UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Implementación eficiente de núcleos computacionales básicos de álgebra lineal numérica en el procesador Wafer-Scale Engine

Trabajo Fin de Máster

Máster Universitario en Computación en la Nube y de Altas Prestaciones / Cloud and High-Performance Computing

AUTOR/A: Sandoval Moreno, Abdel

Tutor/a: Alonso Jordá, Pedro

Cotutor/a externo: Carmo Boratto, Murilo Do

CURSO ACADÉMICO: 2023/2024

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

VALENCIA – ESPAÑA

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

IMPLEMENTACIÓN EFICIENTE DE NÚCLEOS COMPUTACIONALES BÁSICOS DE ÁLGEBRA LINEAL NÚMERICA EN EL PROCESADOR WAFER-SCALE ENGINE

ABDEL SANDOVAL MORENO

TRABAJO DE FIN DE MÁSTER PARA OPTAR AL TÍTULO DE
MÁSTER UNIVERSITARIO EN COMPUTACIÓN EN LA NUBE Y DE ALTAS PRESTACIONES

TUTOR: PEDRO ALONSO

TUTOR EXTERNO: MURILO BORATTO

SEPTIEMBRE 2024

UNIVERSITAT POLITÈCNICA DE VALÈNCIA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN
VALENCIA – ESPAÑA

IMPLEMENTACIÓN EFICIENTE DE NÚCLEOS COMPUTACIONALES BÁSICOS DE ÁLGEBRA LINEAL NÚMERICA EN EL PROCESADOR WAFER-SCALE ENGINE

ABDEL SANDOVAL MORENO

TRABAJO DE FIN DE MÁSTER PARA OPTAR AL TÍTULO DE MÁSTER UNIVERSITARIO EN COMPUTACIÓN EN LA NUBE Y DE ALTAS PRESTACIONES

TUTOR: PEDRO ALONSO
TUTOR EXTERNO: MURILO BORATTO

SEPTIEMBRE 2024

Resumen

La cantidad de datos que se deben procesar ha ido en aumento año tras año. La opción más destacada para enfrentar este problema ha sido la computación paralela. En este contexto ha surgido recientemente la arquitectura Wafer-Scale Engine, que cuenta con cerca de 850,000 núcleos de procesamiento, la cual, aunque ha sido diseñada para la IA, diversas investigaciones han mostrado su potencial en otras áreas.

El foco de este trabajo es la implementación en esta arquitectura de núcleos computacionales básicos de álgebra lineal numérica, tales como la descomposición LU, la descomposición QR y la multiplicación de matrices y, con esto, crear una base para la resolución de problemas en diversas áreas, como la geociencia, grafos, redes, entre otras. Este trabajo realiza la propuesta e implementación de los algoritmos para los núcleos computacionales mencionados en la arquitectura y lleva a cabo pruebas de rendimiento de los mismos.

Los resultados muestran que la carga comunicacional tiene un alto impacto sobre la tendencia de crecimiento del tiempo de ejecución, teniendo que a menor carga comunicacional, el tiempo de ejecución tiene una tendencia lineal, mientras que a mayor carga la tendencia es exponencial. Adicionalmente, se muestra que usar más núcleos con menos datos cada uno obtiene un mejor rendimiento que usar pocos núcleos más poblados.

Finalmente, la experiencia de este trabajo entrega bases sobre las que construir nuevos algoritmos, ya sea mejorando o haciendo uso de lo desarrollado aquí, o bien, implementando otros algoritmos haciendo uso de los consejos de desarrollo propuestos.

Palabras clave: Wafer-Scale Engine, descomposición LU, descomposición QR, algoritmo de Cannon.

Resum

La quantitat de dades que s'han de processar han anat en augment any rere any. L'opció més destacada per a enfrontar este problema ha sigut la computació paral·lela. En este context ha sorgit recentment l'arquitectura Wafer-Scale Engine, que compta amb prop de 850,000 nuclis de processament, la qual, encara que ha sigut dissenyada per a la IA, diverses investigacions han mostrat el seu potencial en altres àrees.

El focus d'este treball és la implementació en esta arquitectura de nuclis computacionals bàsics d'àlgebra lineal numèrica, com ara la descomposició LU, la descomposició QR i la multiplicació de matrius i, amb això, crear una base per a la resolució de problemes en diverses àrees, com la geociència, grafs, xarxes, entre altres. Este treball realitza la proposta i implementació dels algorismes per als nuclis computacionals esmentats en l'arquitectura i duu a terme proves de rendiment d'estos.

Els resultats mostren que la càrrega comunicacional té un alt impacte sobre la tendència de creixement del temps d'execució, tenint que a menor càrrega comunicacional, el temps d'execució té una tendència lineal, mentres que a major càrrega la tendència és exponencial. Addicionalment, es mostra que usar més nuclis amb menys dades cadascun obté un millor rendiment que usar pocs nuclis més poblats.

Finalment, l'experiència d'este treball entrega bases sobre les quals construir nous algorismes, ja siga millorant o fent ús del desenrotllat ací, o bé, implementant altres algorismes fent ús dels consells de desenrotllament proposats.

Paraules clau: Wafer-Scale Engine, descomposició LU, descomposició QR, algorisme de Cannon.

Abstract

The amount of data to be processed has been increasing year after year. The most prominent option to address this problem has been parallel computing. In this context, the Wafer-Scale Engine architecture has recently emerged, which has about 850,000 processing cores, which, although it has been designed for AI, various investigations have shown its potential in other areas.

The focus of this work is the implementation in this architecture of basic computational cores of numerical linear algebra, such as LU decomposition, QR decomposition and matrix multiplication, and with this, create a basis for solving problems in various areas, such as geoscience, graphs, networks, among others. This work proposes and implements the algorithms for the computational cores mentioned in the architecture and carries out performance tests of them.

The results show that the communication load has a high impact on the growth trend of the execution time, with a lower communication load having a linear trend in the execution time, while a higher load has an exponential trend. Additionally, it is shown that using more cores with less data each obtains better performance than using fewer, more populated cores.

Finally, the experience of this work provides bases on which to build new algorithms, either by improving or making use of what was developed here, or by implementing other algorithms making use of the proposed development tips.

Key words: Wafer-Scale Engine, LU decomposition, QR decomposition, Cannon's algorithm.

Índice de Contenidos

Índice de Tablas

Índice de Figuras

Capítulo 1

Introducción

La cantidad de datos que necesitan ser procesados en distintos sectores, tanto científicos como comerciales, ha ido en aumento año tras año, haciendo necesario adoptar diversas formas de cómputo para enfrentar el creciente volumen de datos. La opción que más fuerza ha tomado y, probablemente, la más vigente en la actualidad es el uso de tecnologías de computación paralela. Es posible encontrar distintas formas de computo paralelo, cada una con sus ventajas y desventajas, algunas de ellas son múltiples procesos en entornos de memoria compartida, múltiples procesos en entornos de memoria distribuida, procesadores con instrucciones vectoriales, arquitecturas masivamente paralelas, entre otras. Es posible encontrar también entornos que manejan una combinación de las distintas formas de computación paralela, buscando aprovechar lo mejor de cada una.

Entre las opciones de arquitecturas masivamente paralelas, la más conocida y utilizada son las GPUs, diseñadas originalmente para el procesamiento de gráficos pero, con el tiempo, se vio su potencial para el procesamiento de grandes cantidades de datos, ganando terreno hasta ser la más popular del mercado. Siguiendo la línea de las arquitecturas masivamente paralelas, en los recientes años surgió la arquitectura Wafer-Scale Engine (WSE) [6] de la compañía Cerebras Systems Inc. que está diseñada originalmente para tareas de aprendizaje profundo e inferencia y cuenta con cerca de $850,000$ cores distribuidos en una malla bidimensional, cerca de dos órdenes de magnitud más que las GPUs actuales, por ejemplo, una

GPU NVIDIA Blackwell puede contener hasta 24, 576 CUDA cores. Algunas de sus características principales son que provee comunicación eficiente entre cores adyacentes, siendo capaz de enviar 32 bits de información en un ciclo de reloj, además, cada core tiene memoria independiente situada cerca de la unidad de cómputo, lo que permite un rápido acceso a ella. Dadas estas y otras características de la arquitectura, aunque su diseño haya sido pensado para IA, es posible extender su uso a otras áreas, por ejemplo, la modelización climática, simulaciones sísmicas avanzadas, técnicas de imagen, entre otras. Para lograr esta extensión, parte del trabajo consiste en la implementación de núcleos computaciones básicos de álgebra lineal, como lo son las descomposiciones matriciales y la multiplicación de matrices.

El presente trabajo busca implementar núcleos básicos de álgebra lineal en la arquitectura WSE, en específico, la multiplicación de matrices, la descomposición LU y la descomposición QR, dada sus aplicaciones en distintos problemas y campos, tales como grafos, redes, computación gráfica, teoría de la probabilidad, sistemas de ecuaciones lineales, simulación numérica, procesamiento de señales, problemas de mínimos cuadrados, problemas de autovalores, compresión de imágenes, entre muchos otros. Algunas de estas aplicaciones incluso son parte de aplicaciones más grandes, de ahí su importancia no sólo como métodos de aplicación directa, si no como elementos de otros métodos más grandes que hacen uso de estos en algún punto. Dado que esta arquitectura presenta un comportamiento distinto a otras y define su propio lenguaje de programación, es necesario también llevar a cabo el aprendizaje de este nuevo lenguaje y comprender la arquitectura.

1.1. Motivación

Dado que la arquitectura WSE [6] está originalmente pensada para IA y el hecho de que es reciente, la exploración en usos fuera de la IA no es un tema muy tratado, sin embargo dadas sus capacidades y propiedades, es interesante explorar su rendimiento en algoritmos que no puedan explotar del todo las propiedades de arquitecturas más utilizadas actualmente, o bien, se puedan ver beneficiados de las propiedades de esta arquitectura.

La arquitectura WSE ya ha mostrado su potencial en soluciones fuera de la IA, así como

ocurrió con las GPUs que originalmente diseñadas para el procesamiento de gráficos y hoy en día están presentes en una gran cantidad de superordenadores, estos resultados son una muestra de que es posible lograr avances a través de esta arquitectura en campos para los que no fue diseñada.

Si nos enfocamos en aplicaciones que se han visto beneficiadas por la capacidad de paralelismo que ofrece la arquitectura WSE, en "Disruptive Changes in Field Equation Modeling" [10] se utiliza para crear una API que resuelve ecuaciones de campo, superando en rendimiento otros métodos de computación distribuida utilizados en el área por cerca de dos ordenes de magnitud. Siguiendo la misma línea "Efficient Algorithms for Monte Carlo Particle Transport on AI Accelerator Hardware" [9] implementa una solución en la arquitectura WSE al problema de transporte de partículas de Monte Carlo, donde se hace uso de estrategias de balanceo de carga para distribuir de manera más uniforme los datos, logrando un rendimiento 130 veces superior en comparación a la GPU NVIDIA A100. De la misma manera, en "Wafer-Scale Fast Fourier Transforms" [8] se implementan la transformada rápida de Fourier, obteniedo un tiempo de 959 [μs] para un problema complejo de tamaño 512^3, siendo la paralelización mas grande realizada para este tamaño de problema y la primera vez que se rompe la barrera del milisegundo.

Con todo lo anterior en consideración, se puede pensar que la ventaja de la arquitectura WSE se encuentra sólo en su capacidad para lograr paralelizaciones a gran escala, sin embargo, otros trabajos muestran que una característica importante de esta arquitectura es también un acceso eficiente de memoria y rápida comunicación entre sus componentes, así lo detalla "Scaling the 'Memory Wall' for Multi-Dimensional Seismic Processing with Algebraic Compression on Cerebras CS-2 Systems" [7], que hace uso de operadores de convolución multidimensional para crear imágenes de alta resolución del subsuelo, donde a través del uso de técnicas para aprovechar las características de las matrices dispersas, logran un ancho de banda de 92.85 [PB/s] utilizando 48 WSEs, quedando un orden de magnitud por debajo de la supercomputadora Frontier que tiene el primer lugar en la lista TOP 500 [1]. Por otra parte, en "Massively scalable stencil algorithm" [5] se implementa en la arquitectura WSE un algoritmo de stencil para resolver problemas numéricos relacionados con la ecuación de

onda tridimensional, lo que es exigente en accesos a memoria debido a que hay una baja reutilización de los datos. Este algoritmo históricamente ha estado limitado por memoria, pero los autores logran una implementación limitada por la capacidad de procesamiento, logrando un escalado débil casi perfecto y obteniendo un rendimiento hasta 229 veces mejor que la GPU NVIDIA A100.

Es claro que la arquitectura WSE tiene el potencial de traer mejoras en áreas donde la naturaleza de los algoritmos y problemas a resolver se ajusten de mejor manera a la estructura que esta ofrece, por lo que este trabajo busca aprovechar estas cualidades para explorar en áreas donde aún no se ha intentado utilizar esta arquitectura.

1.2. Objetivos

1.2.1. Objetivo General

- Desarrollar, analizar y validar núcleos computacionales básicos de álgebra lineal en la arquitectura Wafer-Scale Engine.

1.2.2. Objetivos Específicos

- Analizar los algoritmos de multiplicación de matrices, descomposición LU y descomposición QR para seleccionar aquellos que se adapten mejor a la arquitectura WSE.
- Proponer la implementación de estos algoritmos considerando los patrones de comunicación y cómputo de la WSE.
- Realizar la implementación de los algoritmos propuestos y evaluar su desempeño utilizando las métricas de tiempo de ejecución y flops por ciclo de reloj.

1.3. Estructura del documento

Este documento cuenta con 5 capítulos. En el Capítulo 2 se presentan las definiciones y los algoritmos de los núcleos de álgebra lineal junto con los detalles de la arquitectura WSE. En el Capítulo 3 se detalla la implementación de los distintos algoritmos sobre la WSE. En el Capítulo 4 se muestran las pruebas realizadas y se analizan los resultados obtenidos de las mismas. Finalmente, en el capítulo 5 se exponen las conclusiones y se discute sobre el trabajo futuro.

