UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería de Computadores

Aceleración en FPGA de la descomposición QR de matrices usando rotadores de Givens mediante herramientas HLS

FPGA acceleration of QR decomposition of matrices using Givens rotations by the use of HLS tools

Realizado por
Kostadin Stefanov Gecov

Tutorizado por
Francisco Javier Hormigo Aguilar

Departamento
Arquitectura de computadores

MÁLAGA, julio de 2024

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADUADO EN INGENIERÍA DE COMPUTADORES

Aceleración en FPGA de la descomposición QR de matrices usando rotadores de Givens mediante herramientas HLS

FPGA acceleration of QR decomposition of matrices using Givens rotations by the use of HLS tools

Realizado por
Kostadin Stefanov Gecov

Tutorizado por
Francisco Javier Hormigo Aguilar

Departamento
Arquitectura de computadores

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JULIO DE 2024

Fecha defensa: julio de 2024

Abstract

This undergraduate thesis aims to present the design of a digital accelerator circuit implementing QR decomposition of square matrices with elements of fixed point data type using Givens rotations implemented with the CORDIC algorithm for FPGAs. QR decomposition is a crucial component in various applications, such as solving linear systems and numerical optimization. The QR decomposition algorithm is based on Givens rotations computed with the CORDIC algorithm, providing an efficient solution that only requires addition/subtraction and bit-shifting operations. To leverage the high level of parallelism offered by the FPGA system, a solution has been developed to divide or slice a large matrix into smaller dimension matrices or ”tiles”that are processed in parallel. The adopted methodology focuses on using high-level synthesis tools to efficiently translate C++ code into hardware description language understandable by the FPGA. The results obtained demonstrate a significant improvement in terms of execution time for a intensive computational load algorithm like this one.

Keywords: **FPGA, Givens, CORDIC, high level synthesis, tile**

Resumen

Este trabajo de fin de grado tiene la finalidad de presentar el diseño de un circuito digital acelerador que implementa la descomposición QR de matrices cuadradas con datos de punto fijo empleando rotaciones de Givens implementadas con el algoritmo CORDIC para FPGAs. La descomposición QR es un componente crucial en diversas aplicaciones, como la resolución de sistemas lineales y la optimización numérica. El algoritmo de descomposición QR se basa en rotaciones de Givens computadas con el algoritmo CORDIC, proporcionando una solución eficiente este algoritmo solo requiere de operaciones de suma/resta y desplazamiento de bits. Para aprovechar el gran nivel de paralelismo que ofrece el sistema FPGA, se ha desarrollado una solución la cual divide o trocea una matriz de gran tamaño en matrices de menor dimensión o *"tiles"* que se procesan en paralelo. La metodología adoptada se centra en la utilización de herramientas de síntesis de alto nivel para traducir eficientemente el código C++ a lenguaje hardware adecuado para la FPGA. Los resultados obtenidos demuestran una mejora significativa en términos de tiempo de ejecución para un algoritmo de alta carga computacional como este.

Palabras clave: **FPGA, Givens, CORDIC, síntesis de alto nivel, tile**

Índice

1 Introducción

En esta introducción, se abordan diversos aspectos para contextualizar el propósito de este trabajo, incluyendo la motivación principal y los objetivos perseguidos. Además, se describen las tecnologías empleadas en su desarrollo, así como la estructura que seguirá esta memoria.

1.1. Motivación

Las comunicaciones inalámbricas desempeñan un papel indispensable en la sociedad contemporánea, siendo una columna vertebral de la conectividad global en la era digital. En la actualidad, la importancia de las comunicaciones inalámbricas radica en su capacidad para proporcionar acceso instantáneo y ubicuo a la información, facilitando la comunicación entre personas, dispositivos y sistemas en todo el mundo. Desde la transmisión de datos en tiempo real hasta la habilitación de servicios críticos como la telemedicina y el control remoto de infraestructuras, las redes inalámbricas son fundamentales para el funcionamiento fluido de una amplia gama de aplicaciones y servicios. Además, las comunicaciones inalámbricas son un catalizador para la innovación y el desarrollo económico, permitiendo la creación de nuevas industrias, modelos de negocio y oportunidades laborales. Con el continuo avance tecnológico y la proliferación de dispositivos conectados, las comunicaciones inalámbricas seguirán desempeñando un papel central en la transformación digital de la sociedad, promoviendo la inclusión digital, la colaboración global y el progreso en áreas clave como la salud, la educación y el desarrollo sostenible.

Los sistemas MIMO (Multiple Input Multiple Output) son sistemas de comunicación inalámbrica que utilizan múltiples antenas tanto en el transmisor como en el receptor para transmitir y recibir datos de manera simultánea. Esto permite que el sistema MIMO explote la naturaleza espacial del canal de comunicación, ya que cada antena transmite o recibe

una señal independiente, y el procesamiento de señales avanzado para mejorar la eficiencia espectral, la capacidad y la confiabilidad de los sistemas de comunicación inalámbrica. Estos sistemas se utilizan ampliamente en tecnologías de comunicación inalámbrica, como Wi-Fi, LTE, 5G y sistemas de comunicación por satélite. Sin embargo, como bien se menciona en [5], esta arquitectura incrementa de manera considerable la complejidad a la hora de procesar las señales inalámbricas.

A pesar de las ventajas significativas que ofrecen los sistemas MIMO en términos de rendimiento y capacidad de transmisión, es importante tener en cuenta que su implementación conlleva desafíos técnicos significativos. Por ejemplo, la gestión de la interferencia entre las múltiples señales transmitidas y recibidas, así como la sincronización precisa de los datos en las diferentes antenas, son aspectos críticos que requieren soluciones sofisticadas. Además, la complejidad computacional asociada con el procesamiento de señales en sistemas MIMO puede afectar la eficiencia energética y la latencia de la red. Por lo tanto, mientras que los sistemas MIMO representan una evolución importante en las comunicaciones inalámbricas, su implementación exitosa requiere un enfoque cuidadoso para abordar estos desafíos técnicos y optimizar su desempeño en diferentes entornos de aplicación.

Las operaciones algebraicas sobre matrices son una herramienta fundamental para el procesamiento de señales, más concretamente la descomposición QR de matrices. Sin embargo, realizar todo este procesamiento empleando una (CPU) puede derivar en tiempos de ejecución muy elevados y limitaciones de rendimiento. Aún disponiendo de varios núcleos y técnicas de paralelización, su hardware no está diseñado para abordar operaciones con un alto grado de paralelismo como la descomposición QR de matrices.

A diferencia de las CPUs, las FPGA son dispositivos diseñados específicamente para ejecutar tareas altamente paralelas de manera eficiente. Su arquitectura configurable permite implementar circuitos digitales personalizados que se adaptan perfectamente a los requisitos de la descomposición QR y otras operaciones matriciales.

En 2015, Javier Hormigo y Sergio D. Muñoz [8] presentan una nueva arquitectura para FPGAs basada en el uso de arrays sistólicos de 2 dimensiones, donde los elementos de procesamiento están interconectados y forman una matriz bidimensional, para acelerar la descom-

posición QR de matrices de pocos elementos.

1.2. Objetivos

Este trabajo de fin de grado pretende unir los conceptos descritos en [5] y [8] y diseñar un sistema acelerador para FPGAs de la descomposición QR, empleando el método de rotaciones de Givens y el algoritmo CORDIC, de matrices cuadradas de grandes dimensiones, dividiendo dichas matrices en ”tiles”de menor tamaño para su procesamiento en paralelo.

Para ello, se utiliza el lenguaje de programación C++ junto al entorno de desarrollo de Vitis, que permite sintetizar el código que describe un circuito digital escrito en un lenguaje más accesible para el usuario, como lo son C o C++, en código RTL (Register Transfer Level) para su implementación en dispositivos FPGA de Xilinx.

Previo al inicio del diseño y la implementación del sistema acelerador, se llevó a cabo una preparación exhaustiva sobre las materias relacionadas con el tema del proyecto. Este preparación se basa en:

- Estudio de la descomposición QR de matrices mediante rotaciones de Givens.
- Estudio del algoritmo CORDIC.
- Análisis de los distintos diseños posibles.
- Manejo de herramientas de síntesis de alto nivel.

1.3. Estructura de la memoria

Esta memoria está compuesta de 6 capítulos y 2 apéndices que detallan el trabajo realizado en este trabajo de fin de grado (TFG).

- **Capítulo 1. Introducción**: Se realiza una introducción de la importancia de las comunicaciones inalámbricas en el mundo actual y de los sistemas MIMO. También se describe de forma resumida la finalidad del proyecto y el estudio previo realizado para llevarlo a cabo.

- **Capítulo 2. Sistemas aceleradores mediante Vitis y FPGAs**: Descripción del diseño de una FPGA y de la implementación de aceleradores hardware mediante herramientas de síntesis de alto nivel.
- **Capítulo 3. Algoritmos utilizados para llevar a cabo la descomposición QR**: Describe en profundidad cada uno de los algoritmos empleados en el diseño del sistema acelerador.
- **Capítulo 4. Implementación del circuito acelerador**: Explica la arquitectura elegida para implementar el sistema.
- **Capítulo 5. Pruebas y resultados obtenidos.**: Compara los resultados de la ejecución del sistema acelerador con los resultados de la ejecución de una versión software.
- **Capítulo 6. Conclusiones y líneas de trabajo futuras**: Se comenta una serie de conclusiones obtenidas a partir de los resultados obtenidos. También se listan una serie de posibles mejoras sobre el proyecto a realizar en un futuro.
- **Apéndice A. Manual de utilización**: Una serie de directrices que describe cómo compilar y ejecutar el sistema acelerador.
- **Apéndice B. Ficheros fuente**: Su contenido comprende los ficheros C++ empleados para construir el sistema acelerador.

2

Sistemas aceleradores mediante Vitis y FPGAs

Este capítulo trata sobre la construcción de un sistema acelerador mediante la utilización de dispositivos con hardware reprogramable, como son las FPGAs, así como de las herramientas software empleadas para su diseño.

2.1. FPGA: Arquitectura y Aplicaciones

Una FPGA (Field Programable Gate Array) es un tipo de dispositivo semiconductor que ofrece una flexibilidad excepcional al permitir a los ingenieros reconfigurar la lógica interna y la conectividad de manera programable, lo que las distingue de los circuitos integrados tradicionales. Internamente, está compuesta por CLBs (Configurable Logic Blocks) distribuidos en forma de malla o matriz bidimensional y que están enrutados entre sí mediante conexiones programables, como se observa en la figura 1. Alrededor de estos bloques lógicos configurables, hay distribuidas una serie de interfaces de E/S que permiten la comunicación con dispositivos externos.

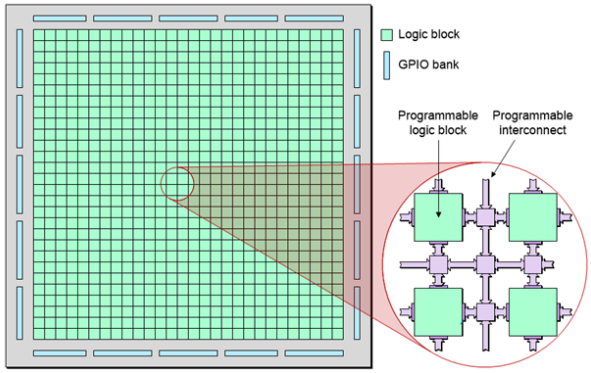

Figura 1: Estructura interna FPGA. Fuente: www.latticesemi.com

Un CLB está formado por diversos elementos entre los que se incluyen. La figura 2 muestra la estructura básica de un CLB:

- **Puertas lógicas**: Permiten realizar funciones lógicas específicas según los requisitos del diseño.
- **Flip-Flops**: Empleados para almacenar estados temporales o para la sincronización de señales dentro del circuito.
- **Multiplexores**: Permiten seleccionar entre varias entradas y dirigir una de ellas a la salida.
- **Memoria Configurable**: Utilizada para almacenar datos temporales o configuraciones de circuito. Estas memorias pueden ser útiles para implementar tablas de búsqueda (LUT), almacenamiento de coeficientes u otras estructuras de datos necesarias para el funcionamiento del diseño.

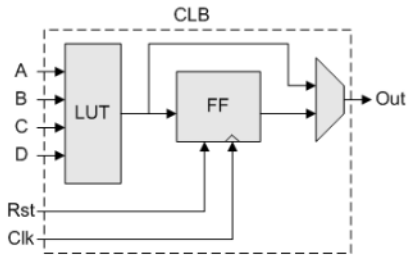

Figura 2: CLB básico. Fuente: www.fpgakey.com

Se observa que la arquitectura de una FPGA difiere mucho en la arquitectura de una CPU convencional. Los microprocesadores carecen de hardware configurable, tienen una arquitectura fija que está diseñada para ejecutar un conjunto específico de instrucciones de manera eficiente. Sin embargo, mientras que una CPU puede ser altamente eficiente para ejecutar algoritmos de propósito general, puede enfrentar dificultades para manejar tareas altamente paralelas. Por ello, se ha elegido diseñar un circuito acelerador para la ejecución en FPGA de la descomposición QR de matrices.

Gracias a su alta versatilidad, las FPGAs pueden utilizarse en una amplia variedad de aplicaciones en diferentes industrias. A continuación se listan una serie de posibles áreas de aplicación:

- **Diseño de prototipos hardware**: Permite verificar tanto el diseño hardware como la aplicación software/firmware de dispositivos de distinto tipo, como ASICs o SoCs, antes de la producción en masa de estos.

- **Criptografía y seguridad**: Son útiles de cara a la implementación de algoritmos de cifrado de datos, autenticación y firmas digitales. Es por esto que se suelen utilizar para el minado de criptomonedas.

- **Procesamiento de datos**: Utilizadas para la aceleración de algoritmos computacionalmente intensivos, que requieren del procesamiento de grandes volúmenes de datos.

2.2. Composición de un sistema acelerador

Una vez visto el diseño de una FPGA, sus ventajas frente a un microprocesador y alguno de los distintos campos de aplicación, es importante conocer de qué elementos está compuesto un sistema acelerador. Habitualmente, está formado por un programa que se ejecuta fuera de la FPGA, en una CPU y se denomina *Host*, y por otro programa denominado *Kernel*, que se ejecuta dentro del hardware de la FPGA. Ambas aplicaciones se comunican entre ellas mediante una interfaz de comunicación que suele ser PCIe.

La aplicación *Host* se aloja en cualquier entorno ejecutado por una CPU. Es la encargada de comunicarse con el *Kernel* y proporcionarle los datos necesarios para su ejecución. En el

presente trabajo, esta parte del sistema acelerador se ejecuta en el servidor *fgpa2.ac.uma.es* del departamento de arquitectura de computadores de la escuela técnica superior de informática.

La otra parte de este sistema está compuesta por uno o más *Kernels*. El *Kernel* es el programa acelerador ubicado dentro del hardware de la FPGA, el cual ejecuta el algoritmo diseñado por el programador con los datos proporcionados por el *Host*. Para la aceleración de la descomposición QR de matrices, se ha diseñado un único *Kernel* para ser ejecutado en la tarjeta *Alveo U200* de Xilinx.

La ejecución de un sistema acelerador puede dividirse en los siguientes pasos, tal y como se describe en [2]:

1. El *Host* escribe en la memoria global de la FPGA los datos necesarios para la ejecución del *Kernel*, establece los parámetros de entrada del *Kernel* y activa su ejecución.

2. Una vez configurado, el *Kernel* comienza a leer los datos alojados en la memoria global mientras que realiza las operaciones pertinentes con estos datos y almacena los resultados de vuelta en la memoria global.

3. Tras finalizar el programa *Kernel*, el *Host* lee y almacena los resultados en su memoria asociada y continúa con su ejecución.

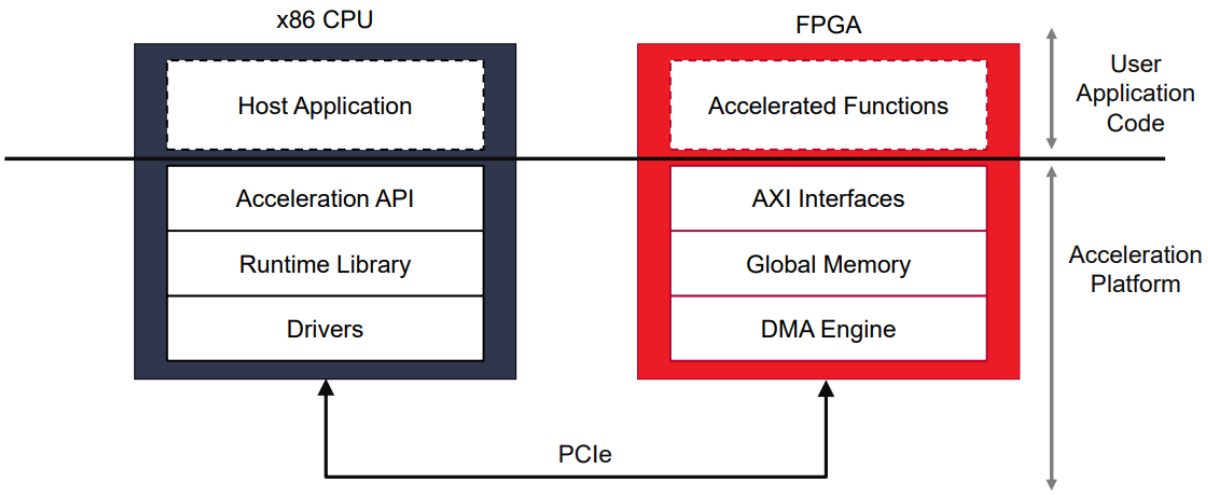

Figura 3: Arquitectura sistema acelerador. Fuente: www.xilinx.com

2.3. Diseño del circuito

El proceso de diseñar un circuito acelerador para FPGAs difiere del proceso de desarrollo de cualquier otro tipo de software. Es un proceso complejo que requiere de más tiempo tanto para un análisis previo del circuito que se quiere llegar a diseñar como tiempo para su posterior desarrollo.

