UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

El producto matricial distribuido en entornos computacionales de alto rendimiento

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Rodrigo Huerta Gañán

Tutor: Pedro Alonso Jordá

Curso 2019-2020

Resum

La multiplicació de matrius és una operació fonamental per a la resolució d'innombrables problemes matemàtics que tenen a aquesta operació en la seua base. El cost d'aquesta operació, fins i tot l'algorisme utilitzat, és diferent segons la forma particular que tinguen les matrius (denses, diagonals, disperses, ...). En el cas de matrius denses sense estructura coneguda a priori, és a dir, de matrius generals, aquesta operació té un cost computacional elevat ($O(n^3)$). És per aquest motiu pel qual existeixen des de fa temps algorismes paral·lels diversos segons l'entorn de computació particular en el qual s'executen. Entre aquests algorismes té especial rellevància l'algorisme SUMMA, dissenyat per a entorns de *memòria distribuïda*.

En aquest document, en primer lloc, es porta a terme una implementació de l'algorisme SUMMA per a clústers d'ordinadors utilitzant la coneguda llibreria MPI, tenint en compte i aprofitant la capacitat *multicore* dels nodes. En segon lloc, i aprofitant l'experiència anterior, s'implementa el mateix algorisme per a un entorn de computació d'alt rendiment, objectiu real del treball, format per 4 GPUs interconnectades mitjançant una xarxa d'alta capacitat (NVLink) i utilitzant, per a això, la llibreria NCCL de NVIDIA.

Finalment, el treball construeix un embolcall C++ que permet una utilització accessible de la llibreria desenvolupada. A més, s'ofereix una interfície Matlab que proporciona l'habilitat d'utilitzar el programari desenvolupat eficientment a usuaris no programadors. Es demostra l'eficàcia de la solució proposada mitjançant un exemple real: el càlcul de funcions de matrius basat en l'avaluació de polinomis matricials.

Paraules clau: Algorisme SUMMA, MPI, NCCL, C++, BLAS, GEMM, CUDA, CUBLAS, Matlab, Multiplicació de Matrius, Computació Paral·lela, Computació d'altes Prestacions

Resumen

La multiplicación de matrices es una operación fundamental para la resolución de innumerables problemas matemáticos que tienen a esta operación en su base. El coste de esta operación, incluso el algoritmo utilizado, es diferente según la forma particular que tengan las matrices (densas, diagonales, dispersas, ...). En el caso de matrices densas sin estructura conocida a priori, es decir, de matrices generales, esta operación tiene un coste computacional elevado ($O(n^3)$). Es por este motivo por el que existen desde hace tiempo algoritmos paralelos diversos según el entorno de computación particular en el que se ejecutan. Entre estos algoritmos tiene especial relevancia el algoritmo SUMMA, diseñado para entornos de *memoria distribuida*.

En este documento, en primer lugar, se lleva a cabo una implementación del algoritmo SUMMA para clusters de ordenadores utilizando la conocida librería MPI, teniendo en cuenta y aprovechando la capacidad *multicore* de los nodos. En segundo lugar, y aprovechando la experiencia anterior, se implementa el mismo algoritmo para un entorno de computación de alto rendimiento, objetivo real de este trabajo, formado por 4 GPUs interconectadas mediante una red de alta capacidad (NVLink) y utilizando, para ello, la librería NCCL de NVIDIA.

Finalmente, el trabajo construye una envoltura C++ que permite una utilización accesible de la librería desarrollada. Además, se ofrece una interfaz Matlab que proporciona la habilidad de utilizar el software desarrollado eficientemente a usuarios no programadores. Se demuestra la eficacia de la solución propuesta mediante un ejemplo real: el cálculo de funciones de matrices basado en la evaluación de polinomios matriciales.

Palabras clave: Algoritmo SUMMA, MPI, NCCL, C++, BLAS, GEMM, CUDA, CUBLAS, Matlab, Multiplicación de Matrices, Computación Paralela, Computación de Altas Prestaciones

Abstract

Matrix multiplication is a fundamental operation for solving uncountable mathematical problems that have this operation at their base. The cost of this operation, including the algorithm used, is different depending on the particular structure of the matrices (dense, diagonal, sparse, ...). In the case of dense matrices, with no known structure, that is, of general matrices, this operation has a high computational cost ($O(n^3)$). It is for this reason that parallel algorithms have existed for a long time, being different according to the particular computing environment in which they are executed. Among these algorithms it is specially relevant the SUMMA algorithm, designed for *distributed memory* environments.

In this document, firstly, we build an implementation of the SUMMA algorithm for computer clusters using the MPI library, taking into account and exploiting the multicore capacity of the nodes. Secondly, using the previous experience, we implement a version of the same algorithm for another high-performance computing environment, which is the real objective of this work, featuring 4 GPUs interconnected by a high-speed network (NVLink) and using, for this, the NVIDIA NCCL library.

Finally, the work builds a C++ wrapper that allows a handy use of the developed library. In addition, it offers a Matlab interface that provides the ability to use the developed software efficiently for non-programmer users. We demonstrate the effectiveness of the proposed solution on a real example: the computation of matrix functions based on the evaluation of matrix polynomials.

Key words: SUMMA algorithm, MPI, NCCL, C++, BLAS, GEMM, CUDA, CUBLAS, Matlab, Matrix Multiplication, Parallel Computing, High Performance Computing

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

CAPÍTULO 1
Introducción

En este primer capítulo de presentación de la memoria, se introduce al lector en la temática del trabajo de fin de grado, empezando con la sección de motivación, en la que se describe el interés del tema. En el segundo apartado, objetivos, se explica qué se pretende conseguir con este trabajo. A continuación se explicará la metodología empleada y la organización seguida en la memoria.

1.1 Motivación

Siempre me han resultado de gran interés los temas relacionados con el cálculo computacional en *Graphics processing unit*s (GPUs) y de la computación de alto rendimiento. En el futuro me gustaría desarrollar mi carrera profundizando más en estos campos y por ello se ha elegido un trabajo que combina ambas áreas. Además, se han ampliado conocimientos cursados y agregados otros para poder realizar este trabajo, como podría ser el uso de distintas tecnologías de NVIDIA.

Por otro lado, es un reto tecnológico conseguir que el *software* consiga sacar el máximo partido al *hardware*. En este caso concreto se trata de conseguir que se utilicen eficientemente cuatro tarjetas gráficas interconectadas entre ellas con una conexión de alta velocidad. Además, también es una motivación conseguir que cualquier usuario (matemáticos, ingenieros, etc.) sea capaz de utilizar el código desarrollado de forma sencilla.

1.2 Objetivos

La multiplicación de matrices es una operación matemática caracterizada, a la vez, por ser muy utilizada y por tener un alto coste computacional. Por ello, en este trabajo de fin de grado se van a realizar distintas implementaciones del algoritmo *Scalar Universal Matrix Multiplication Algorithm* (SUMMA) [1] en distintos entornos computacionales para mejorar su cálculo.

Los objetivos principales del trabajo son:

- Realizar una primera implementación de aproximación al algoritmo SUMMA para lo que fue ideado, una configuración de memoria distribuida, en nuestro caso compuesta por un cluster de ordenadores *multicore* de propósito general.
- Realizar una segunda implementación del algoritmo SUMMA adaptada a otra configuración de memoria distribuida compuesta, en este caso, por 4 GPUs interconectadas por una red de alto rendimiento.

- Desarrollar una librería en C++ para GPUs compuesta por distintas operaciones matemáticas sobre matrices distribuidas necesarias para realizar el cálculo de funciones de matrices como, por ejemplo, sumas, restas o normas matriciales.
- Adaptar la librería desarrollada al entorno Matlab [2]. En particular, utilizaremos esta librería con objeto de ofrecer una alternativa de cálculo de funciones de matrices como las que muestra el artículo [3] que sea, al mismo tiempo, eficiente y accesible para ingenieros y científicos.

Los objetivos anteriores se han perseguido con la mente puesta en un objetivo secundario, pero no menos importante, que ha consistido en facilitar la comprensión del código, su fácil distribución, mantenimiento y extensión. Para ello, el software de este trabajo ha sido desarrollado en lenguaje C++ [4]. Para facilitar la compilación a los usuarios entre distintos sistemas se ha utilizado la herramienta CMake [5]. Para su mantenimiento, se ha utilizado un repositorio Git alojado en Github que se puede encontrar en [6].

1.3 Metodología y organización de la memoria

La metodología seguida ha pasado por trabajar, en primer lugar, en un clúster con una arquitectura homogénea (nodos iguales) compuesta por procesadores de propósito general. En particular, hemos tenido en cuenta el hecho de que estos procesadores son *multicore*. Posteriormente, se ha pasado a trabajar sobre un sistema moderno bien distinto, pero que cae también en la categoría de sistema paralelo de *memoria distribuida*, que cuenta con un conjunto de tarjetas gráficas interconectadas por una red de altas prestaciones.

Una vez desarrollados los algoritmos se ha pasado a la fase de evaluación. En base a los resultados experimentales que permitan comparar los beneficios obtenidos de una versión paralela del algoritmo frente a una alternativa secuencial, se ha realizado la comparación con el resultado teórico esperado y, en su caso, se ha procedido a refinar la implementación con objeto de aproximar teoría y práctica. Este ciclo se ha repetido un número de veces suficiente como para validar la solución desde el punto de vista de las prestaciones.

En cuanto a la distribución del trabajo seguido para la consecución de los distintos objetivos planteados en la Sección 1.2, se ha pasado por analizar, en primer lugar, el estado del arte en el Capítulo 2, estudiando cómo se puede mejorar la operación de multiplicar matrices en varias GPUs de forma distribuida, tal y como se explica en el Capítulo 3, dedicado a los fundamentos del algoritmo SUMMA. A continuación, se ha optado por realizar una aproximación inicial mediante una tecnología, que ya era familiar, como Message Passing Interface (MPI) [7], tal y como se puede apreciar en el Capítulo 4. Una vez con los conceptos del algoritmo SUMMA [1] asimilados, se ha proseguido a la implementación en tarjetas gráficas, lo que se puede consultar en el Capítulo 5. En el capítulo posterior (Capítulo 6) se explica la implementación de otras operaciones matemáticas que ha sido necesario programar para, de esa forma, conseguir satisfactoriamente el objetivo de utilizar la librería en casos de estudio prácticos como lo es el del cálculo de funciones de matrices. Además, en ese mismo capitulo se explica cómo puede el usuario interaccionar con el programa creado en su lenguaje nativo (C++), instalar la librería mediante CMake y en el Capítulo 7 su integración con Matlab. Tras haber realizado todo el desarrollo se pasa a exponer las conclusiones en el Capítulo 8 y las posibles ampliaciones futuras de este trabajo fin de grado.

CAPÍTULO 2
Entorno de trabajo

A lo largo de este capítulo se va a explicar el conjunto de elementos hardware y software que interaccionan con el algoritmo SUMMA, cuya implementación figura como uno de los objetivos principales de este trabajo.

2.1 Sistemas de computación utilizados

Clústers de computadores

A lo largo del tiempo, la computación de alto rendimiento ha pasado por diversas etapas y paradigmas de programación. Sin ánimo de ahondar en muchos detalles, podemos resumir algunos aspectos importantes relativos a la computación paralela y de alto rendimiento (*High Performance Computing* o HPC) que se relacionan con este trabajo. Por un lado, encontramos en los centros dedicados a este tipo de cálculo clústers de ordenadores, los cuáles están compuestos por un cierto numero de nodos formados por computadoras independientes que están conectadas entre sí. Este tipo de configuración cae dentro de un conjunto conocido como supercomputadores de *memoria distribuida*, dado que los elementos de cálculo poseen memorias propias independientes y los datos se comparten entre los nodos mediante paso explícito de mensajes. Las redes de interconexión que unen estos nodos (Gogabit, Myrinet, SCI, Infiniband, ...) han evolucionado a lo largo del tiempo haciendo que estas configuraciones para HPC sean cada vez más eficientes y escalables.

Por otro lado, los nodos también han evolucionado integrando procesadores multinúcleo o *multicore*. Esto ha permitido que su potencia aumentara de forma significativa a costa de un paralelismo interno que sigue el modelo de *memoria compartida*. De esta manera, se hace necesario implementar programas paralelos *híbridos*, es decir, que necesitan utilizar ambos modelos de programación para poder aprovechar toda la potencia del clúster.

El clúster utilizado en este trabajo fin de grado, denominado Kahan[1], es un clúster de prácticas del Departamento de Sistemas Informáticos y Computación de la Universitat Politècnica de València. Su composición puede verse en la Figura 2.1. A pesar de que se ven seis nodos en la Figura 2.1, se van a utilizar cuatro debido a que uno de ellos está estropeado y, por conveniencia, necesitamos utilizar un número par de nodos. Cada uno de los nodos de cómputo posee 2 procesadores AMD Opteron(TM) 6272 a 2.1 GHz. con 16 núcleos o *cores* cada uno, ofreciendo un total de 32 *cores*, y 32 GB. de memoria RAM. Con este equipo se estudiará el comportamiento del sistema con diferentes configuracio-

[1] En honor al matemático y científico computacional William Kahan.

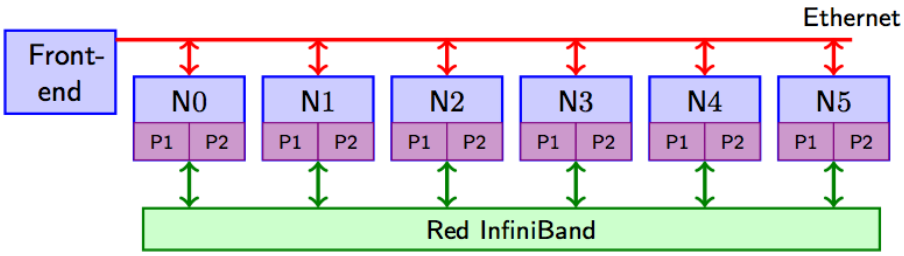


Imagen extraída de [8].

Figura 2.1: Organización de Kahan.

nes que utilicen 128 elementos de proceso (*cores*). Se combinarán los procesos de que se compone la aplicación entre los cuatro nodos con diferentes procesos por nodo y otro número distinto para los *threads* o hilos con los que se ejecutarán las rutinas de la librería Intel MKL, dado que esta librería soporta ejecuciones multi-núcleo de sus rutinas. La finalidad de los procesos se configura lanzando trabajos mediante un sistema *Terascale Open-source Resource and Queue Manager* (TORQUE) al Kahan. Se puede encontrar más información sobre este clúster en [8].

La librería de paso de mensajes MPI

Message Passing Interface (MPI) es una especificación sobre la cual se implementan diversas librerías como OpenMPI [9] o MPICH [10]. La finalidad de la especificación es conseguir la comunicación entre distintos procesos mediante paso explícito de mensajes con el que poder implementar aplicaciones paralelas orientadas a arquitecturas de *memoria distribuida*. Para conseguir que se comuniquen entre sí los procesos, es necesario un comunicador, el cual es un elemento que se encarga de conectar grupos de procesos entre sí. Dicha especificación cuenta con rutinas que permiten:

- realizar comunicaciones punto a punto entre procesos,
- realizar comunicaciones colectivas de reparto o recolección, o
- gestionar topologías para establecer una clasificación entre procesos gestionando comunicadores para así crear subconjuntos de procesos.