Capítulo 2

Estado del Arte

2.1. Multiplicación de matrices

Dadas dos matrices A y B, tales que $A \in \mathbb{R}^{m \times k}$ y $B \in \mathbb{R}^{k \times n}$, con a_{ij} el elemento en la fila i, columna j de la matriz A y b_{ij} el elemento en la fila i, columna j de la matriz B, el resultado de su multiplicación es una matriz $C \in \mathbb{R}^{m \times n}$, cuyo elemento (i, j) está dado por la Ecuación 2.1.

$$c_{ij} = \sum_{r=1}^{k} a_{ir} b_{rj} \tag{2.1}$$

2.1.1. Algoritmo de Cannon

El algoritmo de Cannon [3] es un algoritmo de multiplicación de matrices pensado para entornos paralelos que subdivide las matrices A y B en p^2 bloques cuadrados. Los procesos son etiquetados desde $P_{0,0}$ hasta $P_{p-1,p-1}$, asignándole inicialmente las submatrices $A_{i,j}$ y $B_{i,j}$ al proceso $P_{i,j}$. Para realizar la multiplicación de matrices, todos los procesos de la i-ésima fila requieren las p submatrices $A_{i,k}$ con $(0 \leq k < p)$, de la misma manera, todos los procesos de la i-ésima columna necesitan las p submatrices $B_{k,j}$ con $(0 \leq k < p)$. Es posible organizar

el computo y comunicación de los p^2 procesos para que, en cualquier momento dado, cada proceso esté usando una submatriz $A_{i,k}$ y $B_{k,j}$ distinta y cada proceso $P_{i,j}$ se encargue del computo de la submatriz $C_{i,j}$. Los pasos del algoritmo son los siguientes:

1. Las submatrices de A en la i-ésima fila se desplazan i lugares hacia la izquierda, los desbordamientos se tratan como si la fila fuese un anillo.

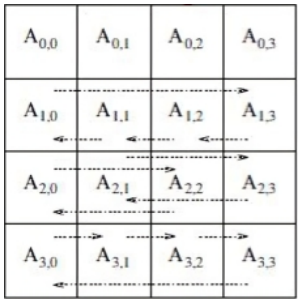
2. Las submatrices de B en la i-ésima columna se desplazan i lugares hacia arriba, los desbordamientos se tratan como si la columna fuese un anillo.

3. Se inicializa la matriz C en 0 en todas sus posiciones (o a un valor inicial dado).

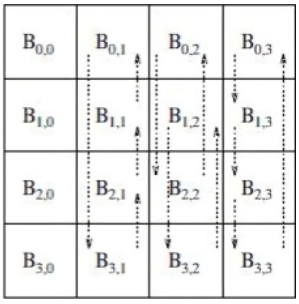
4. Se suma a la matriz C el resultado de la multiplicación de las submatrices que están en la posición de cada proceso, luego se transmiten las submatrices de A una posición hacia la izquierda y las submatrices de B una posición hacia arriba, haciendo el mismo tratamiento de anillo para los desbordamientos. Este paso se repite $p - 1$ veces.

5. Se realiza la última suma a la matriz C con el resultado de la multiplicación de cada proceso.

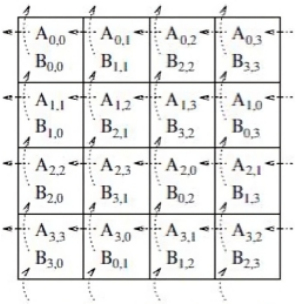
La Figura 2.1 ilustra cómo se mueven las submatrices durante todo el proceso para 16 procesos. Considerando las matrices A y B ambas $\in \mathbb{R}^{n \times n}$, una multiplicación de matrices requiere $2n^3$ flops.



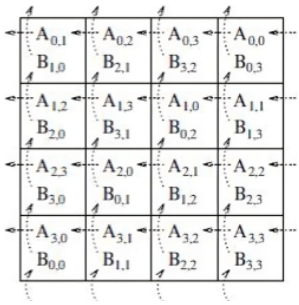
(a) Alineación inicial de A.



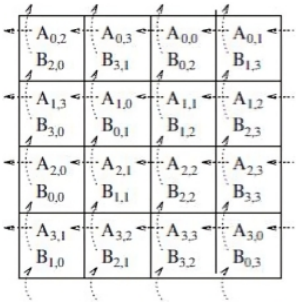
(b) Alineación inicial de B.



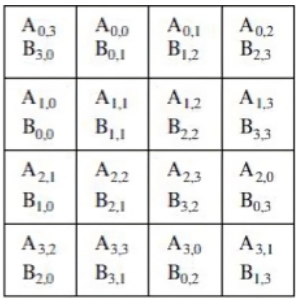
(c) A y B luego de su alineación inicial.



(d) Posición de las submatrices luego del primer cambio de posiciones.



(e) Posición de las submatrices luego del segundo cambio de posiciones.



(f) Posición de las submatrices luego del tercer cambio de posiciones.

Figura 2.1: Pasos de comunicación en el algoritmo de Cannon para 16 procesos.

2.2. Descomposición LU

Sea una matriz A invertible, con a_{ij} su elemento en la fila i, columna j, la descomposición LU [2] de A cumple que $A = LU$, donde L es una matriz triangular inferior con l_{ij} su elemento en la fila i, columna j, y U es una matriz triangular superior con u_{ij} su elemento en la fila i, columna j. En la Ecuación 2.2 se puede ver un ejemplo para una matriz de 3×3.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix} \tag{2.2}$$

El algoritmo de la descomposición LU [3] puede crear dos nuevas matrices para almacenar

el resultado, o bien, almacenar el resultado en el mismo espacio donde se encuentra la matriz original A. Esto es posible debido a que la diagonal de la matriz L son unos, por lo que este espacio puede ser utilizado por la diagonal de la matriz U. El algoritmo para realizar esto se detalla en el Algoritmo 1, el cual requiere $2n^3/3$ flops.

Algoritmo 1 Descomposición LU

```
1:  function DESCOMPOSICIÓN_LU(A)
2:      for <it = 0 To n - 1> do
3:          for <row = it + 1 To n - 1> do                    ▷ Paso de división
4:              A[row, it] = A[row, it]/A[it, it];
5:          end for
6:          for <row = it + 1 To n - 1> do
7:              for <col = it + 1 To n - 1> do                ▷ Paso de eliminación
8:                  A[row, col] = A[row, col] - A[it, col] * A[row, it];
9:              end for
10:         end for
11:     end for
12: end function
```

Esta descomposición es útil para la resolución de sistemas de álgebra lineal, encontrar la matriz inversa o el cálculo del determinante de una matriz. Existe otra versión llamada PA=LU, donde la matriz P es una matriz de permutación, que no requiere que la matriz A sea estrictamente invertible y es más precisa que la descomposición LU, sin embargo esta versión se encuentra fuera del alcance de este trabajo.

2.2.1. Versión paralela de tipo pipeline

La descomposición LU puede ser implementada en entornos paralelos de diferentes maneras. Una de estas formas de paralelismo consiste en realizar un pipeline [3]. Si se tienen p^2 procesos y una matriz $A \in \mathbb{R}^{n\times n}$, tal que $n/p \in \mathbb{N}$, es posible dividir la matriz A en p^2 submatrices cuadradas de n^2/p^2 elementos, asignando una submatriz distinta a cada proceso. Con la distribución de datos anterior, es posible llevar a cabo el Algoritmo 1. Para esto, es necesario que en cada proceso siga el siguiente comportamiento:

1. Si un proceso tiene datos destinados a otro proceso, realiza el envío de esos datos al proceso apropiado. Esto envía los valores necesarios para que cada proceso pueda realizar los pasos de división y eliminación.

2. Si un proceso puede calcular parte de la descomposición con los datos que posee, lo hace. Aquí se realizan los pasos de división y eliminación según corresponda.

3. En otro caso, el proceso espera a recibir datos a ser utilizados en alguno de los casos anteriores.

Con esto, un proceso no necesita esperar que otro termine la iteración k para avanzar a la iteración $k + 1$ mientras tenga los datos necesarios para ello.

2.3. Rotaciones de Givens

Las rotaciones de Givens [2] son correcciones de rango-2 a la identidad. La Ecuación 2.3 muestra una rotación de Givens para un θ dado, donde $c = \cos\theta$ y $s = \sin\theta$. Multiplicar a la izquierda por $G(i, k, \theta)^T$ es equivalente a una rotación en sentido antihorario de θ radianes en el plano de coordenadas (i, k), esta rotación es ortogonal.

Las rotaciones son computacionalmente atractivas dado que es fácil construirlas y pueden ser utilizadas para introducir ceros en un vector si se elige el ángulo de rotación adecuadamente.

$$G(i,k,\theta) = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \begin{matrix} \\ \\ i \\ \\ k \\ \\ \\ \end{matrix} \qquad (2.3)$$
$$\begin{matrix} & & i & & k & & \end{matrix}$$

2.4. Descomposición QR

Sea una matriz rectangular $A \in \mathbb{R}^{m \times n}$, esta puede ser expresada como el producto de una matriz ortogonal $Q \in \mathbb{R}^{m \times m}$ y una matriz triangular superior $R \in \mathbb{R}^{m \times n}$ tal que $A = QR$. Esta descomposición es conocida como la descomposición QR [2] y es útil para la resolución de sistemas de álgebra lineal, problemas de mínimos cuadrados, problemas de autovalores, entre otros.

En varias aplicaciones el cálculo de la matriz Q puede ser omitido, lo que es preferible pues esta es computacionalmente costosa de calcular, por lo que hay versiones que sólo calculan la matriz R, o bien, la matriz R y algún elemento adicional menos costoso que Q.

Existen diferentes métodos para computar esta descomposición, como el método de ortogonalización de Gram-Schmidt, las reflexiones de Householder o, el elegido para este trabajo y uno de los más recomendados, el de las rotaciones de Givens [2], ya mencionado en la Sección 2.3. El algoritmo para calcular una rotación de Givens entre dos elementos tal que, al ser aplicada, uno de ellos se vuelva 0, se muestra en el Algoritmo 2.

Algoritmo 2 Rotación de Givens

```
1:  function GIVENS(a, b)
2:      if b == 0 then
3:          c = 1; s = 0;
4:      else
5:          if |b| > |a| then
6:              τ = -a/b; s = 1/√(1+τ²); c = s·τ;
7:          else
8:              τ = -b/a; c = 1/√(1+τ²); s = c·τ;
9:          end if
10:     end if
11: end function
```

La forma de utilizar las rotaciones de Givens para calcular la matriz R se detalla en el Algoritmo 3, el cual requiere $3n^2(m - n/3)$ flops para una matriz $A \in \mathbb{R}^{m \times n}$. Este algoritmo hace los ceros desde la primera columna hacia la derecha, partiendo desde la última fila hacia arriba en cada columna, sin embargo estos se pueden hacer en diferente orden. Es importante mencionar que aunque el algoritmo trabaja siempre con filas adyacentes, es posible hacer estas operaciones con cualquier par de filas siempre que tengan la misma cantidad de ceros

a la izquierda.

Algoritmo 3 Descomposición QR

```
1: function GIVENSQR(A)
2:     for <j = 1 To n - 1> do
3:         for i = m - 1 To j do
4:             [c,s] = givens(A[i-1, j], A[i, j]);
5:             (A[i-1, j], A[i, j]) = (A[i-1, j]*c - A[i, j]*s,
6:                                     A[i-1, j]*s + A[i, j]*c);
7:         end for
8:     end for
9: end function
```

2.5. Arquitectura Wafer-Scale Engine

Una tendencia notable en la industria de procesadores es la adopción de chips diseñados para aplicaciones específicas, conocidos como ASIC (Circuitos Integrados para Aplicaciones Específicas). La arquitectura Wafer-Scale Engine [6] es uno de ellos, el cual integra cerca de 850,000 núcleos de procesamiento, conocidos como elementos de procesamiento independientes (PEs), en un solo chip. Los PEs están interconectados por canales de comunicación en una malla rectangular de dos dimensiones en una única oblea de silicio. Cada PE tiene su propia memoria (que no comparte con ningún otro PE) y su propio contador de programa.

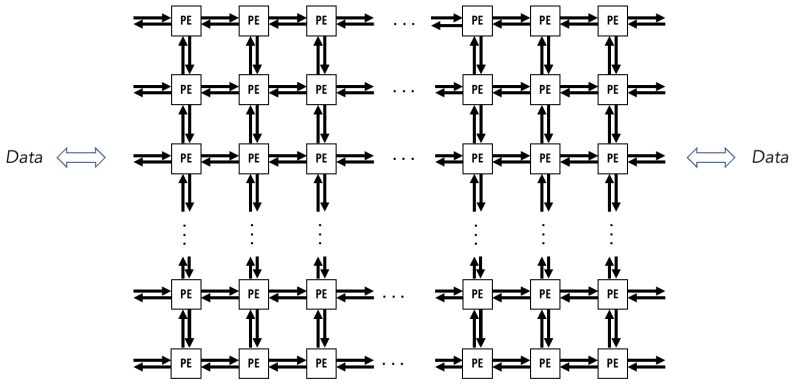


Figura 2.2: Representación visual de la malla de PEs que componen una WSE.

El Sistema Cerebras (CS) es un sistema autocontenido montado en un rack que contiene empaquetado, suministro de energía, enfriamiento e I/O para una WSE. El CS se comunica

a través de conexiones Ethernet paralelas de 100 Gigabit (host I/O) a un cluster CPU host. La Figura 2.2 muestra una representación visual de la malla de PEs que componen la WSE. Los datos son transmitidos al CS a través del host I/O y entra a la WSE a través de una serie de conexiones a lo largo de sus bordes.

2.5.1. Elementos de procesamiento

Un PE contiene tres elementos claves:

1. Un procesador, también conocido como motor de computo (CE).
2. Un router, el que está directamente conectado mediante conexiones bidireccionales con su propio CE y a los routers de los cuatro PEs vecinos más cercanos en la malla. La conexión con su propio CE se llama RAMP, mientras que las conexiones con los vecinos son llamadas según sus direcciones cardinales. Este componente es el único que los PEs utilizan para enviar y recibir datos.
3. La memoria local del PE, con una capacidad de 48 kB, todos los datos y código de un PE es almacenado en esta memoria. Ni el CE ni la memoria local de un PE es accesible por otros PEs.

Estos elementos son ilustrados visualmente en la Figura 2.3.

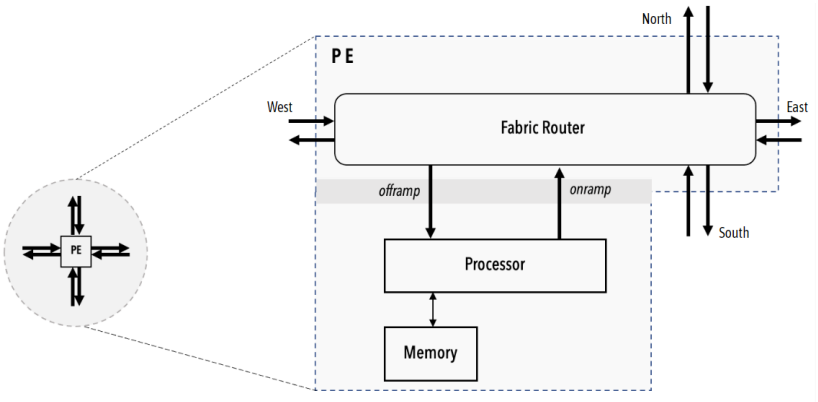


Figura 2.3: Representación visual de un PE.

Cuando se hace referencia a un PE específico en la malla, este se nombra como PE(x,y), donde x es la columna a la que pertenece e y es la fila a la que pertenece, en ambos casos partiendo desde 0, teniendo el origen de las coordenadas en la esquina superior izquierda.

