Universidad Zaragoza

Trabajo fin de Grado

Diseño de un acelerador específico para árboles de decisión

Design of a specific accelerator for decision trees

AUTOR

Vlad Teletin

DIRECTORES

Javier Resano Ezcaray

ESCUELA DE INGENIERÍA Y ARQUITECTURA

DICIEMBRE 2019

RESUMEN

Desde la fabricación del los primeros procesadores, la densidad de transistores que los forman no ha hecho más que aumentar. Esto ha llevado a la aparición de System on Chip (SoC) muy complejos, con varios procesadores, redes y recursos de memoria. Una forma de mejorar el rendimiento y la eficiencia energética de estos sistemas para algunas funcionalidades críticas es incluir en los chips recursos de computación especializados denominados aceleradores.

Por otro lado, es incuestionable que uno de los campos de la informática que más popularidad está teniendo actualmente es la inteligencia artificial, concretamente el aprendizaje automático.

En este proyecto se han juntado estos dos conceptos y se ha diseñado un acelerador específico para la evaluación de una técnica concreta del aprendizaje automático, los árboles de decisión, siendo éstos entrenados usando una técnica que ayuda a mejorar sus resultados denominada *Gradient Boosting*. Para ello se ha estudiado cómo se generan los árboles de decisión utilizando el *framework* LightGBM [4], uno de las entornos más populares que se usan actualmente para los árboles de decisión, para poder diseñar un acelerador específico capaz de realizar los mismos cálculos de la forma más eficiente posible. El acelerador se ha diseñado usando el lenguaje de descripción de *hardware* VHDL, y su funcionamiento se ha comprobado en un SoC real que incluye dos procesadores y una Field Programming Gate Array (FPGA), un recurso de lógica programable, donde se ha implementado el acelerador. Para la evaluación se ha utilizado un caso de estudio complejo, la clasificación de píxeles de imágenes hiperespectrales. Junto a ésto, se ha diseñado una herramienta que permita adaptar los datos de un modelo entrenado con este *framework* para su utilización dentro de un acelerador *hardware* de este tipo.

Además, con el fin de analizar la viabilidad de este tipo de acelerador, se compara con un código escrito en C que realiza los mismos cálculos, ejecutado en uno de los procesadores de propósito general de los cuales dispone el SoC. De esta forma se han obtenido datos de tiempo de ejecución y consumo energético que apoyan la utilización de un acelerador frente a un procesador de propósito general para esta tarea concreta, dado que es capaz de realizar los cálculos 8.8 veces más rápido y reducir el consumo de energía dinámico en un factor de 8.7.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

LISTINGS

ACRÓNIMOS

FPGA Field Programming Gate Array

LUT Lookup Table

SoC System on Chip

GBDT Gradient Boosting Decision Trees

RAM Random Access Memory

BRAM Block Random Access Memory

DMA Direct Memory Access

SDK Software Development Kit

AXI Advanced eXtensible Interface

CPU Central Processing Unit

DDR Double Data Rate

JSON JavaScript Object Notation

INTRODUCCIÓN

1.1 MOTIVACIÓN

En la época actual, gracias a los avances de la tecnología, los circuitos integrados que se están utilizando tanto en el ámbito doméstico como en el ámbito industrial tienen una densidad de transistores enorme. Esto permite crear dentro de un chip sistemas realmente complejos en los que se pueden incluir varios procesadores, redes en chip o recursos de memoria. Sin embargo poco a poco se está alcanzando el límite de este crecimiento dejando como una de las pocas opciones de mejora de las capacidades de los sistemas la especialización. La idea es crear sistemas heterogéneos con procesadores de propósito general e incluyendo recursos especializados que actúen como aceleradores altamente eficientes para determinados problemas relevantes como pueden ser la computación gráfica, el tratamiento de señal o la criptografía. Estos aceleradores pueden tener una funcionalidad fija o implementarse sobre dispositivos programables en los que se pueden cargar aceleradores nuevos en tiempo de ejecución. Un buen ejemplo son las FPGA que se pueden encontrar en los SoC de Intel y Xilinx.

Uno de los campos en los que más se está investigando en el desarrollo de aceleradores es el de aprendizaje automático debido a la revolución que se ha vivido en éste en los últimos años. Existen diversas técnicas de implementación dentro del aprendizaje automático, pero una de los que más éxito está teniendo actualmente son los árboles de decisión. Se ha convertido en uno de los métodos preferidos para la clasificación gracias a su baja exigencia computacional y a que se obtienen rendimientos muy buenos con un bajo coste temporal y consumo energético.

Este proyecto plantea la utilización de una FPGA para la implementación de un acelerador de este tipo, lo que resulta realmente interesante al reducir los costes de producción y aportar versatilidad y mejoras en el rendimiento y en la eficiencia energética.

1.2 OBJETIVOS

El objetivo de este proyecto es el diseño y la implementación de un acelerador de árboles de decisión sobre una FPGA. Sobre éste se realizará un estudio analizando las mejoras temporales y de consumo obtenidas con la utilización de éste frente a la de un *software* que lleve a cabo la misma tarea utilizando un procesador de propósito general. Los dos sistemas sobre los que se ejecutan las tareas forman parte del mismo SoC con lo cual están implementados en la misma tecnología, haciendo la comparación más equitativa.

Para la ejecución de los árboles de decisión se utiliza un conjunto de datos multivariante formado por píxeles de una imágen hiperespectral que, dada su complejidad, sitúan el estudio en un entorno más realista y exigente.

1.3 ALCANCE

La realización de un acelerador de este tipo no es sencillo ya que se trata de un trabajo muy especializado. Se deben conocer muy bien tanto el campo del aprendizaje automático como el del diseño e implementación de *hardware*. Es por ello que una gran parte del tiempo se ha tenido que emplear en profundizar en estos dos temas que, a priori, no tienen mucha relación entre sí.

Sin embargo, siguiendo las tendencias del mercado, parece inevitable que en pocos años todos los dispositivos que salgan al mercado tengan incluidos uno o varios de estos aceleradores. Algunos serán para cifrado, otros serán para el tratamiento de señal, pero serán frecuentes los enfocados al aprendizaje automático para tareas como reconocimiento facial o procesamiento de lenguaje natural.

Lo que se intenta con este proyecto es dar una primera visión de cómo estarán creados estos aceleradores y cuáles son sus capacidades. Para ello se presenta un estudio con las diferencias, tanto en consumo de energía y potencia como de tiempo, entre un programa en C que simula el acelerador y el simulador en sí. Pudiendo, de esta forma, poder analizar su viabilidad.

Tras la finalización del proyecto se desea obtener un diseño *hardware*, utilizando el lenguaje de descripción de *hardware* VHDL, del acelerador, realizado de la forma más modular y parametrizable posible con el fin de su uso en cualquier tipo de FPGA [8]. Junto a éste, también se entrega un programa con el cual se puedan tratar los modelos entrenados para ajustar su formato al empleado internamente en el acelerador [9]. Cabe mencionar que todos los frutos de la realización de este proyecto se publicarán de forma pública y con una licencia absolutamente libre (MIT) para intentar apoyar la utilización y mejora de este tipo de proyectos, los cuales parecen ser cada vez más habituales.

FUNDAMENTOS

En este capítulo se explican una serie de términos que están presentes a lo largo de todo el documento para que, de esta forma, la lectura pueda ser llevada a cabo con más facilidad. Entre estas explicaciones están qué es y cuál es la función de un acelerador, qué son las FPGAs y por qué son tán importantes hoy en día, también se explica qué es un árbol de decisión y su entrenamiento y por último, qué son las imágenes hiperespectrales ya que estas son objeto de evaluación de los árboles entrenados en este proyecto.

2.1 ACELERADOR

Un acelerador es un componente *hardware* específico para la realización de una tarea en concreto de una manera muy eficiente, mucho más que ejecutando esa tarea en un procesador comercial de propósito general. Se utilizan para disminuir latencias y aumentar el rendimiento de los procesadores. Un ejemplo de aceleradores *hardware* son los utilizados en criptografía, el tratamiento de tramas de red, procesamiento de gráficos o la evaluación de técnicas de aprendizaje automático como redes neuronales.