2.3.1. High Level Synthesis

Una de las opciones de las que dispone el programador para diseñar el circuito digital es utilizar lenguajes de descripción hardware como VHDL o Verilog, con los que podrán tener un control preciso del comportamiento del hardware generado. Sin embargo, el uso de estos lenguajes requiere un conocimiento profundo de la arquitectura del hardware y los principios de diseño digital. Además, escribir y depurar código en HDL deriva en un desarrollo lento y propenso a cometer errores de programación.

Por otro lado, está la opción de usar herramientas de síntesis de alto nivel (HLS). La principal función de estas herramientas es traducir un programa escrito en lenguaje de alto nivel, como C++, a lenguaje de descripción hardware. A pesar de tener estas herramientas a su favor, es crucial que el programador comprenda que no está desarrollando un software ordinario. A través del lenguaje de programación de alto nivel está definiendo el comportamiento de los bloques lógicos disponibles en la tarjeta FPGA y estableciendo las conexiones entre ellos. El uso de estas herramientas agiliza notablemente el proceso de desarrollo [7]. Además de verificar el código escrito a nivel de software para asegurar su correcta ejecución, ofrecen la posibilidad de simular el circuito hardware diseñado a nivel de C/C++. Otra gran ventaja es la capacidad de generar un informe post-síntesis que proporciona una estimación de los tiempos y recursos requeridos por el circuito diseñado. Esto facilita la depuración y permite ajustar el diseño según sea necesario. También dan la posibilidad al programador de incluir *pragmas* en su diseño. Los *pragmas* son directivas especiales que permiten modificar el comportamiento de la síntesis de alto nivel y optimizar el diseño del hardware. Entre los más comunes se incluye el pipeline, la partición de arrays o el desenrollado parcial o total de bucles *for*.

2.3.2. Vitis Unified IDE

Vitis Unified es un *framework* de desarrollo creado por Xilinx y que, entre otras cosas más, proporciona un entorno unificado que facilita el diseño de circuitos digitales mediante lenguajes de programación de alto nivel, su verificación y posterior despliegue en los dispositivos FPGA.

La construcción del sistema final implica tres etapas:

1. **Programación de la aplicación *Host* y del *Kernel***: Descripción del sistema mediante el uso del lenguaje C++ y la API de OpenCL para poder comunicar *Host* y *Kernel*.

2. **Compilación**: El *Host* es compilado utilizando el compilador habitual para ficheros C++, g++. Por otra parte, Vitis integra un compilador, v++, que se encarga de construir el objeto Xilinx, con extensión *.xo*.

3. **Enlazado**: En esta etapa descrita en [2], el objeto *.xo* creado se enlaza con fichero de configuración del dispositivo XSA (Xilinx Support Archive), para generar así el binario *.xclbin*, que servirá como argumento de entrada a la aplicación *Host*.

El contenido del binario final puede variar según el tipo de compilación elegido. Vitis dispone de tres modos distintos de construir el programa *Kernel* final:

- **Emulación software**: Crea un modelo del *Kernel* en C/C++ simplemente funcional que sirve para verificar la correcta ejecución del sistema completo, permitiendo depurar con más facilidad en caso de haber errores de programación.

- **Emulación hardware**: Diseña una simulación del modelo RTL (Register Transfer Level) del *Kernel* con la que poder obtener una primera estimación de los tiempos de ejecución, latencias y recursos utilizados. Al ser una simulación del *Kernel*, es recomendable emplear un conjunto reducido de datos para su ejecución.

- **Hardware**: El *Host* ejecuta en la FPGA el modelo hardware real del *Kernel*, proporcionando resultados precisos sobre los tiempos y la utilización de recursos tras la ejecución. En este modo, las operaciones de compilación del *Kernel* y, sobre todo, la fase de enlazado, pueden llevar un tiempo prolongado para completarse.

3

Aplicación de la descomposición QR de matrices

En el siguiente capítulo, se realizará un análisis exhaustivo de los algoritmos fundamentales utilizados en la construcción de la aplicación aceleradora final. Exploraremos su base teórica en el ámbito de las matemáticas, su utilidad dentro de los sistemas MIMO y su adaptación e implementación práctica utilizando el lenguaje de programación C++. Este análisis detallado permitirá comprender plenamente el funcionamiento interno del sistema.

3.1. Sistemas MIMO

Antes de comenzar con el desarrollo del algoritmo de descomposición QR, es pertinente hacer un paréntesis para explicar brevemente la arquitectura de un sistema MIMO. Esto nos proporcionará una mejor comprensión de la utilidad de este algoritmo. MIMO es un sistema utilizado dentro de las comunicaciones inalámbricas en el que, tanto en la parte receptora como en la emisora, se utilizan múltiples antenas para aprovechar la dimensión espacial y tener múltiples caminos entre ambas partes.

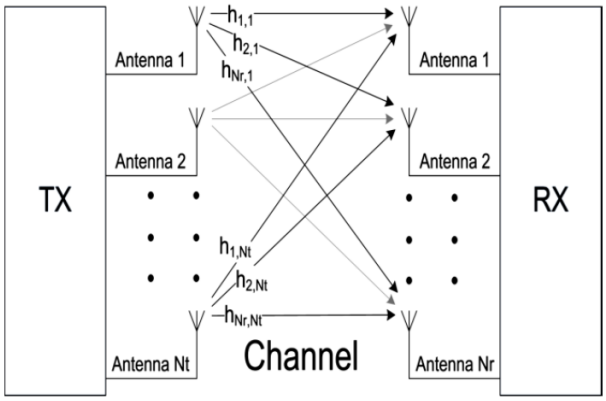

Figura 4: Sistema MIMO. Fuente: www.researchgate.net

3.1.1. Detección MIMO

Uno de los principales retos dentro de estos sistemas es la *detección MIMO*, que consiste en recuperar los datos transmitidos en la parte receptora. Esto implica estimar los símbolos transmitidos en base a las señales recibidas y la información conocida del canal. La detección MIMO se basa en el siguiente sistema lineal:

$$y = Hx + n$$

donde:

- **y** es el vector de señales recibidas.
- **H** es la matriz con información sobre el canal de comunicación.
- **x** es el vector de símbolos transmitidos.
- **n** es el vector de ruido.

La detección se considera un problema de optimización consistente en estimar el vector x a partir de la matriz H y el vector y. Existen varios algoritmos específicos para abordar este problema pero en [5] se enfocan en el uso del algoritmo SD (Sphere Detector), donde se aplica el algoritmo QRD (QR-Decomposition) a la matriz H. A medida que el sistema MIMO aumenta en tamaño, también lo hace la magnitud de la matriz H, por lo que es crucial optimizar al máximo los cálculos necesarios para ofrecer una comunicación de calidad.

3.2. Descomposición QR de matrices

Tras haber visto una introducción al problema en el apartado anterior, esta sección se centrará en describir el algoritmo QRD. La descomposición QR de matrices es un algoritmo del álgebra lineal por el cual se descompone una matriz A en el producto de una matriz ortogonal Q y una matriz triangular superior R [12]

$$A = QR \tag{1}$$

Existen diferentes métodos para el cálculo de la descomposición QR: ortogonalización de Gram-Schmidt, uso de reflexiones de Householder y utilizar rotaciones de Givens: Este último método es el utilizado en este trabajo debido a su alto grado de paralelización.

3.2.1. Rotaciones de Givens

Una rotación de Givens es una operación que se aplica a un par de elementos x e y pertenecientes a la misma columna de una matriz, permitiendo "eliminar" la componente y introduciendo un cero en su posición dentro de la matriz. 2 representa una rotación de Givens para una matriz de dimensiones 2x1, donde c y s se obtienen de la siguiente forma:

- $c = \cos(\theta) = a/r$
- $s = \sin(\theta) = -b/r$

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} \tag{2}$$

El valor de r viene dado por $\sqrt{a^2 + b^2}$, dando la posibilidad de calcular c y s utilizando 3 y 4.

$$c = a/\sqrt{a^2 + b^2} \tag{3}$$

$$s = b/\sqrt{a^2 + b^2} \tag{4}$$

La figura 5 representa la matriz de rotación, donde c y s se colocan en las filas i-ésimas y j-ésimas de la matriz, donde todos los elementos g distintos de cero se dan de la siguiente forma [11]:

- $g_{kk} = 1 \quad \forall k \neq i, j$
- $g_{kk} = c \quad \forall k = i, j$
- $g_{ji} = -g_{ij} = -s$

$$G(i, j, \theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}$$

Figura 5: Matriz rotación de Givens. Fuente: en.wikipedia.org

En el trabajo [8] se detalla con claridad un ejemplo de las rotaciones de Givens aplicada a una matriz 4x4, obteniendo la matriz R como resultado tras aplicar las rotaciones a la matriz inicial. Q se obtiene de la misma forma, solo que aplicando las rotaciones a la matriz identidad de mismas dimensiones que la matriz inicial. Los elementos dentro del área circular son los datos con los que calcular *r*, mientras que los elementos dentro de los rectángulos son aquellos que rotan aplicándoles dicho ángulo.

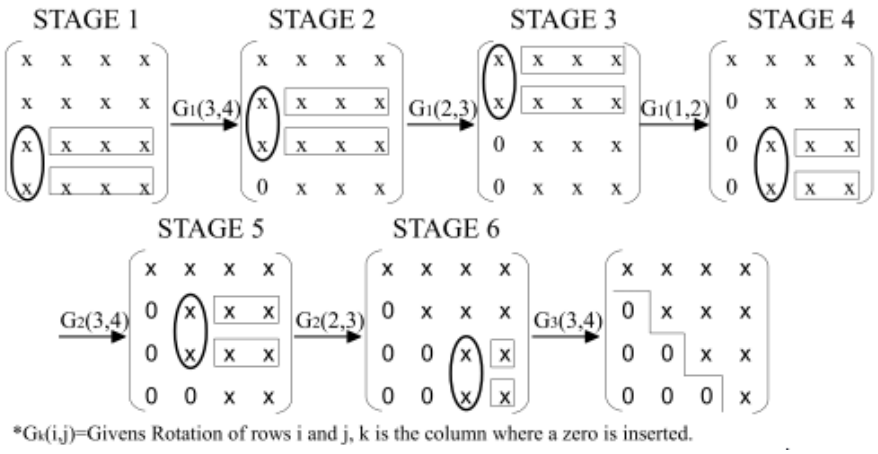

Figura 6: Rotaciones de Givens para una matrix 4x4. Fuente: [8]

Las rotaciones de Givens son operaciones altamente paralelizables. Analizando en detalle el ejemplo [3], se observa que varias de las rotaciones se pueden calcular en el mismo paso, como

las rotaciones $G_1(1,2)$ y $G_1(3,4)$, ya que actúan sobre filas distintas. Lo mismo ocurre con las rotaciones $G_2(1,2)$ y $G_2(3,4)$ en el paso 4. Esta característica permite aprovechar la naturaleza paralela de las FPGA, mejorando el rendimiento y la eficiencia del sistema acelerador.

Tomando como base una de las características del lenguaje C++, la programación orientada a objetos, se crea una clase *Rotator* que describe una rotación de Givens. Esta clase presenta los siguientes atributos y métodos:

- ***row_x_in*** **y** ***row_y_in***: son dos *streams* de datos hls que actúan como colas FIFO, en las que solo se puede escribir y leer su contenido una única vez debido a que se leen y escriben de forma secuencial. Resultan de gran utilidad ya que no requieren de manejo de posiciones de memoria. En este caso, estos dos *streams* almacenan las filas de las coordenadas iniciales X e Y que van a ser rotadas.

- ***row_x_out*** **y** ***row_y_out***: se encargan de almacenar el resultado de la rotación. Los valores almacenados en estos dos *streams* servirán como datos de entrada para otro rotador posterior, así hasta llegar a la solución.

- **Los escalares** ***x, y*** **y** ***col***: identifican las filas y la columna de las correspondientes coordenadas X e Y. Las variables x e y no se utilizan dentro del código, simplemente tienen un rol informativo y auxiliar de cara a la declaración de objetos rotadores.

- **El constructor** ***Rotator***: es el constructor de la clase *Rotator*. Recibe los tres escalares mencionados previamente y sirve para declarar nuevos objetos rotadores.

- **La función** ***givens_rotation***: recibe *row_x_in*, *row_y_in* y *col* como argumentos de entrada y computa la rotación de Givens para esos datos. Las filas rotadas resultantes se almacenan en *row_x_out* y *row_y_out*. Si la coordenada rotada en el eje Y es menor que el umbral de 0'001, se aproxima dicho valor a 0, evitando así acumular más error del necesario para la siguiente rotación.

3.2.2. Algoritmo CORDIC

En este trabajo, se ha utilizado el algoritmo CORDIC (COordinate Rotation DIgital Computer) para calcular el ángulo r de una rotación de Givens. Es un algoritmo iterativo que tiene

la ventaja de que su implementación hardware solo requiere de elementos básicos como sumadores y desplazadores lógicos de bits, evitando tener que implementar elementos hardware más complejos que calculen la raíz cuadrada de un número o su elevación al cuadrado. CORDIC suele estar implementado con datos de punto fijo para operar de forma eficiente. Para ello, utilizando la librería *ap_fixed.h* de Vitis, se ha construido un tipo de dato de punto fijo llamado *data_t*, con 10 bits para la parte entera y 22 para la parte decimal. El resultado obtenido al operar con este tipo de dato será más o menos preciso dependiendo del nº de bits totales asignados para implementar el dato, el nº de bits para la parte entera y el nº de bits para la parte decimal. Esto también influye en el desempeño del sistema acelerador, ya que una rotación CORDIC tomará (nº bits *data_t* - 1) iteraciones en completarse. Cuenta con dos modos de operación:

- **Vectorización**: modo empleado para "eliminar" la coordenada *y*, rotando el ángulo de entrada a 0. Esta operación corresponde con la delimitada por los círculos en 6.

- **Rotación**: se utiliza para rotar el resto de las dos filas que han intervenido en la operación. Se corresponde con la operación realizada en las áreas rectangulares en 6.

Para el presente trabajo, se ha elegido la arquitectura CORDIC propuesta en [8], debido a que es una arquitectura en pipeline que permite iniciar el procesamiento de nuevos elementos antes de que se complete el procesamiento de los anteriores. El cálculo del ángulo *r* se sustituye por el signo de la coordenada *Y* para cada iteración. Siendo *X* e *Y* dos vectores a rotar, *i* la posición del dato actual dentro de los vectores y *k* la iteración actual, se opera de la siguiente forma:

- $x_i = x_i - (y_i \gg k), \quad y_i = y_i + (x_i \gg k) \quad \text{si} \quad y_0 < 0$

- $x_i = x_i + (y_i \gg k), \quad y_i = y_i - (x_i \gg k) \quad \text{si} \quad y_0 \geq 0$

Para compensar el factor de ganancia acumulado en cada valor después de las *n* iteraciones, es necesario multiplicar dicho valor por una constante *Kn*. En [10] se proporciona una tabla que relaciona el número total de iteraciones con la constante *Kn* requerida. En el diseño considerado, con 31 iteraciones totales, la constante *Kn* es igual a $0{,}607252935008881$.

3.3. Tiled QRD

Para una matriz cuadrada, existe una relación matemática entre el número de rotaciones necesarias para obtener la matriz R y su dimensión. Para una matriz de dimensiones N×N, el número de rotaciones requeridas es dado por la fórmula $(N * (N - 1))/2$. Por ejemplo, para una matriz de dimensión 256, se necesitarán un total de 32.640 rotaciones. No obstante, gracias a la arquitectura descrita en [5], se logra reducir considerablemente la cantidad de rotaciones necesarias, lo que permite diseñar hardware eficiente para la descomposición QR de matrices de grandes dimensiones. Esta arquitectura se basa en descomponer una matriz de gran tamaño en matrices o *tiles* más pequeñas que se podrán procesar de forma individual. El *Host* es el encargado de dividir en *tiles* la matriz grande e ir suministrando al *Kernel* dichos *tiles*.

Se denomina panel al conjunto de *tiles* agrupados en forma de columna, uno debajo del otro. Por ejemplo, para un *tile* de 8×8, el panel 1 agrupará los *tiles* entre las columnas 1 y 8 de la matriz grande; el panel 2 agrupará los *tiles* comprendidos entre las columnas 9 y 16. Este proceso se repite sucesivamente hasta alcanzar el máximo de columnas. La figura 7 explica de forma visual este concepto, donde cada cuadrado corresponde con un *tile* y cada color con un panel.

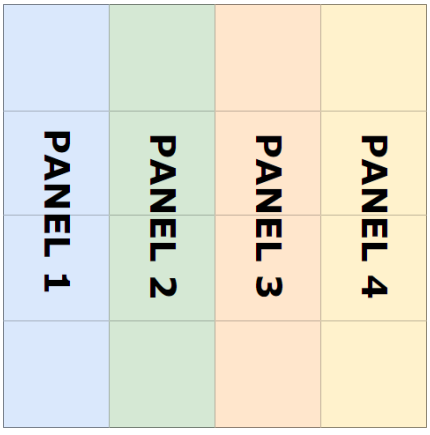

Figura 7: Paneles de una matriz

Dentro de este algoritmo, se llevan a cabo dos operaciones principales que se ejecutan de forma alternada, ilustradas en 8:

1. ***GEQRT***: Se aplica una sola vez por panel. En ella, se computa la descomposición QR

para cada *tile* y dando como resultado la matriz triangular superior de dicho *tile*.

2. ***TTQRT***: Esta operación sirve para ”eliminar” un *tile*. Para ello, se toman dos *tiles* triangulares superiores, se superponen y se aplica la descomposición QR. Los valores del *tile 1* serán las coordenadas *X* y los valores del *tile 2* serán las coordenadas *Y*.

Para distinguir entre ambas operaciones en el código del proyecto, se crean dos *#define*, uno para asociar el valor 0 con la operación GEQRT y otro para asociar el 1 con TTQRT.

