Universidad Zaragoza

Trabajo Fin de Grado

Programación y optimización de sistemas heterogéneos basados en FPGA

Programming and optimization of heterogeneous systems based on FPGA

Autor

Pablo Cambra Acín

Directores

Rubén Gran Tejero

Darío Suárez Gracia

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2023

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe remitirse a seceina@unizar.es dentro del plazo de depósito)

D./Dª. Pablo Cambra Acín ,

en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de Estudios de la titulación de Grado en Ingeniería Informática (Título del Trabajo)

Programación y optimización de sistemas heterogéneos basados en FPGA

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 31/08/2023

CAMBRA ACIN PABLO - 73429748V

Firmado digitalmente por CAMBRA ACIN PABLO - 73429748V
Fecha: 2023.08.31 17:30:29 +02'00'

Fdo: Pablo Cambra Acín

AGRADECIMIENTOS

A mis tutores, Darío y Rubén, por guiarme y ayudarme en la realización de este trabajo y resolver las dudas que tenía.

A mi familia, por apoyarme cuando tenía dificultades y no progresaba en este proyecto.

Y por último a Denis Navarro, por dejarnos utilizar la máquina del I3A.

Índice

Capítulo 1

Introducción

1.1. Motivación

En informática, hay tipos de problemas que por su estructura o naturaleza, no se adaptan bien a los procesadores de propósito general, haciendo que su resolución no sea eficiente en términos de tiempo y tampoco de energía, aspecto tan importante hoy en día.

Para mejorar estos problemas, se empezaron a utilizar aceleradores dedicados funcionando junto al procesador de propósito general, para que fuera el acelerador el que se encargara de resolver el problema, dejando al procesador de propósito general la tarea de controlar el acelerador.

Estos aceleradores aportan numerosas ventajas con respecto a los problemas mencionados anteriormente, y es que al ser específicos pueden obtener una mayor eficiencia energética a la vez que un mayor rendimiento que un procesador de propósito general.

Entre los aceleradores de los que se disponen, la FPGA se presenta como una alternativa interesante en términos de rendimiento y eficiencia energética, aunque no tanto con respecto a la programabilidad, la cual es baja. Esto se podrá comprobar más adelante.

1.2. Objetivos

Este TFG pretende abordar la problemática que hay en torno a la programación de las FPGAs para resolver dichos problemas, y cómo herramientas actuales nos permiten realizar dicha tarea con un nivel de abstracción mayor, obteniendo con ello una facilidad y velocidad en el desarrollo superior.

También abordaremos el uso de la memoria de la manera óptima para alcanzar el mayor rendimiento, ya que en problemas livianos, esta suele ser el limitante.

Por último, se abordarán las diferencias entre utilizar un lenguaje de alto nivel para desarrollar código o un lenguaje de descripción de hardware.

1.3. Alcance

El alcance de este TFG, con respecto a los problemas de programabilidad se han cumplido, ya que durante la elaboración del mimso se ha podido comprobar la veracidad de estas afirmaciones.

También se ha cumplido el segundo objetivo, ya que se ha realizado una labor de investigación y experimentación con respecto al uso de la memoria.

El tercer objetivo no se ha cumplido tanto por conseguir desarrollar código, sino por comparar un mismo código ya desarrollado con ambas alternativas y las diferencias y dificultades a la hora de realizar modificaciones sobre estos.

Hay que añadir que nunca se había trabajado con FPGAs, ni con las herramientas que nos permiten diseñar código para estas, y al ser específicas y complicadas no se ha llegado todo lo lejos que se hubiera querido.

En el anexo 1 podemos observar la organización y el tiempo de las diferentes tareas realizadas durante este TFG.

1.4. Descripción de este documento

En el capítulo 2 hablaremos de los diferentes aceleradores que existen, como se puede desarrollar código para ellos y las herramientas que nos lo permiten. En concreto nos centraremos en las FPGAs, sus problemas, limitaciones y las herramientas que tenemos para intentar subsarnarlos.

En el capítulo 3 hablaremos de la memoria HBM2 en una FPGA y su estructura, de las interfaces que se utilizan para interconectar la memoria con el dispositivo y cómo utilizarla de manera eficiente para obtener un mayor rendimiento.

En el capítulo 4 hablaremos del entorno donde se han desarrollado los experimentos, tanto el hardware como el software utilizados y del flujo de trabajo seguido.

En el capítulo 5 hablaremos de los experimentos realizados, explicando que se quería obtener con ellos y posteriormente conclusiones comparando los resultados esperados con los finalmente obtenidos. En concreto nos vamos a centrar en diferentes experimentos para tratar de comparar el rendimiento utilizando diferentes tipos de datos, diferentes organizaciones de los datos en los distintos bancos de memoria, y una comparación del mismo código implementado utilizando lenguajes de descripción de hardware (HDL) y un lenguaje de alto nivel (HLS).

Por último, en el capítulo 6 hablaremos sobre las conclusiones que se han obtenido realizando este trabajo.

Capítulo 2

Estado del arte

Una vez que el escalado de Moore parece que ha llegado a su fin [1], se han propuesto arquitecturas alternativas a los procesadores de propósito general para seguir aumentando el rendimiento de los computadores. Muchas de estas propuestas se basan en el uso de un procesador de propósito general junto con uno o varios dispositivos de cómputo específico, o aceleradores, que colaboran entre sí.

En este capítulo vamos a hablar de los diferentes tipos de aceleradores, de la programación que nos encontramos en cada uno de ellos y por último de los entornos y herramientas de programación que tenemos en cada uno de ellos.

2.1. Tipos de aceleradores

En los sistemas heterogéneos disponemos de varios tipos de dispositivos aceleradores. A continuación, vamos a describir que tipos existen y sus principales características.

2.1.1. ASIC

Como primera opción, tenemos los ASICs, del inglés Application-Specific Integrated Circuit, que son dispositivos que disponen un hardware específico para desarrollar una tarea concreta. Estos dispositivos ofrecen un muy buen rendimiento, con el punto negativo de que su hardware no es programable, por lo que solo podrán realizar la tarea para la que han sido diseñados. Estos dispositivos son interesantes en entornos en los que no hay variabilidad con el código, ya que son los más eficientes y que más rendimiento obtienen, aunque también son los dispositivos más caros.

2.1.2. FPGA

Como segunda opción, tenemos las FPGAs, que son dispositivos basados en bloques lógicos programables, también llamados CLBs a partir de ahora, que pueden ser modificados vía programación para realizar una tarea u otra. Estos CLBs están

interconectados entre sí por IOBs, que sirven de buses de datos y señales entre los distintos bloques.

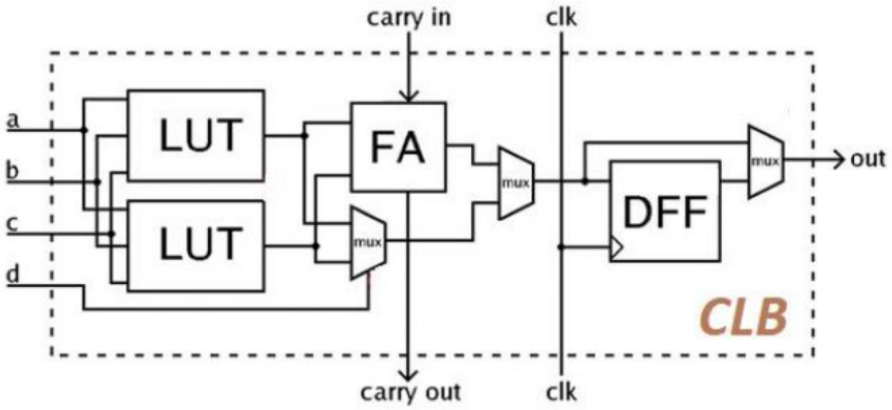

Figura 2.1: Estructura de un CLB (akka-technologies)

En la figura 2.1 se pueden ver los distintos elementos lógicos programables que componen cada CLB. A continuación vamos a pasar a describir cada elemento que componen los CLBs y que función realizan.

Por un lado tenemos las look-up tables (LUT), que son tablas de búsqueda programables y que para entrada tienen una salida definida. Estas tablas se pueden utilizar por ejemplo para implementar diferentes operaciones como una suma, o puertas lógicas.

Por otro lado tenemos los multiplexores, que nos sirven para elegir entre múltiples valores de entrada.

Por otro lado tenemos los Flip Flops D (DFF), que sirven para guardar el estado del circuito durante varios flancos de reloj.

Por último tenemos los Full Adders (FA), que sirven para sumar 2 señales binarias.

Las FPGAs se situan un paso por delante de los ASICs en programabilidad como hemos visto, aunque obtienen un rendimiento y eficiencia inferiores. Estos dispositivos se vuelven interesantes en entornos de investigación o desarrollo, donde el comportamiento cambia con frecuencia.

Para este apartado se ha tomado como referencia la web akka-technologies [2].

Tenemos 2 principales fabricantes de FPGAs, los cuales son Intel [3] y Xilinx [4]. Cada fabricante tiene sus propias herramientas para desarrollar código tanto en RTL como en HLS. Intel dispone de OneAPI [5] para HLS y Quartus [6] para RTL. Mientras tanto Xilinx nos proporciona Vitis [7] para HLS y Vivado [8] para RTL. Estas dos herramientas funcionan en conjunto utilizando Vitis. Ambos fabricantes proporcionan funcionalidades parecidas, cada una compatible con sus dispositivos.

2.1.3. GPU

Por último, tenemos las GPUs, cuyo hardware no es programable. En ese sentido estarían cerca de los ASICs debido a que su hardware es fijo, aunque tienen un modelo de cómputo parecido al de una CPU. La diferencia entre ambos es que las GPUs tienen núcleos mucho más sencillos pero cuentan con un gran número de estos, por lo que se hacen interesantes para tareas de aceleración.

Estos dispositivos son los más programables y más extendidos a nivel general, debido a su popularidad en el procesamiento de imágenes como por ejemplo en videojuegos. También es una ventaja su modelo de computación más parecido al de una CPU.