Tarjetas Gráficas

De unos años a esta parte, los computadores han aumentado su capacidad de cómputo gracias a la incorporación de tarjetas gráficas (entre otros dispositivos) que han visto ampliado el rango de aplicaciones para las que pueden utilizarse a prácticamente cualquier aplicación de propósito general que requiera de una gran cantidad de cómputo, en lugar de usarse únicamente para el cálculo de gráficos por ordenador. Este camino fue iniciado por la compañía NVIDIA que, para aprovechar la tremenda capacidad computacional de sus GPUs, dio lugar a su conocida Compute Unified Device Architecture (CUDA) [11]. Aunque hoy en día existen otras opciones, esta compañía sigue siendo puntera en el mercado las GPUs empleadas para propósito general o *General Purpose Graphics processing unit* (GPGPU). Una opción alternativa, y bien conocida también, para programar dispositivos de diversa arquitectura e integrarlos en la resolución del mismo problema computacional es *Open Computing Language* (OpenCL) [12]. Gracias a este tipo de computación, conocida como *computación paralela heterogénea*, es posible emplear las tarjetas grá-

ficas para, por ejemplo, realizar cálculos matemáticos intensivos como la multiplicación de matrices.

Un paso evidente ha sido el de unir las GPUs a los nodos dando lugar a los clústers híbridos o heterogéneos. Es difícil encontrar hoy en día un clúster de alto rendimiento que no incorpore nodos con uno o más "aceleradores" como GPUs. Véase, por ejemplo, la lista *TOP500* [13] en su actualización de Junio 2020, que muestra los clústers de supercomputación más potentes del mundo ordenados por rendimiento. Sin embargo, en este trabajo no vamos a tratar con este tipo de configuración.

Concretamente, las tarjetas gráficas utilizadas en este trabajo son cuatro NVIDIA Tesla P100-SXM2. Estas tarjetas, de arquitectura *Pascal*, poseen 3584 CUDA *cores* cada una y 16GB. HBM2 de memoria principal. La versión del driver es la 10.2, la última hasta el momento. Las 4 GPUs están interconectadas entre sí mediante una red de interconexión de altas prestaciones de tecnología NVLink [14], que se explica en el apartado siguiente.

Comunicación de datos entre tarjetas gráficas

Las GPUs que suelen utilizarse para GPGPU suelen encontrarse conectadas al equipo o *host* mediante *Peripheral Component Interconnect* (PCI). Sin embargo, existe, además, un protocolo de comunicaciones desarrollado por NVIDIA y denominado NVLink [14], que sirve como tecnología de interconexión entre distintas tarjetas gráficas existentes en un equipo. NVLink especifica conexiones punto a punto pudiendo llegar a los $25GB/s$ de velocidad de transferencia entre dos GPUs. La máxima velocidad posible en la configuración utilizada en este trabajo (NVLink v1) que se puede obtener es de $20GB/s$ por enlace. Cada GPU dispone de 4 enlaces que, conectados a las otras GPUs existentes en el equipo, dan lugar a distintas topologías de interconexión posibles. Para el caso concreto que nos ocupa, que es el sistema utilizado en este trabajo, contamos con la topología que se puede observar en la Figura 2.2.

El sistema que cuenta con diferentes tarjetas gráficas utilizado durante este trabajo de fin de grado es el nodo conocido con el nombre de Nowherman, perteneciente al Instituto de Telecomunicaciones y Aplicaciones Multimedia (ITEAM) de la Universitat Politècnica de València. Este sistema cuenta con las siguientes características:

- Una CPU Intel Xeon E5-2698.
- Cuatro GPUs NVIDIA Tesla P100 SMX2 16GB HBM2 con soporte para NVLINK.
- 504 GB DDR4 de *Random Access Memory* (RAM).
- Sistema operativo Ubuntu 16.04.6.

La interfaz software para realizar las comunicaciones de datos entre GPUs puede ser la NVIDIA Collective Communications Library (NCCL) [15], una librería de comunicaciones entre GPUs, normalmente utilizada junto con NVLink aunque puede ser utilizada también sobre PCI. Esta librería implementa primitivas de comunicación colectivas multi-GPU optimizadas tales como *all-gather*, *all-reduce*, etc. En este trabajo utilizaremos esta librería, particularmente, la versión 2.6.4.

En [16] puede verse una comparativa con gran detalle de rendimientos en función de la topología de comunicaciones utilizada y de la tecnología y librerías utilizadas (NVLink v1/v2[2], NCCL v1/v2 e InfiniBand). Se puede observar que el número de enlaces varía dependiendo del sistema que se esté utilizando.

[2]En el momento de escribir este documento se acaba de hacer pública la versión 3 que cuenta con mejoras de rendimiento y un aumento de enlaces entre tarjetas gráficas.

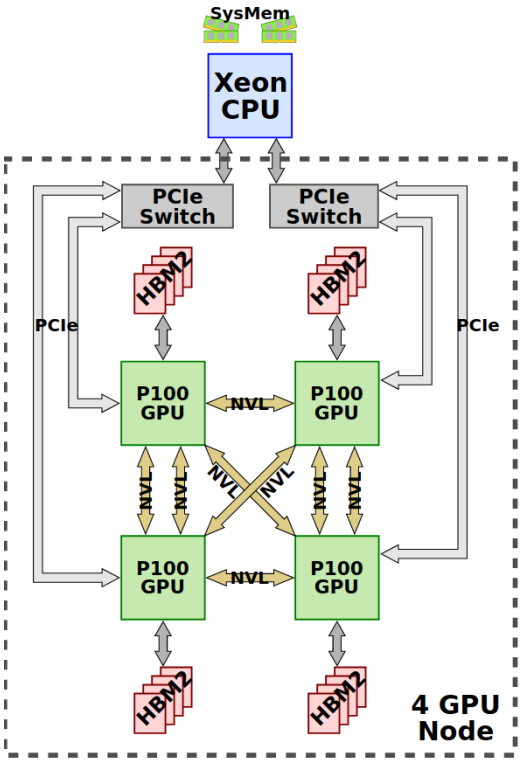


Imagen extraída de https://wikichip.org.

Figura 2.2: Topología de NVLink empleada.

2.2 Software para la multiplicación de matrices

Como es bien sabido, la multiplicación de matrices es una operación que se caracteriza tanto por su utilidad como por su coste computacional ($O(2n^3)$ [17] para matrices densas de orden n). Esta operación ha quedado completa y perfectamente especificada por la operación denominada *General matrix multiply* (GEMM) (en el caso de matrices densas) en la librería Basic Linear Algebra Subprograms (BLAS) [18] que, más que una librería es, en realidad, una colección de especificaciones de interfaces de las operaciones básicas de álgebra lineal numérica que en ella se describen. Esta interfaz está universalmente admitida hoy en día y, por tanto, es seguida por los fabricantes de procesadores para ofrecer implementaciones eficientes de la misma para sus dispositivos. Estas implementaciones suelen incorporar las últimas y más eficientes herramientas y opciones de dichos procesadores como, por ejemplo, las instrucciones *Single Instruction, Multiple Data* (SIMD), que redundan en una aceleración de cálculo notable, especialmente para el caso de la multiplicación de matrices. Esta operación también se benefició en su día de la aparición de los procesadores multi-núcleo, ya que la operación tiene un alto índice de rendimiento por dato leído de memoria y, por tanto, resulta relativamente sencillo obtener una versión paralela eficiente en estos procesadores. Implementaciones optimizadas podemos nombrar, por ejemplo, la que incorpora la Intel *Math Kernel Library* (MKL) [19] para sus procesadores (o compatibles x86) y que se va a utilizar en este trabajo de fin de grado. La versión de Intel MKL usada en el *cluster* Kahan es la 2012, en cambio, la utilizada en Nowherman es la 2019. También, para las GPUs de NVIDIA, esta compañía implementa su propia versión de BLAS, denominada cuBLAS [20], y que también se empleará en este trabajo, en particular, la versión distribuida con el kit de desarrollo CUDA 10.2.

CAPÍTULO 3

El algoritmo de multiplicación de matrices distribuidas SUMMA

En este trabajo partimos de la operación de multiplicación de matrices

$$C = A \times B, \tag{3.1}$$

para $C \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{m \times k}$ y $B \in \mathbb{R}^{k \times n}$. El coste de esta operación es $O(2mnk)$ FLOPs[1], o $O(2n^3)$ FLOPs si las matrices son cuadradas ($m = n = k$).

Uno de los algoritmos más conocidos para la multiplicación de matrices en entornos distribuidos de CPUs es el algoritmo SUMMA [1]. Este algoritmo puede implementarse, por ejemplo, utilizando MPI.

El algoritmo consiste en particionar las matrices en bloques para distribuirlos entre los distintos procesos. Estos procesos serán los encargados de realizar las multiplicaciones de bloques locales junto a bloques recibidos de otros procesos. La operación resultante es muy parecida a la realización del producto "exterior" de dos matrices por bloques, una de las diferentes formas de realizar la multiplicación. Vamos a asumir que las matrices en (3.1) son cuadradas, están particionadas en una malla de 4×4 bloques cuadrados, y cada proceso, de un total de 16, contiene su bloque correspondiente de A, B y C,

$$C = A \times B = \begin{pmatrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} & B_{02} & B_{03} \\ B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{pmatrix}$$

entonces, la multiplicación matricial anterior puede expresarse de la siguiente forma:

$$\begin{pmatrix} A_{00} \\ A_{10} \\ A_{20} \\ A_{30} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} & B_{02} & B_{03} \end{pmatrix} + \begin{pmatrix} A_{01} \\ A_{11} \\ A_{21} \\ A_{31} \end{pmatrix} \times \begin{pmatrix} B_{10} & B_{11} & B_{12} & B_{13} \end{pmatrix} +$$
$$\begin{pmatrix} A_{02} \\ A_{12} \\ A_{22} \\ A_{32} \end{pmatrix} \times \begin{pmatrix} B_{20} & B_{21} & B_{22} & B_{23} \end{pmatrix} + \begin{pmatrix} A_{03} \\ A_{13} \\ A_{23} \\ A_{33} \end{pmatrix} \times \begin{pmatrix} B_{30} & B_{31} & B_{32} & B_{33} \end{pmatrix} =$$
$$\begin{pmatrix} A_{00} \cdot B_{00} & A_{00} \cdot B_{01} & A_{00} \cdot B_{02} & A_{00} \cdot B_{03} \\ A_{10} \cdot B_{00} & A_{10} \cdot B_{01} & A_{10} \cdot B_{02} & A_{10} \cdot B_{03} \\ A_{20} \cdot B_{00} & A_{20} \cdot B_{01} & A_{20} \cdot B_{02} & A_{20} \cdot B_{03} \\ A_{30} \cdot B_{00} & A_{30} \cdot B_{01} & A_{30} \cdot B_{02} & A_{30} \cdot B_{03} \end{pmatrix} + \begin{pmatrix} A_{01} \cdot B_{10} & A_{01} \cdot B_{11} & A_{01} \cdot B_{12} & A_{01} \cdot B_{13} \\ A_{11} \cdot B_{10} & A_{11} \cdot B_{11} & A_{11} \cdot B_{12} & A_{11} \cdot B_{13} \\ A_{21} \cdot B_{10} & A_{21} \cdot B_{11} & A_{21} \cdot B_{12} & A_{21} \cdot B_{13} \\ A_{31} \cdot B_{10} & A_{31} \cdot B_{11} & A_{31} \cdot B_{12} & A_{31} \cdot B_{13} \end{pmatrix} +$$
$$\begin{pmatrix} A_{02} \cdot B_{20} & A_{02} \cdot B_{21} & A_{02} \cdot B_{22} & A_{02} \cdot B_{23} \\ A_{12} \cdot B_{20} & A_{12} \cdot B_{21} & A_{12} \cdot B_{22} & A_{12} \cdot B_{23} \\ A_{22} \cdot B_{20} & A_{22} \cdot B_{21} & A_{22} \cdot B_{22} & A_{22} \cdot B_{23} \\ A_{32} \cdot B_{20} & A_{32} \cdot B_{21} & A_{32} \cdot B_{22} & A_{32} \cdot B_{23} \end{pmatrix} + \begin{pmatrix} A_{03} \cdot B_{30} & A_{03} \cdot B_{31} & A_{03} \cdot B_{32} & A_{03} \cdot B_{33} \\ A_{13} \cdot B_{30} & A_{13} \cdot B_{31} & A_{13} \cdot B_{32} & A_{13} \cdot B_{33} \\ A_{23} \cdot B_{30} & A_{23} \cdot B_{31} & A_{23} \cdot B_{32} & A_{23} \cdot B_{33} \\ A_{33} \cdot B_{30} & A_{33} \cdot B_{31} & A_{33} \cdot B_{32} & A_{33} \cdot B_{33} \end{pmatrix}.$$

[1] Operaciones en coma flotante (Floating Point operationS).

Un detalle esencial a considerar es que los procesos no tienen todos los datos. Para conseguir la información necesaria para realizar cada operación en el momento adecuado será necesario el uso de comunicaciones colectivas de tipo *broadcast*. Si observamos la ecuación anterior, veremos que cada proceso necesita recibir dos bloques para llevar a cabo la multiplicación local, un bloque de su misma fila y otro de su misma columna. Una vez recibidos los bloques, es fácil ver que el bloque correspondiente de C se calcula como:

$$C_{ij} = C_{ij} + A_{ik} \times B_{kj} \ .$$

Por ejemplo, el proceso P_{ij} recibirá el bloque A_{i0} desde el proceso P_{i0} (misma fila i), tal como se aprecia en la Figura 3.1. El mismo proceso recibirá el bloque B_{0j} desde el proceso P_{0j} (misma columna i), tal y como se ve en la Figura 3.2.

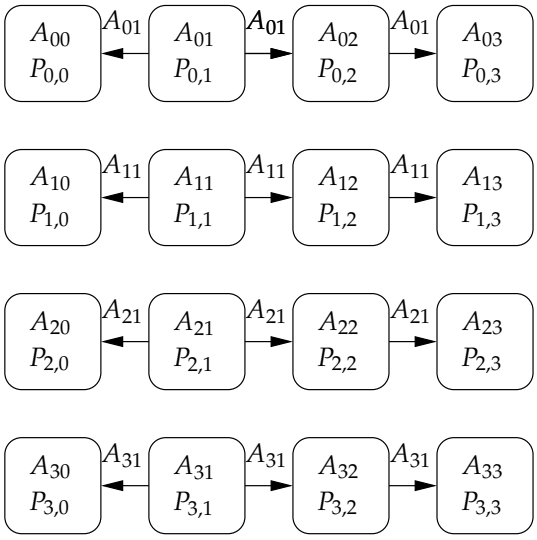


Figura 3.1: Difusión de la columna en la segunda iteración del algoritmo.

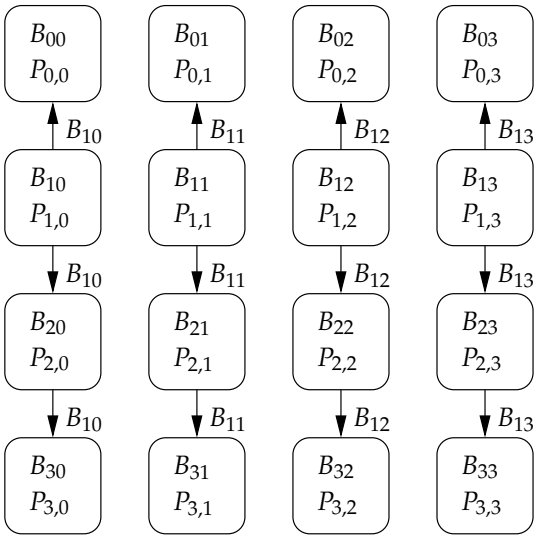


Figura 3.2: Difusión de la fila en la segunda iteración del algoritmo.