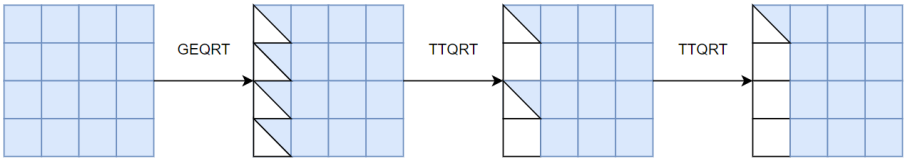

Figura 8: GEQRT y TTQRT aplicados a un panel

Gracias a este procedimiento, se reduce significativamente el número de rotadores requeridos. Para un *tile* de 8x8, se necesitan 28 rotadores para implementar la operación GEQRT y 36 para la operación TTQRT. requiriendo simplemente un total de 64.

4

Implementación del circuito acelerador

En este capítulo se expone la configuración de la aplicación *Host*, abordando tanto el proceso de configuración mediante las funciones de OpenCL para la comunicación con el *Kernel*, como la estrategia de división en *tiles* aplicada a la matriz objeto de descomposición. Además, se examina el diseño del *Kernel* y se detallan los *pragmas* utilizados para aumentar la paralelización y mejorar la latencia del circuito.

4.1. Distribución de la aplicación *Host*

En la programación del *Host* se distinguen tres partes principales: la configuración y programación de la FPGA con el binario *.xclbin*, el control de la ejecución del *Kernel* y el procesamiento post-ejecución. En todas estas partes será necesario utilizar las funciones ofrecidas por OpenCL.

Según [3], *OpenCL* es un estándar multiplataforma que da la posibilidad de escribir aplicaciones paralelas que se ejecutarán dentro de un sistema heterogéneo formado por CPUs, GPUs, FPGAs y otras unidades aceleradoras. Para poder integrar la API de *OpenCL* en el proyecto, es necesario incluir la librería "*xcl2.hpp*" al inicio del fichero del *Host*.

4.1.1. Configuración y programación de la FPGA

Para poder programar la FPGA, lo primero de todo será definir e inicializar las estructuras necesarias dentro de la función *main* del programa. Estas estructuras son las siguientes:

- **string binaryFile**: Almacena el fichero *.xclbin* utilizado para programar la FPGA, que

se recibe como argumento de entrada al programa.

- **cl::Context context**: contiene información sobre el dispositivo *Alveo U200*.
- **cl::CommandQueue q**: Es el mecanismo que actúa como canal de comunicación entre la CPU y el dispositivo. Hay varios tipos de comandos como ejecutar el *Kernel*, iniciar la transferencia de buffers de memoria entre *Host* y *Kernel*, sincronización, etc.
- **cl::Program program**: Este objeto se encarga de programar la FPGA dados un contexto, un dispositivo y un binario.
- **cl::Kernel qrd_kernel**: Objeto que referencia al programa *Kernel* que se va a ejecutar en el dispositivo. Se crea asociándole el nombre de la función del *Kernel* a ejecutar, en este caso "*kernel_givens_rotation*". Además, antes de iniciar su ejecución, hay que establecerle los argumentos que empleará para ello.
- **auto devices**: Es una lista que comprende los dispositivos disponibles encontrados por la función "*xcl::get_xil_devices*". En este proyecto solo aparecerá la tarjeta *Alveo U200* en la lista.
- **auto fileBuf**: Es un puntero al buffer asociado al fichero binario. Se crea a partir de la función "*xcl::read_binary_file*" y el string *binaryFile*.
- **cl::Program::Binaries bins**: Es un objeto que contiene la información asociada al binario utilizado para programar la FPGA.

4.1.2. Ejecución del *Kernel*

Una vez configurado el dispositivo y programado con el binario correspondiente, el siguiente paso será procesar los datos descomponiendo la matriz grande en *tiles* y transferirlos a través de buffers de memoria a la FPGA. En [5] se habla sobre la ejecución en paralelo de los *tiles*, para ello es necesario que el *Kernel* siga un modelo de ejecución paralelo a nivel de tareas e implemente el *pragma INTERFACE ap_ctrl_chain*. Esto se verá más adelante en el capítulo. De cara al *Host*, es necesario que sea capaz de suministrar *tiles* al *Kernel* en cuanto sea posible y que consiga mantener coherencia entre los resultados de cada ejecución del *Kernel* para cada *tile*. Esto se lograría disponiendo de tantos buffers de memoria como *tiles* totales

haya, evitando entrecruzar resultados de distintas ejecuciones. La sincronización entre las distintas ejecuciones se consigue gracias al uso de *Eventos*. Estos objetos se pueden usar para determinar el estado de un comando dentro de una cola de comandos y para establecer ciertas dependencias entre ellos [6]. Sin embargo, el *Host* desarrollado para el sistema acelerador que comprende este trabajo de fin de grado sigue un modelo de ejecución secuencial, en el cual es necesario que termine la ejecución de un *tile* para que pueda comenzar otra.

En las figuras 9 y 10 muestra una pequeña parte de la línea temporal de ejecución del sistema acelerador, donde se observan las diferentes instrucciones que se han ejecutado en cada instante y el comportamiento secuencial del sistema. La primera de ellas corresponde con las operaciones que realiza el *Host* antes de la primera llamada al *Kernel*. Entre ellas están la creación y asociación de los buffers de entrada/salida del sistema, el establecimiento de los argumentos de entrada del *Kernel* y la escritura del primer *tile* en el buffer. Una vez el *Host* haya terminado de preparar los datos para la ejecución del *Kernel*, encola su ejecución y se mantiene a la espera hasta que haya terminado de procesar el *tile*. En la segunda figura mencionada, se observa la línea temporal una vez finalizada la ejecución del *Kernel* y procesado el *tile*. El *Host* se mantiene esperando a que el *Kernel* escriba el resultado en el buffer correspondiente, para recogerlo y preparar el nuevo *tile* a procesar por el *Kernel*. Así sería la línea de ejecución hasta el procesado del último *tile*, tal y como se ha descrito en las figuras 9 y 10

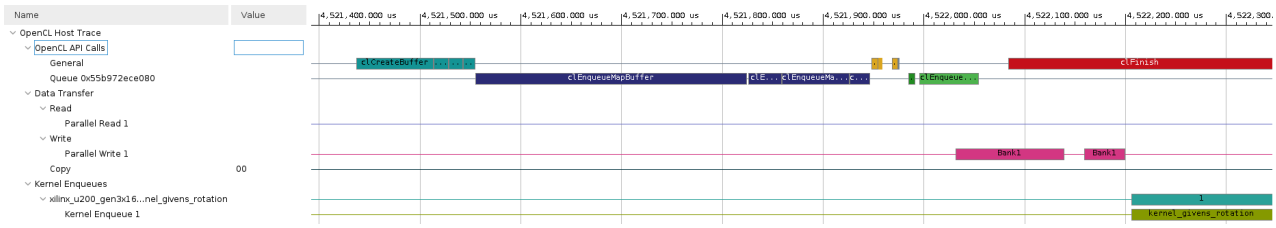

Figura 9: Línea temporal pre-ejecución del *Kernel*

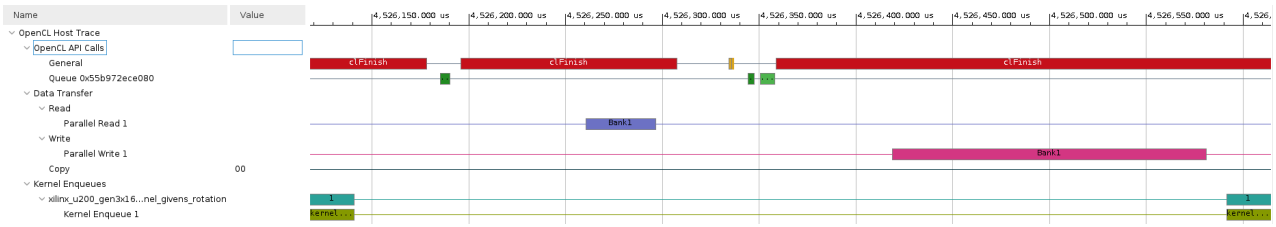

Figura 10: Línea temporal post-ejecución del *Kernel*

Tras leer el fichero de con los datos de entrada y sus valores en un array bidimensional, *A*, se dividirá dicho array en *tiles* de 8 filas cada uno. Los *tiles* son almacenados en un array tridimensional *A_tiled*. La primera dimensión de este array indica el índice del *tile* al que se accede, la segunda indica el índice de la la fila dentro del *tile* y la tercera indica el índice de la columna dentro de esta fila.

Para permitir la transferencia de datos entre *Host* y *Kernel* se crean cuatro buffers distintos: dos de entrada y dos de salida. Se crean dos y dos ya que la operación TTQRT toma dos *tiles* como argumento, al contrario que la operación GEQRT que toma solo uno. Todos ellos se definen con el flag *CL_MEM_ALLOC_HOST_PTR* [4], indicando que tanto la CPU como el dispositivo tienen acceso a ella, y tienen un tamaño asignado de 8 KB cada uno. Los buffers de entrada se definirán utilizando los flags *CL_MEM_HOST_WRITE_ONLY* y *CL_MEM_READ_ONLY*, indicando que la FPGA solo puede leer estos datos y el *Host* solo puede escribir en ellos. Los de salida se definirán de forma contraria utilizando los flags *CL_MEM_HOST_READ_ONLY* y *CL_MEM_WRITE_ONLY*.

Para seleccionar que tipo de operación realizar en cada momento, se emplea un bucle *for* que recorre los índices, empezando en 0, hasta llegar al nº de operaciones totales, indicado por el *#define NUM_OPERATIONS 63*. Los índices pares corresponden a la operación GEQRT (i mód $2 = 0$) y los impares a TTQRT. Una vez elegida la operación, una sentencia *switch* seleccionará el/los *tile/s* a rotar y a partir de qué columna gracias a dos variables auxiliares: *n_iter_xxQRT* y *col_offset_xxqrt*.

El último paso necesario antes de ejecutar el *Kernel* es establecer los argumentos del *Kernel*. OpenCL no permite utilizar arrays con más de una dimensión como argumento, por esto,

antes de mapear el *tile* con su buffer asociado, es necesario ”aplanar” el *tile* a un array de una única dimensión. Hecho esto, se establecen los demás argumentos necesarios, se copian los datos de entrada desde la memoria de la CPU hacia el buffer asociado al dispositivo utilizando *q.enqueueWriteBuffer* y se encola la tarea de ejecución del *Kernel* mediante la llamada a *q.enqueueTask* y se espera a su finalización con la función *q.finish*.

4.1.3. Procesado post-ejecución

El resultado de la operación realizada en el *Kernel* se encuentra almacenado en los buffers de salida. Utilizando *q.enqueueReadBuffer*, se transcribe el contenido del buffer de salida de nuevo al array utilizado previamente para ”aplanar” el *tile*. En el caso de la operación GEQRT, solo será necesario realizar este proceso para un buffer de salida; mientras que para la operación TTQRT, se requerirá para ambos buffers. Una vez procesados todos los *tiles*, se escribe de vuelta en la matriz A la matriz triangular superior R resultado de la descomposición.

4.2. Arquitectura del *Kernel*

El *Kernel* diseñado para el sistema acelerador expuesto en este TFG opera con un tamaño de *tile* de 8×8.

El circuito diseñado seguirá el modelo *dataflow*. Declarando dicho pragma, se consigue aplicar paralelismo entre las distintas tareas del *Kernel* sin necesidad de tener activos distintos hilos de ejecución. A partir de la implementación secuencial de las funciones escritas en C++, Vitis HLS añade de forma automática comunicación y sincronización entre ellas [1]. También hay que destacar el uso del *pragma INTERFACE ap_ctrl_chain*, el cual proporciona una mejor sincronización entre bloques IP (Intellectual Property) que se ejecutan de forma secuencial utilizando varias señales de control.

Este modelo mejora notablemente el rendimiento y la latencia del circuito, logrando un intervalo de inicio (II) máximo de 2 y una frecuencia de 300 MHz para tamaños de matriz de 1024×1024, o de 203,6 MHz para tamaños de 2048×2048 y 4096×4096, siendo la máxima de 300 MHz para la tarjeta *Alveo U200*.

Para implementar correctamente una región *dataflow*, hay que seguir una serie de reco-

mendaciones impuestas por Vitis. Una de las restricciones en estas regiones es que no deberían incluir sentencias condicionales, bucles, ni manejo de excepciones. Por lo tanto, la función *kernel_givens_rotation*, al contener un bloque *if_else* para seleccionar la operación correspondiente, no es adecuada para la aplicación de *dataflow*. Como solución, se han creado las funciones *kernel_givens_rotation_GE* y *kernel_givens_rotation_TT*, las cuales cumplen con las recomendaciones para implementar *dataflow* y siguen el patrón de diseño *load-compute-store* para minimizar el nº de accesos a memoria, como se observa en el diagrama 11.

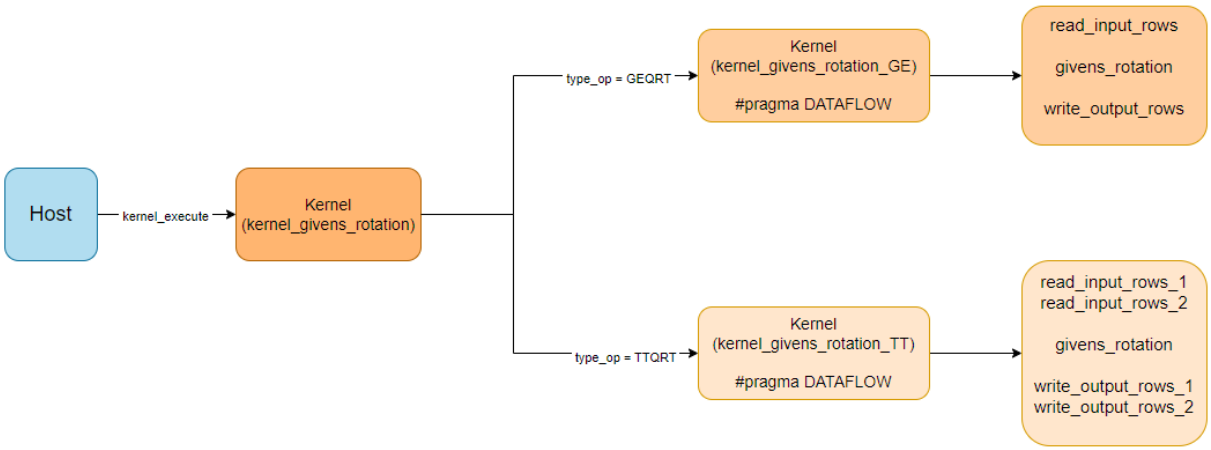

Figura 11: Diagrama de diseño del *Kernel*

La estructura de *kernel_givens_rotation_GE* y *kernel_givens_rotation_TT* es prácticamente igual, las únicas diferencias son el nº de *tiles* sobre las que operan y los rotadores necesarios creados para cada tipo de operación. La función *read_input_rows* se encarga de cargar en los *streams* correspondientes a los rotadores iniciales cada una de las filas del *tile* almacenado en **input*. Una vez se hayan cargado los *streams* con datos, se pasa a ejecutar la función *givens_rotation*, que rota dos filas del *tile* siguiendo el mismo algoritmo explicado en el capítulo 3.

Dentro de la función, los valores almacenados en los *streams* se transfieren a arrays para poder realizar operaciones con ellos. Esto es necesario porque los *streams* actúan como colas FIFO (First-In, First-Out), donde los datos solo pueden ser leídos o escritos una vez, limitando su reutilización directa.

Una vez que los datos están en los arrays, se procede a rotar las filas según sea necesario. Después de esta operación, los datos modificados en los arrays se escriben de nuevo en los *streams* de salida asociados a los rotadores.Este proceso se repite iterativamente hasta que se hayan vectorizado las 8 columnas del *tile* y se hayan rotado todas las filas restantes.

Finalmente, para concluir la operación del *Kernel*, se realiza un proceso inverso al de la función *read_input_rows*. Utilizando la función *write_output_rows*, los resultados se escriben en el puntero de salida **output*, completando así la operación de manera efectiva.

5

Pruebas y resultados obtenidos

En este capítulo se analiza el rendimiento alcanzado por el sistema acelerador de la descomposición QR y los recursos FPGA utilizados para su implementación. También se realiza una comparativa de los tiempos de ejecución y la precisión en el resultado entre la implementación hardware y una implementación software de la descomposición QR de matrices mediante rotaciones de Givens para tamaños 1024×1024, 2048×2048 y 4096×4096. La implementación software difiere en varios detalles respecto a la hardware. Se cambia el tipo de dato utilizado por el tipo *float*, ya que no existen librerías que implementen y soporten tipos de datos de punto fijo, como lo es la librería *ap_fixed* dentro del entorno de Vitis. Debido a esto, no es posible aplicar el algoritmo CORDIC, ya que los datos de tipo *float* no admiten operaciones de desplazamiento de bits. Tampoco se fragmentará la matriz de entrada en *tiles*, sino que se procesará como una única matriz. Las CPUs no siguen la misma arquitectura que las FPGAs y tienen otras técnicas distintas de paralelización, por lo que la fragmentación no aportaría ningún beneficio y supondría una ejecución más lenta. Esta implementación software equivalente se ejecutará dentro del servidor ”nombre servidor”, compuesto por un procesador Intel i9-9820X y el compilador *g++* v11.3.0.

5.1. Pruebas

Como se comenta al inicio del capítulo, se procede a evaluar el desempeño del acelerador comparando el tiempo de ejecución y la precisión del resultado. Sin embargo, primero se expone la latencia de operación y el porcentaje de ocupación del *Kernel* dentro de la FPGA para distintos tamaños de matriz.

5.1.1. Latencia

Para realizar este análisis, se toma como ejemplo los resultados de latencia obtenidos para una matriz de tamaño 4096x4096. En la tabla 5.1.1 se presenta información sobre la latencia en el mejor caso, el caso promedio y el peor caso para cada módulo dentro del *Kernel*. Realmente, debido al *pragma DATAFLOW*, Vitis extrae un mayor número de módulos, sin embargo, para evitar redundancias, se incluye solo un módulo representativo de cada tipo en la tabla.

