UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Implementación de aceleradores OpenCL sobre FPGA de funciones básicas necesarias en algoritmos utilizados en Deep Learning

Juan Compadre Ochando

Tutor: Rafael Gadea Gironés

Trabajo de Fin de Máster para optar por el título de Máster Universitario en Ingeniería de Sistemas Electrónicos

Universitat Politécnica de Valencia,

Departamento de Ingeniería Electrónica

Valencia, 18 de septiembre de 2018

ÍNDICE

1. Objetivos

Este trabajo tiene como principal objetivo el estudio de distintas técnicas de implementación del algoritmo de multiplicación de matrices, mediante técnicas de computación paralela. Para ello se utilizará el lenguaje OpenCL, que se ejecutará principalmente sobre FPGAs, y en algunas ocasiones también sobre GPGPUs.

Las distintas implementaciones estudiadas tratan de aprovechar los recursos disponibles en la plataforma FPGA. Por ello se utilizan operaciones en coma fija y en coma flotante, transformación entre ellas, implementaciones en single work ítem y en NDRange, uso de librerías de Verilog, uso de canales, etc.

En primer lugar, se toma un algoritmo proporcionado por Intel FPGA que utiliza operaciones en coma flotante, el cual se compila utilizando diferentes alternativas, buscando la óptima para la FPGA objetivo, una Alaric Arria 10. Tras evaluar el rendimiento de cada una de estas alternativas, se procede a convertir el algoritmo para utilizar operaciones en coma fija, para lo cual se utilizan librerías escritas en Verilog.

Tras analizar las opciones derivadas del algoritmo de Intel FPGA, se escribirá una implementación propia que aproveche las características de OpenCL, tales como canales y multikernel. Esta implementación está basada en un array sistólico, que es una estructura que encaja muy bien dentro de un modelo de programación paralela y una plataforma como la FPGA.

Además, otro objetivo del presente proyecto es la creación de un entorno de trabajo que facilite y simplifique la programación de FPGAs y GPGPUs con OpenCL. Para ello se utilizarán herramientas externas que permitirán integrar un entorno de trabajo adaptado a las necesidades de estas plataformas.

2. Introducción teórica

2.1. Multiplicación de matrices

La multiplicación de matrices es una de las operaciones más utilizadas hoy en día [1]. Se encuentra presente en la mayoría de los algoritmos de procesado de imagen (que permiten realizar rotaciones, escalados, translaciones, etc.) y procesado de señal (donde permite realizar convoluciones), se utiliza para resolver sistemas de ecuaciones, para cambiar de coordenadas, para realizar transformaciones lineales o para redes neuronales. Ha sido implementada con numerosos algoritmos y ha sido objeto de diversas técnicas de optimización.

Con la llegada de la computación paralela, las posibilidades de optimización han crecido enormemente y se utilizan implementaciones con lenguajes de programación como OpenCL y CUDA. Estos lenguajes permiten aprovechar las ventajas de los nuevos dispositivos como GPUs, GPGPUs y FPGAs frente a las CPU de propósito general, utilizando el paralelismo que ofrecen sus múltiples núcleos de manera que es posible lanzar un altísimo número de hilos simultáneos [2].

Las aplicaciones relacionadas con redes neuronales y deep learning son un área donde la aceleración de esta operación es de máximo interés, tanto por su gran aplicación en reconocimiento de voz y reconocimiento de imágenes, como por la enorme necesidad de aceleración de los complejos algoritmos utilizados [3].

La multiplicación de matrices es una operación que produce una matriz C a partir de otras dos A y B. Una de las principales propiedades de esta operación es que no cumple la propiedad conmutativa. Siendo la matriz A de tamaño m filas y n columnas, y la matriz B n filas y p columnas, la matriz C obtenida tendrá un tamaño de m filas y p columnas.

En esta operación, se hace el producto escalar de los vectores que forman los elementos de las filas de A y las columnas de B, realizando todas las posibles combinaciones.

$$C = AB = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{np} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix}$$

Figura 1. Obtención de matriz C

Cada elemento calculado únicamente depende de la fila y columna de A y B correspondientes, por lo que podría paralelizarse su cálculo. Esto consistiría la técnica más simple de paralelización de la multiplicación de matrices.

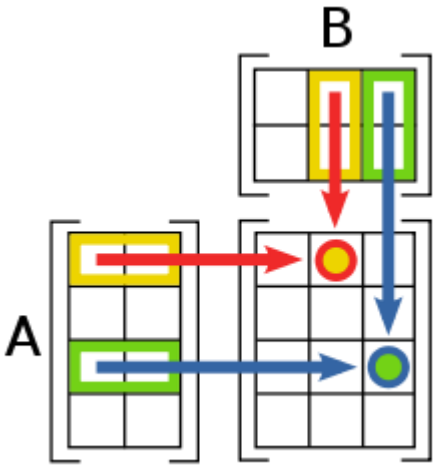


Figura 2 Cálculo de cada elemento

Un paso más allá consiste en dividir la matriz en bloques, de forma que a su vez cada bloque sea una matriz. Si cada bloque de la matriz C se calcula por un elemento de procesamiento, obtenemos una paralelización en la que cada elemento de procesamiento calcula varios índices de la matriz pero que comparte unos datos de entrada. Dependiendo del tamaño de las matrices y de la plataforma disponible, las técnicas de paralelización tienen distintos rendimientos. Todo esto se explicará con detalle en secciones posteriores, donde se implementará el algoritmo en OpenCL basado en estas técnicas.

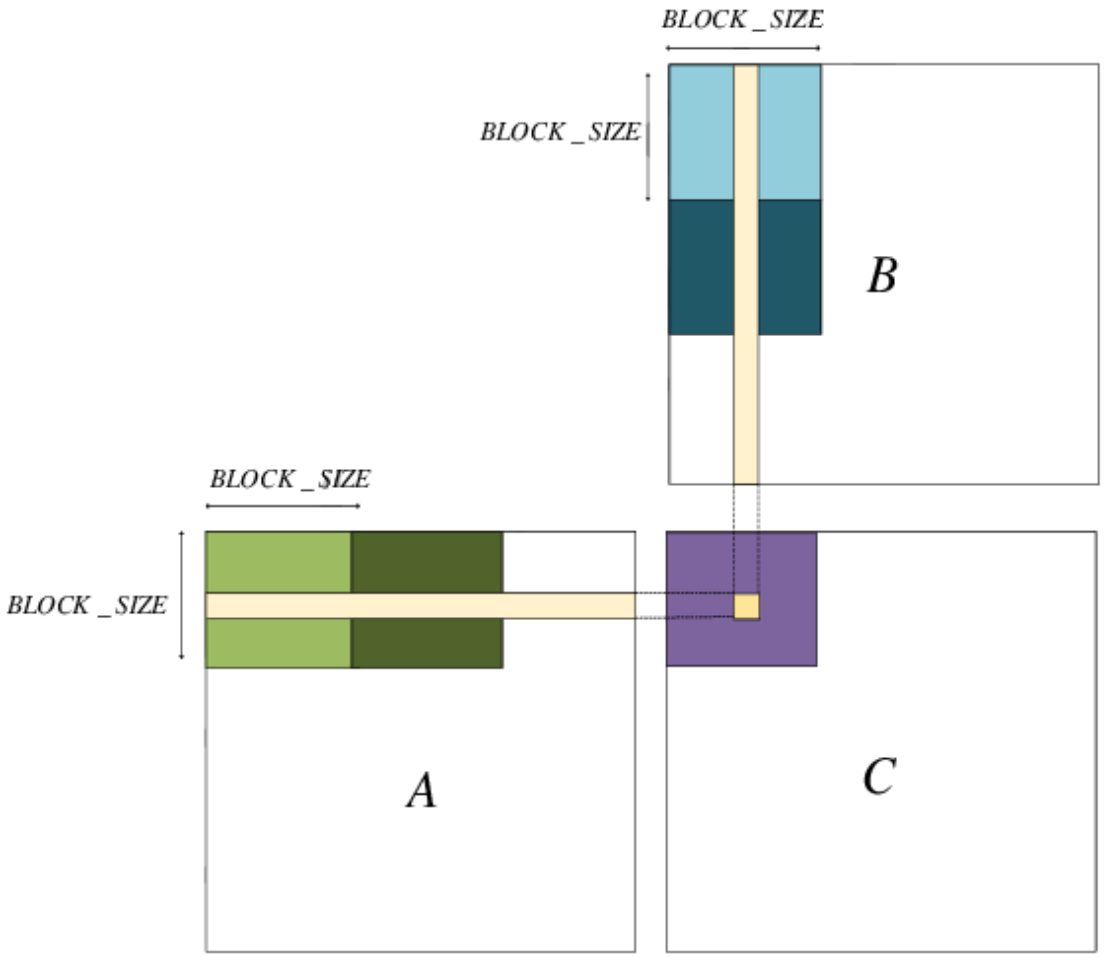


Figura 3 Multiplicación por bloques

Además de la multiplicación tradicional y su paralelización, se han desarrollado diversos algoritmos para optimizar el número de operaciones necesarias y de esta forma optimizar esta operación. En el caso más simple se necesitan n^3 multiplicaciones y $(n-1) \cdot n^2$ sumas para la multiplicación de dos matrices de tamaño $n\,x\,n$. Esta complejidad puede ser reducida, tal y como demostró Volker Strassen en 1969, obteniendo una complejidad de $n^{log_2(7)} = n^{2,807}$. A este exponente se le denomina ω y ha sido reducido durante el transcurso de los años hasta llegar a 2,37 en la actualidad (gracias a François Le Gall). El límite inferior es n^2, ya que las dos matrices han de ser leídas al menos una vez.

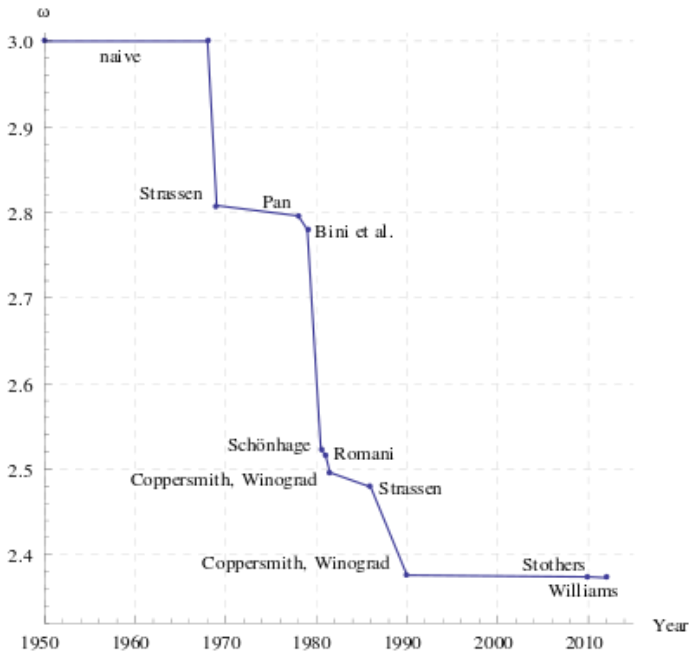


Figura 4 Complejidad algoritmos de multiplicación de matrices

2.2. OpenCL

OpenCl es un framework creado para la programación de sistemas compuestos de varios procesadores. Por ejemplo, un PC tiene CPU y GPU (además de otros procesadores), una FPGA puede tener varios procesadores de distintos tipos embebidos, un teléfono móvil también tiene varias unidades de procesamiento. Todos estos sistemas son llamados sistemas heterogéneos y son utilizados en todos los ámbitos.

OpenCL permite escribir un programa que se pueda ejecutar en cualquier dispositivo heterogéneo, sea un smartphone, un superordenador o un ordenador tradicional. Por ello tiene un gran potencial para revolucionar la industria y la filosofía de programación tradicional, más enfocada a un solo tipo de elemento de procesamiento.

La industria de los semiconductores hace algunos años que ha experimentado un cambio en su trayectoria. Inicialmente el objetivo de mejora era aumentar la capacidad de procesamiento, sin tener en cuenta el consumo energético. Llegado un punto, este camino se hizo inviable debido a los altos consumos, que hacían inviable integrar un chip con un único elemento de procesamiento muy potente. Fue entonces cuando el uso de varios elementos de procesamiento en paralelo cobró fuerza.

Estos elementos de procesamiento en paralelo necesitan un software que permita repartir tareas y ejecutar múltiples operaciones a la vez. Aquí nace el concepto de concurrencia, que es una propiedad de los sistemas por la cual dos o más hilos de procesamiento pueden realizar progreso y ejecutarse a la vez. Cuando la concurrencia ocurre en un sistema con varias unidades de procesamiento tenemos computación paralela.

El principal reto de la computación paralela es encontrar la concurrencia en el problema a resolver. Esto puede ser muy sencillo, como obtener un elemento de la matriz realizando el producto escalar de la fila y columna correspondientes. O puede ser muy complicado, como en un problema donde se obtengan resultados parciales que dependan de varias operaciones y que sean bloqueantes, para lo cual es necesario definir el orden de las operaciones y sus dependencias.

Una de las partes más complicadas de la computación paralela es manejar los detalles de bajo nivel. Ordenar perfectamente todos los hilos de ejecución o distribuir la memoria evitando conflictos. Por ello, una solución para la computación paralela es la creación de un modelo de abstracción de alto nivel, al que se le confíen las tareas de bajo nivel y que permita no tener en cuenta la gran complejidad del hardware. OpenCL implementa esta capa de abstracción.

Una aplicación de OpenCL deberá realizar las siguientes acciones:

- Encontrar los elementos de procesamiento que forman el sistema heterogéneo
- Probar las características de estos elementos para adaptar el software a ellos
- Crear los conjuntos de instrucciones, llamados kernels, que se ejecutarán en la plataforma
- Inicializar y manejar la memoria global y la asignada a cada kernel
- Ejecutar los kernel en el orden correcto y en los elementos correspondientes del sistema.
- Recoger ordenadamente los resultados de cada uno de los elementos de procesamiento.

Para realizar estas acciones, se dividirá el modelo de OpenCL en 4 modelos [4]:

- Modelo de plataforma: describe el sistema heterogéneo a alto nivel
- Modelo de ejecución: representación de cómo los hilos de procesamiento se ejecutan en el sistema heterogéneo
- Modelo de memoria: describe la división de la memoria y como interactúa con el modelo heterogéneo
- Modelo de programación: consiste en las abstracciones de alto nivel que se utilizan al diseñar algoritmos para una aplicación.

2.2.1. Modelo de plataforma

El modelo de plataforma describe una representación de alto nivel de cualquier sistema heterogéneo. Consta de dos partes principales. Por un lado, están los dispositivos OpenCL, donde se ejecutan los kernel. Por otro lado está un único host, cuya función es interactuar con el entorno externo al dispositivo OpenCL, lo que incluye las entradas y salidas y los programas de usuario.

Un dispositivo OpenCl puede ser la GPU, la CPU, un DSP o cualquier otro hardware soportado por OpenCL. Los dispositivos OpenCL se dividen en compute units que a su vez se dividen en Processing Elements (PE).

El host es quien maneja el sistema, comunicándose con los dispositivos OpenCL mediante colas de comandos y además envía y recibe datos desde la memoria propia de estos dispositivos

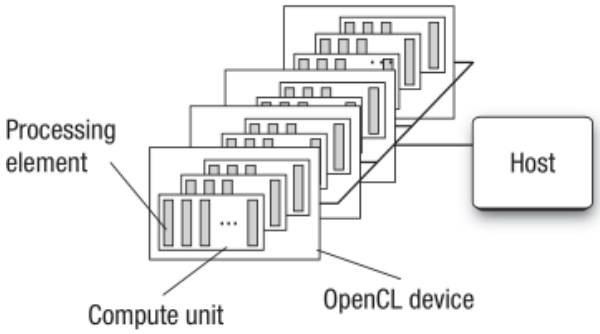


Figura 5 Modelo de plataforma

2.2.2. Modelo de ejecución

Como ya hemos visto, las aplicaciones OpenCL constan de dos partes: el programa host y uno o más kernels. El programa host se ejecuta en la parte host del modelo de plataforma y es el que define el contexto y maneja la ejecución de los kernel. OpenCL no define cómo debe ejecutarse, solo de cómo debe interactuar con los objetos OpenCL. Se puede implementar en cualquier lenguaje de programación para el cual esté definida la API de OpenCL. Los más comunes son C y C++ aunque existen nuevas APIs para usar otros, como Python. Este programa host deberá ser compilado con el compilador tradicional para el lenguaje en el que esté escrito.

Por su parte, los kernels se ejecutan en los dispositivos OpenCL y son los que realizan la computación paralela. Normalmente un kernel es una función sencilla que transforma datos de entrada en datos de salida. Están escritos en OpenCL y deben compilarse con el compilador proporcionado por el fabricante del dispositivo OpenCL. Esta compilación puede ser offline (antes de la ejecución del host) u online (durante la ejecución del host) dependiendo de tipo de dispositivo utilizado.

Cómo se ejecuta un kernel en un dispositivo OpenCL

Cuando el host envía un comando para ejecutar un kernel en un dispositivo, se crea un espacio multidimensional indexado en el dispositivo. Cada "posición" de este espacio ejecuta una instancia del kernel. A esta instancia se le llamará work-item. Cada work-item ejecuta la misma secuencia de instrucciones, aunque su comportamiento puede variar en función de los datos de entrada o de condiciones internas. Un work-item está identificado por un ID, al que se llama global ID. Este ID es único para cada work-item dentro de una ejecución OpenCL.

Los work-item se organizan en work-groups. Todos los work-groups son del mismo tamaño, es decir, tienen el mismo número de work-items. Además ocupan todo el espacio global disponible en cada dimensión. A los work-groups se les asigna un identificador único, pero además los work-item reciben otro identificador único dentro del work-group, al que se le llama local ID. Por tanto, un work-item puede identificarse mediante su global ID o mediante su local ID más el ID de su workgroup.

El índice (llamado de rango NDRange) en el cual se organizan work-groups y work-items puede ser de N dimensiones, aunque actualmente el límite está en 3. Por ejemplo, en un índice NDRange de 2 dimensiones, si el tamaño global es (Gx, Gy), el rango de identificadores para los work-item será (0..Gx-1, 0..Gy-1). Cuando dividimos este índice en work-groups tenemos que si el tamaño es (Wx, Wy), el rango de identificadores será (0..Wx-1, 0..Wy-1). Por tanto, los work-group serán de tamaño Lx = Gx/Wx , Ly = Gy/Wy. En la siguiente figura podemos encontrar un ejemplo del índice descrito.

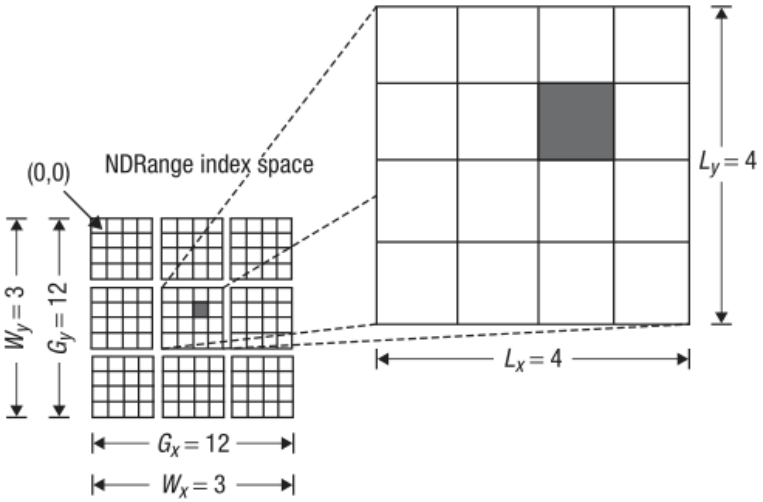


Figura 6 Workgroups y workitems

Dentro de un work-group, los work-item se ejecutan concurrentemente utilizando los PE (elementos de procesamiento) dentro de una misma Computer Unit. La concurrencia es la propiedad de los sistemas por la cual dos o más hilos de procesamiento pueden ejecutarse a la vez, pero no necesariamente realizan el progreso de forma simultánea. Cuando la concurrencia ocurre en un sistema con varias unidades de procesamiento que realizan progreso a la vez, tenemos computación paralela.

Dependiendo del dispositivo y de la compilación, una implementación puede serializar la ejecución de invocaciones de kernels o de work-groups. OpenCL solo asegura que los work-items dentro de un work-group se ejecutan de forma concurrente y por tanto comparten recursos de procesamiento. Esta característica lleva a una de las claves de programación en OpenCL: nunca se puede asumir que los work-groups o distintas invocaciones de kernels se ejecutan de forma concurrente.

Vista esta estructura, se puede mapear sobre el modelo de plataforma visto anteriormente, de manera que vemos la relación entre ellos en la siguiente imagen:

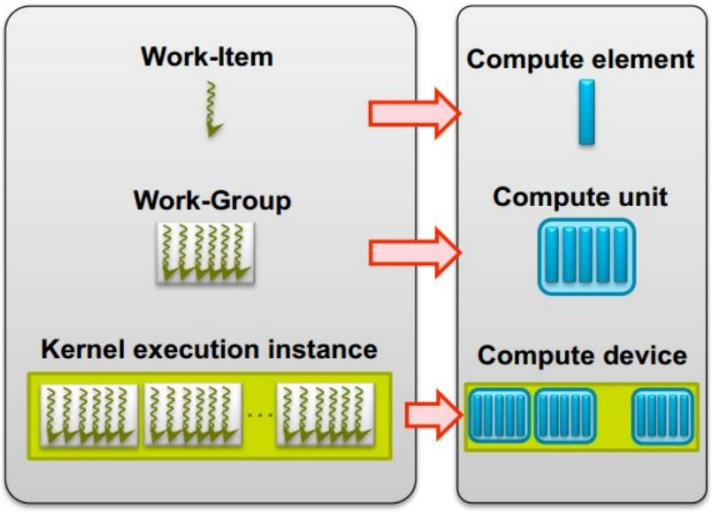


Figura 7 Relación entre modelo de ejecución y modelo de plataforma

Se observa cómo un work-item es ejecutado en un único compute element y que estos se agrupan en compute units de la misma forma que un work-item se agrupa en work-groups. Además, podemos observar como una instancia de ejecución de kernel se mapea en un compute device.

Contexto

El contexto define el entorno en el cual se ejecutan los kernels OpenCL. Este contexto está compuesto de una serie de recursos:

- **Devices**: dispositivos OpenCL utilizados por el host
- **Kernels**: las funciones OpenCL que se ejecutan en los dispositivos OpenCL
- **Program objects**: el código fuente y ejecutables que implementan los kernels
- **Memory objects**: objetos en memoria a los que pueden acceder los dispositivos OpenCL y con los cuales pueden operar los kernel.

La tarea de definir el contexto es propia del programa host. En él se definen los kernels, el índice NDRange, las colas y los detalles de cómo y cuándo se ejecutan los kernels. Todas estas tareas se realizan utilizando funciones de la API de OpenCL.

En un dispositivo OpenCL con GPU, CPU y algún procesador adicional, el host explorará el sistema para descubrir todos los recursos y elegirá una combinación de ellos, lo que define los dispositivos OpenCL dentro del contexto.

Dentro del contexto existen los llamados program objects. Para definir este concepto pensemos en una librería de la cual se cogen las funciones utilizadas por el kernel. El program object tiene que haber sido compilado (online u offline) para los dispositivos en los que se va a ejecutar (que deberán cumplir con la especificación OpenCL).

Por último, en lo que se refiere a la memoria, OpenCL tiene que manejar distintos tipos de memoria, ya que se ejecuta en una plataforma heterogénea donde los dispositivos pueden tener arquitecturas muy distintas. Esto complica enormemente el direccionamiento al tener múltiples espacios de memoria. La solución consiste en crear objetos de memoria (memory objects) de más alto nivel, lo que complica algo más la programación, pero permite utilizar la memoria de todos los dispositivos.