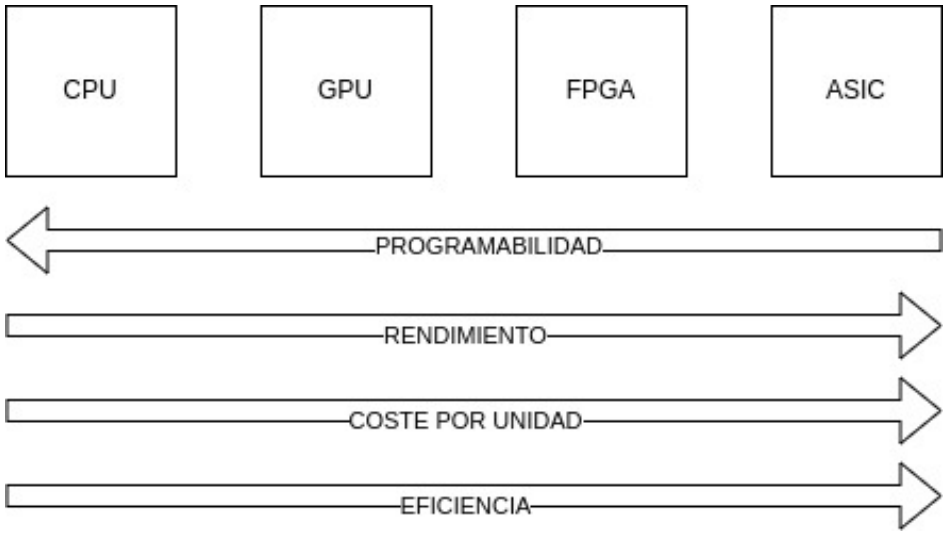

Figura 2.2: Clasificación de los diferentes aceleradores

En la figura 2.2 podemos observar la relación que hay entre los diferentes tipos de aceleradores en términos de rendimiento, programabilidad, eficiencia y coste. Siendo los ASICs los más eficientes pero más costosos y menos programables y las CPUs convencionales las más baratas y programables pero las que menos rendimiento y eficiencia ofrecen.

2.2. Programación de los aceleradores

El desafio principal de los aceleradores es su programación, ya que suele resultar muy costoso obtener grandes rendimientos, sin embargo esta no es igual para todos los tipos de aceleradores que hemos nombrado.

En este apartado se centra de la programación en FPGAs, por ser el tipo de dispositivo elegido en este TFG, y de la programación de GPUs. De los ASICs no vamos a hablar, por su incapacidad de ser programados mencionada anteriormente.

2.2.1. FPGA

Las FPGAS tradicionalmente se programaban en RTL como podían ser HDL o Verilog. Estos lenguajes de descripción de hardware le indican al hardware como tiene que comportarse. El mayor punto negativo de estos lenguajes es su dificultad y lentitud para desarrollar código para los dispositivos debido a lo específicos que son.

Posteriormente, para tratar de agilizar estas tareas, se añadió un paso intermedio, en el que el programador desarrolla el código en un lenguaje de alto nivel y es el compilador el que se encarga de traducir el código en alto nivel a RTL, para que finalmente pueda ser programado en la FPGA. Este nivel superior se conoce como High Level Synthesis(HLS).

Dentro de HLS, tenemos varias alternativas como pueden ser C/C++ u OpenCL, aunque en la actualidad hay alternativas para muchos lenguajes de propósito general.

Utilizando HLS hay que tener en cuenta que no en todos los casos el compilador va a ser capaz de sintetizar el código en alto nivel de una forma óptima y será tarea del programador tratar de optimizarlo más si es necesario.

```
module Acumulador (
  input wire clk,
  input wire reset,
  input wire [7:0] vector [0:7],
  input wire [3:0] size,
  output wire [7:0] acumulado
);

  reg [7:0] acc;

  always @(posedge clk or posedge reset) begin
    if (reset) begin
      acc <= 0;
    end else begin
      for (integer i = 0; i < size; i = i + 1) begin
        acc <= acc + vector[i];
      end
    end
  end

  assign acumulado = acc;

endmodule
```

Listing 2.1: Ejemplo de código de un acumulador en RTL

```
std::vector vector = ...;
float acc = 0.0f;

for(int i = 0; i < size; i++){
acc += vector[i];
}
```

Listing 2.2: Ejemplo de código de un acumulador en C++

En el listing 2.1 podemos ver las grandes diferencias que hay tanto en términos de longitud de código, como en complejidad del mismo. Podemos ver la cantidad de señales que tenemos, así como la estructura del bucle, que es mucho más compleja. Por otro lado, tenemos el listing 2.2 donde se puede ver el mismo código pero programado en C++. Se puede observar la menor complejidad tanto en tamaño como en facilidad de lectura. Esto también es debido a que los programadores están mucho más familiarizados con lenguajes de alto nivel que tienen una sintaxis parecida.

A todo esto tenemos que sumar el hecho de que cuando se genera código en RTL para una FPGA, también se generan toda una serie de ficheros relacionados con las interfaces de memoria, lo cual hace que el código no sea tan sencillo como el mostrado en el ejemplo 2.1 y dificulte aún más su comprensión.

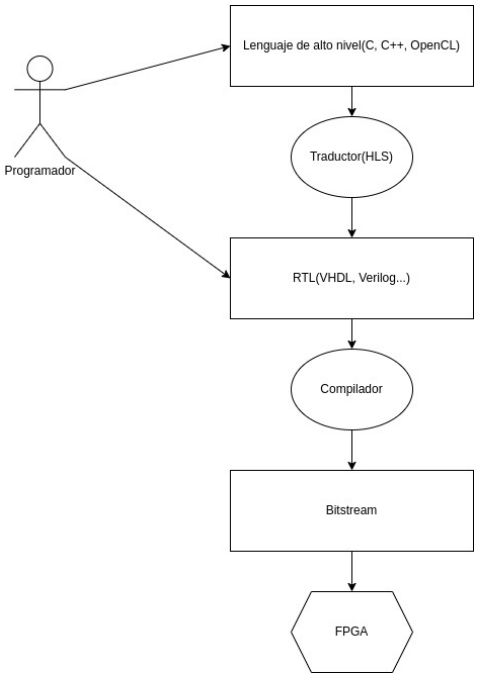

Figura 2.3: Pasos que sigue un kernel hasta ejecutarse en una FPGA

En la figura 2.3, de elaboración propia, podemos ver los pasos que sigue el código desde que es desarrollado por el programador hasta que se programa en la FPGA. Podemos ver que hay dos alternativas como hemos comentado anteriormente y el programador puede elegir diseñar el código directamente en RTL, ahorrándose el primer paso del proceso.

El proceso completo consiste en:

- El código en alto nivel es traducido por el compilador a RTL.
- El código RTL se convierte en un bitstream, un archivo binario.
- El archivo se carga en la FPGA para su ejecución.

En definitiva, la programabilidad de las FPGAs es mala, por el hecho de que en ocasiones nos veamos obligados a desarrollar el código en RTL, como hemos mencionado anteriormente. También nos encontramos que compilar código para estos dispositivos es muy lento, del orden de horas, ya que tiene que programar el hardware del dispositivo, y realizar cambios a una gran velocidad no es posible si no se utiliza emulación. Por último, los errores de ejecución no son fácilmente depurables, por lo que se puede hacer complicado encontrar un fallo en el código.

2.2.2. GPU

Las GPUs se programan siguiendo el mismo concepto de paralelizar las tareas lo máximo posible para sacar el máximo provecho de la gran cantidad de núcleos que tienen estos dispositivos. La gran diferencia es que al tener núcleos convencionales, no es necesario programar el hardware en sí.

Las GPUs tienen una mejor programabilidad ya que utilizan el modelo de programación Von Neumann, como las CPUs. Esto hace que la programación sea igual a la de un procesador convencional utilizando paradigmas de paralelización para aprovechar al máximo el gran número de núcleos.

2.3. Entornos de programación

2.3.1. FPGA

A continuación nos vamos a centrar en los entornos de programación Vitis [7] y OneAPI [5].

Por un lado OneAPI [5] es compatible con sus procesadores y FPGAs. Utlizando HLS es compatible con lenguajes de programación como C++, Fortran, Python y Data Parallel C++ (DPC++), que es una extensión de C++ de programación paralela. Gracias a la plataforma Quartus [6] también se puede desarrollar código en HDL como puede ser Verilog. OneAPI ofrece librerías más extensas adaptadas a diferentes trabajos, ya que es compatible con mayor cantidad de hardware.

Por otro lado Vitis [7] está pensada para aprovechar las FPGAs de Xilinx. Esta herramienta está basada en OpenCL y permite generar kernels tanto en HLS (C++) como en RTL utilizando Vivado [8].

2.3.2. GPU

Existen 2 fabricantes principales de GPUs, NVIDIA [9] y AMD [10], cada una con su propia plataforma para generar código para sus dispositivos.

NVIDIA utiliza CUDA [11], que es un modelo de programación escrito en C/C++, pero que tiene compatibilidad con otros lenguajes como Fortran o Python.

Mientras tanto, AMD, con ROCm [12], permite utilizar múltiples paradigmas para paralelizar el código, como OpenMP u OpenCL, la herramienta en la que está basada Vitis para generar los kernels. También admite Python. Una característica importante es que dispone de una plataforma de programación llamada HIP [13], que ofrece un nivel de abstracción para que los programadores puedan desarrollar su código para las GPUs pensando en la portabilidad entre los distintos fabricantes.

Capítulo 3

Memoria HBM2 en FPGAs de Xilinx

En este capítulo vamos a hablar de la memoria de las FPGAs Xilinx, de su estructura, limitaciones y de las interfaces que utilizan las FPGAs para interconectar diferentes componentes entre sí.

Para el tipo de problemas que se resuelven con las FPGAs es muy importante gestionar correctamente los accesos a memoria, y para ello es importante conocer su estructura y limitaciones. En concreto, la Alveo U280, que es el modelo con el que hemos trabajado, cuenta con memoria HBM2, que obtiene un ancho de banda teórico cuando se agregan todos los bancos de 460 GB/s. El ancho de banda real que se consigue es de 420 GB/s.