2.5.2. El modelo de programación

Para desarrollar código para una WSE el lenguaje utilizado es el Cerebras Software Language (CSL) [4], y el código del host en Python. El código para el CS es compilado y luego ejecutado en un CS conectado a la red, o bien, en un simulador de la arquitectura. El código del host se encarga de copiar datos desde y hacia el CS, así como de iniciar la ejecución de programas llamados kernels.

Un código completo para un CS está compuesto de 3 partes:

- Código del host: Código Python que se encarga de enviar/recibir datos hacia el CS, interactuar con elementos externos si la información inicial se encontrase en, por ejemplo, una base de datos, así como formatear y ordenar los datos de manera que puedan ser enviados correctamente hacia el CS. También podrá iniciar la ejecución de kernels. Para todo esto se dispone de un SDK que se encuentra en la sección "SdkRuntime API Reference" de la documentación [4].
- Layout: Código CSL el cual define cuantos PEs serán utilizados y su disposición, el código que será asignado a cada PE (distintos PEs pueden tener distinto código a ejecutar), los parámetros de cada PE en caso de ser necesarios, así como la configuración de las comunicaciones de cada PE. Adicionalmente, puede exportar variables para ser utilizadas por el host, como las variables sobre las que se enviarán/recibiran datos, o los kernels que el host podrá iniciar su ejecución.
- Código de los PEs: Uno o más códigos CSL que indicarán el comportamiento del PE al que son asignados, en estos se definen variables, funciones, tareas, las variables o kernels a ser exportados, entre otros.

Programas y tareas

Un programa CSL consiste en uno o más subprogramas, algunos de ellos son funciones invocables y otros son tareas. Una tarea es un procedimiento que no puede ser llamado desde el código, si no que son iniciadas por el hardware del PE, se ejecutan hasta completarse y, en ese punto, el hardware elige una nueva tarea que ejecutar. Las tareas pueden ser activadas, no retornan valores y están ligadas a un identificador (ID) para identificarlas, el cual es un valor numérico entre 0 y 63.

Existen 3 tipos de tareas:

- Tareas de datos: Su ID está asociada a un elemento que transmite información hacia el PE, el cual varía según la arquitectura WSE utilizada. En la arquitectura WSE-2 se asocia a un canal de comunicación, mientras que en la arquitectura WSE-3 se asocia a una cola de entrada, la que a su vez está asociada a un canal de comunicación. Una cola de entrada es un buffer a nivel de hardware donde los datos son almacenados temporalmente antes de entrar a un CE. Estas se activan al recibir un mensaje a través del elemento al que están asociado. Deben tener al menos un argumento, el que representa el mensaje recibido, en caso de tener más de un argumento el mensaje se divide en partes iguales entre los argumentos.
- Tareas locales: Su ID está ligada a un identificador activable, estas no reciben parámetros y se pueden activar manualmente desde el código del PE.
- Tareas de control: Su ID está ligada a una de las IDs en el rango 0 a 63 y, similar a las tareas de datos, estas sólo se ejecutan cuando el PE recibe un mensaje de control que contiene su ID en el payload.

Todas las tareas mencionadas pueden ser bloqueadas y desbloqueadas a través del código de un PE, para que una tarea pueda ser añadida al programa de ejecución debe estar desbloqueada en el momento que se cumpla el requisito de su activación. Todas las tareas comienzan desbloqueadas por defecto. El Código 2.1 muestra el aspecto que tiene una tare de datos en el lenguaje CSL.

```
task recv_x(x_val: f32) void {
  @fmacs(y_dsd, y_dsd, A_dsd, x_val);
  A_dsd = @increment_dsd_offset(A_dsd, M_per_PE, f32);
  num_recv_x += 1;
  if (num_recv_x == N_per_PE) {
    @activate(reduce_task_id);
  }
}
```

Código 2.1: Ejemplo de una tarea de datos en el lenguaje CSL.

2.5.3. Comunicaciones

Comunicación entre PEs

Existen 24 canales de comunicación virtual usados por el hardware para enviar datos entre los PEs. Estos canales son llamados colores enrutables o colores, los que se identifican con una ID entre 0 y 23, donde la congestión en un color no bloquea el tráfico en otro.

Mensajes de 32 bit de tamaño, llamados wavelets, pueden ser enviados o recibidos por un PE vecino en un ciclo de reloj. La WSE provee comunicaciones eficientes de grano fino entre PEs, los cuales deben ser capaces de, en pocos ciclos, responder a la llegada de un wavelet, actualizar su estado interno y enviar wavelets. Cada wavelet tiene un tag de 5 bits que codifica su color. Este color determina la ruta del wavelet a través de la malla y qué tarea, si es que hay alguna, lo utilizará cuando sea recibido. Los wavelets pueden ser de datos o de control. Es posible enviar wavelets de control que contengan datos si esta requiere el uso de, a lo más, los últimos 16 bits, ya que los wavelets de control configuran sus opciones en los primeros 16 bits, correspondiente a su cabecera.

Para enviar y recibir datos, los PEs cuentan con colas de entrada y salida, que corresponden a buffers donde los datos son almacenados antes de entrar o salir del CE. El uso de estas colas se lleva a cabo a través de vectores de entrada y salida, los cuales son representaciones de secuencias de wavelets entrantes o salientes. La definición de estos vectores puede ser realizada en tiempo de ejecución y se les debe indicar el color a través del que enviarán o recibirán los datos, la cantidad de wavelets que esperan enviar o recibir cada vez que sean utilizados, la cola a la que estarán asociados y, sólo en el caso de los vectores de salida, si los

wavelets serán de control o no. La cantidad de colas disponibles para cada PE es limitada, siendo 8 colas de entrada y 6 de salida en la WSE-2, mientras que la cantidad de colas de salida asciende a 8 en la WSE-3. Las operaciones de envío y recepción de datos pueden realizarse de manera asíncrona si así se desea, sin embargo, dado que una cola puede ser utilizada para distintos fines, por ejemplo, dos vectores de entrada pueden estar asociados a la misma cola pero a distintos colores, se debe asegurar que no habrá concurrencia en el uso de una misma cola.

Los colores son configurados en el código del layout, o bien, en el código del PE, en tiempo de compilación, de manera independiente para cada PE. Los elementos más importantes a definir cuando se configura un color son las direcciones (norte, sur, este, oeste y RAMP) de origen (rx) y destino (tx), las que pueden tener una o más direcciones configuradas. Estas direcciones indican desde donde el router del PE espera recibir datos en ese color y donde los enviará luego de recibirlos respectivamente.

Los colores no pueden cambiar su configuración una vez definidos, a menos que se haga uso de switches. Un switch permite cambiar la configuración de un color en tiempo de ejecución, pero la forma de estos cambios es definida en tiempo de compilación. Cada configuración de color para un PE puede tener asignado como máximo un switch, el cual presenta las siguientes configuraciones principales:

- Posiciones: Es posible definir hasta tres posiciones en un switch. Cada posición corresponde a sólo una dirección y se deberá indicar si es de origen o destino.
- Modo anillo: Esta configuración determinará si el switch se comportará como un anillo o no. En caso de que no, cuando llegue a la última posición, no avanzará más.
- Posición de inicio: Indica la posición de inicio del switch, considerando la posición 0 la descrita en la configuración base del color.

El switch de un color sólo avanzará cuando el router reciba un wavelet de control que contenga la opción de hacer avanzar el switch a través del mismo color, pero luego de haberlo enviado a su destino. Mientras el switch se encuentra en alguna de sus posiciones configuradas, su descripción sobrescribe la indicada en la configuración base del color. Por ejemplo,

si un PE necesita enviar cuatro mensajes M1, M2, M3 y M4, donde cada uno será un entero de 16 bits y M1 debe ir hacia el oeste, M2 hacia el norte, M3 hacia el este y M4 hacia el sur, en lugar de configurar cuatro colores distintos, es posible realizar esto con sólo un color y switches utilizando la siguiente configuración del color:

- Dirección de origen: RAMP.
- Dirección de destino: Oeste.
- Switch:
  - Posición 1: Destino norte.
  - Posición 2: Destino este.
  - Posición 3: Destino sur.

Dado que sólo se desea enviar 16 bits de información en cada mensaje, es posible empaquetarla en un wavelet de control tal como se mencionó anteriormente, por lo que cada wavelet será de control y contendrá en sus primeros 16 bits la opción para hacer avanzar el switch, mientras que en sus últimos 16 bits el número que se desea enviar. La Figura 2.4 muestra el comportamiento de este ejemplo, donde se puede observar que luego de cada mensaje la dirección de destino del color cambia a la siguiente posición configurada en el switch.

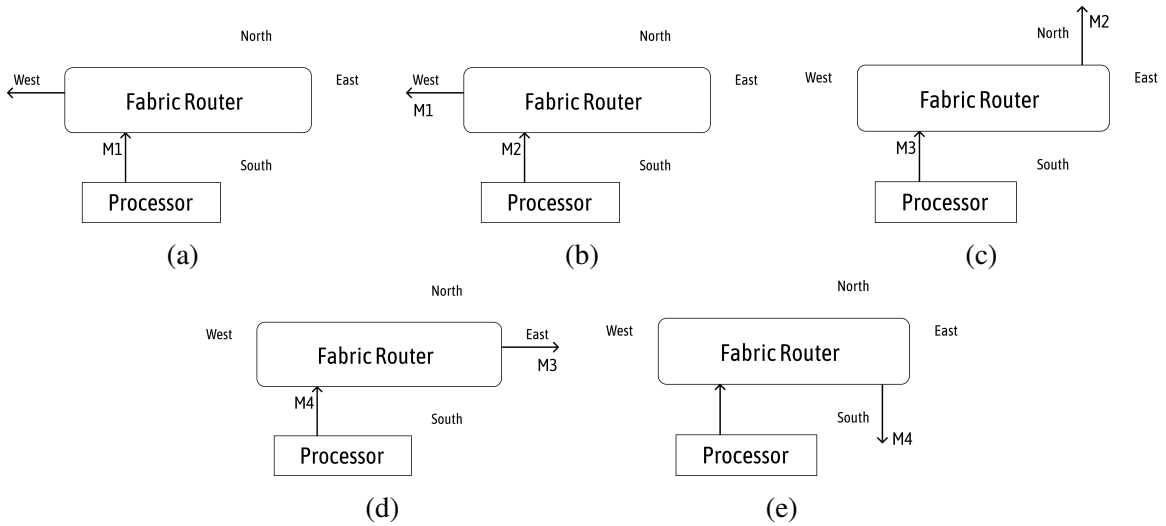


Figura 2.4: Ejemplo de envío de mensajes a través de un color utilizando switches.

Envío de datos desde el host

Cuando se envían las matrices desde el host al CS, es necesario hacer un reordenamiento de los datos previo al envío debido a la forma en que este recibe los datos. La Figura 2.5 muestra un ejemplo de envío en orden mayor de fila de una matriz de 4×4 elementos a una malla de 2×2 PEs, donde los datos deben ser reorganizados previo al envío para que los primeros 4 elementos de la información a enviar correspondan al PE(0,0), los siguientes 4 a los del PE(1,0) y así sucesivamente con el resto de elementos.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

(a) Matriz original en orden mayor de fila.

1	2	5	6	3	4	7	8	9	10	13	14	11	12	15	16

(b) Matriz reordenada previo al envío al CS.

PE(0,0)

1	2
5	6

PE(1,0)

3	4
7	8

PE(0,1)

9	10
13	14

PE(1,1)

11	12
15	16

(c) Distribución de la matriz en 2×2 PEs con envío en orden mayor de fila.

Figura 2.5: Ejemplo de envío de una matriz de 4×4 en orden mayor de fila desde el host al CS.

2.5.4. Simulador

Existe un simulador de la arquitectura si no se cuenta con acceso a un CS conectado a la red, sobre el cual es posible compilar y ejecutar código CSL. Al ejecutar un programa, se generarán archivos que nos indicarán distintas estadísticas de la ejecución, siendo una de las más relevantes la cantidad de ciclos necesarios para la ejecución del programa, ya que esta

puede ser traducida a tiempo de ejecución haciendo uso de la Ecuación 2.4.

$$\text{Tiempo } [\mu s] = (\text{ciclos}/0{,}85) * 1{,}0 \text{ e} - 3 \ [\mu s] \tag{2.4}$$

Aunque útil, al ser un simulador no deja de presentar limitaciones propias de uno. Las dos principales son:

1. Cantidad de PEs de las simulaciones: Debido a restricciones de memoria, no es posible hacer pruebas con una cantidad de PEs arbitraria, lo que limita el tamaño de los experimentos que es posible realizar.

2. Tiempo de ejecución: Dado que es un simulador de la arquitectura, el tiempo necesario para obtener los resultados es sumamente elevado en comparación a cuanto se tardaría en la máquina real.

Capítulo 3

Implementación

3.1. Diseño de los algoritmos en la arquitectura WSE

Dada la cantidad de memoria disponible por cada PE es necesario subdividir en pequeñas submatrices las matrices a procesar, por lo que todos los diseños que se presentan en este capítulo tienen esto en cuenta. Por lo mismo, los PEs necesitarán comunicar parte de sus elementos o cálculos realizados a otros PEs, para lo que se hará necesario el uso de los colores previamente discutidos. Cada propuesta de implementación en este capítulo detallará las necesidades comunicacionales de los algoritmos, la cantidad de colores a utilizar y el cómo serán utilizados, así como cualquier particularidad relevante que convenga mencionar en cada caso. Adicionalmente, dado que es un primer paso en la implementación de estos algoritmos en la arquitectura WSE, por conveniencia, se trabaja con matrices cuadradas y todos los PEs almacenan la misma cantidad de elementos correspondientes a submatrices, también cuadradas.

3.1.1. Multiplicación de matrices de Cannon

El algoritmo de Cannon [3] se puede dividir en dos fases, la de organización inicial de los datos, correspondiente al movimiento inicial de las submatrices de A y B y, por otra parte, la fase de cómputo. Para simplificar la implementación, considerando que de todas formas se debe realizar un reordenamiento de los datos antes de enviarlos desde el host, el movimiento inicial se realiza en el código del host, por lo que el código CSL necesita realizar las siguientes comunicaciones:

- Envío de las submatrices de A de manera horizontal.
- Envío de las submatrices de B de manera vertical.

Teniendo en cuenta que todos los PEs le enviarán datos a un vecino, excepto por aquellos que producen desbordamiento, se decide hacer uso de seis colores, tres para comunicación horizontal de A y tres para la comunicación vertical de B. La configuración de los colores será como sigue:

- Color rojo: Utilizado para enviar las submatrices de A desde la primera columna a la última columna de PEs.
- Color azul: Los PEs con componente x impar envían su submatriz A hacia el oeste, mientras que los PEs con componente x par reciben la misma submatriz utilizando este color.
- Color cian: Similar al color azul, pero es utilizado para enviar por los PEs con componente x par y para recibir por los PEs con componente x impar.
- Color morado: Utilizado para enviar las submatrices de B desde la primera fila a la última fila de PEs.
- Color verde: Los PEs con componente y impar envían su submatriz B hacia el oeste, mientras que los PEs con componente y par reciben la misma submatriz utilizando este color.

