Co-projeto de Hardware e Software - 2020/2021

PL1

Rafael Alves Vieira - 2017257239

22 de janeiro de 2022

Classificador MNIST

UNIVERSIDADE D
COIMBRA

INTRODUÇÃO

Este relatório descreve os resultados/desenvolvimento da implementação de um classificador para o dataset MNIST[7] numa FPGA Cyclone IV, sendo o processamento primeiramente realizado dentro do processador embebido NIOS II [6], e posteriormente com uma aceleração de hardware.

Inspiração

Decidi o tema do projeto por ter visto uma implementação de uma CNN também na DE2-115 no github[3], que usa uma câmara como input das imagens, e inicialmente a intenção foi a minha versão desta implementação. Acabei por não implementar uma CNN, e usei uma rede mais simples, e também não cheguei a usar a câmara.

DETALHES SOBRE A REDE NEURONAL

A rede neuronal usada para o treino foi a mais simples possível, com o intuito de criar poucos pesos já que os recursos de memória da FPGA são limitados. O código para o treino e para a classificação está escrito em C, usando como exemplo um antigo tutorial que costumava de estar no website *tensorflow*[5], tendo sido removido. No entanto, código ainda se encontra presente no github[4]. Há várias implementações de CNNs e outros tipos de rede mas a que foi usada neste trabalho apenas tem a *input layer* com 784 neurónios (28*28 pixeis) e uma output layer com 10 (número de classes) neurónios, não tendo quaisquer *hidden layers*. Isto cria $784 * 10 = 7840$ pesos e 10 *bias* que depois são declarados localmente para usar no classificador.

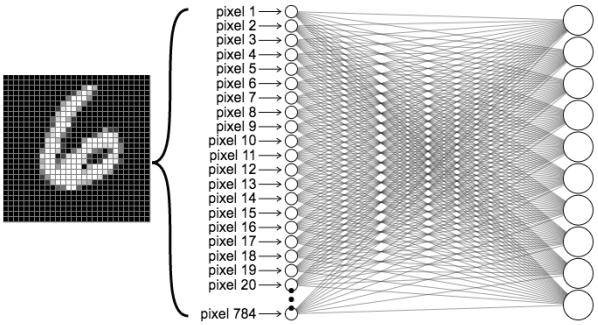


Figura 1. Esquema da arquitetura da rede.[2]

Figura 2. Resultados da evolução do treino.[4]

Funcionamento do classificador

Para decidir qual é a classe de uma imagem precisamos de calcular os resultados da activation function. Para isso temos um array de activations (dimensão 10) que tem em cada elemento o valor da activation para cada classe (isto feito para cada imagem). A classe que tiver maior valor de ativação é escolhida como a classe atribuída à imagem pelo classificador. O array das ativações é inicializado com a bias de cada classe. O calculo da activation para uma classe é a soma acumulada do produto do valor do peso de uma determinada classe e pixel pelo valor da intensidade do pixel:

$$activation[i] = \sum_{j=0}^{j=784} weights[i][j] * pixel\_value[j], \text{ onde i = classe}$$

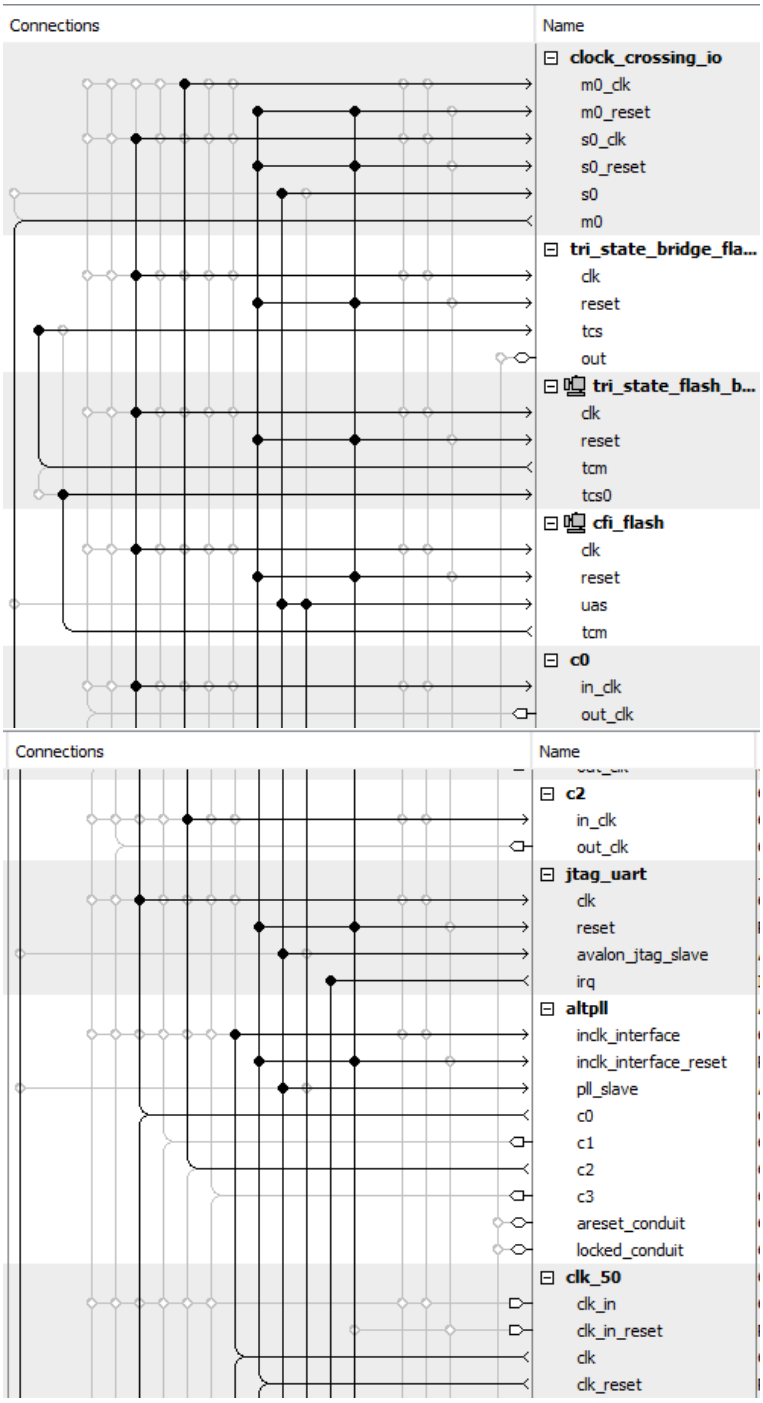
Depois ainda é calculada a softmax de cada activation (ou seja, a activation function é a softmax).