En el ejemplo mostrado en las figuras, el proceso P_{21} realizaría el siguiente cálculo:

$$C_{21} = A_{20} \times B_{01} + A_{21} \times B_{11} + A_{22} \times B_{21} + A_{23} \times B_{31} \ .$$

El número de veces que se realizarán estos pasos será el número de columnas (o filas) que tenga la malla de procesos, que coincide con el número de sumandos de la ecuación anterior. Por tanto, cada bloque C_{ij} se habrá actualizado de la siguiente manera

$$C_{ij} = \sum_{k=0}^{n/b-1} A_{ik} \times B_{kj} \, ,$$

siendo b el tamaño de bloque. Por simplicidad asumiremos en lo sucesivo que n es múltiplo de b.

El algoritmo SUMMA podría especificarse como muestra el Algoritmo 1. Las operaciones de difusión o "broadcast" implican implícitamente que el bloque a enviar es efectivamente enviado por el proceso que lo contiene y los demás procesos lo reciben.

Algoritmo 1 Esquema del algoritmo SUMMA.

Sea n el tamaño de la matriz y b el tamaño de bloque, entonces, el proceso P_{ij}, $i = 0, \ldots, n/b-1$, ejecuta
for $k = 0$ **to** $n/b-1$ **do**
1. Difundir el bloque A_{ik} a toda la fila i de procesos
2. Difundir el bloque B_{kj} a toda la columna j de procesos
3. $C_{ij} = C_{ij} + A_{ik} \times B_{kj}$

end

La eficiencia del algoritmo SUMMA es asintóticamente del 100 %. Suponiendo que las matrices a multiplicar son cuadradas de dimensión n y que p es el número de procesadores (o nodos del cluster) y p es potencia de 2, entonces, según [1], la Ecuación 3.2,

$$S(n,p) = \frac{2n^3\gamma}{\frac{2n^3}{p}\gamma + n\log(p)\alpha + \log(p)\frac{n^2}{\sqrt{p}}\beta} = \frac{p}{1 + \frac{p\log(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}\log(p)}{2n}\frac{\beta}{\gamma}}, \tag{3.2}$$

muestra la aceleración o *speed-up* de dicho algoritmo. Esta ecuación asume que el coste de enviar un mensaje de n elementos entre dos nodos es $\beta + \alpha n$, siendo β el tiempo de establecer la comunicación (*startup* time) y α el tiempo de transferir un elemento. El parámetro γ representa el coste temporal de un FLOP. La eficiencia queda como expresa la Ecuación 3.3,

$$E(n,p) = \frac{S(n,p)}{p} = \frac{1}{1 + \mathcal{O}(\frac{p\log(p)}{n^2}) + \mathcal{O}(\frac{\sqrt{p}\log(p)}{n})}. \tag{3.3}$$

Lo importante de la Ecuación 3.3 es que, ignorando el término $\log p$, puesto que crece muy lentamente con p, el algoritmo SUMMA es escalable, es decir, si incrementamos p, n debe crecer con $\sqrt{p}$ para mantener la eficiencia. Puede consultarse la demostración de esto en [1].

CAPÍTULO 4

Solución del problema en clústers de computadores *multicore*

Antes de proceder a la implementación de la librería final, se ha decidido que sería conveniente escribir un boceto en una tecnología familiar como la librería MPI de la operación comúnmente llamada $C = A \cdot B$. La implementación realizada en este capítulo, así como en el resto del trabajo, ha sido realizada en el lenguaje de programación C++, cumpliendo con el estándar C++11. El objetivo de ello es poder tener una base de conocimiento y código antes de diseñar la versión final en NCCL, además de estudiar el comportamiento de este algoritmo en un clúster multi-núcleo y obtener una versión eficiente.

4.1 Cálculo de las propiedades de la operación

Un tema esencial en la realización de la multiplicación de forma distribuida es obtener cómo se van a distribuir las matrices que se van a ver implicadas en la operación.

Para que los bloques de una matriz puedan multiplicarse con otra debe cumplirse que las dimensiones de estos bloques respeten la relación $M \times N \cdot N \times K$, es decir, el número de columnas de la primera matriz debe coincidir con el número de filas de la segunda.

Para poder respetar esta propiedad puede que sea necesario extender los bloques de la parte inferior, de la parte derecha o ambas partes de la matriz. La forma de extender un bloque es mediante filas o columnas de 0's, las cuales no afectan al resultado final. Todos estos valores se calculan a partir de las dimensiones de la malla de procesos. En la Figura 4.3 se puede ver un ejemplo de cómo queda la matriz extendida y dividida en bloques para una malla de 2×4 procesos.

4.2 Almacenamiento de matrices

Para la programación de ambas librerías se ha optado por un modelo que almacena los elementos de la la matriz de forma contigua en memoria. Dentro de esta forma de guardar las matrices en RAM existen dos variantes:

- La primera es *Row major order* y ha sido la elegida para esta primera aproximación debido a que es una forma más familiar de pensar en su representación.

- La segunda es *Column major order* y ha sido la escogida para el desarrollo en Solución del problema en GPUs (Capítulo 5) porque la librería cuBLAS usada para el cálculo de GEMM solo trabaja con esta variante.

Hay que destacar que una parte crucial del algoritmo es la división de las matrices en distintos bloques. Los atributos más destacables de estos bloques son su tamaño y el índice global de la primera posición de cada bloque. Esta última varía dependiendo de cómo se guarde la información de las matrices en memoria. Las alternativas que existen se explicarán a continuación junto con las figuras que ilustrarán gráficamente los modos de almacenamiento y qué posición de memoria tiene cada elemento de la matriz. Es importante señalar que la matriz A, a la hora de distribuirse por primera vez, tendrá un bloque por rango y ambos números serán iguales.

Para ilustrar un ejemplo de ocho procesos en una configuración de 2×4 se ha procedido a numerar los bloques de la siguiente forma:

$$\left(\begin{array}{c|c|c|c} 0 & 1 & 2 & 3 \\ \hline 4 & 5 & 6 & 7 \end{array}\right)$$

Figura 4.1: Numeración de los bloques.

Como consecuencia de que existan bloques con tamaño no cuadrado, sucede que la matriz B pueda tener una distribución de bloques y rangos de la siguiente forma:

0 | 0	1 | 1	2 | 2	3 | 3
4 | 4	5 | 5	6 | 6	7 | 7
8 | 0	9 | 1	10 | 2	11 | 3
12 | 4	13 | 5	14 | 6	15 | 7

número de bloque | rango CPU

Figura 4.2: Distribución de bloques no cuadrados en la matriz B.

Row major order

En este modelo se guardan los elementos de las filas de forma contigua en memoria tal y como se muestra en la Figura 4.3. En dicha figura se ha decidido que se va a subdividir la matriz de tamaño 7×7 en ocho bloques. El resultado es una malla de 2×4.

$$\left(\begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 10 & 11 & 12 & 13 & 0 \\ 14 & 15 & 16 & 17 & 18 & 19 & 20 & 0 \\ 21 & 22 & 23 & 24 & 25 & 26 & 27 & 0 \\ \hline 28 & 29 & 30 & 31 & 32 & 33 & 34 & 0 \\ 35 & 36 & 37 & 38 & 39 & 40 & 41 & 0 \\ 42 & 43 & 44 & 45 & 46 & 47 & 48 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

Elementos extendidos representados como `0` (relleno o *padding*).

Figura 4.3: Matriz de índices *Row major order*.

Por tanto, el primer índice de cada bloque es 0, 2, 4, 6, 28, 30, 32 y 34 de forma respectiva. Se puede observar que es de gran importancia encontrar una forma de hallar esos índices. Tras analizar detenidamente el problema, se ha deducido el siguiente código:

Algoritmo 2 Cálculo de índices de bloque en *Row major order*.

```
//i indica el número de bloque
int posRowBelong = (i / meshColumnSize) * columnasMatrizGrande * blockRowSize;
int posColumnBelong = (i % meshColumnSize) * blockColumnSize;
int rowMajorOrderIndex =+ posRowBelong + posColumnBelong;
```

En cambio, si lo deseado es acceder a una posición concreta de la matriz lo podremos hacer mediante el siguiente cálculo:

Algoritmo 3 Cálculo de una posición específica de la matriz en *Row major order*.

```
int position = rowDesired * numberOfColumns + columnDesired;
```

Column major order

Al contrario que en el anterior modelo aquí se guardan los elementos de las columnas de forma contigua. La Figura 4.4 muestra el almacenamiento para la misma matriz anterior de tamaño 7×7.

$$\left(\begin{array}{cccccccc} 0 & 7 & 14 & 21 & 28 & 35 & 42 & 0 \\ 1 & 8 & 15 & 22 & 29 & 36 & 43 & 0 \\ 2 & 9 & 16 & 23 & 30 & 37 & 44 & 0 \\ 3 & 10 & 17 & 24 & 31 & 38 & 45 & 0 \\ \hline 4 & 11 & 18 & 25 & 32 & 39 & 46 & 0 \\ 5 & 12 & 19 & 26 & 33 & 40 & 47 & 0 \\ 6 & 13 & 20 & 27 & 34 & 41 & 48 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

Elementos extendidos representados como `0` (relleno o *padding*).

Figura 4.4: Matriz de índices *Column major order*.

Ahora podemos ver que el primer índice de cada bloque ha cambiado siendo 0, 14, 28, 42, 4, 18, 32 y 46, respectivamente. El código resultante capaz de calcular estos índices correctamente, independiente del tamaño de los bloques, es el que se muestra a continuación:

Algoritmo 4 Cálculo de índices de bloque en *Column major order*.

```
1 //i indica el número de bloque
2 int posColumnBelong = (i %numberOfColumnBlocks) * rowsReal * blockColumnSize;
3 int posRowBelong = (i / numberOfColumnBlocks) * blockRowSize;
4 int columnMajorOrderIndex =+ posRowBelong + posColumnBelong;
```

En cuanto al cálculo específico de una posición quedaría de la siguiente forma:

Algoritmo 5 Cálculo de una posición específica de la matriz en *Column major order*.

```
1 int position = columnnDesired * numberOfRows + rowDesired;
```

4.3 Distribución y recuperación de matrices

Una parte de vital importancia en el diseño de la librería es la forma de distribuir y recuperar las matrices. Existen diversas alternativas en MPI para realizar ambas tareas teniendo en cuenta todo el conjunto de instrucciones que existen en la librería y que se pueden consultar en [7].

Las posibilidades para la distribución son:

- Reparto colectivo variado mediante el uso de tipos personalizados.
- Comunicaciones punto a punto con Send para el proceso que contiene la matriz sin distribuir y Recv para el resto[1]. También entran en este ámbito todas las variantes de estas instrucciones.

Las posibilidades para la recuperación son :

- Recolección colectiva mediante uso de tipos personalizados.
- Comunicaciones punto a punto con Send para cada proceso que contenga un bloque de la matriz y Recv para el proceso que almacenará la matriz sin distribuir[1]. También abarca este ámbito a todas las variantes de estas instrucciones.
- Creación de matrices auxiliares del mismo tamaño que la matriz sin distribuir y rellenado a 0 excepto las posiciones que correspondan al proceso. Mediante una operación Reduce de tipo suma, recuperar la matriz sin distribuir en el proceso que se desee.

Tras analizar las instrucciones disponibles en [15] se decidió optar por las variantes de Send y Recv ya que es lo más parecido a pasar información del *host* al dispositivo, es decir, pasar información de la memoria principal del sistema a la de la GPU, lo que será de utilidad en el siguiente capítulo.

[1]En el caso del proceso tenga que enviarse una matriz consigo mismo, se sustituirá Send por memcpy.

4.4 Implementación del algoritmo SUMMA

El diseño del algoritmo para esta versión se basa en la proporcionada en [1] mediante MPI. Sin embargo, se ha decidido no realizar una implementación directa de él debido a que en la librería NCCL [15] no cuenta con rutinas de gestión de topologías disponibles en MPI [7] tales como MPI_Cart_create, MPI_Cart_coords, MPI_Cart_sub o MPI_Comm_split, las cuáles facilitarían enormemente el desarrollo de este código en particular. Estas instrucciones permiten crear grupos de procesos fácilmente. A estas agrupaciones las llamaremos **color**. Este termino, utilizado también en MPI, se usa para agrupar procesos que pertenecen a una misma fila de la malla o a una misma columna.

Debido a la limitación técnica que supone prescindir del manejo de topologías se optó por calcular de forma propia el color de cada proceso de la siguiente forma:

Algoritmo 6 Cálculo del color de un proceso.

```
int rowColor = cpuRank / numberOfColumnBlocks;
int columnColor = cpuRank % numberOfColumnBlocks;
```

Como consecuencia, cada proceso ha debido calcular con quién se debe comunicar en las operaciones de difusión de filas y de columnas.

Algoritmo 7 Obtención de los miembros de un mismo color.

```
int colGroup[meshRowsSize];
int rowGroup[meshColumnsSize];
for (i = 0; i < meshColumnsSize; i++)
    rowGroup[i] = rowColor * meshColumnsSize + i;
for (i = 0; i < meshRowsSize; i++)
    colGroup[i] = columnColor + i * meshColumnsSize;
```

Una vez conseguidos todos esos valores ya somos capaces de crear los comunicadores necesarios para la correcta ejecución del algoritmo original [1].

4.5 Evaluación experimental del algoritmo

Una vez terminada la implementación es hora de comprobar su efectividad. Para ello, se empleó el clúster Kahan, descrito en el Capítulo 2 (Figura 2.1), junto a la librería Intel *Math Kernel Library* (MKL) para la ejecución de las multiplicaciones en cada uno de los procesos mediante su implementación GEMM del estándar BLAS.

Para comprobar el correcto funcionamiento del código implementado se ha decidido que se va a comparar una matriz calculada en secuencial C con la matriz calculada en paralelo $\hat{C}$ obteniendo el error relativo ε_F definido como

$$\varepsilon_F = \frac{||C - \hat{C}||_F}{||C||_F}, \tag{4.1}$$

donde $||C||_F$ representa la norma de Frobenius [17]. Esto se ha realizado con el simple objetivo de comprobar que el código desarrollado es correcto, los datos de precisión no se ofrecen pues son irrelevantes para este trabajo una vez se ha demostrado que el algoritmo es correcto.

La primera prueba experimental consiste en ver cómo se ve afectada la aceleración dependiendo de la configuración de los procesos. En esta prueba, el número total de

Tabla 4.1: Tabla de rendimiento dependiendo de la configuración de los procesos.

PPN	OMP	Tiempo Paralelo	Aceleración
1	32	32,22s	1,53
4	8	26,04s	1,89
16	2	39,24s	1,25

procesos/hilos que se va a utilizar siempre será de 128 para los cuatro nodos, variará el número de procesos MPI por nodo (PPN) y el número de hilos OpenMP utilizados por Intel MKL para ejecutar su rutina de multiplicación de matrices. También se ha decidido fijar el tamaño de las matrices aleatorias a 15000×15000. Se ha utilizado como tiempo de referencia para medir la aceleración el tiempo obtenido en un solo nodo utilizando los 32 *cores* del mismo para ejecutar la rutina de multiplicación de matrices de MKL que utiliza 32 hilos OpenMP. Este tiempo ha sido de 49,15s.

La Figura 4.5 muestra la eficiencia obtenida como una medida del aprovechamiento que se obtiene de los recursos computacionales disponibles.

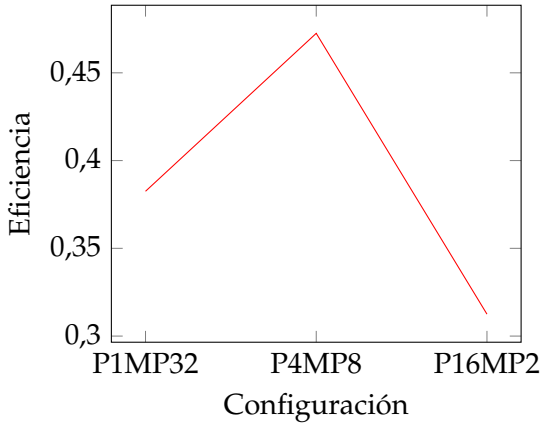


Figura 4.5: Gráfica de eficiencia dependiendo de la configuración de los procesos.

