Proyecto final: Perceptrón Multicapa para reconocimiento de dígitos

(42339) Sistemas Empotrados – Jesús Barba Romero

Contents

Introducción a la Multi-Layer Perceptron (MLP) y su Relación con el Acelerador

1.1 ¿Qué es una Multi-Layer Perceptron (MLP)?

Una Multi-Layer Perceptron (MLP) es un tipo de red neuronal artificial (ANN) compuesta por múltiples capas de neuronas interconectadas. Es un modelo fundamental en el aprendizaje profundo y se utiliza ampliamente en problemas de clasificación y regresión.

Una MLP consta de tres tipos de capas:

- Capa de Entrada: Recibe las características de entrada.
- Capas Ocultas: Procesan la entrada a través de conexiones ponderadas y funciones de activación.
- Capa de Salida: Proporciona la predicción o clasificación final.

Implementación del Acelerador en Vitis HLS

2.1 Introducción a Vitis HLS y su Rol en el Proyecto

Para la implementación del acelerador del MLP, utilizamos Vitis HLS (High-Level Synthesis), que permite diseñar hardware en FPGA mediante el uso de C/C++, en lugar de descripciones de bajo nivel en Verilog o VHDL. Esto nos facilita la experimentación y optimización del diseño sin necesidad de modificar código RTL directamente.

La idea principal del acelerador es procesar una imagen de 28x28 píxeles y pasarla a través de una MLP con dos capas ocultas de 64 y 128 neuronas respectivamente, generando una salida de 10 valores que representan la probabilidad de cada clase (dígitos del 0 al 9).

2.2 Análisis del Código en Vitis HLS

El código se organiza en diferentes módulos para cada capa de la red:

1. layer_hidden1: Implementa la primera capa oculta de 64 neuronas.
2. layer_hidden2: Implementa la segunda capa oculta de 128 neuronas.
3. layer_output: Calcula la capa de salida con 10 neuronas.
4. MLP: Función principal que gestiona la ejecución de todas las capas y utiliza flujos AXI para la entrada y salida de datos.

2.2.1 Implementación de las Capas Ocultas

Cada capa sigue una estructura similar:

- Se multiplica la entrada por la matriz de pesos.
- Se suma el bias correspondiente a cada neurona.
- Se aplica la **función de activación** Sigmoide.

Por ejemplo, en layer_hidden1:

```
32  void layer_hidden1(float intro[C_INTRO], float outputs[C_HIDD]){
33          L1:for(int i = 0; i < C_HIDD ; i++) { // Each neuron in the hidden layer
34                  float aux = 0;
35
36                  L2:for(int j = 0; j<C_INTRO ; j++) {    // Each input neuron in the input layer
37                          aux += intro[j]*WeightHidden1[i][j];
38                  }
39
40                  aux += BiasesHidden1[i];        // Add the bias
41                  outputs[i] = sigmoid_function(aux);     // Apply the activation function
42          }
43  }
```

Aquí se observa el uso de bucles anidados para recorrer los pesos y entradas, acumulando el resultado en aux antes de aplicar la activación.

2.2.2 Implementación del Módulo Principal MLP

El módulo principal MLP sigue la siguiente secuencia de ejecución:

1. Recibe los datos de entrada a través de un stream AXI.
2. Convierte los datos del flujo AXI a un array de valores flotantes, lo que permite trabajar con los datos en el formato adecuado.
3. Llama secuencialmente a cada capa de la MLP (layer_hidden1, layer_hidden2 y layer_output).
4. Obtiene el resultado y lo convierte nuevamente en un flujo AXI para enviarlo como salida.

Este diseño modular facilita la reutilización y optimización del código, además de permitir futuras mejoras sin necesidad de cambiar la estructura principal del MLP.

Decisiones de Diseño y Código

Manejo del flujo de datos

El MLP utiliza **streams AXI** como interfaz de entrada y salida, lo que optimiza la transferencia de datos en hardware. Se declara de la siguiente manera:

```
#pragma HLS INTERFACE mode=s_axilite bundle=control port=return
#pragma HLS INTERFACE mode=axis      port=INPUT_STREAM
#pragma HLS INTERFACE mode=axis      port=OUTPUT_STREAM
```

Estos pragmas configuran el acceso de la función a los datos a través de la interfaz AXI, permitiendo una comunicación eficiente con otros módulos en el FPGA.