Las operaciones de lectura/escritura mantienen una latencia constante porque siempre manejan la misma cantidad de datos, correspondiente al tamaño de una fila de la matriz. Tanto *read_input_data* como *write_output_data* tienen una latencia muy similar al número de elementos en una fila, que en este caso es 4096. Esto significa que prácticamente leen o escriben un elemento por cada ciclo de reloj. Dicha eficiencia se debe a que la lectura o escritura se está realizando sobre un *hls-stream* con estructura *FIFO* cuya latencia de operación es cero. La diferencia de dos ciclos observada corresponde con la latencia de iteración, es decir, la diferencia de ciclos entre el inicio de una iteración y el inicio de la siguiente. El impacto de la latencia de iteración se reduce porque los datos se procesan utilizando una arquitectura *pipeline*.

El módulo *compensate_scale_factor* también mantiene un comportamiento similar, pero con una latencia de iteración de 5 ciclos. En cuanto al módulo *cordic*, este incluye dos bucles: uno procesa una fila completa de la matriz y el otro procesa dos filas. Ambos son capaces de procesar un elemento por ciclo. Multiplicando el tamaño de una fila por el número de filas procesadas, se obtiene $12{,}288$ elementos, un valor muy cercano al número de ciclos necesarios en el peor caso. Este comportamiento es similar en otros módulos, que actúan como módulos padres que engloban varios submódulos.

La variabilidad en el número de ciclos que un mismo módulo puede requerir se debe a que, a medida que avanzan las rotaciones de Givens, el número de elementos a procesar en una fila disminuye porque se han ido reduciendo a cero con las rotaciones anteriores.

Para convertir los resultados mostrados en la tabla a unidades de tiempo, basta con dividir el número de ciclos entre la frecuencia máxima alcanzada, que en este caso es de $203,6$ MHz. Los resultados obtenidos para matrices de tamaños 2048×2048 y 1024×1024, con frecuencias máximas alcanzadas de $203,6$ MHz y 300 MHz, respectivamente, son similares a los mostrados

en la tabla, pero divididos por factores de dos y cuatro, respectivamente. Esto implica que al reducir el tamaño de la matriz a la mitad o a un cuarto, el número de ciclos necesarios también se reduce proporcionalmente.

Module Name	**Start Interval**	**Best (cycles)**	**Avg (cycles)**	**Worst (cycles)**
read_row	4171	4171	4171	4171
read_input_rows	33383	33383	33383	33383
read_input_data	4098	4098	4098	4098
update_quadrant	18 ∼8194	18	4106	8194
cordic	4121 ∼12297	4121	8209	12297
compensate_scale_factor	4101	4101	4101	4101
givens_rotation	140184 ∼401840	140184	268970	401840
write_row	4170	4170	4170	4170
write_output_rows	33375	33375	33375	33375
write_output_data	4098	4098	4098	4098
kernel_givens_rotation_TT	140185 ∼401841	140389	269175	402045
kernel_givens_rotation_GE	140185 ∼401841	140440	269226	402096
kernel_givens_rotation	2 ∼402098	1	201908	402097

Cuadro 1: Información sobre la latencia de los distintos módulos del *Kernel*

5.1.2. Área

Para este análisis se tendrán en cuenta cuatro tipos distintos de recursos: *Look Up Tables*, *Flip Flops*, *Digital Signal Processors* y *Block RAM*. En la tabla 2 se puede ver el porcentaje de utilización de cada recurso sobre el total disponible en la tarjeta *Alveo U200* para cada tamaño de matriz. Se observa que los resultados apenas varían de una implementación a otra, a diferencia de la utilización de BRAM. Debido a que el tamaño de *tile* es pequeño y a la optimización del diseño, el uso de LUTs y FFs es muy bajo. Por otro lado, el bajo porcentaje de utilización de los bloques DSP se debe, en parte, al uso del algoritmo CORDIC dentro del diseño, ya que este algoritmo permite evitar la implementación de operaciones matemáticas complejas. Los únicos recursos cuyo porcentaje de uso sí varía más entre las distintas implementaciones son las

BRAM. Los BRAM son recursos de memoria que pueden servir como memorias ROM/RAM o colas FIFO para almacenar ciertos datos o implementar buffers. Es por esto que, a medida que aumente el número de elementos por fila de la matriz, también lo hará el número de BRAM necesarios.

Tamaño matriz	**LUT**	**FF**	**BRAM**	**DSP**
1024×1024	120345 [11.89 %]	87811 [4.14 %]	375 [19.64 %]	256 [3.75 %]
2048×2048	124436 [12.29 %]	89213 [4.21 %]	719 [37.66 %]	256 [3.75 %]
4096×4096	124723 [12.32 %]	93595 [4.41 %]	799 [41.85 %]	256 [3.75 %]

Cuadro 2: Utilización de recursos de la FPGA para cada tamaño de matriz

5.1.3. Comparativa de tiempos y precisión de resultados

En la tabla 12, utilizando una escala logarítmica en el eje Y para mejorar la visualización, se observa que el diseño hardware implementado no mejora en ninguno de los casos a la implementación software en cuestión de tiempos de ejecución. De hecho, la implementación software es 24 veces más rápida en los casos 4096 y 2048, y 17 veces más rápida en el caso 1024.

Esta gran diferencia de tiempos esta provocada por el procesamiento de forma secuencial de los *tiles* por parte del *Host*, como se explicaba con anterioridad en el capítulo 4. La ineficiencia del *Host* puede comprobarse revisando, con Vitis Analyzer, el *Profile Summary*, dentro del *Run Summary*. Este resumen contiene una comparativa entre las diferentes llamadas ejecutadas sobre la API de OpenCL, el número de veces que se han ejecutado y el tiempo total que han tardado en completarse.

En el cuadro 3 aparecen los datos registrados para el caso 1024. Estos datos muestran que el 88 % de la suma de todos los tiempos totales corresponde a la llamada a *clFinish*. Dicha API es llamada a la hora de lanzar una ejecución del *Kernel* y también a la hora de encolar la operación de lectura de resultados. El tiempo que el *Host* está en ese estado de ”espera” es tiempo de procesamiento que se desperdicia.

En el caso de tener un *Host* capaz de procesar *tiles* en paralelo, se podrían procesar 128,

256 o 512 *tiles*, dependiendo del tamaño de la matriz de entrada, a la hora de descomponer el primer panel. Esto reduciría considerablemente los tiempos de ejecución totales, ya que el *Host*, en vez de estar en estado *idle* esperando la terminación del procesamiento de un *tile*, estaría ocupado gestionando la ejecución del resto de *tiles*.

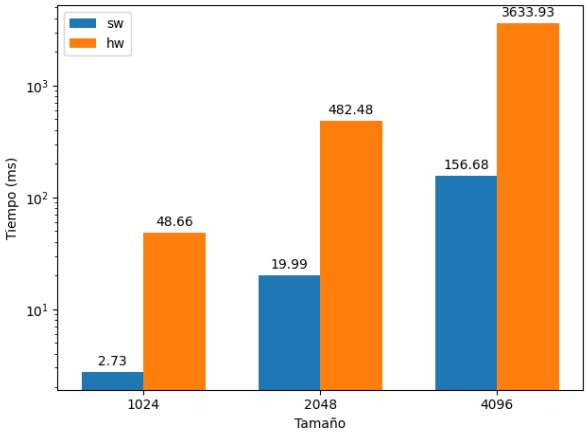

Figura 12: Comparativa de tiempos de ejecución entre las implementaciones SW y HW para los diferentes tamaños de matriz

Los datos de entrada del circuito se extraen de un fichero llamado *data_in_N.dat*, siendo N las dimensiones de la matriz cuadrada creada. Este fichero se genera mediante un script escrito en Python que se encarga de crear una matriz de NxN números en coma flotante de simple precisión, cuyos valores están comprendidos en el intervalo [-1, 1] y escribir la matriz en el fichero. Utilizando otro script desarrollado en Python, se calcula la descomposición QR de la matriz creada previamente utilizando la función *numpy.linalg.qr* de la librería *numpy* [9], obteniendo como resultado la matriz *R* y almacenándola en un fichero llamado *data_out_gold_N.dat*. Este fichero se utilizará para comprobar que el resultado obtenido tras la ejecución del *Kernel* sea correcto. Para ello, se diseña una función llamada *mse* que, para cada una de las posiciones no nulas de la matriz, es decir, compara el valor obtenido de la descomposición y el valor calculado a través del script mencionado anteriormente, ambos en valor absoluto, ya que el resultado obtenido puede tener el signo opuesto al resultado computado con Python. La diferencia se acumula en la variable *err* y luego se divide entre el nº de elementos comprendidos en la matriz triangular superior, incluyendo la diagonal. Estos cálculos hacen referencia a la formula 5, siendo $M * N$ el nº de elementos mencionado.

$$MSE = \frac{1}{MN} \sum_{i=1}^{M} \sum_{j=1}^{N} (|A_{ij}| - |out\_gold_{ij}|)^2 \tag{5}$$

El acelerador hardware opera con un tipo de dato de punto fijo de 32 bits, con 12 bits destinados a la parte entera y 20 a la parte decimal. Por otro lado, la implementación software emplea datos de punto flotante de precisión simple. En la tabla 4 se observa que la precisión de la implementación software, a pesar de ser mayor en los dos primeros casos, va disminuyendo a medida que se aumenta el tamaño de la matriz. Los valores en punto flotante tienen un rango finito de número reales que pueden representar debido a su limitado número de bits. Estos valores se representan utilizando 23 bits para la mantisa, 8 para el exponente y 1 para el signo. Entonces, cuando un valor numérico real no se puede representar por culpa de la falta de precisión, se suele redondear al valor representable más cercano. Por lo tanto, al aumentar el tamaño de la matriz, aumenta el número de operaciones aritméticas a realizar, por lo que el error en el redondeo se hace más grande.

Sin embargo, los valores numéricos representados con datos de punto fijo son más consistentes ya que siguen un sistema de representación distinto a los datos de punto flotante. Los datos de punto fijo se almacenan en memoria como un entero y se representan como ese número entero dividido entre un factor de escalado. Es por esto que la precisión se mantenga constante independientemente del tamaño de la matriz.

A pesar de que la precisión sea mayor en los dos primeros casos en la implementación software, la precisión obtenida por el acelerador para los distintos tamaños de matriz es más que suficiente para computar descomposición QR de matrices.

API Name	Calls	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)
clCreateBuffer	4	0.11	0.01	0.03	0.08
clCreateCommandQueue	1	0.01	0.01	0.01	0.01
clCreateContext	1	12.12	12.12	12.12	12.12
clCreateKernel	1	0.59	0.59	0.59	0.59
clCreateProgramWithBinary	1	3789.70	3789.70	3789.70	3789.70
clEnqueueMapBuffer	4	0.39	0.02	0.10	0.27
clEnqueueMigrateMemObjects	32768	499.20	0.00	0.02	0.40
clEnqueueTask	16384	565.04	0.01	0.03	0.17
clFinish	32768	44377.10	0.00	1.35	4.35
clGetDeviceIDs	2	0.00	0.00	0.00	0.00
clGetDeviceInfo	2	8.75	0.01	4.37	8.74
clGetPlatformIDs	2	0.02	0.00	0.01	0.02
clGetPlatformInfo	2	0.00	0.00	0.00	0.00
clReleaseCommandQueue	16386	46.31	0.00	0.00	0.36
clReleaseContext	16386	40.53	0.00	0.00	0.35
clReleaseDevice	4	0.01	0.00	0.00	0.00
clReleaseKernel	16386	46.36	0.00	0.00	0.36
clReleaseMemObject	163588	420.96	0.00	0.00	3.13
clReleaseProgram	1	4.87	4.87	4.87	4.87
clRetainContext	16385	37.76	0.00	0.00	0.38
clRetainDevice	4	0.01	0.00	0.00	0.01
clRetainKernel	16385	34.10	0.00	0.00	0.36
clRetainMemObject	163584	361.39	0.00	0.00	3.33
clSetKernelArg	32772	85.43	0.00	0.00	0.72

Cuadro 3: Comparativa entre las diferentes APIs de OpenCL, el nº de llamadas a estas y los tiempos de ejecución para la ejecución del acelerador con una matriz de tamaño 1024×1024.

Tamaño Matriz	**HW**	**SW**
1024×1024	7.8507e-08	2.00438e-11
2048×2048	2.27105e-07	1.46589e-10
4096×4096	7.8507e-08	7.00853e-07

Cuadro 4: Precisión de resultados entre la implementación HW y SW para los distintos tamaños de matriz.

6

Conclusiones y líneas de trabajo futuras

En este último capítulo del presente trabajo de fin de grado se exponen los puntos fuertes del circuito acelerador diseñado, las limitaciones encontradas durante y después del desarrollo y las posibles líneas futuras de trabajo sobre este proyecto, que permita optimizar y mejorar la solución expuesta en este proyecto.

6.1. Conclusiones

A partir de los datos y resultados mostrados en el anterior capítulo, se exponen tanto las ventajas como los inconvenientes que presenta este sistema acelerador para la descomposición QR de matrices:

- **El acelerador implementa eficientemente el paradigma *productor-consumidor*.** Uno de los principales principios de diseño enumerado en [1] es el paradigma del productor-consumidor, que consiste en dividir la funcionalidad de las tareas para aprovechar el paralelismo. Gracias a la estructuración del código del *Kernel* en un patrón *load-compute-store* y al uso de las directivas *pipeline* y *dataflow* y al uso de *streams*, se consigue paralelizar el diseño.

- **El circuito diseñado solo acepta un tamaño de *tile*.** La funcionalidad del *Kernel* está limitada debido a que solo admite el procesamiento de *tiles* de tamaño de 8×8. Este tamaño de *tile* permite reducir la utilización de recursos de la FPGA, sin embargo, puede suponer un mayor *overhead* a la hora de descomponer matrices de gran tamaño, debido

a las múltiples llamadas necesarias al *Kernel*. Este problema de escalabilidad proviene del enfoque individual que se le ha dado a la clase *Rotator* en el código. Las rotaciones de Givens están muy interdependientes, ya que la salida de un rotador es la entrada de otro. Debido a esta forma de abordar las rotaciones y a la ausencia de un patrón que permita generalizar las instrucciones dentro de un bucle, no se ha podido versatilizar el tamaño de *tile* aplicado.

- **Porcentaje bajo de la utilización de recursos del dispositivo.** Como se ha podido ver en el capítulo anterior, el porcentaje de utilización de recursos disponibles en la FPGA es bastante reducido debido al pequeño tamaño de *tile*. Esto da la posibilidad para añadir más unidades de cómputo y procesar varios *tiles* en paralelo. Sin embargo, hay que tener en cuenta que, cuanto mayor sea el tamaño de la matriz a descomponer, mayor será el porcentaje de utilización de BRAM, llegando a limitar el tamaño máximo de matriz a descomponer.

- **Procesamiento secuencial de *tiles* por parte del *Host*.** En el capítulo anterior se ha expuesto el problema que supone esto, ya que dispara los tiempos de ejecución totales del acelerador.

Según las pruebas realizadas y el análisis de los resultados, se puede concluir que el *Kernel* cumple con un diseño e implementación eficientes en términos de latencia y utilización de recursos, gracias a la división de la descomposición QR en tareas más reducidas y su procesamiento en paralelo y al uso del algoritmo CORDIC, que reduce la complejidad en las operaciones de rotación y vectorización. El verdadero cuello de botella del diseño expuesto en este TFG es el programa *Host* por lo que se ha expuesto a lo largo de los capítulos anteriores. A pesar de que el *Kernel* implementa el *pragma INTERFACE ap_ctrl_chain*, permitiendo solapar múltiples ejecuciones, siguiendo un modo de funcionamiento en *pipeline*, es importante que el *Host* mantenga la coherencia entre los resultados de distintas ejecuciones del *Kernel*. Esto se puede lograr mediante el uso de colas de comandos OOO (Out-Of-Order) y eventos para encolar múltiples peticiones de ejecución. Además, debe albergar tantos buffers como sea necesario, según el número de ejecuciones solapadas, para poder enviar y recibir datos sin alterar o entrelazar los resultados. Sin embargo, a medida que la matriz a descomponer sea mayor, es más probable que aparezcan problemas a la hora de intentar definir buffers de tantos ele-

mentos. Por ejemplo, es posible que sistemas que utilicen un direccionamiento de 32 bits no sean capaces de asignar direcciones de memoria a posiciones de un buffer, en caso de que este tenga un tamaño muy elevado.

6.2. Líneas de trabajo futuras

Este proyecto está abierto a varias opciones para continuar con el trabajo en un futuro. Entre ellas, se proponen las siguientes líneas de trabajo:

1. **Diseñar un acelerador versátil con tamaño de *tile* configurable.** Como se ha comenta en el apartado anterior, el sistema acelerador solo admite una configuración de *tile* fija. Una posible mejora sería permitir distintas configuraciones de tamaños de *tile*, por ejemplo, a través de un script *Tcl*. Permitiendo escoger un tamaño distinto para el *tile* da la posibilidad de probar distintos diseños y elegir el que más adecuado para cada situación según se requiera aumentar el rendimiento o reducir el área.

2. **Utilizar más de una unidad de cómputo.** Poder usar varias unidades de cómputo del mismo *Kernel* daría la posibilidad al sistema acelerador a dividir la carga computacional y aumentando el grado de aceleración del algoritmo.

3. **Paralelizar el procesamiento de *tiles* desde el *Host***. Esto se puede lograr añadiendo más unidades de cómputo o como se ha expuesto con anterioridad en el capítulo 4

Referencias

[1] Inc. Advanced Micro Devices. *Vitis High-Level Synthesis User Guide*. Ver. UG1399 (v2022.2). Dic. de 2022.

[2] Inc. Advanced Micro Devices. *Vitis Unified Software Platform Documentation: Application Acceleration Development*. Ver. UG1393 (v2022.2). May. de 2023.