Como resultado de las pruebas (Tabla 4.1 y Figura 4.5) se observa que la configuración más provechosa para el *cluster* Kahan usando cuatro nodos consiste en utilizar cuatro procesos por nodo y dedicar ocho hilos OpenMP por proceso para ejecutar la librería Intel MKL.

Continuando con el análisis, se han lanzado diversas ejecuciones para ver cómo afecta el incremento de la complejidad del problema, es decir, del tamaño de las matrices que se van a multiplicar. Se pueden ver los resultados de este análisis en la Tabla 4.2 y en la Figura 4.6. Para la realización de esta prueba se ha escogido la mejor configuración en paralelo obtenida en la Tabla 4.1. De esta prueba se puede sacar en claro que se consiguen mejoras de rendimiento conforme aumenta el tamaño de la matriz. Sin embargo, hasta multiplicaciones de un tamaño mínimo de 10000×10000 no comienza a merecer la pena el uso del algoritmo paralelo. Se puede apreciar que la aceleración no aumenta de forma constante, aunque a mayor tamaño, mayor es la tendencia a conseguir una mejor eficiencia. Esta variabilidad puede ser debida a que MPI está asignando procesos de forma no contigua en un mismo nodo, es decir, no está claro que los procesos pertenecientes a una misma columna (fila) estén mapeados en el mismo nodo. Por tanto, los broadcast no estarían siendo tan eficientes como deberían al tener un mayor coste de comunicación.

Tabla 4.2: Tabla de rendimiento dependiendo del tamaño de la matriz.

Tamaño	Tiempo Serie	Tiempo Paralelo	Aceleración
1000	0.03	2.01	0.02
5000	3.12	11.34	0.28
10000	24.20	12.70	1.91
15000	49.15	26.04	1.89
20000	181.65	47.55	3.82
25000	193.22	82.46	2.34
30000	327.22	124.96	2.61

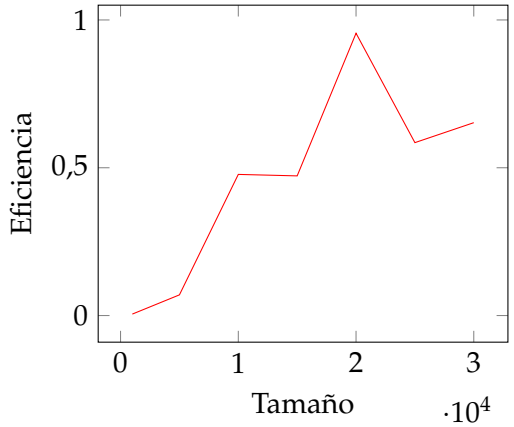


Figura 4.6: Gráfica de eficiencia dependiendo del tamaño de la matriz.

Un importante aspecto a comentar es que no se han podido realizar pruebas de dimensiones más grandes de matrices por el tiempo limitado de computo que se tiene en el clúster Kahan. Este tiempo es agotado principalmente por la generación aleatoria de la matriz mediante la CPU y la ejecución secuencial de la multiplicación. Así pues, no se ha podido llegar al tamaño que teóricamente conseguiría la mayor eficiencia, aunque sí que se puede observar la tendencia. Tampoco se pretende realizar una descomposición exhaustiva para encontrar la mejor configuración a la hora de multiplicar una matriz ya que excede los límites de los objetivos del trabajo, que se centran en el capítulo siguiente.

Finalmente, comentar que todos estos resultados pueden variar dependiendo de la implementación de BLAS escogida y de la propia arquitectura de la *Central processing unit* (CPU) ya que en ejecuciones informales en otro entorno durante el periodo de desarrollo se han observado aceleraciones mayores.

4.6 Conclusiones

Tras las pruebas realizadas, se puede pensar que no se ha conseguido explotar todo el potencial deseado. Sin embargo, se cree que esto es puede ser debido a que la asignación de los procesos MPI no ha sido la óptima en todos los casos. También se piensa que una misma configuración no tiene porqué ser la ideal para todos los tamaños del problema. A pesar de todo, el objetivo principal, que es conseguir familiarizarse con el algoritmo SUMMA con un conjunto de operaciones reducido de MPI que tan solo existe en NCCL se ha conseguido.

CAPÍTULO 5

Solución del problema en entornos HPC formados por GPUs

Una vez ya con una base consolidada conseguida gracias al Capítulo 4, se procederá a realizar la implementación en tarjetas gráficas. Diversos estudios indican que las tarjetas gráficas pueden ser de gran beneficio a la hora de realizar operaciones matemáticas. En el caso particular de la operación *General matrix multiply* (GEMM) en [21] se observa un gran incremento del rendimiento en librerías que emplean el poder de computo de una tarjeta gráfica.

En esta ocasión se ha escogido como lenguaje de programación a C++ y la tecnología de comunicaciones a NCCL. Esta última librería ofrece un gran rendimiento tal y como se ve en [22]. Cabe destacar que el estudio [22] trata la versión 1.3 y el desarrollo de la librería de este trabajo ha sido sobre la versión 2.6.4, por tanto, han habido mejoras de rendimiento que se pueden observar en las notas de parche de la librería que se pueden consultar aquí [23]. El secreto consiste en emplear la tecnología de comunicación entre tarjetas gráficas NVLink. A pesar de ser el pilar fundamental, NCCL también admite otros estándares de comunicación tradicionales como *Peripheral Component Interconnect* (PCI). Por lo que respecta a las librerías de las GPUs se ha elegido usar CUDA junto con cuBLAS para la realización de operaciones matemáticas.

Para compilar de forma efectiva y ordenada el código escrito en CUDA y en C++, se ha decidido usar la herramienta CMake siguiendo las instrucciones especificadas en [24]. Para poder emplear las instrucciones de BLAS implementadas en MKL ha sido necesario añadir el módulo CMake que se encuentra en [25]. Mediante esta herramienta se compilan por separado los archivos fuente escritos en C, C++ y CUDA a código objeto y luego se enlazan para formar el ejecutable.

5.1 Análisis de coste de operaciones en la GPU

Primero que todo, se ha realizado un breve análisis mediante un pequeño programa que mide por separado del coste temporal que tienen las operaciones más significativas de la GPU. Para medir los tiempos de la forma más precisa posible se han usado los cudaEvents seguido de las instrucciones de [26] para obtener las métricas de la Tabla 5.1. En caso de que existiese una variante asíncrona de la instrucción, ha sido esta la escogida.

Como consecuencia de esta investigación podemos sacar en claro varias cosas:

Tabla 5.1: Tiempos de operaciones de la GPU dependiendo del tamaño de la matriz. Unidades en milisegundos.

N	Malloc	memset	HToD	DToH	cublasDgemm	cublasCreate	Broadcast	commCreate
1000	0.22	0.01	1.18	4.93	0.58	131.72	0.40	992.88
2000	0.24	0.02	3.65	18.94	4.02	132.99	1.05	1020.74
3000	0.28	0.05	22.00	43.00	12.37	133.39	2.18	974.42
4000	0.29	0.08	21.51	75.15	27.50	131.28	3.78	1000.89
5000	0.32	0.12	30.53	129.41	54.31	127.91	5.92	952.39
6000	0.37	0.17	47.33	184.97	89.94	131.51	8.35	1015.90
7000	0.44	0.23	60.16	228.50	142.77	129.13	11.31	1050.86
8000	0.53	0.29	82.24	297.09	211.35	131.32	14.76	1021.49
9000	0.59	0.37	100.84	376.11	302.54	130.60	18.74	1014.87
10000	0.80	0.46	126.01	483.12	415.39	130.58	23.16	979.93

- Las operaciones cublasCreate y commCreate se mantienen constantes con el tamaño del problema. Ambas son operaciones muy costosas, por lo tanto, habrá que tratar de minimizar su peso en la medida de lo posible.
- Las operaciones Malloc, memset, HToD (transferencia Host-To-Device), DToH (transferencia Device-To-Host), cublasDgemm y broadcast ven incrementado su tiempo de ejecución conforme al aumento del tamaño de las matrices. Las cuatro primeras son evitables entre ejecuciones sucesivas del algoritmo SUMMA, es decir, una vez las matrices han sido distribuidas entre las GPUs, se pueden realizar multiplicaciones de matrices sin necesidad de realizar transferencias CPU-GPU. En cambio las dos últimas son operaciones que se ejecutan siempre que se realiza una multiplicación matricial.
- Las operaciones HToD y DToH ocupan un tiempo significativo que ha de tenerse muy en cuenta al tratarse de operaciones de transferencia de información entre dispositivo y *host* por el PCI, mucho más lento que NVLink.

5.2 Abstracción de GPUs lógicas

Una característica importante del programa desarrollado ha sido la inclusión del concepto de las GPUs lógicas. En MPI se puede exceder el número de procesos que van a ejecutar el programa por encima de los procesadores físicos. Gracias a ello, es posible observar el comportamiento lógico del código desarrollado en distintas configuraciones de mallas. Como consecuencia, es posible encontrar errores en el código y adelantarse a posibles problemas en ejecuciones a pesar de, evidentemente, no obtener el mismo rendimiento que si se ejecutara sobre núcleos físicos. La finalidad de ello es poder probar el funcionamiento sin tener que realizar distintas inversiones en nodos con distinto números de GPUs. Esta característica permitirá utilizar este software en configuraciones reales que tienen más GPUs conectadas entre sí mediante NVLink en un futuro.

El primer paso consiste en conseguir que cada GPU lógica se corresponda con una tarjeta gráfica física(por física se entiende que es una GPU que está en el sistema, que es real y no se va a simular su comportamiento). Es importante señalar, que en un sistema con el mismo número de GPUs lógicas que de físicas, cada GPU física tendrá asociada únicamente una GPU lógica. Sin embargo, si se decide que el sistema posee mas tarjetas gráficas lógicas que físicas, las GPUs físicas tendrán asignadas varias GPUs lógicas. Para la consecución de este objetivo hemos empleado una operación modular del identificador de la GPU contra el número total de tarjetas gráficas físicas que tiene el sistema tal y como se ve en el Algoritmo 8.

A continuación se muestra cómo queda la asignación de GPUs lógicas y físicas en un sistema de cuatro tarjetas gráficas simulando ser ocho en una malla de 2×4.

Algoritmo 8 Obtención de la GPU física

```
1 int gpuRank %gpuSizeSystem;
```

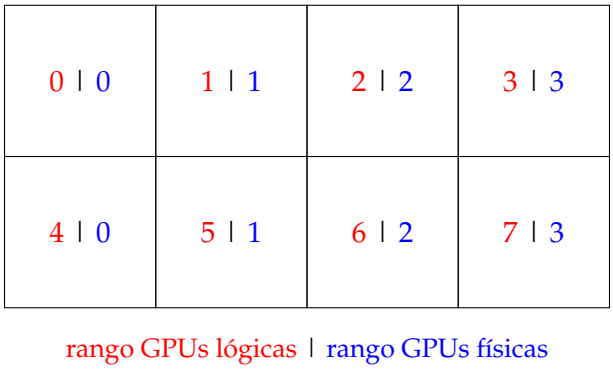


Figura 5.1: Malla que muestra el rango de GPUs lógicas y físicas.

Un escollo sustancial en el desarrollo de esta propiedad ha sido el comportamiento de los comunicadores en NCCL. En [15] se avisa de que no se puede usar el mismo dispositivo múltiples veces con distintos rangos para un mismo comunicador. También se advierte de que usar distintos comunicadores concurrentemente puede acabar en interbloqueos. Durante el desarrollo se experimentaron ambos problemas. Por otro lado, se descubrió que si en un mismo broadcast se llama el mismo número de veces a las GPUs implicadas se obtenía el funcionamiento esperado y no se sufría ningún interbloqueo ni cierre del programa inesperado. Además, se descubrió que no existía ningún problema en que una tarjeta gráfica se comunicase consigo misma.

Tras reflexionar sobre los comportamientos observados se optó por crear una estructura de un vector de vectores para cada gráfica, la cual almacenara la información de sus colores. En el primer vector estarían los rangos globales de las GPUs lógicas sin que se repita su rango físico. Por lo que respecta a los sucesivos vectores, estarían las GPUs que ven su rango físico repetido. Estas últimas realizarán comunicaciones consigo mismas y así se evitan los problemas comentados con anterioridad.

Una vez listo este sistema, se pudo probar satisfactoriamente el funcionamiento del programa en dos sistemas: 1) con una única NVIDIA GTX 1080TI y; 2) con cuatro NVIDIA TESLA P100, actuando sobre 4[1], 5, 6, 7, 8, 9, 10, 11 y 12[2] GPUs lógicas.

5.3 Implementación de SUMMA adaptado a GPUs

Antes de ponerse a realizar las modificaciones, se consultaron los ejemplos proporcionados por NVIDIA en [15] sobre cómo programar con la librería NCCL. La documentación proporciona tres alternativas:

- Un único hilo maneja todos los dispositivos.
- Un proceso MPI por cada dispositivo.
- Un proceso MPI es capaz de manejar distintos dispositivos.

[1] Cuatro es el número mínimo de procesos para el algoritmo SUMMA.

[2] Los números primos actúan como el número anterior inmediatamente más grande no primo. La razón es que no se pueden tener mallas desiguales.

En la implementación del programa se eligió la primera alternativa ya que requería el uso de una librería menos y eso hace que sea compatible con más sistemas al no ser requerida para la compilación y ejecución. Un esquema a modo de ejemplo de como se suele programar con esa alternativa se puede ver en el Algoritmo 9.

Algoritmo 9 Patrón de programación de NCCL.

```
1 //Pre-operaciones de la cpu
2 for(i=0;i<gpuSize;i++)
3 {
4     cudaSetDevice(i);
5     //Realizar las instrucciones necesarias por la gpu
6 }
7 //Post-operaciones de la cpu
8 //Esperar streams de las gpus en caso de que sea necesario
```

Debido a esta elección, un patrón de programación que se repite es el de un bucle que itera sobre los distintos dispositivos antes de que estos realicen las operaciones pertinentes. Además, se han usado las variantes asíncronas de las instrucciones de las tarjetas gráficas siempre que ha sido posible. Para la ejecución asíncrona de instrucciones en una GPU es necesario el uso de *streams*. Un *stream* está formado por una secuencia de instrucciones secuenciales que se ejecutan en una GPU. Sin embargo, se pueden ejecutar distintos *streams* de forma concurrente en una misma GPU o en diversas tarjetas gráficas a la vez. Al utilizar *streams* será necesario esperar a la terminación de los diferentes cudaStream si se quiere asegurar que los datos a los cuales se va a acceder son los esperados.

En cuanto al tema de las comunicaciones, se han cambiado las llamadas broadcast de MPI por las de NCCL. Es importante señalar que se han agrupado todas ellas entre las instrucciones ncclGroupStart y ncclGroupEnd debido a que, tal como se comenta en [15], es imprescindible hacerlo así para el patrón escogido. Además, realizando las comunicaciones de esa forma, NCCL se encarga de optimizarlas reduciendo la sobrecarga y su latencia. Otro cambio realizado con respecto a las comunicaciones consiste en que no es necesario inicializar el entorno al principio del programa. A cambio, se debe de ejecutar la instrucción ncclCommInitAll cada vez que deseemos crear un nuevo comunicador. Cuando se desee conocer el rango se debe emplear cudaSetDevice seguido de ncclCommUserRank. Es necesario que sea ese el orden u obtendremos el rango de una tarjeta gráfica que no deseamos en el mejor de los casos o una excepción en el peor, debido a que la GPU solicitada no se encuentre en dicho comunicador.