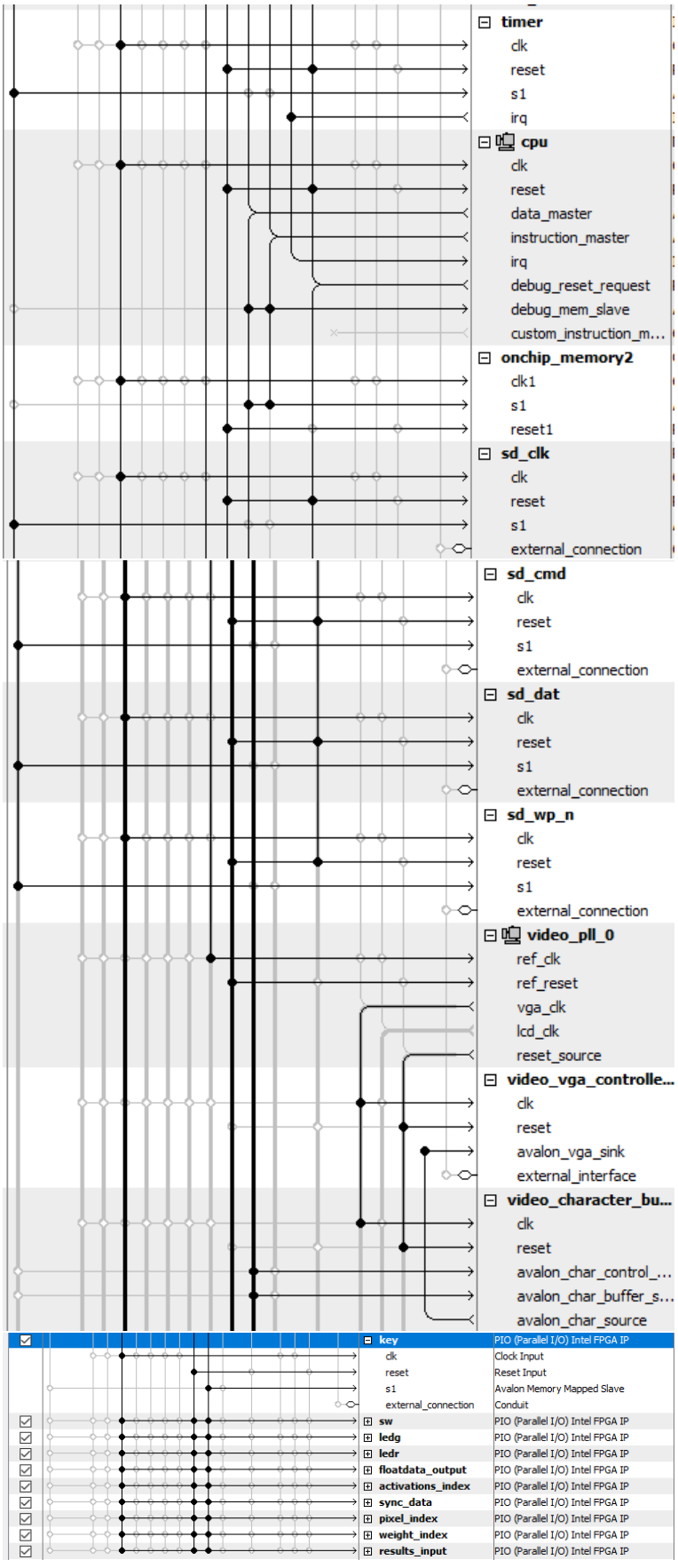
IMPLEMENTAÇAO DO CLASSIFICADOR NO NIOS II

A interface para o utilizador é feita usando os switches e botões da FPGA e um ecrã VGA. As imagens do dataset são armazenadas num cartão SD.

Para a leitura das imagens do cartão SD foi usado como template a demonstração presente no manual de utilizador da FPGA DE2-115[1], onde o código-fonte se encontra presente no CD que vem com a placa. Recorri a este método porque na versão atual do quartus(18.1) não existe nada das bibliotecas do *University Program* para o cartão SD.

Após algumas adaptações, obtive o seguinte design:




timer
clk
reset
s1
irq
cpu
clk
reset
data_master
instruction_master
irq
debug_reset_request
debug_mem_slave
custom_instruction_m...
onchip_memory2
clk1
s1
reset1
sd_clk
clk
reset
s1
external_connection
sd_cmd
clk
reset
s1
external_connection
sd_dat
clk
reset
s1
external_connection
sd_wp_n
clk
reset
s1
external_connection
video_pll_0
ref_clk
ref_reset
vga_clk
lcd_clk
reset_source
video_vga_controlle...
clk
reset
avalon_vga_sink
external_interface
video_character_bu...
clk
reset
avalon_char_control_...
avalon_char_buffer_s...
avalon_char_source
key
PIO (Parallel I/O) Intel FPGA IP
clk
Clock Input
reset
Reset Input
s1
Avalon Memory Mapped Slave
external_connection
Conduit
sw
PIO (Parallel I/O) Intel FPGA IP
ledg
PIO (Parallel I/O) Intel FPGA IP
ledr
PIO (Parallel I/O) Intel FPGA IP
floatdata_output
PIO (Parallel I/O) Intel FPGA IP
activations_index
PIO (Parallel I/O) Intel FPGA IP
sync_data
PIO (Parallel I/O) Intel FPGA IP
pixel_index
PIO (Parallel I/O) Intel FPGA IP
weight_index
PIO (Parallel I/O) Intel FPGA IP
results_input
PIO (Parallel I/O) Intel FPGA IP

Detalhes acerca dos componentes (número de bits, endereços de memória) estão presentes na datasheet que vai junto com o relatório.

Para o envio de dados e a sincronização entre o *Hard Processor System* e o RTL existem as entradas/saídas:

- floatdata_output: saída de 32 bits usada para enviar dados para o RTL;
- activations_index: saída de 4 bits usada para indicar o endereço da memória ram das ativações;
- sync_data: saída de 16 bits (no entanto não são todos usados, deixei assim para haver flexibilidade em adaptações futuras e não ter de perder tempo a gerar um novo design) usada para sincronizar o HPS e o RTL;
- pixel_index: saída de 10 bits para o endereço da memória ram dos pixeis;
- weight_index: saída de 13 bits para o endereço da memória ram dos pesos;
- results_input: input de 32 bits para ler o resultado das operações acumulado na ram que tem os valores das ativações.