[3] Khronos Group. *OpenCL 3.0 Reference Guide*. Ver. Rev. 0922. Jun. de 2023. URL: https://www.khronos.org/files/opencl30-reference-guide.pdf.

[4] Vincent Hindriksen. *OpenCL Basics: Flags for the creating memory objects*. (accessed Oct. 7, 2023). URL: https://streamhpc.com/blog/2013-02-03/opencl-basics-flags-for-the-creating-memory-objects/.

[5] Cang Liu et al. "Hardware Architecture Based on Parallel Tiled QRD Algorithm for Future MIMO Systems". En: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.5 (2017), págs. 1714-1724. DOI: 10.1109/TVLSI.2016.2645841.

[6] Matt Sellitto, Dana Schaa. *OpenCL Events, Synchronization, and Profiling*. Mar. de 2024. URL: https://ece.northeastern.edu/groups/nucar/Analogic/Class5-B-Events.pdf.

[7] Wim Meeus et al. "An overview of today's high-level synthesis tools". En: *Design Autom. for Emb. Sys.* 16 (sep. de 2012), págs. 31-51. DOI: 10.1007/s10617-012-9096-8.

[8] Sergio D. Muñoz y Javier Hormigo. "High-Throughput FPGA Implementation of QR Decomposition". En: *IEEE Transactions on Circuits and Systems II: Express Briefs* 62.9 (2015), págs. 861-865. DOI: 10.1109/TCSII.2015.2435753.

[9] NumPy Developers. *Linear algebra (numpy.linalg)*. (accessed Dec. 10, 2022). URL: https://numpy.org/doc/stable/reference/routines.linalg.html#module-numpy.linalg.

[10] Inc. The MathWorks. *Perform QR Factorization Using CORDIC*. accessed Nov. 17, 2022. URL: https://es.mathworks.com/help/fixedpoint/ug/perform-qr-factorization-using-cordic.html#d126e120025.

[11] Wikipedia contributors. *Givens rotation — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Givens_rotation&oldid=1218150354. (accessed Oct. 16, 2022).

[12] Wikipedia contributors. *QR decomposition — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=QR_decomposition&oldid=1216197164. (accessed Oct. 16, 2022).

Apéndice A
Manual de uso

En este apéndice se detallan los pasos a seguir para ejecutar correctamente el sistema acelerador utilizando el entorno de desarrollo de Vitis.

A.1. Ficheros necesarios

Antes de nada, es necesario tener el entorno de desarrollo de Vitis instalado. En este trabajo se utiliza la versión 2022.2. También es necesario disponer del paquete de soporte para la tarjeta Alveo U200 de Xilinx. Una vez completados estos requerimientos, se pueden descargar los ficheros del sistema acelerador clonando el repositorio desde Github o descargando directamente el .zip del propio repositorio. El repositorio contiene los ficheros con el código fuente del *Host* y del *Kernel*, la librería de *OpenCL* y un conjunto de ficheros .dat con los datos de entrada del programa y la salida *Gold* utilizada para el cálculo del error en la salida de la ejecución:

- **host.cpp**
- **kernel.cpp**
- **xcl2.cpp**
- **xlc2.hpp**
- **data_in_1024.dat**
- **data_out_gold_1024.dat**
- **data_in_2048.dat**
- **data_out_gold_2048.dat**
- **data_in_4096.dat**
- **data_out_gold_4096.dat**

A.2. Modificaciones en el código fuente

Tanto en el fichero *host.cpp* como en *kernel.cpp*, se incluyen una serie de *#define* esenciales para garantizar la correcta ejecución del sistema acelerador. Uno de estos define se refiere a la dimensión de la matriz a descomponer. Dependiendo del conjunto de datos que se pretenda utilizar en la ejecución, es necesario ajustar el valor del *#define TAM* en consecuencia. Por ejemplo, si los ficheros de datos elegidos son *data_in_1024.dat* y *data_out_gold_1024.dat*, *TAM* deberá tener el valor 1024 asociado.

A.3. Ejecución de la implementación utilizando el entorno Vitis

Tal y como se menciona en [2], el primer paso es ejecutar el comando *source <ruta_instalación_Vitis>/Vitis/*
para configurar las variables de Vitis.

```
kgecov@fpga2:/tools/Xilinx/Vivado/2022.2$ source settings64.sh
```

Figura 13: Comando para configurar el entorno Vitis

El siguiente comando a ejecutar es *vitis*. Esto abrirá una pestaña con el entorno de desarrollo de Vitis.

```
kgecov@fpga2:/tools/Xilinx/Vivado/2022.2$ vitis

****** Xilinx Vitis Development Environment
****** Vitis v2022.2 (64-bit)
  **** SW Build 3671529 on 2022-10-13-17:52:08
    ** Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.

Launching Vitis with command /tools/Xilinx/Vitis/2022.2/eclipse/lnx64.o/eclipse
-vmargs -Xms64m -Xmx1024m -Dorg.eclipse.swt.internal.gtk.cairoGraphics=false -Do
sgi.configuration.area=@user.home/.Xilinx/Vitis/2022.2 --add-modules=ALL-SYSTEM
--add-opens=java.base/java.nio=ALL-UNNAMED --add-opens=java.desktop/sun.swing=AL
L-UNNAMED --add-opens=java.desktop/javax.swing=ALL-UNNAMED --add-opens=java.desk
top/javax.swing.tree=ALL-UNNAMED --add-opens=java.desktop/javax.swing.plaf.basic
=ALL-UNNAMED --add-opens=java.desktop/javax.swing.plaf.synth=ALL-UNNAMED --add-o
pens=java.desktop/com.sun.awt=ALL-UNNAMED --add-opens=java.desktop/sun.awt.X11=A
LL-UNNAMED &
```

Figura 14: Comando para abrir el entorno de desarrollo de Vitis

Una vez abierto el entorno de desarrollo de Vitis, se siguen los siguientes pasos:

1. **Abrir un nuevo *Application Project.***

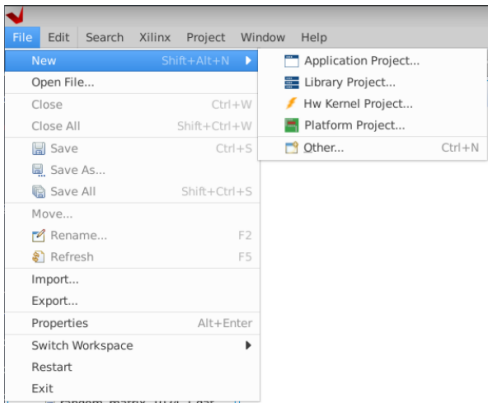

Figura 15: Crear nuevo *Application Project*

2. **Escoger la plataforma a utilizar**. En este caso aparece únicamente la tarjeta Alveo U200. Se selecciona y se pincha en *Next*.

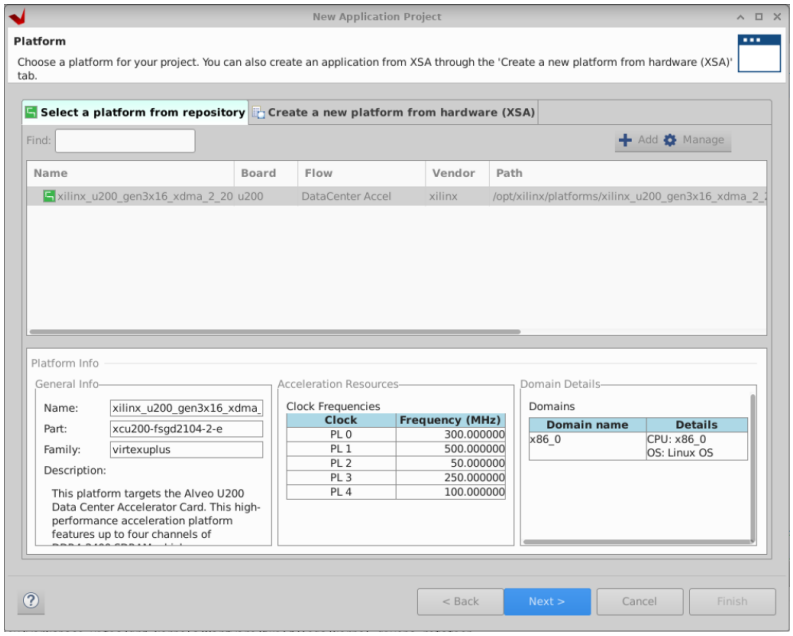

Figura 16: Ventana para escoger la plataforma a utilizar

3. **Nombrar el proyecto**. El siguiente paso es elegir un nombre para el proyecto y pinchar en *Next* para continuar.

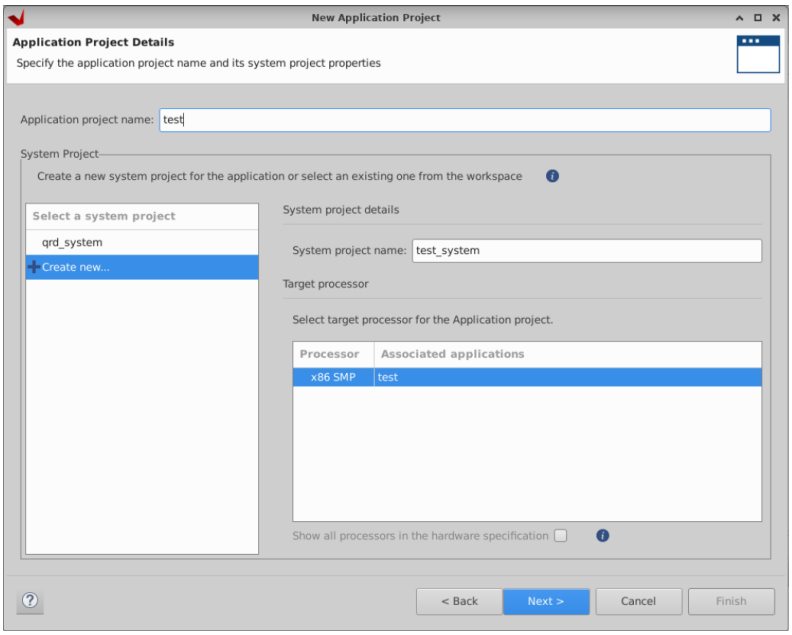

Figura 17: Ventana para elegir un nombre para la aplicación

4. **Elegir una plantilla para utilizar.** En este proyecto no se ha utilizado ninguna plantilla, por lo que se elige la opción *Empty Project*. Una vez seleccionada esta opción, se pincha en *Finish* para terminar con el proceso de creación de un nuevo proyecto.

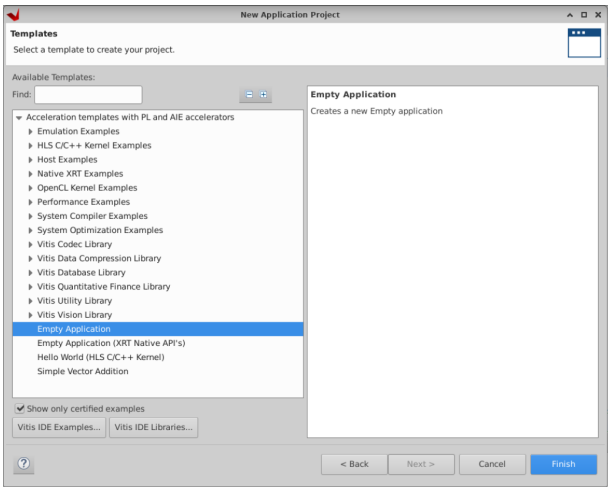

Figura 18: Ventana para elegir una plantilla para iniciar el proyecto

5. **Importar los ficheros del proyecto.** Una vez creado el proyecto, hay que importar los ficheros. El árbol de directorios del proyecto quedaría como en la figura 19.

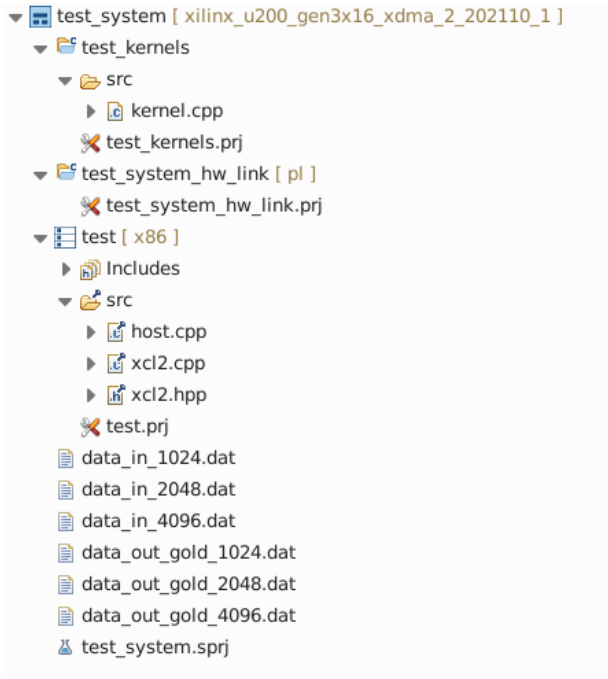

Figura 19: Árbol de directorios del proyecto.

6. **Crear configuración de ejecución.** Es necesario crear una configuración de ejecución y modificarla para indicarle al acelerador los ficheros necesarios como argumentos de entrada. Estos ficheros serán: *data_in_N.dat*, *data_out_gold_N.dat* y **.xclbin* (binario para programar la FPGA). MUY IMPORTANTE introducir el nombre de los ficheros en el mismo orden.

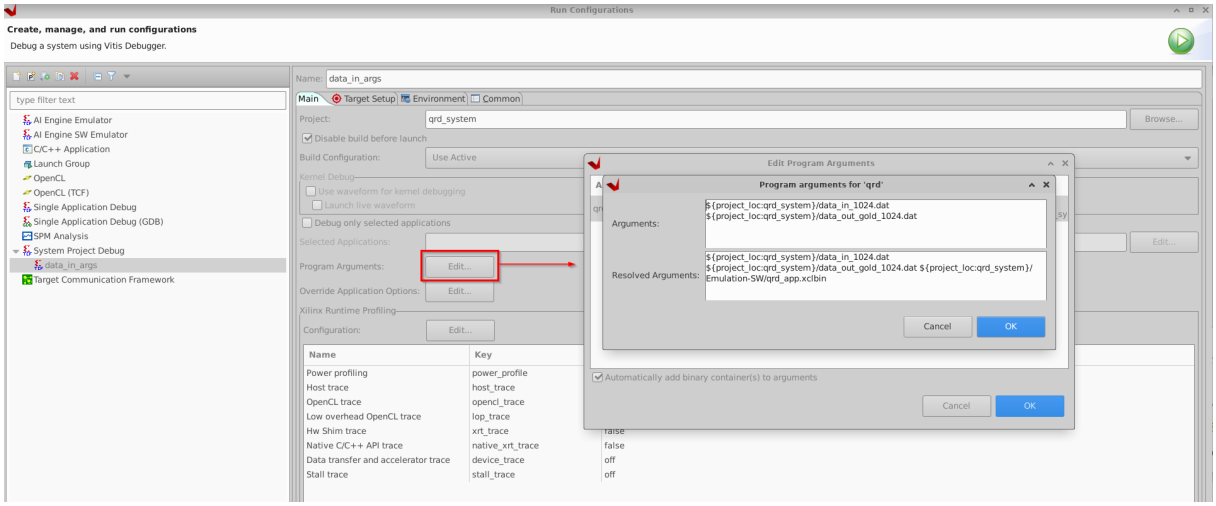

Figura 20: Ventana para editar argumentos de entrada del acelerador.

7. **Elegir tipo de compilación y ejecución.** Por último, se elige el tipo de compilación que se desee: *Emulación Software*, *Emulación Hardware*, *Hardware*. Una vez completada la compilación, se lanza la ejecución del acelerador.