Por otra parte, respecto de la parte computacional, se ha pasado de usar una implementación de BLAS en CPUs a la versión de cuBLAS para tarjetas gráficas NVIDIA.

De esta forma, el algoritmo SUMMA resultante sigue la siguiente estructura:

- Copia de las matrices desde el *host* que contenga la información al *buffer* de la GPU. Al igual que sucedía en la versión de MPI pero con procesos.
- Ejecución de los broadcast agrupados entre ncclGroupStart y ncclGroupEnd.
- Espera de las comunicaciones entre las diferentes tarjetas gráficas.
- Ejecución de las multiplicaciones en cada GPU.
- Espera de las multiplicaciones.
- Repetir todos los pasos anteriores tantas veces como tamaño de columnas tenga la malla.

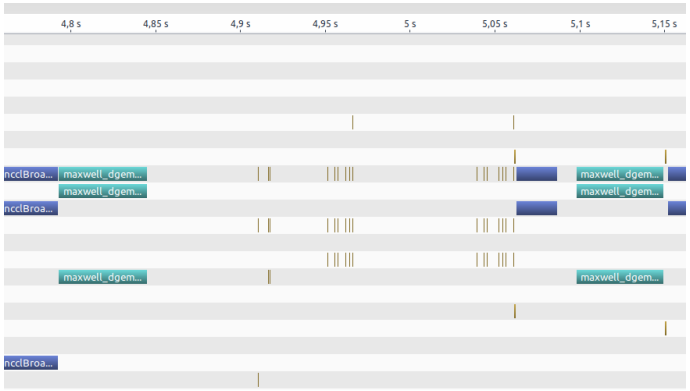
Asimismo, se ha cambiado el modo de generar matrices aleatorias para realizar las distintas pruebas. Ahora son creadas por una GPU y devueltas al *host* antes de ser destruidas de la tarjeta gráfica y comenzar el proceso de evaluación. Gracias a ello, la generación tarda menos tiempo. Esta implementación se ha conseguido mediante cuRAND [27][3].

Otro aspecto, el cual se ha mejorado respecto a lo realizado en el Capítulo 4, ha sido el uso de genericidad con soporte para los tipos double y float. Continuando con las mejoras, se ha dado soporte a la sobrecarga de operadores de C++ para facilitar el uso al usuario. Se puede encontrar más información de ello en el Capítulo 6.

Por último, se ha comprobado el código igual que en el Capítulo 4, comparando dos matrices. Una obtenida mediante el código diseñado y otra en secuencial en una tarjeta gráfica para después proceder a obtener el error relativo, utilizando la norma de Frobenius, según se define en la Ecuación 4.1.

5.4 Análisis y optimización del rendimiento

Tras finalizar el desarrollo principal y comprobar que todo funcionaba se halló un importante contratiempo. La implementación desarrollada arrojaba una aceleración negativa. Para tratar de solucionar los problemas se recurrió a las herramientas nvprof y NVIDIA Visual Profiler [28]. Por un lado, nvprof es un profiler sin interfaz gráfica para GPUs de NVIDIA capaz de generar ficheros para un posterior análisis. Por otro lado, NVIDIA Visual Profiler, es un profiler con interfaz gráfica para GPUs de NVIDIA y que también permite importar ficheros generados con nvprof. Gracias a estas herramientas y a la información obtenida en la Sección 5.1 se pudo descubrir cuáles podrían ser las razones para las bajas prestaciones obtenidas inicialmente y tratar de solventarlas. En las distintas figuras de este apartado, se ha representado ncclbroadcast con el siguiente color ■ y cublasDGemm con este otro color ■. La superposición de distintos colores significa que esas operaciones se están realizando de forma simultánea. El principal objetivo de optimización es tratar de conseguir que haya el mínimo espacio posible entre las operaciones ncclbroadcast y cublasDGemm.



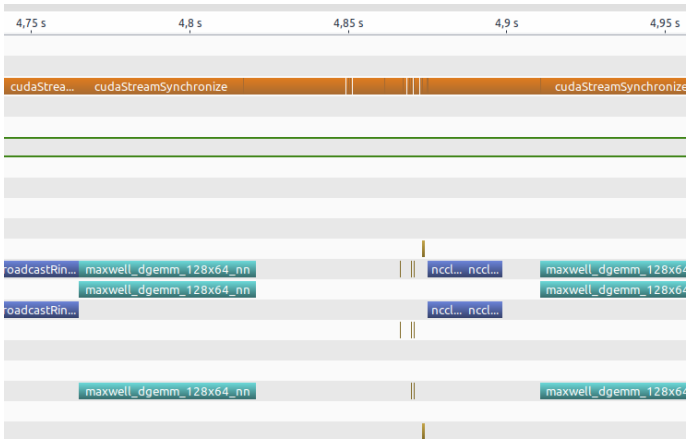
Dimensiones de la matriz 10000 × 10000 realizando 2 multiplicaciones.

Figura 5.2: *profiler* del programa sin optimizar.

En la Figura 5.2 se puede ver que entre la finalización de una multiplicación y el comienzo de otra pasan alrededor de 0,25 s. Este tiempo es consumido en la creación de comunicadores, reserva de memoria y liberación de recursos cada vez que se ejecuta SUMMA.

[3]Librería para la generación de números aleatorios en una tarjeta gráfica.

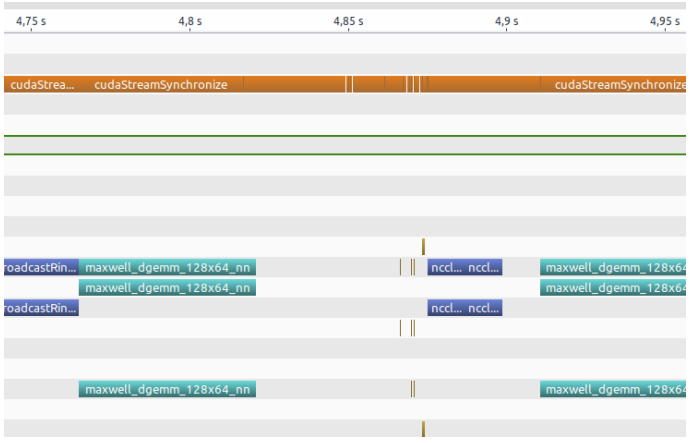
Como primera medida de optimización se procuró que los comunicadores solo se crearan si no se habían creado antes. Estos se guardarían en un árbol binario donde la clave es la dimensión de las filas de la malla y como valor los comunicadores. Debido a que normalmente y salvo alguna rara excepción siempre se va a mantener la estructura de la malla, los comunicadores son creados solo una vez y se obtiene una notable mejoría de rendimiento tal y como se ve en la Figura 5.3. En ella se puede observar que los $0,25$ s se han reducido a unos $0,05$ s.



Dimensiones de la matriz 10000×10000 realizando 2 multiplicaciones.

Figura 5.3: *profiler* del programa con la optimización de los comunicadores.

Continuando con la optimización, se consiguió que para las matrices del *buffer* para el algoritmo SUMMA no reserve memoria ni se libere en cada ejecución. Solo realizarán estas operaciones en caso de que las matrices del *buffer* no sirvan porque se necesiten más (cambio de malla, algo muy poco frecuente) o porque su tamaño sea distinto. En la Figura 5.4 se ve que apenas ha mejorado $0,01$ s o $0,02$ s respecto a la Figura 5.3. Puede parecer una nimiedad, pero en un programa que realice múltiples iteraciones puede significar una diferencia.



Dimensiones de la matriz 10000×10000 realizando 2 multiplicaciones

Figura 5.4: *profiler* del programa con la optimización de los comunicadores y de memoria

Para finalizar con la optimización, se ha intentado optimizar el tiempo de espera hasta que se reciben los datos de las comunicaciones en las distintas GPUs. En la Figura 5.5a se puede ver que se emplea un tiempo precioso (alrededor de $12ms$ para cada iteración del algoritmo) en esperar a que llegue la información de los *Broadcast* del algoritmo SUMMA.

En un inicio, las comunicaciones se realizaban a la vez en un mismo bucle que recorría todas las GPUs y realizaba el *Broadcast* pertinente para las filas y columnas. Se puede observar en la Figura 5.5b que se ha conseguido reducir a unos $2\mu s$. Dicha reducción se ha producido gracias a separar en dos el bucle para la comunicación de las filas y de las columnas. Es decir, primero habrá un bucle que recorra todas las tarjetas gráficas para comunicar la información relacionada con las filas y, posteriormente, otro bucle similar que se encargará de las columnas. De esa forma se consigue solapar el tiempo empleado en la espera de las comunicaciones.

(a) *Broadcast* sin optimizar de multiplicación matriz 10000×10000

(b) *Broadcast* optimizado de multiplicación matriz 10000×10000

Figura 5.5: Comparación de optimización de *Broadcast*.

A continuación se evalúa la ocupación de las GPUs. En la Figura 5.6 se aprecia cómo aumenta significativamente más el tiempo empleado en las multiplicaciones que en los broadcast cuando aumenta el tamaño del problema (de 10000 a 30000). Si deseamos estimar la velocidad de transferencia del broadcast lo podremos realizar tal y como se aprecia en la Ecuación 5.1 para el caso de una multiplicación 10000×10000, donde cada bloque transmitido será de 5000×5000,

$$\frac{(\text{Nº filas}) \cdot (\text{Nº columnas}) \cdot 8}{10^9 \cdot (\text{Tiempo en segundos})} = \frac{5000 \cdot 5000 \cdot 8}{10^9 \cdot 12 \cdot 10^{-3}} = 16,67GB/s. \tag{5.1}$$

El tiempo empleado en esa transmisión ha sido de unos $12ms$. Señalar que en Nowherman hay un enlace de NVLink caído y es posible que en caso de que no fuese así el tiempo empleado en la transmisión de datos entre tarjetas gráficas sería menor, cercano a unos $20GB/s$, que es el valor teórico de NVLink v1 por enlace.

(a) Ocupación de multiplicación matriz 10000×10000

(b) Ocupación de multiplicación matriz 30000×30000

Figura 5.6: Comparación de ocupación.

Finalmente, en la Figura 5.7 podemos ver cómo las cuatro GPUs operan de forma paralela y que no están ociosas.

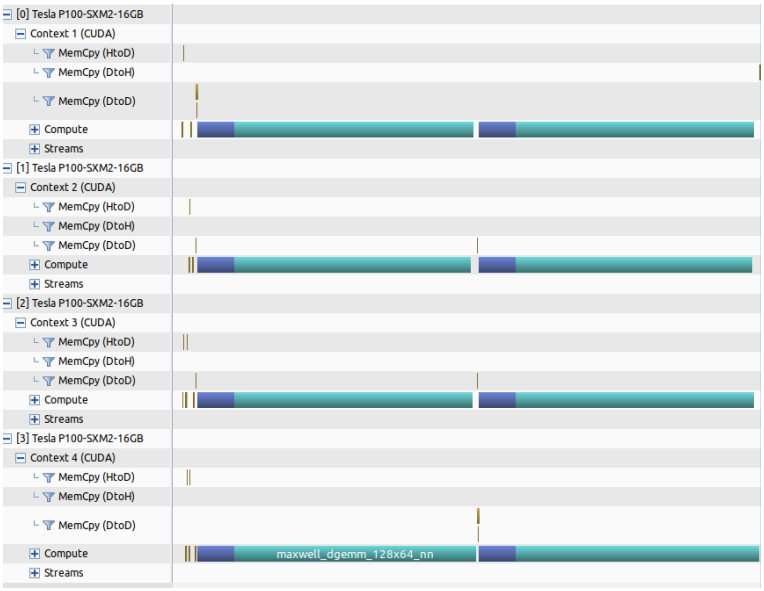


Figura 5.7: Imagen del *profiler* que muestra el uso de las GPUs.

5.5 Rendimiento

A continuación se va a proceder a analizar la eficacia del programa con el tipo de dato double. Las pruebas se han realizado en nodo Nowheman, si se desean conocer sus especificaciones se puede consultar el Capítulo 2.

La prueba realizada consiste en realizar una multiplicación variando su tamaño y, de esa forma, ver cuanto tarda en los distintos modos. El tiempo de CPU mostrado es el del procesador de Nowherman usando todos sus núcleos ejecutando la operación de multiplicación mediante Intel MKL. En la prueba se ha decidido medir únicamente el tiempo que tarda la multiplicación, es decir, el algoritmo SUMMA en sí. Por ello, no se han contabilizado el tiempo empleado en la creación de los comunicadores u otros elementos. En cuanto a las operaciones serie, tampoco se han contabilizado los tiempos de transferencia del *host* al dispositivo y viceversa.

Tabla 5.2: Tabla de rendimiento en GPUs dependiendo del tamaño de la matriz.

Tamaño	CPU	1 GPU	4 GPUs	Aceleración 1 GPU vs 4 GPUs
1000	0.10	0.00	0.00	0.21
5000	0.70	0.06	0.03	1.74
10000	3.06	0.42	0.17	2.50
15000	9.01	1.40	0.51	2.77
20000	17.46	3.31	1.08	3.08
25000	30.92	6.51	1.99	3.27
30000	42.24	11.25	3.34	3.37

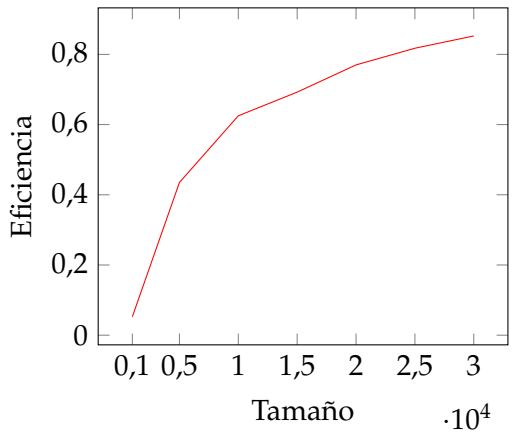


Figura 5.8: Gráfica de eficiencia en GPUs en función del tamaño de la matriz.

Tras el análisis mostrado en la Tabla 5.2 y la Figura 5.8 se observa que el programa desarrollado empieza a ser efectivo para multiplicaciones de tamaño 5000×5000. A medida que el tamaño va aumentando, se consigue apreciar una mayor diferencia a favor del algoritmo SUMMA.

Adicionalmente, indicar que mediante la implementación desarrollada es posible multiplicar matrices de mayores dimensiones que de forma secuencial. Mediante una única GPU se pueden multiplicar matrices de unas dimensiones que ocupen el total de la memoria de esa tarjeta gráfica, en este caso 16 GB, que sería un tamaño aproximado de 31750×31750, mientras que en paralelo variaría dependiendo del número de GPUs disponibles. Este sería el caso con matrices de tamaño 40000×40000 (ambos tamaños se obtienen multiplicándo la matriz por sí misma).

5.6 Conclusiones

Tras completar el desarrollo y evaluar su rendimiento, se puede afirmar que se han conseguido los objetivos planteados de esta librería de C++. Para multiplicaciones de gran tamaño se obtiene una buena eficiencia y, además, es posible multiplicar matrices de mayores dimensiones que si tan solo se contase con una única tarjeta gráfica para realizar la operación. También hay que señalar que se han realizado pruebas informales con la librería de NVIDIA CublasXT que permite utilizar múltiples GPUs para la multiplicación de matrices. Para estos casos concretos se observa que la librería de NVIDIA rinde peor que operando con una única tarjeta gráfica. Por tanto, la librería desarrollada en este trabajo rinde mejor que Cublas y CublasXT para los casos analizados. En definitiva, se puede afirmar también que el algoritmo SUMMA, propuesto para configuraciones de memoria distribuida, es un buen algoritmo para llevar a cabo multiplicaciones de matrices en esta configuración especial HPC formada por GPUs interconectadas mediante NVLink.