- Color café: Similar al color verde, pero es utilizado para enviar por los PEs con componente y par y para recibir por los PEs con componente y impar.

Adicionalmente, dado que no es predecible el comportamiento de recibir datos en un espacio de memoria mientras se envían los mismos, se hace necesario que los PEs, adicional a las submatrices de A, B y C, tenga una submatriz temporal donde almacenar los datos recibidos, se utiliza sólo una submatriz temporal para ser capaz de aprovechar de mejor manera la memoria de cada PE. La Figura 3.1 muestra esta configuración de manera gráfica para una malla de 4×4 PEs.

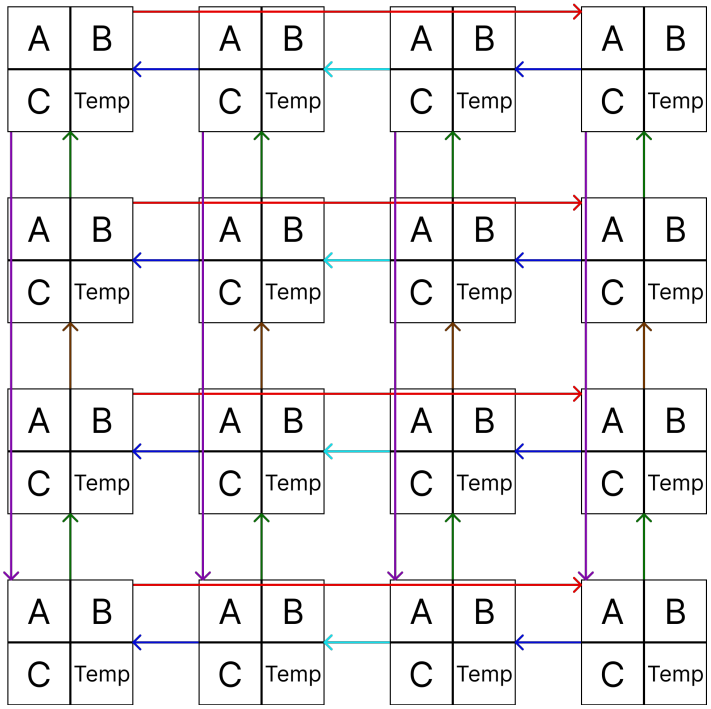


Figura 3.1: Ejemplo de la configuración de colores en una malla de 4×4 PEs para la multiplicación de matrices.

La Figura 3.2 muestra el flujo que el algoritmo sigue entre dos actualizaciones de la matriz C. Partiendo desde una actualización de la matriz C, como muestra la Subfigura a, los PEs con componente x par envían sus submatrices A (Subfigura b), una vez recibidas por los PEs con componente x impar, estos mismos proceden a hacer el envío de sus propias submatrices de A (Subfigura c). Una vez finalizados estos envíos, todos los PEs intercambian la dirección de memoria de su submatriz A con la submatriz Temp, que contiene la última submatriz

recibida, estos cambios se ven reflejados en el cambio de posiciones de la matriz *A* y Temp que ocurre entre las Subfiguras c y d. Una vez terminado esto, se repite el mismo proceso, iniciando con los PEs con componente *y* par enviando las submatrices de *B*, como se ve en las Subfiguras d y e, para luego poder realizar una nueva actualización de la matriz *C* (Subfigura f). Es necesario realizar las comunicaciones, tanto de *A* como *B*, en dos pasos debido a que hacerlo en uno requeriría una mezcla de envío asincrónico y recepción sincrónica de datos, sin embargo la combinación de estos en un mismo PE no se recomienda ya que puede generar comportamientos inesperados o que los PEs entren en un deadlock.

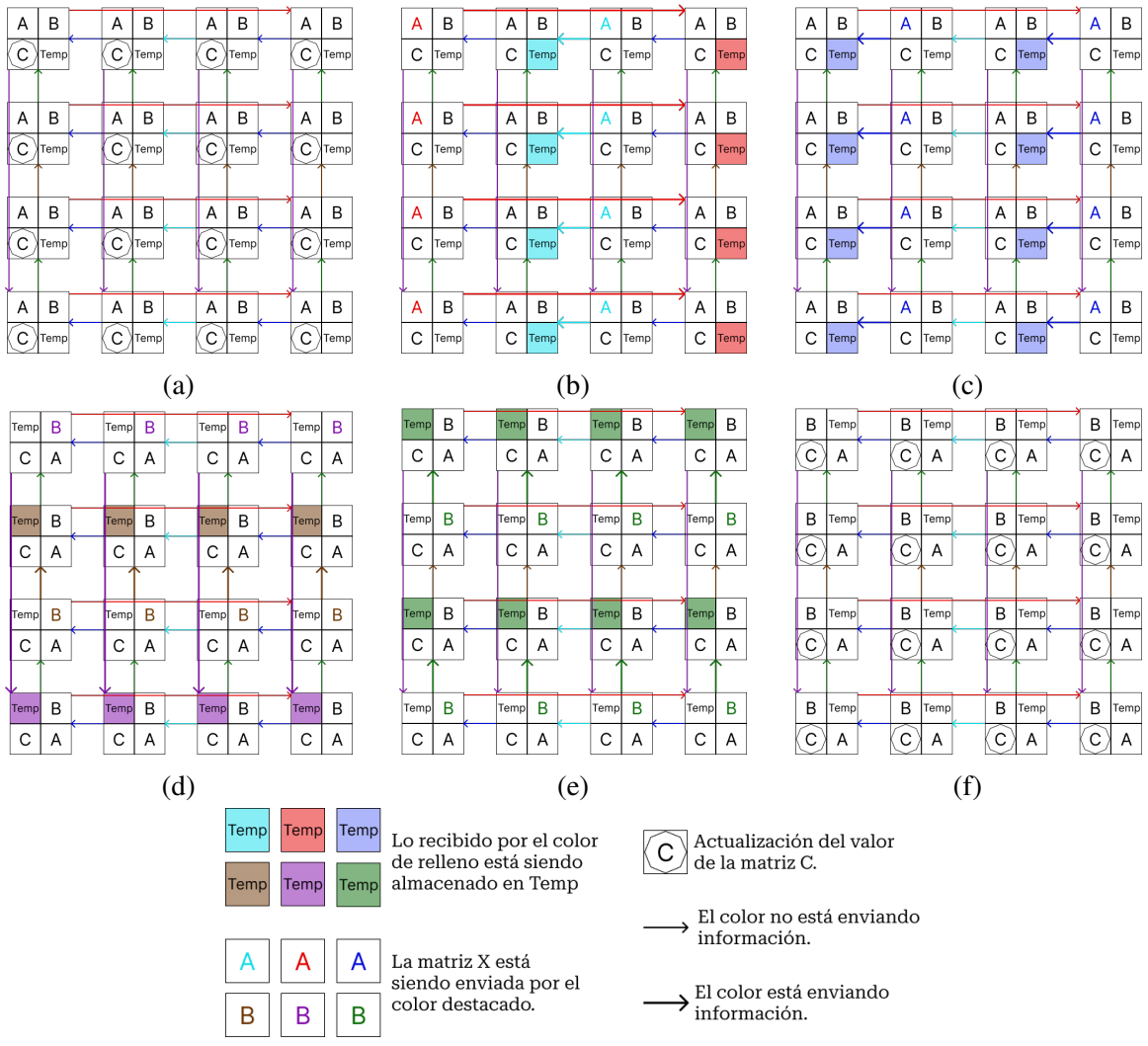


Figura 3.2: Ejemplo del flujo de información entre dos pasos de cálculo para la multiplicación de matrices en una malla de 4×4 PEs.

Algo que es importante notar, es que aunque el color rojo y morado envían desde un borde al otro, físicamente esos datos deben pasar por todos los routers, por lo que todos los PEs a través de los que pasan deben tener configurado el color correctamente. El Código 3.1 muestra la configuración del color rojo considerando pe_x como la coordenada x de un PE dado y grid_width el ancho de la malla. En esta configuración es posible notar que todos los PEs intermedios tienen configurado el color rojo para recibir desde el oeste y enviar hacia el este, de esta manera todos los wavelets que se reciban por el color rojo serán enviados al PE que se encuentra al oeste sin que ninguno entre al CE. El color morado tiene una configuración similar pero en lugar de ir de este a oeste lo hace de norte a sur.

```
1  if (pe_x == 0) {
2    @set_color_config(pe_x, pe_y, color_rojo, .{
3      .routes = .{ .rx = .{RAMP}, .tx = .{EAST}}
4    });
5  }
6  else if (pe_x == grid_width - 1) {
7    @set_color_config(pe_x, pe_y, color_rojo, .{
8      .routes = .{ .rx = .{WEST}, .tx = .{RAMP}}
9    });
10 }
11 else {
12   @set_color_config(pe_x, pe_y, color_rojo, .{
13     .routes = .{ .rx = .{WEST}, .tx = .{EAST}}
14   });
15 }
```

Código 3.1: Configuración del color rojo para el algoritmo de Cannon.

3.1.2. Descomposición LU

Para el diseño de la descomposición LU se hizo uso de lo expuesto en la Subsección 2.2.1, donde es importante identificar los tipos de mensajes que los PEs necesitarán enviar. Considerando lo expuesto en el Algoritmo 1, en cada iteración los valores que se necesitan comunicar son:

1. El valor del elemento en la diagonal debe ser enviado a los elementos bajo este, en la misma columna, para realizar el paso de división.
2. Los valores de los elementos bajo el elemento de la diagonal, en su misma columna,

deben ser enviados a los elementos a la derecha de ellos, en sus respectivas filas, para realizar el paso de eliminación.

3. Los valores de los elementos a la derecha del elemento de la diagonal, en su misma fila, deben ser enviados a los elementos bajo ellos, en sus respectivas columnas, para realizar el paso de eliminación.

Con esto en consideración, es intuitivo pensar en el uso de 3 colores para la comunicación en la malla, sin embargo, se propone el uso de un color adicional para enviar una señal que le indique a los elementos a la derecha de la diagonal que deben enviar sus valores hacia abajo para simplificar el código en este aspecto, de modo que el envío se realice como la respuesta de una tarea de datos en lugar de utilizar contadores, puesto el el programa ya utiliza uno para otras funciones, lo que le da una mayor claridad al código con un bajo impacto de rendimiento, puesto que la comunicación de un wavelet tarda muy pocos ciclos de reloj. La Figura 3.3 muestra una matriz de 3×3, con un elemento por PE con 4 colores para su comunicación. Los colores son utilizados de la siguiente forma:

- El color verde envía desde un PE diagonal hacia todos los PEs al sur el valor del elemento divisor en el paso de división.
- El color azul envía desde un PE diagonal a todos los PEs al este una señal.
- El color morado envía desde un PE al este de la diagonal, hacia todos los PEs al sur, el valor necesario para el paso de eliminación. Esto sólo ocurre luego de haber recibido una señal por el color azul.
- El color rojo envía desde un PE al sur de la diagonal, hacia todos los PEs al este, el valor necesario para el paso de eliminación. Esto sólo ocurre luego de haber realizado el paso de división.

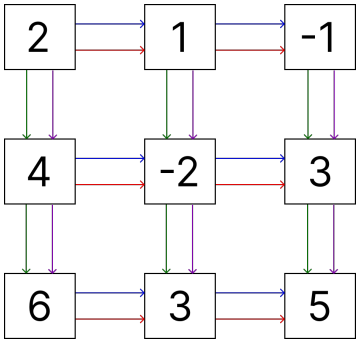


Figura 3.3: Ejemplo de una matriz de 3×3 con un elemento por PE utilizando la configuración de colores propuesta para la descomposición LU.

La Figura 3.4 muestra el flujo de información para una matriz de 3×3 con un elemento por PE. El Algoritmo 1 muestra que la cantidad de actualizaciones que recibe un elemento de la matriz es igual al índice menor entre su fila y su columna, lo que se ve reflejado en esta propuesta, ya que existe un desbalance en la carga de trabajo de los PEs, siendo los de la primera fila y columna los que menos carga tienen mientras que el PE de la esquina inferior derecha el que más carga tiene, por lo que durante la ejecución existirán PEs ociosos.

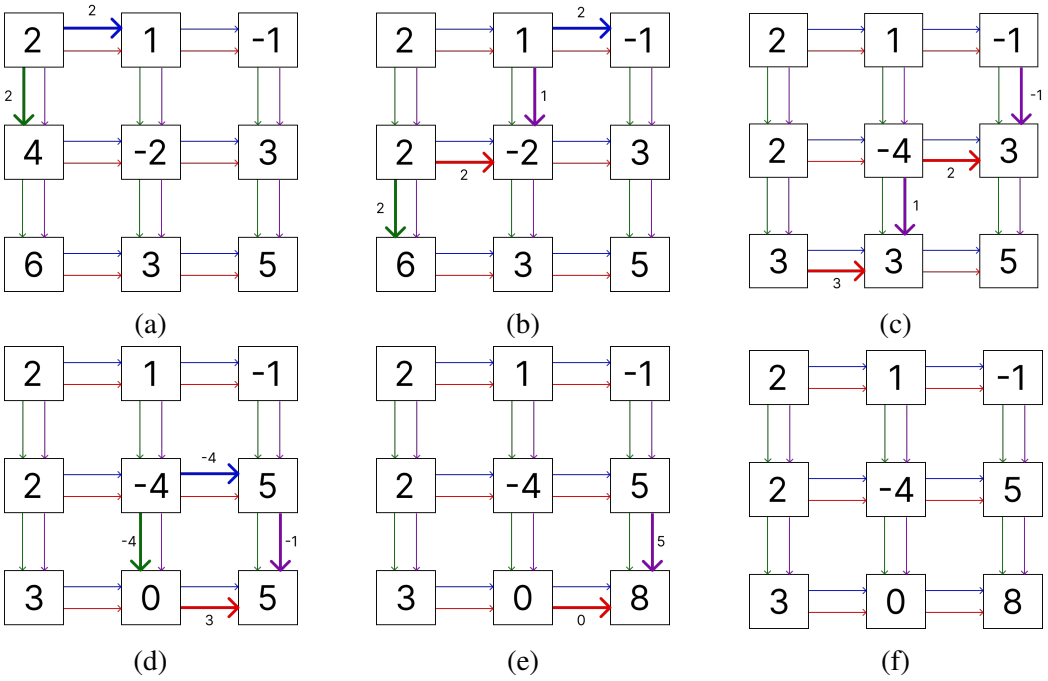


Figura 3.4: Ejemplo de cómputo de una descomposición LU para una matriz de 3×3 con un elemento por PE.