Exitem 4 versões do classificador:

- versão 1: implementada apenas em C sem aceleração de hardware;
- versão 2: cujo o cálculo das ativações é feito no RTL;
- versão 3: onde existe paralelismo entre RTL e a cpu;
- versão 4: onde podemos avançar as samples manualmente, observar a imagem a ser classificada, o tempo de execução, classe atribuida pelo classificador e a verdadeira classe. (esta versão é uma adaptação da segunda versão, ou seja os cálculos são realizados no RTL).

Os pesos são declarados localmente da seguinte forma:

```
const neural_network_t network =
{       //bias
        {-0.343634,0.732933,0.557800,0.275675,0.468390,1.817463,0.463807,0.824511,-1.144381,0.292256},

        {//weights
                {0.563585,0.193304,0.808740,0.585009,0.479873,0.350291,0.895962,0.822840,0.746605,0.174108,0
                {0.018555,0.978423,0.647633,0.053652,0.914151,0.637471,0.431715,0.021058,0.242592,0.391949,0
                {0.699240,0.248604,0.936705,0.277444,0.299539,0.822443,0.819636,0.629902,0.795068,0.007202,0
                {0.271310,0.510178,0.202643,0.875973,0.169866,0.149022,0.722098,0.691519,0.556810,0.622150,0
                {0.735466,0.712058,0.581195,0.986694,0.002289,0.946196,0.581378,0.458571,0.429273,0.358226,0
                {0.212592,0.475509,0.248543,0.983917,0.895230,0.466811,0.362438,0.763787,0.036775,0.061586,0
                {0.089785,0.518937,0.934568,0.453261,0.476852,0.537461,0.849696,0.266243,0.950621,0.614490,0
                {0.074221,0.556261,0.920438,0.492538,0.996918,0.678640,0.978057,0.758324,0.744774,0.215094,0
                {0.037446,0.137761,0.771813,0.623066,0.634816,0.925901,0.504898,0.617573,0.610736,0.496658,0
                {0.612384,0.574572,0.292459,0.119938,0.810724,0.368725,0.853969,0.269509,0.861690,0.710776,0
        }
};
```

(Foram previamente escritos para um ficheiro após realizar o treino no meu laptop.)

Onde a struct *neural_network_t* é:

```
typedef struct neural_network_t_ {
    float b[10];
    float W[10][784];
} neural_network_t;
```

Dentro das funções para cada versão do classificador estão os cálculos dos valores das *activations* para cada classe, estes cálculos são depois discutidos em mais detalhe na secção *"Implementação de um acelerador de Hardware"*.

Interface

A interface com o utilizador é feita usando switches, botões e o ecrã vga.

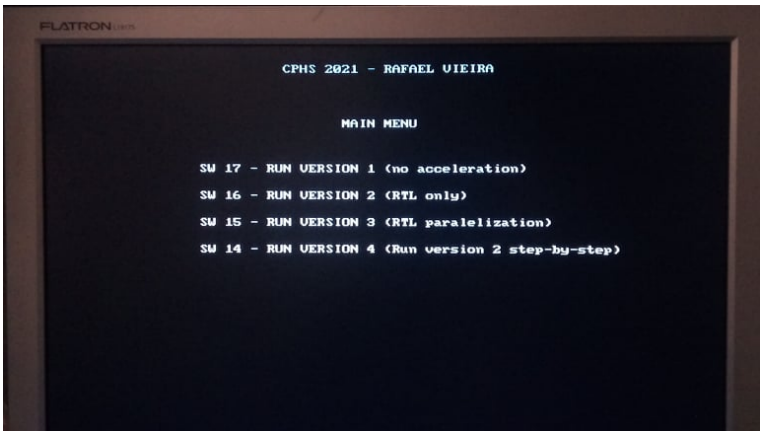


Figura 3. Menu inicial onde o utilizador escolhe entre as diferentes versões do classificador.

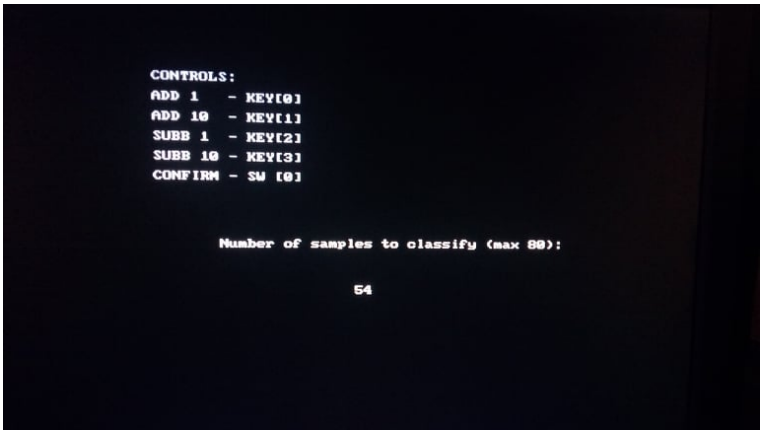


Figura 4. Caso escolha uma das primeiras três versões, seguidamente tem de indicar quantas imagens quer classificar.