Colas de comandos

La interacción entre el host y los dispositivos OpenCL se produce mediante colas de comandos. El host crea estas colas y se las asigna a un dispositivo OpenCL. Dentro de ellas, los comandos OpenCL esperan hasta que son ejecutados en un dispositivo OpenCL. Cuando el host envía los comandos a la cola, no espera a que se ejecuten, sino que sigue con el resto de instrucciones. Para que espere en ese punto es necesario utilizar comandos de sincronización.

Existen 3 tipos de comandos:

- **Kernel execution commands**: son los comandos encargados de ejecutar un kernel en los dispositivos OpenCL.
- **Memory commands**: su función es escribir y leer memory objects y mapearlos en el espacio de memoria.
- **Synchronization commands**: añaden restricciones para la ejecución de los comandos. Es posible que un kernel necesite los datos producidos por otro kernel, por lo que un comando de sincronización fuerza a un kernel a no ejecutarse hasta que el que proporciona los datos acabe.

Además, dentro de una cola los comandos se pueden ejecutar de dos formas distintas:

- **In-order**: Los comandos se ejecutan en serie en el mismo orden en que llegaron a la cola. Sería un comportamiento FIFO.
- **Out-of-order**: Los comandos se ejecutan en el orden en que llegaron a la cola pero no se espera a que el anterior acabe para ejecutar uno nuevo. Cualquier orden debe ser implementado por el programador.

La ejecución de los comandos en una cola genera eventos, que pueden utilizarse para sincronizar entre ejecuciones del kernel. Se puede esperar a que se cumplan ciertas condiciones de un evento para lanzar el siguiente kernel o esperar a que finalicen una combinación de eventos. Además, los eventos sirven para coordinar la interacción entre dispositivos OpenCL y host.

Y, además, se pueden asociar varias colas dentro de un contexto a un mismo dispositivo OpenCL. Por defecto, estas colas no tienen ningún mecanismo de sincronización interno para los comandos, aunque pueden utilizar los eventos generados para obtener el orden de ejecución deseado.

2.2.3. Modelo de memoria

Como ya se ha mencionado, la memoria en OpenCL se utiliza a través de objetos de memoria. El modelo de memoria describe la estructura y comportamiento de la memoria en OpenCL.

La memoria se divide en 5 regiones:

- **Memoria del host:** se trata de una memoria que es solo visible para el host. Como con otros muchos aspectos relativos al host, OpenCL solo define cómo el host maneja y construye los objetos de memoria. El host utilizará el método del lenguaje en el que esté escrito para acceder a esta memoria.
- **Memoria global:** Esta región de memoria puede ser leída y escrita por cualquier work-item de cualquier work-group. Todos ellos tienen permiso para leer o escribir cualquier elemento de cualquier objeto de memoria.
- **Memoria constante:** Es una memoria en la que el host inicializa objetos de memoria que no se cambiarán durante la ejecución. Los work-item solamente tienen permiso de lectura.
- **Memoria local**: Se llama así a una región de memoria que es compartida por todos los work-item dentro de un mismo work-group. Se suele utilizar para almacenar datos que se comparten entre todos estos work-item. Si es posible se implementa con memoria dedicada del dispositivo OpenCL, lo que hace que sea una memoria más rápida que la global. Esto es útil para intercambiar información entre los work-item de un work-group y acelerar ciertos procesos. Si no es posible implementarla mediante memoria dedicada se utiliza una región de la memoria global.

- **Memoria privada**: Esta memoria pertenece a un único work-item y sólo él puede acceder a ella. Aquí se almacenan las variables que no pueden ser vistas por otros work-items.

En la siguiente figura se puede observar un esquema de las distintas regiones de memoria.

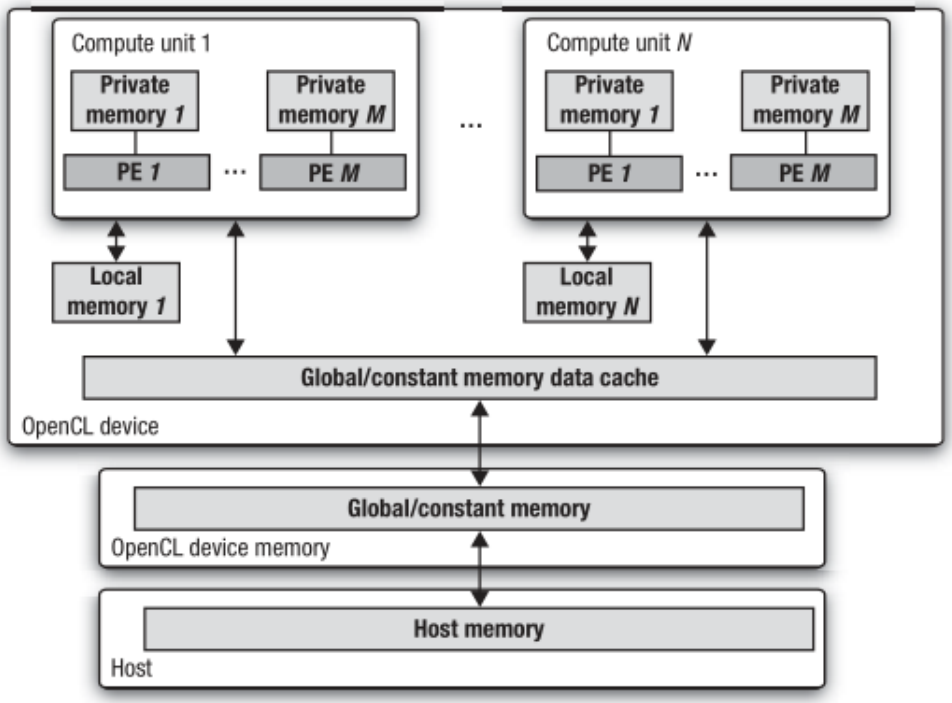


Figura 8 Regiones de memoria

Como se puede observar, los work-items se ejecutan en los Procesing Elements y tienen su propia memoria privada. Un work-group se mapea en una Compute Unit y comparte memoria local (y normalmente dedicada dentro del device) con sus work-items. Entre la memoria global del OpenCL device y la del host forman la memoria global y la constante.

Los modelos de memoria del host y del OpenCL device son independientes, aunque interactúan de dos formas distintas: copiando datos y mapeando objetos de memoria. Para copiar datos el host utiliza comandos dedicados a ellos, que pone en la cola correspondiente. Estos comandos pueden ser bloqueantes, que esperarán a que la memoria esté lista para usarse para acabar; o no bloqueantes, que acaban una vez se empieza a ejecutar el proceso de copia. En cuanto al mapeo de objetos de memoria, consiste en que el host "relaciona" estos objetos con su dirección de memoria correspondiente dentro del host. De esta manera puede leer o escribir estos objetos de memoria.

2.2.4. Modelo de programación

Por último, en cuanto al modelo de programación, tenemos dos alternativas: task parallelism (paralelización de tareas) y data parallelism (paralelización de datos), aunque solo utilizaremos data parallelism en este trabajo.

Data parallelism

Encaja de forma natural con el modelo de ejecución de OpenCL. El diseñador debe tener en cuenta el tamaño del NDRange para alinear correctamente los datos del problema y mapearlos en los objetos de memoria. El kernel definirá la secuencia de instrucciones que se le aplicará a estos datos.

En problemas complicados, los work-item dentro de un work-group necesitan compartir datos. Esto pueden hacerlo utilizando la memoria local, pero cuando se introducen dependencias, es necesario tener en cuenta que el resultado no debe variar dependiendo de qué kernels se ejecuten primero. Para ello existen algunas herramientas con las que coordinar la ejecución de los work-item. Una de estas es el uso de eventos, ya mencionada anteriormente. Otra es utilizar una barrera, a modo de punto de control, en la cual los work-item detendrán su ejecución hasta que todos ellos hayan llegado a ese punto. No existe ningún mecanismo para coordinar la ejecución de distintos work-groups.

Este modelo de ejecución también se divide en:

- SIMD (Single Instruction Multiple Data): Los kernels ejecutan operaciones idénticas (indicadas en el kernel) en un conjunto de datos.
- SPMD (Single Program Multiple Data): Cuando las operaciones ejecutadas son muy distintas debido a condiciones internar y barreras. Los work-item ejecutan el mismo "programa", el kernel, pero sus acciones son muy distintas.

2.3. OpenCL y FPGAs

Las FPGAs son solo uno de tantos dispositivos heterogéneos en los cuales se puede utilizar OpenCL. Sin embargo, aportan grandes ventajas [5]:

- Una FPGA proporciona un hardware reconfigurable con el cual se puede diseñar un acelerador personalizado para cada aplicación. De esta manera se añade la flexibilidad que aportan las FPGAs a la libertad que permite OpenCL. Los límites en este caso no son los tipos de datos u operaciones que nos imponen dispositivos como CPUs o GPUs, sino que las herramientas de síntesis crean el hardware para adaptarlo a nuestras necesidades.
- Acorta enormemente los tiempos de desarrollo con respecto al uso de lenguajes de descripción de hardware como Verilog o VHDL.
- Resuelve por si solo los problemas relacionados con la lógica de control y los buses de comunicaciones
- La transcodificación de algoritmos escritos en C/C++ es sencilla, ya que OpenCL tiene la misma base y es muy similar.

Como se puede deducir a partir de las secciones anteriores, para utilizar OpenCL en una FPGA, necesitaremos una parte que ejecute el programa host. Esta parte puede ser tanto interna como externa a la FPGA, por lo que tendremos dos tipos: aquellas donde existe un procesador interno embebido a la FPGA donde se ejecuta el host y aquellas que necesitan instalarse en un sistema con un procesador externo que ejecute la parte host. Durante la realización de este trabajo, se utilizó mayoritariamente una FPGA Alaric Arria 10, de la compañía Altera, ahora Intel FPGA, que necesita instalarse en un PC y se comunica a través del puerto PCI express.

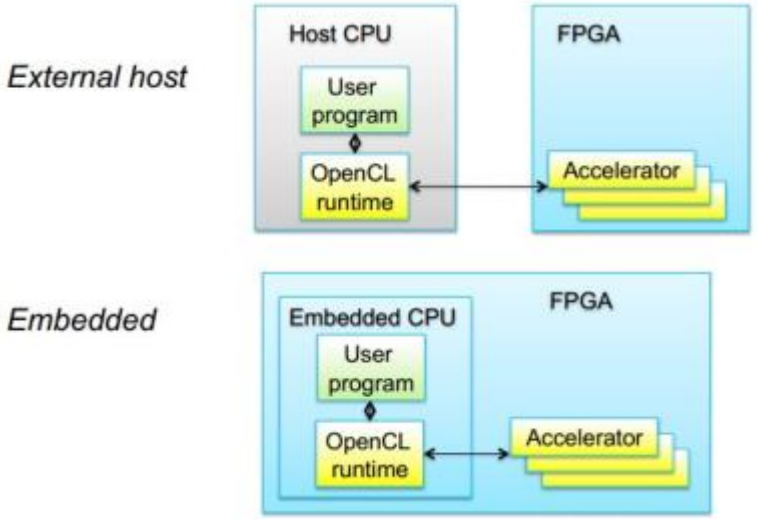


Figura 9 Host en FPGA

Para poder utilizar OpenCL en una FPGA son necesarias distintas herramientas [6], que en la mayoría de casos consisten en una suite que engloba otras herramientas. Veamos de forma general las más importantes:

- Altera Software Development Kit (SDK) for OpenCL (AOCL): consiste en una suite de herramientas proporcionada por Altera para sus FPGAs que incluye todo lo necesario para diseñar y cargar aplicaciones en la FPGA. Entre sus componentes se incluyen un compilador para OpenCL y otras herramientas para construir, ejecutar y depurar aplicaciones OpenCL.
- Altera Runtime Enviroment (RTE). Se trata de un conjunto de componentes necesarios para compilar y ejecutar la aplicación host y que este pueda utilizar la FPGA como acelerador. Normalmente viene dentro del SDK y se instala a la vez.
- Board Support Package (BSP): su función es permitir la comunicación entre el host y la FPGA. Un BSP es un diseño de referencia que tiene archivos de descripción de la plataforma, librerías, drivers, IPs, interfaces, etc. Estos archivos forman la parte de más bajo nivel del diseño. Son necesarios para utilizar el compilador, ya que este necesita un diseño base para ajustarse al dispositivo utilizado y aprovechar correctamente todos los recursos hardware disponibles. Este BSP es proporcionado por el fabricante de la placa, no por el fabricante de la FPGA. En este caso es ReFLEX CES quien lo proporciona.

2.4. Intel FPGA SDK Programming Flow

El proceso a la hora de crear, compilar y ejecutar una aplicación OpenCL utilizando las herramientas proporcionadas por Intel FPGA para sus dispositivos se esquematiza en la imagen siguiente:

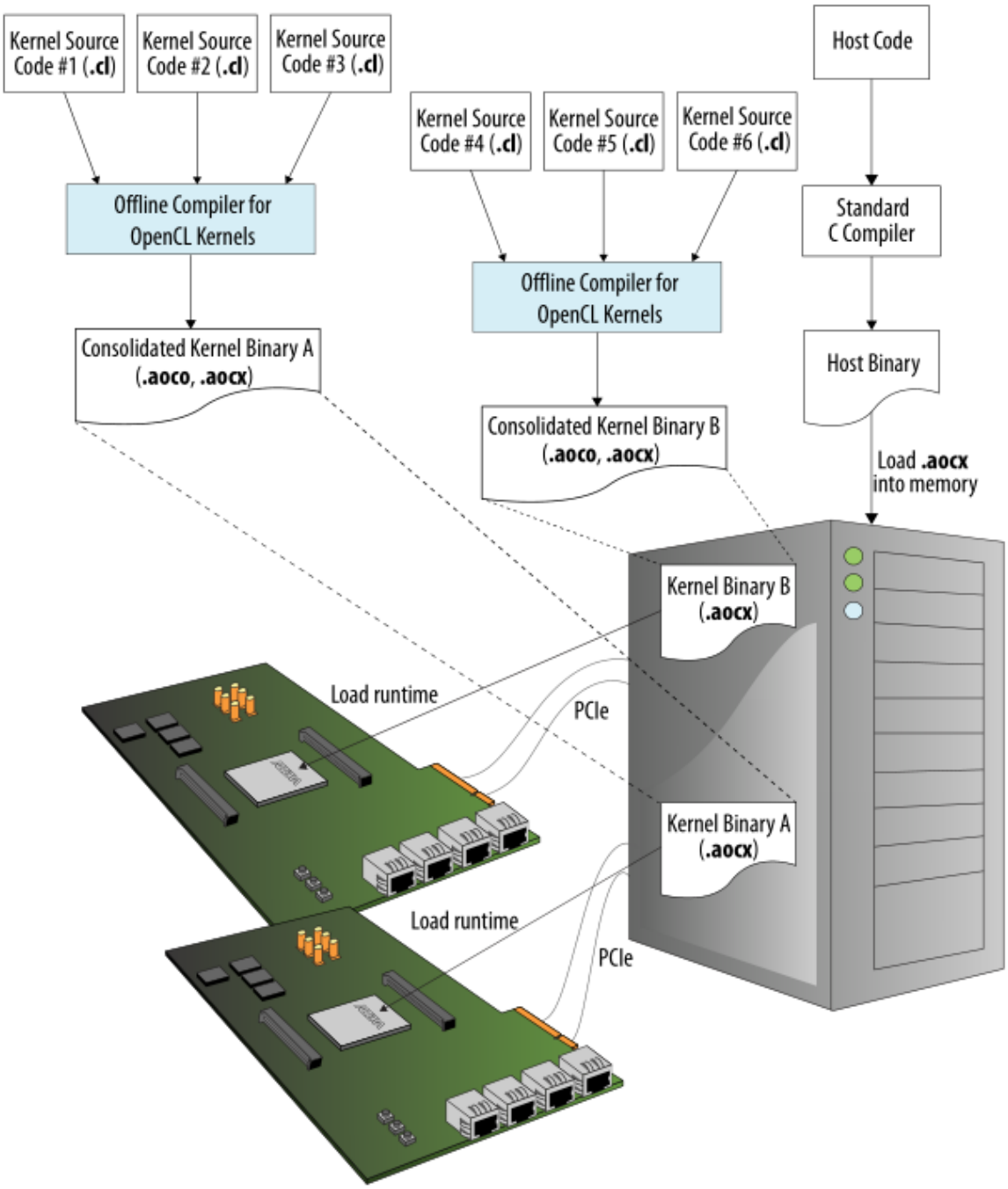


Figura 10 Programming Flow

En primer lugar, es el programador quien debe crear los archivos de código que contienen el kernel (archivos .cl) y la aplicación host (en el lenguaje disponible elegido). Los kernel se compilan en un largo proceso llevado a cabo por el offline compiler, proporcionado en el SDK de Intel FPGA [7]. El proceso consta de 3 fases que son:

- Conversión del código kernel escrito a HDL
- Compilación del diseño con la herramienta Quartus
- Creación del archivo de programación .aocx

Para la compilación de los kernels es necesario el uso del BSP, de forma que se adapten a la placa destino, teniendo en cuenta su hardware y estructura. El proceso de compilación del kernel da como resultado varios archivos, de los cuales son los más importantes:

- Altera Offline Compiler Executable file (.aocx): es el archivo de configuración del hardware de la FPGA.
- Altera Offline Compiler Object file (.aoco): es un archivo intermedio que contiene información necesaria para etapas posteriores de compilación.
- Una carpeta con varios archivos entre los que se encuentran el proyecto Quartus compilado, reports creados por la herramienta de compilación (que dan información de recursos utilizados, tipo de implementación, etc) y logs creados durante la compilación.

Una vez compilados los kernel se cargan en la FPGA, de nuevo utilizando la herramienta de Intel FPGA.

En cuanto a la parte del host, al estar escrito en un lenguaje “tradicional”, el proceso no tiene ninguna variación. Se utiliza el compilador apropiado y la aplicación se ejecuta como se haría normalmente. Al utilizar la API de OpenCL, este programa host es capaz de manejar los kernels y ejecutar la aplicación.

3. Desarrollo

3.1. Estudio del algoritmo propuesto por Intel FPGA y conversión a coma fija

En esta primera sección se trabajará con el algoritmo propuesto por Intel FPGA para la multiplicación de matrices utilizando sus FPGAs con OpenCL [8]. Se trata de una implementación en la cual se divide las matrices en bloques, siendo calculados los elementos dentro del bloque de forma concurrente.

El primer estudio consistirá en modificar distintos parámetros tratando de adaptar y optimizar el algoritmo para nuestra FPGA, de manera que el rendimiento sea el máximo. A continuación, se experimentará cambiando la multiplicación en coma flotante por una multiplicación en coma fija. Para esto nos apoyaremos en librerías externas escritas en Verilog, desarrollando el proceso necesario para realizar este cambio.

3.1.1. Algoritmo de Intel FPGA

El objetivo de este algoritmo, como cualquiera escrito en OpenCL, es encontrar operaciones que se puedan paralelizar y memoria que se pueda compartir. En una multiplicación de matrices se puede paralelizar el cálculo de cada elemento de la matriz resultado (llamada C de aquí en adelante), ya únicamente depende de la fila y columna correspondientes de las matrices origen (llamadas A y B ahora) y estos datos están disponibles desde el principio.

Sin embargo, como ya se trató anteriormente, estos datos al ser transferidos al dispositivo OpenCL estarán en memoria global, más lenta que aquella que usan los work-item dentro de un work-group, memoria local.

Para matrices pequeñas podríamos pensar en utilizar un solo work-group y llenar su memoria local con todos los datos de las matrices A y B. De esta forma podríamos acceder una sola vez a la memoria global para copiar los datos a memoria local, de manera que el resto de accesos sean más rápidos. No es una mala idea si únicamente vamos a utilizar matrices muy pequeñas cuyos datos quepan en la memoria local de nuestro dispositivo.

Sin embargo, esta solución no es escalable ni eficiente para matrices grandes. Aquí se ve la necesidad de dividir las matrices en bloques de un tamaño fijo, de manera que a cada work-group se le asigne la tarea de calcular un bloque de la matriz resultado. Así, cada work-group leerá una vez de memoria global los datos que se le asignen y todos los work-item dentro de este work-group accederán a memoria local para realizar los cálculos. Esta forma de acceder a memoria es mucho más eficiente. Lo más probable es que el dispositivo OpenCL no tenga memoria local suficiente para almacenar todos los datos de las matrices. Por tanto, los work-group se irán turnando el uso de esta memoria.

Al dividir la matriz en bloques, la multiplicación se realiza tratando estos bloques (que siempre deberán ser de tamaño idéntico) como si fueran un solo número, pero aplicando las operaciones propias de trabajar con matrices. Podemos observar el proceso en la siguiente figura.

$$AB = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \cdot \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right] = \left[\begin{array}{c|c} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ \hline A_{21}B_{11} + A_{22}B_{21} & A_{22}B_{12} + A_{22}B_{22} \end{array}\right]$$

Figura 11 Multiplicación de matrices por bloques

Si dividimos la matriz A en los bloques A11, A12, A13 y A14 y la matriz B de la misma forma, obtenemos dos matrices donde cada elemento es una submatriz. Así, se puede realizar la multiplicación de la forma que se hace cuando los elementos son números, pero teniendo en cuenta que las operaciones entre los elementos (multiplicación y suma) serán operaciones propias de matrices (en vez de multiplicaciones simples de números). Por tanto, los tamaños de los bloques deben ser tales que estas operaciones entre submatrices sean posibles. Es decir, el número de columnas de las submatrices de A y de filas de las submatrices de B deben ser iguales.

El algoritmo creará un work-group para calcular cada submatriz de la matriz C. Por tanto, durante la ejecución de cada work-group el proceso consistirá en cargar una submatriz de la fila correspondiente de A y otra de la columna correspondiente de B. Una vez cargadas se realizará una multiplicación de matrices tradicional, pero calculando todos los elementos de forma concurrente y cargando los datos de la memoria local. Ambos aspectos aportan una eficiencia muy alta.

A continuación, se acumulará el resultado en una memoria privada y se cargará la siguiente submatriz de A y de B, para volver a repetir el proceso recorriendo la fila y columna. Al acabar la última multiplicación y acumulación, ya está calculado un bloque de la matriz final. En este momento es cuando se cargan los datos a la memoria global.

Dentro del work-group, cada work-item tendrá asignada la tarea, primero, de cargar a memoria local un elemento de la submatriz, segundo, calcular el elemento resultado correspondiente, tercero, realizar la acumulación de ese elemento y, por último, cargar el elemento final en la memoria global.