Apéndice B
Ficheros fuente

B.1. Código fuente: host.cpp

```
#include <ap_fixed.h>
#include <ap_int.h>
#include <hls_stream.h>

#include <chrono>
#include <fstream>
#include <iostream>
#include <vector>

#include "xcl2.hpp"

#define FIXED_POINT 32
#define FX_POINT_INT 12

#define TAM_TILED 8
#define TAM 1024
#define NUM_TILED (TAM / TAM_TILED)
#define FLATTEN_SIZE (TAM * TAM_TILED)

#define NUM_OP_GEQRT (TAM / TAM_TILED)
#define NUM_OP_TTQRT ((TAM / TAM_TILED) - 1)
#define NUM_OPERATIONS (NUM_OP_GEQRT + NUM_OP_TTQRT)

#define GEQRT 0
#define TTQRT 1

/**
 * @brief 32 bits fixed point data, 12 for integer value and 26 for decimals.
 * The more bits it has, more shifts can be performed later, so the
 * approximation to 0 will be more precise. For bigger matrices, we need bigger
 * data formats to be able to calculate the right result.
 *
 */
typedef ap_fixed<FIXED_POINT, FX_POINT_INT, AP_RND> data_t;

/**
 * @brief Initialize data_t type matrix with values from input file
 *
 * @param matrix data_t type values
 * @param file input file
 */
bool init_matrix(data_t matrix[TAM][TAM], std::fstream *file);

/**
 * @brief Calculate mean squared error of the result
 *
 * @param A result matrix
 * @param file data gold file
 * @return float error
 */
float mse(data_t A[TAM][TAM], std::fstream *file);
```

```
/**
 * @brief Prior to kernel execution
 *
 * @param qrd_kernel kernel to execute
 * @param q command queue for the device
 * @param context context of the device
 */
void tiled_qr_decomposition(data_t A_tiled[NUM_TILED][TAM_TILED][TAM],
                            cl::Kernel qrd_kernel, cl::CommandQueue q,
                            cl::Context context);

/**
 * @brief Launches the kernel execution
 *
 * @param A_tiled
 * @param type_op
 * @param col_offset
 * @param idx_mat_1
 * @param idx_mat_2
 * @param qrd_kernel
 * @param q
 * @param context
 * @param input_matrix_1_ptr
 * @param input_matrix_2_ptr
 * @param output_matrix_1_ptr
 * @param output_matrix_2_ptr
 * @param flattened_matrix_write_1
 * @param flattened_matrix_write_2
 * @param flattened_matrix_read_1
 * @param flattened_matrix_read_2
 */
void kernel_execute(
    data_t A_tiled[NUM_TILED][TAM_TILED][TAM], uint8_t type_op,
    uint16_t col_offset, uint16_t idx_mat_1, uint16_t idx_mat_2,
    cl::Kernel qrd_kernel, cl::CommandQueue q, cl::Context context,
    cl::Buffer input_matrix_1_ptr, cl::Buffer input_matrix_2_ptr,
    cl::Buffer output_matrix_1_ptr, cl::Buffer output_matrix_2_ptr,
    data_t *flattened_matrix_write_1, data_t *flattened_matrix_write_2,
    data_t *flattened_matrix_read_1, data_t *flattened_matrix_read_2);

/**
 * @brief Writes elements of multidimensional matrix in order into an array
 *
 * @param matrix
 * @param fl_matrix
 * @param idx_mat
 */
void flatten_matrix(data_t matrix[NUM_TILED][TAM_TILED][TAM],
                    data_t fl_matrix[FLATTEN_SIZE], uint16_t idx_mat);

/**
 * @brief Writes elements of result array from kernel in order back into the
 * multidimensional matrix
 *
 * @param matrix
 * @param fl_matrix
 * @param idx_mat
 */
void unflatten_matrix(data_t matrix[NUM_TILED][TAM_TILED][TAM],
                      data_t fl_matrix[FLATTEN_SIZE], uint16_t idx_mat);

// Offset to access the right column for GEQRT operation
```

```
static uint16_t col_offset_geqrt = 0;

// Offset to access the right column for TTQRT operation
static uint16_t col_offset_ttqrt = 0;

// To control the GEQRT operations in each step
static uint16_t n_iter_GEQRT = NUM_OP_GEQRT;

// To control the TTQRT operations in each step
static uint16_t n_iter_TTQRT = NUM_OP_TTQRT;

// Stores all tiled matrices needed for tiled operations
data_t A_tiled[NUM_TILED][TAM_TILED][TAM];

// Stores input matrix
data_t A[TAM][TAM];

int main(int argc, char **argv) {
  if (argc < 2) {
    std::cout << "Usage: " << argv[0] << " <XCLBIN File>" << std::endl;
    return EXIT_FAILURE;
  }

  std::fstream data_in(argv[1], std::ios::in);
  std::fstream data_out_gold(argv[2], std::ios::in);
  // XCLBIN file to program the FPGA
  std::string binaryFile = argv[3];
  cl_int err;
  cl::Context context;
  cl::CommandQueue q;
  cl::Program program;
  cl::Kernel qrd_kernel;

  unsigned int tile_index = 0;
  unsigned int tile_offset = 0;
  float ecm = 0.0;

  // OPENCL HOST CODE AREA START
  auto start1 = std::chrono::high_resolution_clock::now();

  // get_xil_devices() is a utility API which will find the Xilinx
  // platforms and will return list of devices connected to Xilinx platform
  auto devices = xcl::get_xil_devices();

  // read_binary_file() is a utility API which will load the binaryFile
  // and will return the pointer to file buffer.
  auto fileBuf = xcl::read_binary_file(binaryFile);

  cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}};

  bool valid_device = false;

  for (unsigned int i = 0; i < devices.size(); i++) {
    auto device = devices[i];
    // Creating Context and Command Queue for selected Device
    OCL_CHECK(err,
              context = cl::Context(device, nullptr, nullptr, nullptr, &err));
    OCL_CHECK(err, q = cl::CommandQueue(context, device,
                                        CL_QUEUE_PROFILING_ENABLE, &err));

    std::cout << "Trying to program device[" << i
              << "]: " << device.getInfo<CL_DEVICE_NAME>() << std::endl;
    OCL_CHECK(err,
```

```
            program = cl::Program(context, {device}, bins, nullptr, &err));

    if (err != CL_SUCCESS) {
      std::cout << "Failed to program device[" << i << "] with xclbin file!\n";
    } else {
      std::cout << "Device[" << i << "]: program successful!\n";
      OCL_CHECK(err, qrd_kernel =
                         cl::Kernel(program, "kernel_givens_rotation", &err));
      valid_device = true;
      break; // we break because we found a valid device
    }
  }
  if (!valid_device) {
    std::cout << "Failed to program any device found, exit!\n";
    exit(EXIT_FAILURE);
  }

  // Finished creating context, command queue and kernel
  auto end1 = std::chrono::high_resolution_clock::now();
  auto diff1 = end1 - start1;
  std::cout << "FPGA programming time: "
            << std::chrono::duration<double, std::milli>(diff1).count()
            << std::endl;

  if (!init_matrix(A, &data_in))
    return -1;

  for (uint16_t r = 0; r < TAM; r++) {
    tile_index = r / TAM_TILED;
    tile_offset = r % TAM_TILED;

    for (uint16_t c = 0; c < TAM; c++) {
      A_tiled[tile_index][tile_offset][c] = A[r][c];
    }
  }

  // Now, we start to execute the kernel
  auto start = std::chrono::high_resolution_clock::now();
  tiled_qr_decomposition(A_tiled, qrd_kernel, q, context);
  auto end = std::chrono::high_resolution_clock::now();
  auto diff = end - start;

  for (uint16_t r = 0; r < TAM; r++) {
    tile_index = r / TAM_TILED;
    tile_offset = r % TAM_TILED;

    for (uint16_t c = 0; c < TAM; c++) {
      A[r][c] = A_tiled[tile_index][tile_offset][c];
    }
  }

  ecm = mse(A, &data_out_gold);
  std::cout << "Mean Squared Error = " << ecm << std::endl;
  std::cout << "FPGA Execution time: "
            << std::chrono::duration<double, std::milli>(diff).count();

  return 0;
}

void tiled_qr_decomposition(data_t A_tiled[NUM_TILED][TAM_TILED][TAM],
                            cl::Kernel qrd_kernel, cl::CommandQueue q,
                            cl::Context context) {
  cl_int error = 0;
```

```
// Create two kernel input buffers and two output buffer
OCL_CHECK(error, cl::Buffer input_matrix_1_ptr(
                     context,
                     CL_MEM_ALLOC_HOST_PTR | CL_MEM_HOST_WRITE_ONLY |
                         CL_MEM_READ_ONLY,
                     FLATTEN_SIZE * sizeof(data_t), NULL, &error));
OCL_CHECK(error, cl::Buffer input_matrix_2_ptr(
                     context,
                     CL_MEM_ALLOC_HOST_PTR | CL_MEM_HOST_WRITE_ONLY |
                         CL_MEM_READ_ONLY,
                     FLATTEN_SIZE * sizeof(data_t), NULL, &error));
OCL_CHECK(error, cl::Buffer output_matrix_1_ptr(
                     context,
                     CL_MEM_ALLOC_HOST_PTR | CL_MEM_HOST_READ_ONLY |
                         CL_MEM_WRITE_ONLY,
                     FLATTEN_SIZE * sizeof(data_t), NULL, &error));
OCL_CHECK(error, cl::Buffer output_matrix_2_ptr(
                     context,
                     CL_MEM_ALLOC_HOST_PTR | CL_MEM_HOST_READ_ONLY |
                         CL_MEM_WRITE_ONLY,
                     FLATTEN_SIZE * sizeof(data_t), NULL, &error));

// Map input and output buffers
data_t *flattened_matrix_write_1 = (data_t *)q.enqueueMapBuffer(
    input_matrix_1_ptr, CL_TRUE, CL_MAP_WRITE, 0,
    FLATTEN_SIZE * sizeof(data_t), nullptr, nullptr, &error);
data_t *flattened_matrix_write_2 = (data_t *)q.enqueueMapBuffer(
    input_matrix_2_ptr, CL_TRUE, CL_MAP_WRITE, 0,
    FLATTEN_SIZE * sizeof(data_t), nullptr, nullptr, &error);
data_t *flattened_matrix_read_1 = (data_t *)q.enqueueMapBuffer(
    output_matrix_1_ptr, CL_TRUE, CL_MAP_READ, 0,
    FLATTEN_SIZE * sizeof(data_t), nullptr, nullptr, &error);
data_t *flattened_matrix_read_2 = (data_t *)q.enqueueMapBuffer(
    output_matrix_2_ptr, CL_TRUE, CL_MAP_READ, 0,
    FLATTEN_SIZE * sizeof(data_t), nullptr, nullptr, &error);

// Set kernel arguments
OCL_CHECK(error,
          error = qrd_kernel.setArg(0, sizeof(cl_mem), &input_matrix_1_ptr));
OCL_CHECK(error,
          error = qrd_kernel.setArg(1, sizeof(cl_mem), &input_matrix_2_ptr));
OCL_CHECK(error,
          error = qrd_kernel.setArg(2, sizeof(cl_mem), &output_matrix_1_ptr));
OCL_CHECK(error,
          error = qrd_kernel.setArg(3, sizeof(cl_mem), &output_matrix_2_ptr));

for (uint16_t i = 0; i < NUM_OPERATIONS; i++) {
  // GEQRT operation
  if (i % 2 == 0) {
    uint16_t start_idx = (NUM_OP_GEQRT - n_iter_GEQRT);

    for (uint16_t idx_mat_1 = start_idx; idx_mat_1 < NUM_OP_GEQRT;
         idx_mat_1++) {
      kernel_execute(A_tiled, GEQRT, col_offset_geqrt, idx_mat_1, 0,
                     qrd_kernel, q, context, input_matrix_1_ptr,
                     input_matrix_2_ptr, output_matrix_1_ptr,
                     output_matrix_2_ptr, flattened_matrix_write_1,
                     flattened_matrix_write_2, flattened_matrix_read_1,
                     flattened_matrix_read_2);
    }
    n_iter_GEQRT--;
    col_offset_geqrt += TAM_TILED;
```

```
    } else if (i % 2 == 1) {
      uint16_t start_idx_1 = (NUM_OP_TTQRT - n_iter_TTQRT);
      uint16_t start_idx_2 = ((NUM_OP_TTQRT - n_iter_TTQRT) + 1);

      for (uint16_t idx_mat_1 = start_idx_1, idx_mat_2 = start_idx_2;
           idx_mat_2 < NUM_TILED; idx_mat_2++) {
        kernel_execute(A_tiled, TTQRT, col_offset_ttqrt, idx_mat_1, idx_mat_2,
                       qrd_kernel, q, context, input_matrix_1_ptr,
                       input_matrix_2_ptr, output_matrix_1_ptr,
                       output_matrix_2_ptr, flattened_matrix_write_1,
                       flattened_matrix_write_2, flattened_matrix_read_1,
                       flattened_matrix_read_2);
      }
      n_iter_TTQRT--;
      col_offset_ttqrt += TAM_TILED;
    }
  }
}

void kernel_execute(
    data_t A_tiled[NUM_TILED][TAM_TILED][TAM], uint8_t type_op,
    uint16_t col_offset, uint16_t idx_mat_1, uint16_t idx_mat_2,
    cl::Kernel qrd_kernel, cl::CommandQueue q, cl::Context context,
    cl::Buffer input_matrix_1_ptr, cl::Buffer input_matrix_2_ptr,
    cl::Buffer output_matrix_1_ptr, cl::Buffer output_matrix_2_ptr,
    data_t *flattened_matrix_write_1, data_t *flattened_matrix_write_2,
    data_t *flattened_matrix_read_1, data_t *flattened_matrix_read_2) {
  cl_int error;

  OCL_CHECK(error, error = qrd_kernel.setArg(4, type_op));
  OCL_CHECK(error, error = qrd_kernel.setArg(5, col_offset));

  if (type_op == GEQRT) {
    // Flatten input matrix
    flatten_matrix(A_tiled, flattened_matrix_write_1, idx_mat_1);

    // Send input buffer to kernel
    OCL_CHECK(error,
              error = q.enqueueMigrateMemObjects({input_matrix_1_ptr}, 0));

    // Execute kernel
    OCL_CHECK(error, error = q.enqueueTask(qrd_kernel));

    // Wait for kernel to finish
    OCL_CHECK(error, error = q.finish());

    // Write output buffer data back to A_tiled
    OCL_CHECK(error, error = q.enqueueMigrateMemObjects(
                         {output_matrix_1_ptr}, CL_MIGRATE_MEM_OBJECT_HOST));

    // Wait for kernel to finish
    OCL_CHECK(error, error = q.finish());

    unflatten_matrix(A_tiled, flattened_matrix_read_1, idx_mat_1);
  } else if (type_op == TTQRT) {
    // Flatten input matrices
    flatten_matrix(A_tiled, flattened_matrix_write_1, idx_mat_1);
    flatten_matrix(A_tiled, flattened_matrix_write_2, idx_mat_2);

    // Send input buffer to kernel
    OCL_CHECK(error, error = q.enqueueMigrateMemObjects(
                         {input_matrix_1_ptr, input_matrix_2_ptr}, 0));
```

```
    // Execute kernel
    OCL_CHECK(error, error = q.enqueueTask(qrd_kernel));

    // Wait for kernel to finish
    OCL_CHECK(error, error = q.finish());

    // Write output buffer data back to A_tiled
    OCL_CHECK(error, error = q.enqueueMigrateMemObjects(
                         {output_matrix_1_ptr, output_matrix_2_ptr},
                         CL_MIGRATE_MEM_OBJECT_HOST));

    // Wait for kernel to finish
    OCL_CHECK(error, error = q.finish());

    unflatten_matrix(A_tiled, flattened_matrix_read_1, idx_mat_1);
    unflatten_matrix(A_tiled, flattened_matrix_read_2, idx_mat_2);
  }
}

bool init_matrix(data_t matrix[TAM][TAM], std::fstream *file) {
  if (!file->is_open()) {
    std::cerr << "Error opening file" << std::endl;
    return false;
  } else {
    std::cout << "Opened file" << std::endl;
  }

  for (uint16_t r = 0; r < TAM; r++) {
    for (uint16_t c = 0; c < TAM; c++) {
      *file >> matrix[r][c];
    }
  }
  file->close();
  return true;
}

float mse(data_t A[TAM][TAM], std::fstream *file) {
  float err = 0.0;
  data_t resA = 0.0;
  float res_out = 0.0;

  if (!file->is_open()) {
    std::cerr << "Error opening data_out_gold" << std::endl;
  } else {
    std::cout << "Opened file" << std::endl;
  }

  // to access just to the non zero elements (upper triangular matrix)
  for (uint16_t r = 0; r < TAM; r++) {
    for (uint16_t c = 0; c < TAM; c++) {
      resA = A[r][c];
      *file >> res_out;
      err += pow((abs((float)resA) - abs(res_out)), 2);
      res_out = 0;
    }
  }

  // err / number of elements in the upper triangular matrix (including
  // diagonal)
  return err / ((TAM * (TAM + 1)) / 2);
}

void flatten_matrix(data_t matrix[NUM_TILED][TAM_TILED][TAM],
```

```
                    data_t fl_matrix[FLATTEN_SIZE], uint16_t idx_mat) {
  // Flatten the 3D array into 1D array
  for (int j = 0; j < TAM_TILED; j++) {
    for (int k = 0; k < TAM; k++) {
      fl_matrix[j * TAM + k] = matrix[idx_mat][j][k];
    }
  }
}

void unflatten_matrix(data_t matrix[NUM_TILED][TAM_TILED][TAM],
                      data_t fl_matrix[FLATTEN_SIZE], uint16_t idx_mat) {
  // Unflatten the 1D array into 3D matrix
  for (int j = 0; j < TAM_TILED; j++) {
    for (int k = 0; k < TAM; k++) {
      matrix[idx_mat][j][k] = fl_matrix[j * TAM + k];
    }
  }
}
```