Los ejemplos expuestos considerando un elemento por PE ayudan a entender el flujo de las comunicaciones y datos a través de la malla utilizada, sin embargo, en aplicaciones reales es necesario que cada PE tenga más de un elemento debido a las dimensiones de los problemas. La Figura 3.5 muestra el diseño utilizado para una matriz de 9×9 en una malla de 3×3 PEs. A pesar del aumento de elementos por PE, no se hace necesario añadir nuevos colores. La Figura 3.6 muestra un ejemplo del flujo de información en esta matriz hasta que el PE(2,2) hace su primer paso de eliminación. Es posible apreciar que si bien al inicio el paralelismo es poco, en la medida que avanzan las comunicaciones, los elementos siendo calculados en paralelo van en aumento, lo que en algún momento llegará a un máximo y luego comenzará a decrecer debido a que cada vez se trabajará con menos elementos de la matriz. Los ceros enviados a través del canal azul corresponden al wavelet necesario para activar la tarea de datos que se mencionó anteriormente, el valor enviado no tiene mayor relevancia puesto que la tarea no hace uso de este.

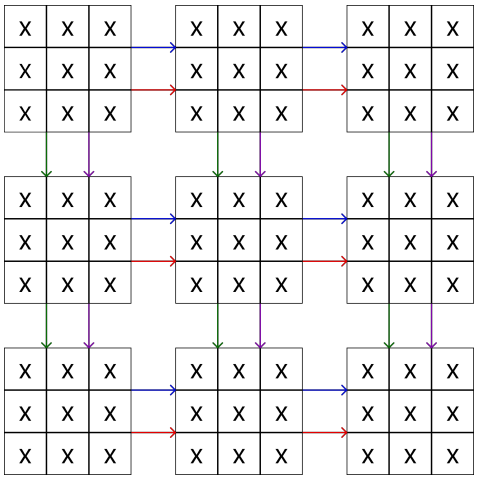


Figura 3.5: Ejemplo de una matriz de 9×9 en una malla de 3×3 PEs utilizando la configuración propuesta para la descomposición LU.

Dado que las comunicaciones de los PEs suelen hacerse desde un PE a todos los PEs al sur de él, o bien, a todos los PEs al este del mismo, se optó por una configuración de colores que permita a un PE recibir la información al mismo tiempo que la envía al siguiente PE. El Código 3.2 muestra la configuración del color verde considerando pe_x la coordenada en el eje x de un PE, pe_y la coordenada en el eje y de un PE y grid_height la altura en PEs de la malla donde se ejecutará la descomposición.

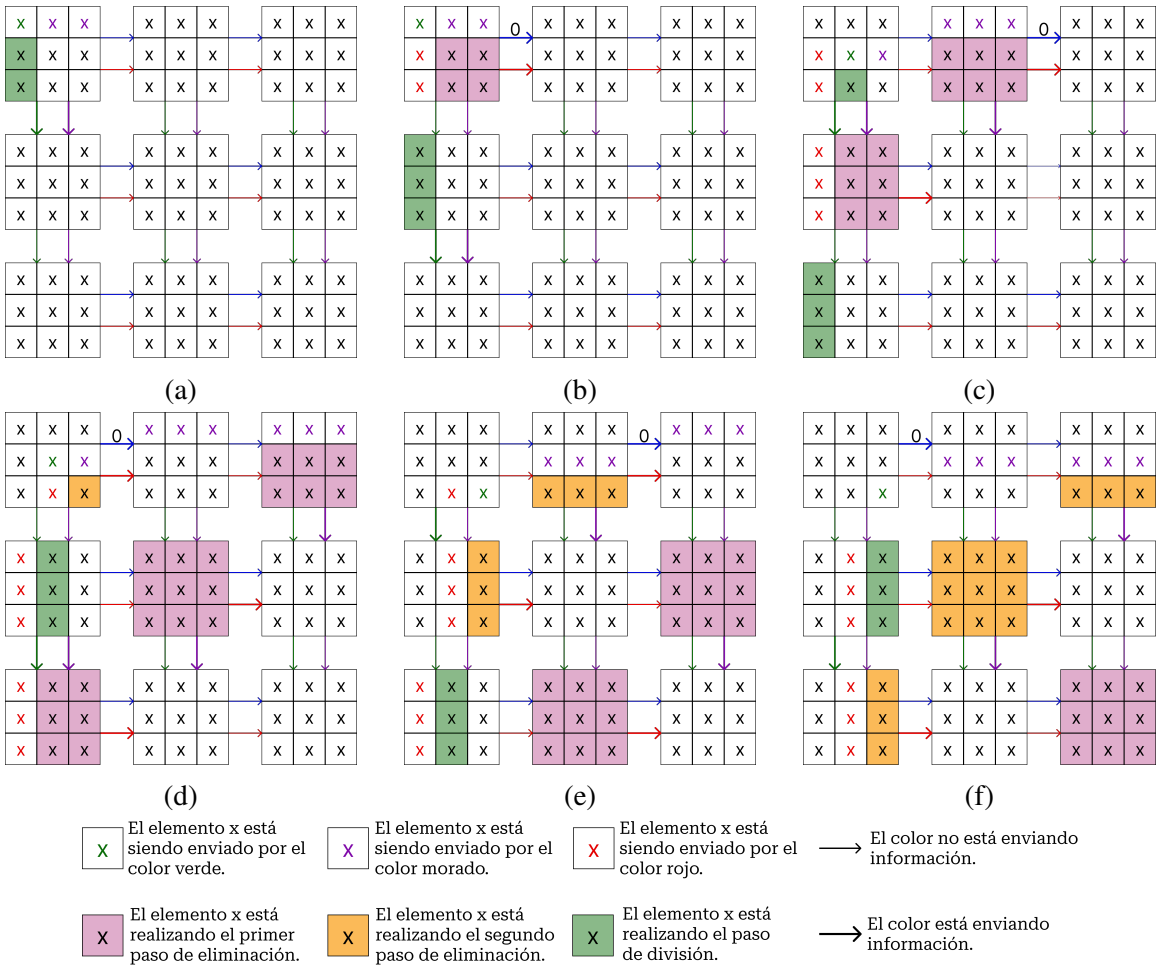


Figura 3.6: Ejemplo de cómputo de una descomposición LU para una matriz de 9×9 con 9 elementos por PE hasta que el PE(2,2) realiza el primer paso de eliminación.

```
if (pe_x == pe_y) {
  @set_color_config(pe_x, pe_y, color_verde, .{
    .routes = .{ .rx = .{RAMP}, .tx = .{SOUTH}}
  });
}
else if ( pe_x < pe_y) {
  @set_color_config(pe_x, pe_y, color_verde, .{
    .routes = .{
      .rx = .{NORTH},
      .tx = if (pe_y < grid_height - 1) .{RAMP, SOUTH} else .{RAMP}
    }
  });
}
```

Código 3.2: Configuración del color verde para la descomposición LU.

Con esta configuración, a través del color verde los PEs de la diagonal envían desde el RAMP hacia el sur y los PEs al sur de estos reciben desde el norte, y envían los datos hacia el RAMP y el sur, de esta manera, para pasar la información al siguiente PE, no hace falta que la información entre al CE de cada PE para luego tener que volver a salir y ser enviada. El color azul tiene una configuración similar pero con envío desde la diagonal hacia el este.

En cuanto a los colores de eliminación, para mantenerlos en dos colores, hizo falta algunas configuraciones adicionales debido a que los PEs no sólo reciben y transmiten la información recibida a los siguientes PEs, si no que también, en algún punto, deben enviar su propia información. Para lograr lo anterior los colores utilizados para la eliminación tienen una configuración similar al color verde y azul en lo respectivo a las direcciones utilizadas, pero se adiciona como origen válido la dirección RAMP y se incluye un switch que omite la dirección de destino RAMP. El Código 3.3 muestra la configuración del color rojo en las columnas que no se encuentran en el borde de la malla. El modo anillo es necesario ya que el wavelet de control también será enviado a los PEs al este, pero estos deben seguir teniendo la dirección de destino RAMP en su configuración, por lo que los PEs a la derecha de aquellos que necesiten cambiar su configuración, también enviarán un wavelet de control para volver a la configuración inicial. Una configuración similar es realizada para el color morado, pero con comunicaciones de norte a sur.

```
1 @set_color_config(pe_x, pe_y, color_rojo, .{
2   .routes = .{ .rx = .{RAMP, WEST}, .tx = .{RAMP, EAST}},
3   .switches = .{ .pos1 = .{ .tx = EAST}, .ring_mode = true}
4 });
```

Código 3.3: Configuración del color rojo en las columnas intermedias para la descomposición LU.

3.1.3. Descomposición QR

Para aprovechar de manera correcta la arquitectura WSE para la descomposición QR, lo principal es buscar maximizar la cantidad de ceros en paralelo que se están insertando en la matriz. Para realizar la inserción de un cero es necesario realizar una rotación de Givens y actualizar las dos filas involucradas en la rotación, tal como muestra el Algoritmo 3, con esto

en cuenta, hay dos necesidades comunicacionales para la descomposición QR:

- Enviar y recibir los valores de la matriz de manera vertical para calcular el $\sin\theta$ y $\cos\theta$ de la rotación.
- Enviar y recibir los valores de la matriz de manera vertical para aplicar la rotación.
- Enviar y recibir de manera horizontal los valores de $\sin\theta$ y $\cos\theta$ para la actualización de las filas.

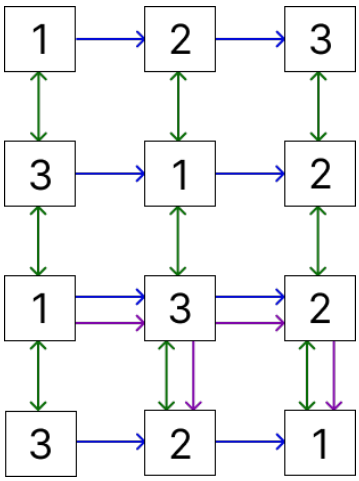


Figura 3.7: Ejemplo de una matriz de 4×3 con un elemento por PE utilizando la configuración de colores propuesta para la descomposición QR.

Con esto en consideración, es posible realizar las comunicaciones con dos colores, uno para enviar los valores de la matriz y otro para enviar los valores de la rotación. Primero se ejemplificará como es el flujo de las comunicaciones en el caso de un elemento por PE, donde los ceros se harán partiendo desde abajo hacia arriba, desde izquierda hacia la derecha, tal como en el Algoritmo 3, pero, luego de que un PE de la penúltima fila se vuelva 0, el PE al sur este del mismo recibirá una señal para comenzar a insertar los ceros en su columna. Para el envío de esta señal hace falta el uso de un color adicional a los dos mencionados previamente. La Figura 3.7 muestra una matriz de 4×3 con un elemento por PE haciendo uso de esta configuración. Los colores son utilizados de la siguiente forma:

- Color verde: Comunica el valor del elemento de la matriz.

- color azul: Comunica el valor de $\sin\theta$ y $\cos\theta$ calculado en cada rotación.
- Color morado: Comunica la señal para iniciar la inserción de ceros en la columna.

La Figura 3.8 muestra el flujo de comunicaciones para una matriz de 4×3 hasta que se inicia la inserción de ceros en la segunda columna, como se ve en la Subfigura f, donde paralelamente se está realizando la comunicación de los valores de la rotación calculados para el PE (0,1) desde los PEs (0,0) y (0,1) hacia el este, el intercambio de los valores de la matriz para actualizar su valor de los PEs (2,1) y (2,2) y, finalmente, el intercambio de los valores de la matriz para insertar un cero en el PE (1,3) entre este mismo y el PE (1,2).

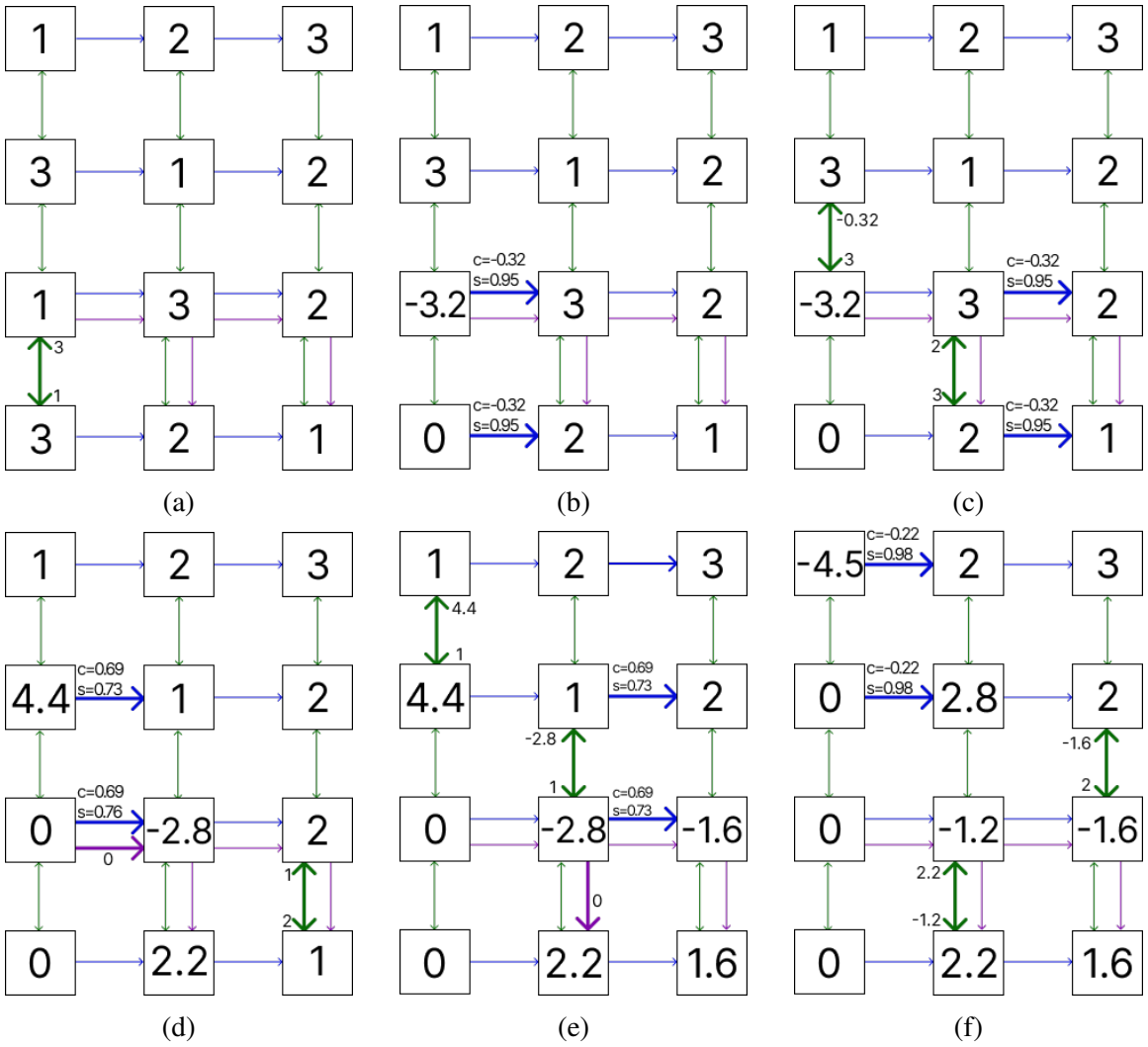


Figura 3.8: Ejemplo de cómputo de una descomposición QR para una matriz de 4×3 con un elemento por PE hasta que se inicia la inserción de ceros en la segunda columna.

La versión que utiliza varios elementos por PE sigue una idea similar, hacer ceros desde oeste hacia este y de sur a norte, sin embargo, para aprovechar de mejor manera el paralelismo de la arquitectura los PEs sacan ventaja del hecho de que la inserción de ceros puede ser realizada entre cualquier par de filas. Para lo anterior los PEs seguirán el siguiente comportamiento:

- Los PEs que están sobre la diagonal principal, sólo esperarán los valores de las rotaciones para realizar las actualizaciones a sus valores.
- Los PEs en la diagonal principal insertarán los ceros posibles con sus propios elementos, es decir, los ceros bajo la diagonal principal de la submatriz que contienen, cuando el PE al oeste haya terminado de insertar todos sus ceros. Luego realizará los intercambios y actualizaciones necesarios con el PE del sur para que este termine de insertar los ceros que le corresponden. Si el PE está en la primera columna, inicia insertando los ceros propios.
- Los PEs que están bajo la diagonal principal insertarán los ceros posibles con sus propios elementos, luego, desde el PE más al sur de la columna se iniciará un intercambio de información con el PE del norte para poder insertar un cero en el elemento superior izquierdo de la submatriz donde no se haya insertado un cero, lo que permitirá continuar insertando ceros en la diagonal de este elemento utilizando sus propios datos. Este proceso se repite hasta que todos los elementos de la submatriz se hayan llenado de ceros, adicionalmente, esto sólo inicia cuando el PE al oeste ya ha llenado su submatriz de ceros. Los PEs de la primera columna inician con este comportamiento al lanzar el programa.

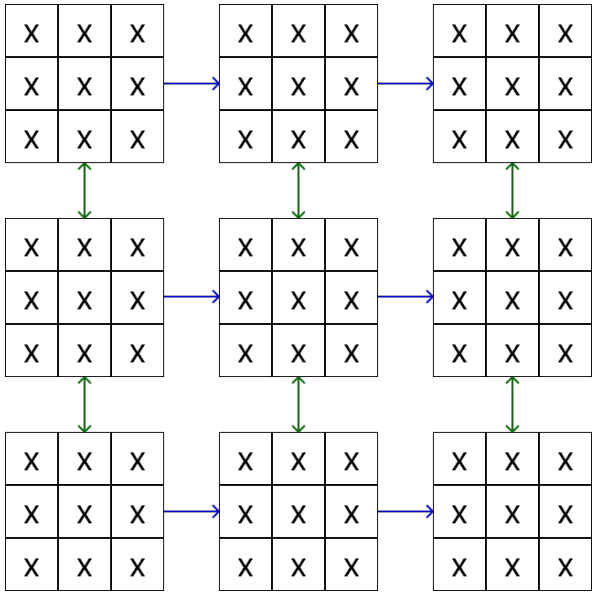


Figura 3.9: Ejemplo de una matriz de 9×9 en una malla de 3×3 PEs utilizando la configuración propuesta para la descomposición QR.

La Figura 3.9 muestra la configuración de dos colores en una malla de 3×3 PEs para una matriz de 9×9 elementos. El flujo de comunicaciones es expuesto en la Figura 3.10, donde las Subfiguras a, b y c muestran como los PEs de la primera columna insertan los ceros bajo la diagonal principal de sus submatrices mientras actualizan y propagan los valores de las rotaciones a sus respectivas filas, luego, en la Subfigura d, ya que el PE (0,2) no puede insertar más ceros con sus datos, intercambia información de los valores de su primera fila con el PE (0,1), para así insertar un cero en la esquina superior izquierda de su submatriz, como se ve en la Subfigura e, lo que le permite seguir insertando ceros en su diagonal con sus datos, como muestra la Subfigura f. El PE (0,1) repetirá el mismo intercambio de información con el PE (0,0) e insertará ceros en su diagonal como el PE (0,2). Una vez que este último haya llenado de ceros su submatriz, el PE (1,2) iniciará este proceso nuevamente.

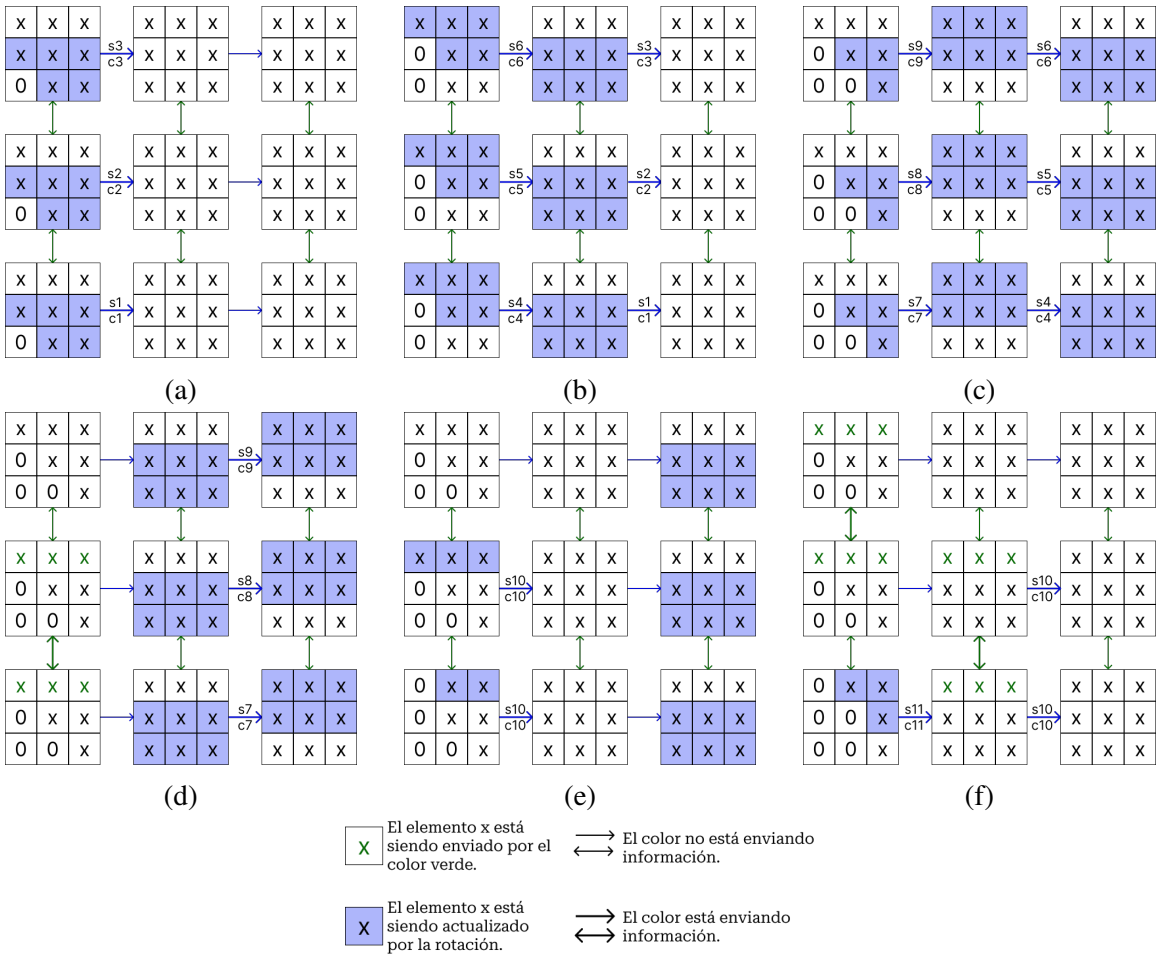


Figura 3.10: Ejemplo de cómputo de una descomposición QR para una matriz de 9×9 en una malla de 3×3 PEs hasta que el PE (0,2) inserta su quinto cero.

La configuración del color azul en esta descomposición es igual a la del color rojo en la descomposición LU debido a las mismas razones, los PEs no sólo recibirán y transmitirán los valores de $\sin\theta$ y $\cos\theta$, si no que también deberán enviar los propios en algún momento, por lo que se debe evitar que reciban sus propios valores. En cuanto al color verde, como se mencionó anteriormente, hace falta el uso de switches para ser capaces de comunicar de manera vertical todos los PEs haciendo uso de sólo un color. Considerando *pe_y* la coordenada en el eje *y* de un PE y *grid_height* la cantidad de PEs en la malla en el eje *y* donde se ejecutará la descomposición, el Código 3.4 muestra la configuración del color verde, con la cual los PEs de la fila superior se alternarán entre entre recibir desde el sur y enviar hacia el sur, los PEs de la fila inferior se alternarán entre enviar hacia el norte y recibir desde el norte,

mientras que los PEs intermedios se alternarán entre recibir desde el sur, enviar hacia el sur, enviar hacia el norte y recibir desde el norte.

```
if (pe_y == 0) {
  @set_color_config(pe_x, pe_y, A_color, .{
    .routes = .{ .rx = .{RAMP, SOUTH}, .tx = .{RAMP}},
    .switches =.{.pos1 = .{ .tx = SOUTH}, .ring_mode = true}
  });
}
else if (pe_y == (grid_height - 1)) {
  @set_color_config(pe_x, pe_y, A_color, .{
    .routes = .{ .rx = .{RAMP, NORTH}, .tx = .{RAMP}},
    .switches = .{
      .pos1 = .{ .tx = NORTH},
      .ring_mode = true,
      .current_switch_pos = 1
    }
  });
}
else {
  @set_color_config(pe_x, pe_y, A_color, .{
    .routes = .{ .rx = .{RAMP, NORTH, SOUTH}, .tx = .{RAMP}},
    .switches = .{
      .pos1 = .{ .tx = SOUTH},
      .pos2 = .{ .tx = NORTH},
      .pos3 = .{ .tx = RAMP},
      .ring_mode = true
    }
  });
}
```

Código 3.4: Configuración del color verde para la descomposición QR.

3.2. Repositorio

Todo el código desarrollado para este trabajo se puede encontrar en el repositorio público de Github apdelsm/Matrix-algorithms-in-cerebras. En el repositorio se puede encontrar los códigos generales que manejan varios elementos por PE y, para el caso de las descomposiciones LU y QR, versiones que manejan 1 elemento por PE, realizadas para familiarizarse con la arquitectura y el lenguaje. Adicionalmente, todos los algoritmos desarrollados incluyen en el host una etapa de validación de resultados para asegurar que estos son correctos.

Capítulo 4

Resultados

4.1. Configuración de los experimentos

Los experimentos realizados buscan analizar las prestaciones de la arquitectura en la medida que el tamaño de las matrices cambia, así como descubrir el impacto de la cantidad de información que maneja cada PE. Para esto se establecieron dos tipos de experimentos, uno que mantiene la cantidad de elementos por PE constantes en la medida que el tamaño del problema crece y otro donde la cantidad de elementos por PE va en aumento junto con el tamaño del problema.

Todos los experimentos utilizan matrices cuadradas de $n \times n$, donde n está dado por el tamaño del problema. De la misma manera, todas las mallas de PEs serán cuadradas de $p \times p$ PEs, donde p está dado por el tamaño de la malla. Adicionalmente, el tamaño de las submatrices de cada PE estará dado por $n\_pe \times n\_pe$.

La Tabla 4.1 muestra las distintas configuraciones de los experimentos realizados para este trabajo. Debido a la cantidad de memoria de los PEs, los experimentos fueron limitados a un máximo de 64×64 elementos por PE ya que no es posible almacenar la siguiente potencia de dos en un PE. Con lo anterior en consideración y el hecho de que el algoritmo de Cannon debe almacenar 4 submatrices, los tamaños de las submatrices de esta implementación

fueron limitadas a 32×32, lo que da una cantidad de elementos por PE igual a los otros experimentos. Todos los elementos de las matrices son punto flotantes de precisión simple, la máxima permitida por la arquitectura.

Tabla 4.1: Configuraciones de los distintos tamaños de problema n, tamaños de la malla p y tamaños de submatrices n_pe para cada tipo de experimento y cada algoritmo.

	Descomposición LU			Descomposición QR			Algoritmo de Cannon		
	n	p	n_pe	n	p	n_pe	n	p	n_pe
Elementos por PE constantes	128	2	64	128	2	64	128	4	32
	256	4	64	256	4	64	256	8	32
	512	8	64	512	8	64	512	16	32
	1024	16	64	1024	16	64	1024	32	32
	2048	32	64	2048	32	64	2048	64	32
	4096	64	64	4096	64	64	-	-	-
Elementos por PE en aumento	128	64	2	128	64	2	128	64	2
	256	64	4	256	64	4	256	64	4
	512	64	8	512	64	8	512	64	8
	1024	64	16	1024	64	16	1024	64	16
	2048	64	32	2048	64	32	2048	64	32
	4096	64	64	4096	64	64	-	-	-

La información obtenida de cada experimento fue la cantidad de ciclos informada por el simulador, los que incluyen la transmisión de los datos desde las conexiones de los bordes de la WSE hasta sus respectivos PEs, la ejecución del algoritmo y la transmisión de los datos desde los PEs hasta las conexiones antes mencionadas.

Es relevante mencionar que dadas las limitaciones del simulador, realizar experimentos más grandes que los mencionados requiere una gran cantidad de tiempo, ya que los experimentos de mayor tamaño realizados para este trabajo requirieron más de 8 días de simulación, mientras que los ciclos informados indican que en un sistema real hubiera tardado 303[ms], por lo que hacerlos crecer al siguiente tamaño relevante para el análisis sin tener acceso a un CS real es bastante costoso en tiempo.

4.2. Resultados experimentales

El tiempo necesario para la ejecución de los algoritmos se puede calcular utilizando la Ecuación 2.4. La Figura 4.1 muestra el tiempo de ejecución de la descomposición QR para los distintos experimentos, donde es posible observar que el tiempo de ejecución de la versión de elementos por PE en aumento es más rápida que aquella que mantiene la cantidad de elementos por PE constantes, sin embargo, la tendencia de crecimiento en el tiempo de ejecución para el caso de elementos por PEs constantes es más cercana a un crecimiento lineal que a uno exponencial, lo que es una muestra de que para este algoritmo, las comunicaciones no son el factor principal de consumo de tiempo, esto debido a que la carga comunicacional de la descomposición QR es baja y, aunque necesita coordinación, esta ocurre sólo entre pocos PEs al mismo tiempo.

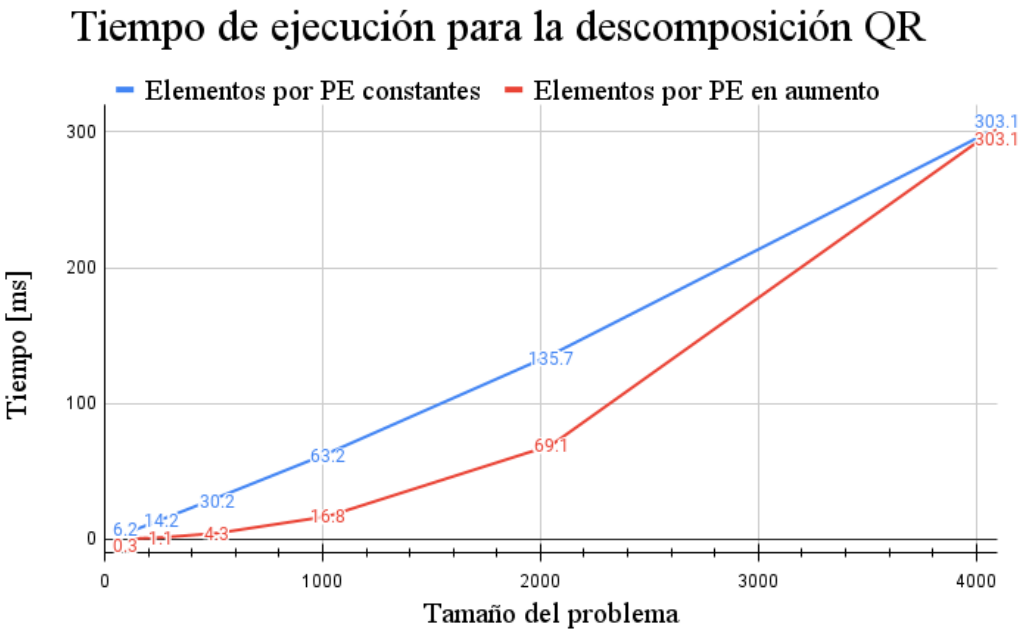


Figura 4.1: Tiempo de ejecución de la descomposición QR en función del tamaño del problema para las distintas cantidades de elementos por PE.

La Figura 4.2 muestra el tiempo de ejecución de la descomposición LU, donde se observa que el crecimiento del tiempo del caso de elementos por PE constantes presenta un comportamiento más cercano a una curva exponencial en comparación a la descomposición QR.

Lo anterior es debido a que, aunque la carga comunicacional sea similar en ambas descomposiciones, la descomposición LU necesita una mayor coordinación en sus comunicaciones debido a la diferencia de carga de las operaciones a realizar entre los emisores y receptores de los mensajes.

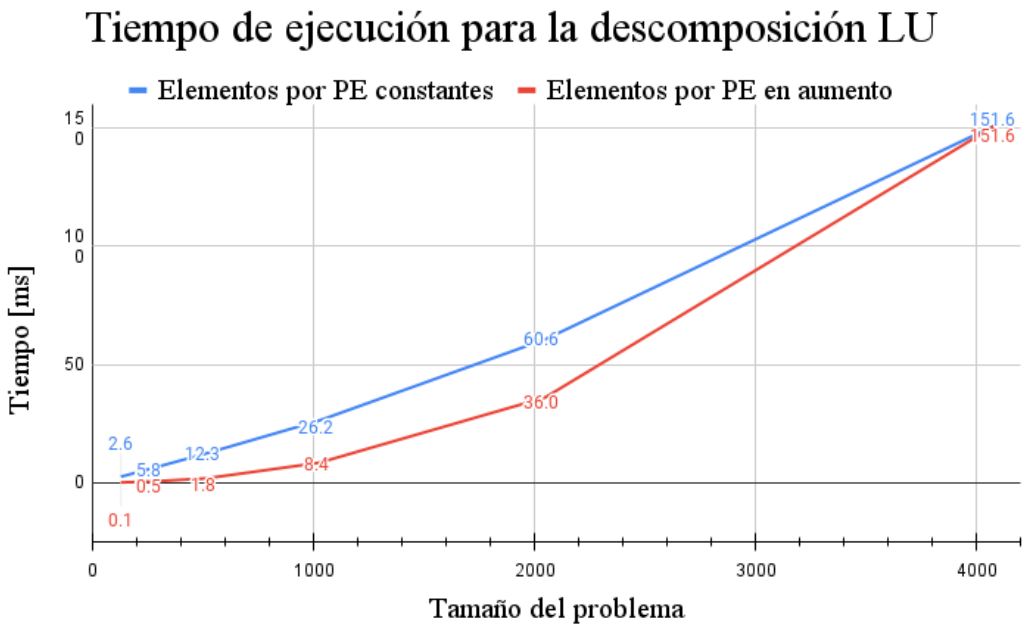


Figura 4.2: Tiempo de ejecución de la descomposición LU en función del tamaño del problema para las distintas cantidades de elementos por PE.

La Figura 4.3 muestra el tiempo de ejecución del algoritmo de Cannon, donde se puede observar que el crecimiento del tiempo en función del problema sigue una clara tendencia exponencial para el caso de elementos por PE constantes, lo que es un claro indicio de que las comunicaciones son un factor sumamente importante en este algoritmo, dado que la coordinación necesaria es mucho más alta que en los demás algoritmos, por lo que optimizarlas tiene el potencial de mejorar ampliamente el rendimiento del mismo.

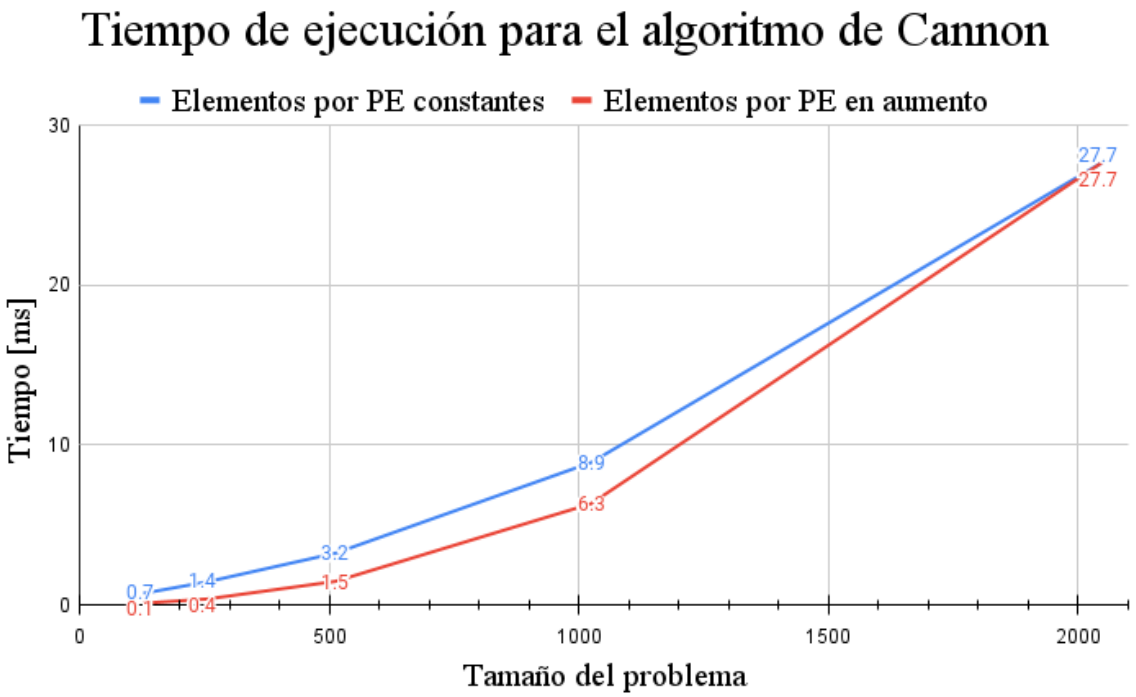


Figura 4.3: Tiempo de ejecución de el algoritmo de Cannon en función del tamaño del problema para las distintas cantidades de elementos por PE.

Para los tres algoritmos el caso de elementos por PE en aumento presenta un crecimiento exponencial en el tiempo de ejecución, esto es lo esperado debido a que la cantidad de elementos por PE y, por consecuencia, la cantidad de cálculos que deben realizar, crece de manera exponencial junto con el problema. Por otra parte, el caso de elementos por PE constantes deja a un lado las consecuencias del aumento de elementos en los PEs y muestra cómo afecta la carga comunicacional del algoritmo y la necesidad de coordinación, dejando en claro que el aumento de estas variables tiene un gran impacto en el coste en tiempo de los algoritmos.

Utilizando la cantidad de flops que necesita cada algoritmo junto con la cantidad de ciclos obtenidos de las ejecuciones de los códigos en el simulador es posible construir gráficos que muestren cómo cambia la cantidad de flops por ciclo de reloj en función del tamaño del problema. La Figura 4.4 muestra este cambio para la descomposición QR, donde se puede observar que las prestaciones del caso donde los PEs tienen una cantidad de elementos en aumento tiene un mejor desempeño que su contraparte, que utiliza menos PEs pero más

memoria de cada uno. La curva de los elementos por PE en aumento tiene una tendencia a decaer en la medida que los PEs se van llenando, esto se debe a que la carga de los PEs va creciendo de manera exponencial mientras todo lo demás se mantiene constante. Por otra parte, el caso de los elementos por PEs constantes muestra una tendencia hacia el crecimiento de manera exponencial, ya que al aumentar la cantidad de elementos a procesar aumenta también la cantidad de PEs, lo que mantiene un equilibrio entre la carga de trabajo y el paralelismo.

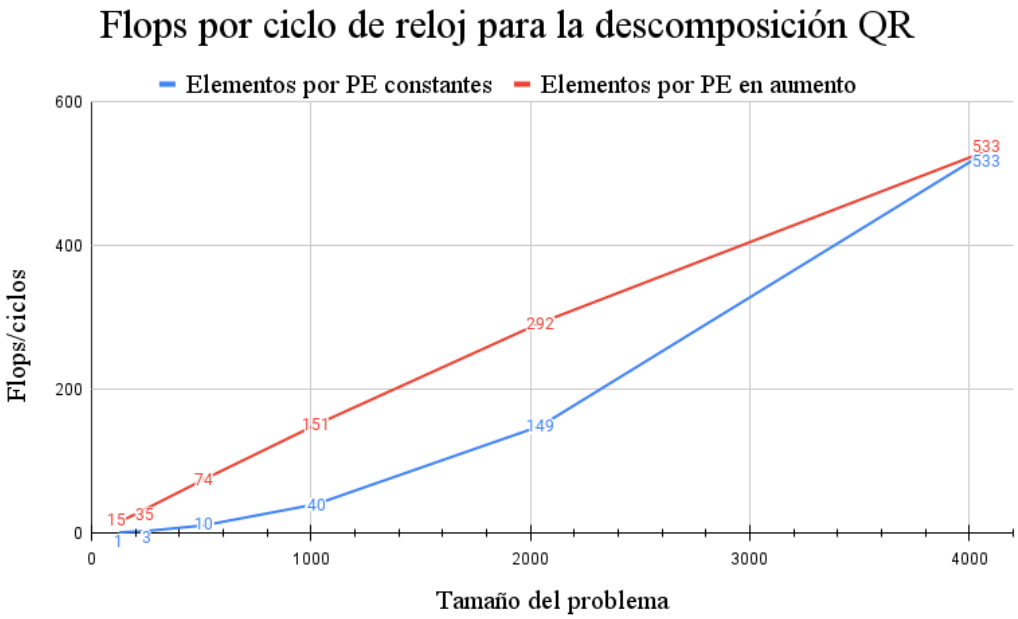


Figura 4.4: Flops por ciclo de reloj para la descomposición QR en función del tamaño del problema para las distintas cantidades de elementos por PE.

La Figura 4.5 muestra los flops por ciclo de reloj para la descomposición LU, en donde es posible observar la misma tendencia de desempeño respecto a que tan llenos se manejan los PEs durante la ejecución del programa. Adicionalmente, la descomposición LU es la que presenta el menor número de flops por ciclo, esto se puede deber a que este algoritmo al realizar olas de computación desde un PE, el tiempo que tarda en escalar el paralelismo no es comparable al de los demás, ya que al compararlo con el algoritmo de Cannon, este último incluso trabajando con un tamaño de problema menor, alcanza el doble de flops por ciclo que la descomposición LU para los tamaños de problema más grandes de cada uno.

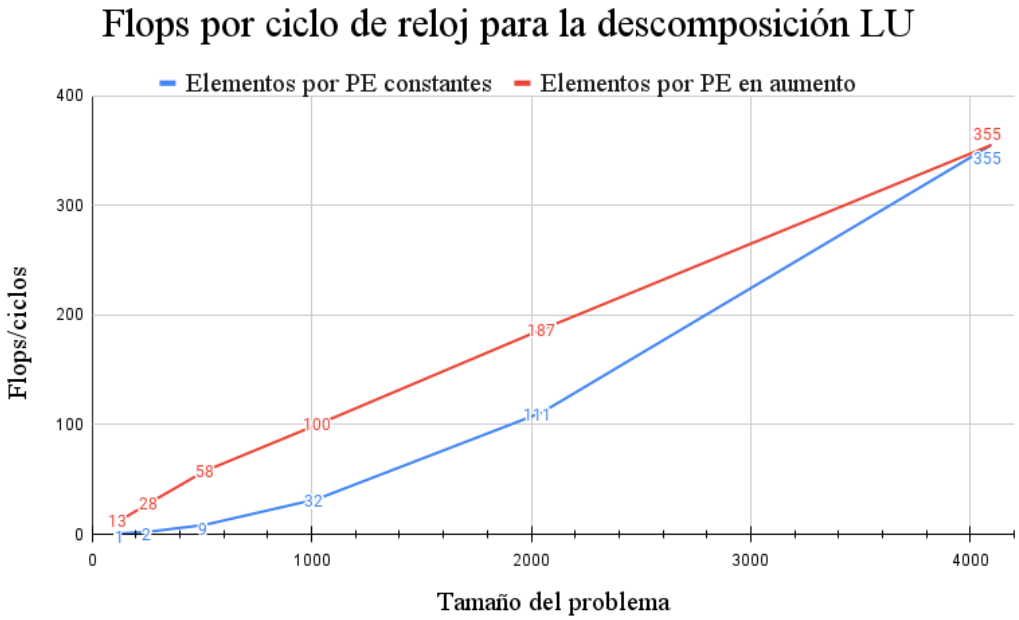


Figura 4.5: Flops por ciclo de reloj para la descomposición LU en función del tamaño del problema para las distintas cantidades de elementos por PE.