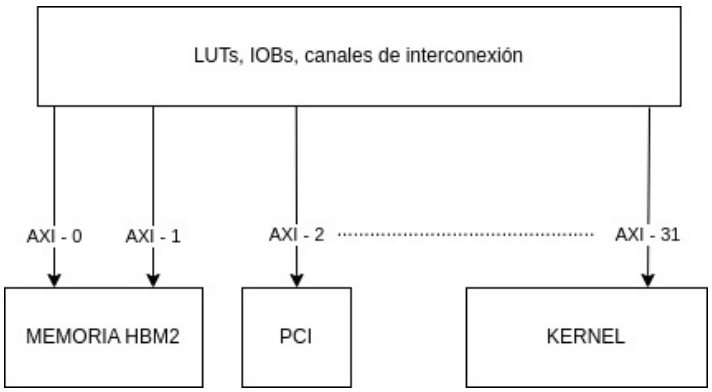

Figura 3.1: Ejemplo de interconexión de la FPGA con diferentes componentes

En la figura 3.1 podemos ver un esquema general de como diferentes componentes de la FPGA y dispositivos se conectan entre sí. Para ello se hace uso de unas interfaces multipropósito llamadas AXI (Advanced eXtensible Interface), que fueron diseñadas por ARM [14]. Estas interfaces pueden ser usadas para conectar la memoria, dispositivos PCI o inclusive otros kernels para transferir datos entre varios de ellos. A continuación vamos a pasar a describir en mayor detalle tanto las interfaces AXI como la estructura

de la memoria de la FPGA.

3.1. Interfaces AXI

Las interfaces AXI4 se utilizan como se ha visto en la figura 3.1 para conectar el hardware con diferentes dispositivos de la FPGA. Disponemos de 32 canales AXI. Las interfaces AXI tienen un ancho de 512 bits, y disponemos de varios tipos de ellas, de las cuales vamos a explicar sus características a continuación:

- AXI4-Lite, pensada para operaciones de memoria que no necesiten un gran ancho de banda, como señales de control o registros.
- AXI4-Full, pensada para operaciones que necesitan una gran cantidad de ancho de banda.
- AXI4-Stream, pensada para operaciones de stream de gran ancho de banda, principalmente utilizado para comunicar diferentes kernels entre sí.

Para nuestro trabajo se van a usar las interfaces AXI4-Full para los datos de entrada y de salida, como son vectores con los que vamos a operar y las interfaces AXI4-Lite, para señales de control como puede ser el tamaño de los vectores.

Uno de los principales dispositivos que interconectan estas interfaces AXI, es la memoria de la FPGA, que tiene un ancho de 256 bits, por lo que no estaremos aprovechando toda la interfaz, aunque debido a que estas interfaces han sido diseñadas para conectar una gran variedad de dispositivos, se decidió implementar un ancho de 512 bits, aunque para la memoria se esté desaprovechando.

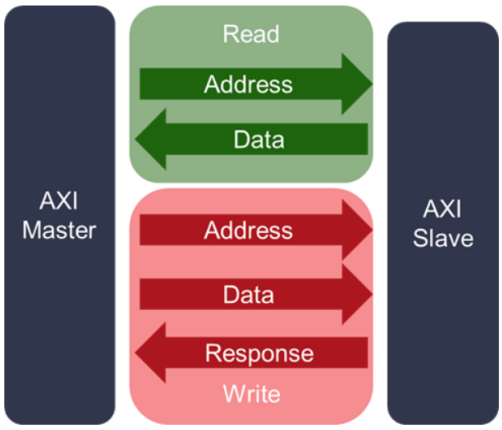

Figura 3.2: Canales interfaz AXI

Cada interfaz AXI, tiene 3 canales para la escritura y 2 canales para la lectura, como se puede ver en la figura 3.2. Tenemos una excepción la interfaz AXI4-Lite, que

como solo puede ser de lectura o de escritura, pero no ambas a la vez, solo tenemos los 3 canales de escritura o los 2 de lectura.

Para las lecturas, el master envía la dirección que quiere leer junto a varias señales de control y el esclavo responde con el dato por el otro canal.

Para las escrituras, el master envía la dirección donde quiere escribir junto con las señales de control. Acto seguido envía el dato que quiere escribir y espera a que el esclavo envíe la respuesta para comprobar si se ha escrito bien o ha habido algún tipo de error.

Para agilizar las lecturas y escrituras, podemos hacer que cuando se quiera leer o escribir, a la hora de mandar los datos, se envíen varios seguidos sin necesidad de volver a enviar la dirección, para leer o escribir en direcciones consecutivas. Esto se denomina lectura y escritura en ráfaga o burst.

3.2. Memoria HBM2

La memoria HBM2 es una memoria de gran ancho de banda, utilizada en dispositivos de alto rendimiento aunque muy costosa. A continuación vamos a pasar a describir sus caracterísitcas y su organización en una FPGA de Xilinx.

3.2.1. Estructura interna de un stack HBM2

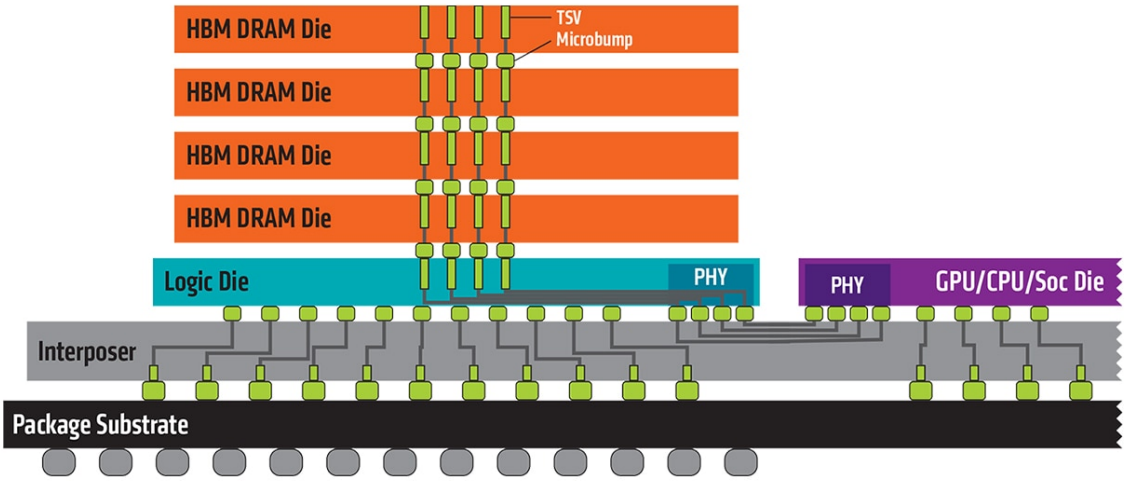

Figura 3.3: Estructura de un stack HBM2

Esta memoria se organiza apilando los chips unos encima de otros como se puede ver en la figura 3.3, formando stacks de memoria e interconectandolos con el dispositivo mediante la interfaz AXI mencionada anteriormente, colocando un componente encima de otro, consiguiendo una distancia entre componentes mucho menor que en una memoria DDR convencional, reduciendo con ello latencia y aumentando el rendimiento.

3.2.2. Organización de la memoria HBM2 de una FPGA Xilinx

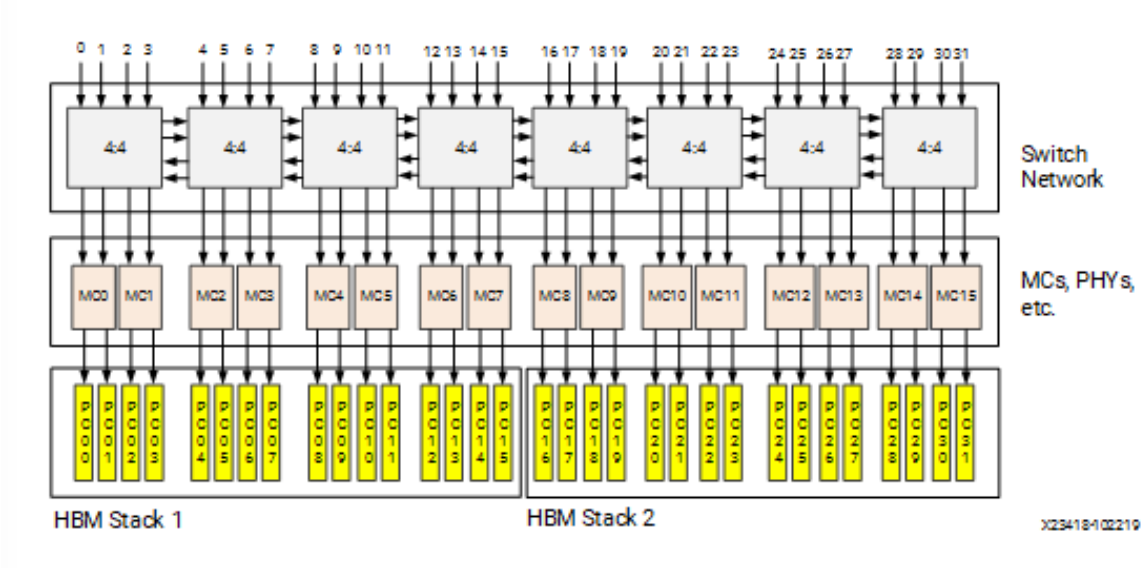

Figura 3.4: Estructura de la memoria de una FPGA de Xilinx

En la figura 3.4 podemos ver la estructura de la memoria HBM2 de una FPGA de Xilinx. Podemos ver que está formada por 2 stacks de memoria, cada uno formado por 8 controladores de memoria, denominados MC en la figura, que gestionan 2 pseudocanales (PC) cada uno, marcados en amarillo. El ancho de bits total de un stack HBM2 es de 1024 bits, dividido entre los 8 controladores, lo que supone un ancho cada uno de 128 bits. Cada PC tiene un ancho de 64 bits, aunque son independientes, por lo que se les pueden enviar 2 comandos, uno detrás del otro, pero no hay necesidad de esperar a que el primero termine para enviar el segundo.