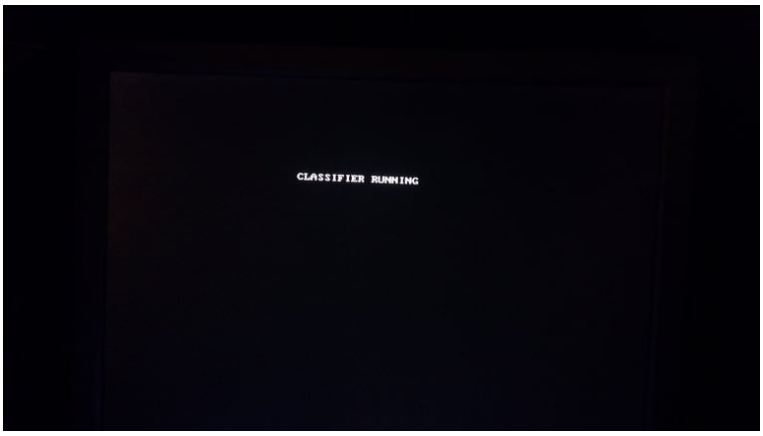


Figura 5. Depois de escolher as imagens, existe este *loading screen*

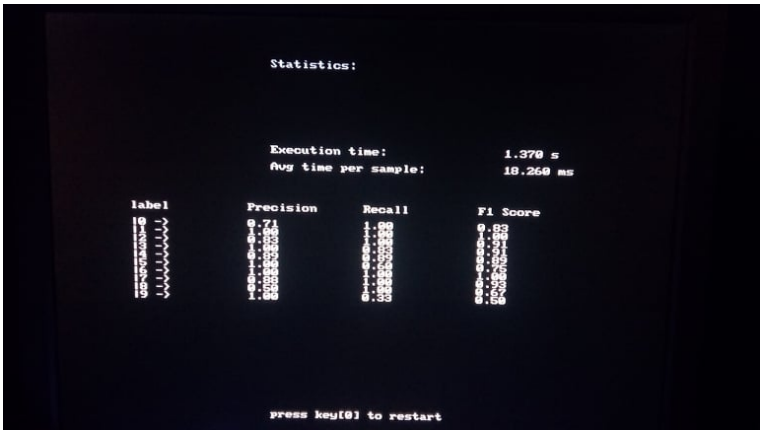


Figura 6. Por fim, os F1 scores e o tempo de execução. Se primir a KEY[0] volta ao menu inicial.

Figura 7. Caso tenha escolhido a versão 4 primeiro encontra este menu. Para avançar entre as samples é necessário primir a KEY[0].

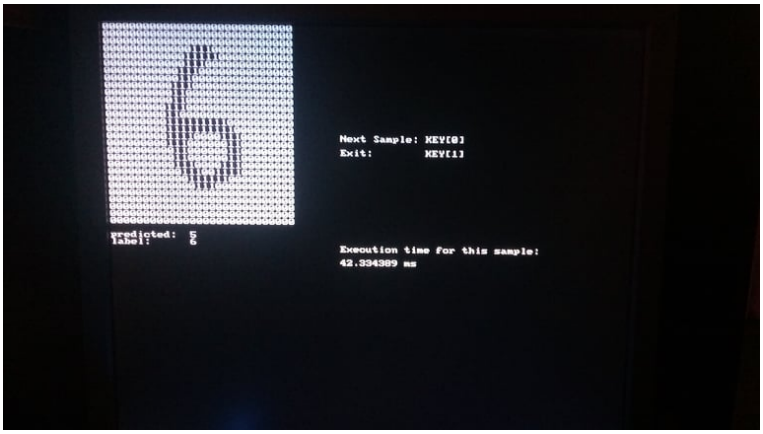


Figura 8. Aqui (dentro da versão 4) podemos observar as imagens, e os detalhes referentes à classificação. Quando já tiver acabado de analisar as samples uma a uma, pode primir a KEY[1] e terá também uma tabela de métricas como foi mostrado para as outras versões.

LEITURA DAS IMAGENS E CLASSES

As imagens a serem classificadas são de pequenas dimensões (28*28 pixeis) e estão armazenadas num cartão SD.

O dataset original foi alterado e adaptado para este caso de aplicação, de modo a leitura ser mais rápida: o header foi removido e invez de ter um ficheiro em separado com a informação da label (classe), essa informação foi concatenada ao ficheiro da imagem, passando a ser o último byte do ficheiro.

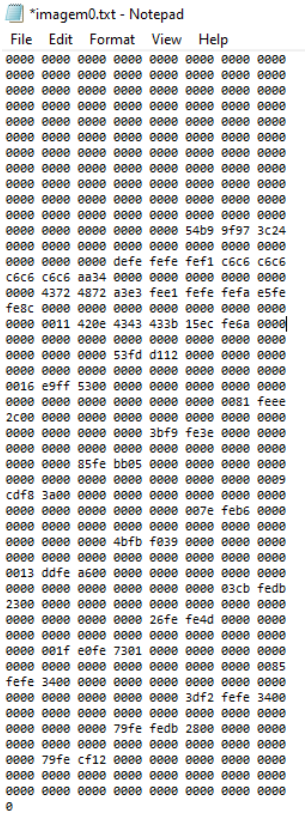


Figura 9. Exemplo de imagem armazenada dentro do cartão SD

A leitura das imagens é um processo lento e é o que ocupa mais tempo de processamento, mas não interpretei esta parte do trabalho como algo que fosse para paralelizar, e por isso mantive esta parte fora da anotação dos tempos de execução.

Por ter feito uso das bibliotecas disponibilizadas pelo exemplo de uso do cartão SD, notei que havia menos problemas quando a leitura dos bytes era feita em documentos de texto (o formato das imagens do dataset é invulgar, e a sua leitura realizada do modo tradicional estava a funcionar apenas no laptop, mas quando lido do cartão SD não funciona), mas por ter armazenado em documentos de texto a informação das imagens, tenho de ter em conta os caracteres dos espaços e os de mudança de linha quando quero copiar a informação dos bytes.

A função *readimage* é a função usada para leitura de imagens. Para não alocar muita memória de uma só vez, aloco uma linha de cada vez da imagem (48 caracteres), e ao longo de um ciclo vou filtrando a informação dos bytes copiando-a para um array (array dos pixeis). A função por fim devolve a label, que é o último byte da imagem, neste caso a imagem é o número 0.

Problemas de implementação

Inicialmente o objetivo seria fazer uso de centenas de imagens e anotar métricas temporais para vários números de imagens classificadas, comparando entre versões do classificador. Mas por problemas que devem estar associados com as bibliotecas para o cartão SD ou limitações de hardware apenas consigo usar 80 imagens. As próprias funções da biblioteca que são usadas para listar os ficheiros do cartão SD não conseguem listar a partir de 80 imagens adiante. Mas como existem muitas operações com ponteiros manuseamento de memória nesta etapa, pode também ser algo vindo da minha implementação. No entanto, tenho sempre cuidado para não alocar mais do que uma imagem de cada vez, e liberto sempre a memória que aloco.

A versão que tem o acelarador de hardware paralelizado em quase todas as situações tem o mesmo speedup, visto que, como se vê na secção seguinte, as iterações do ciclo são divididas em 2, e metade são executadas pela cpu, e outra metade pelo RTL, e fazemos sempre um número de transferências de memória igual ao número de imagens classificadas (pouco eficiente). Portanto também não traria grande proveito de executar o classificador para grandes números de imagens, a conclusão tirada seria a mesma.

IMPLEMENTAÇÃO DE UM ACELERADOR DE HARDWARE

Diagrama do RTL

A cada imagem mostrada do diagrama será feita um *zoom-in* da parte que está a ser descrita.

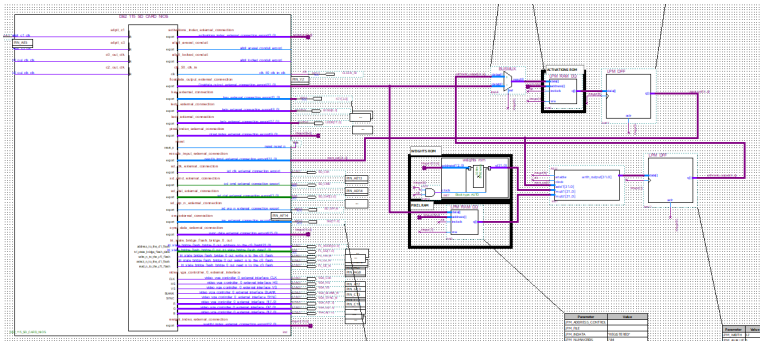


Figura 10. Para implementar um acelerador de hardware, um RTL que calcula uma acumulação de somas e multiplicações é controlado pelo NIOS.

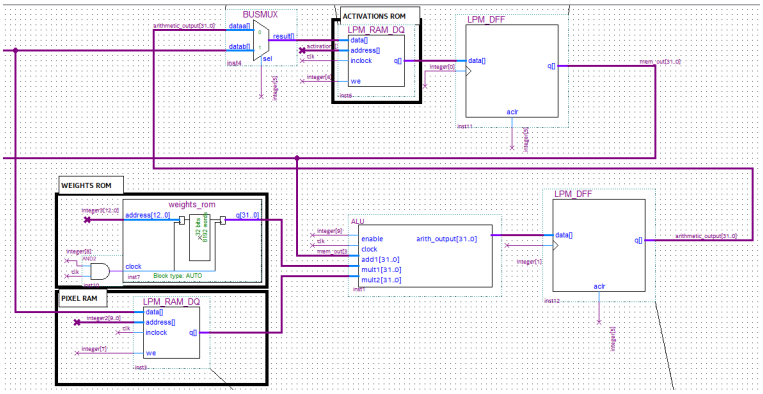


Figura 11. Para isso existem duas RAMs: uma que recebe informação dos pixeis sempre que existe uma nova imagem, e outra que acumula os valores das ativações para cada classe. Podemos reparar que existem flip-flops à saída da memória RAM e da ALU. Isto serve para sincronizar quando quero escrever na RAM o valor calculado durante a iteração do ciclo, e quando quero enviar para a ALU os valores atualizados da activation. Se não fizesse isto, como existem inúmeros ciclos de relógio dentro de uma iteração do ciclo for, estaria a calcular mais do que uma soma e multiplicação por ciclo, e isto seria escrito na memória das ativações. O relógio destes flip-flops é uma saída controlada pela CPU, que pulsa entre 0 e 1 cada vez a iteração do ciclo for é incrementada.

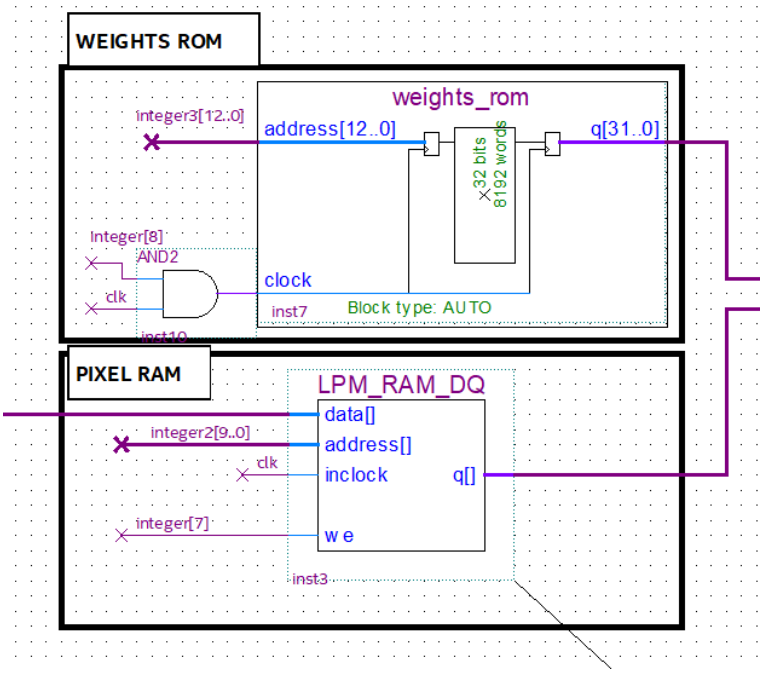


Figura 12. Há também uma ROM inicializada com um ficheiro MIF que contém os valores dos pesos. Os acessos aos endereços destas memórias são feitos através outputs da CPU (são as iterações do ciclo que está a ser acelerado).

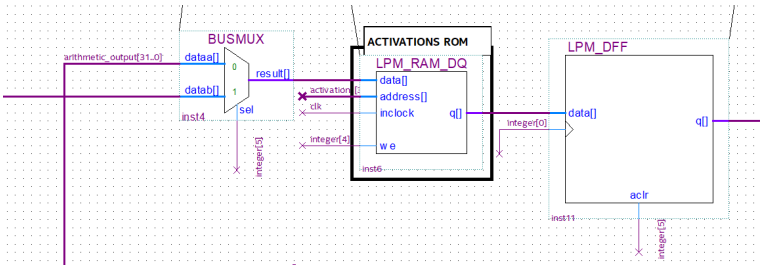


Figura 13. O multiplexer à entrada da RAM das ativações serve para escolher entre um output da ALU (para acumular a soma de valores) e o output da CPU (para inicializar com os valores da bias).

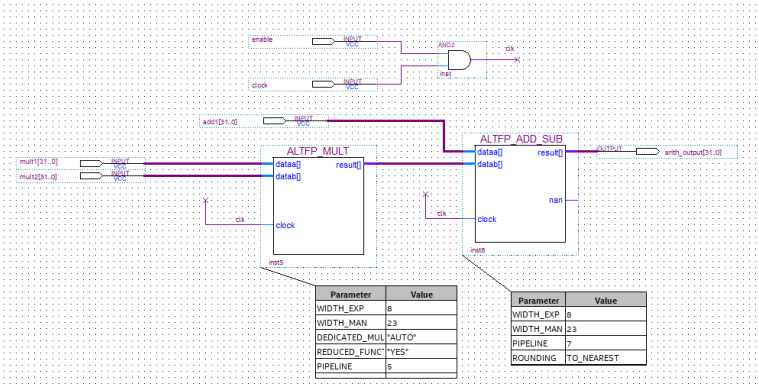


Figura 14. Este é o diagrama da ALU composto apenas por um multiplicador e um adder.

Código a ser corrido pelo NIOS II

```
//normalize pixel values between 0 and 1
float f[784];
for (j = 0; j<784;j++){
    f[j]=pixels[j]/255.0;
}

// Calculate the activations for each image using the neural network
//start timer
alt_timestamp_start();
for (ki = 0; ki < MNIST_LABELS; ki++) {
    activations[ki] = network.b[ki];
    for (kj = 0; kj < MNIST_IMAGE_SIZE; kj++) {
        activations[ki] += network.W[ki][kj] * f[kj];
    }
}
time = alt_timestamp();//end timer
```

Figura 15. Este é o ciclo que vai ser acelerado pelo acelerador de hardware, nesta imagem ainda está apenas a correr dentro da cpu, ou seja é a versão 1 do classificador.

```
//pixel preprocessing and initializing
IOWR_ALTERA_AVALON_PIO_DATA(SYNC_DATA_BASE,1<<7);
for (kj = 0; kj < MNIST_IMAGE_SIZE; kj++){
    f[kj] = pixels[kj] / 255.0;
    float f3 = f[kj];
    memcpy(&ui, &f3, sizeof (ui));
    IOWR_ALTERA_AVALON_PIO_DATA(PIXEL_INDEX_BASE,kj);
    IOWR_ALTERA_AVALON_PIO_DATA(FLOATDATA_OUTPUT_BASE,ui);
}

//activations initializing
IOWR_ALTERA_AVALON_PIO_DATA(SYNC_DATA_BASE, (1<<4)+(1<<5));
for (ki = 0; ki < MNIST_LABELS; ki++) {
    float f2 = network.b[ki];
    memcpy(&ui, &f2, sizeof (ui));
    IOWR_ALTERA_AVALON_PIO_DATA(ACTIVATIONS_INDEX_BASE, ki);
    IOWR_ALTERA_AVALON_PIO_DATA(FLOATDATA_OUTPUT_BASE, ui);
}
```

Figura 16. Antes de chegar ao ciclo que controla o acelerador tem de ser feito um preprocessamento e inicialização das memórias RAM. Os *memcpy* servem para transformar os valores que estão em vírgula flutuante para unsigned int (um cast aqui não funciona) para poder enviar a informação destes *floats* como se fosse uma string de 32 bits com a notação da *IEEE Standard for Floating-Point Arithmetic (IEEE 754)* para as memórias RAM.

```
for (ki = 0; ki < 10; ki++) {
    //load activation index
    IOWR_ALTERA_AVALON_PIO_DATA(ACTIVATIONS_INDEX_BASE, ki);
    //clear flip-flops
    IOWR_ALTERA_AVALON_PIO_DATA(SYNC_DATA_BASE, (1<<5));
    //load new value for the flip-flops
    IOWR_ALTERA_AVALON_PIO_DATA(SYNC_DATA_BASE,0);
    IOWR_ALTERA_AVALON_PIO_DATA(SYNC_DATA_BASE,1);
    //cpu-wise initialization
    activations2[ki] = network.b[ki];
    //pixel cicle, only half of iterations are needed
    for (kj = 0; kj < 392; kj++) {
        //load weight index
        IOWR_ALTERA_AVALON_PIO_DATA(WEIGHT_INDEX_BASE,ki*784+kj);
        //enable rom read
        IOWR_ALTERA_AVALON_PIO_DATA(SYNC_DATA_BASE, (1<<8));
        //load pixel index
        IOWR_ALTERA_AVALON_PIO_DATA(PIXEL_INDEX_BASE,kj);
        //enable ALU clock and keep rom read enabled
        IOWR_ALTERA_AVALON_PIO_DATA(SYNC_DATA_BASE, (1<<8)+(1<<9));
        //nios-wise calculation with 392 pixels of offset (calculating the other half of the image)
        activations2[ki] += network.W[ki][kj+392] * f[kj+392];
        //load value from ALU to flip flop
        IOWR_ALTERA_AVALON_PIO_DATA(SYNC_DATA_BASE,0);
        IOWR_ALTERA_AVALON_PIO_DATA(SYNC_DATA_BASE,2);
        //enable writes on activation RAM
        IOWR_ALTERA_AVALON_PIO_DATA(SYNC_DATA_BASE, (1<<4)+(1<<8)+(1<<9));
        //load new value for the activation DFF
        IOWR_ALTERA_AVALON_PIO_DATA(SYNC_DATA_BASE,0);
        IOWR_ALTERA_AVALON_PIO_DATA(SYNC_DATA_BASE,1);
        //repeat for next pixel
    }
}
```

Figura 17. Este é o ciclo que tem o cálculo das activations paralelizado entre RTL e HPS. O seu funcionamento está descrito nos comentários do código.

```
//copy back the RTL calculations and sum them to CPU activations
for (ki = 0; ki < MNIST_LABELS; ki++) {
    IOWR_ALTERA_AVALON_PIO_DATA(ACTIVATIONS_INDEX_BASE, ki);
    IOWR_ALTERA_AVALON_PIO_DATA(SYNC_DATA_BASE,0);
    IOWR_ALTERA_AVALON_PIO_DATA(SYNC_DATA_BASE,1);
    unsigned int a = IORD_ALTERA_AVALON_PIO_DATA(RESULTS_INPUT_BASE);
    activations[ki] = ConvertNumberToFloat(a);
    activations[ki] += activations2[ki];
}

//calculate softmax
```

Figura 18. No fim de calcular as activations no NIOS e no RTL temos de juntar os seus resultados neste ciclo.

SPEEDUP COM O ACELERADOR

Os tempos de execução foram tirados com a biblioteca do timer do University Program. Para este timer funcionar corretamente o BSP tem de ser alterado, e nas suas configurações alterar o tipo de clock que é usado para o time, como mostra na seguinte figura:

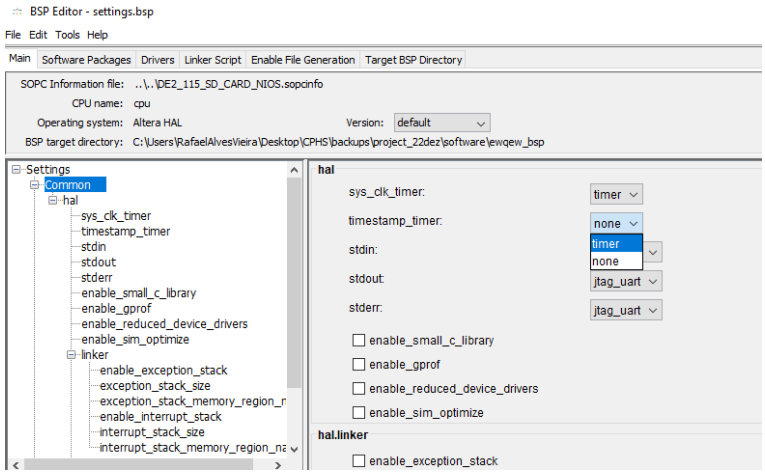


Figura 19. No bsp editor que está na figura deve ser escolhido para o timer, a seleção que diz ”timer”e para o system timer clock ”none”.

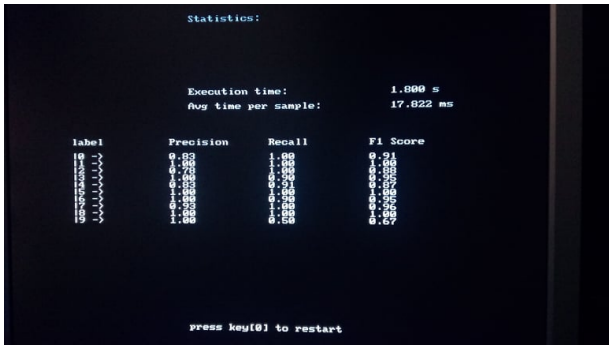


Figura 20. Tempo de execução e métricas para 80 samples sem acelerador de hardware

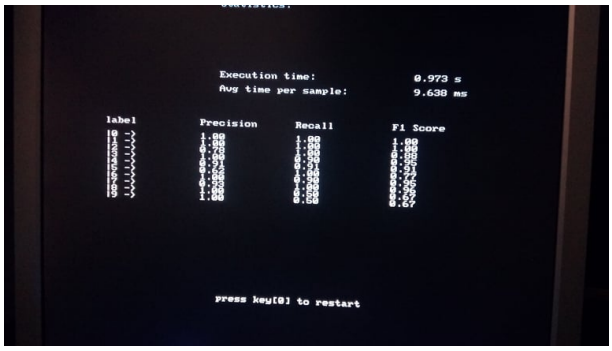


Figura 21. Tempo de execução e métricas para 80 samples com acelerador de hardware

Tempo de execução em cálculos é 1.800s para a versão sem aceleração e 0.973s para a versão com acelerador. Ou seja, poupa-se quase 50% do tempo em cálculos com aceleração de hardware.

Asseguir está representada uma tabela para tempos de execução em função do número de imagens. É de notar que existe alguma precisão e coerência temporal de algo que executa na FPGA, porque posso tentar fazer esta tabela várias vezes, e obtenho sempre os mesmos tempos, ao contrário do que acontece quando se faz uso de GPUs ou CPUs para acelerar programas (por isso é que as FPGAs são melhores para aplicações em sistemas de tempo real). A versão 1 é a que apenas faz uso da ALU do NIOS e a versão 3 tem o acelerador de Hardware.

	Tempos de execução	
nº imagens	versão 1	versão 3
10	0.225s	0.122s
20	0.449s	0.242s
30	0.673s	0.363s
40	0.898s	0.485s
50	1.124s	0.607s
60	1.349s	0.729s
70	1.575s	0.851s
80	1.800s	0.973s

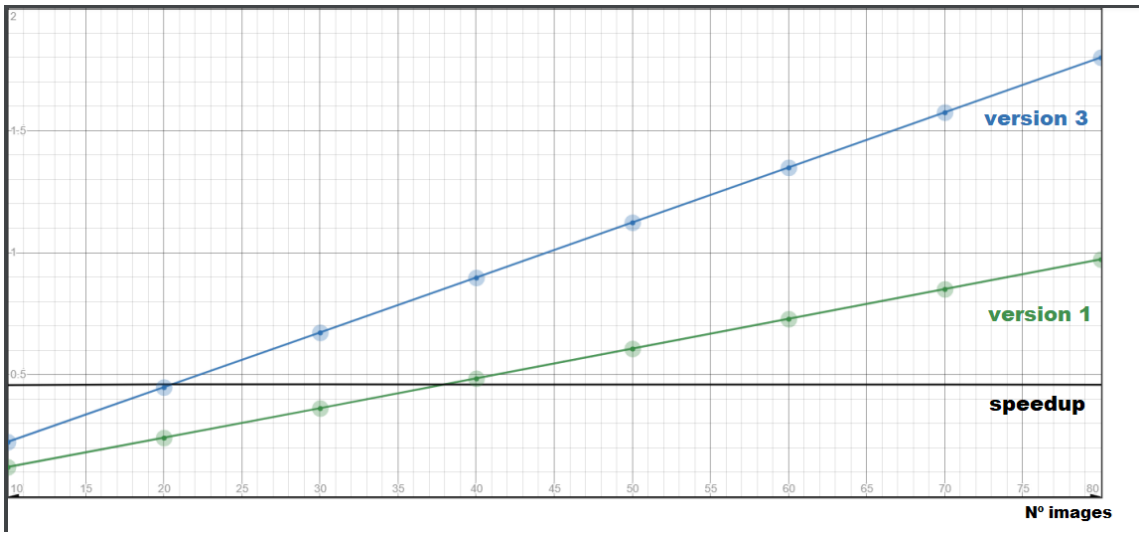


Figura 22. Representação gráfica da tabela anterior, bem como o valor do speedup. Como estamos apenas a contar neste caso os tempo de execução dos cálculos, vamos ter sempre um speedup próximo dos 50%, porque as iterações do ciclo são divididas em metade.

Se contarmos com as transferências e inicialização de dados, temos a seguinte tabela:

	Tempos de execução	
nº imagens	versão 1	versão 3
10	0.271s	0.486s
20	0.537s	0.965s
30	0.805s	1.448s
40	1.077s	1.934s
50	1.348s	2.421s
60	1.621s	2.908s
70	1.896s	3.398s
80	2.170s	3.887s

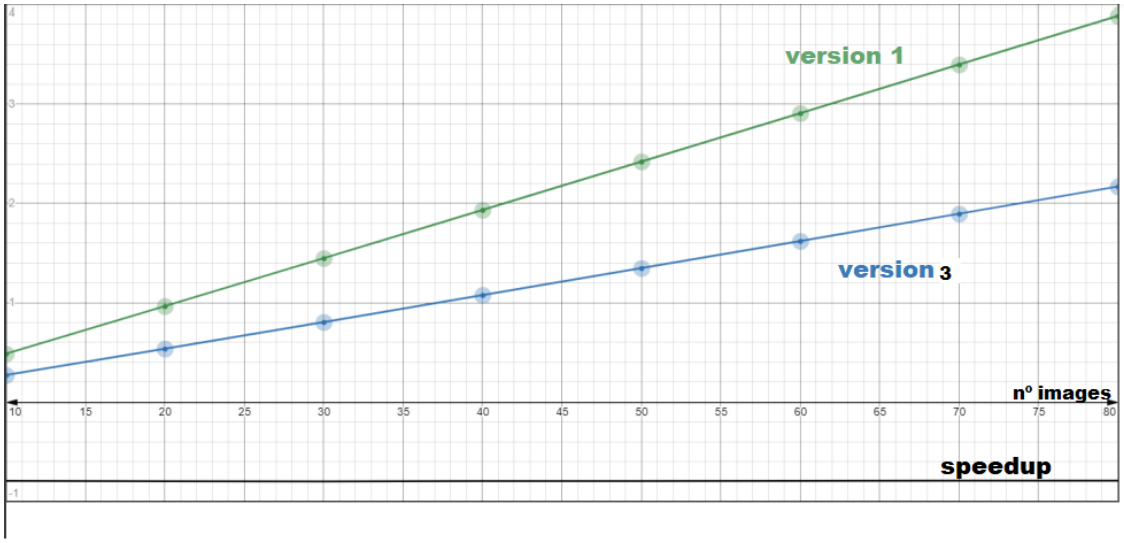


Figura 23. Representação gráfica da tabela anterior, bem como o valor do speedup. As curvas de speedup nos dois gráficos são quase constantes. No gráfico anterior era de esperar porque não tínhamos transferências de memória, mas neste gráfico já estamos a contar com essa parte, daí o speedup negativo. A verdade é que aumentar o número de imagens a classificar não vai ajudar em nada no caso deste acelerador, ao contrário do que seria de esperar. Normalmente tira-se mais proveito de aceleradores quando o peso computacional e tempo de execução dos cálculos aritméticos são compensados pelo tempo gasto em transferências de memória. Portanto, se aumentarmos o número de imagens que são classificadas então supostamente o speedup deveria aumentar, mas esse não é o caso porque no acelerador de hardware que foi implementado. Como apenas há espaço na RAM dos pixeis para uma imagem, não tiramos proveito de ter mais imangens para classificar porque o número de vezes que vamos fazer as transferências de memória será sempre igual ao número de imagens que vamos classificar. Se a transferência de dados ocorresse apenas uma vez (ou seja no caso de 80 imagens, transferir as 80 imagens de uma só vez) então como já só estaríamos a perder tempo com transferências de memória uma vez e talvez já conseguiríamos ver um gráfico de speedup que fosse crescente. Mas esse não foi o caso nesta aplicação.

CONCLUSÕES

Com este trabalho aprendi acerca de aceleradores de hardware e que a sua implementação/sincronização com a CPU não é tão trivial quanto pensava (também foi o primeiro acelerador que desenvolvi por mim mesmo).

Melhorias a fazer

Há muitas melhorias a fazer no que já está implementado, nomeadamente permitir que mais do que uma imagem seja classificada de uma só vez pelo RTL, para diminuir o número de transferências de dados. Para a inicialização dos pixeis existe um ciclo com uma divisão, que é chamado muitas vezes. Sendo a divisão uma das operações que ocupa mais tempo de processamento, tería algum ganho em speedup se acelerasse também esse ciclo.

Observações

Ao começar a desenvolver o acelerador, vi que havia a possibilidade de usar custom instructions chamadas de "Floating point Hardware", que servem para o propósito deste trabalho. No entanto achei mais interessante tentar desenvolver o meu próprio RTL e tratar da sua sincronização. Também reparei que existem *floating point accumulators* no mega wizard com frequências de operação costumizáveis, mas não fiz uso para ter uma melhor visão de tudo o que se estava a passar no RTL.

Recursos usados

Flow Status	Successful - Sat Jan 22 11:09:51 2022
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Standard Edition
Revision Name	projeto_CPHS
Top-level Entity Name	new_toplevel
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	7,573 / 114,480 (7 %)
Total registers	4861
Total pins	121 / 529 (23 %)
Total virtual pins	0
Total memory bits	2,508,941 / 3,981,312 (63 %)
Embedded Multiplier 9-bit elements	13 / 532 (2 %)
Total PLLs	2 / 4 (50 %)

Figura 24. Relatório da compilação/ recursos usados

Fiquei surpreso que para um Hello World que use printf já é necessário ocupar uma grande quantidade de memória da FPGA. Só para o NIOS, está dedicada metade da memória total, tendo em conta que não só estão lá declarados os pesos (os que estão na ROM apenas são usados no RTL, por isso também teem de estar declarados no código em C), como também tem de ter memória sufeciente para a leitura das imagens.

REFERÊNCIAS

[1] DE2-115 User Manual . https://www.intel.com/content/dam/altera-www/global/en$_US/portal/dsn/42/doc-us-dsnbk-42-1404062209-de2-115-user-manual.pdf$.

[2] Looking inside neural nets . https://ml4a.github.io/ml4a/looking_inside_neural$_nets/$.

[3] MNIST Classification on DE2-115 . https://github.com/grant4001/MNIST$_Classification_FPGA$.

[4] MNIST For ML Beginners . https://github.com/AndrewCarterUK/mnist-neural-network-plain-c.

[5] TensorFlow tutorials website . https://www.tensorflow.org/tutorials.

[6] NIOS II - Altera. https://www.intel.com/content/www/us/en/products/details/fpga/nios-processor.html.

[7] THE MNIST DATABASE. http://yann.lecun.com/exdb/mnist/.