em C. Desse modo, as entradas da rede (as imagens do dataset MNIST), a saída, as camadas intermediárias e os pesos foram declarados estaticamente como arrays de float, e seus valores estão presentes no arquivo compilado, ou seja, "hard coded". O exemplo abaixo apresenta os pesos do kernel da primeira camada convolucional no arquivo de texto exportado em Python.

```
#define KERNEL_SIZE_1 3
float first_convolution[KERNEL_SIZE_1][KERNEL_SIZE_1];
first_convolution[0][0]=-0.06962999701499939;
first_convolution[0][1]=0.5331646800041199;
first_convolution[0][2]=-0.199904203414917;
first_convolution[1][0]=0.3026471436023712;
first_convolution[1][1]=0.5846962928771973;
first_convolution[1][2]=0.3332327604293823;
first_convolution[2][0]=-0.17035962641239166;
first_convolution[2][1]=0.05097603052854538;
first_convolution[2][2]=0.2782385051250458;
```

Abaixo apresenta-se a função principal do código em C e a inclusão dos arquivos de texto citados. Nesse exemplo, a imagem a ser classificada é a imagem do arquivo `image0.txt`, que contém a primeira imagem do dataset de teste.

```
int main()
{
    #include "first_convolution.txt"
    #include "second_convolution.txt"
    #include "third_convolution.txt"
    #include "fully_connected.txt"
    #include "image0.txt"

    int layer1_size = IMAGE_SIZE - KERNEL_SIZE_1 + 1;
    float layer1[layer1_size][layer1_size];

    int layer2_size = IMAGE_SIZE - KERNEL_SIZE_1 - KERNEL_SIZE_2 + 2;
    float layer2[layer2_size][layer2_size];

    int layer3_size = IMAGE_SIZE - KERNEL_SIZE_1 - KERNEL_SIZE_2 - KERNEL_SIZE_3 + 3;
    float layer3[layer3_size][layer3_size];

    int flat_size = layer3_size*layer3_size;
    float flat[flat_size];
```

```
    float output[OUTPUT_SIZE];

    convolution(layer1_size, &layer1[0][0], IMAGE_SIZE, &image[0][0],
                KERNEL_SIZE_1, &first_convolution[0][0]);
    convolution(layer2_size, &layer2[0][0], layer1_size, &layer1[0][0],
                KERNEL_SIZE_2, &second_convolution[0][0]);
    convolution(layer3_size, &layer3[0][0], layer2_size, &layer2[0][0],
                KERNEL_SIZE_3, &third_convolution[0][0]);
    multiply(OUTPUT_SIZE, &output[0], flat_size, &layer3[0][0],
            &fully_connected[0][0]);

    float max = output[0];
    int prediction = 0;
    for (int i = 1; i < OUTPUT_SIZE; i++)
    {
        printf("%f ", output[i]);
        if(output[i] > max){
            max = output[i];
            prediction = i;
        }
    }
    printf("\n%d\n", prediction);

    return 0;
}
```

3.2 Convolução

A função responsável por executar a convolução é apresentada abaixo. Como parâmetros, recebe ponteiros para os arrays das camadas de entrada, saída e dos pesos do kernel, além de seus tamanhos.

```
void convolution(int output_size, float *output, int input_size, float *input,
                int kernel_size, float *kernel)
{
    for (int i = 0; i < output_size; i++)
    {
        for (int j = 0; j < output_size; j++)
        {
            float result = 0;
            for (int ki = 0; ki < kernel_size; ki++)
            {
                for (int kj = 0; kj < kernel_size; kj++)
                {
                    result += kernel[ki*kernel_size + kj] *
                              input[(i + ki)*input_size + j + kj];
                }
            }
            *(output + i*output_size + j) = result >= 0 ? result : 0;
        }
    }
}
```

3.3 Multiplicação de matrizes

A multiplicação de matrize por vetor, usada na camada fully-connected, é apresentada abaixo. Como na operação de convolução, a função recebe os ponteiros dos arrays das camadas de entrada. saída e dos pesos, além dos tamanhos de entrada e saída, que determinam também o tamanho do array de pesos.

Devida à manipulação dos índices feita nessa função, não é necessária a implementação de uma função para realizar o achatamento da camada de entrada, de dimensão 22×22, para 484×1.

```
void multiply(int output_size, float *output, int input_size, float *input,
              float *weights)
{
    for (int i = 0; i < output_size; i++)
    {
        float result = 0;
        for (int j = 0; j < input_size; j++)
        {
            result += weights[i + j*output_size] * input[j];
        }
        output[i] = result;
    }

    // Ativação softmax
    float sum = 0;
    for(int i=0; i<output_size; i++){
        output[i] = expf(output[i]);
        sum += output[i];
    }
    for(int i=0; i<output_size; i++){
        output[i] /= sum;
    }
}
```

4 Tentativa de pipeline

Houve a tentativa de paralelizar o código, formando um *pipeline* com processadores NIOS II, onde cada etapa do *pipeline* correspondendo ao cálculo de uma camada da CNN. Desse modo, o *throughput* é uma classificação por tempo de processamento da camada mais lenta, após a primeira classificação.

Para isso, cada camada possuiria um *buffer* interno, de mesmo tamanho da seu saída, onde escreveria sua saída enquanto a convolução ou a multiplicação de matrizes, a depender da camada, está em execução. Quando o processamento da camada for finalizado, esse resultado será então copiado para a entrada da próxima camada, dado que essa próxima camada já processou os dados presentes nessa região de memória, para a próxima etapa do pipeline. Quando a cópia estiver completa, ambas as camadas podem voltar ao seu processamento, dado que a camada anterior tem dados novos para processar e o buffer da camada posterior está disponível, ou seja, não está em processo de cópia.