Para compensar las diferencias de ancho de los canales, los chips HBM2 funcionan al doble de frecuencia que las interfaces AXI para comunicar con las memorias, es decir, si la interfaz AXI funciona a 450 Mhz, el chip tendrá que funcionar a 900 Mhz para compensar que en cada transacción él puede enviar la mitad de datos.

3.2.3. Crossbar de la memoria

Para que todas las interfaces AXI puedan acceder a todo el espacio de memoria disponible se utiliza un crossbar segmentado compuesto por 8 segmentos. Cada segmento del crossbar, marcado en gris en la figura 3.4, se comunica directamente con 2 controladores de memoria, es decir con 4 PCs. Cada interfaz AXI está conectada directamente a un solo segmento del crossbar.

Esto implica que si se desconoce la estructura de memoria y repartimos los datos a los que va a acceder una interfaz AXI en varios bancos, puede provocar que la interfaz tenga que acceder a bancos que no corresponden a su segmento del crossbar, por lo que tenga que cruzar varios segmentos hasta llegar al banco y hacer la operación de memoria.

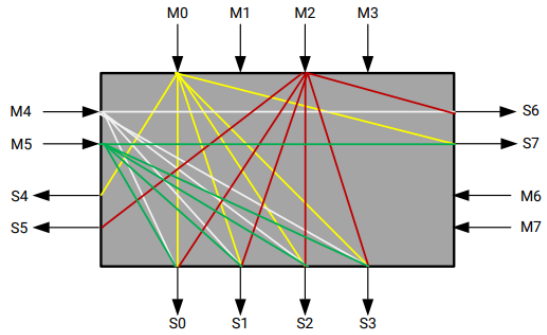

Figura 3.5: Estructura de un segmento del crossbar

En la figura 3.5 podemos ver la estructura de las conexiones de un segmento del crossbar. Podemos observar que tiene conexiones directas con todos los PCs que están alineados a él y como las conexiones laterales son las que comunican un segmento con sus contiguos. Es el hecho de pasar estas conexiones lo que provoca una latencia extra, y por tanto pérdida de rendimiento, ya que no se está accediendo directamente al banco de memoria.

Para este capítulo se ha tomado como referencia tanto para la información como para las figuras, salvo la figura 3.1, de elaboración propia, un repositorio de Xilinx [15] acerca de las interfaces AXI y dos apartados de la documentación oficial de Vitis [16] [17] acerca de la memoria HBM2.

Capítulo 4

Entorno de desarrollo

En este capítulo se va a hablar de las herramientas de desarrollo que se han utilizado y del entorno donde se ha realizado el trabajo. Vamos a hablar tanto del software como del hardware que se ha usado y sus características. También vamos a hablar de la organización que se ha seguido a la hora de trabajar en el proyecto y de como se ha configurado el entorno para poder llevarlo a cabo.

4.1. Entorno

Para el entorno de desarrollo se ha utilizado una máquina en el I3A (Instituto de investigación e Ingenieria de Aragón) para poder hacer uso de las FPGAs disponibles en dicho laboratorio. Esta máquina ha sido aportada por Denis Navarro, al cual se le agradece por dejarnos utilizarla.

4.1.1. Hardware

El hardware con el que cuenta esta máquina es una Xilinx Alveo U280.

En la figura 4.1 podemos ver las principales características de la memoria de la U280. Se pueden observar diferencias de organización en términos de capacidad y de número de bancos. Dado que la memoria HBM2 está pensada para obtener el máximo rendimiento trabajando en paralelo se colocan un gran número de bancos. Con respecto a la capacidad, las memorias HBM2 son bastante más costosas que las DDR4.

Los módulos HBM2 están organizados en 2 stacks de 4 GB cada uno, como se ha visto en el capítulo 3.

Memoria	Bancos	Capacidad total	Ancho de banda total	Ancho de banda por banco
HBM2	32	8 GB	460 GB/s	14.37 GB/s
DDR4	2	32 GB	38 GB/s	19 GB/s

Tabla 4.1: Memorias disponibles en la FPGA [18]

El dispositivo tiene un consumo de 225W y un total de 1079000 LUTs.

La máquina remota tiene como procesador un AMD Epyc 7443P. Este procesador cuenta con 24 núcleos y 48 hilos.

4.1.2. Software

El entorno de desarrollo que se ha utilizado ha sido Vitis, que es la plataforma de desarrollo de software para los aceleradores de Xilinx.

En esta herramienta disponemos de 3 modos de compilación. La emulación por software, que simplemente emula el comportamiento, pero no tiene nada en cuenta del hardware contra el que se va a ejecutar. Este método es el más rápido de compilar. La emulación hardware, que también emula el comportamiento, pero ya tiene en cuenta características de la FPGA. Este método es más lento de compilar, por lo que solo se ha utlizado cuando el código funcionaba correctamente y había que probar diferentes configuraciones del hardware. Por último, disponemos de la compilación hardware, que genera el bitstream completo para la FPGA, y que lo manda a ejecutar en el hardware real. Este método solo se ha utilizado cuando se han querido obtener números reales de ancho de banda y de rendimiento, ya que el tiempo de compilación era de unas 2-3 horas, por lo que solo se ha utilizado después de asegurarse de que el código funcionaba correctamente.

Esta herramienta puede compilar tanto código en HLS como código en HDL, utilizando Vivado. La única diferencia es que al compilar código en HLS se añade un paso intermedio, como está explicado en el capítulo 2, donde se traduce el código HLS a RTL.

Al principio, se optó por desarrollar en local durante las primeras pruebas para conocer el entorno de desarrollo de Vitis y como se programaba y se configuraba un kernel . Para ello se utilizó por completo el entorno gráfico utilizando el código del proyecto de ejemplo. Una vez se hicieron las primeras pruebas y pasamos a querer ejecutar en hardware real, se pasó a la máquina remota mencionada anteriormente, para poder utilizar la Alveo U280.

Para realizar las distintas pruebas, se tomó como base el repositorio GitHub de Xilinx VitisAccelExamples [19], y a partir de ahí, siguiendo la misma forma de desarrollar el código, se fueron haciendo diferentes versiones de los kernels.

Para desarrollar el código se ha utilizado el editor Gedit, que venía instalado en la máquina remota.

Para compilar el código, se ha utilizado el comando make en la terminal, junto con ficheros Makefile, proporcionados en el repositorio de GitHub antes mencionado, siendo estos modificados según se requería para las distintas versiones.

Para visualizar los reportes de compilación y de ejecución una vez realizadas las pruebas se ha utilizado la herramienta gráfica del entorno Vitis, Vitis Analyzer, que permite visualizar varios parámetros como pueden ser el ancho de banda, las memorias usadas, el uso de recursos en la FPGA, el esquema de conexiones de los kernels con la memoria, entre otros.

4.2. Método de conexión remoto

Para poder acceder a la máquina remota, se ha utilizado un proxy SSH con la máquina central.cps.unizar.es. La configuración para poder realizar la conexión se puede ver en el anexo 1.

Para poder conectarnos a la máquina remota de manera gráfica y así poder utilizar las aplicaciones de reportes y editar el código remotamente de manera más cómoda, se ha utilizado la herramienta VNCServer[1]. Se ha configurando el servidor en la máquina remota y un tunel SSH en el puerto 5904 de la máquina local, para poder acceder mediante VNC Viewer. El tunel se puede ver como está configurado en el anexo 1.

[1]https://www.realvnc.com/es/

Capítulo 5

Validación experimental

En HLS, siguiendo la terminología de la programación del modelo de OpenCL, y por simplificar en este TFG, se considerará qué la programación de las FPGA se realiza mediante kernels, que son las funciones que nuestro dispositivo va a acabar programando en su hardware y va a ejecutar para resolver el problema.

Estos kernels tienen argumentos de entrada y de salida, que tendrán que ser proporcionados al acelerador por el host. En nuestro caso es el procesador de propósito general. Para ello antes de ejecutar el kernel necesitaremos generar los datos con los que vamos a trabajar y copiarlos en la memoria del acelerador. Una vez acabe la ejecución del kernel, tendremos que recuperar los resultados calculados.

Estos kernels se ejecutan en lo que se llama Compute Unit (CU), que es una región del hardware del dispositivo que se va a encargar de ejecutar nuestro kernel. A la hora de resolver un problema, se pueden usar varias CUs para utilizar una mayor cantidad de hardware y obtener un rendimiento mayor puesto que varias instancias del kernel se ejecutarán concurrentemente.

Todas las versiones analizadas en este TFG son modificaciones sobre un kernel de suma de vectores de enteros del tipo 5.1.

```
1 res[i] = a[i] + b[i]
```

Listing 5.1: cuerpo principal del kernel analizado de suma de vectores

Analizándolo, podemos ver que las iteraciones son independientes entre sí, por lo que al no haber dependencias este problema será muy pararelizable.

5.1. Metodología de trabajo

Para realizar los experimentos, primero se ha partido de un kernel base de suma de vectores ya que era sencillo y existían bastantes ejemplos ya desarrollados. Como base se ha utilizado el repositorio de GitHub de Xilinx VitisAccelExamples [19]. Posteriormente, se ha analizado su rendimiento y características.

Con los análisis e intentando paliar las limitaciones, se han ido creando versiones de manera iterativa, para comprobar si podían mejorar el rendimiento. Para ello primero se ha teorizado sobre las posibles mejoras y luego se ha comprobado si efectivamente lo eran. En caso negativo, se ha intentado averiguar la razón.

También se han creado versiones cuyo objetivo es verificar las características de la memoria, aunque no fueran simplemente para mejorar el rendimiento.

La primera prueba ha consistido en analizar versiones desarrolladas en HLS y HDL del mismo código. Primero se han analizado ambas versiones utilizando un solo banco de memoria y posteriormente se han modificado para que hagan uso de varios bancos. El objetivo de esta prueba ha sido comprobar las diferencias desde el punto de vista de la versatilidad y programabilidad entre las dos alternativas y ver si se podía ganar algo de rendimiento simplemente utilizando más bancos de memoria, sin modificar nada más del código.

La segunda prueba ha consistido en vectorizar un kernel en HLS para comprobar como el hecho de emplear datos escalares hacía que no se estuviera aprovechando todo el ancho de la memoria, y analizar la ganancia de rendimiento.

La tercera prueba ha consistido en explorar las dificultades de un buen uso de la memoria HBM y de comprobar como, en caso de no conocer la estructura de la memoria mencionada en el capítulo 3, no se obtendrá todo el rendimiento posible. Para ello se han creado dos ejemplos, uno que no viola ninguna restricción de rendimiento, y otro que si que lo hace.

Por último, se ha tratado de aumentar el rendimiento todo lo posible, y explorar el uso de múltiples Compute Units, generando múltiples instancias del mismo kernel y poniendo a trabajar esas CUs en paralelo resolviendo distintas partes del problema.

5.1.1. Flujo de trabajo

El flujo de trabajo para cada kernel ha sido, generar un código que compilara en modo software para comprobar su corrección. Para revisar los diferentes modos de compilación ir a la sección 4.1.2. Una vez funcionaba correctamente, se probaba con emulación hardware que las conexiones de la memoria con los kernels y las CUs iban a funcionar correctamente. Por último, se compilaba en modo hardware para extraer el rendimiento real y comprobar los datos numéricos con lo esperado, para posteriormente sacar conclusiones.

5.1.2. Reportes

Las mejoras de rendimiento se han podido observar en los reportes de compilación y de ejecución. En el reporte de compilación se puede observar los recursos de la FPGA utilizados, así como las conexiones de los puertos del kernel con la memoria o las estimaciones de frecuencia que va a alcanzar nuestro kernel.

En el reporte de ejecución podemos observar diferentes apartados. A continuación vamos a resaltar los que nos han parecido más interesantes.

Uno de ellos es el tiempo de ejecución, como se puede ver en la figura 5.1 donde se nos muestra el tiempo de ejecución por cada instancia del kernel y en que instante de tiempo se ha empezado a ejecutar. Es decir, si tuviéramos varias CUs, se mostraría el tiempo de ejecución de cada una de ellas.

Top Kernel Execution

Kernel	Kernel Instance Address	Context ID	Command Queue ID	Device	Start Time (ms)	Duration (ms)
krnl_vadd	0x201b4f0	0	0	xilinx_u280_gen3x16_xdma_base_1-0	11544.300	39.679

Figura 5.1: Ejemplo de tiempo de ejecución

Otro reporte muy útil para todas estas pruebas va a ser el del ancho de banda obtenido. Este reporte se puede ver en la figura 5.2, donde se nos muestra tanto el número de transferencias realizadas como el ancho de banda obtenido, todo ello desglosado por puerto y banco.

Data Transfer: Kernels from/to Global Memory

Compute Unit Port	Kernel Arguments	Device	Memory Resources	Transfer Type	Number of Transfers	Transfer Rate (MB/s)	BW Util wrt Current Port Config (%)	BW Util wrt Ideal Port Config (%)	Max BW on Current Port Config (MB/s)
krnl_vadd_1/m_axi_gmem0	in1	xilinx_u280_gen3x16_xdma_base_1-0	HBM[0]	READ	625000	1010.540	0.000	5.250	0.000
krnl_vadd_1/m_axi_gmem1	in2	xilinx_u280_gen3x16_xdma_base_1-0	HBM[1]	READ	625000	1010.540	0.000	5.250	0.000
krnl_vadd_1/m_axi_gmem2	out_r	xilinx_u280_gen3x16_xdma_base_1-0	HBM[2]	WRITE	625000	1010.520	0.000	5.249	0.000

Figura 5.2: Ejemplo de ancho de banda

Por último, otro reporte que se ha considerado interesante ha sido el del *timerline* de ejecución, observable en la figura 5.3. Se puede ver la linea temporal de la encolación del kernel en la FPGA, la ejecución del mismo y las lecturas y escrituras que se realizan en los diferentes bancos HBM y para los diferentes argumentos del kernel.

5.2. Diferencias entre versiones HDL y HLS

Lo primero que vamos a probar son varias versiones desarrolladas en HDL y en HLS, para ver las diferencias que hay entre ambas, y la dificultad para cambiar ciertos parámetros sin cambiar el kernel en sí.

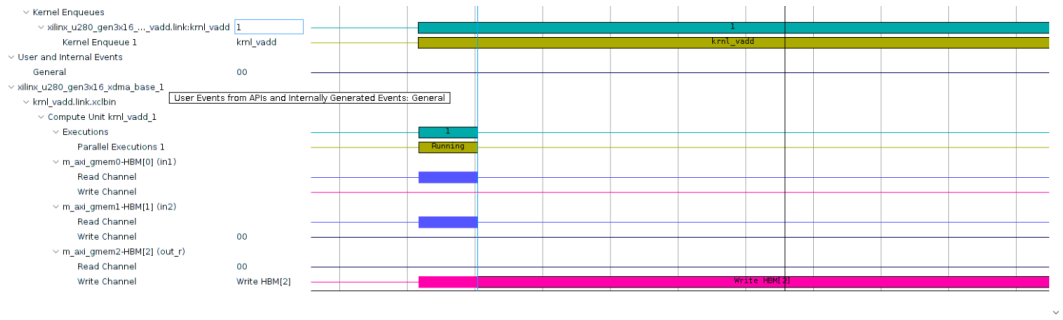

Figura 5.3: Timeline de ejecución

5.2.1. Versión HDL

Esta primera versión está desarrollada en HDL. El código del sumador se puede encontrar en el anexo 3. Este kernel almacena todos los datos en el banco HBM 0 y utiliza tipos de datos escalares de 32 bits. Al trabajar con un solo banco de memoria podemos esperar un bajo ancho de banda utilizado, debido a acumular todas las operaciones de memoria en un solo banco, teniendo 32 de ellos disponibles, y a utilizar solamente 32 de los 512 bits de ancho por cada interfaz AXI. Esto es debido a que estamos utilizando tipos de datos escalares en lugar de vectoriales, como se usarán en versiones posteriores.

Como resultados tenemos un tiempo de ejecución de 66.70 ms, con un ancho de banda de 1.80 GB/s. Es un ancho de banda bastante bajo, teniendo en cuenta que el máximo para un banco HBM son unos 13 GB/s.

La siguiente prueba parte del código anterior, con la diferencia de que se va a utilizar un banco de memoria para cada vector de entrada o salida del problema. Esto se hace para comprobar la facilidad de cambiar ciertas configuraciones, como en este caso los bancos de memoria, e intentar mejorar el ancho de banda y el rendimiento del código.

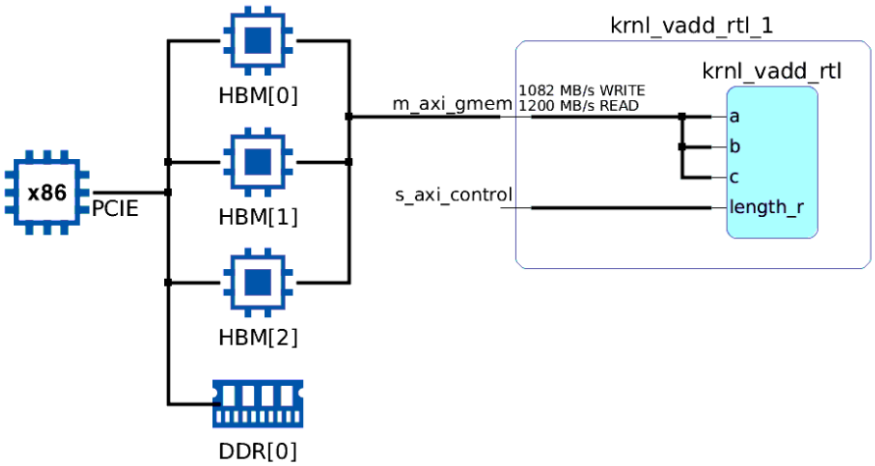

Figura 5.4: Diagrama AXI del kernel en RTL con múltiples bancos

Podemos observar en la figura 5.4, que aunque se están utilizando varios bancos

HBM, todos comparten la misma interfaz AXI. En siguientes versiones se verá si esto es un limitante o no, mapeando los diferentes argumentos del kernel a interfaces AXI diferentes.

Para lograr estos cambios, se han añadido unos flags al fichero de compilación Makefile, que se pueden ver en el código 5.2.

```
    CONN_FLAGS := --connectivity.sp krnl_vadd_rtl_1.a:HBM[0]
--connectivity.sp krnl_vadd_rtl_1.b:HBM[1]
--connectivity.sp krnl_vadd_rtl_1.c:HBM[2]

v++ -l $(TRACE_FLAGS) $(CONN_FLAGS) $(VPP_FLAGS) -t
$(TARGET) --platform $(PLATFORM) $(VPP_LDFLAGS)
--temp_dir $(TEMP_DIR) -o'$(LINK_OUTPUT)' $(+)
```

Listing 5.2: Mapeo a diferentes bancos de memoria

Durante todos los experimentos, el proceso habrá sido el mismo, modificando el índice de los bancos a los que se quieran mapear las variable.

Los resultados de esta prueba son un tiempo de ejecución de 66.70 ms y un ancho de banda de 1.80 GB/s.

Tenemos el mismo rendimiento que en la versión anterior, aunque se haya cambiado la configuración del kernel para usar varios bancos de memoria. Aunque no tenemos una respuesta clara para esto, puede ser debido a que al no poder explotar un solo banco de memoria, por mucho que se añadan más bancos no se aumente el rendimiento ya que el limitante no está siendo el banco en sí.

5.2.2. Versión HLS

Ahora vamos a probar una versión que hace lo mismo que la primera vista en RTL pero en HLS. Este código también almacena todas las variables en un solo banco y utiliza tipos de datos escalares de 32 bits.

En esta prueba obtenemos un tiempo de ejecución de 39.67 ms y un ancho de banda de 3.02 GB/s. Estamos obteniendo el doble de rendimiento que con la versión en RTL, probablemente debido a diferencias en la implementación en la versión en HLS genera un código diferente, que al final logra sacar un mayor rendimiento aunque ambos estén resolviendo el mismo kernel de suma de vectores.

Como última prueba, vamos a generar otra versión en HLS, que utilice varios bancos de memoria como la versión en RTL. Para ello se ha seguido el mismo método explicado anteriormente. La diferencia con la versión en RTL es que en esta se han podido configurar las interfaces AXI para que fueran diferentes para cada banco de memoria, simplemente añadiendo un `pragma` por argumento del kernel, como se puede ver en el código 5.3.

```
1    #pragma HLS INTERFACE m_axi port = in1 bundle = gmem0
2    #pragma HLS INTERFACE m_axi port = in2 bundle = gmem1
3    #pragma HLS INTERFACE m_axi port = out bundle = gmem2
```

Listing 5.3: Mapeo de los puertos a las interfaces AXI

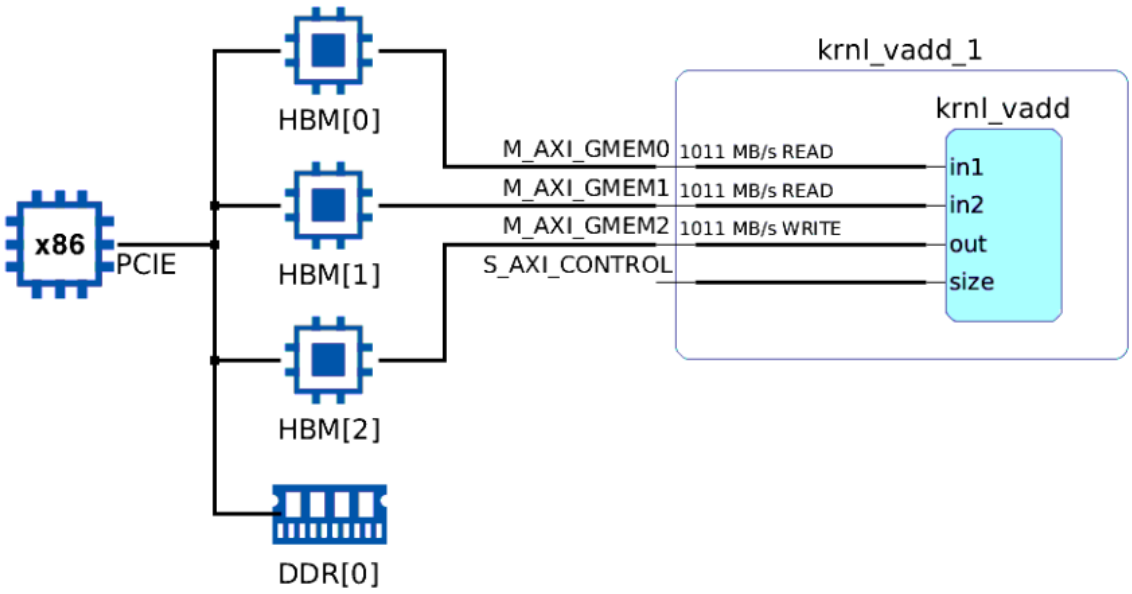

Figura 5.5: Diagrama de la conexión de múltiples interfaces AXI y multiples bancos

Podemos ver en la figura 5.5 que los diferentes argumentos del kernel ya no comparten interfaz AXI, sin embargo, el tiempo de ejecución obtenido ha sido de 39.67 ms y el ancho de banda ha sido de 3.02 GB/s. No notamos ninguna mejoría en el rendimiento, como en las dos versiones RTL. Esto es debido a que, como no se está logrando explotar un solo banco, por mucho que se agregen más bancos el rendimiento no va a aumentar.

5.2.3. Comparativa de los diferentes códigos

Vamos a calcular diferentes métricas de complejidad a la hora de programar el código y a la hora de evaluar la mantenibilidad de las dos versiones escalares mencionadas como el número de lineas o el volumen de Halstead. Para estas métricas solo se ha estudiado el código del acelerador ya que la parte del host es idéntica para ambas versiones, sean RTL o HLS.

Versión	Número de lineas	Volumen de Halstead
RTL escalar	68	122.62
HLS escalar	7	53.15

Tabla 5.1: Tabla con métricas de complejidad de las diferentes versiones escalares

Para ambas métricas solo se ha tenido en cuenta el fichero del sumador en RTL y la función que realizaba la suma en HLS. La metodología para calcular el número de lineas ha sido simplemente contar cuantas de ellas tenía cada fichero que conformaba el kernel en las diferentes versiones. Para calcular el volumen de Halstead se ha utilizado la fórmula 5.6 donde `N` es el número total de operandos y operadores y `n` es el número de operandos y operadores diferentes.

$$V = N \cdot log_2 n$$

Figura 5.6: Volumen de Halstead

Mirando el número de lineas podemos ver la enorme diferencia que hay entre las dos versiones, haciendo mucho más compleja la tarea de revisar el código en RTL, así como su mantenibilidad ya que está implementado en varios ficheros. Por último, el volumen de Halstead nos indica que el código en RTL es más complejo que la versión en HLS. Esto es por el mayor número de señales que en RTL tenemos que controlar, mientras que en alto nivel no son necesarias.

En estas métricas hay que recalcar que aunque no se han medido sobre todo el código RTL, a la hora de modificar los parámetros del kernel como pueden ser ciertos parámetros sobre la memoria, tendremos que modificar los ficheros que genera la herramienta cuando se genera un kernel en RTL, como pueden ser los ficheros que controlan las interfaces AXI tanto para lectura como para escritura.

5.3. Aumento del ancho de banda utilizando vectorización

En esta prueba vamos a utilizar tipos de datos vectoriales para tratar de aumentar el rendimiento de nuestro kernel, aunque se volverá a utilizar un solo banco para todos los datos. Vamos a utilizar el tipo de dato vectorial `hls::vector`, cada uno de 16 elementos, para ocupar el ancho de 512 bits de la interfaz AXI y realizar varias operaciones al mismo tiempo. El código se puede encontrar en el anexo 4. En esta prueba se espera un mayor uso del ancho de banda debido a que estaremos moviendo más datos por transacción en memoria al utilizar todo el ancho del canal, aunque seguimos dependiendo de un solo banco de memoria, el cual será el limitante.

Los resultados de la prueba han sido mejores que en pruebas anteriores, con un tiempo de ejecución de 9.27 ms y un ancho de banda de 12.94 GB/s.

Con estos resultados estamos dividiendo el tiempo de ejecución por 4 a la vez que

se multiplica por 4 el ancho de banda obtenido. Con estos datos podemos concluir que el ancho de banda obtenido y el tiempo de ejecución tienen una gran relación entre sí, puesto que si aumenta uno disminuye otro y viceversa.

Por último, respecto al ancho de banda obtenido, podemos decir que estamos bastante cerca del límite de ancho de banda para un solo banco de memoria. Si quisiéramos aumentar más el ancho de banda, tendríamos que pensar en utilizar varios bancos con tipos de datos vectoriales o varias CUs. Después de esta prueba, podemos concluir que el hecho de estar utilizando tipos de datos escalares es mucho más limitante que solo utilizar un banco HBM o utilizar varios de ellos. Esto es debido a que, como hemos mencionado anteriormente, estamos aprovechando mucho mejor el ancho del canal, y siendo más eficientes con las operaciones en memoria.

5.4. Diferencias en la organización de los datos en memoria

En estas pruebas vamos a verificar como la diferente organización de los datos del kernel en los distintos bancos de memoria puede afectar al ancho de banda obtenido, cambiando con ello el rendimiento de nuestro kernel como se explica en el capítulo 3.

En ambas versiones se van a usar tipos de datos vectoriales, tal y como se usaron en la prueba 5.3.

En la primera versión vamos a organizar los datos como se ve en la figura 5.7, para comprobar que efectivamente no hay limitación en el ancho de banda con esa distribución de los datos.

Una vez realizada la prueba, el tiempo de ejecución ha sido de 3.12 ms y el ancho de banda ha sido de 39 GB/s. Estos números están cerca del límite para 3 bancos siendo usados por el kernel, ya que estamos utilizando 3 interfaces AXI, que sería de unos 43 GB/s, por lo que el rendimiento obtenido es satisfactorio.

A continuación vamos a probar si con otra organización de los datos, a priori peor, el rendimiento cae, o por el contrario se mantiene. Para ello vamos a intentar organizar los datos de tal manera que el hecho de tener que pasar entre diferentes segmentos del crossbar limite el ancho de banda, como se ha explicado en el capítulo 3. Para lograrlo vamos a intercalar los datos de in2 y de res en los bancos de 16 a 31 y los datos de in1 en los bancos de 0 a 15. Esta organización se puede observar en la figura 5.8.

En esta prueba, el tiempo de ejecución ha sido de 6.22 ms, obteniendo un ancho de banda de 19.30 GB/s. Estos resultados son bastante peores, teniendo en cuenta la cantidad de bancos que se están utilizando, visibles en en la figura 5.8.

Podemos ver, comparando con la versión anterior, que el tiempo de ejecución ha

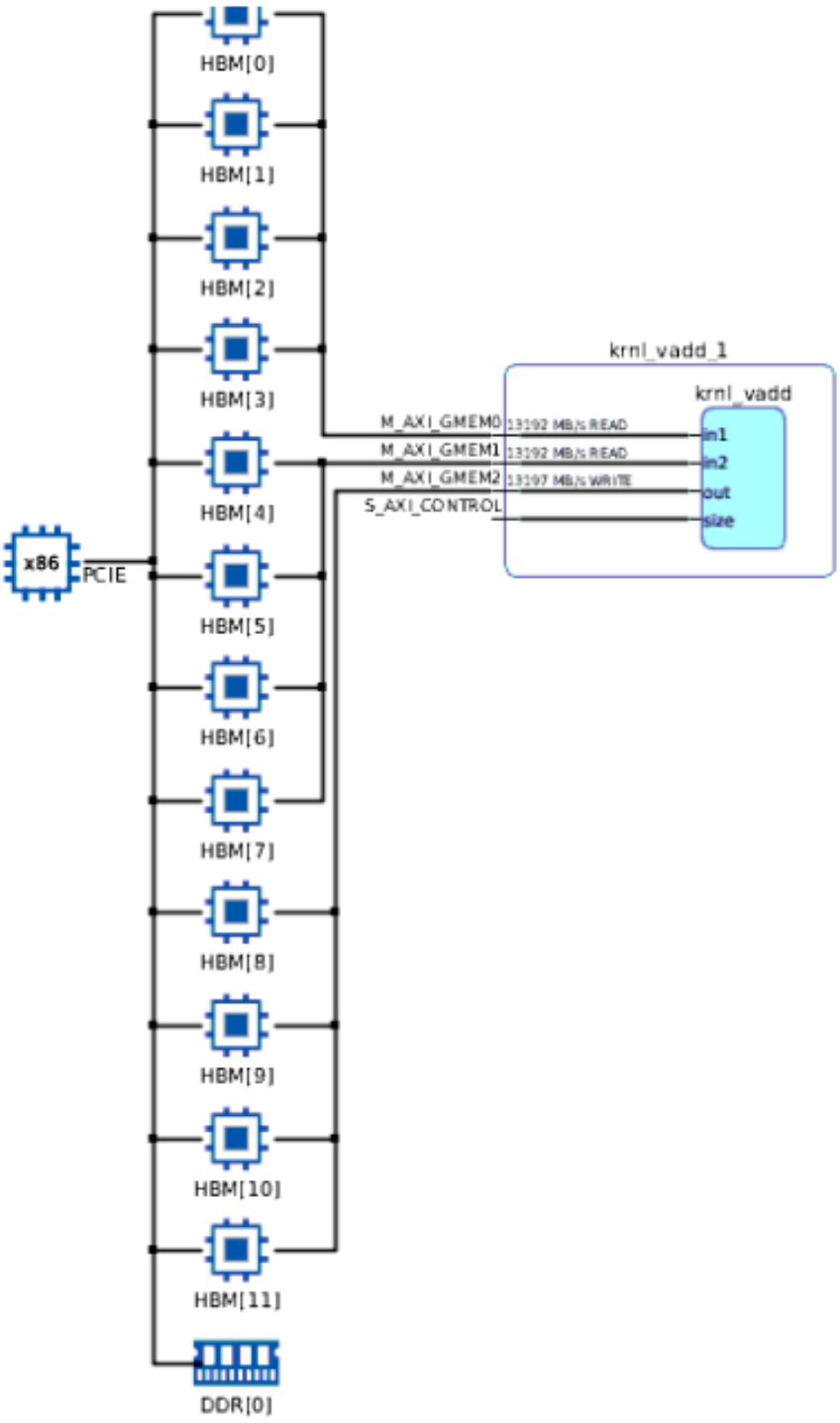

Figura 5.7: Diagrama de conexión de un kernel usando múltiples bancos por argumento

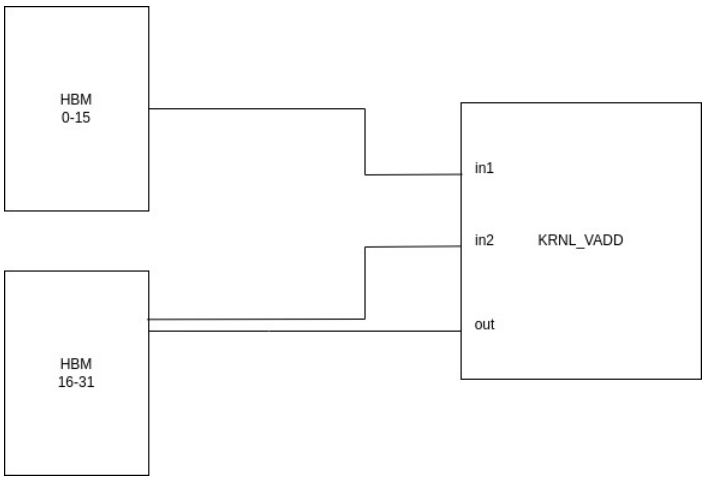

Figura 5.8: Diagrama de conexión de un kernel usando múltiples bancos por argumento con una mala organización

sido el doble, y el ancho de banda obtenido ha sido menor.

Con estos resultados, podemos asegurar que si no se tiene cuidado y una interfaz AXI tiene que recorrer varios segmentos del crossbar, debido a la forma en la que se organizan los datos, experimentaremos una pérdida en el rendimiento tal como se explicó en el capítulo 3.

5.5. Uso de varias Compute Units

En esta prueba, vamos a instanciar múltiples Compute Units (CU), para tratar de aumentar el ancho de banda y el rendimiento, ya que vamos a partir el problema en trozos, y los vamos a resolver en múltiples CUs en paralelo. Cada CU utilizará operaciones vectoriales, y tendrá cada argumento, es decir tanto los vectores de entrada como el de salida, mapeados a un módulo HBM. Cada módulo HBM solo será usado por un argumento de una CU para no saturar un banco y repartir los accesos a memoria. Al tener varias instancias del mismo kernel resolviendo porciones pequeñas del problema en paralelo y con memoria separada entre ellos, esperamos que el rendimiento pueda mejorar considerablemente.

En el código 5.4 se puede ver como se pasan los argumentos para cada CU, tanto de entrada como de salida, y se pone en ejecución. Posteriormente habrá que esperar a que todas las CUs acaben para recoger los resultados.

```
    for (int i = 0; i < NUM_KERNEL; i++) {
    if(i == (NUM_KERNEL - 1)) resto = DATA_SIZE % NUM_KERNEL;

        // Setting the k_vadd Arguments
        OCL_CHECK(err, err = krnls[i].setArg(0, buffer_input1[i]));
        OCL_CHECK(err, err = krnls[i].setArg(1, buffer_input2[i]));
        OCL_CHECK(err, err = krnls[i].setArg(2, buffer_output_add[i]));
        OCL_CHECK(err, err = krnls[i].setArg(3, size + resto));

        // Invoking the kernel
        OCL_CHECK(err, err = q.enqueueTask(krnls[i]));
    }
```

Listing 5.4: Instancia de varias CUs

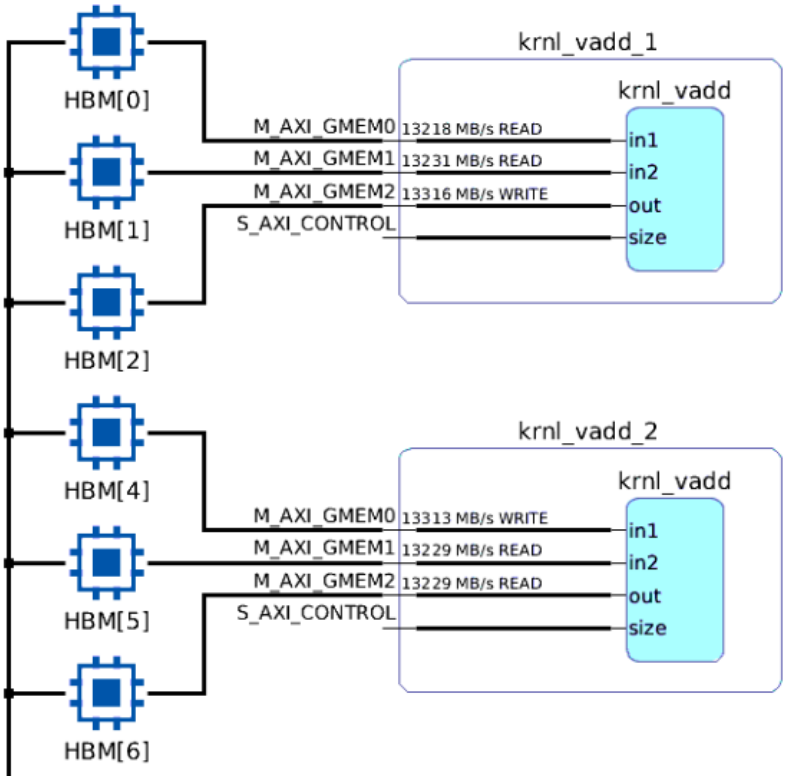

Figura 5.9: Diagrama del conexionado de múltiples CUs con sus bancos

Esta figura muestra las conexiones de los bancos con los puertos de las dos primeras CUs de nuestro kernel. El resto de conexiones sería de la misma manera, utilizando los 3 bancos contiguos para cada puerto. Mirando esta figura, podemos asumir que el ancho de banda va a ser mucho mayor que en versiones anteriores, ya que vamos a estar resolviendo trozos del problema en paralelo usando bancos HBM independientes entre sí.

Para esta prueba, se ha cambiado la metodología de mapeo de las variables a los diferentes bancos, puesto que también había que elegir las diferentes CUs. Para ello se ha utilizado un fichero `.cfg` que se puede consultar en el anexo 2. Para poderlo utilizar

también se ha añadido una linea al Makefile como se puede ver en el código 5.5. Con esas lineas de código se le dice al compilador que busque un fichero .cfg para leer la configuración y aplicarla.

```
VPP_LDFLAGS_krnl_vadd += --config ./krnl_vadd.cfg
----
v++ -l $(TRACE_FLAGS) $(VPP_FLAGS) $(VPP_LDFLAGS)
-t $(TARGET) --platform $(PLATFORM) --temp_dir $(TEMP_DIR)
$(VPP_LDFLAGS_krnl_vadd) -o'$(LINK_OUTPUT)' $(+)
```

Listing 5.5: Modificaciones al fichero Makefile para leer el fichero de configuración

En esta prueba el tiempo de ejecución ha sido de 0.45 ms y el ancho de banda obtenido de 316.80 GB/s. Estas métricas son bastante buenas, ya que nos acercamos al límite de ancho de banda que es posible obtener utilizando 24 bancos de memoria, que es de 344 GB/s, si multiplicamos el ancho de banda teórico de 14.37 GB/s por los bancos que estamos utilizando.

Hay que recalcar el reporte original medía un tiempo de 3.60 ms, pero revisando los reportes más a fondo esa métrica es una suma del tiempo de ejecución de todas las CUs, por lo que nos puede dar una idea equivocada del rendimiento del kernel.

5.6. Comparación final

Por último vamos a analizar una tabla comparativa de todas las versiones que se han ido realizando, añadiendo como métrica las OPs, es decir, las operaciones por segundo que han realizado las diferentes versiones.