2.2 FPGA

Una FPGA es un circuito integrado diseñado para ser programado después de su fabricación. Esto normalmente se hace utilizando un lenguaje de descripción *hardware* como pueden ser VHDL o Verilog. En el Listing 1 se puede ver un ejemplo sencillo que describe un multiplexor 2 a 1. Aunque parece un lenguaje de programación, lo que se hace no es programar, sino describir los componentes y sus conexiones para que las herramientas de síntesis puedan convertir esta descripción en puertas lógicas que después se configurarán en una región de la FPGA.

```
1   entity mux2_1_16b is
2       Port ( Din0 : in STD_LOGIC_VECTOR (15 downto 0);
3              Din1 : in STD_LOGIC_VECTOR (15 downto 0);
4              Ctrl : in STD_LOGIC;
5              Dout : out STD_LOGIC_VECTOR (15 downto 0));
6   end mux2_1_16b;
7
8   architecture Behavioral of mux2_1_16b is
9   begin
10      -- Selects an output based on the ctrl signal
11      Dout <= Din0 when ctrl = '0' else Din1;
12  end Behavioral;
```

Listing 1: Ejemplo de VHDL MUX2:1

Estos circuitos están compuestos por una matriz de celdas o bloques lógicos y una malla de interconexión reconfigurable que los conecta todos entre sí tal y como se puede apreciar en la Figura 1.

Figura 1: Matriz de celdas lógicas

En la Figura 2 se muestra una de estas celdas en detalle. De izquierda a derecha se puede apreciar como estas celdas están formadas por Lookup Table (LUT)s, multiplexores, puertas lógicas simples y registros tipo *flip-flop*. Los LUTs son unas tablas que pueden utilizarse como un recurso de memoria o como un recurso lógico. Para implementar lógica combinacional basta con escribir en los LUTs la tabla de verdad de la lógica deseada.

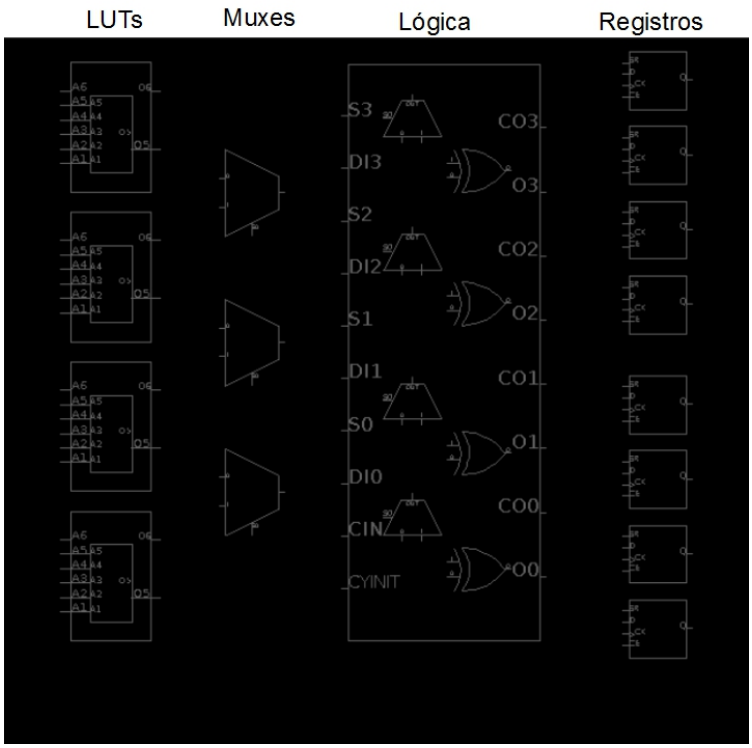


Figura 2: Celda lógica

Con estos elementos y utilizando un lenguaje de descripción de *hardware* las FPGAs se pueden programar para que actúen como cualquier diseño *hardware* que se pueda implementar, incluso un procesador completo, aunque en este trabajo en concreto se va a utilizar para la implementación de un acelerador.

Utilizar una FPGA para implementar un acelerador proporciona peor rendimiento que hacer el acelerador directamente en un circuito integrado a medida, sin embargo es un recurso versátil que se puede programar tantas veces como se quiera. De esta forma los mismos recursos de la FPGA se pueden usar para múltiples aceleradores en función de las necesidades del sistema. Además, al poderse configurar en muy poco tiempo, unos pocos milisegundos, se podrían utilizar para cargar los aceleradores necesarios en el arranque de una aplicación concreta.

2.3 ÁRBOLES DE DECISIÓN

Dentro del aprendizaje automático una de las técnicas utilizadas para la toma de decisiones y la predicción son los árboles de decisión. Existen dos tipos, los árboles de clasificación, de los cuales se obtiene un valor discreto y los árboles de regresión donde el valor obtenido es continuo. Generalmente los árboles de decisión se utilizan con conjuntos de datos multivariante en los que el problema tiene más de una variable o propiedad a evaluar.

Estos árboles están formados por dos tipos de nodos diferentes. Los **nodos de decisión** y los **nodos resultado**. Los nodos de decisión son los nodos intermedios del árbol, los que permiten llevar a cabo la evaluación. Cada uno de éstos tiene una lista de nodos hijo y una propiedad asociada, la cual, mediante una comparación, permite elegir uno de los caminos para avanzar dentro del árbol en la evaluación del nodo. Por otro lado, se encuentran los nodos resultados los cuales son las hojas del árbol. Éstos, a diferencia de los anteriores, en lugar de tener una propiedad y una lista de nodos hijo solamente tienen un valor que será el resultado de la clasificación o regresión.

Entrenando un árbol de decisión utilizando el conjunto de datos multivariante iris de Fisher se obtiene un ejemplo sencillo de cómo un árbol, como el de la Figura 3, puede clasificar flores de distintas familias basándose en el ancho y el largo de los pétalos y los sépalos.

La evaluación de un árbol consite en utilizar un objeto de evaluación, en este caso una flor, y comenzando por el nodo raíz, comparar el valor de las propiedades del objeto con la que indica el nodo en cada momento. Se escoge como siguiente nodo a evaluar el hijo izquierdo de éste si la comparación es cierta y el derecho en caso contrario. Este proceso se realiza hasta que se alcance un nodo resultado, que es el que indica a qué clase pertenece el objeto evaluado.

Este es un ejemplo muy sencillo para la práctica, en realidad existen multitud de maneras de entrenar y con ello generar estos árboles de decisión. La técnica que se utiliza en este proyecto en concreto se denomina *Gradient Boosting* que consiste en un aprendizaje iterativo donde cada una de las iteraciones genera un árbol de decisión que intenta minimizar el error residual de la anterior. Estos árboles, denominados Gradient Boosting Decision Trees (GBDT), se enlazan dando lugar a lo que se denomina un estimador o clasificador. Para este entrenamiento en concreto

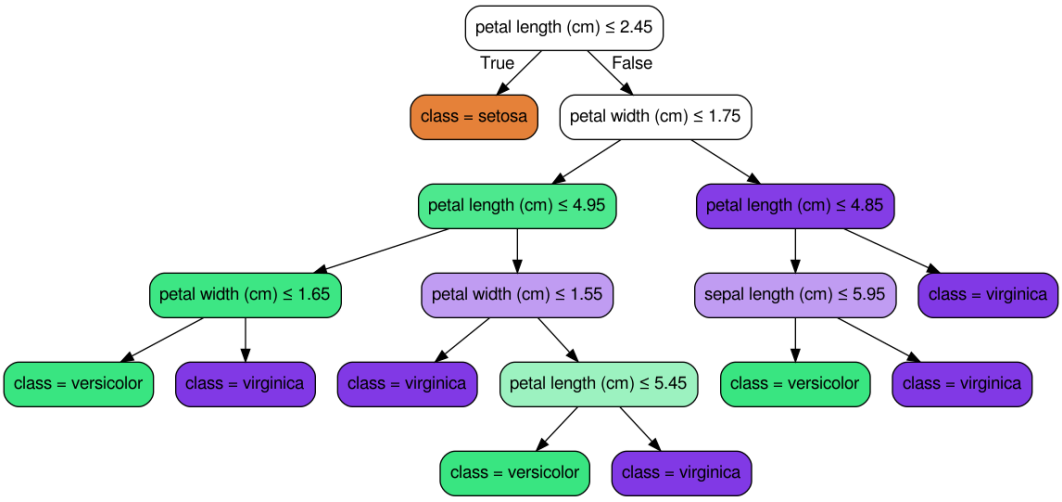


Figura 3: Árbol de decisión para clasificación de flores

existen diferentes estratégias, de las cuales se ha elegido una denominada *one-vs-all* que, para cada una de las clases de la clasificación, produce un estimador que indica el grado de pertenencia del objeto de evaluación a esa clase. Por lo tanto, en el entrenamiento, cada una de las iteraciónes genera un árbol por clase y en sucesivas iteraciones se minimizan los errores residuales para cada una de ella de manera independiente.

El resultado de cada uno de los estimadores consiste en la suma de los resultados obtenidos en la evaluación de cada uno de los arboles perteneciente a éste y en general, el resultado de la clasificación del modelo es el resultado de la aplicación de una función *argmax*[1] sobre los resultados obtenidos de cada uno de los estimadores. Es decir, la clase cuyo grado de pertenencia sea mayor es la elegida como salida del clasificador.