Nesse sistema, cada camada pode ser processada por um processador diferente, sendo 3 responsáveis pelas camadas de convolução e 1 pela camada de

multiplicação.

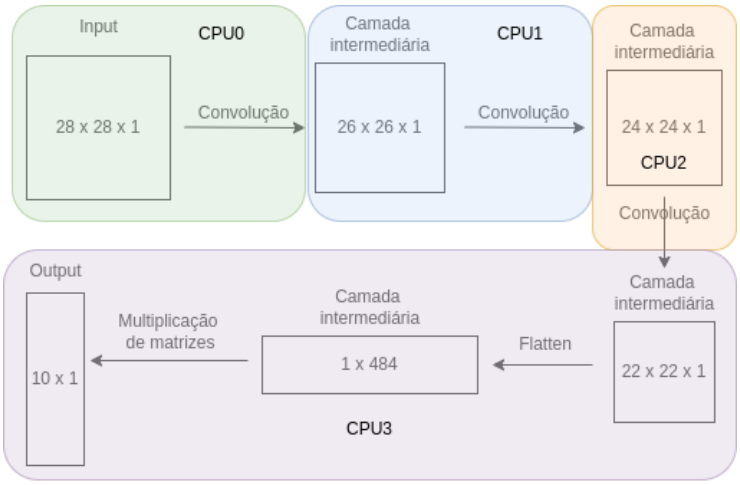


Figura 3: Diagrama da arquitetura da CNN paralelizada

Antes de passar para o processador, foi tentado fazer uma versão inicial em C++, rodando no Linux Desktop normalmente. Cada processador seria representado por uma thread nesse modelo, e os espaços de memória compartilhada seriam representados por variáveis globais acessíveis por todas as threads. Durante a implementação, tivemos problemas devido a deadlocks: a layer i ficava esperando que a layer $i+1$ copiasse a entrada para seu buffer interno, enquanto a layer $i+1$ esperava que a layer i enviasse os dados. Isso resultou em um impasse, impedindo a execução adequada do pipeline.

A tentativa em softcore com Nios II no Platform Designer foi feita utilizando uma memória compartilhada do tipo RAM para os 4 CPUs. A ideia inicial era limitar uma parte do endereçamento para cada CPU e disponibilizar uma outra seção para memória compartilhada. No entanto, durante a execução do código no Eclipse com múltiplas CPUs, surgiram problemas críticos: quando qualquer coisa era impressa via UART, a saída não funcionava corretamente, inviabilizando o desenvolvimento futuro.

Na imagem 4, observa-se a tentativa de projeto realizada no Platform Designer. Nota-se a presença das 4 CPUs e da memória compartilhada, bem como o assignement de regiões de memórias distintas para cada processador.

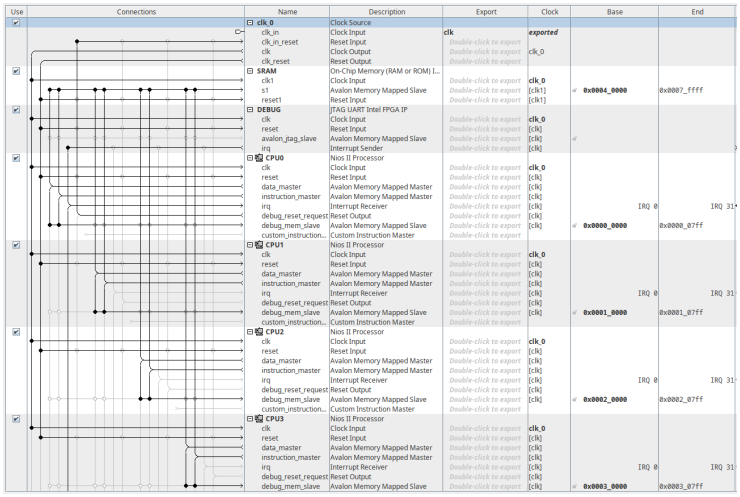


Figura 4: Conexões no Platorm Designer para modelo Multicore com Pipeline

5 Execução Single-Core

Após a tentativa falha de execução do código multicore, optou-se pela abordagem mais simples da execução em um único processador Nios 2. O código é simples, e tem seu funcionamento descrito com detalhes na seção 3.

Podemos citar que a configuração no Platform Designer, visível na imagem 5, mostrou-se simples, envolvendo apenas um processador softcore Nios 2, com uma memória RAM On-Chip de 256K e um dispositivo para permitir o uso do JTAG/USB. Não há uma razão específica para esse tamanho de memória, apenas buscou-se algo capaz de suportar o código com folga, sem ultrapassar a capacidade lógica da placa DE10.

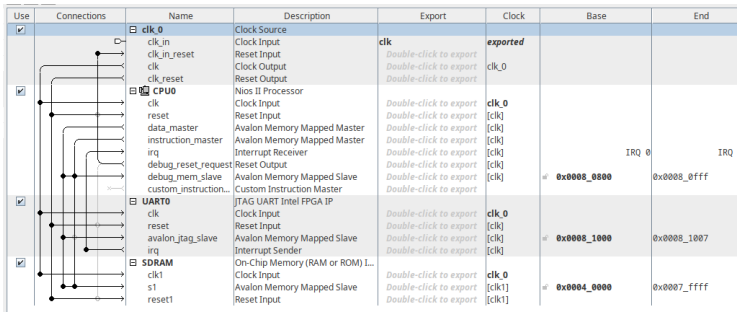


Figura 5: Conexões no Platorm Designer para modelo Single-Core com Pipeline