Versión	Tiempo de ejecución (ms)	Ancho de banda (GB/s)	OPs (GOPs)
RTL escalar	66.67	1.80	0.15
HLS escalar	39.60	3.02	0.25
HLS vectorial	9.27	12.94	1.08
Organización de los datos correcta	3.12	39.00	3.20
Organización de los datos incorrecta	6.22	19.30	1.61
Varias CUs	0.45	316.80	22.22

Tabla 5.2: Tabla comparativa de las diferentes versiones

En la tabla 5.2 podemos observar las diferencias de rendimiento y de ancho de banda que hay entre las diferentes versiones.

Podemos ver que la versión para la que más rendimiento se obtiene es en la versión con múltiples CUs, ya que tenemos más grado de paralelismo, tanto de las operaciones como del ancho de banda. Los datos de GOPs son coherentes, ya que obtenemos aproximadamente 8 veces más GOPs que la versión que tiene los datos organizados correctamente y puede obtener todo el ancho de banda de los bancos.

Aunque no estemos en el límite del ancho de banda de la FPGA podemos esperar que si se utilizara un kernel que pudiera utilizar todos los bancos disponibles el ancho de banda escalaría en consonancia, llegando al límite de rendimiento de la FPGA.

También, que el hecho de vectorizar los kernels ayuda mucho en el rendimiento, ya que utilizando versiones escalares no se ha podido aumentar el rendimiento aunque se haya aumentado el número de bancos de memoria utilizados.

Capítulo 6

Conclusiones

En conclusión, la elaboración de este TFG ha representado un esfuerzo en la exploración e investigación de la programación de sistemas heterogéneos.

Como primera conclusión, la baja programabilidad de las FPGAs es cierta, debido a sus largos tiempos de compilación sin emulación, su dificultad para desarrollar un código correcto debido a que hay que ser muy específico para que todo funcione correctamente y debido a la vaguedad de los errores de compilación proporcionados por el compilador.

Con respecto a los accesos a memoria, se ha llegado a la conclusión de que es muy importante conocer la estructura y la organización de la memoria que estamos utilizando para optimizar en la manera de lo posible el rendimiento de nuestro código y evitar cuellos de botella.

Con respecto a las diferencias entre desarrollar código utilizando HLS y HDL, aunque no se ha conseguido desarrollar código en HDL, si que se intentó al comienzo del trabajo, no lográndose. También hemos visto, analizando las 2 versiones del mismo kernel implementadas de las 2 diferentes maneras, las ventajas que ofrece HLS, permitiendo una comprensión mucho mayor del código que su equivalente en HDL, así como una mayor versatilidad a la hora de realizar cambios sobre este.

Como conclusión adicional, podemos decir que parte de la dificultad para desarrollar código para estos sistemas es debido a lo específicos que son, ya que solo tenemos disponible la documentación oficial y apenas hay ejemplos concretos de diseños, o si los hay, solo se implementan las optimizaciones individualmente y es complicado si se quiere sacar provecho de varias de estas a la vez.

Capítulo 7

Bibliografía

[1] Jeffrey S. Vetter, Erik P. DeBenedictis, and Thomas M. Conte. Architectures for the post-moore era. *IEEE Micro*, 37(4):6–8, 2017.

[2] Juan Manuel Manchado Ortega y Jorge Antonio García Pérez. Fpga: qué es y cuáles son las características de este componente. *akka-technologies*. https://www.akka-technologies.com/fpga.

[3] Intel. Intel official website. https://www.intel.com.

[4] Xilinx Inc. Vitis unified software platform documentation: Application acceleration development (ug1393), 2023. https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/HBM-Configuration-and-Use.

[5] Intel. Intel oneapi toolkits, 2023. https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html.

[6] Intel. Intel quartus prime. https://www.intel.la/content/www/xl/es/products/details/fpga/development-tools/quartus-prime.html.

[7] Xilinx Inc. Vitis unified software platform, 2023. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.

[8] Intel. Vivado ml overview, 2023. https://www.xilinx.com/products/design-tools/vivado.html.

[9] Nvidia. Nvidia official website. https://www.nvidia.com/es-es/.

[10] AMD. Amd official website. https://www.amd.com.

[11] Nvidia. Cuda-x bibliotecas aceleradas por gpu para ia y hpc, 2023. https://www.nvidia.com/es-es/technologies/cuda-x/.

[12] AMD. Computación de alto rendimiento(hpc) en rocm, 2023. https://www.amd.com/es/graphics/servers-solutions-rocm-hpc.

[13] AMD. Hip github repository, 2023. https://github.com/ROCm-Developer-Tools/HIP.

[14] ARM. Especificación de arm del protocolo axi. https://developer.arm.com/documentation/ihi0022/latest/.

[15] florentw. Axi basics 1 - introduction to axi, 2023. https://support.xilinx.com/s/article/1053914?language=en_US.

[16] Xilinx Inc. Hbm performance concepts. axi high bandwidth memory controller logicore ip product guide (pg276), 2023. https://docs.xilinx.com/r/en-US/pg276-axi-hbm/HBM-Performance-Concepts.

[17] Xilinx Inc. Vitis unified software platform documentation: Application acceleration development (ug1393), 2023. https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/HBM-Configuration-and-Use.

[18] Xilinx Inc. Alveo u280 data center accelerator card. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html#specifications.

[19] Xilinx Inc. Vitis_accel_examples. https://github.com/Xilinx/Vitis_Accel_Examples.

Lista de Figuras

Lista de Tablas

Anexos

Figura 1: Diagrama de Gantt del proyecto

```
Host central
    HostName central.cps.unizar.es
    User a779935

Host TFGLab
    HostName 155.210.134.18
    Port 3334
    User pcabra
    ProxyJump central
    LocalForward 5904 localhost:5904
```

Listing 1: Configuración fichero ssh

```
[connectivity]
sp=krnl_vadd_1.in1:HBM[0]
sp=krnl_vadd_1.in2:HBM[1]
sp=krnl_vadd_1.out:HBM[2]

sp=krnl_vadd_2.in1:HBM[4]
sp=krnl_vadd_2.in2:HBM[5]
sp=krnl_vadd_2.out:HBM[6]

sp=krnl_vadd_3.in1:HBM[8]
sp=krnl_vadd_3.in2:HBM[9]
sp=krnl_vadd_3.out:HBM[10]

sp=krnl_vadd_4.in1:HBM[12]
sp=krnl_vadd_4.in2:HBM[13]
sp=krnl_vadd_4.out:HBM[14]

sp=krnl_vadd_5.in1:HBM[16]
sp=krnl_vadd_5.in2:HBM[17]
sp=krnl_vadd_5.out:HBM[18]

sp=krnl_vadd_6.in1:HBM[20]
sp=krnl_vadd_6.in2:HBM[21]
sp=krnl_vadd_6.out:HBM[22]

sp=krnl_vadd_7.in1:HBM[24]
sp=krnl_vadd_7.in2:HBM[25]
sp=krnl_vadd_7.out:HBM[26]

sp=krnl_vadd_8.in1:HBM[28]
sp=krnl_vadd_8.in2:HBM[29]
sp=krnl_vadd_8.out:HBM[30]

nk=krnl_vadd:8

[profile]
data=all:all:all
```

Listing 2: Fichero .cfg para asignar diferentes bancos con diferentes CUs

```
    /**
* Copyright (C) 2019-2021 Xilinx, Inc
*
* Licensed under the Apache License, Version 2.0 (the "License"). You may
* not use this file except in compliance with the License. A copy of the
* License is located at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
* WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
* License for the specific language governing permissions and limitations
* under the License.
*/

////////////////////////////////////////////////////////////////////////////////
// Description: Basic Adder, no overflow. Unsigned. Combinatorial.
////////////////////////////////////////////////////////////////////////////////

`default_nettype none

module krnl_vadd_rtl_adder #(
  parameter integer C_DATA_WIDTH   = 32,
  // Data width of both input and output data
  parameter integer C_NUM_CHANNELS = 2
  // Number of input channels.  Only a value of 2 implemented.
)
(
  input wire                                          aclk,
  input wire                                          areset,

  input wire  [C_NUM_CHANNELS-1:0]                    s_tvalid,
  input wire  [C_NUM_CHANNELS-1:0][C_DATA_WIDTH-1:0] s_tdata,
  output wire [C_NUM_CHANNELS-1:0]                    s_tready,

  output wire                                         m_tvalid,
  output wire [C_DATA_WIDTH-1:0]                      m_tdata,
  input  wire                                         m_tready

);

timeunit 1ps;
timeprecision 1ps;

////////////////////////////////////////////////////////////////////////////////
// Variables
////////////////////////////////////////////////////////////////////////////////
logic [C_DATA_WIDTH-1:0] acc;

////////////////////////////////////////////////////////////////////////////////
// Logic
////////////////////////////////////////////////////////////////////////////////

always_comb begin
  acc = s_tdata[0];
  for (int i = 1; i < C_NUM_CHANNELS; i++) begin
```

```
58      acc = acc + s_tdata[i];
59    end
60  end
61
62  assign m_tvalid = &s_tvalid;
63  assign m_tdata = acc;
64
65  // Only assert s_tready when transfer has been accepted.
66  tready asserted on all channels simultaneously
67  assign s_tready = m_tready & m_tvalid ? {C_NUM_CHANNELS{1'b1}} :
68  {C_NUM_CHANNELS{1'b0}};
69
70  endmodule : krnl_vadd_rtl_adder
71
72  `default_nettype wire
```

Listing 3: Sumador de una suma de vectores en HDL

```
1
2  void krnl_vadd(hls::vector<int,16>* in1, hls::vector<int,16>* in2,
3  hls::vector<int,16>* out, int size) {
4      #pragma HLS INTERFACE m_axi port = in1 bundle = gmem0
5      #pragma HLS INTERFACE m_axi port = in2 bundle = gmem1
6      #pragma HLS INTERFACE m_axi port = out bundle = gmem0
7
8      for( int i = 0; i < c_size;i++){
9          #pragma HLS pipeline II=1
10         out[i] = in1[i] + in2[i];
11     }
12
13 }
```

Listing 4: Ejemplo de suma de vectores vectorial