La Figura 4.6 muestra los flops por ciclo de reloj para el algoritmo de Cannon. Es posible observar un comportamiento similar en la cantidad de flops por ciclo alcanzada por los tipos de experimentos planteados, pero relevante señalar que la diferencia entre estos es mucho menor en comparación a los demás resultados, lo que indica que aunque los resultados son mejores para el caso de elementos por PE en aumento, este algoritmo presenta características que reducen esa ventaja respecto a los demás. Probablemente esto se deba a la carga comunicacional, la que es mayor para el algoritmo de Cannon no sólo en cantidad, si no que también en distancias, ya que también existe una comunicación directa entre los extremos de la malla. Adicionalmente, el algoritmo de Cannon es el que presenta un aumento más grande en la cantidad de flops por ciclo, ya que incluso trabajando con un tamaño de problema menor alcanza las magnitudes más altas entre los tres algoritmos. Esto es debido a que este algoritmo no reduce la cantidad de PEs activos en la medida que su ejecución avanza, a diferencia de las descomposiciones QR y LU, las que si presentan PEs ociosos.

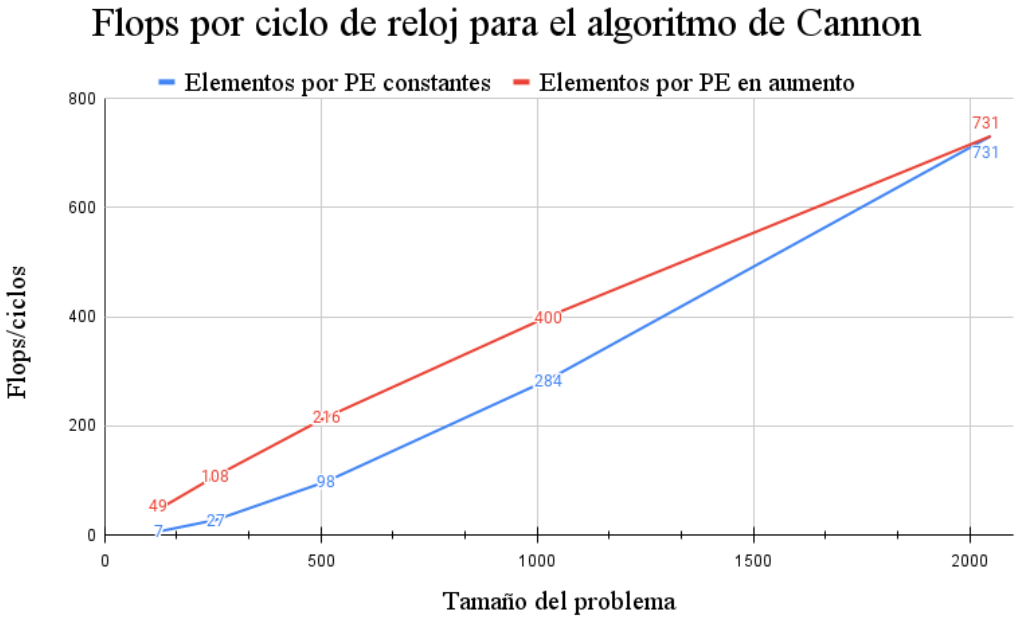


Figura 4.6: Flops por ciclo de reloj para el algoritmo de Cannon en función del tamaño del problema para las distintas cantidades de elementos por PE.

Es transversal a los tres algoritmos que el caso de elementos por PE en aumento logra un mejor desempeño que el caso de elementos por PE constantes. Esta mejora de desempeño, en la mayoría de casos varias veces mejor, indica que aunque los datos deben recorrer una mayor distancia para llegar a sus destinos y pasar por una mayor cantidad de PEs, el paralelismo alcanzado por una mayor cantidad de PEs compensa con creces este costo. Adicionalmente, en la medida que la cantidad de elementos en los PE aumenta, la mejora de las prestaciones tiende a atenuarse, a diferencia del caso de los elementos constantes por PE, donde el aumento de los flops por ciclo presenta un crecimiento exponencial en la medida que el tamaño de la malla aumenta.

El algoritmo de Cannon fue la implementación que alcanzó el número más alto de flops por ciclo de reloj entre los experimentos realizados. Haciendo uso de los datos obtenidos de la experimentación junto con la ecuación 2.4 es posible construir la Tabla 4.2, la cual muestra que el algoritmo de Cannon alcanzó 621 Gflops por segundo. Es importante tener en cuenta que el porcentaje de hardware utilizado es inferior al 5 %, ya que sólo se hace uso de 4,096 PEs de los cerca de 850,000 presentes en la arquitectura.

Tabla 4.2: Cálculo de los Gflops por segundo en las configuraciones donde cada algoritmo alcanzó su mejor rendimiento.

Algoritmo	n	Gflops	Ciclos [M]	Tiempo [s]	Gflops/s
Descomposición QR	4096	137.439	257.650	0.303	533
Descomposición LU	4096	45.813	128.889	0.152	302
Algoritmo de Cannon	2048	17.180	23.514	0.028	621

Capítulo 5

Conclusiones

5.1. Contribuciones y resultados del trabajo

El trabajo realizado logró la correcta implementación de los núcleos computacionales básicos de álgebra lineal correspondientes a la multiplicación de matrices, descomposición LU y descomposición QR en la arquitectura Wafer-Scale Engine, los que componen una base para poder extender a más aplicaciones. Si bien, los algoritmos de la descomposición LU de tipo pipeline y el algoritmo de Cannon se ajustaban bastante bien a la arquitectura, este trabajo propone una versión de la descomposición QR que aprovecha sus propiedades para adaptarla al entorno en el que se encuentra y mejorar el paralelismo que se logra en comparación a hacer uso de la versión clásica de manera directa.

Por otra parte, los experimentos realizados ayudan a entender de mejor manera algunos aspectos de la arquitectura WSE que no son abordados en la documentación, como lo es diferencia de rendimiento en función de la cantidad de elementos por PE, donde fue posible observar que es preferible utilizar una mayor cantidad de PEs pero más vacíos que pocos PEs muy llenos. Adicionalmente se pudo observar el impacto en el tiempo de la carga comunicacional y la necesidad de coordinación de los mensajes, lo que deja en claro que la optimización de las comunicaciones es clave para lograr un mejor rendimiento en la arquitectura.

Finalmente, la experiencia obtenida durante el desarrollo de este trabajo permite también dejar una base para quien desee explorar esta arquitectura, como lo es el código desarrollado para este trabajo. Adicionalmente, detallado en la Sección 5.2, se describe una metodología a seguir para el planteamiento de algoritmos e ideas de cómo enfrentar ciertas problemáticas propias de la arquitectura y su modelo comunicacional, así como consejos que ayudan a facilitar el desarrollo y conocer de antemano las limitaciones que presenta el sistema, lo que permitirá abordar proyectos relacionados de mejor manera.

5.2. Sugerencias para el desarrollo en la arquitectura WSE

Luego de haber realizado la implementación de los algoritmos descritos en este capítulo en la arquitectura WSE, hay aprendizajes útiles que pueden ayudar a quien desee extender lo expuesto en este escrito, o bien, realizar nuevas implementaciones. A continuación se detallan los desafíos y problemáticas que surgieron durante el desarrollo de este trabajo, así como sugerencias desprendidas de estas experiencias.

5.2.1. Diseño de comunicaciones

Uno de los apartados más complejos de trabajar en esta arquitectura es que la manera en que los componentes del programa se comunican es diferente de las arquitecturas más tradicionales, por lo que el diseño de las comunicaciones entre los PEs es sumamente importante para poder realizar un buen programa. Se recomienda realizar el diseño de una manera que sea cómoda y fácil de entender previo al inicio de la escritura del código de los programas, por ejemplo, para este trabajo, todas las figuras que detallan el diseño y flujo de las comunicaciones se realizaron antes de escribir el código, lo que ayudó a visualizar cómo segmentar los mensajes y dar un orden a la forma en que debía comportarse cada PE. Estos diseños no siempre serán perfectos y es probable que necesiten ajustes, pero tener una buena visualización de lo que se busca implementar ayuda a avanzar de mejor manera y detectar posibles errores, dado que el simulador no envía mensajes de error si queda en un punto en el que ya no puede avanzar, como lo son los posibles deadlocks de comunicaciones que se pueden

generar al utilizar comunicaciones sincrónicas.

5.2.2. Comprobación de la documentación

La documentación para el desarrollo en la arquitectura detalla bastante bien todo lo básico para el desarrollo, sin embargo, aunque pocos, hay detalles que no son especificados o no quedan claros con lo que la misma expone. Por ejemplo, una configuración de comunicaciones que se pretendía utilizar en lugar de los switches para los colores que sólo propagaban información, como los colores de eliminación de la matriz LU, eran los filtros que, a grandes rasgos, permite o no el paso de ciertos mensajes al CE basado en condiciones configurables, sin embargo, el filtro que era de utilidad para el trabajo no especificaba que sus parámetros eran enteros sin signo de 16 bits, lo que es insuficiente para las necesidades de los programas, pero sólo fue descubierto al ejecutar casos de prueba suficientemente grandes para rebosar el límite, lo que obligó a reestructurar las comunicaciones de las implementaciones que lo utilizaban. Dado lo anterior, se recomienda que ante la no especificación de algo en la documentación, se hagan pruebas que aclaren lo que faltase en la descripción para evitar situaciones como la mencionada.

5.2.3. Versión del simulador

Durante el desarrollo de este trabajo fueron publicadas dos versiones del simulador, la versión 1.1 y la versión 1.2, sin embargo esta última no fue publicada hasta que los códigos de las implementaciones de las descomposiciones LU y QR estuvieron casi terminados. Hubo un momento de la implementación de la descomposición LU en que el código se quedaba estancado en un rango de tamaño de problema, si este crecía o decrecía fuera de ese rango, el programa funcionaba con normalidad. En ese momento estaba siendo utilizada la versión 1.1 del simulador, la cual no permitía extraer información de los PEs sin que la ejecución terminase correctamente y la forma de depuración disponible no era viable para este caso. Luego de unos días con este problema fue lanzada la versión 1.2 del simulador, se realizó la prueba de ejecutar el código sobre esta versión del simulador y funcionó sin problemas, aunque las

notas de la versión no hiciesen alusión a las características utilizadas por el programa. Por lo anterior se recomienda utilizar la última versión disponible del simulador cuando se desarrolle código CSL. Adicionalmente, la versión 1.2 añadió la posibilidad de que los PEs escriban información en un archivo en tiempo de ejecución, lo que ayuda a detectar más fácilmente cualquier problema.

5.3. Trabajo futuro

Las posibilidades de extensión de este proyecto son amplias, pues una de las finalidades es plantear una base sobre la que construir, lo que plantea tres líneas de trabajo futuro correspondientes a la mejora de lo presentado en este trabajo, extensión de las capacidades de lo desarrollado y la implementación de nuevos elementos.

En cuanto a la mejora de lo presentado en este trabajo, es interesante explorar alternativas que no fueron utilizadas para el desarrollo de los algoritmos, en especial en el área de las comunicaciones entre PEs, algunas de estas son el uso de comunicación asincrónica, la que potencialmente podría mejorar el desempeño de los algoritmos, en especial para el caso del algoritmo de Cannon el cual, debido a las limitaciones de la comunicación sincrónica y la cantidad de matrices temporales utilizadas, debe separar sus comunicaciones en varias partes. De la misma manera es interesante experimentar otras formas de comunicación, por ejemplo, usar más colores en lugar de switches para los casos de las descomposiciones LU y QR con lo que se puede comparar ambas versiones y dejar en claro las ventajas y desventajas que cada opción supone, así como cuando es conveniente cada alternativa.

Respecto a la extensión de las capacidades de lo desarrollado en este trabajo, las descomposiciones LU y QR es donde hay un mayor rango para realizar esto, pues se puede realizar la implementación de la descomposición PA=LU, así como añadirle la capacidad de resolver sistemas de ecuaciones a lo ya implementado, mientras que la descomposición QR, como se mencionó en el desarrollo de este trabajo, tiene versiones que generan distintos elementos adicionales a la matriz R los que son de interés para distintas aplicaciones.

Finalmente, la implementación de nuevos elementos es donde existe mayor libertad para

explorar nuevas oportunidades de trabajo futuro, pues es posible utilizar lo desarrollado en este trabajo como parte de algoritmos que hacen uso de los núcleos computacionales implementados, como en algoritmos de computación gráfica, problemas de mínimos cuadrados, problemas de autovalores, procesamiento de imágenes, entre muchos más. Por otra parte, existen otros algoritmos de álgebra lineal que no tienen relación directa con lo desarrollado en este trabajo, por lo que otra opción de trabajo futuro es desarrollar nuevos núcleos computacionales que amplíen la base entregada por este trabajo.

Bibliografía

[1] *TOP500*. https://top500.org/. (Accessed on 09/12/2024).

[2] Golub, Gene H y Charles F Van Loan: *Matrix computations*. JHU press, 2013.

[3] Grama, Ananth, Ananth Grama, Anshul Gupta, George Karypis y Vipin Kumar: *Introduction to parallel computing [electronic resource]*. Addison-Wesley, Harlow, England;, 2nd ed. edición, 2003, ISBN 0-201-64865-2.

[4] Inc., Cerebras Systems: *Documentation for Developing with CSL — SDK Documentation (1.2.0)*. https://sdk.cerebras.net/. (Accessed on 08/15/2024).

[5] Jacquelin, Mathias, Mauricio Araya-Polo y Jie Meng: *Massively scalable stencil algorithm*, 2022. https://arxiv.org/abs/2204.03775.

[6] Lie, Sean: *Cerebras Architecture Deep Dive: First Look Inside the Hardware/Software Co-Design for Deep Learning*. IEEE Micro, 43(3):18–30, 2023.

[7] Ltaief, Hatem, Yuxi Hong, Leighton Wilson, Mathias Jacquelin, Matteo Ravasi y David Elliot Keyes: *Scaling the "Memory Wall" for Multi-Dimensional Seismic Processing with Algebraic Compression on Cerebras CS-2 Systems*. En *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery, ISBN 9798400701092. https://doi.org/10.1145/3581784.3627042.

[8] Orenes-Vera, Marcelo, Ilya Sharapov, Robert Schreiber, Mathias Jacquelin, Philippe Vandermersch y Sharan Chetlur: *Wafer-Scale Fast Fourier Transforms*. En *Proceedings of the 37th ACM International Conference on Supercomputing*, ICS '23, página 180–191, New York, NY, USA, 2023. Association for Computing Machinery, ISBN 9798400700569. https://doi.org/10.1145/3577193.3593708.

[9] Tramm, John, Bryce Allen, Kazutomo Yoshii, Andrew Siegel y Leighton Wilson: *Efficient Algorithms for Monte Carlo Particle Transport on AI Accelerator Hardware*, 2023. https://arxiv.org/abs/2311.01739.

[10] Woo, Mino, Terry Jordan, Robert Schreiber, Ilya Sharapov, Shaheer Muhammad, Abhishek Koneru, Michael James y Dirk Van Essendelft: *Disruptive Changes in Field Equation Modeling: A Simple Interface for Wafer Scale Engines*, 2022. https://arxiv.org/abs/2209.13768.