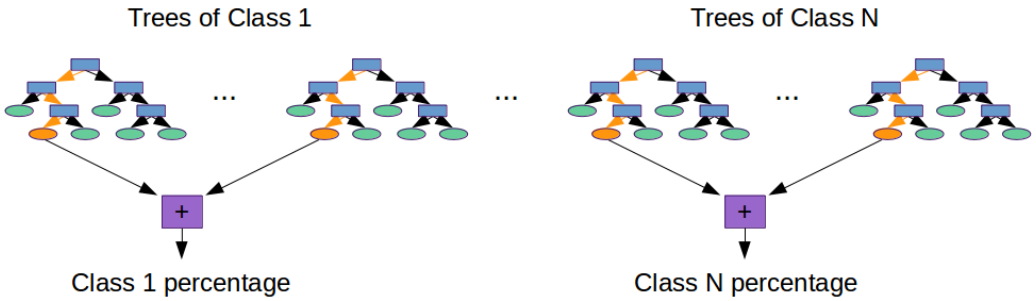


Figura 4: Acumulación de los resultados de los GBDT utilizando one-vs-all

2.4 IMÁGENES HIPERESPECTRALES

Son el resultado de la recopilación y el tratamiento de información de todo el espectro electromagnético. Se consiguen utilizando un sensor que combina un conjunto de imágenes, donde cada una de ellas representa un estrecho conjunto de longitudes de onda en el espectro electromagnético, en una estructura tri-dimensional (x,y,λ) denominada cubo hiperespectral. En este cubo las dimensiones x e y representan dos dimensiones espaciales de la escena y λ representa la dimensión espectral.

1 argmax: En este caso indica la el estimador que tiene un mayor resultado

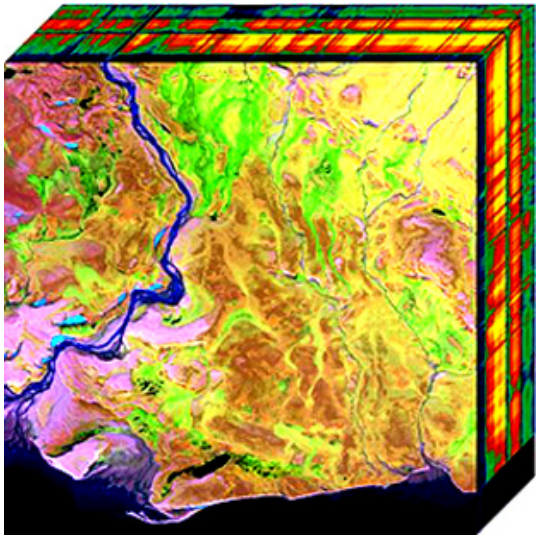


Figura 5: Cubo Hiperespectral

El objetivo de estas imágenes es obtener el espectro específico de cada píxel, denominada firma espectral, y de esta forma poder encontrar objetos o identificar materiales. Estas imágenes tienen multitud de aplicaciones, entre las que se encuentran la detección de anomalías, la monitorización medioambiental y de cultivos o el diagnósticos médicos.

TRABAJO RELACIONADO

Los árboles de decisión son un algoritmo conocido que consigue muy buenos resultados en problemas de regresión y clasificación con muy bajo coste computacional. Una de sus principales ventajas es que necesitan mucho menos procesamiento de datos que otro tipo de algorítmos presentes en el aprendizaje automático ya que no combinan los diferentes parámetros de entrada [2].

Últimamente están surgiendo soluciones *software* y métodos que están especializados en los árboles de decisión como por ejemplo el *Gradient Boosting* que, combinando varios árboles, consigue reducir el tiempo de entrenamiento mientras incrementa la precisión de las predicciones. LightGBM [4] es uno de estos algoritmos optimizados para la eficiencia que, además, provee de un *framework* especializado con un interfaz para Python que permite una experimentación muy ligera y rápida. Todos los GBDT utilizados en este trabajo han sido entrenados utilizando este *framework*.

Existen ya algunos trabajos previos que utilizan una FPGA para ejecutar aceleradores de árboles de decisión [6, 5, 7]. [6] se centra en el proceso del entrenamiento de los árboles. En el caso de este trabajo se asume que el proceso de entrenamiento va a ser totalmente ajeno a la FPGA y se centra en el proceso de inferencia, el cual sí que es llevado a cabo dentro de ésta. [5] propone utilizar un lenguaje de alto nivel para la síntesis de un diseño de acelerador. Se centran en otro método de entrenamiento de los árboles de decisión, los *Random Forest*, que agrupan los árboles de decisión entrenados con diferentes datos de forma que generan una salida más precisa y robusta. La elección de utilizar los GBDT en lugar de estos *Random Forest* es debida a que los GBDT han demostrado tener un enorme potencial [4]. Por ejemplo, en el caso de las imágnes hiperespectrales, el grupo de investigación al que pertenece mi director de proyecto había comparado las dos opciones utilizando el mismo número de árboles con diversas imágenes y los GBDT alcanzaban una precisión del 92 % mientras que los *Random Forest* sólo llegaban al 86 %. Además se ha querído desarrollar una arquitectura personalizada en lugar de utilizar un lenguaje de alto nivel para la síntesis para poder controlar el diseño final y ser capaces de hacer ajustes finos. Por último [7] presenta una arquitectura en *pipeline* que demuestra el potencial de los aceleradores de árboles de decisión. Sin embargo, no soporta le ejecución de GBDT ni de los *Random Forest*, aplican su diseño a unos casos de estudio muy sencillos.

METODOLOGÍA

4.1 PLATAFORMA

Para el desarrollo de este proyecto se utiliza la placa de desarrollo Zynq-7000 de Xilinx [11] que tiene un SoC basado en ARM que incluye una FPGA embebida. El procesador es un *dual-core* ARM Cortex-A9 que trabaja a una frecuencia de 667MHz y la FPGA trabaja a una frecuencia de 100MHz y dispone de 85000 celdas lógicas, 53200 LUTs, 106,400 registros tipo *flip-flop* y 4.9Mb de Block Random Access Memory (BRAM), un tipo de RAM síncrona embebida dentro del FPGA. Además de ésto, también se dispone de 512MB de memoria DDR3 como memoria principal para el sistema.

Junto a esto, se utiliza en entorno de desarrollo Vivado de Xilinx. Este entorno permite sintetizar los diseños especificados en lenguajes de descripción *harware* y generar el bitstream[1] para programar la FPGA. Provee además un Software Development Kit (SDK) basado en Eclipse con el que se pueden crear y ejecutar programas en C en la Central Processing Unit (CPU) e interactuar con la FPGA.

El programa que se ejecuta en la CPU es el encargado de varias tareas. Se encarga del interfaz de entrada/salida con el exterior, enviando distintos mensajes para el usuario. También se ocupa de la gestión de los datos: el almacén en memoria principal de los árboles y los datos de entrada, de su envío al acelerador y de leer la salida de éste.

La interacción entre la CPU y la FPGA se hace utilizando el interfaz Advanced eXtensible Interface (AXI) [10] que define distintos métodos de comunicación. De ellos se han implementado dos. En primer lugar se ha utilizado un interfaz AXI-lite que permite la comunicación a través de registros de 32 bits para cantidades pequeñas de datos. Este interfaz se utiliza tanto para las labores de control (permite acceder en todo momento al estado del acelerador, así como la interfaz de depuración de este), como para leer la salida del acelerador. Sin embargo, este interfaz no es adecuado para enviar grandes volúmenes de datos como pueden ser las estructuras de los estimadores o los píxeles (que tienen un tamaño de 224 palabras de 16 bits). Para realizar estas comunicaciones de forma eficiente se utiliza el protocolo AXI-Stream que funciona utilizando un controlador Direct Memory Access (DMA). Éste tiene un ancho máximo de palabra de 32 bits y, aunque es bidireccional, solamente se utiliza para enviar los datos a la FPGA dado que la salida del acelerador ocupa sólo una palabra, y para tamaños pequeños es más eficiente usar el AXI-lite.

4.2 DISEÑO

En esta sección se comentan brevemente las decisiones de diseño más importantes tomadas para adaptar el problema al entorno descrito en la sección anterior. Además, en cada uno de las siguiente subsecciones se tratarán estas decisiones en

1 bitstream: Fichero que incluye toda la información necesaria para la programación de la FPGA

mayor profundidad.

Uno de los objetivos principales del diseño es que sea totalmente modular donde cada componente, si tiene suficiente entidad, sea encapsulado y si es posible parametrizado para que el mismo diseño se pueda ajustar a cambios en los parámetros del problema. De esta forma se puede ajustar a cambios en el tamaño de los árboles o de los datos de evaluación.

Se ha utilizado un desarrollo iterativo e incremental en el cual cada una de las funcionalidades del acelerador, la transmisión de los datos, la evaluación de los árboles o el procedimiento de depuración se han implementado e integrado gradualmente en el diseño hasta conseguir un diseño completamente funcional.
El diseño ha comenzado con el estudio del entrenamiento y funcionamiento de los árboles de decisión en general y posteriormente de los más específicos GBDT. Como primera aproximación, se ha diseñado un módulo específico para la evaluación de un solo árbol de decisión con datos estáticos con el fin de comprobar la validez del algoritmo de evaluación. Sin embargo, dado que el objetivo es la evaluación de GBDT, este diseño inicial se ha modificado para ajustarlo a éstos. Las modificaciónes permiten la carga dinámica de datos, tanto de los árboles como de las propiedades y la ejecución de un conjunto de árboles de decisión encadenados en lugar de uno solo. De igual manera, como el objetivo es realizar una clasificación de N clases, se ha implementado un módulo específico para contener los árboles de cada una de las clases de tal manera que se pueden evaluar paralelamente. Con este fin, se ha creado también un módulo que alberga las diferentes propiedades del objeto a evaluar de tal forma que se pueda acceder desde todas las clases y no se almacene localmente a cada estimador desperdiciando espacio. Por último se necesita un módulo que aplique la función *argmax* sobre los resultados de las N clases con el fin de obtener un único valor como resultado de la clasificación tal y como se ha comentado en la Sección 2.3.

Una de las decisiones de diseño más importantes es la representación de los datos. Es un apartado sumamente importante ya que es el que compromete la memoria disponible así como el rendimiento, dado que el envío de datos es la operación más costosa en este tipo de problemas.

Finalmente se explica la importancia de tener un método de depuración y de verificación de la integridad de los datos para asegurar la correcta ejecución del acelerador.

4.2.1 *Datos*

Como se ha indicado antes, la forma de representar los datos del problema es sumamente importante. No solo porque el envío de datos es una operación costosa en tiempo sino porque también se debe tener en cuenta el limitado espacio de almacenamiento de las FPGAs.
Tal y como se indica en la Sección 2.3 existen dos tipos de datos, los **nodos de los árboles** y los **objetos de evaluación**, los píxeles de una imágen hiperespectral.
Los árboles utilizados son árboles binarios[2] pero no son necesariamente completos, lo que implica que cada uno de los nodos debe tener información acerca de

2 árboles binarios: Árboles en los que cada nodo tiene un máximo de dos hijos

dónde se encuentran cada uno de sus nodos hijo así como dónde se encuentra el siguiente árbol a evaluar. Además de ésto, necesita información de cómo realizar la evaluación y qué tipo de nodo es. Cabe destacar que hay algunos campos como *is_leaf* y *last tree mark* en los que no se utilizan todos los bits, pero se ha decidido dejarlos, de tal forma los datos se pueden transmitir de una manera más sencilla.

Para cada tipo de nodo se tiene una estructura ligeramente diferente, de tal forma que los nodos intermedios necesitan indicar qué propiedad se va a comparar (*split feature*), cuál es el valor para la elección del camino (*comparison value*), la dirección de sus dos hijos (*left child y right child*), si los tiene, y si el nodo es un nodo hoja o no (*is leaf*). Todo esto está representado de forma gráfica en la Figura 6.

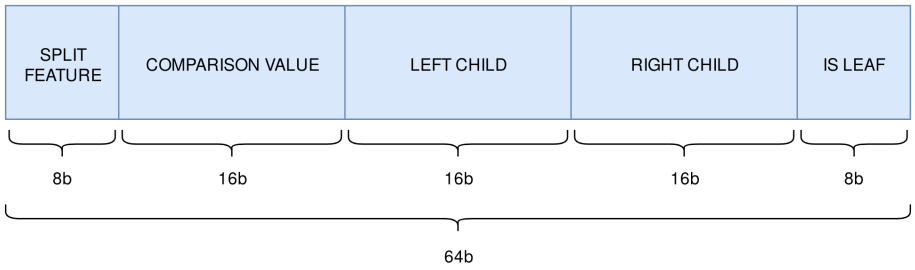


Figura 6: Nodo intermedio de un árbol

En cambio, los nodos hoja tienen una estructura diferente como se puede ver en la Figura 7. Contienen el resultado de la evaluación de dicho árbol (*prediction value*) además de la dirección del siguiente árbol a evaluar (*next tree*) si no se trata el último árbol de la clase (*last tree mark*), en caso contrario la evaluación de la clase ha finalizado y no se evalúan más árboles.

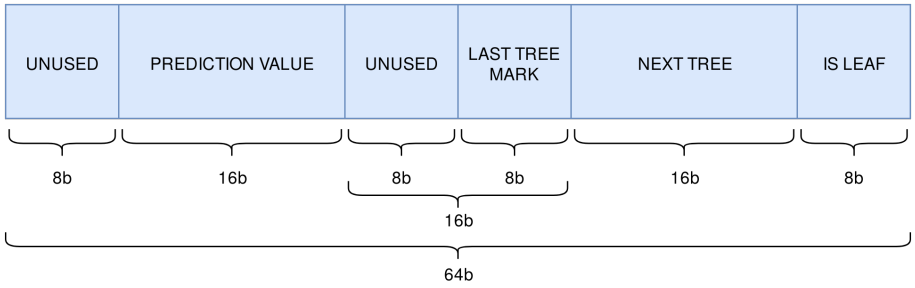


Figura 7: Nodo hoja de un árbol

En este punto, es muy importante explicar que toda la aritmética que se utiliza en las evaluaciónes es aritmética de enteros. Tanto el valor de comparación (*comparison value*) como el de la predicción (*prediction value*) son enteros. Poder utilizar los valores de comparación como enteros es posible gracias a que las propiedades de cada uno de los píxeles de una imágen hiperespectral son valores enteros, con lo cual no se pierde precisión en las comparaciones. En cuando a los valores de predicción se complica, ya que el *framework* LightGBM utiliza valores en coma flotante para realizar las predicciones. Por esta razón hace falta un tratamiento de estos datos antes de poderlos utilizar dentro de la FPGA. El tratamiento consiste en pasar los datos de coma flotante a coma fija de manera que se pierda la menor precisión posible y que el porcentaje de acierto siga siendo aceptable. Con esto en mente se ha decidido utilizar coma fija de 16 bits, de los cuales se utilizan 13 bits para la parte decimal, lo que se conoce como Q2.13. Haciendo estas dos modificaciones en

los datos originales se reduce el *hardware* necesario para la implementación y se aumenta considerablemente el rendimiento, pudiendo evaluar los árboles a razón de un nodo por ciclo.

En el caso de los píxeles, cada uno de ellos esta formado por 224 valores diferentes, uno por cada longitud de onda que el sensor ha sido capaz de identificar. Estos valores no son muy grandes, todos ellos están en el rango de los 16 bits, lo cual es muy conveniente ya que, dadas las especificaciónes del DMA mencionadas en la Sección 4.1, se pueden agrupar de dos en dos formando palabras de 32 bits, lo cual implica reducir a la mitad el número de ciclos necesarios para la transmisión de un píxel.

4.2.2 *Módulo para procesar una clase*

A lo largo del documento se ha comentado que los árboles de decisión se agrupan en estimadores o clases las cuales son capaces de identificar en qué medida el objeto de evaluación pertenece a una de las clases de la clasificación. A continuación se detalla cómo es posible realizar ésto utilizando un circuito digital como el que se puede ver en la Figura 8.

Para realizar el diseño se necesitan cumplir dos requisitos. Tener una memoria lo suficientemente grande para almacenar todos los árboles que puede contener una clase y tener una manera de poder evaluar estos árboles de una manera correcta y eficiente.

Tras un análisis de los recursos de almacenamiento de la FPGA, y del modelo obtenido tras el entrenamiento se ha decidido utilizar como almacenamiento una Random Access Memory (RAM)(*trees ram*) con 9 bits de direccionamiento, teniendo hasta 512 entradas de 64 bits, ya que cada una de las clases va a tener 100 árboles con 5 nodos de media en cada uno de ellos. Esto es así porque durante el entrenamiento, LightGBM busca tener resultados precisos utilizando muchos árboles pequeños en lugar de árboles muy grandes. Además de ésto, es necesario tener un registro que guarde la dirección con la que se indexa esta memoria (`reg0`) para poder leer y escribir en ella. Como se ha comentado al principio de esta sección, los módulos son parametrizables, con lo cual se puede ajustar el parámetro del tamaño de la memoria en función de los árboles que se necesite guardar.

En cuanto a la ejecución, se necesita un módulo para realizar las comparaciones entre el valor de la propiedad del nodo y la del objeto a evaluar además de un multiplexor para poder elegir entre los dos hijos del nodo. En caso de que se trate de un nodo hoja, se debe poder seleccionar directamente el campo donde se encuentra la dirección del siguiente árbol a evaluar, usando un multiplexor, así como un registro para acumular el valor de predicción de la hoja (`reg1`).

Para la gestión de las distintas señales de escritura y lectura se utiliza una unidad de control.

Como puede observarse en la Figura 9, cada una de las clases pueden encontrarse en uno de los siguientes estados: reposo (*IDLE*), carga de datos (*LOAD*), ejecución (*EXEC*) o finalización (*FINISH*).

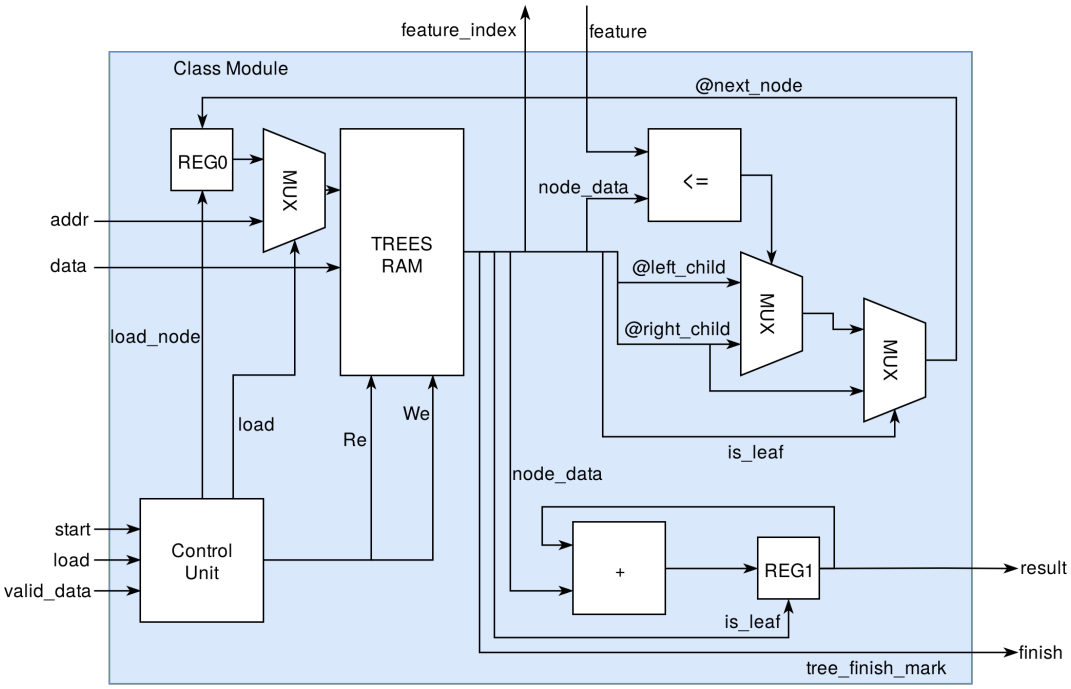


Figura 8: Diagrama del módulo que procesa una clase

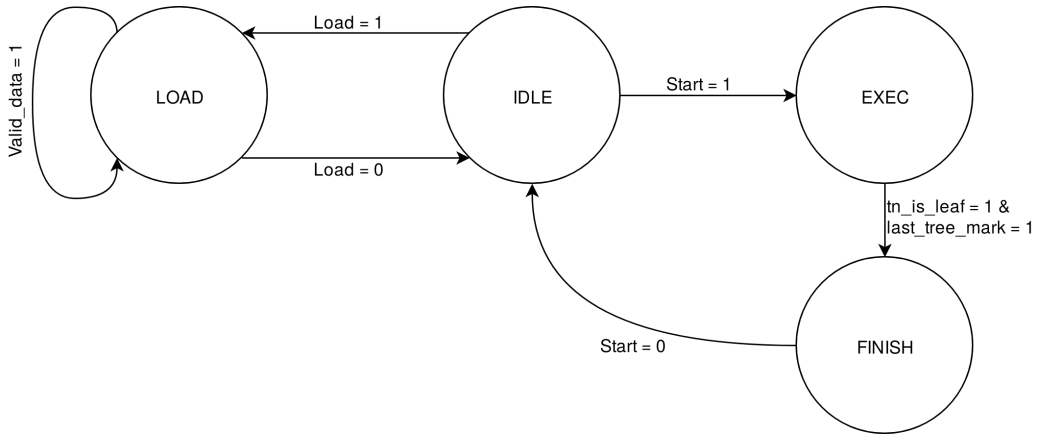


Figura 9: Diagrama simplificado de estados de la clase

Durante el estado *IDLE*, la clase está a la espera de cargar los datos de los árboles o de comenzar a evaluar los que ya tiene cargados. Como al iniciar el sistema la RAM se encuentra vacía, antes de comenzar a evaluar, es necesario pasar al estado *LOAD* y cargar los nodos de los árboles activando la señal *Load*. Desde este momento la RAM es direccionada desde el exterior de la clase, permitiendo cargar el dato proveniente por el bus *data* dentro de ésta cada vez que la señal *valid_data* esté activa. Dado que cada nodo se recibe a través del DMA, el cual tiene un tamaño máximo de palabra de 32 bits, y que cada uno de los nodos del árbol tiene un tamaño de 64 bits, un nodo estará disponible cada dos ciclos es por eso que se precisa de una señal aparte para indicar si el dato es valido para escritura o no. Este proceso de escritura en la RAM cada dos ciclos se mantiene hasta que todos los nodos se han transmitido. Al poner la señal *Load* a `0` se pasa de nuevo al estado *IDLE*.

Trás la carga de los datos, se puede pasar al estado *EXEC* y evaluar los árboles. Esto se hace cuando la señal *Start* toma valor `1`. La evaluación consiste en, como se ha explicado en la Sección 2.3, empezando por el nodo raíz, evaluar cada uno de los nodos eligiendo uno de los hijos en base al resultado de la comparación y guardando su dirección en el **`reg0`**, de esta forma será el siguiente en ser evaluado. Una vez se alcanza un nodo hoja, se activará la señal *is_leaf* permitiendo almacenar el resultado de las predicciónes en el registro resultado (**`reg1`**) y se carga, en este caso, la dirección del nodo raíz del siguiente árbol. La evaluación concluye cuando se encuentra una hoja que pertenezca al último árbol de la clase. En este punto, se acumula el valor de esta última hoja y se pasa al estado *FINISH*, en el que el resultado de la clase está listo para ser leído y utilizado fuera de ésta. Se volverá a pasar al estado *IDLE* cuando la señal *Start* se baje a cero de nuevo. Este proceso se repite hasta que todos los píxeles de la imágen hiperespectral hayan sido evaluados.

Utilizando este esquema de carga y ejecución independientes, se podrían dividir lo árboles de cada clase en lotes e ir cargándolos y ejecutándolos por partes en caso de que no fuese posible almacenar todos los árboles dentro del acelerador.

4.2.3 *Módulo principal del acelerador*

Todos los estimadores del modelo se agrupan dentro de una estructura más compleja, que se puede apreciar en la Figura 10, la cual se encarga del envío y recepción de datos a través del AXI, la evaluación de las clases, la selección del resultado de ésta mediante la aplicación de la función *argmax* a los resultados de las clases, la depuración y la verificación de integridad de los datos.

De igual forma que con las clases, esta estructura tiene una serie de estados predefinidos en los cuales puede encontrarse. Estos estados, que se pueden ver en la Figura 11, son: reposo (*IDLE*), carga de árboles (*LOAD_T*), carga de propiedades (*LOAD_F*), ejecución (*EXEC*) y dos estados para depuración, los cuales permiten leer una dirección arbitraria de la memoria de una clase (*CHECK_TREES*) o de la memoria del píxel (*CHECK_FEATURES*).

El proceso de carga de datos consiste en activar la señal *axi_load* y *axi_load_trees*, si lo que se cargan son árboles, o no activar esta última para cargar el píxel a evaluar. Una vez el autómata pase al estado deseado, se programa el DMA desde la CPU para enviar los datos de la memoria principal a la FPGA. Para la carga de los

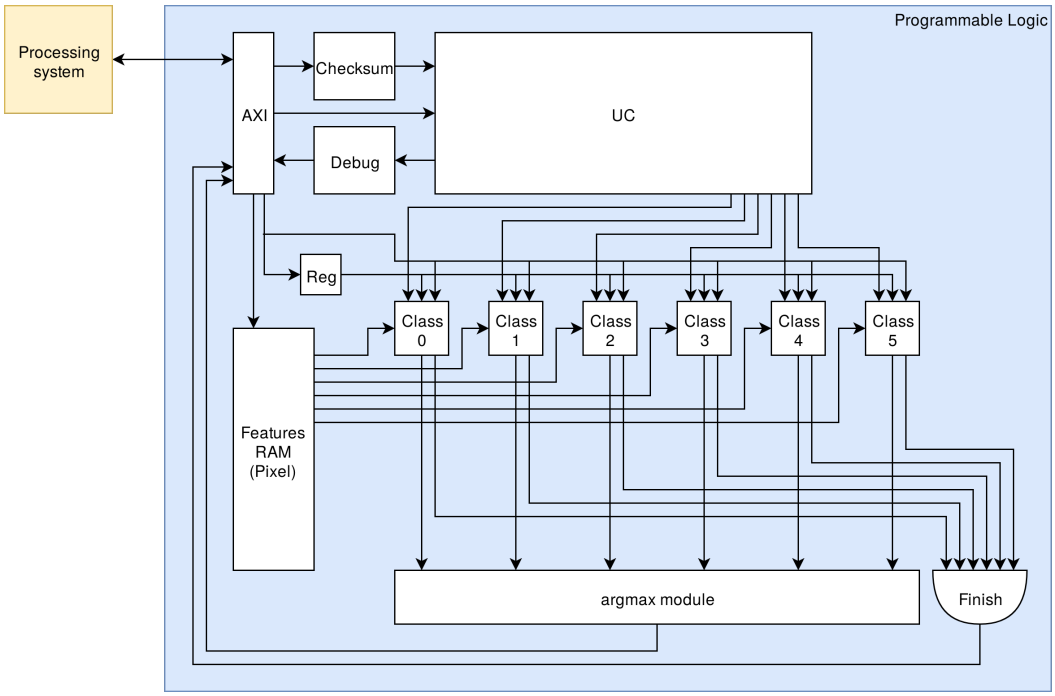


Figura 10: Diagrama del módulo principal del acelerador

árboles, el acelerador pasa al estado *LOAD_T* y, tras la programación e inicialización del DMA, los árboles llegan a razón de un nodo cada dos ciclos tal y como se ha comentado en la Subsección 4.2.2. Inicialmente, los nodos se escriben en la primera clase hasta encontrar un nodo marcado como último, lo cual provoca que los siguientes nodos se escriban en la siguiente clase. Este proceso se repite hasta encontrar un nodo marcado como último en la última clase, lo cual concluye la carga de árboles. El caso de la carga de propiedades del píxel es más sencillo dado que todas ellás se almacenan en la misma memoria. Además, a diferencia de los nodos, las propiedades solo ocupan 16 bits, lo cual hace que en cada envío se transmitan dos de ellas, reduciendo tanto el tiempo de envío como la gestión que conlleva el hecho de recibir un dato útil cada dos ciclos. Una vez se transmitan los datos, tanto de árboles como del píxel, se baja la señal *axi_load* provocando que el autómata pase a modo *IDLE* de nuevo.

Uno de los procesos más sencillos es la evaluación, ya que ésta se delega a cada una de las clases. Al activar la señal *axi_start*, el autómata pasa al estado *EXEC* y cada una de ellas comienza a evaluar los árboles que contiene indicando su finalización con la señal *finish* como ya se ha comentado en la Subsección 4.2.2. Al terminar todas las clases, se debe indicar que todas ellas han finalizado activando la señal *axi_finish* y escribiendo el resultado, ya calculado por el módulo *argmax*, en un registro de salida.

Tanto la depuración como la verificación de integridad se detalla en la Sección 4.3.

Todo el diseño que se puede ver tanto con la Figura 8 como en la Figura 10 requiere de 2961 *flip-flops* y 17533 LUTs. Un detalle importante de este diseño que se debe tener en cuenta es que las memorias RAM de las clases y las propiedades se traducen directamente en LUTs, no utilizando la memoria BRAM disponible en

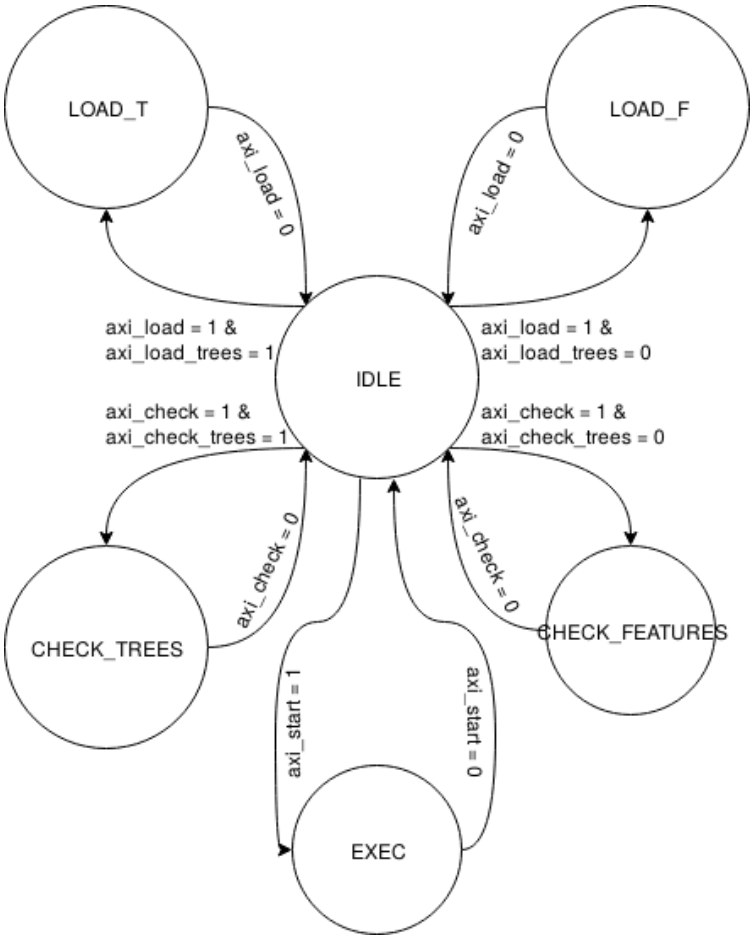


Figura 11: Diagrama simplificado de estados del acelerador

la FPGA. Con lo cual, 13110 de las 17533 LUTs son utilizadas como memoria y solamente 4423 son usadas como lógica. De esta forma, existe un gran margen de mejora en las siguientes iteraciones del diseño donde se utilicen estos bloques BRAM en lugar de las LUTs, incrementando la eficiencia energética y reduciendo el tiempo de operación. Por otro lado, la ventaja de usar LUTs es que el diseño se puede implementar en cualquier FPGA, incluyendo las que no dispongan de memorias BRAM.

4.3 DEPURACIÓN Y CONTROL DE ERRORES

Sin duda uno de los inconvenientes más importantes a la hora de trabajar con una FPGA es la falta de un modo de depuración. No disponer de una manera sencilla de depurar hace que diseñar sobre éstas sea sumamente complicado.

Como herremienta de depuración se ha desarrollado un programa en C que debe generar la misma salida que el módulo implementado en la FPGA, por lo que se puede verificar el correcto funcionamiento de éste. Además, los módulos desarrollados se han verificado mediante la simulación del código VHDL para distintos valores de entrada. Las herramientas de simulación del entorno de desarrollo permiten simular ciclo a ciclo el funcionamiento de todos los módulos del sistema. Permitiendo replicar lo que está ocurriendo dentro de la FPGA e identificar la causa de los errores.

En este proyecto en concreto es muy importante tener un método para comprobar que los datos transmitidos, tanto de los árboles como de los píxeles estén íntegros, tanto durante el desarrollo como en tiempo de ejecución. Aunque la memoria se centra principalmente en la descripción de los módulos que realizan el procesamiento de los datos, es importante destacar que en este proyecto también se han desarrollado los módulos que realizan las comunicaciones. Estos módulos también se pueden simular ciclo a ciclo, pero no es tan sencillo simular su interacción con el procesador y la red de interconexión. Por ello, antes de comprobar si el módulo realizaba los cálculos bien o no, debíamos estar seguro de que las comunicaciones hubiesen funcionado correctamente. Si se detecta un mal funcionamiento, lo primero que se debe comprobar es si los datos de los estimadores están guardados de forma correcta en cada una de las clases o si los datos del píxel han sufrido modificaciones.

Por esta razón se han implementado los estados *CHECK_T* y *CHECK_F* en el autómata del acelerador. El primero de ellos permite leer de manera arbitraria cualquier dirección de la memoria de cualquier clase. Con ésto se consigue un mecanismo para comprobar que los datos enviados se han mandado y almacenado de forma correcta. El segundo permite lo mismo, pero en este caso para la memoria donde se almacena el píxel.

Todas las lecturas precisan de un total de tres registros. En uno de ellos, el de control, se indica al acelerador que se pase a modo *check* y si se quiere acceder la memoria de una clase o del píxel. En caso de que se quiera leer la memoria de la clase, se debe indicar qué clase y si se quiere leer la parte alta de un nodo o la parte baja. Esto último es necesario porque como ya se ha explicado en la Subsección 4.2.1 los nodos de los árboles tienen un tamaño total de 64 bits pero los registros son de 32 bits, con lo cual para leer un nodo entero hacen falta dos lecturas. Además de estas señales de control, se necesita un registro para indicar qué dirección se quiere leer y otro donde se devuelva el dato que se ha leído.

Además de estos estados de depuración, en el diseño, se ha incluido un módulo de verificación de integridad de los datos que consta de un contador y un módulo de *checksum*, el contador mantiene la cuenta de los datos enviados a través del DMA de esta forma podemos saber que todos los datos enviados se han recibido. El módulo *checksum* suma todos los datos recibidos a través del DMA. De esta forma, leyendo dos registro dedicados para esta función, se puede comprobar tanto la cantidad de datos recibida en la FPGA como la integridad de éstos y sólo hace falta recurrir a los estados de depuración en caso de que se detecte un error al hacer estas dos lecturas.

4.4 IMPLEMENTACIÓN

En esta sección se van a detallar todos los pasos necesarios para llevar a cabo todo el proyecto. Estos pasos se pueden dividir en tres etapas: etapa de estudio y entrenamiento de los árboles, etapa de manipulación de los datos del modelo para su uso en la FPGA y por último, la fase de desarrollo y depuración del acelerador.

4.4.1 *Estudio y entrenamiento de los árboles de decisión*

Desde un principio se ha decidido utilizar el *framework* LightGBM para el entrenamiento de los árboles de decisión. Para ésto, se ha implementado un programa en Python que ha permitido obtener un modelo entrenado a partir de una imágen hiperespectral. Este programa está parametrizado de tal forma que se puede elegir cómo dividir los datos de entrada, en datos para entrenamiento, validación y test así como la elección por parte del usuario de cuántos árboles se quieren por cada clase, y cómo agrupar estos árboles para obtener un número determinado de nodos por árbol. Las características del modelo utilizado se tratan en más profundidad en el Capítulo 5.

Tras el entrenamiento se han exportado estos datos a un fichero JavaScript Object Notation (JSON) para su posterior procesamiento y adaptación para su utilización dentro de la FPGA. Esto se explica forma más detallada en la siguiente subsección.

Antes del tratamiento de datos se ha tenido que estudiar cómo se organizan los árboles en este fichero JSON. Cómo ya se ha mencionado en la Sección 2.3, el método de entrenamiento *Gradient Boosting* es un método de entrenamiento iterativo que genera un número variable de árboles por clase en función de las iteraciones. En este caso, en el fichero los árboles se organizan de la siguiente forma: el primer árbol que aparece en el fichero se ha generado en la primera iteración y forma parte de la primera clase, el siguiente es el primer árbol de la segunda clase generado en esta misma iteración. De esta forma, los árboles se agrupan por iteraciones en grupos de N árboles, donde N es el número de clases del modelo. Esto es muy importante para saber cómo analizar el fichero y adaptar los datos para la FPGA.

4.4.2 *Tratamiento de los datos del modelo para su uso en la FPGA*

Una vez se tiene el modelo entrenado y está exportado en un formato que se puede analizar y tratar, se debe adaptar para su uso en la FPGA. Todo el estudio de la organización de los datos en función de las necesidades del diseño tratadas en la Subsección 4.2.1 se lleva a cabo en este momento concreto.

El tratamiento de los datos de los árboles consiste en recorrer cada uno de ellos y convertir cada uno de los nodos en el formato descrito en la Figura 6 y la Figura 7. Para cada uno de los nodos se deben rellenar los campos *left child* y *right child* ya pre-calculados de forma que en la evaluación se pueda acceder a éstos fácilmente. También es en este punto cuando se convierten los datos de predicción de las hojas a coma fija y se marcan las hojas para saber cuáles son las últimas de cada clase. El tratamiento de los píxeles es más sencillo ya que solamente se deben agrupar las propiedades de dos en dos formando palabras de 32 bits.

Para llevar a cabo este proceso se ha creado un programa en Go. Este programa tiene como entrada el fichero JSON del modelo entrenado y el fichero con todos los píxeles. Como salida, para cada fichero de entrada se obtiene un fichero .h que contiene un vector de palabras de 32 bits con todos los árboles del modelo o los píxeles, además de éste, también hay una serie de *defines* que ayudan al control de flujo de los datos en su transmisión de la CPU a la FPGA. Este fichero es el que posteriormente se incluye en el programa que se ejecuta en la CPU.

4.4.3 *Desarrollo del acelerador*

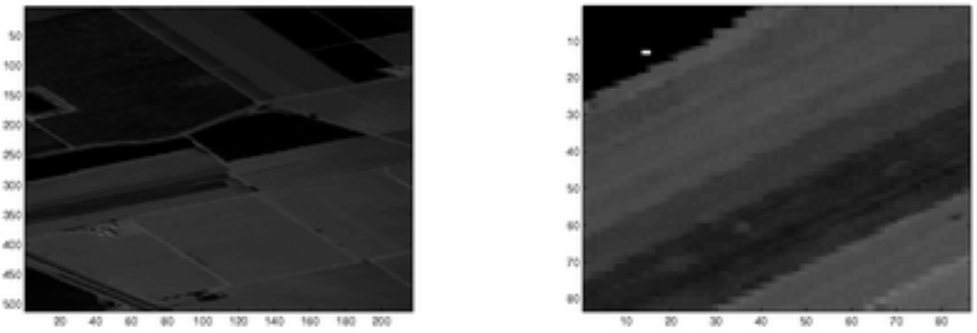
El desarrollo del acelerador ha consistido a su vez en varias etapas. La primera de ellas ha sido desarrollar el modelo de una clase, parecido al de la Figura 8, que albergara un solo árbol y que tuviera los datos de evaluación embebidos, de tal forma que no hiciera falta ningún tipo de transmisión. Primero, este modelo se ha probado utilizando el simulador del entorno analizando cada una de las señales para comprobar que todo funcionase correctamente. En una siguiente iteración del diseño se ha añadido la capacidad de poder utilizar árboles GBDT con todos los añadidos al diseño que esto conlleva (sumadores y multiplexores para poder enlazar con los siguientes árboles).

Al margen de la evaluación de los árboles se ha estudiado el uso del DMA desde la CPU ya que una parte importante del diseño es poder cargar los árboles dinámicamente. Es en este punto en concreto cuando se ha tomado la decisión de implementar un método de depuración mediante el cual se pueda leer cada una de las memorias RAM del acelerador y la implementación del módulo de *checksum* para la verificación de la integridad de los datos.

Todo el proceso de transmisión y carga de datos dentro de la FPGA ha sido el más difícil dada la poca experiencia de la que se disponía realizando este tipo de implementaciones. Es muy importante tener en cuenta todas las señales de control que deben activarse, las del DMA, las de escritura en las memorias RAM, las de direccionamiento de cada clase. Además todas ellas deben estar totalmente coordinadas ya que, el hecho de que una sola señal se desfase resulta en que ninguno de los datos se han escrito o lo que es peor, que se escriban en posiciones de memoria no deseadas, lo que hace la depuración sumamente difícil en estos casos.

Como última etapa se ha desarrollado un programa en C que realiza los mismos cálculos que el acelerador ejecutado en la FPGA. Gracias a este programa se ha podido comprobar que el acelerador se comporta correctamente al procesar los 2276 píxeles. Además, este programa se ha utilizado para la comparación tanto de tiempos como de consumos ya que se podía ejecutar en uno de los Cortex-A9 incluidos dentro del SoC de la placa de desarrollo.

Para la experimentación, el modelo GBDT ha sido entrenado utilizando una parte de una imágen hiperespectral del valle Salinas, California [1]. Se han utilizado un 15 % de los píxeles de cada clase para el entrenamiento, dividiendo el resto de datos en validación y test, de tal forma que se han utilizado 2276 píxeles para test. El modelo se ha entrenado usando 100 iteraciones utilizando el *framework* LightGBM [4], lo cual ha generado 100 árboles por clase consiguiendo una precisión del 97 % en las clasificaciones.



(a) Salinas (b) Parte de la escena Salinas: SalinasA

Figura 12: Imágenes hiperespectrales del valle Salinas, California

Se han comparado el tiempo de ejecución y el consumo de poténcia y energía de una predicción completa, utilizando todos los píxeles de test, utilizando un programa C ejecutado sobre la CPU y el diseño mostrado sobre la FPGA.

Todos los experimentos se han realizado sobre la plataforma ya descrita en la Sección 4.1. Los resultados se pueden observar la Tabla 1. Se muestra el tiempo total de la predicción para cada uno de los escenarios, añadiendo también los diferentes niveles de optimización del programa C. La CPU se ejecuta a una frecuencia máxima de 667MHz mientras que la FPGA funciona a 100MHz. Como se puede apreciar, la ejecución de la FPGA consigue un *speedup* de 8.8 sobre la ejecución de la CPU con una optimización O3. En este caso, las medidas de la FPGA incluyen también el tiempo necesario para enviar cada uno de los píxeles desde la memoria principal hasta la FPGA, lo cual implica que 7.479ms de los 15.277ms se utilizan solamente para este envío mientras que solo 7.798ms para la clasificación. Este tiempo se podría reducir considerablemente si en vez de utilizar un diseño secuencial se utilizara una arquitectura en *pipeline* que permitiera evaluar un dato en paralelo con el envío del siguiente dato a procesar, ocultando de esta forma la latencia de las comunicaciones.

El consumo energético se ha medido utilizando un medidor digital de consumo de la marca Yokogawa, en concreto el modelo WT210 [12]. Primero se ha medido y obtenido la media de consumo estático del entorno en estado de reposo, 3.1W de valor medio, para poder restarlo del consumo total. De esta forma todos los

Tabla 1: Comparación de tiempos de ejecución y consumo energético.

	CPU(-O0)	CPU(-O1)	CPU(-O2)	CPU(-O3)	FPGA
Time (ms)	806.910	189.942	134.001	133.980	15.277
Power (W)	1.436	1.436	1.436	1.436	1.453
Energy (J)	1.159	0.273	0.192	0.192	0.022

datos de consumo presentes en la Tabla 1 solamente tienen en cuenta el consumo energético dinámico, es decir el derivado de la ejecución del programa.

La Figura 13 presenta el consumo de potencia medido con el vatímetro durante un periodo de tiempo. Como se puede observar este consumo es muy similar tanto en la versión con acelerador, como en la versión en la que la CPU realiza los cálculos. Por tanto si ambos consumen la misma potencia, el ahorro en consumo de energía debido al acelerador será proporcional al tiempo de ejecución. De hecho, el consumo se reduce en un factor d 8.7 con respecto a la ejecución en CPU con un nivel de optimización O3.

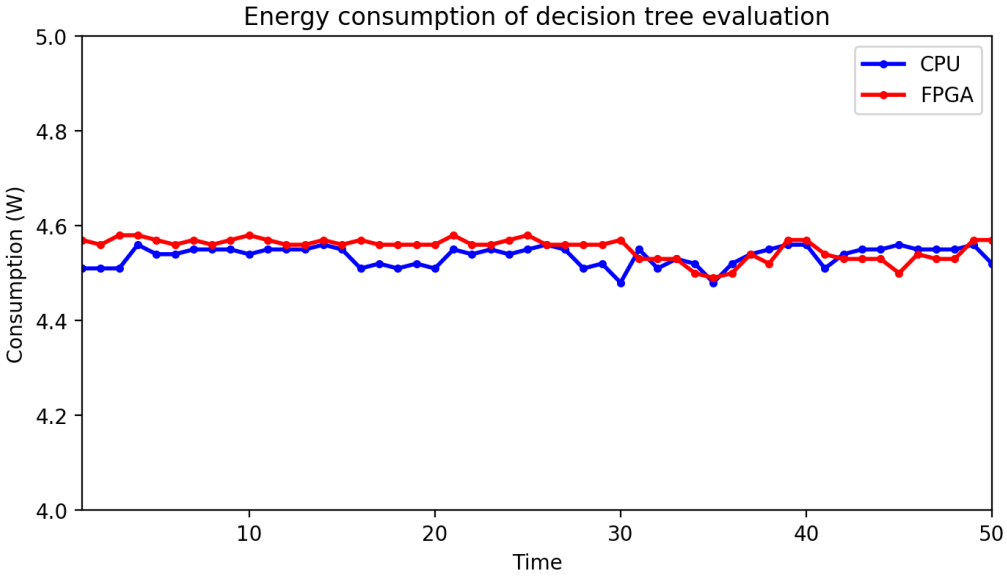


Figura 13: Comparación de consumo energético

CONCLUSIONES

Con este proyecto se ha demostrado que el uso de una FPGA para la implementación de un acelerador de árboles de decisión no solo es posible, sino que se pueden obtener unos resultados magníficos tanto de tiempos de ejecución como de consumo energético al comparar los resultados con los que se obtienen al hacer los mismos cálculos en un procesador convencional.

Dado que tanto la CPU como la FPGA utilizadas se encuentran en el mismo SoC estos dos elementos están implementados utilizando la misma tecnología, lo que hace que estas comparaciones sean totalmente justas.

Además hay que recordar que el procesador trabaja a una frecuencia de reloj muy superior (667 vs 100 MHz), y que se ha tratado de mejorar el rendimiento del código C con las opciones de compilación. También hay que señalar que como desarrollador dispongo de mucha más experiencia en el desarrollo de código C que en la descripción de *hardware* utilizando VHDL. Por lo que considero que los resultados obtenidos han sido muy satisfactorios. Estos resultados se han recogido en una contribución enviada al *HiPEAC – 2020 Workshop on Accelerated Machine Learning (AccML)* [3] que se celebrará el próximo 20 de enero en Bologna. El artículo enviado se incluye como anexo.

Naturalmente, existe mucho margen de mejora. En primer lugar se podría modificar el diseño para que utilice los recursos de memoria BRAM en vez de los LUTs para el almacenamiento de los árboles y las propiedades. Este cambio podría tener un impacto positivo en el consumo de energía y mejorar también el retardo del sistema. También se puede aumentar el nivel de paralelismo, para ello bastaría con dividir los árboles en varios subconjuntos que se podrían ejecutar en paralelo. Por último, analizando los tiempos de ejecución, se puede ver que aproximadamente la mitad del tiempo se utiliza en mandar los datos a evaluar y no a evaluarlos. Modificando el diseño para poder superponer las etapas de envío de datos y la evaluación de estos supondría una gran mejora también.

BIBLIOGRAFÍA

[1] GIC. (). Hyperspectral Remote Sensing Scenes, Grupo de Inteligencia Computacional de la Universidad del País Vasco, dirección: http://www.ehu.eus/ccwintco/index.php/Hyperspectral_Remote_Sensing_Scenes#Salinas_scene (visitado 14-10-2019).

[2] A. Géron, *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Inc, mar. de 2017, ISBN: 9781491962282.

[3] HiPEAC. (). Accelerated Machine Learning (AccML), dirección: http://workshops.inf.ed.ac.uk/accml/ (visitado 07-11-2019).

[4] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye y T.-Y. Liu, «LightGBM: A Highly Efficient Gradient Boosting Decision Tree», en *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan y R. Garnett, eds., Curran Associates, Inc., 2017, págs. 3146-3154. dirección: http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf.

[5] R. Kułaga y M. Gorgon, «FPGA Implementation of Decision Trees and Tree Ensembles for Character Recognition in Vivado Hls», *Image Processing & Communications*, vol. 19, sep. de 2014. DOI: 10.1515/ipc-2015-0012.

[6] R. Narayanan, D. Honbo, G. Memik, A. Choudhary y J. Zambreno, «An FPGA Implementation of Decision Tree Classification», en *2007 Design, Automation Test in Europe Conference Exhibition*, 2007, págs. 1-6. DOI: 10.1109/DATE.2007.364589.

[7] F. Saqib, A. Dutta, J. Plusquellic, P. Ortiz y M. S. Pattichis, «Pipelined Decision Tree Classification Accelerator Implementation in FPGA (DT-CAIF)», *IEEE Transactions on Computers*, vol. 64, n.º 1, págs. 280-285, 2015. DOI: 10.1109/TC.2013.204.

[8] V. Teletin. (2019). Modulo del acelerador de árboles de decisión, dirección: https://github.com/ChromaMaster/tfg_decision_trees_container.

[9] ——, (2019). Parser del modelo entrenado para usarlo en el acelerador, dirección: https://github.com/ChromaMaster/tfg_tree_to_fpga.

[10] Xilinx. (). AMBA AXI4 Interface Protocol, dirección: https://www.xilinx.com/products/intellectual-property/axi.html (visitado 07-11-2019).

[11] ——, (). Zynq-7000, SoC and FPGA platform, dirección: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf (visitado 07-11-2019).

[12] Yokogawa. (). WT210/WT230 Digital Power Meters, dirección: http://tmi.yokogawa.com/products/digital-power-analyzers/digital-power-analyzers/wt210wt230-digital-powermeters/#tm-wt210_01.htm (visitado 07-11-2019).