CAPÍTULO 6
Librería C++ desarrollada

Para la consecución de todos los objetivos indicados en la Sección 1.2 son necesarias otras operaciones diferentes de la multiplicación.

Es de vital importancia señalar que, como la librería desarrollada tiene como fundamento operar sobre matrices distribuidas, el primer paso que realizará esta será distribuir las matrices, en caso de que no lo estén, antes de realizar alguna de las operaciones explicadas en la Sección 6.2, la Sección 6.3 y la Sección 6.4. La decisión que se tomará para distribuir estas matrices será la misma que si se multiplicasen por ellas mismas. Por lo que respecta a las operaciones descritas en la Sección 6.5, estas están restringidas al caso de que la malla de GPUs sea cuadrada, es decir, con cuatro, nueve, ... GPUs. Esto es debido a que con esas características de malla el tamaño de los bloques de la matriz en las distintas tarjetas gráficas será el mismo.

En cuanto a la dificultad de desarrollo, es menor que en el caso de la multiplicación expuesta en el Capítulo 5. Aun así, se puede ver que algunas de estas operaciones representan un reto debido al hecho de encontrarse distribuidas. Esto último tiene implicaciones en las prestaciones, lo que supone un doble reto.

La Sección 6.7 muestra cómo se usan las operaciones descritas a lo largo del capítulo.

6.1 Explicación de las operaciones BLAS utilizadas

A continuación se explicarán las distintas funciones de BLAS utilizadas.

- void axpy(N,DA,DX,INCX,DY,INCY): realiza la operación $Y = \alpha \cdot X + Y$. El valor N indica el número de elementos del vector DY, que representa al vector Y. El escalar DA representa a α. El valor DX representa el vector que opera como X. Los valores enteros INCX e INCY indican la separación entre elementos de para los vectores X e Y, respectivamente, lo que permite utilizar vectores cuyos elementos no están almacenados consecutivamente en memoria.

- void scal(N,DA,DX,INCX): realiza la operación $X = \alpha \cdot X$. El valor N indica el número de elementos del vector DX, que representa al vector X. DA representa al escalar α. Por último, INCX indica la separación entre elementos de X.

- int asum(N,DX,INCX): realiza la operación $res = \sum_{\substack{i=1 \\ i+=INCX}}^{N} |X_i|$. El valor N indica el número de elementos del vector DX, que representa al vector X, mientras que INCX indica la separación elementos de X.

- int amax(N,DX,INCX): realiza la operación $res = \text{máx}\,|a_1|, |a_2|, ..., |a_n|$. El valor N indica el número de elementos del vector DX, que representa al vector X, mientras que INCX indica la separación entre elementos de X.

6.2 Multiplicación de una matriz por un escalar

En el ámbito de esta operación se pueden encontrar los siguientes casos:

- Cambio de signo a toda la matriz $(-A)$.
- Multiplicación de un número por la matriz $(3 * A)$.
- Multiplicación de una matriz por un número $(A * 3)$.
- Multiplicación y asignación de una matriz por un número $(A* = 3)$.
- División de una matriz por un número $(A/3)$.
- División y asignación de una matriz por un número $(A/ = 3)$.

Para implementar estas operaciones se ha escogido la operación $* =$ como la principal, es decir, servirá de base para la implementación del resto de operaciones, que llamarán a esta para realizar el trabajo que se espera de ellas. La implementación de la operación $* =$ consiste, básicamente, en que cada GPU lógica ejecute cublasScal sobre el bloque que contiene. Para ello, se ha sobrecargado el operador correspondiente siguiendo la implementación canónica de clases de C++. La cabecera de definición del operador se muestra en el Algoritmo 10. El uso de template en dicho algoritmo es debido a seguir un modelo de genericidad donde Toperation es el nombre del tipo de la variable que usa dicha propiedad. Por lo que respecta a MainMatrix, es el tipo del objeto que almacena toda la información de la matriz en la librería desarrollada.

Algoritmo 10 Sobrecarga de operaciones con la identidad

```
1 template <class Toperation> MatrixMain<Toperation> MatrixMain<Toperation>&
    MatrixMain<Toperation>::operator*=(const Toperation& alpha)
```

Al margen de las operaciones comentadas arriba, también se ha facilitado la opción al usuario de modificar el parámetro α de GEMM, permitiendo una mejora del rendimiento al no tener que realizar SUMMA por un lado y posteriormente una operación cublasScal por otro.

6.3 Sumas y restas de una matriz con la identidad

Este tipo de operación involucra a un número real que multiplicará a la matriz identidad y después ésta será la que se operará junto con la matriz que se desee. Es importante señalar que dichas operaciones sólo se podrán realizar si la matriz con la cual queremos operar es cuadrada debido a la definición de matriz identidad. En el espacio de esta operación se pueden encontrar los siguientes casos, donde el número 3 es solo un ejemplo de escalar:

- Suma de un número y una matriz $(3 + A)$.
- Suma de una matriz y un número $(A + 3)$.

- Suma y asignación de una matriz y un número ($A+=3$).
- Resta de un número y una matriz ($3-A$).
- Resta de una matriz y un número ($A-3$).
- Resta y asignación de una matriz y un número ($A-=3$).

Se ha utilizado notación de tipo Matlab, es decir, la operación $3+A$ representa realmente la operación $3I+A$, siendo I la matriz identidad del mismo orden que la matriz A.

Se ha escogido que la implementación principal será sobre la operación $+=$ y que el resto de operaciones utilizarán la implementación de esta para realizar el trabajo que se espera de ellas. La cabecera del operador sobrecargado se muestra en el Algoritmo 11. En dicho algoritmo podemos encontrar la variable constantAddition que representa el escalar por el cual irá multiplicada la matriz identidad.

Algoritmo 11 Sobrecarga de operaciones con la identidad

```
1 template <class Toperation> MatrixMain<Toperation> MatrixMain<Toperation>::
    operator+=(const Toperation constantAddition)
```

La programación de esta operación ha sido bastante mas compleja que la del resto de la sección, pues no se genera la matriz identidad, sino que se utiliza la función axpy de una manera particular. El parámetro DA contendrá la constante que multiplica a la identidad. El parámetro DX tendrá un solo valor, que será 1, es decir, el valor de las diagonales de la identidad. Para que la operación axpy funcione correctamente es necesario que el parémetro INCX tenga un valor de 0, Por último, para acceder únicamente a los elementos de la diagonal, el parámetro INCY deberá contener el valor N+1, siendo N la dimensión de la matriz o bloque que se modifica con esta operación.

Las constantes que intervienen deberán de ser previamente pasadas del *host* a los distintos dispositivos. Por otro lado, es necesario saber cuántos elementos hay en la diagonal de cada bloque y donde empieza dicha diagonal dentro de él. Para resolver este inconveniente se calcula el *column major order* (Figura 4.4) y el *row major order* (Figura 4.2). El primero fue útil para conseguir las características mencionadas con anterioridad en caso de que la diagonal se encuentre en la primera columna del bloque. En cambio, si la diagonal no se encuentra ahí, se encontrará en alguna posición de la primera fila del bloque. Se puede ver que la diagonal se detectará en un lado o en otro de forma obligatoria al tratarse precisamente de una diagonal. Para ilustrar mejor la problemática de esta operación, en la Figura 6.1 se puede ver un ejemplo donde se aprecian todos los problemas.

0 − 0	6 − 1	12 − 2	18 − 3	24 − 4	30 − 5
1 − 6	7 − 7	13 − 8	19 − 9	25 − 10	31 − 11
2 − 12	8 − 13	14 − 14	20 − 15	26 − 16	32 − 17
3 − 18	9 − 19	15 − 20	21 − 21	27 − 22	33 − 23
4 − 24	10 − 25	16 − 26	22 − 27	28 − 28	34 − 29
5 − 30	11 − 31	17 − 32	23 − 33	29 − 34	35 − 35

Elementos de la diagonal representados como 0-0 . Elementos representados como *column major order-row major order*.

Figura 6.1: Matriz con índices diagonal.

Una operación que es una excepción dentro de las comentadas en este apartado es $\alpha - A$, la cual realizará primero la operación $- =$ y después la operación $* = -1$.

6.4 Norma 1

Esta operación tiene como objetivo obtener el valor máximo (y) de la suma de los valores absolutos de los elementos de las columna de una matriz dada $A = [a_{i=1...m,j=1...n}]$ [17],

$$y = \max_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{ij}|.$$

Se han realizado dos implementaciones para esta operación. La primera de ellas está pensada para el caso en el que existen más GPUs lógicas que físicas. En ese caso no funciona correctamente la comunicación colectiva ncclReduce. Si se cuenta con el mismo número de dispositivos lógicos que físicos se empleará una segunda alternativa que utilizará NCCL. Es importante señalar que en la primera forma de resolver la norma, se emplea la CPU para algunas operaciones mientras que en la segunda opción solo se usan las GPUs. Esta segunda alternativa es más rápida que la primera.

Las etapas de la operación implementada según la segunda alternativa, cuando hay más tarjetas gráficas lógicas que físicas son las siguientes:

1. Cada bloque en el dispositivo suma sus columnas en valor absoluto mediante la rutina asum de cuBLAS y almacena los resultados en vectores del *host*.
2. El *host* suma los bloques de cada color de columna de forma contigua mediante axpy de BLAS. De esa forma ya tenemos la suma de cada columna separa en bloques dependiendo del color.
3. Posteriormente, el *host* llama a amax de BLAS por cada color de columna y lo almacenamos en un vector que tendrá el máximo valor de la suma de cada columna para cada color.
4. Finalmente se llama, por última vez, a amax para obtener el máximo entre todos los bloques. Ese será el resultado de la norma.

En cuanto a la segunda versión, la que realiza todos los pasos con NCCL dentro del sistema de GPUs sin intervención del *host*, sus etapas son las siguientes:

1. Cambiar el modo de trabajar de punteros de cuBLAS al modo dispositivo.
2. Realizar asum mediante cuBLAS con cada bloque y almacenar los resultados en cada tarjeta gráfica.
3. Utilizar ncclReduce de tipo suma entre los bloques del mismo color de columna. El *root* será la gráfica que da número a ese color de columna.
4. Restablecer el modo de trabajar de punteros de cuBLAS al modo *host*.
5. Conseguir el máximo para cada color de columna realizando amax con cuBLAS en cada GPU.
6. Ejecutar ncclReduce de tipo máximo entre las gráficas del color cero de fila para conseguir el valor de la norma final. Esta operación tendrá el *root* a la tarjeta gráfica cero.

7. Copiar ese valor al *host* desde dispositivo.

En ambos casos, como último paso, se liberan los recursos que han sido reservados para ejecutar la operación.

6.5 Sumas y restas entre matrices

Para la realización de este tipo de operaciones se realizará primero la comprobación de que las matrices A y B son del mismo tamaño, lo que supone una restricción matemática para realizar la operación. En caso contrario se lanzará una excepción.

En la versión actual del software es necesario que ambas matrices se encuentren igualmente distribuidas y que, por tanto, el tamaño de sus bloques sea exactamente el mismo. De no ser así, una excepción será lanzada. En principio, no será un problema en mallas cuadradas, pero en mallas que no sean de esa singularidad actualmente no se puede realizar la operación. La principal razón es que necesita una distribución de las características de los bloques muy distinta a la multiplicación. En la Sección 8.4 se discuten posibles formas de realizar su implementación para trabajos futuros. Las operaciones implementadas son las siguientes:

- Suma de una matriz y una matriz ($A + B$).
- Suma y asignación de una matriz y una matriz ($A+ = B$).
- Resta de una matriz y una matriz ($-A$).
- Resta y asignación de una matriz y una matriz ($A- = B$).

La implementación se ha realizado mediante llamadas a la función axpy de cuBLAS para los distintos bloques de ambas matrices con un α a 1 si se trata de una suma o -1 en caso de la resta. Un ejemplo de la sobrecarga de estos operadores en C++ se ve en Algoritmo 12.

Algoritmo 12 Sobrecarga de operaciones con la identidad

```
1 template <class Toperation> MatrixMain<Toperation> MatrixMain<Toperation>&
    MatrixMain<Toperation>::operator+=(MatrixMain<Toperation>& maMain)
```

Además, se ofrece la posibilidad de llamar a un método en el cual se puede cambiar explícitamente el valor de α para, de ese modo, optimizar la operación y realizar una sola función axpy en vez de un Scalar y posteriormente axpy.

6.6 Otros operadores de C++

Finalmente, destacar que otros operadores han sido sobrecargados para la gestión de objetos de C++ tal y como soporta el estándar C++11. La sobrecarga de dichos operadores ha seguido las recomendaciones de la forma canónica de implementar clases de C++11, la cual es una orientación de implementación de algunas funciones de una clase por lo que a parámetros y utilidad se refiere aunque a la hora de la implementación de cada función, cada una será distinta. Con ellos se consigue gestionar la memoria de forma más eficiente. Estos operadores sobrecargados son:

- Operador de copia constructor. Crea un nuevo objeto a partir de uno ya existente. Es útil para cuando estás declarando un objeto a partir de uno ya existente
- Operador de copia de asignación. Elimina el objeto actual y copia el nuevo objeto.
- Operador constructor de movimiento. Mueve los punteros del objeto que había creado al que se está creando. Con él se consigue ahorrar memoria. Un ejemplo de uso es cuando estás agregando un objeto a un vector.
- Operador de copia de movimiento. Mueve los punteros del objeto que había creado al que se está creando. Con él se consigue ahorrar memoria. Un ejemplo de uso es cuando estás agregando un objeto a un vector.
- Destructores. Debido a la gestión manual de memoria tanto en el *host* como en el dispositivo ha sido necesario implementar los destructores a mano para, de esa forma, liberar la memoria de forma correcta.

6.7 Utilización de la librería

En el Algoritmo 13 se puede ver un fragmento de código que se explica a continuación.

- En la línea dos, se crea el objeto del entorno multiplicativo que se encarga de tener toda la infraestructura para que las diferentes operaciones entre las distintas matrices funcionen. gpuSizeWorldArgument es él número de GPUs lógicas que se van a usar. En caso de que se pase el valor -1 se usará el valor máximo entre las que se encuentren en el sistema y 4. La variable gpuRoot índica qué tarjeta gráfica actua como *root*, normalmente la 0. Continuando con la variable opt es un enumerado que indica el tipo los datos que se trasmitirán mediante NCCL. Debe de coincidir con el tipo que se usa en genericidad, los tipos posibles son double o float que tendrán que coincidir con los valores de opt como MultDouble o MultFloat respectivamente. El último argumento es un booleano que se usa para indicar si se desean imprimir las matrices al multiplicarse y otros parámetros como tamaño de la malla o datos de las matrices que van a realizar la multiplicación.
- De la línea ocho a la 12, se pueden ver las diferentes formas de multiplicar matrices entre sí. Las dos primeras lineas son multiplicaciones usuales entre matrices. Las tres líneas siguientes son una multiplicación alterando el valor de α que se usa en GEMM.
- Por lo que respecta a las sumas y restas entre matrices podemos ver de la línea 14 a la 17, una forma ordinaria de operar con ellas. Sin embargo en la línea 18 se puede apreciar una forma en la que ma obtendrá el valor del resulta ma=2∗ma+mb, donde 2 es el argumento α de la conocida operación de BLAS llamada axpy.
- De la línea 20 a la 25, se pueden ver las diferentes formas de operar las matrices con la identidad.
- De la línea 27 a la 30, se pueden ver las diferentes formas de operar con escalares.
- En la línea 32, se puede obtener el valor de la norma 1 de la matriz.
- En la línea 34, se muestra el cambio de signo de forma completa a la matriz.

- Finalmente, de la línea 36 a la 41 se recogen atributos de la matriz y se imprime. Se pueden apreciar dos formas de obtener la matriz completa sin distribuir. En la línea 38 se obtiene en un puntero que es un atributo del objeto. En la línea 39 y 40 se reserva memoria y se obtiene en dicho puntero. Por último en la línea 41 se imprime.

Algoritmo 13 Código ejemplo de la librería en C++

```
1  //Creación de objetos
2  NcclMultiplicationEnvironment<double> ncclMultEnv =
       NcclMultiplicationEnvironment<double>(gpuSizeWorldArgument, gpuRoot, opt,
       debugMatrix);
3  MatrixMain<double> ma = MatrixMain<double>(&ncclMultEnv, rowsA, columnsA,
       matrixA);
4  MatrixMain<double> mb = MatrixMain<double>(&ncclMultEnv, rowsA, columnsA);
5  mb.setMatrixHostToFullValue(1);
6  // mb.setMatrixHost(matrixA);
7  //Multiplicaciones
8  ma=ma*mb;
9  ma*=ma;
10 ma.setAlphaGemm(3);
11 ma=ma*mb;
12 ma.setAlphaGemm(1);
13 //Sumas y restas entre matrices
14 ma=ma+ma;
15 ma+=mb;
16 ma=ma-mb;
17 ma-=mb;
18 ma.axpy(2,mb);
19 //Sumas y restas con la identidad
20 ma=mb+1;
21 ma=1+mb;
22 ma+=1;
23 ma=mb-1;
24 ma=1-mb;
25 ma-=1;
26 //Operaciones con escalares
27 ma=ma/3;
28 ma/=3;
29 ma=ma*6;
30 ma*=ma;
31 //norma 1
32 double resNorm1=ma.norm1();
33 //Cambio de signo
34 ma=-ma;
35 //Obtención de atributos
36 int rowsC=ma.getRowsReal();
37 int columnsC=ma.getColumnsReal();
38 double *distributedRes=ma.getHostMatrix();
39 //double *distributedRes=MatrixUtilitiesCuda<double>::matrixMemoryAllocationCPU
       (rowsC, columnsC);
40 //ma.getHostMatrixInThisPointer(distributedRes);
41 MatrixUtilitiesCuda<double>::printMatrix(rowsC, columnsC, distributedRes);
```

Si se desea conocer más detalles de la librería se pueden consultar en la documentación que se ha generado a partir de DOxygen [29] y que se puede leer en [30].

6.8 Compilación e instalación de la librería

Para compilar la librería tan sólo es necesario usar la herramienta CMake, la cual se encargará de encontrar todas las dependencias en el sistema. Se puede generar un nuevo objetivo personalizado modificando el CMakeList proporcionado. Otra opción consiste en instalar en el sistema la librería mediante `make install` en un directorio personalizado que previamente ha debido de ser especificado mediante `−DCMAKE_INSTALL_PREFIX=path` al usar CMake. Por defecto se instala en una carpeta denominada bin en el directorio del archivo CMakeList.

CAPÍTULO 7
Utilización de la librería desde Matlab

El propósito de esta librería es proporcionar una alternativa de cálculo para los métodos discutidos en [3]. Estos métodos tienen por objeto calcular una función matricial (seno, coseno, tangente hiperbólica, etc.). Los métodos propuestos se basan en la evaluación de un polinomio que contiene los primeros términos de la serie de Taylor (entre otras). Por tanto, computacionalmente se trata de productos matriciales. Antes de evaluar el polinomio, su construcción constituye el primer paso. Para llevar a cabo el diseño del polinomio, es decir, calcular el grado del mismo, el escalado de las matrices, etc., es necesario conocer las normas matriciales que tendrá cada potencia de la matriz cuando sea calculada. Dado que este cálculo es costoso y no se conoce hasta que, efectivamente, es calculada la potencia matricial, lo que se realiza es obtener una aproximación a dicho valor, una *estimación*, por tanto, la *estimación de norma* es una técnica que permite aproximar este valor. Desde el punto de vista computacional, para el propósito de este trabajo, utilizar estimación de norma significa reducir el número de operaciones pero a costa de aumentar el trasiego de información GPU-CPU dado que, en principio, la estimación de norma se realiza en CPU.

La librería desarrollada aquí toma como esqueleto el de la ya utilizada en [3] para una GPU.

7.1 Compilación y ejecución

Para la compilación de esta librería se ha continuado usando la herramienta CMake. Se ha modificado el fichero CMakeLists.txt para que compile el fichero .mex, tal y como se muestra en [31], indicándole un nuevo objetivo y que solo genere los ficheros necesarios en caso de que se indique, mediante un parámetro, que se desea generar la librería para Matlab. Si CMake no encuentra el directorio de Matlab también será necesario indicarle dónde está. Un ejemplo de compilación de la librería para Matlab sería el siguiente: mkdir build && cd build && cmake .. −DBUILD_FOR_MATLAB=ON −DMatlab_ROOT_DIR="/usr/local/MATLAB/R2018a" && make call_gpu.

En cuanto a la ejecución, puede que, dependiendo de la versión de Matlab y del sistema operativo, sea necesario lanzar Matlab con LD_PRELOAD. Para saber si es necesario, tan solo hará falta ejecutar las llamadas de call_gpu listadas en el Algoritmo 14 y, en caso de que se muestre un error de falta de símbolos, es que será necesario. Un ejemplo de línea de lanzamiento que ha sido necesaria en Nowherman para la versión 2015b de Matlab ha sido: LD_PRELOAD="/usr/local/lib64/libstdc++.so.6" matlab, mientras que en otro equipo de

Algoritmo 14 Código ejemplo de la librería en Matlab.

```
1 call_gpu('init',f,metodo_f,A);
2 a(1)=call_gpu('norm1',1);
3 pA{2}=call_gpu('power');
4 nProd=call_gpu('evaluate',p);
5 call_gpu('scale',s);
6 call_gpu('unscale',s);
7 call_gpu('free',1);
8 fA = call_gpu('finalize');
9 call_gpu('destroy');
```

desarrollo con la versión 2018a ha sido necesario: LD_PRELOAD=/usr/lib/x86_64−linux−gnu/libnccl.so.2 /usr/local/MATLAB/R2018a/bin/matlab.

7.2 Explicación de uso

Con la librería ya lista para su uso, tan solo hará falta llamar al fichero mex generado mediante instrucciones de código Matlab donde el primer argumento es la especificación de la operación que se desea ejecutar. A continuación, se proporciona una explicación con referencia al Algoritmo 14 de los elementos que aparecen ahí:

- a es un vector unidimensional.
- A es la matriz inicial.
- f es la función a aplicar sobre la matriz('exp', 'cos' o 'cosh'), especificada como una cadena de caracteres.
- metodo_f es el método para calcular fA. Los posibles valores son: 'taylor', 'bernoulli' o 'hermite'.
- s son números por los cuales se va a escalar o desescalar las matrices.
- nProd indica el número de productos realizados por el método llamado evalPatMey, que evalúa un polinomio matricial mediante el método de Paterson-Stockmeyer [32].
- La línea 1 es la invocación al constructor de la librería o, en su defecto, si ya había sido ejecutada con anterioridad se hará un *casting* mediante reinterpret_cast de C++ al nuevo tipo de esta. La línea 8 se encarga de limpiar la librería con todas las matrices almacenadas. En cuanto a la línea 9, esta se encarga de destruir la librería por completo. Con esta llamada se eliminan los comunicadores y el resto de objetos auxiliares necesarios para realizar los cálculos quedando liberados todos los recursos para que pueda ejecutarse la aplicación de nuevo.
- El argumento que acompaña a 'free' indica cuántas matrices se van a liberar de memoria desde el final del vector que almacena las potencias de matrices.

7.3 Rendimiento

Seguidamente se ha analizado el rendimiento de la librería. Para ello, se han empleado los métodos de Taylor y Bernoulli para la función exponencial con la modalidad de

Tabla 7.1: Tabla de rendimiento Matlab con estimación de norma. Unidades en segundos.

	Matlab-CPU		1-GPU		4-GPU	
Tamaño	Taylor	Bernoulli	Taylor	Bernoulli	Taylor	Bernoulli
1000	0.14	0.10	0.16	0.10	0.27	0.20
2000	0.88	0.52	0.27	0.29	0.53	0.48
3000	1.31	1.04	0.65	0.64	1.04	1.04
4000	2.47	2.23	1.28	1.21	1.57	1.71
5000	4.14	4.44	2.34	1.91	2.50	2.52
6000	6.50	7.24	3.26	2.97	3.21	3.40
7000	10.65	10.35	4.21	3.86	4.16	4.70
8000	14.13	15.41	5.33	5.16	5.75	5.50
9000	20.49	18.05	6.94	6.92	7.32	7.13
10000	23.82	25.41	8.57	9.06	9.32	8.49
11000	34.94	31.53	10.84	10.59	10.87	10.71
12000	39.25	39.34	12.75	12.78	11.92	12.37
13000	49.52	45.63	16.36	16.31	14.85	14.56
14000	55.80	54.17	19.20	19.35	17.00	16.97

con estimación de norma y *sin estimación de norma* empleados en [3]. En ambos casos siempre se ha obtenido una m de valor 20. Esta m indica el número de términos que tendrá la función. Este hecho tiene implicaciones importantes en las prestaciones que es necesario explicar. Este parámetro, como se ha indicado, representa el número de términos de la serie (de Taylor o de Bernoulli) y, por tanto, cuanto mayor es, mayor es el coste. La naturaleza de las matrices utilizadas (aleatorias) implica que la estimación de norma no ofrezca un parámetro m menor que si no se estima, por tanto, el coste de computación será el mismo. Sin embargo, la estimación tiene su propio coste. En el caso de ejecución en CPU no afecta mucho, pero en el caso de ejecución en GPU está muy influenciado por la transferencia de datos GPU-CPU. Esto explica las diferencias de coste en GPU.

En la Tabla 7.1, la Figura 7.1 y Figura 7.2 se pueden apreciar los resultados obtenidos para la modalidad *con estimación de norma*.

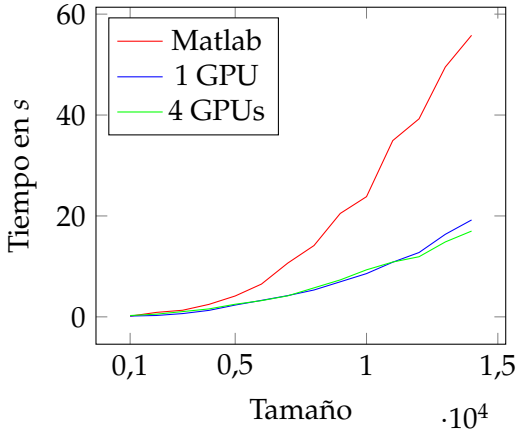


Figura 7.1: Gráfica de rendimiento Matlab Taylor con estimación de norma.

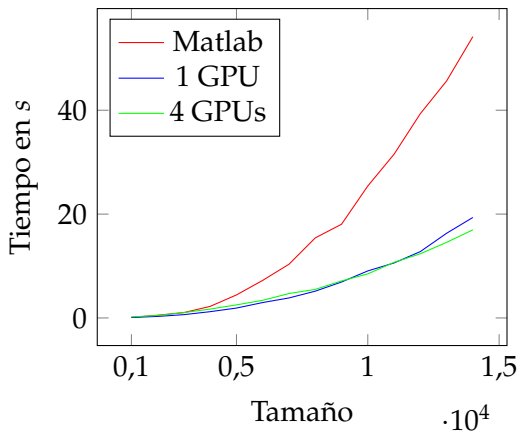


Figura 7.2: Gráfica de rendimiento Matlab Bernoulli con estimación de norma.

A continuación, en la Tabla 7.2, la Figura 7.3 y la Figura 7.4 se pueden ver los resultados obtenidos para la modalidad *sin estimación de norma*.

Tabla 7.2: Tabla de rendimiento Matlab sin estimación de norma. Unidades en segundos.

	Matlab-CPU		1-GPU		4-GPU	
Tamaño	Taylor	Bernoulli	Taylor	Bernoulli	Taylor	Bernoulli
1000	0.15	0.11	0.19	0.18	0.24	0.24
2000	0.61	1.52	0.36	0.35	0.39	0.37
3000	1.23	1.22	0.64	0.61	0.64	0.59
4000	1.92	2.16	0.96	0.96	0.95	0.89
5000	4.15	4.18	1.44	1.41	1.30	1.26
6000	6.38	6.07	1.97	1.94	1.70	1.68
7000	9.56	9.51	2.64	2.69	2.20	2.32
8000	13.67	13.90	3.48	3.52	2.77	2.78
9000	17.97	17.16	4.52	4.45	3.62	3.46
10000	24.16	23.35	5.79	5.65	4.38	4.32
11000	31.21	30.91	7.14	6.95	5.34	5.20
12000	36.37	36.72	8.74	8.77	6.13	6.04
13000	46.65	47.96	11.01	10.56	7.18	7.04
14000	54.36	54.67	12.79	12.55	8.35	8.25

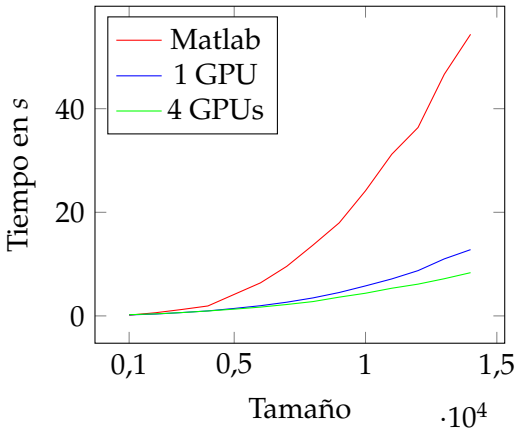


Figura 7.3: Gráfica de rendimiento Matlab Taylor sin estimación de norma.

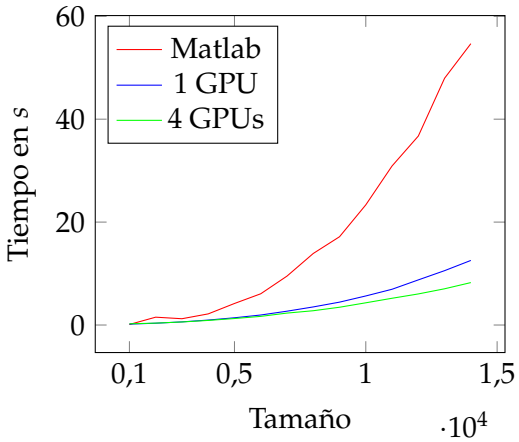


Figura 7.4: Gráfica de rendimiento Matlab Bernoulli sin estimación de norma.

De ambos análisis se pueden sacar diversas conclusiones. La primera de ellas, que la variante *sin estimación de norma* rinde considerablemente mejor que la variante *con estimación de norma*. Esto se debe a lo explicado anteriormente. Es necesario utilizar otro tipo de matrices para encontrar diferencias que actúen a favor de la *estimación de norma* pero que se encuentran fuera del ámbito de este trabajo. También se puede ver que, tanto Taylor como Bernoulli, tardan un tiempo similar en ambas variantes. Las diferencias entre estos dos métodos son dependientes también de la matriz objetivo y esto también se encuentra fuera de nuestro ámbito, salvo por el hecho de que el comportamiento en GPU es el mismo que en CPU.

Desde nuestro punto de vista, es más interesante la diferencia de rendimiento del sistema multi-GPU. La diferencia con respecto a una única GPU ha resultado estar por debajo de lo esperado, siendo lo esperado algo parecido a lo obtenido en la multiplicación de matrices en C++ (Capítulo 5). Existen varios motivos, entre otros, que se pierde una parte importante del tiempo en la distribución de las matrices para ejecutar el SUMMA.