Conversión de datos de entrada

Los datos de entrada se leen del INPUT_STREAM y se convierten en un array:

```
82              for(int i=0; i<IMG_DIM; i++)
83                      for(int j=0; j<IMG_DIM; j+= NUM_ELEMS_WORD){
84                              #pragma HLS PIPELINE II=1
85                              WORD_MEM w = INPUT_STREAM.read().data;
86                              conv_t c;
87                              for (int k=0; k<NUM_ELEMS_WORD;k++) {
88                                      c.in = w((k+1)*32-1,k*32);
89                                      image_pixels[(i*IMG_DIM) + (j+k)] = c.out;
90                              }
91                      }
```

Este fragmento de código descompone los datos en palabras de 32 bits (WORD_MEM) y los almacena en el array image_pixels. La directiva #pragma HLS PIPELINE II=1 se usa para optimizar la latencia del procesamiento de datos.

Ejecución de las capas de la red

Las funciones que implementan las capas ocultas y de salida son llamadas secuencialmente:

```
layer_hidden1(image_pixels, l_hidden1_outputs);
layer_hidden2(l_hidden1_outputs, l_hidden2_outputs);
layer_output(l_hidden2_outputs, l_out_outputs);
```

Cada capa procesa sus datos utilizando los pesos y biases almacenados en coef.c y bias.c, asegurando que la red neuronal funcione de manera consistente y eficiente en hardware.

Escritura de datos en la salida

Después de obtener el resultado, este se convierte nuevamente a un flujo AXI para su salida:

```
// Write results in the output stream
for(int j=0; j<C_OUT; j+= NUM_ELEMS_WORD) {
        AXI_VAL e;
        conv_t c;
        WORD_MEM w;

        for (int k=0; k<NUM_ELEMS_WORD;k++) {

                c.out = l_out_outputs[j+k];
                w((k+1)*32-1,k*32)= c.in;
        }
        e.data = w;
        e.strb = -1;
        e.keep = 15; //e.strb;
        e.user = U;
        e.last = (j == (C_OUT-NUM_ELEMS_WORD));
        e.id = TI;
        e.dest = TD;
        OUTPUT_STREAM.write(e);
}
```

Este código garantiza que los datos procesados sean enviados correctamente a través del flujo AXI para su posterior uso en otros módulos del sistema.

Implementación en Vivado

3.1 Generación del Block Design

Para la implementación del sistema en hardware, utilizamos Vivado para crear un Block Design en el que integramos todos los componentes necesarios.

El diseño de bloques está basado en un sistema Zynq UltraScale+ MPSoC, que se encarga de gestionar la comunicación con el acelerador. En este diseño se incluyen:

- Zynq UltraScale+ MPSoC: Procesador principal encargado de gestionar el sistema.
- AXI Interconnect: Permite la comunicación entre los diferentes periféricos.

- AXI Timer: Proporciona capacidades de medición de tiempo para evaluar el rendimiento del sistema.
- AXI DMA: Facilita la transferencia eficiente de datos entre la memoria y el acelerador.
- MLP (Vitis HLS): Nuestro acelerador diseñado en Vitis HLS, el cual recibe los datos de entrada a través de INPUT_STREAM y devuelve los resultados mediante OUTPUT_STREAM.

3.2 Exportación e Integración con Vitis

Una vez completado el diseño del hardware en Vivado:

1. Se genera el bitstream para configurar el FPGA.
2. Se exporta el hardware junto con la información de la plataforma.
3. Se utiliza esta plataforma en Vitis para desarrollar la aplicación software que controla la ejecución del MLP en el FPGA.

Con esta implementación en Vivado, hemos conseguido configurar correctamente el sistema hardware para ejecutar el acelerador del MLP. En la siguiente sección detallaremos el uso de Vitis para programar y probar el sistema final.

Implementación en Vitis

4.1 Procedimiento en Vitis

Después de exportar el archivo XSA desde Vivado, seguimos los siguientes pasos en Vitis:

1. Creamos un nuevo workspace y seleccionamos la plataforma basada en el archivo XSA exportado.
2. Creamos una nueva aplicación embebida y elegimos el template básico para trabajar con el procesador Zynq.
3. Agregamos los archivos fuente proporcionados (main.c, lib_xmlp_hw.c, lib_xmlp_hw.h, platform.c, etc.).
4. Configuramos el entorno de compilación y las dependencias necesarias para el correcto funcionamiento del sistema.

4.2 Análisis del Código

El código principal (main.c) realiza una serie de tareas clave para la ejecución del acelerador MLP en el sistema embebido. A continuación, se detallan estas tareas junto con su justificación y ejemplos de implementación:

Carga de Imágenes y Etiquetas

Para evaluar el rendimiento del acelerador, se necesita una base de datos de imágenes y sus respectivas etiquetas. Se realiza la lectura de archivos del sistema de archivos embebido mediante la biblioteca ff.h. La función Load() se encarga de montar el sistema de archivos y cargar las imágenes:

```
u32 Load(){
    FIL file;
    u32 bytes_read = 0;

    if (mount_filesystem() == XST_FAILURE)
        return XST_FAILURE;

    FRESULT res = f_open(&file, LABELS_FILE, FA_READ);
    if (res != FR_OK) {
        xil_printf("\rCould not open the labels file %d\n\r", res);
        return XST_FAILURE;
    }

    long fileSize = f_size(&file);
    f_read(&file, m_labelData, fileSize, &bytes_read);
    f_close(&file);
    return XST_SUCCESS;
}
```

Esta función garantiza que las imágenes y etiquetas sean leídas correctamente antes de ejecutar el acelerador.