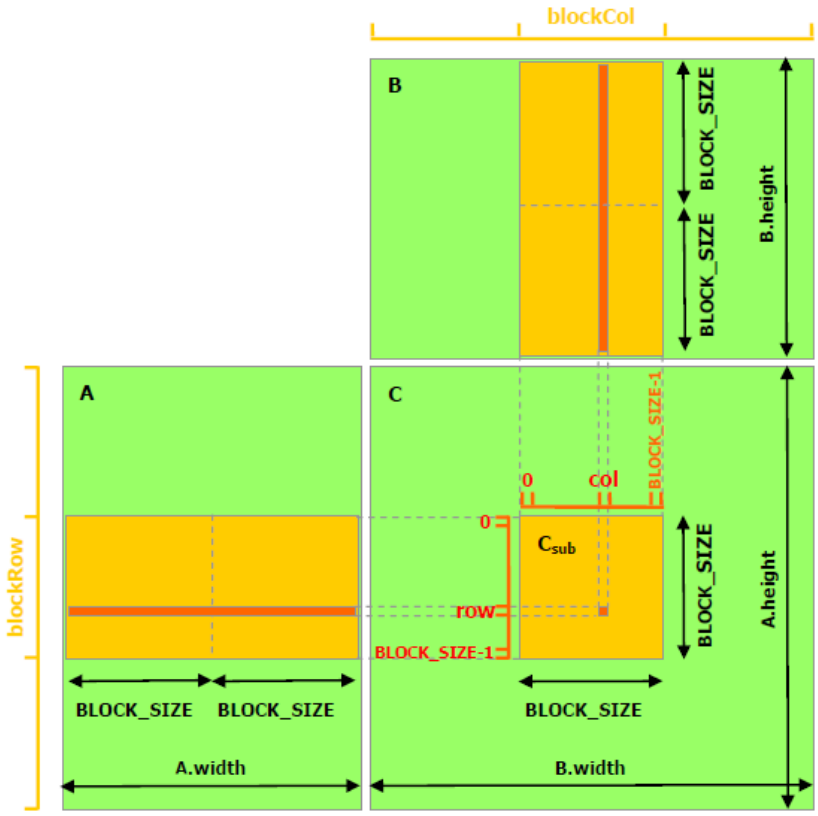


Figura 12 Multiplicación de submatrices

Veamos a continuación, como se implementa este algoritmo en un kernel OpenCL:

```
#include "../host/inc/matrixMult.h"

#ifndef SIMD_WORK_ITEMS
#define SIMD_WORK_ITEMS 4
#endif

__kernel
__attribute((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
__attribute((num_simd_work_items(SIMD_WORK_ITEMS)))
void matrixMult( // Input and output matrices
                 __global float *restrict C,
                 __global float *A,
                 __global float *B,
                 // Widths of matrices.
                 int A_width, int B_width)
{
    // Local storage for a block of input matrices A and B
    __local float A_local[BLOCK_SIZE][BLOCK_SIZE];
    __local float B_local[BLOCK_SIZE][BLOCK_SIZE];

    // Block index
    int block_x = get_group_id(0);
    int block_y = get_group_id(1);

    // Local ID index (offset within a block)
    int local_x = get_local_id(0);
    int local_y = get_local_id(1);

    // Compute loop bounds
    int a_start = A_width * BLOCK_SIZE * block_y;
    int a_end   = a_start + A_width - 1;
    int b_start = BLOCK_SIZE * block_x;

    float running_sum = 0.0f;

    for (int a = a_start, b = b_start; a <= a_end; a += BLOCK_SIZE, b += (BLOCK_SIZE * B_width))
    {
        A_local[local_y][local_x] = A[a + A_width * local_y + local_x];
        B_local[local_x][local_y] = B[b + B_width * local_y + local_x];

        barrier(CLK_LOCAL_MEM_FENCE);

        #pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k)
        {
            running_sum += A_local[local_y][k] * B_local[local_x][k];
        }

        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // Store result in matrix C
    C[get_global_id(1) * get_global_size(0) + get_global_id(0)] = running_sum;
}
```

Figura 13 Kernel OpenCL del algoritmo Intel FPGA

En primer lugar vemos un include, donde lo único que se hace es definir el parámetro BLOCK_SIZE. A continuación vemos el parámetro SIMD, que lo explicaremos más adelante. Empieza el kernel y vemos el atributo “reqd_work_group_size”. Este nos indica cuál va a ser el tamaño del work-group. Como cada work-group se va a ocupar de un bloque, será de dos dimensiones (1 solo índice en la tercera dimensión) de tamaño BLOK_SIZE x BLOCK_SIZE.

Con __global declaramos las variables guardadas en memoria global que se le van a pasar a nuestro kernel, es decir, los buffers que el host rellenará y el kernel leerá. Por otro lado, las variables que se guardan en memoria local se declaran con __local. En este caso, A_local y B_local son las que guardarán los datos de la submatriz con la que se esté operando.

El resto de variables, se almacenarán en la memoria privado del work-item. Entre estas tenemos 4 variables que sirven para controlar en qué work-item y work-group estamos trabajando. Mediante la instrucción "get_group_id(X)" obtenemos el identificador del work-group en la dimensión X del NDRange. Como el work-group se corresponde con el bloque de la matriz, podremos saber en qué submatriz se debe operar. De igual forma, la instrucción "get_local_id(X)" nos da el identificador del work-item en la dimensión X, lo que se traduce en el elemento dentro de la submatriz. Al estar trabajando en dos dimensiones, X solo será 0 o 1. El resultado de estas dos instrucciones variará dependiendo de qué work-item lo ejecute. Es esta variación la que nos permite realizar unas operaciones u otras en función del work-item y/o work-group en ejecución.

La información en los buffers, variables globales A, B y C, no se encuentra ordenada en dos dimensiones, sino que se da en un vector unidimensional y se indica el ancho de la matriz. Por ellos, es necesario calcular los puntos de comienzo y final de los bloques basándose en los anchos y el tamaño de los bloques. Como se puede observar, estos puntos de inicio y final variarán para cada work-group, porque cada uno tendrá un identificador (guardados en las variables block_x y block_y) distintos.

La última variable declarada se corresponde con el acumulador, que será interno al single-work ítem y por tanto habrá uno por cada elemento de la matriz.

Una vez declaradas todas las variables empieza el bucle en el que se recorrerá la fila y columna correspondiente (compuestas de submatrices) de las matrices A y B. La primera operación es copiar los datos de las submatrices de A y B a la memoria local. Cada ejecución del kernel copiará un elemento distinto, por lo que cuando acaben esta instrucción todos los work-item, los bloques estarán copiados por completo. Vemos aquí la necesidad de poder coordinar los distintos work-item para que avancen a las siguientes instrucciones

cuando todos hayan completado la carga de datos. Esta función se consigue con la instrucción “barrier(CLK_LOCAL_MEM_FENCE)”, que indica a los work-item a esperar hasta que todos hayan llegado a ese punto. Una optimización que se realiza en esta carga de datos es transponer la matriz (intercambiar filas por columnas), de manera que el acceso a esta variable de dos dimensiones sea más eficiente.

A continuación, mediante un bucle, se realiza la multiplicación acumulación de los elementos de la fila y columna de los bloques para calcular el elemento correspondiente del work-item. Como este número de elementos es conocido (siempre va a haber BLOCK_SIZE elementos en cada fila y columna) se puede desenrollar el bucle, lo que hace más rápida la multiplicación. Recordemos que ahora se está accediendo a memoria local para coger los datos. Al acabar el bucle, un resultado parcial habrá quedado en el acumulador, que, sumado al resto de resultados parciales de posteriores multiplicaciones, será el elemento final de la matriz C. De la misma forma que antes, no podemos empezar otra operación hasta que todos los work-item no hayan obtenido su resultado de la acumulación.

El bucle principal se repetirá hasta recorrer por completo la fila y columna (de submatrices) correspondientes para calcular el elemento objetivo de este work-item. Una vez acabado, el valor acumulado ya contiene el valor correcto del elemento de la matriz C, por lo que se puede enviar al buffer de salida. De nuevo este buffer es un vector unidimensional, por lo que la posición escrita deberá ser calculada a partir de la información de las dimensiones del work-item.

En esta ocasión, se utilizan las instrucciones “get_global_id(X)”, que proporciona el identificador absoluto del work-item dentro del índice NDRange en la dimensión X, y “get_global_size(X)”, que indica el tamaño del índice NDRange en la dimensión X. Es posible utilizar también las instrucciones que nos proporcionan identificadores de work-item y work-group por separado, pero habría sido necesario realizar más operaciones para combinarlos. Como coincide el tamaño de la matriz con el índice NDRange, en este caso la mejor opción es utilizar los índices absolutos.

3.1.2. Adaptación y optimización para Arria 10

En este apartado modificaremos varios parámetros con el objetivo de mejorar el rendimiento [9]. Hay que destacar que esta adaptación será únicamente válida para la Arria 10, ya que el rendimiento y adaptación va a depender de los recursos de los que dispone la FPGA. Estos mismos parámetros para otra FPGA pueden no ser óptimos.

BLOCK_SIZE: Como ya se ha visto, se trata del tamaño de las submatrices en las que dividimos las matrices de nuestro problema. La utilidad principal es que cada work-group realice operaciones para calcular una de estas submatrices de la matriz resultado. Dentro del work-group se comparten recursos más rápidos, pero es necesario manejar estos work-group, y sobre todo, copiar los datos de memoria global a memoria local, lo que consume un tiempo en el que no se está calculando nada. Ajustar este parámetro es clave para un buen rendimiento, y su tamaño optimo dependerá del tamaño de las matrices.

SIMD: Single Instruction Multiple Data. Se trata de aplicar la misma operación a un conjunto de datos, de manera que se obtiene paralelismo a nivel de datos. Una única unidad de control maneja varias unidades de procesamiento. En OpenCL, implica aumentar el número de operaciones que realiza cada work-item. A consecuencia de esto se reducirá el tamaño del índice NDRange. Por ejemplo, si un work-item tiene un SIMD de 4, realizará 4 veces más operaciones, por lo que serán necesarios 4 veces menos work-items.

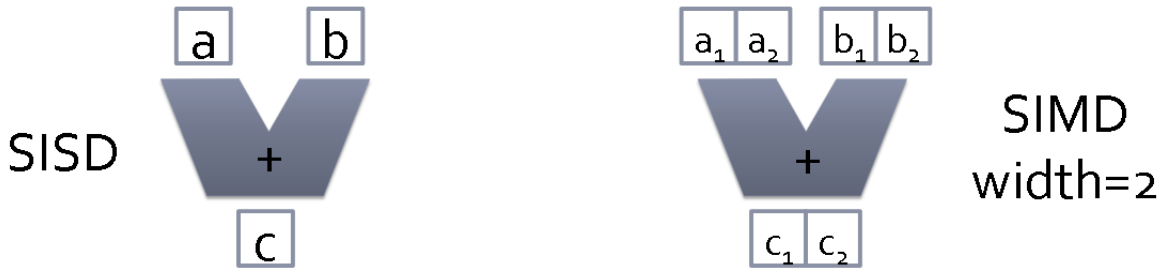


Figura 14 Parámetro SIMD

Compute units: Para conseguir un mayor troughput, el compilador puede generar múltiples compute units para cada kernel. Cada compute units puede ejecutar múltiples work-groups de forma simultánea. Sin embargo, todos los work-group de la compute unit deberán ejecutar el mismo kernel. El hardware de La FGPA asigna los work-groups a las compute units adicionales disponibles que no están al máximo de su capacidad. De esta forma se reduce el tiempo necesario para la ejecución de la tarea y se aumenta el rendimiento.

No se necesita programar esta asignación de tareas por parte del usuario, ya que el "hardware scheduler" de OpenCL lo realiza de forma automática. Sin embargo el uso de compute units aumenta el número de accesos a memoria, de comunicación entre compute units y una serie de operaciones que gastan ancho de banda y capacidad de procesamiento de la FPGA. Por esta razón, es necesario encontrar un equilibrio en el cual el rendimiento sea óptimo. El número de compute units a implementar debe ser especificado por el usuario utilizando el atributo "num_compute_units(N)". IntelFGPA recomienda usar al menos 3 veces más work-groups que compute units.

Comparación de resultados

Para obtener los parámetros que mejor se adaptan a nuestras matrices, se han hecho pruebas con distintos tamaños de matrices. Las pruebas consisten en compilar y ejecutar el algoritmo utilizando distintas combinaciones de parámetros. Para los diferentes tamaños se muestran las pruebas cuyos resultados son los más significativos y que aportan una visión del efecto que tienen los parámetros utilizados.

Los datos disponibles para evaluar el rendimiento son los tiempos de ejecución (media de varias medidas), tanto desde el punto de vista del host como del device; los recursos utilizados en el dispositivo y el consumo eléctrico estimado, obtenido mediante la herramienta de Quartus. Veremos primero el resultado de utilizar matrices de tamaño A (512x 256), B (256 x256) y C (512 x 256):

BLOCKSIZE	SIMD	Comp units	Host Time (ms)	Device time (ms)	Logic utilization	Block memory bits	Power consumption (mW)	Max freq (MHz)
8	4	1	6,765	6,684	43855 (17%)	2989322 (7%)	6859,73	229,04
8	4	2	7,575	7,501	53758 (21%)	5091338 (12%)	7917,49	255,42
16	4	1	3,484	3,397	45371 (18%)	3087626 (7%)	7358,64	269,39
16	8	1	3,421	3,349	50928 (20%)	3398282 (8%)	7938,72	253,16
16	8	2	3,594	3,505	68179 (27%)	5909258 (14%)	9174,15	226,34
32	8	1	2,011	1,935	57322 (23%)	3672330 (8%)	8717,53	237,69
64	4	1	1,426	1,342	67908 (27%)	6214314 (14%)	10064,41	236,4
64	8	1	1,364	1,286	83944 (33%)	5225738 (12%)	10960,23	203,95

En primer lugar, observamos que el tiempo medido por el host es algo mayor al tiempo medido por el device. Esta diferencia se debe a que, desde el punto de vista del host, cuando este lanza la instrucción de ejecutar el kernel, todavía hay un tiempo de latencia hasta que esta orden llega al dispositivo OpenCl. Existen ciertas operaciones previas que toman algo de tiempo y crean una pequeña latencia, de ahí esta pequeña diferencia de tiempo.

Para un tamaño de BLOCKSIZE 8, el cambio del parámetro SIMD no da resultados significativos, por ello no está mostrado. Además, contrariamente a lo que cabría esperar, añadir compute units empeora los tiempos, suponiendo además un incremento en el uso de recursos. Lo mismo ocurre cuando el tamaño del BLOCKSIZE es 16, por lo que para estos tamaños de matrices, el incremento de este parámetro no es una buena idea.

Observamos que el único parámetro que parece incrementar el rendimiento es el tamaño de bloque. Con cada tamaño más grande el tiempo de ejecución disminuye, así como el consumo estimado. Por el contrario, la cantidad de recursos se incrementa. Una vez llegados a un tamaño de bloque de 64 x 64, el uso de un SIMD de 8 en vez de 4 produce un incremento ya apreciable en el rendimiento, aunque no significativo. Deducimos por tanto, que la mejor combinación para este tamaño de matrices es un tamaño de bloque de 64 x 64.

Debido a restricciones de la FPGA y los recursos del servidor donde se realiza la compilación, no es posible compilar kernels con un tamaño de bloque superior a 64. Por tanto, probaremos con matrices de tamaños menores para comprobar si en estos casos un tamaño de bloque menor se acerca más a una solución óptima. Veamos el tamaño A (256 x 128), B (128 x 128) y C (256 x 128).

BLOCKSIZE	SIMD	Comp Units	Host Time (ms)	Kernel time (ms)	Logic utilization	Block memory bits	Power consumption (mW)	Max freq (MHz)
8	4	1	0,769	0,69	43855 (17%)	2989322 (7%)	6859,73	229,04
8	4	2	0,535	0,452	53758 (21%)	5091338 (12%)	7917,49	255,42
8	4	4	0,477	0,404	75466 (30%)	8637450 (20%)	9435,08	239,63
8	4	8	0,576	0,525	119871 (48%)	15729162 (36%)	--	215,93
16	4	1	0,459	0,371	45371 (18%)	3087626 (7%)	7358,64	269,39
16	8	1	0,283	0,225	50928 (20%)	3398282 (8%)	7938,72	253,16
32	8	1	0,243	0,176	57322 (23%)	3672330 (8%)	8717,53	237,69
64	4	1	0,301	0,212	67908 (27%)	6214314 (14%)	10064,41	236,4
64	8	1	0,282	0,192	83944 (33%)	5225738 (12%)	10960,23	203,95

Para este caso observamos que el uso de más compute units aumenta el rendimiento cuando el tamaño del bloque es de 8 x 8, aunque únicamente hasta 4. A partir de ahí el rendimiento decae. Cuando el tamaño del bloque es mayor, el efecto es nulo o negativo, por lo que no se han mostrado estas medidas. Hasta ahora las compute units no aportan ninguna mejora a nuestro kernel.

Para este tamaño de matrices, de nuevo aumentar el Blocksize es la mejor forma de incrementar el rendimiento, aunque esta vez el máximo rendimiento se produce para un Blocksize de 32. En cuanto al parámetro SIMD, con un tamaño de 8 se aumenta mucho el rendimiento con respecto a 4. Con tamaños mayores la mejora apenas es apreciable.

Determinamos que la mejor combinación para este tamaño de matrices es un SIMD de 8 con un Blocksize de 32. En esta ocasión se puede observar que el tamaño de bloque 32 es el punto óptimo comparando con los resultados de tamaño de bloque 16 y 64, que obtienen prácticamente el mismo valor de tiempo de ejecución, mayor al de 32.

Ahora utilizaremos un tamaño de matrices de 64 x 64. Esperamos comprobar si con un tamaño menor, la solución óptima tiene un tamaño de bloque menor también.

BLOCKSIZE	SIMD	Comp Units	Host Time (ms)	Kernel time (ms)	Logic utilization	Block memory bits	Power Cosumption (mW)	Max freq (MHz)
8	4	1	0,145	0,082	43855 (17%)	2989322 (7%)	6859,73	229,04
8	4	2	0,133	0,069	53758 (21%)	5091338 (12%)	7917,49	255,42
8	4	4	0,145	0,085	75466 (30%)	8637450 (20%)	9435,08	239,63
16	4	1	0,117	0,057	45371 (18%)	3087626 (7%)	7358,64	269,39
16	8	1	0,101	0,049	50928 (20%)	3398282 (8%)	7938,72	253,16
16	8	2	0,129	0,057	68179 (27%)	5909258 (14%)	9174,15	226,34
32	8	1	0,105	0,042	57322 (23%)	3672330 (8%)	8717,53	237,69
64	4	1	0,127	0,05	67908 (27%)	6214314 (14%)	10064,41	236,4
64	8	1	0,108	0,046	83944 (33%)	5225738 (12%)	10960,23	203,95

Lo primero que observamos es que, con este tamaño de matrices, los resultados son muy similares. Las matrices ahora son muy pequeñas y por tanto las diferencias de tiempo de ejecución entre un kernel y otro van a reducirse mucho.

Igual que en los casos anteriores observamos que el uso de más compute units no aporta prácticamente ninguna mejora y además consume más recursos. En cuanto al tamaño de bloque, se observa un rendimiento algo superior con un tamaño de 16, aunque las implementaciones con 32 y 64 están prácticamente al mismo nivel. El uso de un SIMD de 8 en lugar de 4 mejora algo el rendimiento, aunque no al nivel que lo hace con matrices de tamaño superior. Por la combinación de rendimiento, uso de recursos y consumo estimado, vemos que la solución óptima es la de BlockSize 16 y SIMD 8.

Durante las medidas realizadas hemos podido comprobar como para matrices de tamaños grandes, aumentar el tamaño del BlockSize es la mejor forma de aumentar el rendimiento. Sin embargo este aumento tiene un límite marcado por los recursos disponibles. Otra opción es aumentar el parámetro SIMD, que consigue mejorar bastante los tiempos. Por el contrario, para este algoritmo en concreto, el uso de más compute units aumenta muy poco el rendimiento y consume bastantes más recursos de la FPGA.

3.1.3. Implementación en coma fija

Tras haber analizado el rendimiento del algoritmo original para distintos tamaños de matrices, lo modificaremos para operar en coma fija. Las operaciones en coma fija son mucho más sencillas que en coma flotante, aunque desde hace tiempo existen unidades de hardware dedicadas a acelerar operaciones en coma flotante. Nuestra FPGA posee unidades para realizar multiplicaciones tanto en coma fija como en coma flotante, por lo que probaremos qué implementación es más rápida.

Usage Example	Multiplier Size (Bit)	DSP Block Resources
Medium precision fixed point	Two 18 x 19	1
High precision fixed or Single precision floating point	One 27 x 27	1
Fixed point FFTs	One 19 x 36 with external adder	1
Very high precision fixed point	One 36 x 36 with external adder	2
Double precision floating point	One 54 x 54 with external adder	4

Figura 15 Multiplicadores Arria 10

Lo que si sabemos desde un principio es que si las operaciones en coma fija no se realizan con un número de bits suficiente, el error de cuantificación producido será muy significativo.

Para realizar esta implementación se creará una librería de OpenCL. Esta librería consiste en un archivo que contiene una o varias funciones. Es posible crear una librería usando OpenCL o usando código RTL. Una vez creada, estas funciones pueden ser utilizadas dentro de los kernels.

En nuestro caso crearemos una librería RTL, a partir de unos módulos Verilog proporcionados que implementan la conversión de coma flotante a coma fija, la multiplicación en coma fija y la conversión de nuevo a coma flotante. Estos módulos has sido proporcionados por el tutor y están sobradamente probados.

En cuanto al tamaño de los datos en coma fija, estos utilizarán 18 bits. La razón es que los multiplicadores en coma fija son de 18 x 19, como podemos ver en la figura anterior, y de esta forma podemos aprovecharlos al máximo. Para poder representar una precisión suficiente con la que el error de cuantificación sea muy reducido se han utilizado 15 bits para la parte decimal, quedando otros 3 para la parte entera. Con este formato, el valor absoluto de los datos de entrada no será mayor de 2.

Son necesarios algunos archivos más para crear la librería. El usuario debe crear a mano un archivo xml para dar información que utilizará el compilador para integrar el componente RTL. Esta información consiste en especificar la latencia de la función, si esta es fija o no, si se pueden combinar varias instancias… Además este fichero indica la ruta a los archivos que forman el código RTL.

Otro archivo necesario es un fichero de cabecera (.h), en el que se indique el nombre la función, los datos de entrada y los tipos de los datos de las funciones implementadas. En nuestro caso, este fichero consiste en una línea: “float matriz_mult(float a, float b);”.

El último archivo es un modelo en C de la función RTL. Este archivo es necesario para las emulaciones. Para las compilaciones para hardware no se utiliza. Para nuestro caso: “double matriz_mult(double a, double b) {return a*b;}”.

Cuando todos los archivos están creados es el momento de empaquetarlos en un archivo aoco para poder ser utilizados en la creación de la librería. El comando a ejecutar es:

```
$ aoc -c <archivo_xml>.xml -o <nombre_paquete>.aoco
```

Una vez creado el archivo aoco, creamos la librería a partir de él. Podría crearse a utilizando varios archivos aoco, tanto los que tienen funciones escritas en OpenCL y los que las tienen escritas en RTL. Para crear la librería utilizamos el comando:

```
$ aocl library create -o <library file name>.aoclib <object file 1>.aoco
[<object file 2>.aoco ... <object fileN>.aoco]
```

En el código OpenCL, se debe incluir la cabecera (archivo .h) creada para cada una de las librerías. Y una vez hecho esto ya se pueden utilizar las funciones definidas en la librería. En el caso de nuestro algoritmo, el único cambio necesario es sustituir la multiplicación dentro del segundo bucle por una llamada a la función de multiplicación en coma fija: "matriz_mult(A_local[local_y][k], B_local[local_x][k]);".

De esta forma, todas las multiplicaciones que se realicen entre los distintos elementos de la matriz, se realizarán utilizando nuestra propia función. Esto significa que el hardware utilizado no será el destinado a coma flotante, sino el destinado a coma fija. Tras esto ya podemos compilar el kernel incluyendo la librería. Se utiliza el comando:

```
$ aoc -l nombre_libreria.aoclib -L ruta_carpeta_libreria
-I ruta_carpeta_libreria path_archivo_cl -o
```

Comparación de resultados

Ahora veremos si utilizar la coma fija ha aportado alguna ventaja en cuanto a rendimiento de los kernels. Para ello buscaremos de nuevo la combinación óptima de parámetros para cada tamaño de matrices.

Para el tamaño de 512x256, tenemos las siguientes medidas:

BLOCKSIZE	SIMD	Comp Units	Host Time (ms)	Kernel time (ms)	Logic utilization	Block memory bits	Power Cosumption (mW)	Max freq (MHz)
8	4	1	8,589	8,496	54409 (22%)	2995978 (7%)	6757,29	152,46
8	4	2	8,151	8,081	74548 (30%)	5104650 (12%)	7515,15	149,16
16	4	1	4,723	4,662	67593 (27%)	3127050 (7%)	7066,08	138,52
16	8	1	3,491	3,417	93736 (37%)	3511690 (8%)	7926,81	137,66
16	8	2	3,26	3,198	153220 (61%)	6136074 (14%)	9056,2	122,41
64	4	1	1,863	1,796	144807 (58%)	6270090 (14%)	8959,39	124,73

Observamos unos resultados con un comportamiento muy parecido a la versión en coma flotante. El mayor incremento en rendimiento se produce al aumentar el tamaño del BlockSize, siendo la mejor combinación la de BlockSize 64. Comparando los resultados con la versión en coma flotante, vemos que, mientras a esta le cuesta 1,426ms realizar la multiplicación, a la versión en coma fija le cuesta 1,863ms. Además, mirando la utilización de lógica vemos un incremento del uso desde un 27% a un 58%, lo que es muy significativo.

Veamos qué ocurre con el tamaño de 256 x 128.

BLOCKSIZE	SIMD	Comp Units	Host Time (ms)	Kernel time (ms)	Logic utilization	Block memory bits	Power Cosumption (mW)	Max freq (MHz)
8	4	1	1,211	1,084	54409 (22%)	2995978 (7%)	6757,29	152,46
8	4	2	0,684	0,624	74548 (30%)	5104650 (12%)	7515,15	149,16
8	4	4	0,57	0,48	117896 (47%)	8664074 (20%)	8706,88	142,93
16	4	1	0,672	0,599	67593 (27%)	3127050 (7%)	7066,08	138,52
16	8	1	0,404	0,333	93736 (37%)	3511690 (8%)	7689,52	127,87
16	8	2	0,363	0,288	153220 (61%)	6136074 (14%)	9056,2	122,41
32	4	1	0,516	0,424	92751 (37%)	3863690 (9%)	7709,03	131,13
64	4	1	0,322	0,252	144807 (58%)	6270090 (14%)	8959,39	124,73

En este set de medidas observamos un comportamiento muy distinto del resto en cuanto al uso de las compute units. Para tamaños de bloque de 8, el uso de 2 compute units prácticamente reduce el tiempo de computación a la mitad. Para el caso de 16 esta reducción es menor pero también significativa, por lo que en estos dos 2 casos el rendimiento se ve afectado positivamente al incluir más compute units.

Aun así, la compilación que mejor rendimiento posee es de nuevo la de tamaño de bloque de 64 x 64. Con respecto a la mejor solución para coma flotante, pasamos de 0,243 ms de tiempo de ejecución a 0,322 ms. El rendimiento sigue siendo menor al usar coma fija, y el consumo de recursos mayor.

Una compilación interesante habría sido con tamaño de bloque 64 y SIMD 8, sin embargo, debido a los recursos limitados esta compilación no producía un resultado válido y no se pudo realizar.

Por último, veamos el tamaño de matriz 64 x 64.

BLOCKSIZE	SIMD	Comp Units	Host Time (ms)	Kernel time (ms)	Logic utilization	Block memory bits	Power Cosumption (mW)	Max freq (MHz)
8	4	1	0,16	0,101	54409 (22%)	2995978 (7%)	6757,29	152,46
8	4	2	0,14	0,081	74548 (30%)	5104650 (12%)	7515,15	149,16
16	4	1	0,128	0,069	67593 (27%)	3127050 (7%)	7066,08	138,52
16	8	1	0,119	0,059	93736 (37%)	3511690 (8%)	7689,52	127,87
16	8	2	0,165	0,065	153220 (61%)	6136074 (14%)	9056,2	122,41
32	4	1	0,117	0,055	92751 (37%)	3863690 (9%)	7709,03	131,13
64	4	1	0,125	0,061	144807 (58%)	6270090 (14%)	8959,39	124,73

De la misma forma que usando coma flotante, el rendimiento de las distintas compilaciones para este tamaño de matrices es muy similar, las diferencias son muy sutiles. Aun así, es posible determinar que en este caso tamaños de bloque de 16 y 32 son los que proporcionan el rendimiento más alto. En coma flotante, el mejor tiempo obtenido es 0,101 ms, mientras que usando coma fija el mejor caso se queda en 0,117 ms. De nuevo la implementación en coma fija no consigue superar a la de coma flotante.

Observamos que, a menor tamaño de las matrices, los tamaños de bloque más pequeños rinden mejor, igual que ocurre en el caso de utilizar coma flotante. Además, vemos que en ningún caso la implementación en coma fija ha superado en rendimiento ni recursos utilizados a la de coma flotante. Una de las razones es que los datos se guardan en coma flotante y es necesario convertirlos a fija para operar, para después convertirlos de nuevo a coma flotante. Estas conversiones requieren cierto tiempo que la implementación en como flotante no (ya que existen unidades hardware para realizar la multiplicación de ambas formas) y por tanto el rendimiento se verá reducido. Además del rendimiento, el uso de recursos aumenta al tener que introducir estos conversores.

Concluimos que operar en coma fija, con los nuevos dispositivos que incluyen DSPs capaces de realizar operaciones en coma flotante, ya no es necesario para reducir el tiempo de cálculo. Si que ayuda a reducir tiempos en dispositivos como la Cyclone V donde no disponemos de estas unidades.

3.2. Creación de un entorno de trabajo multiplataforma para OpenCL

El uso de OpenCL implica la utilización de múltiples herramientas. En primer lugar se necesita una instalación del SDK, BSP y demás herramientas vistas en la sección anterior. Además, es necesario tener una instalación de un lenguaje de programación que use la API de OpenCL, junto con su entorno de desarrollo, que puede ser un simple editor de textos y un compilador que se ejecute por comandos o puede ser un entorno de desarrollo integrado (IDE).

Por otro lado, las FPGA (y también GPU) de una cierta calidad suelen ser bastante caras y la tendencia es conectarlas físicamente a un servidor al que se accede de forma remota. Este servidor puede dar acceso a varios usuarios y además suele estar protegido con SAIs y otros elementos. La desventaja es el hecho de tener que trabajar en una máquina remota y por tanto depender de la conexión con ella.

Todos estos elementos se intentarán abordar en la creación de este entorno de trabajo multiplataforma para OpenCL, que está basado en aplicaciones como Jupyter y Anaconda y el uso de PyOpenCL.

En los últimos años, el uso de Python ha crecido de forma exponencial, gracias a su simplicidad y a su gran comunidad, además de otras razones. El uso de Python para manejar kernels de OpenCL facilita el comienzo de nuevos usuarios con OpenCL y la creación de prototipos. Estas son algunas de las ventajas que nos proporciona PyOpenCL.

El objetivo de esta sección es describir la instalación, configuración y uso de un entorno abierto para el uso de PyOpenCL. Este entorno debe ser fácilmente accesible para cualquier usuario con unos mínimos conocimientos técnicos, sin depender de uso experiencia en el uso de Python u OpenCL.

El punto de partida consiste en un ordenador con un sistema operativo Linux, al cual se han conectado distintos dispositivos OpenCL, como una FPGA y una GPGPU. En este caso concreto la FPGA se trata de una Alaric Arria 10 y la GPU una Nvidia GeForce GTX Titan GK110.

Algo que no se puede pasar por alto y en lo que no tenemos elección es en instalar las herramientas propias de Intel FPGA (como el SDK, BSP…), ya explicadas en secciones anteriores.

3.2.1. Anaconda y PyOpenCL

La primera herramienta elegida para nuestro entorno de desarrollo es la distribución para Python: Anaconda. Se trata de una distribución libre y de código abierto. Una de sus principales ventajas son el uso de un eficiente administrador de paquetes con el que podemos añadir fácilmente distintas librerías desde el repositorio de Anaconda o desde fuentes externas.

Otra de las ventajas que nos ofrece es la posibilidad de tener distintos virtual environments. En un environment se instalan los paquetes mencionados anteriormente, además de utilizar una versión de Python concreta. De esta forma es posible tener en cada uno de ellos unos paquetes distintos e incluso utilizar versiones de Python distintas, lo que nos permite ajustar muy bien el virtual environment al proyecto en el que estamos trabajando.

Para instalar Anaconda, tenemos dos opciones [1]. La primera es un paquete de la distribución que incluye varios paquetes científicos y matemáticos, con los que es posible desarrollar multitud de aplicaciones. La instalación de este paquete nos permite tener, nada más acabar la instalación, un entorno muy completo, a costa de un tiempo de instalación mayor y un mayor espacio en el disco duro.

La segunda opción es instalar una versión muy reducida llamada Miniconda. Esta versión incluye los paquetes básicos para que la instalación sea funcional. Está enfocada a usuarios con mayores conocimientos técnicos que saben exactamente qué paquetes necesitan. De esta forma se empieza a personalizar la instalación desde el primer momento. Con la cantidad de paquetes tan reducida que incluye se podría decir que es prácticamente obligatorio tener que instalar manualmente algunos paquetes tras acabar la instalación. Por otro lado, el espacio requerido en disco es mínimo y puede ser una buena opción para sistemas limitados por su hardware.

Para este trabajo se ha utilizado el paquete completo de Anaconda, que se puede descargar de forma gratuita desde [1].

Tras la instalación, se crea un virtual environment, donde trabajaremos de aquí en adelante. Esto se hace con el siguiente comando, en el cual indicamos la versión de Python que queremos utilizar:

```
$ conda create -n mypyoclenv python=3.6.3 anaconda
```

Una vez creado se activa con:

```
$ conda activate mypyoclenv
```

PyOpenCL

Uno de los problemas que cualquier persona debe afrontar al empezar con un nuevo lenguaje de programación o API es el tiempo de aprendizaje necesario para adaptarse a las nuevas herramientas, tales como librerías, compiladores, etc.

En el caso de OpenCL, la primera acción de enumerar las plataformas y dispositivos disponibles, siguiendo la documentación, requiere usar C o C++ y crear un script completo que será necesario compilar. Durante este proceso es muy probable que ocurran distintos errores: errores de compilación, errores de ejecución, incluso que no se reporte ningún error pero la ejecución sea incorrecta. A la hora de probar un nuevo lenguaje o API, es

preferible usar un intérprete que evite el clásico proceso de crear un programa, compilar, editar, recompilar, ejecutar, depurar… Utilizando el intérprete la única acción consiste en escribir las instrucciones, obteniendo un resultado inmediato o un error. Este hipotético error será por tanto claro y determinado ya que sabremos en qué instrucción se ha producido.

Tras usar el intérprete para encontrar la forma correcta de cumplir nuestro objetivo es el momento de transcribir el código para la implementación definitiva. Seguir este procedimiento puede ahorrar mucho tiempo cuando el usuario es un principiante.

Una de las mejores soluciones para utilizar un intérprete es el uso de Python. Nos ofrece una interfaz en línea de comandos utilizando el Python Shell. Con esta solución es fácil probar comandos de Python. Este intérprete puede ser invocado usando el comando "$ python" en una terminal (tras la instalación de Anaconda).

```
$ python
Python 3.6.3 |Anaconda, Inc.| (default, Oct 13 2017, 12:02:49)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello world")
Hello world
```

Figura 16 Intérprete Python

El objetivo de PyOpenCL es añadir esta accesibilidad a OpenCL de manera que sea posible trabajar utilizando el lenguaje de programación Python en lugar de C o C++, beneficiándonos del uso de este intérprete interactivo. PyOpenCL permite acceder a la API completa de OpenCL desde Python [11], sin tener que dejar de lado nuestro entorno de Python. Además, la velocidad de PyOpenCL es similar a la de C/C++ ya que su capa base está escrita en este lenguaje de programación.

PyOpenCL es uno de tantos paquetes que pueden descargarse e instalarse en Anaconda. Para ello se descarga el paquete desde su sitio web [12]. Si seguimos la guía de instalación recomendada, solo las GPU y CPU serán detectadas. Será posible compilar y ejecutar aplicaciones en estos dispositivos, sin embargo las FPGAs no serán detectadas. Para este tipo de instalaciones, se requieren algunas modificaciones respecto a la instalación "clásica", ya que el uso de PyOpenCL en FPGAs no es muy popular por el momento y la documentación y ejemplos disponibles son muy escasos.

Estas modificaciones consisten en añadir paths en los archivos de instalación de PyOpenCL, hacia algunas librerías de IntelFPGA, de manera que la FPGA sea detectada. En el script descargado setup.py se debe añadir la siguiente información:

- Librerías usadas por Altera:

  ```
  default_libs = ["OpenCL","alterahalmmd", "altera_apb_14_0_mmd","alteracl"]
  ```

- Path hasta la carpeta de librerías de la placa y host. Estas deberán estar dentro de la carpeta de instalación de OpenCL, apuntada por la variable de entorno ALTERAOCLSDKROOT:

  ```
  $ echo $ALTERAOCLSDKROOT
  /opt/altera_pro/16.0.2_b222/hld
  ```

  Añadimos la siguiente línea

  ```
  default_libdir = ["/opt/altera_pro/16.0.2_b222/hld/
  board/alaric_hpc_16.0.2_b222/linux64/lib","/opt/altera_pro/16.0.2_b222/hld
  /host/linux64/lib"]
  ```

- Flags apuntando a las carpetas de las librerías usadas antes:

  ```
  default_ldflags = ["-Wl,--no-as-needed","-Wl,--
  rpath,/opt/altera_pro/16.0.2_b222/hld/board/alaric_hpc_16.0.2_b222/linux64
  /lib","-Wl,--rpath, /opt/altera_pro/16.0.2_b222/hld/host/linux64/lib",
  "/opt/altera_pro/16.0.2_b222/hld/board/custom_platform_toolkit/migration/r
  ef_design/linux64/lib/"]
  ```

Tras haber incluído esta información el siguiente paso consiste en configurar la instalación ejecutando el script modificado. Aunque no es obligatorio, es recomendable incluir de nuevo el path hacia las librerías:

```
$ python configure.py
--cl-inc-dir=/opt/altera_pro/16.0.2_b222/hld/ host/include
--cl-lib-dir=/opt/altera_pro/16.0.2_b222/hld/ host/linux64/lib
--cl-libname=OpenCL
```

Figura 17 Ejecución script modificado

Tras ejecutar este script, se crea un nuevo archivo ejecutable llamado "siteconf.py", dentro del cual es necesario añadir más información:

```
CL_PRETEND_VERSION = '1.1'
CL_LIB_DIR = ['/opt/altera_pro/16.0.2_b222/hld/board/
alaric_hpc_16.0.2_b222/linux64/lib','/opt/altera_pro/16.0.2_b222/hld/host/linu
x64/lib','/opt/altera_pro/16.0.2_b222/hld/board/custom_platform_toolkit/migrat
ion/ref_design/linux64/lib']
CL_LIBNAME = ['reflex_alaric_hpc_16_0_mmd','OpenCL', 'alteracl',
'alterahalmmd','altera_apb_14_0_mmd']
```

La parte más importante añadida es la librería de los dispositivos OpenCL, en este caso, la librería de la placa Alaric: "reflex_alaric_hpc_16_0_mmd".

Finalmente, es el momento de compilar la instalación de PyOpenCL. Como requisito, necesitamos que nuestra versión de gcc sea compatible con el estándar gnu++1. Si no lo fuera, es necesario actualizar la versión de gcc. El comando a utilizar para la compilación es:

```
$ LDFLAGS="-Wl,--no-as-needed" python setup.py build
```

Y, por último, instalamos el paquete PyOpenCL con:

```
$ python setup.py install
```

Una prueba muy simple que podemos realizar tras acabar la instalación es listar los dispositivos OpenCL detectados. Para ello utilizaremos el método get_platforms().

```
(mypyoclenv) bash-4.1$ python
Python 3.6.3 | packaged by conda-forge | (default, Dec 9 2017, 16:18:26)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyopencl as cl
>>> cl.get_platforms()
[<pyopencl.Platform 'NVIDIA CUDA' at 0x16fb230>, <pyopencl.Platform
'Altera SDK for OpenCL' at 0x7fbd4006ad00>]
```

Figura 18 Dispositivos OpenCL detectados

Tal y como queríamos, se han detectado tanto la GPU como la FPGA.

3.2.2. Uso de PyOpenCL

En esta sección se explicará cómo crear un script utilizando PyOpenCL. Se verá cómo realizar los mismos pasos que cuando se utiliza C/C++ : seleccionar el dispositivo, crear el contexto, crear los buffers, etc.

Los kernel utilizados para las demostraciones son los mismos kernels usados anteriormente. Por tanto, no será necesario realizar el proceso de compilación, que tanto tiempo consume, para cada uno de ellos.

La novedad ahora es el uso de la GPU. En esta plataforma, al no tener un hardware configurable no es necesario un proceso de compilación similar al de la FPGA (explicado en la sección 2.4). No necesitamos generar un proyecto Quartus ni traducir el kernel a descripciones HDL. Es un entorno mucho menos flexible pero donde la compilación dura segundos. Guardando el código del kernel en una variable y ejecutando una única instrucción de Python se cargará el kernel en la GPU.

En primer lugar, hay ciertos módulos que deben importarse, entre los que se encuentran el propio PyOpenCL o Numpy, un paquete para cálculos matemáticos muy popular y muy completo, que nos permitirá operar con matrices:

```
import pyopencl as cl
import pyopencl.array as cl_array
from time import time
import timeit
import numpy as np
```

El primer paso consiste en buscar las plataformas disponibles y seleccionar una de ellas. Este paso se realiza con "cl.get_platforms()", que devuelve una lista de las plataformas disponibles, cuyos métodos "name" y "vendor" dan información sobre ellas. De forma análoga, para cada plataforma, el método "get_devices()" devuelve una lista de dispositivos de esa plataforma que contienen métodos para obtener información, como "name" y "device type". Por ejemplo, podemos saber si el dispositivo se trata de una GPU o FPGA usando "`cl.device_type.to_string (mydevice.type)`".

Una vez que se ha seleccionado el dispositivo, se crea el contexto con "ctx = cl.Context([mydevice])", donde mydevice es el dispositivo seleccionado previamente. El siguiente paso consiste en programar el dispositivo con el kernel. Este paso es diferente para GPU y para FPGA. En el caso de la GPU, hemos de tener el código del kernel guardado como string en una variable, lo que se puede hacer fácilmente utilizando las tres comillas. Los parámetros del kernel se deben manejar por separado, lo que se puede hacer fácilmente utilizando un "diccionario", una estructura de datos en Python. Estas dos variables, junto con la que generamos al crear el contexto se deben pasar al método "Program", generando un resultado que se compilará (método "build") y cargará.

En el caso de la FPGA, no es necesario volver a compilar, por lo que el proceso de carga empieza abriendo el archivo previamente compilado como un fichero binario. Se utilizan los métodos "Program" y "build" para cargar, teniendo como argumentos el contexto, el dispositivo seleccionado y el archivo binario.

Por último, en el caso de aplicaciones de un solo kernel, se selecciona el kernel utilizando su nombre. En la siguiente imagen podemos observar el código para cargar el kernel en la GPU y en la FPGA:

```
if (cl.device_type.to_string(mydevice.type)=="GPU"):
    kernel_params = {"block_size": block_size}
    prg = cl.Program(ctx, KERNEL_CODE % kernel_params).build()
else:
    binary = open("matrix_mult_32.aocx", "rb").read()
    prg= cl.Program(ctx, [mydevice], [binary]).build()

mykernel = prg.matrixMult
```

Figura 19 Carga de kernel en GPU y FPGA

A continuación, se crea la cola utilizando el método CommandQueue de PyOpenCL, al que es necesario pasarle información del contexto y distintas opciones de la cola. Entre estas opciones se encuentra la propiedad PROFILING_ENABLE, que permitirá habilitar las medidas de tiempo de ejecución de los kernels utilizando eventos, lo que proporciona unas medidas bastante exactas.

Y, por último, se crean los buffers. Es necesario indicar si son de lectura o de escritura, su tamaño y, en el caso de llenarlos con datos en la creación, la variable que contiene dichos datos.

```
d_a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_a)
d_b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_b)
d_c_buf = cl.Buffer(ctx, mf.WRITE_ONLY, size=h_c.nbytes)
```

Figura 20 Creación de los buffers

En este momento ya se pueden ejecutar los kernel, indicando sus parámetros, los buffers y la cola a la que se asigna la ejecución. Por ejemplo:

```
event = mykernel(queue, (c_width, c_height), (block_size, block_size), d_c_buf, d_a_buf, d_b_buf, a_width, b_width)
event.wait()
```

Figura 21 Ejecución del kernel

Esta ejecución devuelve un evento que permitirá controlar el estado de la ejecución y realizar medidas de tiempo. Con su método wait, el script esperará a que la ejecución en el dispositivo OpenCL finalice. Este tiempo será el que usemos para evaluar el rendimiento del kernel. Podemos obtener las medidas desde el punto de vista del evento con los métodos: “event.profile.end” y “event.profile.start”.

Después de ejecutar el kernel, el siguiente paso es copiar los datos en el buffer de salida a una variable en nuestro script, para poder utilizar los datos. PyOpenCL nos da el método “enqueue_copy” con el que se hace de una forma muy sencilla: “cl.enqueue_copy(queue,h_c,d_c_buf)”.

Y esta es toda la parte donde se usa PyOpenCL. El resto, verificación, medidas de tiempo y muestra de resultados se puede hacer de forma sencilla utilizando Python, que para muchos usuarios es más fácil de utilizar que C o C++.

Como hemos podido observar, este proceso consta de prácticamente los mismos pasos que C/C++ pero simplificando bastante las instrucciones a ejecutar. Además, la sintaxis de Python simplifica bastante el escribir código (aunque esto depende bastante de a qué lenguajes de programación está acostumbrado el usuario) y el uso del intérprete puede ser de gran ayuda a la hora de debuggear o probar nuevas funciones.

Medida del tiempo de ejecución

Ya que uno de los objetivos de este trabajo es acelerar el algoritmo de multiplicación de matrices, es necesario establecer un sistema de medidas de tiempos para evaluar el rendimiento de las distintas implementaciones. Tal y como hemos adelantado antes, el uso de eventos de PyOpenCL para medir el tiempo de ejecución nos proporciona medidas bastante exactas. Además, veremos algún método más que, aunque no sea parte de PyOpenCL, contribuye a completar y añadir herramientas en nuestro entorno de trabajo.

En Python, tenemos distintas funciones que nos permiten medir tiempo [13]. De entre ellas, las dos más utilizadas son “time” y “timeit”, siendo esta última la más exacta y la más indicada para medir el tiempo de ejecución de pequeños trozos de código. Estas medidas se compararán con las obtenidas a través de los eventos. Habrá una diferencia, ya que utilizando eventos se mide el tiempo on-chip, es decir, desde el punto de vista del OpenCL device, mientras que con las funciones de Python obtenemos una medida desde el punto de vista del host, que comprende más operaciones para la transferencia y por tanto incluye esta latencia y medirá más tiempo.

De forma experimental, se ha observado que la primera ejecución de un kernel es, en ocasiones, mucho más lenta que las siguientes, por lo que la medida de esta primera ejecución puede considerarse como un error y no debe ser tenida en cuenta.

Para solventar esta situación, es importante realizar un “calentamiento”, que consiste en ejecutar el kernel algunas veces antes de empezar con las medidas de tiempo. Además, otra buena práctica consiste en ejecutar el kernel varias veces y tomar el valor medio como medida. Si además de esto, descartamos las primeras ejecuciones de este “conjunto definitivo” de medidas, el resultado será bastante más exacto.

Tanto con la función “time”, como la función “timeit” se miden los tiempos antes y después de ejecutar los kernel y se toma la resta como resultado. En el caso de los eventos de OpenCL, el valor será la resta entre el valor devuelto por los métodos “.profile.end” y “.profile.start”.

A continuación, tenemos el código en Python que ejecuta el kernel de multiplicación de matrices proporcionado por Altera, tanto en GPU como en FPGA:

```
 7  from __future__ import print_function
 8  from __future__ import division
 9
10  import pyopencl as cl
11  import pyopencl.array as cl_array
12  from time import time
13  import timeit
14  import numpy as np
15
16
17  def aligned_zeros(shape, boundary=64, dtype='float32', order='C'):
18      N = np.prod(shape)
19      d = np.dtype(dtype)
20      tmp = np.zeros(N * d.itemsize + boundary, dtype=np.uint8)
21      address = tmp.__array_interface__['data'][0]
22      offset = (boundary - address % boundary) % boundary
23      return tmp[offset:offset+N*d.itemsize].view(dtype=d).reshape(shape, order=order)
24
25  def aligned_rand(shape, boundary=64, dtype='float32', order='C'):
26      N = np.prod(shape)
27      d = np.dtype(dtype)
28      tmp = np.random.rand(N * d.itemsize + boundary).astype(np.float32)
29      address = tmp.__array_interface__['data'][0]
30      offset = (boundary - address % boundary) % boundary
31      return tmp[offset:offset+N*d.itemsize].view(dtype=d).reshape(shape, order=order)
32
33  KERNEL_CODE = """
34
35  #define BLOCK_SIZE %(block_size)d
36
37  __kernel
38  __attribute((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
39
40  void matrixMult( // Input and output matrices ...
46  { ...
102 }
103 """
```

Figura 22 Código Python

```
105  ##########MATRIX SIZE#########
106
107  block_size = 32
108  a_height = 256
109  a_width = 128
110  b_width = 128
111  b_height = a_width
112  c_width = b_width
113  c_height = a_height
114
115  if (a_width % block_size != 0):
116      print("Width of matrix A is not multiple of Block size")
117
118  if (a_height % block_size != 0):
119      print("Height of matrix A is not multiple of Block size")
120
121  if (b_width % block_size != 0):
122      print("Width of matrix B is not multiple of Block size")
123
124  ###MATRIX DATA
125  h_a = np.random.rand(a_height, a_width).astype(np.float32)
126  h_b = np.random.rand(b_height, b_width).astype(np.float32)
127  h_c = aligned_zeros((c_height, c_width))
128
129  ###########PLATFORM AND DEVICE SELECTION######
130  print("Choose platform")
131
132  i=0
133  for platform in cl.get_platforms():
134      print('[',i,'] ' + platform.name + ' ' + platform.vendor)
135      i+=1
136  platform_number = int(input())
137  print("\n")
138  platform = cl.get_platforms()[platform_number]
139
140  print('Choose device')
141  i=0
142  for device in platform.get_devices():
143      print('[',i,'] ' + device.name + ' ' + cl.device_type.to_string(device.type))
144      i+=1
145  device_number = int(input())
146  print("\n")
147
148  ###CREATE CONTEXT
149  mydevice = platform.get_devices()[device_number]
150  ctx = cl.Context([mydevice])
151  assert mydevice.local_mem_size > 0
152
153  ###PROGRAM DEVICE
154  if (cl.device_type.to_string(mydevice.type)=="GPU"):
155      kernel_params = {"block_size": block_size}
156      prg = cl.Program(ctx, KERNEL_CODE % kernel_params).build()
157  else:
158      binary = open("matrix_mult_32.aocx", "rb").read()
159      prg= cl.Program(ctx, [mydevice], [binary]).build()
160
161  mykernel = prg.matrixMult
162
163  ###RUN KERNEL
164  mykernel.set_scalar_arg_dtypes([None,None,None,np.uint32,np.uint32])
165
166  ###Queue
167  queue = cl.CommandQueue(ctx,properties=cl.command_queue_properties.PROFILING_ENABLE)
```

Figura 23 Código Python

```python
#Create and fill Buffer
mf = cl.mem_flags
t1 = timeit.default_timer()

d_a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_a)
d_b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_b)
d_c_buf = cl.Buffer(ctx, mf.WRITE_ONLY, size=h_c.nbytes)

t2 = timeit.default_timer()
push_time = t2-t1

##WARM-UP
count = 20
for i in range(count):
    event = mykernel(queue, (c_width, c_height), (block_size, block_size), d_c_buf, d_a_buf, d_b_buf, a_width, b_width)
    event.wait()

###MEASURES
compute_time_event = 0
compute_time_host = 0
compute_time = 0
compute_time_e = 0
count = 1000
j=10
for i in range(count):
    t1 = timeit.default_timer()
    event = mykernel(queue, (c_width, c_height), (block_size, block_size), d_c_buf, d_a_buf, d_b_buf, a_width, b_width)
    event.wait()
    t2 = timeit.default_timer()
    if i>=j:
        compute_time = compute_time + (t2-t1)
        compute_time_e = compute_time_e + (event.profile.end-event.profile.start)*1e-9

compute_time_host = compute_time/(count-j)
compute_time_event = compute_time_e/(count-j)


###COPY RESULT TO BUFFER
t1 = timeit.default_timer()
cl.enqueue_copy(queue,h_c,d_c_buf)
t2 = timeit.default_timer()
pull_time = t2-t1

###TIME RESULTS
total_time = compute_time_host+push_time+pull_time

print ("push+compute (host) +pull total [s]:", total_time)
print ("push [s]:", push_time)
print ("pull [s]:", pull_time)
print ("compute (host-timed) [s]:", compute_time_host)
print ("compute (event-timed) [s]: ", compute_time_event)

gflop = h_c.size * (a_width * 2.) / (1000**3.)
gflops = gflop / compute_time_host

print()
print ("GFlops/s:", gflops)
```

Figura 24 Código Python

```python
# cpu comparison ------------------------------------------------------------
t1 = timeit.default_timer()
count = 2
for i in range(count):
    h_c_cpu = np.dot(h_a,h_b)
t2 = timeit.default_timer()
cpu_time = (t2-t1)/count
# cpu comparison ------------------------------------------------------------
t1 = timeit.default_timer()
count = 20
for i in range(count):
    h_c_cpu2 = np.matmul(h_a,h_b)
t2 = timeit.default_timer()
cpu_time2 = (t2-t1)/count


print ("GPU==CPU:",np.allclose(h_c, h_c_cpu))
print()
print ("GPU==CPU2:",np.allclose(h_c, h_c_cpu2))
print()
print ("CPU 1 time (s)", cpu_time)
print()
print ("CPU 2 time (s)", cpu_time2)
print()

print ("OpenCL vs CPU 1 speedup (with transfer): ", cpu_time/total_time)
print ("OpenCL vs CPU 2 speedup (with transfer): ", cpu_time2/total_time)
print ("OpenCL vs CPU 1 speedup (without transfer): ", cpu_time/compute_time_event)
print ("OpenCL vs CPU 2 speedup (without transfer): ", cpu_time2/compute_time_event)
```

Figura 25 Código Python

Tras todos los pasos explicados anteriormente, vemos además el uso de dos funciones para multiplicar matrices, pertenecientes a la librería Numpy. Esta librería está codificada de forma muy eficiente, aunque solo hace uso de la CPU. Es por tanto un buen punto de referencia para dos comprobaciones:

1. Que nuestro kernel saca un resultado correcto: Se realiza la comprobación de los resultados de la CPU con los del dispositivo OpenCL mediante la función “allclose”, también de la librería Numpy.
2. Que el tiempo de ejecución del kernel es menor que el tiempo de ejecución utilizando únicamente una CPU.

A continuación, tenemos el resultado de la ejecución del código anterior en una GPU, utilizando un tamaño de bloque de 32x32 y un tamaño de matrices de A (256x128) y B (128x128). Como se observa, el tiempo medido a través de OpenCL es menor ya que no incluye el tiempo de latencia desde el host al device.

```
Choose platform
[ 0 ] NVIDIA CUDA NVIDIA Corporation
[ 1 ] Altera SDK for OpenCL Altera Corporation
1


Choose device
[ 0 ] alaric_v3_prod_hpc : Alaric_IDK (aclalaric_hpc_16_00) ACCELERATOR
0


Reprogramming device with handle 1
```

Figura 26 Elección dispositivo

```
push+compute (host) +pull total [s]: 0.0013053163062931613
push [s]: 0.0005244840867817402
pull [s]: 0.0005131233483552933
compute (host-timed) [s]: 0.0002677088711561278
compute (event-timed) [s]:  0.00018320602727272755

GFlops/s: 31.334815181032106
GPU==CPU: True

GPU==CPU2: True

CPU 1 time (s) 0.8241056189872324

CPU 2 time (s) 0.00035001919604837894

OpenCL vs CPU 1 speedup (with transfer):  631.3455328904367
OpenCL vs CPU 2 speedup (with transfer):  0.2681489493089716
OpenCL vs CPU 1 speedup (without transfer):  4498.2451246565
OpenCL vs CPU 2 speedup (without transfer):  1.9105222751614328
```

Figura 27 Resultado ejecución código Python

3.2.3. Jupyter

Hasta ahora, la creación de nuestro entorno de trabajo se ha centrado en simplificar el proceso de programación de forma que sea más sencillo desarrollar aplicaciones. Para este propósito se han aportado una serie de herramientas, como son PyOpenCL y librerías para medir tiempos de ejecución.

En esta sección introduciremos el uso de la herramienta Jupyter. Esta nos permitirá tener un directorio ordenado donde guardar todos los scripts desarrollados. Además, y esto constituye la principal ventaja de Jupyter, permite acceder a nuestros archivos, y a los dispositivos OpenCL instalados en nuestro servidor desde cualquier lugar de Internet.

Jupyter es una aplicación web para colaboración enfocada para proyectos de ciencia de datos. Su propósito es escribir y compartir código, texto, ecuaciones matemáticas… Integra código y sus resultados en un único documento, que permite ejecutar scripts en su interior.

El código se ejecuta en un servidor y los resultados se convierten en HTML. Estos se incorporan a la página web que el usuario está escribiendo, llamada Jupyter Notebook. Aunque es posible utilizar muchos lenguajes de programación en los Jupyter Notebooks, en nuestro caso nos enfocaremos en Python, ya que es el elegido para nuestro entorno multiplataforma.

Los notebooks son fáciles de compartir y varios autores pueden contribuir en un desarrollo al mismo tiempo, lo que hace de Jupyter una herramienta colaborativa perfecta para la programación. Los notebooks pueden ser configurados para poder ser accedidos desde cualquier punto de acceso a internet, en cualquier lugar del mundo.

El servidor de Jupyter se ejecuta en una sola máquina. En nuestro caso, en este mismo servidor se encuentran instalados los dispositivos OpenCL, por lo que el acceso a ellos es muy sencillo y se puede hacer desde la misma herramienta Jupyter sin tener que configurar nada más (siempre que estos dispositivos estén correctamente instalados en el servidor).

Instalación y funcionamiento

Jupyter es un paquete más que puede instalarse en la distribución de Python Anaconda. De hecho, si elegimos Anaconda (y no Miniconda), Jupyter se instalará por defecto [14]. En el caso de que el usuario prefiera instalar los paquetes de forma individual, el procedimiento para instalar Jupyter es el mismo que para cualquier paquete:

```
$ conda install jupyter
```

Como uno de los objetivos de Jupyter es hacer nuestro entorno de PyOpenCL muy accesible, es importante hacer segura la conexión. Así evitaremos que intrusos se metan en nuestro notebook y utilicen nuestros dispositivos.

Para ello crearemos un archivo de configuración en el cuál seleccionaremos varios parámetros. Se creará el archivo "jupyter_notebook_config.py" en el directorio ".jupyter". El comando a ejecutar es:

```
$ jupyter notebook --generate-config
```

Para añadir un login a la hora de entrar en Jupyter es necesario añadir una línea en el archivo de configuración que contenga el hash de nuestra contraseña. Un hash, también conocido como función resumen, consiste en una serie de operaciones que se le aplican a un texto, que da como resultado otro texto. Este resultado es prácticamente irrepetible y siempre será el mismo mientras los datos de entrada no varíen. Sin embargo, es imposible averiguar el texto original a través del resultado hash. Así, Jupyter solo deja acceder a quien escriba una contraseña cuyo hash coincida con el que nosotros hemos configurado (que será obtenido a partir de nuestra contraseña). El texto a añadir a nuestro fichero es:

```
c = get_config()
c.NotebookApp.password = u'sha1:HASHEDPASSWORD'
```

Para obtener el hash de nuestra contraseña, Jupyter nos lo da en un archivo .json tras ejecutar:

```
$ jupyter notebook password
```

Como Jupyter funciona a través de Internet, es una buena idea utilizar SSL para hacer más segura la conexión, de forma que nuestra contraseña no se envíe sin encriptar. Para esto, es necesario utilizar un certificado y una llave, que, si no tenemos unos, podemos crearlos y firmarlos nosotros mismos. Al hacer esto, probablemente nuestro navegador nos avise de que la conexión no es segura, y en un caso extremo, alguien puede copiarlos y hacerse pasar por nosotros con más facilidad (ya que el navegador avisará sin distinción de que la conexión no es segura). Depende de cada usuario el interés que tenga en proteger sus archivos, las amenazas que cree tener y los recursos que quiera invertir en protegerse.

Para crear de forma sencilla un certificado y una llave que tengan una validez de un año, podemos ejecutar en nuestro terminal el siguiente comando:

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout mykey.key
-out mycert.pem
```

Tras ejecutarlo habremos obtenido la clave mykey.key y el certificado mycertm.pem. Estos archivos serán referenciados en el archivo de configuración con las líneas:

```
c.NotebookApp.certfile = u'/absoulte/path/mycert.pem'
c.NotebookApp.keyfile = u'/absoulte/path /mykey.key'
```

Y tras esto, configuramos nuestro servidor para que pueda ser accedido desde cualquier IP a través del puerto seleccionado (PORTNUMBER)

```
c.NotebookApp.ip = '*'
c.NotebookApp.allow_origin = '*'
c.NotebookApp.port= PORTNUMBER
```

A continuación, seleccionamos la carpeta donde se guardarán los archivos de Jupyter con la línea:

```
c.NotebookApp.notebook_dir = u'/Jupyter/files/folder/'
```

Para el funcionamiento de Jupyter se requiere la variable de entorno LD_LIBRARY_PATH

```
import os
os.environ['LD_LIBRARY_PATH'] = '/opt/altera_pro/
16.0.2_b222/hld/host/linux64/lib'
c.Spawner.env.update('LD_LIBRARY_PATH')
```

Y finalmente se añaden las siguientes líneas:

```
c.NotebookApp.tornado_settings = {'headers': {'Content-Security-Policy':
"frame-ancestors 'self' *"}}
c.NotebookApp.extra_static_paths = ["static/custom/custom.js"]
```

En este momento Jupyter está completamente configurado y listo para ser usado. Puede ejecutarse desde el servidor con:

```
$ jupyter notebook
```

Para acceder a él, basta con escribir la dirección IP junto con el puerto escogido en nuestro navegador.: https://SERVERIP:PORTNUMBER

Un problema que podemos tener es que el firewall de nuestro servidor bloquee el acceso a Jupyter. En muchas distribuciones Linux (por ejemplo CentOS), el firewall integrado por defecto es Iptables. Bastará con añadir una regla de entrada para el puerto seleccionado [15] y podremos acceder a nuestro servidor Jupyter:

```
$ sudo iptables -I INPUT 1 -m state --state NEW -p tcp --dport
PORTNUMBER -j ACCEPT
```

Tras acceder por fin a nuestro servidor, podemos crear un nuevo notebook, editar uno existente, duplicarlo… Para nuestro caso particular en el que usamos kernels compilados por el software de Intel FPGA, archivos en formato .aocx, debemos colocarlos en la carpeta de archivos de Jupyter, configurada previamente. Así el servidor podrá localizar estos archivos y utilizarlos en la ejecución de nuestros scripts.

En la siguiente imagen tenemos una muestra del servidor Jupyter donde se han creado varios Notebooks, cada uno ejecutando un código distinto. Además se indica cuáles de ellos están en ejecución en el momento.

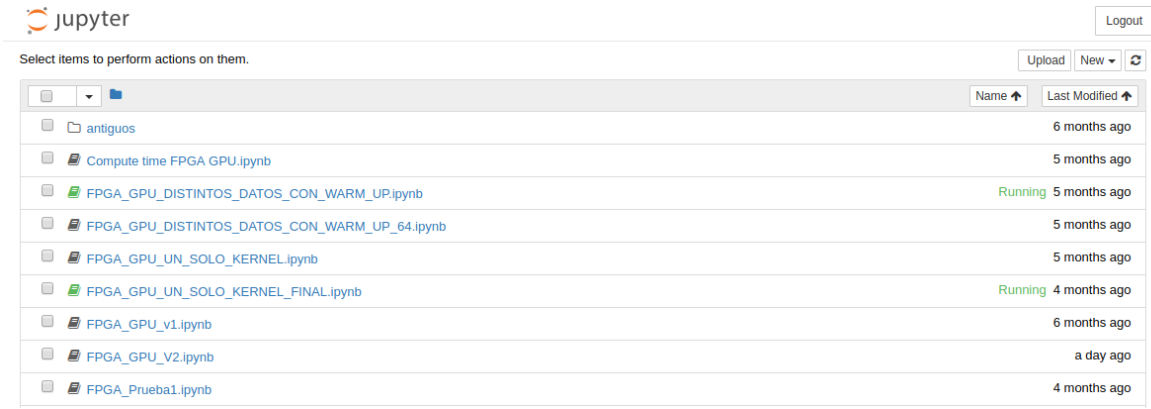


Figura 28 Pantalla principal Jupyter

Dentro de un notebook, es posible mezclar texto, código y otros formatos multimedia. Estos se introducen en unas estructuras llamadas celdas, que componen el notebook. Para el script en Python mostrado antes, podemos separar el código en distintas celdas, de manera que las distintas acciones queden bien diferenciadas y explicadas con un texto.

Los resultados se copian en el buffer de salida. También medimos el tiempo que tarda en realizarse este paso

In [61]:
```
t1 = timeit.default_timer()
cl.enqueue_copy(queue,h_c,d_c_buf)
t2 = timeit.default_timer()
pull_time = t2-t1
```

Tras esto, se representan los resultados de tiempo

In [62]:
```
total_time = compute_time_host+push_time+pull_time

print ("push+compute (host) +pull total [s]:", total_time)
print ("push [s]:", push_time)
print ("pull [s]:", pull_time)
print ("compute (host-timed) [s]:", compute_time_host)
print ("compute (event-timed) [s]: ", compute_time_event)

gflop = h_c.size * (a_width * 2.) / (1000**3.)
gflops = gflop / compute_time_host

print()
print ("GFlops/s:", gflops)
```

```
push+compute (host) +pull total [s]: 0.006103408126856643
push [s]: 0.004847120027989149
pull [s]: 0.0009865749161690474
compute (host-timed) [s]: 0.0002697131826984461
compute (event-timed) [s]:  0.00018376974141414118

GFlops/s: 31.101957702152504
```

Y por último verificamos que la multiplicación ha sido correcta y calculamos la aceleración conseguida

In [63]:
```
# cpu comparison ------------------------------------------------------------
t1 = timeit.default_timer()
count = 2
for i in range(count):
    h_c_cpu = np.dot(h_a,h_b)
t2 = timeit.default_timer()
```

Figura 29 Fragmento de código escrito en un Jupyer Notebook

Como podemos observar, los resultados se muestran en el mismo Notebook, justo donde acaba la celda que los genera. Esto permite tener un código muy ordenado y donde localizar errores resulta más sencillo. Además, y esta es la gran ventaja, no necesitamos acceder con VNC al servidor y utilizar las herramientas instaladas allí. Con tan solo saber la dirección IP y el puerto, somos capaces de conectarnos usando un explorador web y empezar a crear y ejecutar código de una manera muy sencilla.

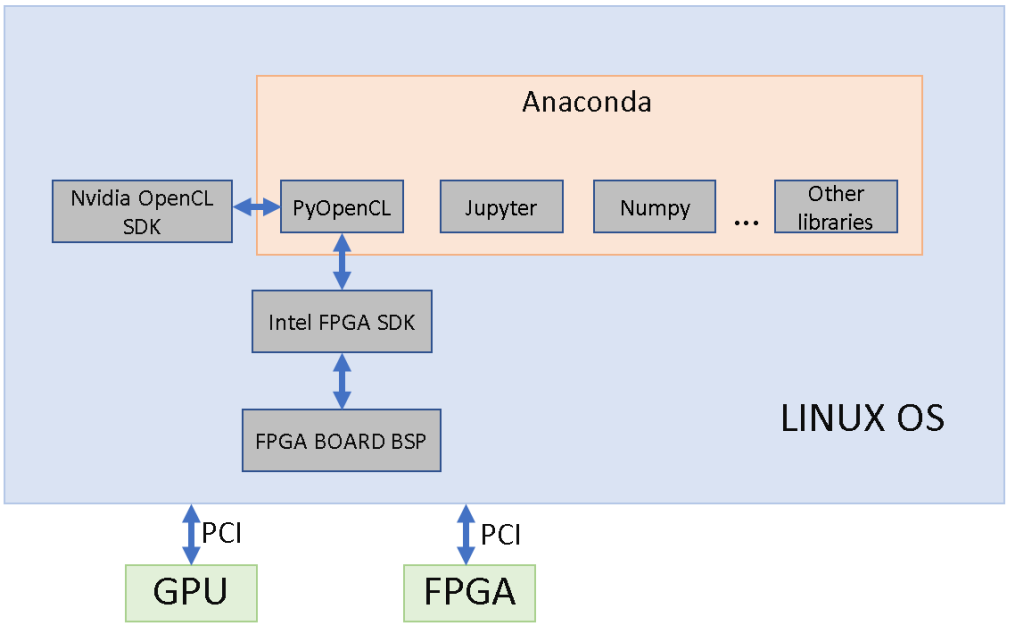


Figura 30 Esquema del entorno multiplataforma para OpenCL

Ejecución de los kernels ya probados

En lo referente al tiempo de ejecución para FPGAs, tenemos unos resultados similares a los que se consiguieron utilizando C++ para la parte host. La razón es que el dispositivo OpenCL es el mismo, y la parte de host escrita en Python, lo único que hace es utilizar la API de OpenCL. Como mucho, la diferencia está en el tiempo que le toma al host usar esta API, que es similar en ambos casos. No se va a acelerar una aplicación OpenCL por utilizar un lenguaje de programación distinto para la parte de host únicamente.

Sin embargo, la novedad ahora es el uso de la GPU, que es un dispositivo nuevo al cual le estamos cargando nuestro kernel. En este caso sí que hay una variación en el rendimiento del kernel.

Para ver esta diferencia, se ha utilizado el kernel evaluado en la sección anterior, con un tamaño de bloque de 32x32. Las medidas se han realizado tanto con la función “timeit” como con la información de los eventos, por lo que tenemos medidas desde el punto de vista del host (incluyendo transferencia de datos) y desde el punto de vista del device.

Función timeit				**Eventos OpenCL**			
Jupyter		**Terminal**		**Jupyter**		**Terminal**	
GPU	FPGA	GPU	FPGA	GPU	FPGA	GPU	FPGA
723 µs	246 µs	756 µs	255 µs	94.2 µs	181 µs	95.2 µs	184 µs

Como podemos observar, las medidas utilizando Jupyter y utilizando la terminal de Linux (ambas usando PyOpenCL) son similares. Las pequeñas diferencias son despreciables. Sin embargo, llama nuestra atención que, desde el punto de vista del host, la ejecución en GPU es mucho más lenta que en FPGA, pero desde el punto de vista del device, la GPU es más rápida. Como hemos comentado antes, el device no tiene en cuenta el tiempo de latencia desde que el host da la orden hasta que se hace efectiva en el device. Al ser tecnologías diferentes, sucede que este tiempo es mucho mayor en la GPU que en la FPGA. Por desgracia, este tiempo es algo en el que no tenemos nada (o casi nada) de control.

Conclusiones

Concluyendo, PyOpenCL ofrece la posibilidad de acceder a la API de OpenCL desde Python, permitiendo integrar todos los beneficios de Python en nuestro entorno de OpenCL. Algunos de estos beneficios son: un lenguaje de programación simple e intuitivo, la posibilidad de usar la herramienta de colaboración Jupyter y algunos buenos métodos de medida de tiempos.

El entorno de desarrollo propuesto tiene todas estas ventajas, ofreciendo una plataforma para diseñar aplicaciones de OpenCL tanto para profesionales como para principiantes. Este entorno facilita el uso de OpenCL desde los primeros pasos hasta tareas complicadas como el uso de múltiples dispositivos OpenCL.

3.3. Algoritmo multikernel para multiplicación de matrices

En esta última sección se desarrollará la creación de un algoritmo propio para implementar la multiplicación de matrices. Uno de los objetivos de este algoritmo es utilizar varios de los recursos que nos proporciona OpenCL, tales como canales y multikernel.

Este algoritmo está basado en la división de las matrices en bloques, pero se sirve de un array sistólico para la multiplicación de los elementos. Está compuesto por varios kernels, cada uno de ellos realiza una función muy específica. Además, las comunicaciones entre los distintos kernels se realizan utilizando canales. Se utiliza la memoria global lo menos posible.

3.3.1. Array sistólico

Un array sistólico es una red homogénea de elementos procesadores de datos a los que llamamos celdas o nodos. Cada nodo calcula de forma independiente un resultado parcial, a partir de los datos recibidos desde sus nodos vecinos [16]. El resultado y otros datos se envían a otros nodos vecinos. Las celdas se combinan de manera que estos resultados parciales, correctamente recogidos, conforman un resultado global.

Las celdas siempre realizan la misma acción, que suele ser una acción muy simple. La arquitectura del array es determinante para que el conjunto de celdas ejecuten el algoritmo deseado. En nuestro caso, las celdas calcularán una multiplicación entre dos números. Se organizarán las celdas y sus conexiones para conseguir un array sistólico que calcule la multiplicación de dos matrices.

Tradicionalmente existen varias arquitecturas de arrays sistólicos que calculan la multiplicación de matrices [17]. Una de ellas utiliza celdas como las mostradas en la figura siguiente. Estas tienen 3 entradas y 3 salidas, que se corresponden con los índices de las matrices A y B y con el resultado acumulado de la matriz C.

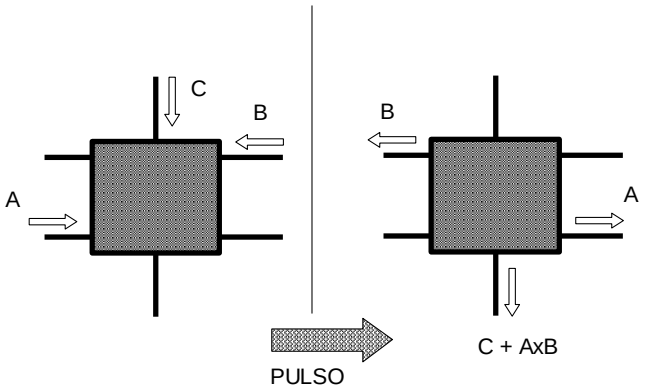


Figura 31 Celda multiplicación A x B

Estas celdas se organizan en una estructura como la siguiente, donde al introducirse los datos en el orden correcto, se obtiene como resultado la matriz C.

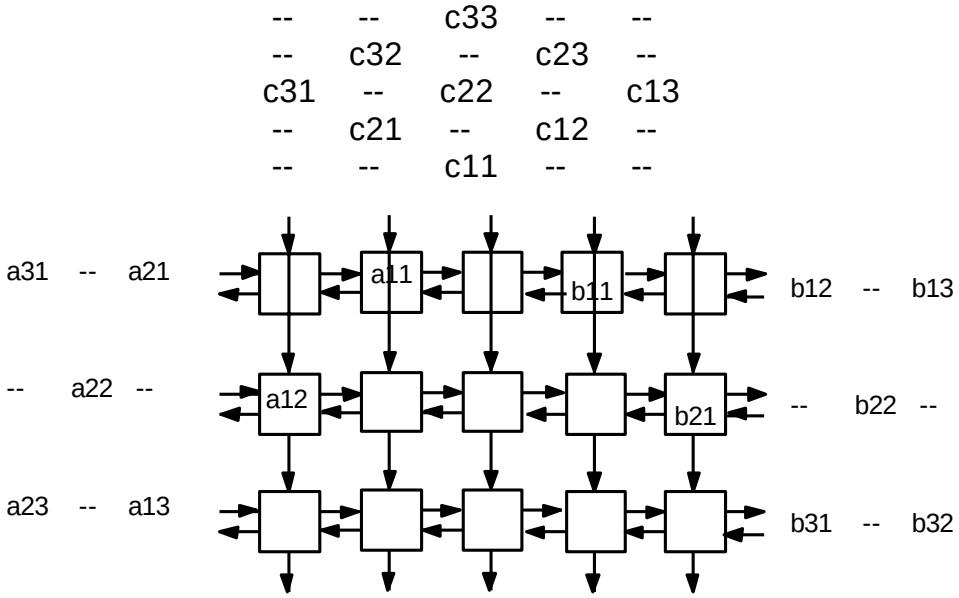


Figura 32 Array sistólico

Para utilizar este array sistólico, debe existir alguna función, hardware, etc., que introduzca los datos en el orden correcto en el momento correcto, y que además recoja los resultados mientras se van produciendo y los organice.

En nuestro caso se utilizará una celda muy parecida a la anterior. La diferencia es que el resultado de la multiplicación se acumulará dentro de cada celda y no se enviará a las celdas vecinas. Estas acumulaciones se enviarán al kernel encargado de recolectar todos los resultados. Además, para reutilizar las celdas en distintos cálculos, el valor de esta acumulación puede ser puesto a 0 de forma manual.

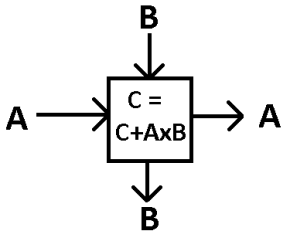


Figura 33 Celda multiplicación acumulación

El array sistólico formado con este tipo de celdas tiene la siguiente arquitectura:

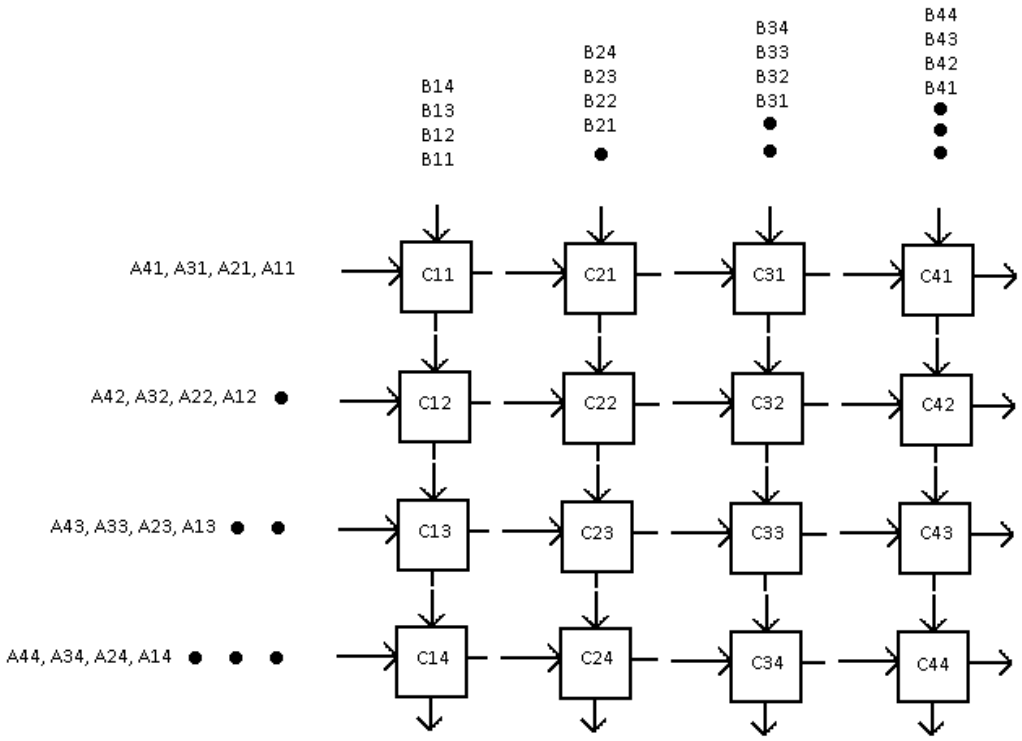


Figura 34 Array sistólico

3.3.2. Canales OpenCL

Los canales son un recurso disponible en OpenCL con el cual los kernels se pueden comunicar entre sí, sin depender del programa host [7]. Se tratan de buffers FIFO que comunican un kernel con otro, de forma que el host no debe coordinar ningún intercambio de datos.

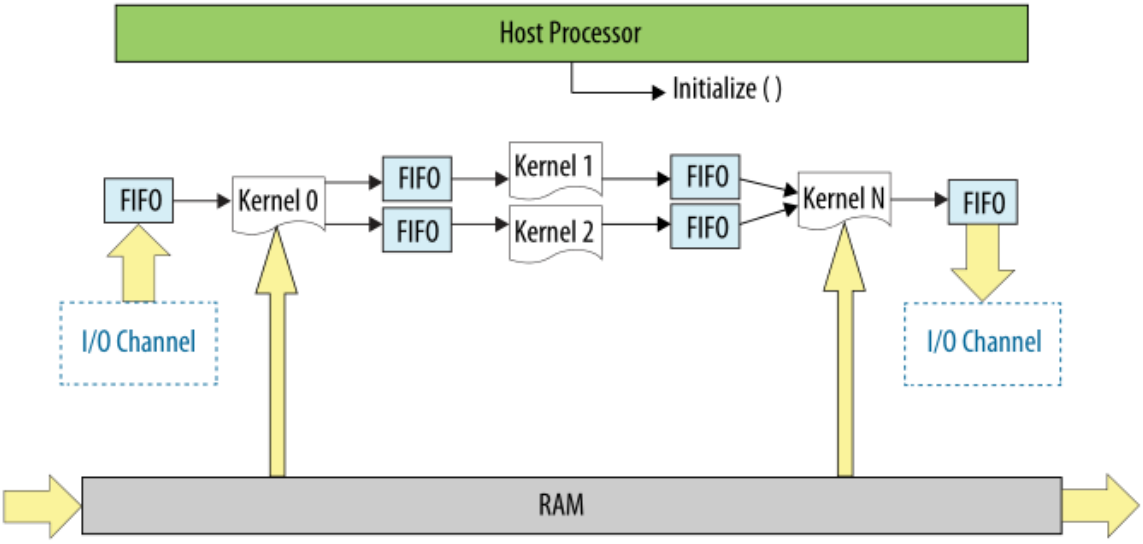


Figura 35 Canales OpenCL

Los datos escritos en un canal permanecen en él mientras el kernel permanezca cargado en la FPGA. Solo cuando un dato es leído de un canal, este es automáticamente eliminado, para cualquier invocación del kernel.

El canal puede comunicar distintos work-groups e incluso kernels NDRange con kernels Single Work Item. Los canales solo pueden ser escritos y leídos desde un único punto en el código. Por esta razón, no se puede vectorizar el uso de un canal, necesitaría algún mecanismo de coordinación que de momento OpenCL no posee.

Si no hay una dependencia de datos, las instrucciones de escritura y lectura de canales no son bloqueantes, es decir, el kernel no pausa su ejecución hasta que estas se ejecutan. Por tanto, es necesario controlar este aspecto en el caso de que se requiera un orden de ejecución.

3.3.3. Algoritmo multikernel

En esta sección se explicará el funcionamiento de los distintos kernels de este algoritmo. En total hay 4 kernels distintos:

- MatrixMult: el encargado de leer los buffers de datos y dividirlos en bloques.
- Feeder: su tarea es enviar los datos correspondientes, de forma ordenada a las celdas del array sistólico.
- Celda PE (Processing Element): Es el kernel correspondiente a una celda del array sistólico. Realiza una multiplicación acumulación y envía los índices a sus celdas vecinas.
- MatrixResult: su función consiste en recoger, en el momento adecuado, los resultados generados en cada celda del array sistólico.

Celda PE

Veamos en primer lugar el kernel de una celda del array sistólico. Se trata de un kernel muy sencillo que lee dos datos provenientes de canales, los multiplica y acumula con el valor guardado y reenvía los datos leídos por otros canales.

```
__kernel
__attribute__((max_global_work_dim(0)))
void PE22(int NMULT)
{
    for(int i = 0; i<NMULT; i++){
    float acc = 0.0f;
    float A =  0.0f;
    float B =  0.0f;

    #pragma unroll
    for (int i = 0 ; i < 4; i++){
            B = read_channel_altera(b0[6]);
            A = read_channel_altera(a0[7]);

            write_channel_altera(a0[8],A);
            write_channel_altera(b0[10],B);
            acc += A*B;
        }
    write_channel_altera(C_coef[10],acc);
    }
}
```

Figura 36 Kernel Celda Array sistólico

El primer bucle for, que depende de NMULT tiene la función de reiniciar el acumulador a 0 cuando se empieza la multiplicación de una submatriz nueva. Además, al finalizar las NMULT operaciones el kernel finaliza, lo que permite completar la ejecución.

El segundo bucle for, se ejecuta 4 veces. Esta 4 veces corresponden al valor elegido para el tamaño de bloque, 4x4. La elección de este tamaño no radica en el rendimiento, sino en la cantidad de código necesario. Para este tamaño es necesario escribir y conectar 16 kernels PE, pero para 8x8 ya serían 64. Cuando los códigos son tan largos, nos planteamos crear un generador de código, que puede ser una ampliación de este TFM.

En cada ejecución se leerá un dato de la submatriz de A y un dato de la submatriz de B. Una vez leídos se enviarán por canales distintos que comunican la celda con las vecinas. Además, se ejecutará la operación de multiplicación acumulación. Por último, al acabar las 4 operaciones se envía el resultado de la acumulación por otro canal, hacia el kernel feeder.

Como el dato calculado no se envía hasta que se completan todas las multiplicaciones acumulaciones, el envío de los índices A y B por los canales no necesitan esperar a que se complete esta multiplicación acumulación, por lo que el orden en el que están escritas las instrucciones, no necesariamente indica en qué orden se ejecutarán realmente. Lo más probable es que el envío de los índices A y B por canales y la multiplicación acumulación se ejecuten a la vez.

Los canales usados por todos los kernels se han declarado en forma de vector, donde cada componente es un canal independiente y se utiliza con un índice. El funcionamiento no varía de ninguna forma, pero utilizar el mismo nombre y seleccionar la componente aporta facilidad a la hora de escribir el código. Veamos la declaración de los distintos canales utilizados en este proyecto:

```
3   channel float a0[12] __attribute__((depth(0)));
4   channel float b0[12] __attribute__((depth(0)));
5
6   channel int Clear __attribute__((depth(0)));
7   channel int get_result __attribute__((depth(0)));
8   channel int position_x __attribute__((depth(0)));
9   channel int position_y __attribute__((depth(0)));
10
11  channel float A_coef[4] __attribute__((depth(0)));
12  channel float B_coef[4] __attribute__((depth(0)));
13  channel float C_coef[16] __attribute__((depth(0)));
14  channel float C_BS[16] __attribute__((depth(0)));
15
16  channel float A_BS[16] __attribute__((depth(0)));
17  channel float B_BS[16] __attribute__((depth(0)));
```

Figura 37 Declaración de canales

El atributo “depth(0)” indica que hasta que no se lea el dato escrito, no se podrá escribir otro. De esta forma, cuando un kernel intente escribir un dato sin que el anterior se haya leído, la ejecución se detendrá hasta que sea posible escribir. Como no podemos controlar el orden de ejecución de los distintos work-items, esto impide que un posible desajuste que provoque un fallo de nuestro proyecto.

MatrixMult

El kernel MatrixMult es el encargado de dividir las matrices en submatrices y enviar los datos al kernel feeder. Su implementación se basa en la del algoritmo de Intel FPGA visto anteriormente. De hecho, el método para recorrer la matriz y extraer los datos de las submatrices es el mismo. Este kernel está compilado como NDRange, y tiene que convivir con otros kernels Single Work Item, comunicándose con ellos a través de canales. Esto requerirá de ciertos cuidados para garantizar que no hay problemas.

```
22  __kernel
23  __attribute((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
24  void matrixMult( __global float *restrict A,
25                   __global float *restrict B,
26                   // Widths of matrices.
27                   int A_width, int B_width)
28  {
29      // Block index
30      int block_x = get_group_id(0);
31      int block_y = get_group_id(1);
32
33      // Local ID index (offset within a block)
34      int local_x = get_local_id(0);
35      int local_y = get_local_id(1);
36
37      // Compute loop bounds
38      int a_start = A_width * BLOCK_SIZE * block_y;
39      int a_end   = a_start + A_width - 1;
40      int b_start = BLOCK_SIZE * block_x;
41      int clear_flag, result;
42
43      clear_flag = 1;
44      result = 0;
45      // Compute the matrix multiplication result for this output element. Each
46      // loop iteration processes one block of the matrix.
47      for (int a = a_start, b = b_start; a <= a_end; a += BLOCK_SIZE, b += (BLOCK_SIZE * B_width))
48      {
49          if(BLOCK_SIZE * local_y + local_x == 0){
50              write_channel_altera(Clear,clear_flag);
51              if((a + BLOCK_SIZE) >= a_end){  //last iteration
52                  result = 1;
53                  write_channel_altera(position_x, get_group_id(0));
54                  write_channel_altera(position_y, get_group_id(1));
55              }else{
56                  result = 0;
57              }
58              write_channel_altera(get_result,result);
59          }
60
61          barrier(CLK_LOCAL_MEM_FENCE);
```

Figura 38 Kernel matrixMult

Se observan las variables utilizadas para recorrer la matriz y seleccionar la submatriz en el bucle for, a_start, a_end y b_start, que se obtienen a partir de los identificadores del work-item y work-group y a partir del ancho de las matrices.

Dentro del bucle for, tenemos una parte de código para el control de la multiplicación. Esta parte de código solo se ejecutará el work-item con identificador 0 en el work-group. Por ello, tras acabar estas operaciones, tenemos una instrucción de espera (barrier) de forma que se sincronicen todos los work-item en ese punto.

Las operaciones de este control son:

- Enviar la señal de clear (envío de un 1) a través de un canal para que el kernel feeder inicialice la variable en la que recogerá los resultados. Esta señal solo se activará en la primera iteración del bucle for. Como el kernel MatrixMult es NDRange, con cada ejecución completa de un work-group se habrá calculado el resultado de una submatriz y por tanto la variable del kernel feeder solo hay que inicializarla una vez en cada work-group.
- Enviar la señal result para que el kernel feeder envíe los datos al kernel MatrixResult. De nuevo, esta señal solo se envía una vez por ejecución de work-group, pero esta vez será en la última iteración del bucle for, cuando el resultado final de la multiplicación de submatrices va a ser calculado. En el caso de no estar en la última iteración se envía un 0, que no tendrá ningún efecto.
- Indicar al kernel matrix mult qué submatrices se están calculando. Como los identificadores de los work-group coinciden con los identificadores de las submatrices, enviando el resultado de la función “get_group_id(0/1)”, el kernel MatrixMult completará la submatriz correcta con los datos obtenidos del kernel feeder.

A continuación, dentro del bucle for, se enviará mediante canales los datos de las matrices A y B, hacia el kernel feeder. Como se ha comentado antes, no podemos escribir un mismo canal desde más de un punto, por lo que serán necesarios 32 canales para enviar los 16 datos del bloque de cada matriz (A y B).

Al usar canales en vectores, que se identifican con un índice, la opción ideal sería usar el índice del work-item para seleccionar el canal. Sin embargo, esta función no está implementada en OpenCL. Por ello necesitaremos recurrir a una estructura case:

```
        switch(BLOCK_SIZE * local_y + local_x){ //BLOCK multiplication finished
            case(0): write_channel_altera(A_BS[0], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[0], B[b + B_width * local_y + local_x]); break;
            case(1): write_channel_altera(A_BS[1], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[1], B[b + B_width * local_y + local_x]); break;
            case(2): write_channel_altera(A_BS[2], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[2], B[b + B_width * local_y + local_x]); break;
            case(3): write_channel_altera(A_BS[3], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[3], B[b + B_width * local_y + local_x]); break;
            case(4): write_channel_altera(A_BS[4], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[4], B[b + B_width * local_y + local_x]); break;
            case(5): write_channel_altera(A_BS[5], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[5], B[b + B_width * local_y + local_x]); break;
            case(6): write_channel_altera(A_BS[6], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[6], B[b + B_width * local_y + local_x]); break;
            case(7): write_channel_altera(A_BS[7], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[7], B[b + B_width * local_y + local_x]); break;
            case(8): write_channel_altera(A_BS[8], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[8], B[b + B_width * local_y + local_x]); break;
            case(9): write_channel_altera(A_BS[9], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[9], B[b + B_width * local_y + local_x]); break;
            case(10): write_channel_altera(A_BS[10], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[10], B[b + B_width * local_y + local_x]); break;
            case(11): write_channel_altera(A_BS[11], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[11], B[b + B_width * local_y + local_x]); break;
            case(12): write_channel_altera(A_BS[12], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[12], B[b + B_width * local_y + local_x]); break;
            case(13): write_channel_altera(A_BS[13], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[13], B[b + B_width * local_y + local_x]); break;
            case(14): write_channel_altera(A_BS[14], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[14], B[b + B_width * local_y + local_x]); break;
            case(15): write_channel_altera(A_BS[15], A[a + A_width * local_y + local_x]); write_channel_altera(B_BS[15], B[b + B_width * local_y + local_x]); break;
        }

        barrier(CLK_LOCAL_MEM_FENCE);
        clear_flag = 0;
    }
}
```

Figura 39 Kernel matrixMult

Para cada caso de nuestra estructura, se enviará el dato correspondiente (que si que se puede leer utilizando las funciones para obtener identificadores) extraído del buffer (memoria global) por el canal correspondiente. Estos canales llegarán hasta el kernel feeder.

Este kernel es el único compilado como NDRange, por lo que cada una de las ejecuciones concurrentes, con tamaño de índice Blocksize x Blocksize, enviará un dato distinto por un canal distinto, enviando las dos submatrices completas.

Feeder

El kernel feeder es el encargado de enviar, también mediante canales, los datos de las matrices A y B hacia el array sistólico y de recoger los resultados que compondrán la matriz C. Este envío y recogida de datos debe hacerlo de forma ordenada, ya que el array sistólico no contiene ninguna lógica que organice los datos recibidos.

Para organizar este envío y recepción de datos en el momento correcto, utiliza varios canales a modo de señales de activación. Estos canales no transportarán datos de las matrices, sino que únicamente comunicarán un 1 o un 0 para realizar, o no, ciertas acciones (envío de datos, reinicio, etc.).

```
__kernel
__attribute__((max_global_work_dim(0)))
void feeder(int BS_dim, int NMULT)
{
  float C_sum[16], BS_A[16], BS_B[16];
  int clear_flag_local, result;


  for(int i = 0; i<NMULT; i++){
    clear_flag_local = read_channel_altera(Clear);
    if(clear_flag_local == 1){
      C_sum[0] = 0;
      C_sum[1] = 0;
      C_sum[2] = 0;
      C_sum[3] = 0;
      C_sum[4] = 0;
      C_sum[5] = 0;
      C_sum[6] = 0;
      C_sum[7] = 0;
      C_sum[8] = 0;
      C_sum[9] = 0;
      C_sum[10] = 0;
      C_sum[11] = 0;
      C_sum[12] = 0;
      C_sum[13] = 0;
      C_sum[14] = 0;
      C_sum[15] = 0;
    }
```

Figura 40 Kernel feeder

Este kernel recibirá los datos de las submatrices de A y B por canales, desde el kernel MatrixMult y los guardará en dos variables internas al kernel (memoria privada). También utilizará una variable para acumular los productos de las submatrices. Esta variable deberá ser reiniciada cada vez que se inicie el cálculo de una nueva submatriz de C.

Dentro del kernel existe un bucle for principal, que se ejecuta el mismo número de veces que el bucle for principal del kernel PE. Cada ejecución es para una multiplicación de submatrices. Este número es calculado en el programa host y pasado como parámetro. De esta forma, la ejecución del kernel finaliza cuando finaliza la multiplicación de matrices completa.

La primera acción es comprobar si el valor del acumulador debe ser reiniciado a 0, lo que significa que es la primera multiplicación para calcular una submatriz. Como hemos visto previamente, es el kernel MatrixMult el que envía un 1 por el canal “Clear” para comenzar este reinicio.

```
BS_A[0] = read_channel_altera(A_BS[0]);
BS_A[1] = read_channel_altera(A_BS[1]);
BS_A[2] = read_channel_altera(A_BS[2]);
BS_A[3] = read_channel_altera(A_BS[3]);
BS_A[4] = read_channel_altera(A_BS[4]);
BS_A[5] = read_channel_altera(A_BS[5]);
BS_A[6] = read_channel_altera(A_BS[6]);
BS_A[7] = read_channel_altera(A_BS[7]);
BS_A[8] = read_channel_altera(A_BS[8]);
BS_A[9] = read_channel_altera(A_BS[9]);
BS_A[10] = read_channel_altera(A_BS[10]);
BS_A[11] = read_channel_altera(A_BS[11]);
BS_A[12] = read_channel_altera(A_BS[12]);
BS_A[13] = read_channel_altera(A_BS[13]);
BS_A[14] = read_channel_altera(A_BS[14]);
BS_A[15] = read_channel_altera(A_BS[15]);

BS_B[0] = read_channel_altera(B_BS[0]);
BS_B[1] = read_channel_altera(B_BS[1]);
BS_B[2] = read_channel_altera(B_BS[2]);
BS_B[3] = read_channel_altera(B_BS[3]);
BS_B[4] = read_channel_altera(B_BS[4]);
BS_B[5] = read_channel_altera(B_BS[5]);
BS_B[6] = read_channel_altera(B_BS[6]);
BS_B[7] = read_channel_altera(B_BS[7]);
BS_B[8] = read_channel_altera(B_BS[8]);
BS_B[9] = read_channel_altera(B_BS[9]);
BS_B[10] = read_channel_altera(B_BS[10]);
BS_B[11] = read_channel_altera(B_BS[11]);
BS_B[12] = read_channel_altera(B_BS[12]);
BS_B[13] = read_channel_altera(B_BS[13]);
BS_B[14] = read_channel_altera(B_BS[14]);
BS_B[15] = read_channel_altera(B_BS[15]);


for (int i = 0 ; i < BS_dim; i++){
    write_channel_altera(A_coef[0],BS_A[i]);
    write_channel_altera(A_coef[1],BS_A[1*BS_dim+i]);
    write_channel_altera(A_coef[2],BS_A[2*BS_dim+i]);
    write_channel_altera(A_coef[3],BS_A[3*BS_dim+i]);

    write_channel_altera(B_coef[0],BS_B[i*BS_dim]);
    write_channel_altera(B_coef[1],BS_B[i*BS_dim+1]);
    write_channel_altera(B_coef[2],BS_B[i*BS_dim+2]);
    write_channel_altera(B_coef[3],BS_B[i*BS_dim+3]);
}
```

Figura 41 Kernel feeder

A continuación, se reciben todos los datos de las submatrices, también enviados desde el kernel MatrixMult. Aquí podemos observar la comunicación entre un kernel NDRange y un kernel Single Work Item. Debido a que el kernel MatrixMult se ejecuta de forma concurrente un total de 16 veces (tamaño Blocksize x Blocksize), cada ejecución envía un valor distinto de las submatrices, cada uno utilizando un canal distinto (una componente del vector de canales).

Tras recibir los datos de las submatrices, existe un segundo bucle, en el cual se enviarán estos datos hacia el array sistólico, como siempre, a través de canales.

```
188         C_sum[0] += read_channel_altera(C_coef[0]);
189         C_sum[1] += read_channel_altera(C_coef[1]);
190         C_sum[2] += read_channel_altera(C_coef[2]);
191         C_sum[3] += read_channel_altera(C_coef[3]);
192         C_sum[4] += read_channel_altera(C_coef[4]);
193         C_sum[5] += read_channel_altera(C_coef[5]);
194         C_sum[6] += read_channel_altera(C_coef[6]);
195         C_sum[7] += read_channel_altera(C_coef[7]);
196         C_sum[8] += read_channel_altera(C_coef[8]);
197         C_sum[9] += read_channel_altera(C_coef[9]);
198         C_sum[10] += read_channel_altera(C_coef[10]);
199         C_sum[11] += read_channel_altera(C_coef[11]);
200         C_sum[12] += read_channel_altera(C_coef[12]);
201         C_sum[13] += read_channel_altera(C_coef[13]);
202         C_sum[14] += read_channel_altera(C_coef[14]);
203         C_sum[15] += read_channel_altera(C_coef[15]);
204
205         result = read_channel_altera(get_result);
206         if(result==1){
207             write_channel_altera(C_BS[0],C_sum[0]);
208             write_channel_altera(C_BS[1],C_sum[1]);
209             write_channel_altera(C_BS[2],C_sum[2]);
210             write_channel_altera(C_BS[3],C_sum[3]);
211             write_channel_altera(C_BS[4],C_sum[4]);
212             write_channel_altera(C_BS[5],C_sum[5]);
213             write_channel_altera(C_BS[6],C_sum[6]);
214             write_channel_altera(C_BS[7],C_sum[7]);
215             write_channel_altera(C_BS[8],C_sum[8]);
216             write_channel_altera(C_BS[9],C_sum[9]);
217             write_channel_altera(C_BS[10],C_sum[10]);
218             write_channel_altera(C_BS[11],C_sum[11]);
219             write_channel_altera(C_BS[12],C_sum[12]);
220             write_channel_altera(C_BS[13],C_sum[13]);
221             write_channel_altera(C_BS[14],C_sum[14]);
222             write_channel_altera(C_BS[15],C_sum[15]);
223         }
224     }
225 }
```

Figura 42 Kernel feeder

En el momento en que el array sistólico envíe los datos del producto, estos serán leídos y acumulados por el kernel feeder. Por último, si la iteración es la última para obtener el resultado final de una submatriz, se enviarán los datos hacia el kernel MatrixResult. Para ejecutar esta acción en el momento correcto, se leerá el canal “get_result”, donde el kernel MatrixMult ha escrito un 0 o un 1.

MatrixResult

Este último kernel cumple la función de rellenar la matriz resultado C con los resultados de las submatrices que le van llegando a través de canales. Para mapear estos datos en C, utiliza la información de los identificadores de work-group, que previamente ha enviado el kernel MatrixMult. Es el kernel que rellena el buffer correspondiente a la matriz C, lo que implica accesos a memoria global.

```
__kernel
__attribute__((max_global_work_dim(0)))
void MatrixResult(__global float *restrict C, int C_width, int N){
    int pos, pos_x, pos_y, i;
    for(i = 0; i < N; i++){
        pos_x = read_channel_altera(position_x);
        pos_y = read_channel_altera(position_y);

        pos = (pos_y * BLOCK_SIZE * C_width) + (pos_x * BLOCK_SIZE);

        C[pos] = read_channel_altera(C_BS[0]);
        C[pos + 1] = read_channel_altera(C_BS[1]);
        C[pos + 2] = read_channel_altera(C_BS[2]);
        C[pos + 3] = read_channel_altera(C_BS[3]);
        C[pos + C_width] = read_channel_altera(C_BS[4]);
        C[pos + C_width + 1] = read_channel_altera(C_BS[5]);
        C[pos + C_width + 2] = read_channel_altera(C_BS[6]);
        C[pos + C_width + 3] = read_channel_altera(C_BS[7]);
        C[pos + 2*C_width] = read_channel_altera(C_BS[8]);
        C[pos + 2*C_width + 1] = read_channel_altera(C_BS[9]);
        C[pos + 2*C_width + 2] = read_channel_altera(C_BS[10]);
        C[pos + 2*C_width + 3] = read_channel_altera(C_BS[11]);
        C[pos + 3*C_width] = read_channel_altera(C_BS[12]);
        C[pos + 3*C_width + 1] = read_channel_altera(C_BS[13]);
        C[pos + 3*C_width + 2] = read_channel_altera(C_BS[14]);
        C[pos + 3*C_width + 3] = read_channel_altera(C_BS[15]);
    }

}
```

Figura 43 Kernel MatrixResult

Consta de un bucle que se repite tantas veces como bloques en los que se divide la matriz. En cada iteración se recibe el identificador en las dos coordenadas del work-group de MatrixMult, que nos indicará qué bloque se ha calculado. Con estos datos se obtienen los índices de cada elemento de la submatriz dentro del buffer C, que, como A y B, es de una sola dimensión y se maneja utilizando el ancho (argumento del kernel C_width). Cuando todas las submatrices se hayan completado, el kernel finalizará su ejecución.

En la siguiente figura podemos observar un esquema que describe la conexión entre los distintos kernels, utilizando canales.

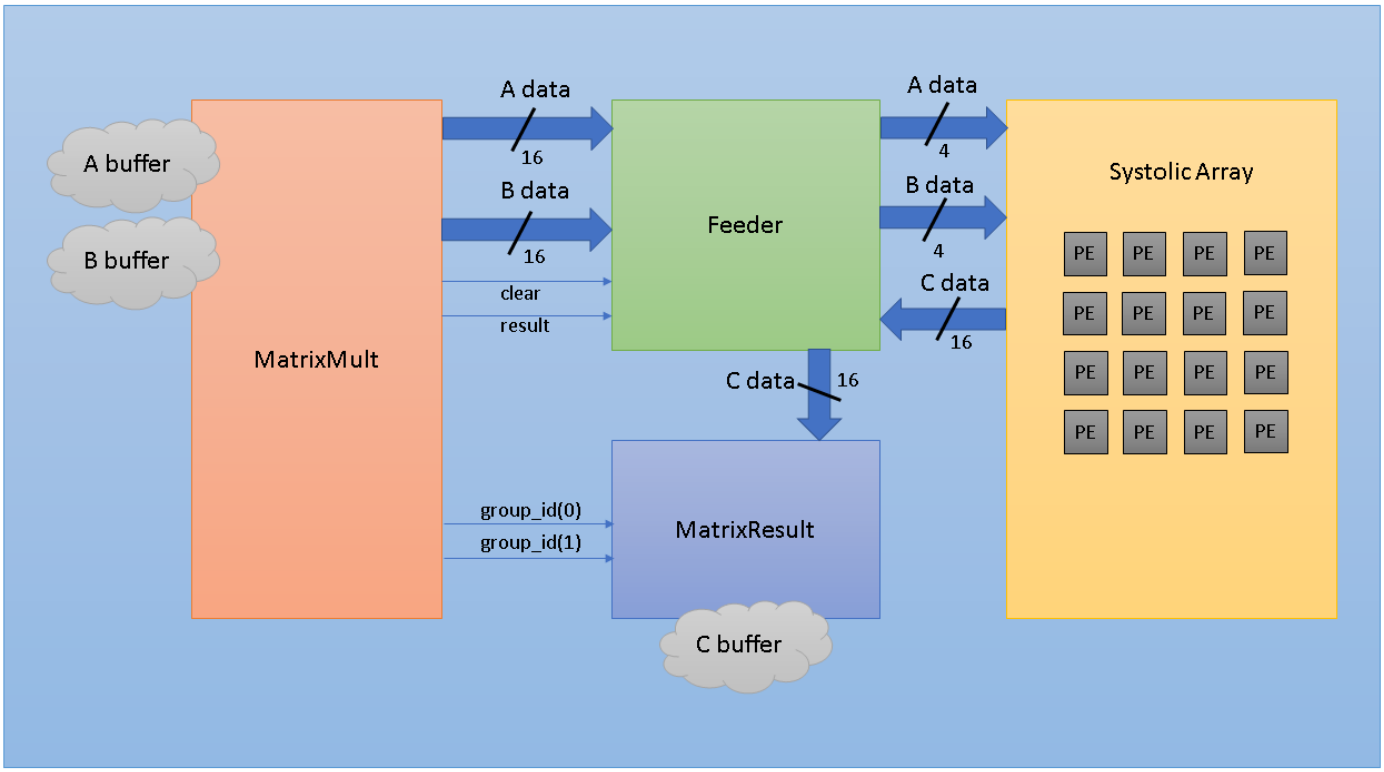


Figura 44 Esquema multikernel

3.3.4. Análisis y rendimiento

En este apartado ejecutaremos nuestro algoritmo multi-kernel y realizaremos unas medidas de rendimiento. Compararemos los resultados obtenidos con las mejores compilaciones obtenidas en la primera sección. Además, analizaremos los puntos fuertes y débiles de esta implementación propia.

Veamos qué tiempos obtiene nuestra implementación para la multiplicación de matrices de los tamaños vistos anteriormente.

Compilación	BLOCKSIZE	SIMD	Comp Units	Time (ms)	Kernel time (ms)
Tamaño 512 x 256					
Mejor 512x256	64	8	1	1,364	1,286
Implementación propia	4	--	--	313,189	311,791
Tamaño 256 x 128					
Mejor 256 x 128	32	8	1	0,243	0,176
Implementación propia	4	--	--	25,991	25,125
Tamaño 64 x 64					
Mejor 64 x 64	16	8	1	0,101	0,049
Implementación propia	4	--	--	2,498	1,736

Los tiempos obtenidos con nuestra implementación son mucho mayores, siendo la diferencia más grande cuanto mayor es el tamaño de las matrices. Las razones de esta diferencia son principalmente dos:

- **Tamaño del bloque:** hemos observado que el tamaño del array sistólico depende del tamaño de las submatrices. Los distintos kernels PE que forman el array sistólico están escritos y conectados manualmente, pues no son exactamente iguales al cambiar el nombre de los canales. Para un tamaño manejable se ha elegido un bloque 4x4. Con un generador de código se podría solventar este problema, por lo que esto es una idea de ampliación del presente trabajo.

- **Concurrencia en la compute unit:** Dentro de una compute unit se ejecutan varios workgroup de forma concurrente [7]. Estos workgroup se encuentran en distintos estados de ejecución y realizan pipeline si el hardware lo permite. Sin embargo, en la nueva implementación, la existencia de un único array sistólico que queda bloqueado hasta que se requiere el resultado elimina las posibilidades de concurrencia. Para solventar este problema, otra posibilidad de ampliación es el uso de otro tipo de array sistólico que "saque" los datos a medida que estén disponibles, de manera que ya se puedan utilizar. De esta forma se podrían enviar directamente a otro array sistólico similar que computara otra multiplicación acumulación. A continuación tenemos un ejemplo de array sistólico que se comporta de esta manera:

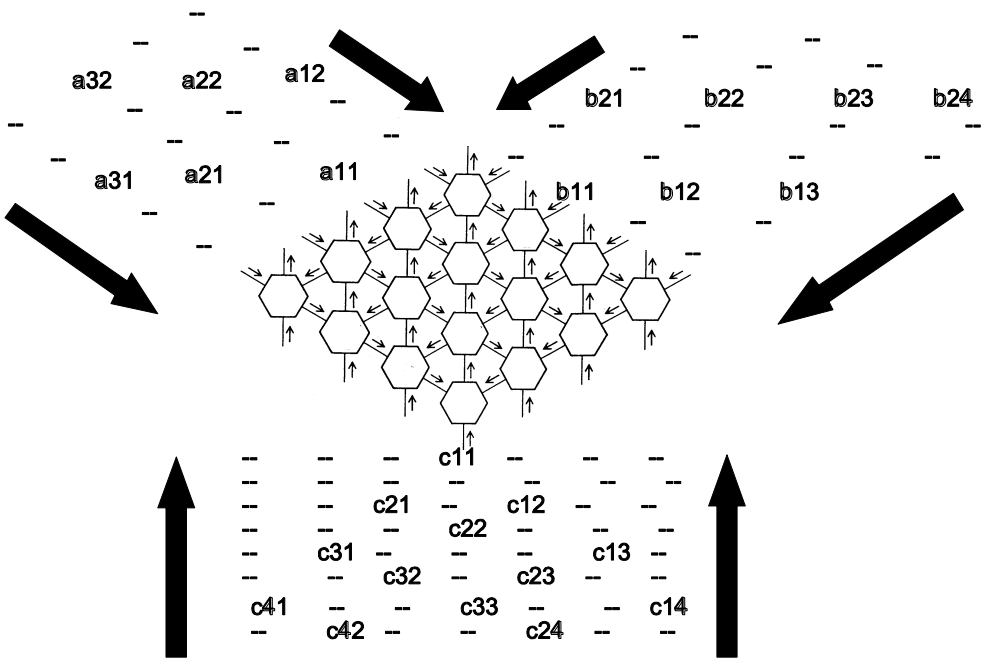


Figura 45 Array sistólico celda hexagonal

Para comprobar esta concurrencia entre workgroups se compararán los resultados de nuestra implementación con la implementación de IntelFPGA compilada para un blocksize de 4.

Compilación	Time (ms)	Kernel time (ms)
Tamaño 512 x 256		
Mejor 512x256	1,364	1,286
Implementación propia	313,189	311,791
Intel FPGA Blocksize 4	56,916	56,797
Tamaño 256 x 128		
Mejor 256 x 128	0,243	0,176
Implementación propia	25,991	25,125
Intel FPGA Blocksize 4	5,806	5,726
Tamaño 64 x 64		
Mejor 64 x 64	0,101	0,049
Implementación propia	2,498	1,736
Intel FPGA Blocksize 4	0,514	0,417

Como podemos observar, los tiempos conseguidos con la implementación original y un blocksize de 4 son mucho mayores que la implementación optimizada, pero siguen siendo menores que los conseguidos con la implementación del array sistólico. Como hemos explicado antes, el hecho de que los work-groups puedan realizar pipeline, aumenta la capacidad de cálculo, algo que no está disponible en nuestra implementación.

Para realizar una comparación más igualada, se ha modificado ligeramente el kernel de Intel FPGA para que únicamente se pueda realizar una multiplicación de submatrices de forma simultánea. De esta forma se recrean las condiciones en las que trabaja nuestra implementación. Unas condiciones nada óptimas y cuya mejora puede es una de las propuestas para continuar este trabajo, dada la insistencia de Intel FPGA por el uso de arrays sistólicos.

```
__kernel
__attribute((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
void matrixMult(  __global float *restrict C,
                  __global float *restrict A,
                  __global float *restrict B,
                  int A_width, int B_width, int A_height)
{
    __local float A_local[BLOCK_SIZE][BLOCK_SIZE];
    __local float B_local[BLOCK_SIZE][BLOCK_SIZE];

    for (int block_x = 0; block_x < B_width/BLOCK_SIZE; block_x++){
        for (int block_y = 0; block_y < A_height/BLOCK_SIZE; block_y++){

            int local_x = get_local_id(0);
            int local_y = get_local_id(1);

            int a_start = A_width * BLOCK_SIZE * block_y;
            int a_end   = a_start + A_width - 1;
            int b_start = BLOCK_SIZE * block_x;

            float running_sum = 0.0f;

            for (int a = a_start, b = b_start; a <= a_end; a += BLOCK_SIZE, b += (BLOCK_SIZE * B_width))
            {
                A_local[local_y][local_x] = A[a + A_width * local_y + local_x];
                B_local[local_x][local_y] = B[b + B_width * local_y + local_x];

                barrier(CLK_LOCAL_MEM_FENCE);

                #pragma unroll
                for (int k = 0; k < BLOCK_SIZE; ++k)
                {
                    running_sum += A_local[local_y][k] * B_local[local_x][k];
                }

                barrier(CLK_LOCAL_MEM_FENCE);
            }

            C[(BLOCK_SIZE * B_width * block_y) + (BLOCK_SIZE * block_x) + (B_width * local_y) + local_x] = running_sum;
        }
    }
}
```

Figura 46 Código kernel adaptado

Este kernel será ejecutado en un solo work-group. Dentro de este, todos los work-item calcularán el resultado de cada multiplicación de submatrices, una a una. La matriz se recorre entera gracias a los dos nuevos bucles for introducidos, que realizan la función que anteriormente tenían los work-group. El resto del algoritmo es igual que el explicado en la primera sección del desarrollo. Además, el tamaño de bloque es de 4.

Con estas modificaciones conseguimos que solamente una multiplicación de matrices se ejecute de forma concurrente. Tenemos un kernel prácticamente idéntico al de Intel FPGA pero añadiendo algunas limitaciones que se encuentran en nuestra implementación. Así conseguimos una comparación más justa y podremos comprobar si el uso de un array sistólico, canales y multikernel mejoran el rendimiento. Esta vez probaremos muchos más tamaños de matrices, para ver en qué punto nuestra implementación es superior.

Tamaño matriz C	Algoritmo array sistólico		Intel FPGA un workgroup	
	Host time (ms)	Kernel time (ms)	Host time (ms)	Kernel time (ms)
4 x 4	0,947	0,033	0,119	0,061
8 x 8	1,373	0,139	0,122	0,063
32 x 32	1,591	0,317	0,282	0,219
64 x 64	2,513	1,762	1,557	1,492
128 x 128	13,557	12,682	11,401	11,328
256 x 256	157,069	156,028	227,333	227,276
512 x 512	1246,631	1244,691	1839,984	1839,899
1024 x 1024	10273,274	10266,291	15168,908	15168,813

Esta vez sí que podemos observar un rendimiento superior de nuestra implementación, a partir del tamaño 256 x 256. En este punto, todo el tiempo que nuestro algoritmo está moviendo datos y realizando operaciones de control se compensa por la eficiencia para realizar la multiplicación del array sistólico.

Compilación	BLOCKSIZE	SIMD	Comp Units	Logic utilization	Block memory bits	Power Cosumption (mW)	Max freq (MHz)
Mejor 512x256	64	8	1	83944 (33%)	5225738 (12%)	10960,23	203,95
Mejor 256 x 128	32	8	1	57322 (23%)	3672330 (8%)	8717,53	237,69
Mejor 64 x 64	16	8	1	50928 (20%)	3398282 (8%)	7938,72	253,16
Intel FPGA Blocksize 4	4	4	1	44114 (18%)	2324618 (5%)	6969,54	265,39
Algoritmo array sistólico	4	--	--	76107 (30%)	3131786 (7%)	8109,77	219,78
Intel FPGA un workgroup	4	--	1	40665 (16%)	2713098 (6%)	6939,72	294.2

En cuanto a recursos utilizados, vemos que nuestro algoritmo utiliza una cantidad acorde al resto de implementaciones, sin ser la más alta. Sin embargo, las dos implementaciones de Intel FPGA con blocksize 4 usan prácticamente el mismo número de recursos, pero tienen un rendimiento muy distinto. Aquí podemos comprobar la eficiencia del pipeline entre workgroups y el incremento de la capacidad de cálculo al aprovechar ciertas ventajas de OpenCL y, como consecuencia, aprovechar mejor el hardware. También confirmamos la necesidad de otorgar concurrencia a nuestra implementación con array sistólico si queremos aprovechar de forma más eficiente todos los recursos utilizados, además de reducir el tiempo de cálculo.

4. Conclusiones y posibles ampliaciones

Como conclusiones de este trabajo podemos extraer que OpenCL nos permite implementar un diseño en una FPGA sin tener que recurrir a lenguajes de más bajo nivel como Verilog o VHDL. Permite utilizar las características y los recursos disponibles en las FPGAs (y también en GPUs) para ejecutar de forma más eficiente distintos algoritmos. Para ello, se apoya en un modelo de computación paralela, en el cuál utiliza todos los elementos procesadores disponibles en nuestro sistema.

Este modelo de programación (presente de forma muy parecida en CUDA) es la nueva tendencia en un mundo donde la mayoría de dispositivos son dispositivos heterogéneos, que pueden sacar partido de sus múltiples unidades de procesamiento. La gran portabilidad de OpenCL lo hace el candidato ideal.

Durante el trabajo se ha creado un entorno de desarrollo donde, apoyándose en varias herramientas externas, se facilita todo el proceso de creación, ejecución y análisis de aplicaciones en OpenCL. Dos de estas herramientas son PyOpenCL, donde mediante el uso del intuitivo lenguaje de programación Python se puede acceder a la API de OpenCL; y Jupyter, que nos permite acceder y ejecutar código en el servidor donde está conectado el dispositivo OpenCL.

En cuanto a los algoritmos utilizados, se ha estudiado cómo para cada problema concreto, unos cambios en distintos parámetros que nos ofrece OpenCL pueden significar un gran aumento en el rendimiento.

Por último, en la implementación propia de la multiplicación de matrices, se ha podido hacer uso de una gran cantidad de herramientas que ofrece OpenCL. Algunas de estas son: la posibilidad de ejecutar varios kernels a la vez, la comunicación entre ellos mediante canales o la interacción entre kernels NDRange y Single Work Item. Mediante el análisis del rendimiento de varias implementaciones, se ha puesto de manifiesto que el uso del pipeline y la concurrencia de tareas aumenta el rendimiento de forma muy significativa.

Las posibilidades de ampliación se centran en resolver los problemas encontrados en la implementación que utiliza el array sistólico. En primer lugar, como ya se ha mencionado anteriormente, el código para crear este array sistólico se ha tenido que crear a mano, ya que, aunque todos los kernels son en esencia iguales, tienen pequeñas diferencias relativas a sus interconexiones. Esto impide replicarlos sin un tratamiento posterior. Por otro lado, estas interconexiones guardan una relación que podría ser programada en un generador de código. De esta manera, la ampliación consistiría en realizar un script que dado un kernel OpenCL correspondiente a una celda del array sistólico y un tamaño de bloque, generara el código para su implementación. Esto permitiría realizar pruebas con arrays sistólicos de distintos tamaños y encontrar la mejor opción para cada problema.

La segunda ampliación propuesta sería dotar de un cierto pipeline o concurrencia a la multiplicación de submatrices en el array sistólico. Como se ha comprobado, esta es una de las limitaciones que impiden a nuestra implementación superar en rendimiento a la propuesta por Intel FPGA. Una vez nuestro array sistólico ha calculado un resultado no puede volver a operar hasta que los resultados se han extraído. Esto elimina la posibilidad de pipeline en las operaciones. El uso de otro tipo de array sistólico como el propuesto en la figura 45 podría resolver este problema. Incluso utilizar varios de estos arrays en cascada o en paralelo reduciría el tiempo de cálculo. Sería necesario además solucionar la limitación que imponen los canales: deben escribirse desde un único punto en el código.

5. Bibliografía

[1] Mike DeHaan Practical Uses of Matrix Mathematics. www.decodedscience.org/practical-uses-matrix-mathematics/40494

[2] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Computing in Science and Engineering 2010 May; 12(3): 66–72.

[3] O. Beaumont, V. Boudet, F. Rastello and Y. Robert, "Matrix multiplication on heterogeneous platforms," in IEEE Transactions on Parallel and Distributed Systems, vol. 12, no. 10, pp. 1033-1051, Oct. 2001.

[4] A. Munshi, "The OpenCL specification," 2009 IEEE Hot Chips 21 Symposium (HCS), Stanford, CA, 2009, pp. 1-314.

[5] T. S. Czajkowski et al., "From opencl to high-performance hardware on FPGAS," 22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, 2012, pp. 531-534.

[6] Intel FPGA. Intel FPGA SDK for OpenCL Pro Edition Getting Started Guide. www.intel.com/content/www/us/en/programmable/documentation/mwh1391807309901.html

[7] Intel FPGA. Intel FPGA SDK for OpenCL Pro Edition Programming Guide. www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html

[8] Intel FPGA. Design Examples. www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/opencl/developer-zone.html

[9] Intel FPGA. Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407.html

[10] Anaconda Installation. https://conda.io/docs/user-guide/index.html

[11] Andreas Kloeckner. PyOpenCL. https://documen.tician.de/pyopencl/

[12] Andreas Kloeckner. PyOpenCL Download. https://pypi.python.org/pypi/pyopencl

[13] Python Software Foundation. Measure execution time of small code snippets https://docs.python.org/2/library/timeit.html

[14] Jupyter Team. Installing Jupyter Notebook http://jupyter.readthedocs.io/en/latest/install.html

[15] ArchWiki. Iptables. https://wiki.archlinux.org/index.php/Iptables

[16] Kung, H. T. (1982). Why systolic architectures?. IEEE computer, 15(1), 37-46.

6. Anexo. Programa host para el algoritmo multi kernel

En este anexo, se explicará el código que compone la parte host para la ejecución de nuestro algoritmo multi kernel. Como ya hay un ejemplo de código host en Python, esta vez el código estará escrito en C++.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "CL/opencl.h"
#include "AOCLUtils/aocl_utils.h"
#include "matrixMult.h"

using namespace aocl_utils;

// OpenCL runtime configuration
const unsigned num_kernels = 16; //Se usaran 16 kernels para BS
static cl_platform_id platform = NULL;
static cl_device_id device = NULL;
static cl_context context = NULL;
static cl_command_queue queues[num_kernels];
static cl_kernel kernels[num_kernels];
static cl_command_queue queue_top;
static cl_kernel kernel_top;
static cl_command_queue queue_feeder;
static cl_kernel kernel_feeder;
static cl_command_queue queue_MatrixResult;
static cl_kernel kernel_MatrixResult;
static cl_program program = NULL;
static cl_uint num_devices;

// Device memory buffers
cl_mem input_buf_A_top, input_buf_B_top, output_buf_C_top;

static const char* kernel_names[num_kernels] =
{
  "PE00",
  "PE01",
  "PE02",
  "PE03",
  "PE10",
  "PE11",
  "PE12",
  "PE13",
  "PE20",
  "PE21",
  "PE22",
  "PE23",
  "PE30",
  "PE31",
  "PE32",
  "PE33"
};

static const char* kernel_top_name ="matrixMult";
static const char* kernel_feeder_name ="feeder";
static const char* kernel_MatrixResult_name ="MatrixResult";
```

En primer lugar, tenemos la inclusión de librerías y la declaración de múltiples variables, utilizando tipos de datos que proporciona la librería de OpenCL. Observamos la declaración, en un array, de los nombres de los kernels que componen el array sistólico. Esto facilitará mucho el manejo de estos kernels más adelante.

A continuación, tenemos todas las declaraciones, correspondientes a los tamaños de las matrices, bloques, número de multiplicaciones necesarias, etc. También observamos los prototipos de las funciones que se van a utilizar.

```
// Problem data.
const unsigned A_height = 1024;
const unsigned A_width  = 1024;
const unsigned B_height = A_width;  //Mandatory
const unsigned B_width  = 1024;
const unsigned C_height = A_height; //Mandatory
const unsigned C_width  = B_width;  //Mandatory
const unsigned Block_Size = BLOCK_SIZE; //Mandatory
const unsigned NMULT = (A_width * A_height * B_width)/(Block_Size*Block_Size*Block_Size); //REQUIRED MULTIPLICATIONS
const unsigned N_BS_C = (C_height/BLOCK_SIZE) * (C_width/BLOCK_SIZE); //Numero de bloques en matriz C


const int NC = C_height * C_width;
const int N2 = 16;  //SYSTOLIC ARRAY SIZE A B C
const int NA = A_height * A_width;
const int NB = B_height * B_width;

float input_A[NA];
float input_B[NB];
float output[NC];
float ref_output[NC];


// Function prototypes
float rand_float();
bool init_opencl();
void init_problem();
void run();
void compute_reference();
void verify();
void cleanup();
```

Nuestra función main llamará a distintas funciones que completarán todo el proceso. Una de estas funciones, rand_float(), devolverá un valor aleatorio entre los límites establecidos.

```
// Entry point.
int main() {
  printf("Matrix sizes:\n  A: %d x %d\n  B: %d x %d\n  C: %d x %d\n",A_height, A_width, B_height, B_width, C_height, C_width);

  // Initialize OpenCL.
  if(!init_opencl()) {
    return -1;
  }

  // Initialize the problem data.
  init_problem();

  // Run the kernel.
  run();

  // Free the resources allocated
  cleanup();
  return 0;
}

/////// HELPER FUNCTIONS ///////

// Randomly generate a floating-point number between -1 and 1.
float rand_float() {
  //return 1.0f;
  return float(rand()) / float(RAND_MAX) * 2.0f - 1.0f;
}
```

La función init_opencl() tiene la función de escanear los dispositivos disponibles, seleccionar el que se va a utilizar, crear el contexto y cargar el kernel en la FPGA. Para ello utiliza las funciones de la API de OpenCL: findPlatform, clCreateContext, clBuildProgram… entre otras.

```
bool init_opencl() {
  cl_int status;

  printf("Initializing OpenCL\n");

  if(!setCwdToExeDir()) {
    return false;
  }

  // Get the OpenCL platform.
  platform = findPlatform("Altera");
  if(platform == NULL) {
    printf("ERROR: Unable to find Altera OpenCL platform.\n");
    return false;
  }

  // Query the available OpenCL device.
  scoped_array<cl_device_id> devices;
  devices.reset(getDevices(platform, CL_DEVICE_TYPE_ALL, &num_devices));
  printf("Platform: %s\n", getPlatformName(platform).c_str());
  device = devices[0];
  printf("Using only first device\n");
  printf("  %s\n", getDeviceName(device).c_str());

  // Create the context.
  context = clCreateContext(NULL, 1, &device, NULL, NULL, &status);
  checkError(status, "Failed to create context");

  // Create the program for all device. Use the first device as the
  // representative device (assuming all device are of the same type).
  std::string binary_file = getBoardBinaryFile("matrix_mult", device);
  printf("Using AOCX: %s\n", binary_file.c_str());
  program = createProgramFromBinary(context, binary_file.c_str(), &device, 1);

  // Build the program that was just created.
  status = clBuildProgram(program, 0, NULL, "", NULL, NULL);
  checkError(status, "Failed to build program");
```

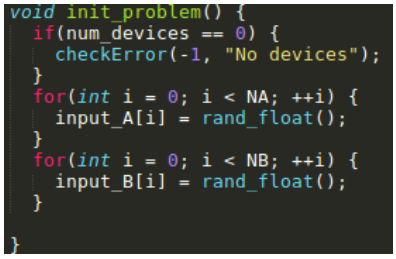
Dentro de esta misma función, se crean los objetos para las colas, los kernels y los buffers.

```
  for(int i = 0; i < num_kernels; ++i){
    kernels[i] = NULL;
    queues[i] = NULL;
  }

  kernel_top = NULL;
  queue_top = NULL;

  kernel_feeder = NULL;
  queue_feeder = NULL;

  kernel_MatrixResult = NULL;
  queue_MatrixResult = NULL;

  for(unsigned i = 0; i < num_kernels; ++i) {

    queues[i] = clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE, &status);
    checkError(status, "Failed to create command queue");

    kernels[i] = clCreateKernel(program, kernel_names[i], &status);
    checkError(status, "Failed to create kernel");
  }

    queue_top = clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE, &status);
    checkError(status, "Failed to create command queue");

    kernel_top = clCreateKernel(program, kernel_top_name, &status);
    checkError(status, "Failed to create kernel");

    queue_feeder = clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE, &status);
    checkError(status, "Failed to create command queue");

    kernel_feeder = clCreateKernel(program, kernel_feeder_name, &status);
    checkError(status, "Failed to create kernel");

    queue_MatrixResult = clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE, &status);
    checkError(status, "Failed to create command queue");

    kernel_MatrixResult = clCreateKernel(program, kernel_MatrixResult_name, &status);
    checkError(status, "Failed to create kernel");

    input_buf_A_top = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * NA, NULL, &status);
    checkError(status, "Failed to create buffer for output");

    input_buf_B_top = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * NB, NULL, &status);
    checkError(status, "Failed to create buffer for output");

    output_buf_C_top = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * NC, NULL, &status);
    checkError(status, "Failed to create buffer for output");

  return true;
}
```

Para cada kernel se creará una cola, de manera que su ejecución pueda ser concurrente. Se utiliza para ello la función de la API de OpenCL “clCreateCommandQueue”. Además, se obtendrá el objeto asociado a cada kernel mediante la función “clCreateKernel”. Estas dos acciones se realizan para cada kernel. Podemos observar como el hecho de tener un array con los nombres de los kernel del array sistólico nos permite utilizar un bucle para crear la cola y el objeto kernel para ellos de manera mucho más simple en nuestro código.

Por otro lado, se crean los buffer, que servirán para la comunicación entre el programa host y el dispositivo OpenCL. La función de la API es “clCreateBuffer” y es necesario indicarle el contexto, el tipo de acceso (read, write o ambos) y el tamaño en bytes del buffer.

En cuanto a la función de nuestro código “init_problem”, su única función será rellenar los vectores de datos con valores aleatorios. Estos valores se obtienen con la función rand_float vista antes.



En este punto, el dispositivo ya está programado, el contexto creado y los objetos kernel, cola y buffer correctamente creados e inicializados. Además, ya se han obtenido los datos de nuestras matrices A y B. Es el momento de enviar estos datos a los buffers y ejecutar los kernel en el dispositivo OpenCL, nuestra FPGA.

Para rellenar los buffers y poner en cola las ejecuciones de distintos kernels, es necesario el uso de eventos, lo que nos permite un cierto control a la hora de ejecutar instrucciones. Tanto las acciones de escritura de buffers como ejecución de kernels tendrán asociado un evento, del tipo de dato cl_event.

En primer lugar encolamos la escritura en los buffers, asociando los eventos write_event[0] y write_event[1]. Al encolar esta escritura, mediante la función clEnqueueWriteBuffer, la cola elegida no tiene importancia, ya que esta escritura será la única acción ejecutándose.

A continuación, se configuran los argumentos de cada kernel, para lo que se utiliza la función "clSetKernelArg" y se inicia la cuenta del tiempo necesario para lanzar y ejecutar los kernels.

```
void run() {
  cl_int status;

  scoped_array<cl_event> write_event(2);
  scoped_array<cl_event> kernel_event(19);
  cl_event finish_event;

  status = clEnqueueWriteBuffer(queues[0], input_buf_A_top, CL_FALSE, 0, sizeof(float) * NA, input_A, 0, NULL, &write_event[0]);
  checkError(status, "Failed to transfer inputA");

  status = clEnqueueWriteBuffer(queues[0], input_buf_B_top, CL_FALSE, 0, sizeof(float) * NB, input_B, 0, NULL, &write_event[1]);
  checkError(status, "Failed to transfer inputA");

  // Set kernel arguments.

  unsigned argi = 0;

  const double start_time = getCurrentTimestamp();

  status = clSetKernelArg(kernel_top, argi++, sizeof(cl_mem), &input_buf_A_top);
  checkError(status, "Failed to set argument %d", argi - 1);

  status = clSetKernelArg(kernel_top, argi++, sizeof(cl_mem), &input_buf_B_top);
  checkError(status, "Failed to set argument %d", argi - 1);

  status = clSetKernelArg(kernel_top, argi++, sizeof(A_width), &A_width);
  checkError(status, "Failed to set argument %d", argi - 1);

  status = clSetKernelArg(kernel_top, argi++, sizeof(B_width), &B_width);
  checkError(status, "Failed to set argument %d", argi - 1);


  argi = 0;

  status = clSetKernelArg(kernel_feeder, argi++, sizeof(Block_Size), &Block_Size);
  checkError(status, "Failed to set argument %d", argi - 1);

  status = clSetKernelArg(kernel_feeder, argi++, sizeof(NMULT), &NMULT);
  checkError(status, "Failed to set argument %d", argi - 1);


  argi = 0;

  status = clSetKernelArg(kernel_MatrixResult, argi++, sizeof(cl_mem), &output_buf_C_top);
  checkError(status, "Failed to set argument %d", argi - 1);

  status = clSetKernelArg(kernel_MatrixResult, argi++, sizeof(C_width), &C_width);
  checkError(status, "Failed to set argument %d", argi - 1);

  status = clSetKernelArg(kernel_MatrixResult, argi++, sizeof(N_BS_C), &N_BS_C);
  checkError(status, "Failed to set argument %d", argi - 1);


  for(int i =0; i<num_kernels; i++){
    argi = 0;

    status = clSetKernelArg(kernels[i], argi++, sizeof(NMULT), &NMULT);
    checkError(status, "Failed to set argument %d", argi - 1);
  }
```

Con los buffers llenos de los datos a procesar y los kernels configurados con sus argumentos es el momento de encolar su ejecución. Aquí distinguimos entre el kernel que se ejecutará como NDRange y los que se ejecutarán como Single Work Item.

En el caso del kernel NDRange, la función necesaria para encolarlo es "clEnqueueNDRangeKernel". Para este tipo de kernels es necesario especificar también el

tamaño que tendrá el work-group y el tamaño general del índice NDRange. En cuanto a los eventos vemos que tendrá que esperar a que finalicen dos eventos, que son los dos dentro del array write_event. La ejecución de este kernel se asociará al evento kernel_event[0]. La cola asociada será la correspondiente a este kernel, donde será la única ejecución.

Para el resto de kernels, que son single work item, la función para encolarlos es "clEnqueueTask", donde los argumentos son iguales que en la función "clEnqueueNDRangeKernel" pero sin especificar tamaño de índice ni de work-group (lo cual es lógico al tener una ejecución single work item, ambos tamaños serían 1). Como en el caso anterior, la ejecución de este kernel tendrá que esperar a que la escritura de los buffers se complete, para lo que espera la finalización de los dos eventos write_event.

```
  const size_t global_work_size[2] = {C_width, C_height};
  const size_t local_work_size[2]  = {BLOCK_SIZE, BLOCK_SIZE};

  status = clEnqueueNDRangeKernel(queue_top, kernel_top, 2, NULL, global_work_size, local_work_size, 2, write_event, &kernel_event[0]);
  checkError(status, "Failed to launch kernel top");

  status = clEnqueueTask(queue_feeder, kernel_feeder, 2, write_event, &kernel_event[1]);
  checkError(status, "Failed to launch kernel feeder");

  status = clEnqueueTask(queue_MatrixResult, kernel_MatrixResult, 2, write_event, &kernel_event[2]);
  checkError(status, "Failed to launch kernel MatrixResult");

  for (int i = 0; i < num_kernels; i++){
    status = clEnqueueTask(queues[i], kernels[i], 2, write_event, &kernel_event[i+3]);
    checkError(status, "Failed to launch kernel %d", i);
  }

  //Read results

  status = clEnqueueReadBuffer(queue_MatrixResult, output_buf_C_top, CL_FALSE, 0, sizeof(float) * NC, output, 1, &kernel_event[18], &finish_event);
  checkError(status, "Failed to read output");

  clWaitForEvents(1, &finish_event);

  const double end_time = getCurrentTimestamp();
  const double total_time = end_time - start_time;

  printf("\nHost time: %0.3f ms\n", total_time * 1e3);

  cl_ulong time_ns = 0;

  time_ns += getStartEndTime(kernel_event[18]);

  printf("Kernel time : %0.3f ms\n", double(time_ns)*1e-6);

  compute_reference();
  verify();

  printf("\n");

  clReleaseEvent(write_event[0]);
  clReleaseEvent(write_event[1]);

  for(int i = 0; i<num_kernels+3; i++){
    clReleaseEvent(kernel_event[i]);
  }

  clReleaseEvent(finish_event);

  for(int i=0; i<num_kernels; ++i) {
    status = clFinish(queues[i]);
    checkError(status, "Failed to finish (%d: %s)", i, kernel_names[i]);
  }
  status = clFinish(queue_feeder);
  status = clFinish(queue_top);
  status = clFinish(queue_MatrixResult);
}
```

Una vez acabada la ejecución, lo que se corresponde con la finalización de los 18 eventos kernel_event, se leerá el buffer de salida, con la función "clEnqueueReadBuffer". Esta es la última acción a encolar, por lo que la ejecución de nuestra implementación finalizará cuando el evento asociado a esta lectura de datos finalice. Para esperar a que finalice este

evento se utiliza la función “clWaitForEvents”. Es en este momento, cuando la ejecución ha finalizado, es cuando se toma de nuevo la muestra del tiempo para obtener el rendimiento de nuestra implementación desde el punto de vista del programa host. Además, obtenemos el tiempo de ejecución desde el punto de vista del dispositivo, para lo que se utiliza la información proporcionada por los eventos.

Tras la llamada a las funciones para la verificación del resultado, que explicaremos a continuación, se liberan los eventos utilizados y se finalizan las colas. Las funciones para este propósito son: “clReleaseEvent” y “clFinish”.

```
void compute_reference() {
  // Compute the reference output.
  printf("Computing reference output\n");

  for(unsigned y = 0; y < C_height; ++y) {
      for(unsigned x = 0; x < C_width; ++x) {
        // Compute result for C(y, x)
        float sum = 0.0f;
        for(unsigned k = 0; k < A_width; ++k) {
          sum += input_A[y * A_width + k] * input_B[k * B_width + x];
        }
        ref_output[y * C_width + x] = sum;
      }
    }

}


void verify() {
  printf("Verifying\n");

  // Compute the L^2-Norm of the difference between the output and reference
  // output matrices and compare it against the L^2-Norm of the reference.
  float diff = 0.0f;
  float ref = 0.0f;
  for(unsigned y = 0; y < C_height; ++y) {
      for(unsigned x = 0; x < C_width; ++x) {
        const float o = output[y * C_width + x];
        const float r = ref_output[y * C_width + x];
        const float d = o - r;
        diff += d * d;
        ref += r * r;
      }
  }

  const float diff_l2norm = sqrtf(diff);
  const float ref_l2norm = sqrtf(ref);
  const float error = diff_l2norm / ref_l2norm;
  const bool pass = error < 1e-6;
  printf("Verification: %s\n", pass ? "PASS" : "FAIL");
  if(!pass) {
    printf("Error (L^2-Norm): %0.3g\n", error);
  }
}
```

Para comprobar si los datos leídos del buffer de salida son correctos, se calculará dentro del programa host el valor de la matriz, de una forma mucho menos eficiente. Sin embargo, esto no es necesario cuando hayamos verificado nuestra implementación y podamos asegurar que el resultado siempre será correcto.

Tras calcular de nuevo la matriz C, comparamos dato a dato la matriz obtenida en la FPGA con la obtenida mediante el algoritmo simple y tradicional en nuestro programa host. Si el error cuadrático es menor que un cierto valor, el resultado se considera correcto. Este pequeño margen se utiliza por posibles errores de cuantificación poco significativos.

Y por último, se liberan todos los recursos OpenCL utilizados, entre los que se encuentran los kernels, las colas, los buffers, el programa y el contexto.

```
void cleanup() {
  for(int i=0; i<num_kernels; ++i) {
    if(kernels[i])
      clReleaseKernel(kernels[i]);
    if(queues[i])
      clReleaseCommandQueue(queues[i]);
  }
  if(kernels[16]) {
    clReleaseKernel(kernels[16]);
  }
  if(queue_top) {
    clReleaseCommandQueue(queue_top);
  }
  if(kernels[17]) {
    clReleaseKernel(kernels[17]);
  }
  if(queue_feeder) {
    clReleaseCommandQueue(queue_feeder);
  }
  if(kernels[18]) {
    clReleaseKernel(kernels[18]);
  }
  if(queue_MatrixResult) {
    clReleaseCommandQueue(queue_MatrixResult);
  }
  if(input_buf_A_top) {
    clReleaseMemObject(input_buf_A_top);
  }
  if(input_buf_B_top) {
    clReleaseMemObject(input_buf_B_top);
  }
  if(output_buf_C_top) {
    clReleaseMemObject(output_buf_C_top);
  }
  if(program) {
    clReleaseProgram(program);
  }
  if(context) {
    clReleaseContext(context);
  }
}
```