Aún así, el incremento de velocidad llega a ser de $1,5\times$ para tamaños grandes para el caso *sin estimación de norma*.

Cabe destacar en favor de la versión implementada con cuatro tarjetas gráficas que esta permite multiplicar matrices más grandes que la versión con una única GPU. Mientras que la versión que cuenta con una única tarjeta gráfica permite realizar el cálculo con un tamaño máximo de 14000, la versión desarrollada es capaz de alcanzar un tamaño mucho mayor, 30000. En este último caso, el tiempo empleado en una ejecución de Taylor *sin estimación de norma* ha sido de 445,66*s* mientras que la versión con cuatro GPUs ha tardado 4,49*s*. Si calculamos la aceleración, será de 9,8 frente a 6,51 para el caso de 14000 de tamaño. Por lo que se ve que la aceleración va en aumento cuando las matrices se van haciendo más grandes.

7.4 Conclusiones

Hemos ofrecido en este capítulo una interfaz Matlab que permita aprovechar el esfuerzo realizado en capítulos anteriores. Con esta propuesta, cualquier científico que utilice Matlab puede aprovechar una configuración como la de Nowherman de forma sencilla. Es necesario, sin embargo, realizar una investigación futura que permita entender mejor cuál es la influencia de cada una de las operaciones involucradas en este cálculo para poder mejorar la eficiencia obtenida, por debajo de la esperada inicialmente.

CAPÍTULO 8
Conclusiones y trabajo futuro

En este trabajo de fin de grado se ha diseñado un método para realizar la multiplicación de matrices de forma paralela con GPUs mediante el algoritmo SUMMA y ofrecer una alternativa de cálculo de funciones de matrices para Matlab.

En una primera aproximación se ha realizado la implementación del algoritmo mediante MPI tal y come se ha visto en el Capítulo 4. Ha sido necesario reforzar los conocimientos en MPI y C++. También se ha aprendido a usar BLAS. Tras haber conseguido de forma satisfactoria el objetivo comentado con anterioridad se pasó a la implementación mediante GPUs tal y como se ve en el Capítulo 5. Para conseguir el objetivo se ha tenido que aprender CUDA y la librería de comunicaciones NCCL. Con motivo de conseguir ejecutar las funciones de matrices, ha sido imprescindible implementar otras operaciones matriciales tal y como se puede consultar en el Capítulo 6.

Ya con el desarrollo principal acabado, se da paso a explicar el uso de la librería en su lenguaje nativo en la Sección 6.7. Posteriormente, se explica la funcionalidad de la librería desarrollada para Matlab en el Capítulo 7. En ambos capítulos se explica la compilación y uso mediante CMake.

El desarrollo del software ha sido exigente dado que muchas de las herramientas utilizadas para realizar este trabajo no se habían estudiado previamente en la carrera. Todas las tecnologías (lenguajes, librerías, comandos, ...) han sido adecuadas y han permitido abordar el trabajo satisfactoriamente de la manera adecuada, es decir, podría haberse utilizado C, sin embargo, el esfuerzo realizado al utilizar C++ ha valido la pena, dadas las características interesantes que posee este lenguaje.

En cuanto a rendimiento se refiere, en el Capítulo 4 se trata de estudiar cómo varía el rendimiento de SUMMA con el clúster Kahan. A continuación, se observará cómo varía el rendimiento en el nodo Nowherman equipado con cuatro NVIDIA TESLA P100. Luego, se ve cómo afecta en ese mismo sistema el rendimiento con la librería de Matlab. En estos últimos casos también se aprecia la capacidad de realizar multiplicaciones de tamaño mucho mayor.

Se considera positiva la experiencia de optimización del código en cada caso en el que se abordado. A veces, el rendimiento no ha sido el esperado, sin embargo, el reto ha sido motivador y lo sigue siendo puesto que han surgido ideas de mejora futura de todo lo desarrollado en este trabajo.

Por otro lado, las matrices son un elemento matemático muy usado en la actualidad. Es por ello que este trabajo tiene diversas vías de ampliación para mejorar el soporte que se dan a estas estructuras de datos así como mejorar aun más el rendimiento. En los apartados que se encuentran en este capítulo se exponen las mejoras que sería posible desarrollar en un futuro.

8.1 Versión conjunta de MPI y NCCL con capacidad multi-nodo

Actualmente el programa es capaz de ejecutarse satisfactoriamente en un único nodo con distintas configuraciones de dispositivos. Para poder aumentar la potencia de computo sería necesario aumentar la potencia de cada tarjeta gráfica o aumentar el número de ellas. Con motivo de que a veces es imposible por tener el máximo número de GPUs posibles y el mejor modelo del mercado, sería sensato dar soporte a ejecuciones multi-nodo y así formar un clúster. Para conseguir tal objetivo sería necesario que las tecnologías MPI y NCCLs trabajen de forma conjunta. En esta simbiosis MPI se encargaría de transmitir la información a los distintos *host*s cuando sea necesario, por ejemplo al inicio de una multiplicación que no esté todavía distribuida y una tarjeta gráfica no tenga memoria suficiente para almacenarla y enviarla mediante NCCL. Esta última tecnología entraría en juego de forma segura en la ejecuión del algoritmo SUMMA.

8.2 Soporte para arquitecturas con Tensor Cores

Cada día que pasa la tecnología mejora cada vez más. Las últimas tarjetas gráficas de NVIDIA tienen innovaciones en su arquitectura que favorecería el cálculo de operaciones con cuBLAS. Una de estos últimos avances son los *Tensor Cores* que se pueden encontrar a partir de la arquitectura Volta o Turing y que no hemos podido usar debido a que en el desarrollo de la librería solo hemos contado con gráficas de la arquitectura Pascal. En [33] se puede ver que las distintas instrucciones de cálculo usadas para cálculos matemáticos pueden mejorar su rendimiento.

8.3 Ampliación de tipos soportados

La versión actual del programa soporta operaciones con tipos de números reales como `float` y `double`. Son tipos que sirven para cálculos generales, sin embargo se pueden realizar mejoras en este aspecto.

Existe la posibilidad de incluir soporte para números decimales de media precisión porque CUDA tiene soporte para ellos con el tipo `__half`. Sin embargo, C++ carece de soporte oficial y habría que estudiar con cuidado qué cambios serían necesarios realizar.

Además, sería de gran interés incluir en un futuro soporte para números complejos de simple y doble precisión. Estos números son de gran utilidad en diversos campos de ingeniería y científicos y, por tanto, un potencial campo de ampliación de la librería.

8.4 Ampliación de las operaciones soportadas

Una operación que es evidente que necesitaría un mejor soporte, sería la suma y resta entre matrices, ya que en la actual versión solo está soportada si la malla de GPUs es cuadrada tal y como se explica en la Sección 6.5. Recuperar y redistribuir las matrices en caso de que lo estén y las dimensiones de sus bloques sean distintos parece una opción viable, aunque habría que ver cómo afecta esa decisión a la operación angular, la multiplicación. Podría ser necesario volver a recuperar y redistribuir después otra vez para satisfacer las dimensiones de las multiplicaciones. Otra posible opción sería que las gráficas se comu-

nicaran y se pasasen los datos necesarios y los devolviesen. Se puede ver que no es un problema trivial y necesita un estudio para elegir la mejor opción entre todas las posibles.

Por otro lado, las matrices tienen múltiples tipos de operaciones. Es evidente que se puede aumentar la agrupación de operaciones con funciones que permitan calcular cosas sobre dichas matrices, como pueden ser normas y la matriz traspuesta o la traspuesta conjugada. Puesto que las operaciones de matrices con vectores son muy comunes sería posible expandir con estos elementos el conjunto de instrucciones soportadas. También interesaría integrar la operación tensorial entre matrices. Esto último, añadido junto con el de soporte de números complejos discutido en la Sección 8.3, permitiría simular circuitos de computación cuántica con un gran rendimiento.

8.5 Disponibilidad de librerías

Finalmente, como última ampliación podría mejorarse la compatibilidad del programa con distintos lenguajes. Últimamente están de moda lenguajes como Python o R, por ello sería de gran beneficio para los programadores de estos lenguajes que deseen usar cálculos de estas características tener soporte. Como se ha visto en Capítulo 7 no es excesivamente complicado conseguir que la librería funcione en otros lenguajes.

Bibliografía

[1] R. A. Van De Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency Practice and Experience*, 1997.

[2] MATLAB - El lenguaje del cálculo técnico - MATLAB & Simulink. https://es.mathworks.com/products/matlab.html. Última visitia: 28-05-2020.

[3] Pedro Alonso, Jesús Peinado, Javier Ibáñez, Jorge Sastre, and Emilio Defez. Computing matrix trigonometric functions with GPUs through Matlab. *Journal of Supercomputing*, 75(3):1227–1240, 2019.

[4] *C++ Reference*. http://www.cplusplus.com/reference/.

[5] *CMake Reference Documentation*. https://cmake.org/cmake/help/v3.17/.

[6] Repositorio tfgMatrixNccl. https://github.com/rodhuega/tfgMatrixNccl. Última visita: 30-06-2020.

[7] *Open MPI v4.0.3 documentation*. https://www.open-mpi.org/doc/current/.

[8] Cluster Kahan. http://personales.upv.es/jroman/kahan.html. Última visita: 29-04-2020.

[9] Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/. Última visita: 1-07-2020.

[10] MPICH. https://www.mpich.org/. Última visita: 1-07-2020.

[11] *CUDA Toolkit Documentation*. https://docs.nvidia.com/cuda/.

[12] OpenCL Overview - The Khronos Group Inc. https://www.khronos.org/opencl/. Última visita: 03-06-2020.

[13] TOP500. The List. https://www.top500.org/lists/top500/2020/06/. Lista de Junio de 2020.

[14] Interconexión de GPU de alta velocidad mediante NVLink. https://www.nvidia.com/es-es/design-visualization/nvlink-bridges/. Última visita: 28-05-2020.

[15] *NVIDIA Collective Communication Library (NCCL) Documentation — NCCL 2.6.4 documentation*. https://docs.nvidia.com/deeplearning/sdk/nccl-archived/nccl{_}264/nccl-developer-guide/docs/index.html.

[16] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2020.

[17] Gene H. Golub and Charles F. van Loan. *Matrix Computations*. JHU Press, fourth edition, 2013.

[18] *BLAS (Basic Linear Algebra Subprograms)*. http://www.netlib.org/blas/.

[19] Intel® Math Kernel Library. https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html. Ultima visita: 2020-05-28.

[20] *cuBLAS :: CUDA Toolkit Documentation*. https://docs.nvidia.com/cuda/cublas/index.html.

[21] Philippe Estival and Luc Giraud. A fight for performance and accuracy of the matrix multiplication routines : CUBLAS on Nvidia Tesla versus MKL and ATLAS on Intel Nehalem. Technical report, CERFACS, 2012.

[22] Ammar Ahmad Awan, Hari Subramoni, Ching Hsiang Chu, and Dhabaleswar K. Panda. Optimized broadcast for deep learning workloads on dense-GPU InfiniBand clusters: MPI or NCCL? *ACM International Conference Proceeding Series*, 2018.

[23] NCCL Release Notes :: NVIDIA Deep Learning SDK Documentation. https://docs.nvidia.com/deeplearning/sdk/nccl-release-notes/index.html. Ültima visita: 28-04-2020.

[24] Building Cross-Platform CUDA Applications with CMake | NVIDIA Developer Blog. https://devblogs.nvidia.com/building-cuda-applications-cmake/. Última visita: 28-04-2020.

[25] Intel mkl cmake. https://repository.prace-ri.eu/git/CodeVault/hpc-kernels/structured_grids/-/blob/bafad6bfbc46012d23b9e9917ceac76a5ac39972/cmake/Modules/FindMKL.cmake. Última visita: 17-05-2020.

[26] How to Implement Performance Metrics in CUDA C/C++ | NVIDIA Developer Blog. https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/. Última visita: 30-042020.

[27] Nvidia. *Curand library*, 2019.

[28] *Profiler :: CUDA Toolkit Documentation*. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[29] *Doxygen: Main Page*. https://www.doxygen.nl/index.html.

[30] *NCCL Matrix Multiplication*. https://rodhuega.github.io/tfgMatrixNccl/doc/html/.

[31] FindMatlab — CMake 3.17.2 Documentation. https://cmake.org/cmake/help/latest/module/FindMatlab.html. 17-05-2020.

[32] M.S. Paterson and L.J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM J. Comput.*, 2(1):60–66, 1973. cited By 143.

[33] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance & precision. In *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2018*, pages 522–531. Institute of Electrical and Electronics Engineers Inc., aug 2018.

Glosario

buffer Memoria temporal de información. 24

host Término usado para referirse a instrucciones en la CPU y que usan como memoria la RAM. 14, 20, 23, 26, 31–34, 44

root Término usado para referirse al proceso o GPU líder en una operación colectiva. 32, 34

BLAS Basic Linear Algebra Subprograms es una librería que ejecuta con un gran rendimiento operaciones de álgebra lineal. 6, 15, 17, 19, 22, 29, 32, 34, 43, 49, 51

clúster Conjunto de servidores que actúa como una única entidad para realizar cálculos. 2–5, 11, 15, 17, 43, 44, 49

color **1.** Significado original. **2.** Término que se refiere para distinguir elementos de distintas filas o columnas. 15, 21, 32

comunicador Elemento que se encarga de conectar a un grupo de procesos. 4, 15, 21, 22

cuBLAS Librería BLAS en CUDA. 6, 12, 19, 22, 32, 33, 44

CUDA Compute Unified Device Architecture es una plataforma de computación paralela para tarjetas gráficas NVIDIA. 4–6, 19, 43, 44, 49, 51

dispositivo Término usado para referirse a la tarjeta gráfica. 14, 20, 21, 26, 31–34, 44

genericidad Característica que permite que el código funcione con distintos tipos de datos. 23

malla Estructura de los procesos en un sistema distribuido. 11, 12, 15, 20, 22, 24, 29, 33

NCCL NVIDIA Collective Communications Library es una librería de comunicaciones colectivas de alto rendimiento entre tarjetas gráficas de la marca NVIDIA que usa un estilo parecido a MPI. 5, 11, 17, 19, 21, 22, 32, 34, 43, 44, 51

nodo Miembro de un clúster. 3–5, 16, 26, 43, 44

rango Identificador del proceso dentro de un grupo. 12, 21, 22

Siglas y acrónimos

BLAS Basic Linear Algebra Subprograms. 6, 15, 17, 19, 22, 29, 32, 34, 43, 49

CUDA Compute Unified Device Architecture. 4–6, 19, 43, 44, 49

NCCL NVIDIA Collective Communications Library. 5, 11, 17, 19, 21, 22, 32, 34, 43, 44, 49

CPU *Central processing unit*. 5, 7, 17, 22, 32, 49

FLOP *FLOating Point operation*. 7, 9

GEMM *General matrix multiply*. 6, 12, 15, 19, 30, 34

GPGPU *General Purpose Graphics processing unit*. 4, 5

GPU *Graphics processing unit*. 1, 2, 4–6, 12, 14, 19–25, 27, 29, 30, 32, 34, 40, 41, 43, 44, 49

MKL *Math Kernel Library*. 4, 6, 15, 16, 19, 26

MPI Message Passing Interface. 2, 4, 7, 11, 14–17, 20–22, 43, 44, 49

OpenCL *Open Computing Language*. 4

PCI *Peripheral Component Interconnect*. 5, 19, 20

RAM *Random Access Memory*. 5, 11, 49

SIMD *Single Instruction, Multiple Data*. 6

SUMMA *Scalar Universal Matrix Multiplication Algorithm*. 1–3, 7, 9, 17, 20–24, 26, 27, 30, 40, 43, 44

TORQUE *Terascale Open-source Resource and Queue Manager*. 4