Inicialización del DMA

El AXI DMA se utiliza para transferir datos entre la memoria y el acelerador sin involucrar al procesador. La función init_dma() inicializa el controlador del DMA:

```
int init_dma(){
    XAxiDma_Config *CfgPtr;
    CfgPtr = XAxiDma_LookupConfig(XPAR_AXI_DMA_0_DEVICE_ID);
    if (!CfgPtr) {
        print("Error looking for AXI DMA config\n\r");
        return XST_FAILURE;
    }
    int status = XAxiDma_CfgInitialize(&AxiDma, CfgPtr);
    if (status != XST_SUCCESS) {
        print("Error initializing DMA\n\r");
        return XST_FAILURE;
    }
    return XST_SUCCESS;
}
```

Esta inicialización es crítica para permitir la comunicación eficiente con el acelerador.

Configuración **del Temporizador**

Para medir el rendimiento, se usa un **temporizador hardware (AXI Timer)**. Su configuración se realiza con XTmrCtr_Initialize y XTmrCtr_Reset:

```
status = XTmrCtr_Initialize(&timer_dev, XPAR_AXI_TIMER_0_DEVICE_ID);
XTmrCtr_SetOptions(&timer_dev, XPAR_AXI_TIMER_0_DEVICE_ID, XTC_ENABLE_ALL_OPTION);
XTmrCtr_Reset(&timer_dev, XPAR_AXI_TIMER_0_DEVICE_ID);
```

Esto nos permite capturar el tiempo de ejecución de cada inferencia y comparar su eficiencia respecto a una ejecución en software.

Ejecución del Acelerador

La inferencia de la red neuronal se realiza en hardware llamando a Run_HW_Accelerator(). Este método transfiere la imagen al acelerador y obtiene la salida:

```
int Run_HW_Accelerator(float image[IMGPIXELES], float results[10]) {
    int status = XAxiDma_SimpleTransfer(&AxiDma, (unsigned int) image, sizeof(float)*I
    if (status != XST_SUCCESS) {
        print("Error: DMA transfer to Vivado HLS block failed\n");
        return XST_FAILURE;
    }

    status = XAxiDma_SimpleTransfer(&AxiDma, (unsigned int) results, sizeof(float)*10,
    if (status != XST_SUCCESS) {
        print("Error: DMA transfer from Vivado HLS block failed\n");
        return XST_FAILURE;
    }
    while (!ResultExample);
    ResultExample = 0;
    return XMlp_Get_return(&xmlp_dev);
}
```

Esta función configura el flujo de datos entre la CPU y el acelerador utilizando DMA, asegurando un procesamiento eficiente.

Comparación de Resultados

El resultado obtenido del acelerador se compara con la etiqueta real para evaluar la precisión:

```
int GetResults(float results[10]) {
    int digit = 0;
    for (int i = 0; i < 10; i++) {
        if (results[digit] < results[i]) {
            digit = i;
        }
    }
    return digit;
}
```

4.3 Problemas y Errores

Durante la compilación y ejecución en Vitis, se encontraron problemas con algunas librerías, en particular:

- Errores de inclusión: Algunas librerías como xparameters.h, xil_printf.h o xaxidma.h pueden no estar correctamente enlazadas.
- Configuración del DMA: En algunos casos, el DMA no se inicializa correctamente, lo que genera fallos en la transferencia de datos.

- Interrupciones no registradas: El controlador de interrupciones (XScuGic) puede fallar si no se inicializa correctamente.
- Biblioteca ff.h no encontrada: No se ha podido incluir correctamente la biblioteca ff.h en el proyecto, lo que impide la gestión del sistema de archivos.

Error Encontrado en la Compilación

Además, durante la compilación, se obtiene el siguiente error:

aarch64-none-elf-ld: cannot find -l xilffs: No such file or directory

collect2.real: error: ld returned 1 exit status

*make: *** [makefile:38: mlp_accel_app.elf] Error 1*

Este error indica que la biblioteca xilffs no está siendo encontrada por el linker. Esto suele deberse a que la biblioteca no ha sido agregada correctamente a las dependencias del proyecto en Vitis.