Listing 1: Código fuente del Host

B.2. Código fuente: kernel.cpp

```
/**
 * @file kernel.cpp
 * @author Kosta Gecov (kostagecov@gmail.com)
 * @brief Vitis Hls kernel that implements the tiled QRD
 * @version 0.1
 * @date 2023-07-29
 *
 * @copyright Copyright (c) 2023
 *
 */

#include <ap_fixed.h>
#include <ap_int.h>
#include <hls_stream.h>

#include <iostream>

#define FIXED_POINT 32
#define FX_POINT_INT 12

#define TAM_TILED 8
#define TAM 1024

#define N_ITER (FIXED_POINT - 1) // word_lenght - 1

#define GEQRT 0
#define TTQRT 1

/**
 * @brief 32 bits fixed point data, 12 for integer value and 18 for decimals.
 * The more bits it has, more shifts can be performed later, so the
 * approximation to 0 will be more precise. For bigger matrices, we need bigger
 * data formats to be able to calculate the right result.
 *
 */
typedef ap_fixed<FIXED_POINT, FX_POINT_INT, AP_RND> data_t;

// Scale factor to compensate rotations
```

```
const data_t SCALE_FACTOR = 0.607252935008881;

// TRIPCOUNT identifiers
const unsigned int N_ELEM_ROW = TAM;
const unsigned int ITER = N_ITER;
const unsigned int TILED_SIZE = TAM_TILED;

class Rotator {
public:
  // after data is read from an hls::stream<>, it cannot be read again
  // Stream is FIFO type

  hls::stream<data_t, TAM> row_x_in;
  hls::stream<data_t, TAM> row_y_in;
  hls::stream<data_t, TAM> row_x_out;
  hls::stream<data_t, TAM> row_y_out;

  // Position of rows to be rotated
  unsigned int row_x, row_y, col;

public:
  /**
   * @brief Constructor to initialize rotator with positions
   *
   * @param x
   * @param y
   * @param c
   */
  Rotator(unsigned int x, unsigned int y, unsigned int c);

  /**
   * @brief Perform a givens rotation to two input streams and stores the
   * results in other two streams
   *
   * @param row_x_in
   * @param row_y_in
   * @param row_x_out
   * @param row_y_out
   * @param col_rotator From where the rotation should start
   */
  void givens_rotation(hls::stream<data_t, TAM> &row_x_in,
                       hls::stream<data_t, TAM> &row_y_in,
                       hls::stream<data_t, TAM> &row_x_out,
                       hls::stream<data_t, TAM> &row_y_out,
                       uint16_t col_rotator);
};

/**
 * @brief Read input rows using blocking write to streams
 *
 * @param input pointer to the flattened input matrix
 * @param row_in_1
 * @param row_in_2
 * @param row_in_3
 * @param row_in_4
 * @param row_in_5
 * @param row_in_6
 * @param row_in_7
 * @param row_in_8
 */
void read_input_rows(data_t *input, hls::stream<data_t, TAM> &row_in_1,
                     hls::stream<data_t, TAM> &row_in_2,
                     hls::stream<data_t, TAM> &row_in_3,
```

```
                    hls::stream<data_t, TAM> &row_in_4,
                    hls::stream<data_t, TAM> &row_in_5,
                    hls::stream<data_t, TAM> &row_in_6,
                    hls::stream<data_t, TAM> &row_in_7,
                    hls::stream<data_t, TAM> &row_in_8);

/**
 * @brief Choose the right sign for the rotation,
 * taking into account the coordinates' quadrants
 *
 * @param x row x
 * @param y row y
 * @param col column offset
 */
void update_quadrant(data_t x[TAM], data_t y[TAM], uint16_t col);

/**
 * @brief Store stream data into arrays
 *
 * @param row_in_1 stream input
 * @param row_in_2 stream input
 * @param x array output
 * @param y array output
 */
void read_input_data(hls::stream<data_t, TAM> &row_in_1,
                    hls::stream<data_t, TAM> &row_in_2, data_t x[TAM],
                    data_t y[TAM]);

/**
 * @brief Performs cordic rotation of two rows
 *
 * @param x row x
 * @param y row y
 * @param x_aux auxiliar array to store previous value of row x
 * @param sign to choose the sign of the operation
 * @param n_iter number of iterations
 * @param col column offset
 */
void cordic(data_t x[TAM], data_t y[TAM], data_t x_aux[TAM], bool sign,
            uint8_t n_iter, uint16_t col);

/**
 * @brief Multiply row values with a constant scale factor
 *
 * @param x row x
 * @param y  row y
 * @param col column offset
 */
void compensate_scale_factor(data_t x[TAM], data_t y[TAM], uint16_t col);

/**
 * @brief Store array data into streams
 *
 * @param row_out_1 stream row output
 * @param row_out_2 stream row output
 * @param x row input
 * @param y row input
 */
void write_output_data(hls::stream<data_t, TAM> &row_out_1,
                       hls::stream<data_t, TAM> &row_out_2, data_t x[TAM],
                       data_t y[TAM]);

/**
```

```
 * @brief Write output rows using blocking read to streams
 *
 * @param output
 * @param row_out_1
 * @param row_out_2
 * @param row_out_3
 * @param row_out_4
 * @param row_out_5
 * @param row_out_6
 * @param row_out_7
 * @param row_out_8
 */
void write_output_rows(data_t *output, hls::stream<data_t, TAM> &row_out_1,
                       hls::stream<data_t, TAM> &row_out_2,
                       hls::stream<data_t, TAM> &row_out_3,
                       hls::stream<data_t, TAM> &row_out_4,
                       hls::stream<data_t, TAM> &row_out_5,
                       hls::stream<data_t, TAM> &row_out_6,
                       hls::stream<data_t, TAM> &row_out_7,
                       hls::stream<data_t, TAM> &row_out_8);

/**
 * @brief GE operation for givens rotation kernel
 *
 * @param input_tile_1
 * @param output_tile_1
 * @param col_offset
 */
void kernel_givens_rotation_GE(data_t *input_tile_1, data_t *output_tile_1,
                               uint16_t col_offset);

/**
 * @brief TT operation for givens rotation kernel
 *
 * @param input_tile_1
 * @param input_tile_2
 * @param output_tile_1
 * @param output_tile_2
 * @param col_offset
 */
void kernel_givens_rotation_TT(data_t *input_tile_1, data_t *input_tile_2,
                               data_t *output_tile_1, data_t *output_tile_2,
                               uint16_t col_offset);

extern "C" {
/**
 * @brief Performs the givens rotation of the tiles. It is the top function
 *
 * @param A_tile A_tiled matrix
 * @param type_op It can be GEQRT or TTQRT
 * @param col_offset Offset used to avoid reading the positions that had already
 * become 0
 * @param idx_mat_1 To access the right A and Q matrices
 * @param idx_mat_2 To access the right A and Q matrices. In case the operation
 * is GEQRT, this value is 0
 */
void kernel_givens_rotation(data_t *input_tile_1, data_t *input_tile_2,
                            data_t *output_tile_1, data_t *output_tile_2,
                            uint8_t type_op, uint16_t col_offset);
}

Rotator::Rotator(unsigned int x, unsigned int y, unsigned int c) {
#pragma HLS INLINE off
```

```
  // actually, row_x and row_y are not used, they have an indicative role
  // while declaring rotators objects
  Rotator::row_x = x;
  Rotator::row_y = y;
  Rotator::col = c;
}

void read_row(data_t *input, hls::stream<data_t, TAM> &row, uint16_t offset) {
#pragma HLS INLINE off
read_row_for:
  for (uint16_t j = 0; j < TAM; j++) {
#pragma HLS LOOP_TRIPCOUNT avg = N_ELEM_ROW max = N_ELEM_ROW min = N_ELEM_ROW
    row.write(input[j + TAM * offset]);
  }
}

void read_input_rows(data_t *input, hls::stream<data_t, TAM> &row_in_1,
                     hls::stream<data_t, TAM> &row_in_2,
                     hls::stream<data_t, TAM> &row_in_3,
                     hls::stream<data_t, TAM> &row_in_4,
                     hls::stream<data_t, TAM> &row_in_5,
                     hls::stream<data_t, TAM> &row_in_6,
                     hls::stream<data_t, TAM> &row_in_7,
                     hls::stream<data_t, TAM> &row_in_8) {
#pragma HLS INLINE off
  read_row(input, row_in_1, 0);
  read_row(input, row_in_2, 1);
  read_row(input, row_in_3, 2);
  read_row(input, row_in_4, 3);
  read_row(input, row_in_5, 4);
  read_row(input, row_in_6, 5);
  read_row(input, row_in_7, 6);
  read_row(input, row_in_8, 7);
}

void read_input_data(hls::stream<data_t, TAM> &row_in_1,
                     hls::stream<data_t, TAM> &row_in_2, data_t x[TAM],
                     data_t y[TAM]) {
#pragma HLS INLINE off
read_input_data:
  for (uint16_t j = 0; j < TAM; j++) {
#pragma HLS LOOP_TRIPCOUNT avg = N_ELEM_ROW max = N_ELEM_ROW min = N_ELEM_ROW
#pragma HLS PIPELINE II = 1
    row_in_1.read(x[j]);
    row_in_2.read(y[j]);
  }
}

void update_quadrant(data_t x[TAM], data_t y[TAM], uint16_t col) {
#pragma HLS INLINE off
sign_for:
  for (uint16_t s = col; s < TAM; s++) {
#pragma HLS LOOP_TRIPCOUNT max = N_ELEM_ROW min = TILED_SIZE
#pragma HLS PIPELINE
    x[s] = -x[s];
    y[s] = -y[s];
  }
}

void cordic(data_t x[TAM], data_t y[TAM], data_t x_aux[TAM], bool sign,
            uint8_t n_iter, uint16_t col) {
#pragma HLS INLINE off
  signed char sign_factor_x = 0;
```

```
  signed char sign_factor_y = 0;

  if (sign) {
    sign_factor_x = -1;
    sign_factor_y = 1;
  } else {
    sign_factor_x = 1;
    sign_factor_y = -1;
  }

aux_var_for:
  for (uint16_t i = col; i < TAM; i++) {
#pragma HLS LOOP_TRIPCOUNT max = N_ELEM_ROW min = N_ELEM_ROW
#pragma HLS PIPELINE II = 1
    x_aux[i] = x[i];
  }

// If Y is negative, we need to add to it so that it gets closer to zero
// and to the contrary with X coordinate
column_rotation_for:
  for (uint16_t j = col; j < TAM; j++) {
#pragma HLS LOOP_TRIPCOUNT max = N_ELEM_ROW min = TILED_SIZE
#pragma HLS PIPELINE
    x[j] = x[j] + (sign_factor_x * (y[j] >> n_iter));
    y[j] = y[j] + (sign_factor_y * (x_aux[j] >> n_iter));
  }
}

void compensate_scale_factor(data_t x[TAM], data_t y[TAM], uint16_t col) {
#pragma HLS INLINE off
scale_factor_for:
  for (uint16_t j = 0; j < TAM; j++) {
#pragma HLS LOOP_TRIPCOUNT max = N_ELEM_ROW min = TILED_SIZE
#pragma HLS PIPELINE
    x[j] = x[j] * SCALE_FACTOR;
    y[j] = y[j] * SCALE_FACTOR;
  }
}

void write_output_data(hls::stream<data_t, TAM> &row_out_1,
                       hls::stream<data_t, TAM> &row_out_2, data_t x[TAM],
                       data_t y[TAM]) {
#pragma HLS INLINE off
write_output_data:
  for (uint16_t j = 0; j < TAM; j++) {
#pragma HLS LOOP_TRIPCOUNT max = N_ELEM_ROW min = N_ELEM_ROW
#pragma HLS PIPELINE II = 1
    row_out_1.write(x[j]);
    row_out_2.write(y[j]);
  }
}

void Rotator::givens_rotation(hls::stream<data_t, TAM> &row_x_in,
                              hls::stream<data_t, TAM> &row_y_in,
                              hls::stream<data_t, TAM> &row_x_out,
                              hls::stream<data_t, TAM> &row_y_out,
                              uint16_t col_rotator) {
#pragma HLS INLINE off
  bool sign = false;
  uint16_t i = 0, j = 0, k = 0, s = 0;
  data_t x[TAM] = {0}, y[TAM] = {0}, x_aux[TAM] = {0};

#pragma HLS BIND_STORAGE variable = x type = RAM_S2P impl = BRAM
```

```
#pragma HLS BIND_STORAGE variable = y type = RAM_S2P impl = BRAM
#pragma HLS BIND_STORAGE variable = x_aux type = RAM_1P impl = BRAM

  read_input_data(row_x_in, row_y_in, x, y);

  if (x[col_rotator] < 0) {
    update_quadrant(x, y, col_rotator);
  }

iterations_for:
  for (k = 0; k < N_ITER; k++) {
#pragma HLS LOOP_TRIPCOUNT max = ITER min = ITER
#pragma HLS PIPELINE off
    sign = (y[col_rotator] < 0);

    cordic(x, y, x_aux, sign, k, col_rotator);
  }

  if ((y[col_rotator] < 0.001) && (y[col_rotator] > -0.001)) {
    y[col_rotator] = 0;
  }

  compensate_scale_factor(x, y, col_rotator);

  write_output_data(row_x_out, row_y_out, x, y);
}

void write_row(data_t *output, hls::stream<data_t, TAM> &row, uint16_t offset) {
#pragma HLS INLINE off
write_row_for:
  for (uint16_t j = 0; j < TAM; j++) {
#pragma HLS LOOP_TRIPCOUNT avg = N_ELEM_ROW max = N_ELEM_ROW min = N_ELEM_ROW
    row.read(output[j + TAM * offset]);
  }
}

void write_output_rows(data_t *output, hls::stream<data_t, TAM> &row_out_1,
                       hls::stream<data_t, TAM> &row_out_2,
                       hls::stream<data_t, TAM> &row_out_3,
                       hls::stream<data_t, TAM> &row_out_4,
                       hls::stream<data_t, TAM> &row_out_5,
                       hls::stream<data_t, TAM> &row_out_6,
                       hls::stream<data_t, TAM> &row_out_7,
                       hls::stream<data_t, TAM> &row_out_8) {
#pragma HLS INLINE off
  write_row(output, row_out_1, 0);
  write_row(output, row_out_2, 1);
  write_row(output, row_out_3, 2);
  write_row(output, row_out_4, 3);
  write_row(output, row_out_5, 4);
  write_row(output, row_out_6, 5);
  write_row(output, row_out_7, 6);
  write_row(output, row_out_8, 7);
}

void kernel_givens_rotation_GE(data_t *input_tile_1, data_t *output_tile_1,
                               uint16_t col_offset) {
  // Rotators for GEQRT operation
  Rotator Rot1_GE(0, 1, 0);
  Rotator Rot2_GE(2, 3, 0);
  Rotator Rot3_GE(4, 5, 0);
  Rotator Rot4_GE(6, 7, 0);
```

```
Rotator Rot5_GE(0, 2, 0);
Rotator Rot6_GE(4, 6, 0);
Rotator Rot7_GE(1, 3, 1);
Rotator Rot8_GE(5, 7, 1);

Rotator Rot9_GE(0, 4, 0);
Rotator Rot10_GE(1, 2, 1);
Rotator Rot11_GE(3, 7, 2);
Rotator Rot12_GE(5, 6, 1);

Rotator Rot13_GE(2, 3, 2);
Rotator Rot14_GE(4, 5, 1);

Rotator Rot15_GE(1, 4, 1);
Rotator Rot16_GE(2, 5, 2);
Rotator Rot17_GE(3, 7, 3);

Rotator Rot18_GE(3, 5, 3);
Rotator Rot19_GE(4, 6, 2);

Rotator Rot20_GE(2, 4, 2);
Rotator Rot21_GE(3, 6, 3);
Rotator Rot22_GE(5, 7, 4);

Rotator Rot23_GE(3, 4, 3);
Rotator Rot24_GE(5, 6, 4);

Rotator Rot25_GE(4, 5, 4);
Rotator Rot26_GE(6, 7, 5);

Rotator Rot27_GE(5, 6, 5);
Rotator Rot28_GE(6, 7, 6);

// Variables needed to avoid DATAFLOW warning
uint16_t column1 = Rot1_GE.col + col_offset;
uint16_t column2 = Rot2_GE.col + col_offset;
uint16_t column3 = Rot3_GE.col + col_offset;
uint16_t column4 = Rot4_GE.col + col_offset;
uint16_t column5 = Rot5_GE.col + col_offset;
uint16_t column6 = Rot6_GE.col + col_offset;
uint16_t column7 = Rot7_GE.col + col_offset;
uint16_t column8 = Rot8_GE.col + col_offset;
uint16_t column9 = Rot9_GE.col + col_offset;
uint16_t column10 = Rot10_GE.col + col_offset;
uint16_t column11 = Rot11_GE.col + col_offset;
uint16_t column12 = Rot12_GE.col + col_offset;
uint16_t column13 = Rot13_GE.col + col_offset;
uint16_t column14 = Rot14_GE.col + col_offset;
uint16_t column15 = Rot15_GE.col + col_offset;
uint16_t column16 = Rot16_GE.col + col_offset;
uint16_t column17 = Rot17_GE.col + col_offset;
uint16_t column18 = Rot18_GE.col + col_offset;
uint16_t column19 = Rot19_GE.col + col_offset;
uint16_t column20 = Rot20_GE.col + col_offset;
uint16_t column21 = Rot21_GE.col + col_offset;
uint16_t column22 = Rot22_GE.col + col_offset;
uint16_t column23 = Rot23_GE.col + col_offset;
uint16_t column24 = Rot24_GE.col + col_offset;
uint16_t column25 = Rot25_GE.col + col_offset;
uint16_t column26 = Rot26_GE.col + col_offset;
uint16_t column27 = Rot27_GE.col + col_offset;
uint16_t column28 = Rot28_GE.col + col_offset;
```

```
#pragma HLS DATAFLOW
  read_input_rows(input_tile_1, Rot1_GE.row_x_in, Rot1_GE.row_y_in,
                  Rot2_GE.row_x_in, Rot2_GE.row_y_in, Rot3_GE.row_x_in,
                  Rot3_GE.row_y_in, Rot4_GE.row_x_in, Rot4_GE.row_y_in);

  Rot1_GE.givens_rotation(Rot1_GE.row_x_in, Rot1_GE.row_y_in, Rot1_GE.row_x_out,
                          Rot1_GE.row_y_out, column1);

  Rot2_GE.givens_rotation(Rot2_GE.row_x_in, Rot2_GE.row_y_in, Rot2_GE.row_x_out,
                          Rot2_GE.row_y_out, column2);

  Rot3_GE.givens_rotation(Rot3_GE.row_x_in, Rot3_GE.row_y_in, Rot3_GE.row_x_out,
                          Rot3_GE.row_y_out, column3);

  Rot4_GE.givens_rotation(Rot4_GE.row_x_in, Rot4_GE.row_y_in, Rot4_GE.row_x_out,
                          Rot4_GE.row_y_out, column4);

  Rot5_GE.givens_rotation(Rot1_GE.row_x_out, Rot2_GE.row_x_out,
                          Rot5_GE.row_x_out, Rot5_GE.row_y_out, column5);

  Rot6_GE.givens_rotation(Rot3_GE.row_x_out, Rot4_GE.row_x_out,
                          Rot6_GE.row_x_out, Rot6_GE.row_y_out, column6);

  Rot7_GE.givens_rotation(Rot1_GE.row_y_out, Rot2_GE.row_y_out,
                          Rot7_GE.row_x_out, Rot7_GE.row_y_out, column7);

  Rot8_GE.givens_rotation(Rot3_GE.row_y_out, Rot4_GE.row_y_out,
                          Rot8_GE.row_x_out, Rot8_GE.row_y_out, column8);

  Rot9_GE.givens_rotation(Rot5_GE.row_x_out, Rot6_GE.row_x_out,
                          Rot9_GE.row_x_out, Rot9_GE.row_y_out, column9);

  Rot10_GE.givens_rotation(Rot7_GE.row_x_out, Rot5_GE.row_y_out,
                           Rot10_GE.row_x_out, Rot10_GE.row_y_out, column10);

  Rot11_GE.givens_rotation(Rot7_GE.row_y_out, Rot8_GE.row_y_out,
                           Rot11_GE.row_x_out, Rot11_GE.row_y_out, column11);

  Rot12_GE.givens_rotation(Rot8_GE.row_x_out, Rot6_GE.row_y_out,
                           Rot12_GE.row_x_out, Rot12_GE.row_y_out, column12);

  Rot13_GE.givens_rotation(Rot10_GE.row_y_out, Rot11_GE.row_x_out,
                           Rot13_GE.row_x_out, Rot13_GE.row_y_out, column13);

  Rot14_GE.givens_rotation(Rot9_GE.row_y_out, Rot12_GE.row_x_out,
                           Rot14_GE.row_x_out, Rot14_GE.row_y_out, column14);

  Rot15_GE.givens_rotation(Rot10_GE.row_x_out, Rot14_GE.row_x_out,
                           Rot15_GE.row_x_out, Rot15_GE.row_y_out, column15);

  Rot16_GE.givens_rotation(Rot13_GE.row_x_out, Rot14_GE.row_y_out,
                           Rot16_GE.row_x_out, Rot16_GE.row_y_out, column16);

  Rot17_GE.givens_rotation(Rot13_GE.row_y_out, Rot11_GE.row_y_out,
                           Rot17_GE.row_x_out, Rot17_GE.row_y_out, column17);

  Rot18_GE.givens_rotation(Rot17_GE.row_x_out, Rot16_GE.row_y_out,
                           Rot18_GE.row_x_out, Rot18_GE.row_y_out, column18);

  Rot19_GE.givens_rotation(Rot15_GE.row_y_out, Rot12_GE.row_y_out,
                           Rot19_GE.row_x_out, Rot19_GE.row_y_out, column19);

  Rot20_GE.givens_rotation(Rot16_GE.row_x_out, Rot19_GE.row_x_out,
```

```
                          Rot20_GE.row_x_out, Rot20_GE.row_y_out, column20);

  Rot21_GE.givens_rotation(Rot18_GE.row_x_out, Rot19_GE.row_y_out,
                          Rot21_GE.row_x_out, Rot21_GE.row_y_out, column21);

  Rot22_GE.givens_rotation(Rot18_GE.row_y_out, Rot17_GE.row_y_out,
                          Rot22_GE.row_x_out, Rot22_GE.row_y_out, column22);

  Rot23_GE.givens_rotation(Rot21_GE.row_x_out, Rot20_GE.row_y_out,
                          Rot23_GE.row_x_out, Rot23_GE.row_y_out, column23);

  Rot24_GE.givens_rotation(Rot22_GE.row_x_out, Rot21_GE.row_y_out,
                          Rot24_GE.row_x_out, Rot24_GE.row_y_out, column24);

  Rot25_GE.givens_rotation(Rot23_GE.row_y_out, Rot24_GE.row_x_out,
                          Rot25_GE.row_x_out, Rot25_GE.row_y_out, column25);

  Rot26_GE.givens_rotation(Rot24_GE.row_y_out, Rot22_GE.row_y_out,
                          Rot26_GE.row_x_out, Rot26_GE.row_y_out, column26);

  Rot27_GE.givens_rotation(Rot25_GE.row_y_out, Rot26_GE.row_x_out,
                          Rot27_GE.row_x_out, Rot27_GE.row_y_out, column27);

  Rot28_GE.givens_rotation(Rot27_GE.row_y_out, Rot26_GE.row_y_out,
                          Rot28_GE.row_x_out, Rot28_GE.row_y_out, column28);

  write_output_rows(output_tile_1, Rot9_GE.row_x_out, Rot15_GE.row_x_out,
                    Rot20_GE.row_x_out, Rot23_GE.row_x_out, Rot25_GE.row_x_out,
                    Rot27_GE.row_x_out, Rot28_GE.row_x_out, Rot28_GE.row_y_out);
}

void kernel_givens_rotation_TT(data_t *input_tile_1, data_t *input_tile_2,
                               data_t *output_tile_1, data_t *output_tile_2,
                               uint16_t col_offset) {
  // Rotators for TTQRT operation
  Rotator Rot1_TT(0, 0, 0);
  Rotator Rot2_TT(1, 1, 1);
  Rotator Rot3_TT(2, 2, 2);
  Rotator Rot4_TT(3, 3, 3);
  Rotator Rot5_TT(4, 4, 4);
  Rotator Rot6_TT(5, 5, 5);
  Rotator Rot7_TT(6, 6, 6);
  Rotator Rot8_TT(7, 7, 7);

  Rotator Rot9_TT(1, 0, 1);
  Rotator Rot10_TT(2, 1, 2);
  Rotator Rot11_TT(3, 2, 3);
  Rotator Rot12_TT(4, 3, 4);
  Rotator Rot13_TT(5, 4, 5);
  Rotator Rot14_TT(6, 5, 6);
  Rotator Rot15_TT(7, 6, 7);

  Rotator Rot16_TT(2, 0, 2);
  Rotator Rot17_TT(3, 1, 3);
  Rotator Rot18_TT(4, 2, 4);
  Rotator Rot19_TT(5, 3, 5);
  Rotator Rot20_TT(6, 4, 6);
  Rotator Rot21_TT(7, 5, 7);

  Rotator Rot22_TT(3, 0, 3);
  Rotator Rot23_TT(4, 1, 4);
  Rotator Rot24_TT(5, 2, 5);
  Rotator Rot25_TT(6, 3, 6);
```

```
  Rotator Rot26_TT(7, 4, 7);

  Rotator Rot27_TT(4, 0, 4);
  Rotator Rot28_TT(5, 1, 5);
  Rotator Rot29_TT(6, 2, 6);
  Rotator Rot30_TT(7, 3, 7);

  Rotator Rot31_TT(5, 0, 5);
  Rotator Rot32_TT(6, 1, 6);
  Rotator Rot33_TT(7, 2, 7);

  Rotator Rot34_TT(6, 0, 6);
  Rotator Rot35_TT(7, 1, 7);

  Rotator Rot36_TT(7, 0, 7);

  // Variables needed to avoid DATAFLOW warning
  uint16_t column1 = Rot1_TT.col + col_offset;
  uint16_t column2 = Rot2_TT.col + col_offset;
  uint16_t column3 = Rot3_TT.col + col_offset;
  uint16_t column4 = Rot4_TT.col + col_offset;
  uint16_t column5 = Rot5_TT.col + col_offset;
  uint16_t column6 = Rot6_TT.col + col_offset;
  uint16_t column7 = Rot7_TT.col + col_offset;
  uint16_t column8 = Rot8_TT.col + col_offset;
  uint16_t column9 = Rot9_TT.col + col_offset;
  uint16_t column10 = Rot10_TT.col + col_offset;
  uint16_t column11 = Rot11_TT.col + col_offset;
  uint16_t column12 = Rot12_TT.col + col_offset;
  uint16_t column13 = Rot13_TT.col + col_offset;
  uint16_t column14 = Rot14_TT.col + col_offset;
  uint16_t column15 = Rot15_TT.col + col_offset;
  uint16_t column16 = Rot16_TT.col + col_offset;
  uint16_t column17 = Rot17_TT.col + col_offset;
  uint16_t column18 = Rot18_TT.col + col_offset;
  uint16_t column19 = Rot19_TT.col + col_offset;
  uint16_t column20 = Rot20_TT.col + col_offset;
  uint16_t column21 = Rot21_TT.col + col_offset;
  uint16_t column22 = Rot22_TT.col + col_offset;
  uint16_t column23 = Rot23_TT.col + col_offset;
  uint16_t column24 = Rot24_TT.col + col_offset;
  uint16_t column25 = Rot25_TT.col + col_offset;
  uint16_t column26 = Rot26_TT.col + col_offset;
  uint16_t column27 = Rot27_TT.col + col_offset;
  uint16_t column28 = Rot28_TT.col + col_offset;
  uint16_t column29 = Rot29_TT.col + col_offset;
  uint16_t column30 = Rot30_TT.col + col_offset;
  uint16_t column31 = Rot31_TT.col + col_offset;
  uint16_t column32 = Rot32_TT.col + col_offset;
  uint16_t column33 = Rot33_TT.col + col_offset;
  uint16_t column34 = Rot34_TT.col + col_offset;
  uint16_t column35 = Rot35_TT.col + col_offset;
  uint16_t column36 = Rot36_TT.col + col_offset;

#pragma HLS DATAFLOW
  // Read X coordinates rows from first matrix
  read_input_rows(input_tile_1, Rot1_TT.row_x_in, Rot2_TT.row_x_in,
                  Rot3_TT.row_x_in, Rot4_TT.row_x_in, Rot5_TT.row_x_in,
                  Rot6_TT.row_x_in, Rot7_TT.row_x_in, Rot8_TT.row_x_in);

  // Read Y coordinates rows from second matrix
  read_input_rows(input_tile_2, Rot1_TT.row_y_in, Rot2_TT.row_y_in,
                  Rot3_TT.row_y_in, Rot4_TT.row_y_in, Rot5_TT.row_y_in,
```

```
                Rot6_TT.row_y_in, Rot7_TT.row_y_in, Rot8_TT.row_y_in);

Rot1_TT.givens_rotation(Rot1_TT.row_x_in, Rot1_TT.row_y_in, Rot1_TT.row_x_out,
                        Rot1_TT.row_y_out, column1);

Rot2_TT.givens_rotation(Rot2_TT.row_x_in, Rot2_TT.row_y_in, Rot2_TT.row_x_out,
                        Rot2_TT.row_y_out, column2);

Rot3_TT.givens_rotation(Rot3_TT.row_x_in, Rot3_TT.row_y_in, Rot3_TT.row_x_out,
                        Rot3_TT.row_y_out, column3);

Rot4_TT.givens_rotation(Rot4_TT.row_x_in, Rot4_TT.row_y_in, Rot4_TT.row_x_out,
                        Rot4_TT.row_y_out, column4);

Rot5_TT.givens_rotation(Rot5_TT.row_x_in, Rot5_TT.row_y_in, Rot5_TT.row_x_out,
                        Rot5_TT.row_y_out, column5);

Rot6_TT.givens_rotation(Rot6_TT.row_x_in, Rot6_TT.row_y_in, Rot6_TT.row_x_out,
                        Rot6_TT.row_y_out, column6);

Rot7_TT.givens_rotation(Rot7_TT.row_x_in, Rot7_TT.row_y_in, Rot7_TT.row_x_out,
                        Rot7_TT.row_y_out, column7);

Rot8_TT.givens_rotation(Rot8_TT.row_x_in, Rot8_TT.row_y_in, Rot8_TT.row_x_out,
                        Rot8_TT.row_y_out, column8);

Rot9_TT.givens_rotation(Rot2_TT.row_x_out, Rot1_TT.row_y_out,
                        Rot9_TT.row_x_out, Rot9_TT.row_y_out, column9);

Rot10_TT.givens_rotation(Rot3_TT.row_x_out, Rot2_TT.row_y_out,
                         Rot10_TT.row_x_out, Rot10_TT.row_y_out, column10);

Rot11_TT.givens_rotation(Rot4_TT.row_x_out, Rot3_TT.row_y_out,
                         Rot11_TT.row_x_out, Rot11_TT.row_y_out, column11);

Rot12_TT.givens_rotation(Rot5_TT.row_x_out, Rot4_TT.row_y_out,
                         Rot12_TT.row_x_out, Rot12_TT.row_y_out, column12);

Rot13_TT.givens_rotation(Rot6_TT.row_x_out, Rot5_TT.row_y_out,
                         Rot13_TT.row_x_out, Rot13_TT.row_y_out, column13);

Rot14_TT.givens_rotation(Rot7_TT.row_x_out, Rot6_TT.row_y_out,
                         Rot14_TT.row_x_out, Rot14_TT.row_y_out, column14);

Rot15_TT.givens_rotation(Rot8_TT.row_x_out, Rot7_TT.row_y_out,
                         Rot15_TT.row_x_out, Rot15_TT.row_y_out, column15);

Rot16_TT.givens_rotation(Rot10_TT.row_x_out, Rot9_TT.row_y_out,
                         Rot16_TT.row_x_out, Rot16_TT.row_y_out, column16);

Rot17_TT.givens_rotation(Rot11_TT.row_x_out, Rot10_TT.row_y_out,
                         Rot17_TT.row_x_out, Rot17_TT.row_y_out, column17);

Rot18_TT.givens_rotation(Rot12_TT.row_x_out, Rot11_TT.row_y_out,
                         Rot18_TT.row_x_out, Rot18_TT.row_y_out, column18);

Rot19_TT.givens_rotation(Rot13_TT.row_x_out, Rot12_TT.row_y_out,
                         Rot19_TT.row_x_out, Rot19_TT.row_y_out, column19);

Rot20_TT.givens_rotation(Rot14_TT.row_x_out, Rot13_TT.row_y_out,
                         Rot20_TT.row_x_out, Rot20_TT.row_y_out, column20);

Rot21_TT.givens_rotation(Rot15_TT.row_x_out, Rot14_TT.row_y_out,
```

```
                          Rot21_TT.row_x_out, Rot21_TT.row_y_out, column21);

  Rot22_TT.givens_rotation(Rot17_TT.row_x_out, Rot16_TT.row_y_out,
                          Rot22_TT.row_x_out, Rot22_TT.row_y_out, column22);

  Rot23_TT.givens_rotation(Rot18_TT.row_x_out, Rot17_TT.row_y_out,
                          Rot23_TT.row_x_out, Rot23_TT.row_y_out, column23);

  Rot24_TT.givens_rotation(Rot19_TT.row_x_out, Rot18_TT.row_y_out,
                          Rot24_TT.row_x_out, Rot24_TT.row_y_out, column24);

  Rot25_TT.givens_rotation(Rot20_TT.row_x_out, Rot19_TT.row_y_out,
                          Rot25_TT.row_x_out, Rot25_TT.row_y_out, column25);

  Rot26_TT.givens_rotation(Rot21_TT.row_x_out, Rot20_TT.row_y_out,
                          Rot26_TT.row_x_out, Rot26_TT.row_y_out, column26);

  Rot27_TT.givens_rotation(Rot23_TT.row_x_out, Rot22_TT.row_y_out,
                          Rot27_TT.row_x_out, Rot27_TT.row_y_out, column27);

  Rot28_TT.givens_rotation(Rot24_TT.row_x_out, Rot23_TT.row_y_out,
                          Rot28_TT.row_x_out, Rot28_TT.row_y_out, column28);

  Rot29_TT.givens_rotation(Rot25_TT.row_x_out, Rot24_TT.row_y_out,
                          Rot29_TT.row_x_out, Rot29_TT.row_y_out, column29);

  Rot30_TT.givens_rotation(Rot26_TT.row_x_out, Rot25_TT.row_y_out,
                          Rot30_TT.row_x_out, Rot30_TT.row_y_out, column30);

  Rot31_TT.givens_rotation(Rot28_TT.row_x_out, Rot27_TT.row_y_out,
                          Rot31_TT.row_x_out, Rot31_TT.row_y_out, column31);

  Rot32_TT.givens_rotation(Rot29_TT.row_x_out, Rot28_TT.row_y_out,
                          Rot32_TT.row_x_out, Rot32_TT.row_y_out, column32);

  Rot33_TT.givens_rotation(Rot30_TT.row_x_out, Rot29_TT.row_y_out,
                          Rot33_TT.row_x_out, Rot33_TT.row_y_out, column33);

  Rot34_TT.givens_rotation(Rot32_TT.row_x_out, Rot31_TT.row_y_out,
                          Rot34_TT.row_x_out, Rot34_TT.row_y_out, column34);

  Rot35_TT.givens_rotation(Rot33_TT.row_x_out, Rot32_TT.row_y_out,
                          Rot35_TT.row_x_out, Rot35_TT.row_y_out, column35);

  Rot36_TT.givens_rotation(Rot35_TT.row_x_out, Rot34_TT.row_y_out,
                          Rot36_TT.row_x_out, Rot36_TT.row_y_out, column36);

  write_output_rows(output_tile_1, Rot1_TT.row_x_out, Rot9_TT.row_x_out,
                    Rot16_TT.row_x_out, Rot22_TT.row_x_out, Rot27_TT.row_x_out,
                    Rot31_TT.row_x_out, Rot34_TT.row_x_out, Rot36_TT.row_x_out);

  write_output_rows(output_tile_2, Rot36_TT.row_y_out, Rot35_TT.row_y_out,
                    Rot33_TT.row_y_out, Rot30_TT.row_y_out, Rot26_TT.row_y_out,
                    Rot21_TT.row_y_out, Rot15_TT.row_y_out, Rot8_TT.row_y_out);
}

extern "C" {
void kernel_givens_rotation(data_t *input_tile_1, data_t *input_tile_2,
                            data_t *output_tile_1, data_t *output_tile_2,
                            uint8_t type_op, uint16_t col_offset) {
#pragma HLS INTERFACE mode = ap_ctrl_chain port = return
#pragma HLS INTERFACE mode = m_axi bundle = amem0 port = input_tile_1 offset = \
    slave
```

```
#pragma HLS INTERFACE mode = m_axi bundle = amem1 port = input_tile_2 offset = \
    slave
#pragma HLS INTERFACE mode = m_axi bundle = amem2 port =                       \
    output_tile_1 offset = slave
#pragma HLS INTERFACE mode = m_axi bundle = amem3 port =                       \
    output_tile_2 offset = slave
#pragma HLS INTERFACE mode = s_axilite port = type_op
#pragma HLS INTERFACE mode = s_axilite port = col_offset

  if (type_op == GEQRT) {
    kernel_givens_rotation_GE(input_tile_1, output_tile_1, col_offset);
  } else if (type_op == TTQRT) {
    kernel_givens_rotation_TT(input_tile_1, input_tile_2, output_tile_1,
                              output_tile_2, col_offset);
  }
}
}
```

Listing 2: Código fuente del Kernel

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA