UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Uso de formatos no convencionales para matrices dispersas en GPUs

Renzo Marini Salsamendi

Programa de Grado en Ingeniería en Computación
Facultad de Ingeniería
Universidad de la República

Montevideo – Uruguay
Abril de 2021

UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Uso de formatos no convencionales para matrices dispersas en GPUs

Renzo Marini Salsamendi

Tesis de Grado de Ingeniería en Computación, presentada al tribunal evaluador de la Facultad de Ingeniería de la Universidad de la República, como parte de los requisitos necesarios para la obtención del título de Ingeniero en Computación

Cotutores:
Dr. Ernesto Dufrechou
Dr. Pablo Ezzatti

Montevideo – Uruguay
Abril de 2021

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE TESIS

Dr. Gustavo Betarte

Dr. Martín Pedemonte

Dra. Aiala Rosá

Montevideo – Uruguay
Abril de 2021

Agradecimientos

En primer lugar, quisiera agradecer a mis cotutores, Ernesto Dufrechou y Pablo Ezzatti, por guiarme en el transcurso de este trabajo mediante críticas constructivas y sugerencias de las cuales aprendí mucho.

Me siento extremadamente afortunado por el soporte que recibí de mi familia y amigos. En particular, me gustaría agradecer a mis padres y hermanos por acompañarme y creer en mí en todo momento. A Gono, Maxi y Nico quisiera agradecerles por el compañerismo y por todas las formas en las que enriquecieron mi experiencia dentro y fuera de facultad. También me gustaría darles las gracias a Giorgio y Florencia por apoyarme y festejar mis logros durante la carrera.

RESUMEN

Una tendencia de las últimas décadas en el diseño de arquitecturas de computadoras es el desarrollo de procesadores orientados al *throughput*. Uno de los ejemplos especialmente prominentes es el de las GPUs (Graphics Processing Units) modernas. Este fenómeno, junto a la creación de circuitos integrados de aplicación específica como los Tensor Cores de NVIDIA y los TPUs (Tensor Processing Units) de Google, han sido de vital importancia en la aceleración de aplicaciones científicas. En particular, el hardware mencionado se utiliza en el contexto del álgebra lineal numérica dispersa para acelerar la operación de multiplicación de matrices dispersas (SpMM).

En este trabajo se presenta una descripción conceptual del formato de almacenamiento disperso bmSPARSE y de algoritmos para computar SpMM de forma eficiente utilizando dicho formato. Luego se describen y evalúan distintas implementaciones en GPU, haciendo énfasis en optimizaciones basadas en Tensor Cores.

Palabras claves:
GPU, Tensor Cores, bmSPARSE, SpMM.

Tabla de contenidos

Capítulo 1

Introducción

En las últimas décadas la tendencia en el diseño de arquitecturas de computadoras ha sido incorporar múltiples cores en un mismo chip. Esto se debe principalmente a que las mejoras de desempeño obtenidas al incrementar la frecuencia de reloj de un procesador de un sólo núcleo no justifican el incremento en el consumo de energía asociado[1]. Una tendencia relacionada es la prominencia de los procesadores orientados al throughput[2], dentro de los cuales se encuentran las GPUs. Estas arquitecturas son usadas frecuentemente para acelerar aplicaciones científicas. En particular, en el contexto del álgebra lineal densa se han realizado implementaciones eficientes de la especificación BLAS[3] en GPU[4], que dentro de sus operaciones incluye la multiplicación de matrices (GEMM). La estructura regular de las matrices permite que las tareas a realizar sean subdivididas en tareas pequeñas que pueden asignarse a distintos recursos de cómputo de la GPU y procesarse en paralelo. Adicionalmente, el amplio uso de algoritmos de aprendizaje profundo, en donde GEMM es un componente esencial, motivó el desarrollo de circuitos integrados de aplicación específica (ASICs) que son utilizados como aceleradores. Un ejemplo de este fenómeno son los *Tensor Cores*, una unidad funcional de punto flotante de precisión mixta incorporada en 2017 por NVIDIA en su nueva microarquitectura Volta[5]. La misma está optimizada para realizar la operación matricial $D = A \times B + C$, en donde A, B, C y D son matrices pequeñas de dimensiones predeterminadas[6]. Otro de los aceleradores en esta categoría son los TPUs de Google[7].

La multiplicación de matrices dispersas (SpMM) opera sobre dos matrices almacenadas en un formato de almacenamiento disperso. SpMM es una componente importante en aplicaciones como solvers para métodos multigrid algebraicos (AMG)[8], conteo de triángulos[9] y búsqueda en anchura (BFS) con múltiples orígenes[10]. La

diferencia principal que la distingue de la variante densa es que una implementación eficiente debe tener en consideración la estructura irregular de los datos. Las matrices dispersas están compuestas mayoritariamente por coeficientes con valor cero, por lo que al implementar SpMM suelen utilizarse estrategias para no desperdiciar recursos de cómputo y memoria en elementos nulos de la matriz resultante. Una distinción que puede hacerse entre los formatos dispersos es según si las posiciones almacenadas hacen referencia a elementos individuales o a un conjunto de elementos de la matriz original. En la segunda variante, la matriz es subdividida en bloques densos de elementos y se almacena información de un bloque cuando éste contiene algún elemento no nulo.

En 2018, Zhang y Gruenwald[11] presentaron un formato disperso a bloques, el cual denominaron bmSPARSE, que adapta la técnica de indexado de mapa de bits usada en el contexto de bases de datos relacionales[1]. Si bien se ha estudiado el desempeño de formatos similares en el contexto de SpMV[12], el trabajo mencionado representa la primera instancia que lo hace en relación a SpMM. Los resultados obtenidos fueron favorables, ya que se obtuvo un desempeño superior al de las bibliotecas CUSP y bhSPARSE en el conjunto de datos considerado.

Este trabajo se centra en implementar y evaluar distintas modificaciones del algoritmo básico propuesto por Zhang y Gruenwald. Entre las optimizaciones consideradas, se explora la utilización de Tensor Cores para acelerar la multiplicación de bloques. Si bien la literatura sobre el uso de Tensor Cores en el contexto de SpMM es prácticamente inexistente, estas unidades fueron diseñadas para multiplicar eficientemente bloques densos pequeños, lo cual vuelve prometedor su uso en conjunto a formatos dispersos a bloques. Por otro lado, se ha observado un buen desempeño en la multiplicación mediante Tensor Cores incluso en situaciones en las que los bloques de entrada no son utilizados en totalidad[13].

El resto del documento está estructurado de la siguiente forma. El Capítulo 2 introduce nociones básicas de arquitecturas, con especial énfasis en arquitecturas paralelas y GPUs, aplicaciones de matrices dispersas y la operación SpMM. El Capítulo 3 describe el formato bmSPARSE y el algoritmo SpMM asociado. En este último caso, se brinda tanto una descripción conceptual como detalles de implementación de las distintas variantes propuestas. El Capítulo 4 consiste en una evaluación experimental que aborda tanto el consumo de memoria del formato como el tiempo de ejecución de distintas implementaciones del algoritmo SpMM basado en bmSPARSE. El Capítulo 5 cumple una función similar, pero considera únicamente im-

[1] Y, en forma histórica, también para formatos dispersos.

plementaciones que hacen uso de Tensor Cores. Por último, el Capítulo 6 sintetiza los resultados obtenidos y plantea líneas de trabajo futuro.

Capítulo 2

Conceptos de base

Este capítulo introduce nociones básicas de GPU, de uso de matrices dispersas en general y de la operación SpMM en particular. La Sección 2.1 describe técnicas a nivel de hardware que son utilizadas para implementar distintas formas de paralelismo. La Sección 2.2 introduce la taxonomía de computadoras de Flynn, la cual puede ser utilizada para clasificar arquitecturas en un conjunto reducido de clases. Luego se presentan dos filosofías de diseño de procesadores que motivan muchas de las decisiones de compromiso realizadas en el diseño de las GPUs. La Sección 2.3 describe la evolución histórica de las GPUs y brinda un panorama general tanto de la arquitectura de una GPU moderna como del modelo de programación asociado. Adicionalmente, se la compara con otras clases de arquitectura y se explican algunas de las principales decisiones de diseño realizadas por los fabricantes. La Sección 2.4 presenta formatos de almacenamiento de matrices dispersas y sus características. Por último, la Sección 2.5 describe conceptualmente la operación SpMM y realiza un análisis de trabajos previos realizados en el área.

2.1 Técnicas de paralelismo

En el contexto de aplicaciones de software, las distintas formas de paralelismo pueden separarse en dos grandes categorías: paralelismo de datos y de tareas. El primero ocurre cuando se puede operar sobre múltiples datos simultáneamente, mientras que el segundo consiste en ejecutar unidades de trabajo independientes en paralelo[14]. A continuación se introducen distintas técnicas implementadas a nivel de hardware para dar soporte a las formas de paralelismo mencionadas.

El paralelismo de instrucciones es la capacidad de un sistema de superponer la

ejecución de múltiples instrucciones perteneciente a un bloque de código secuencial. Una de las técnicas principales para implementar esta clase de paralelismo es el *pipeline* de instrucciones[15]. Conceptualmente, esta técnica consiste en dividir el sistema en k etapas, con registros asociados a cada etapa. Las instrucciones se dividen en k partes distintas que son resueltas por las distintas etapas del pipeline, lo cual permite que se pueda estar ejecutando en paralelo varias instrucciones en distintas etapas. Otra forma de obtener paralelismo de instrucciones es mediante procesadores superescalares, que permiten ejecutar más de una instrucción por ciclo de reloj[16]. A pesar de que suelen implementarse en conjunto, es una técnica distinta al pipelining. Los procesadores superescalares logran ejecutar instrucciones en paralelo a través de múltiples unidades funcionales, mientras que pipelining divide a las instrucciones en etapas y superpone la ejecución de etapas distintas, algo que puede lograrse sin necesidad de disponer de más de una unidad funcional en la etapa de ejecución. Al implementar ambas técnicas surgen distintas posibilidades. Algunas arquitecturas replican todo el pipeline por cada unidad funcional disponible[17], sin embargo, una implementación más frecuente consiste en aumentar la frecuencia del reloj del procesador, mantener un único pipeline e incorporar varias unidades funcionales[16]. Por último, la ejecución especulativa es una optimización de hardware que explota el paralelismo de instrucciones a través de la ejecución de instrucciones sin antes conocer si las mismas deben ser ejecutadas o no[14]. En este modelo de ejecución, los resultados sólo se actualizan en memoria principal una vez que se tiene la certeza de que la instrucción no es especulativa.

Multithreading es una técnica en la que múltiples threads ejecutan de forma concurrente en un mismo procesador sin necesidad de hacer un cambio de contexto a nivel de procesos para ejecutar instrucciones de threads distintos. A diferencia del paralelismo a nivel de threads que se da cuando un multiprocesador ejecuta threads en paralelo en distintos cores, no replica todo el procesador sino únicamente el estado de cada thread, incluyendo el program counter y los registros utilizados[14]. Esto permite reducir la penalización en términos de latencia en que se incurre al realizar accesos a memoria que resultan en un *cache miss*. Por otro lado, el aumento en la velocidad en la que pueden despacharse instrucciones en procesadores superescalares requiere que se explote una mayor cantidad de paralelismo a nivel de instrucciones, el cual es limitado[18]. En estos casos, multithreading permite aumentar la utilización de las unidades funcionales mediante la ejecución de instrucciones provenientes de otro thread.

Existen distintas variantes de multithreading que dependen de si el procesador

puede ejecutar instrucciones de un thread o de varios threads en cada ciclo. En la primera variante, denominada *fine-grained multithreading*, el procesador alterna la ejecución de instrucciones de threads distintos en cada ciclo de manera *round robin*. La idea detrás de esta técnica es lograr que en un momento dado no haya dos o más instrucciones de un mismo thread en el pipeline simultáneamente, lo cual permite eliminar los *stalls* que se producen al ejecutar instrucciones que dependen de instrucciones previas[16]. Sin embargo, esto requiere disponer de una cantidad elevada de threads, lo cual no siempre sucede, dando lugar a la variante *coarse-grained multithreading*, en la que se ejecutan instrucciones de un mismo thread hasta que se ocasione un stall, por lo que se requieren menos threads para mantener al procesador ocupado. Finalmente, *simultaneous multithreading* extiende la variante anterior a procesadores superescalares, permitiendo que un thread ejecute más de una instrucción en cada ciclo siempre que esto sea posible, y alternando threads en caso de *stalls*.

Algunas arquitecutras, como la de las GPUs y procesadores vectoriales, explotan el paralelismo a nivel de datos mediante instrucciones que se aplican sobre un conjunto de datos en paralelo[14].

2.2 Diseño de arquitecturas de hardware

A continuación se describen los principales paradigmas de diseño de arquitectura de hardware. Para esto se utilizará la taxonomía de arquitecturas de computadoras de Flynn[19], que se basa en el concepto de *stream*. Un stream define una secuencia de items (que pueden ser tanto instrucciones como datos) manipulada por un procesador. Dentro de cada tipo de stream, se distinguen dos posibilidades, dando lugar a cuatro combinaciones diferentes:

Single Instruction Single Data stream (SISD)

Esta categoría comprende computadoras secuenciales que no explotan paralelismo ni en los datos ni en las instrucciones. Una arquitectura representativa consiste en una única unidad de control que procesa el stream de instrucciones secuencialmente y envía señales de control a una unidad funcional que ejecuta operaciones sobre un único stream de datos, lo cual corresponde con la clásica arquitectura von Neumann. Sin embargo, típicamente se admiten ciertas formas de paralelismo dentro de esta categoría tales como *pipelining* y procesadores superescalares (sin

multithreading)[20]. En ambos casos, las técnicas son invisibles al programador, con la excepción de mejoras en el desempeño.

Single Instruction Multiple Data stream (SIMD)

Al igual que SISD, en este modelo de ejecución se tiene una única unidad de control y, desde el punto de vista del programador, una única instrucción se está ejecutando en un momento dado. La diferencia es que SIMD explota paralelismo a nivel de datos mediante instrucciones que operan sobre múltiples elementos simultáneamente. En la práctica, esto se traduce a disponer de múltiples unidades funcionales que implementan las instrucciones vectoriales. Un ejemplo de esta arquitectura son los procesadores vectoriales[20]. También pueden considerarse SIMD procesadores *single-core* que admiten operaciones vectoriales, como es el caso de los primeros procesadores que dieron soporte al conjunto de instrucciones MMX[21][17].

Una clasificación relacionada, que no forma parte de la jerarquía de Flynn original, es *Single Instruction, Multiple Threads (SIMT)*. Esta clasificación se utiliza especialmente en el ámbito de las GPUs para describir arquitecturas compuestas por procesadores SIMD que emulan múltiples threads[22]. Desde el punto de vista del programador, el comportamiento SIMD no es visible, ya que en vez de utilizar instrucciones vectoriales, se mapean los elementos de un vector de largo N a N threads que individualmente procesan un único dato.

Multiple Instruction Single Data stream (MISD)

Esta categoría corresponde con arquitecturas en las que múltiples instrucciones operan sobre los mismos datos. No hay un consenso sobre qué sistemas pueden clasificarse como MISD. Algunos autores[20] opinan que los *systolic arrays* y las GPUs pueden considerarse, al menos parcialmente, como MISD, mientras que otros autores[16] opinan que no hay sistemas comerciales que puedan clasificarse de esta forma.

Multiple Instruction Multiple Data stream (MIMD)

Esta clase de sistemas explota el paralelismo tanto en las instrucciones como en los datos. La mayor parte de los sistemas en esta categoría son sistemas multiprocesadores, en los que distintos procesadores cuentan con su propia unidad de control y ejecutan de manera independiente. Cada procesador individual puede ser un procesador escalar, vectorial o un multiprocesador[23]. Otra clase de procesadores MIMD

son los que tienen soporte para multithreading[20]. Estas arquitecturas extienden un procesador con múltiples conjuntos de registros de datos y de instrucciones, que son utilizados por distintos threads de manera independiente para obtener un mejor aprovechamiento de las unidades funcionales disponibles. Una configuración usual consiste en combinar multithreading y múltiples cores en un mismo sistema.

Si bien la clasificación original de Flynn no distingue entre distintos tipos de MIMD, algunos autores subdividen MIMD en *Single Program, Multiple Data (SPMD)*[24] y *Multiple Programs, Multiple Data (MPMD)*[25]. Ambos modelos parten de la base de que pueden ejecutarse distintos streams de instrucciones simultáneamente, pero en el primero las instrucciones provienen del mismo programa, mientras que en el segundo provienen de programas distintos. SPMD también puede verse como una alternativa menos restrictiva de SIMD.

2.2.1 Latencia vs throughput

Las métricas principales para medir el desempeño de un procesador son la latencia y el throughput. La latencia es el tiempo que transcurre entre que se inicia la ejecución de una tarea y finaliza la misma, mientras que el throughput mide la cantidad de tareas finalizadas por unidad de tiempo. En el contexto de diseño de chips, suele presentarse un compromiso entre minimizar la latencia del procesador y aumentar el throughput del mismo al decidir cómo utilizar el área y la energía disponible, lo cual hace que mejorar uno de estos aspectos pueda resultar en un deterioro sobre el otro[2]. Como consecuencia, surgen dos estilos de microarquitecturas: las orientadas a la latencia y las orientadas al throughput.

Tradicionalmente, las arquitecturas de CPU fueron diseñadas para minimizar la latencia asociada a la ejecución de un único thread. Para lograr esto, se utilizan caches *on-chip* grandes que permiten reducir significativamente el costo de los accesos a datos. De la misma forma, se utilizan técnicas como la ejecución fuera de orden y la ejecución especulativa para reducir las latencias asociadas a las unidades aritméticas. Si bien estas decisiones representan una mejora en la latencia, requieren del uso de gran parte del área del chip y la energía que podrían haberse utilizado para agregar unidades aritméticas adicionales y canales de acceso a memoria[26]. La decisión de compromiso tomada en el diseño de la CPU se basa en la suposición de que el trabajo a realizar no es muy paralelizable. Si bien los procesadores de un único core como los Intel Pentium IV solían estar fuertemente orientados a la latencia, las arquitecturas de las CPUs multicore más recientes reflejan una tendencia a diseños

que esperan una mayor medida de paralelismo[2].

Los procesadores orientados al throughput, en el otro lado del espectro, asumen que las tareas que deben realizar tienen asociado un alto grado de paralelismo. El throughput se vincula con la latencia y la concurrencia de instrucciones ejecutadas a través de la ley de Little[27], de la siguiente manera:

$$throughput = \frac{instrucciones\ concurrentes}{latencia}$$

Dado que aumentar la cantidad de instrucciones que están siendo ejecutadas en un momento dado se traduce en un mayor throughput, esta clase de procesadores dedican un gran área del chip para unidades de procesamiento, hacen un gran uso del multithreading y adoptan el modelo de ejecución SIMD[26].

2.3 Procesadores gráficos

Hasta la década de 1970, se utilizaban monitores vectoriales, en donde las imágenes eran construidas y desplegadas como líneas, en vez de una grilla de píxeles. Luego, con la llegada de los gráficos en mapas de bits, se comenzaron a utilizar *framebuffers* para almacenar la información de una imagen. En estos sistemas, cada punto de una línea es almacenado en memoria, a diferencia de los monitores vectoriales, en donde una línea era definida a partir de sus extremos. Uno de los primeros sistemas con framebuffer fue *SuperPaint*, el cual fue precursor de muchas técnicas populares como el uso de tablas de color[28]. Estos avances fueron posibles gracias a invención de chips de DRAM de alta densidad, los cuales significaron una capacidad de almacenamiento muy superior a sus antecesores.

Durante el resto de la década de 1970, y principios de 1980s, el contenido del framebuffer era generado por la CPU. Esto podía representar un impedimento para generar gráficos en tiempo real, ya que la CPU también debía encargarse de procesar la entrada del usuario, el sonido y otras tareas. Esta realidad llevó a que los fabricantes diseñaran un tipo especial de hardware, conocido como acelerador de gráficos, para disminuir la cantidad de tiempo que invierte la CPU en rellenar el framebuffer. Bajo este paradigma, la CPU envía comandos al acelerador de gráficos que corresponden con ciertas primitivas a dibujar. Posteriormente, el acelerador ejecuta estos comandos y almacena los resultados en el framebuffer. En este contexto, en 1984 IBM desarrolló PGC (*Professional Graphics Controller*), uno de los primeros aceleradores gráficos para PC, orientado al mercado del diseño asistido por computadora

(CAD). Consistía en tres PCBs interconectados, que contenían un microprocesador 8088 y 320 kB de RAM. La llegada de la PGC fue un suceso importante en la evolución de los procesadores de gráficos, ya que fue la primera vez en que se puso en práctica la idea de disponer de un coprocesador encargado del procesamiento de gráficos que liberara a la CPU para realizar otra clase de cómputo[29].

En la década de 1990, se desarrollaron APIs (*Application Programming Interface*), como OpenGL y DirectX, que otorgaron a desarrolladores una manera uniforme de manipular dispositivos de hardware gráfico, lo cual permitió que no tuviesen que escribir interfaces y drivers personalizados para cada dispositivo, evitando de ésta forma el trabajo duplicado. Estos esfuerzos también permitieron unificar las distintas ideas sobre las etapas que constituyen el *pipeline gráfico*, un modelo conceptual que describe las etapas que implica la *renderización* de una imagen.

En la década de 1990, el hardware gráfico estaba dominado por el uso de *pipelines* fijos. En los pipeline fijos, las funciones que conformaban las APIs gráficas estaban mapeadas a circuitos lógicos diseñados para implementar dichas funciones, lo cual significaba que muchas veces no era posible introducir cambios sin reemplazar el hardware que las implementaba[26]. Esta arquitectura estaba caracterizada por ser poco flexible para el desarrollador. Por un lado, éste no podía modificar los datos una vez que eran introducidos al pipeline. Por otro lado, la creatividad en la creación de efectos gráficos sofisticados estaba limitada por las características ofrecidas por el estándar utilizado. Si bien a lo largo de este período se incorporaron recursos de hardware nuevos y aspectos de configuración sobre las distintas etapas del pipeline, existía una demanda de funcionalidades nuevas que era difícil de satisfacer a través de pipeline fijos. La solución adoptada para mitigar este problema fue tranformar alguna de estas etapas del pipeline gráfico en procesadores programables. Junto a estas mejoras, las APIs principales de gráficos empezaron a dar soporte de *shaders*, término utilizado para designar a los programas que pasaron a controlar etapas del pipeline gráfico.

La parte del pipeline gráfico que se ejecuta en el procesador de gráficos (o GPU por su sigla en inglés) puede dividirse conceptualmente en dos grandes etapas: geometría y rasterización (una explicación detallada de cada etapa puede encontrarse en[30]). En las GPUs pioneras con soporte para shaders, la primera etapa era implementada mediante procesadores de vértices que ejecutaban shaders de vértices, mientras que la segunda era implementada por procesadores de píxeles que ejecutaban shaders de píxeles. El primer precedente de este tipo de diseño se dio con el modelo GeForce 3 de NVIDIA, que agregó soporte para shaders de píxeles y vérti-

ces. Esto coincidió con el lanzamiento de Microsoft DirectX 8 y las extensiones de shaders de OpenGL. En 2002, la ATI Radeon 9700 incorporó el soporte para shaders de pixeles, a través de DirectX 9 y OpenGL. Este modelo fue el primer acelerador de DirectX 9 disponible y permitió mayor flexibilidad al programar[31].

Impulsados por la industria de los videojuegos, los fabricantes de GPUs continuaron realizando mejoras en el diseño de sus chips para aumentar la cantidad de poligonos renderizados por unidad de tiempo. Según el algoritmo de renderizado utilizado, formar una imagen implica iterar sobre todas las primitivas geométricas que componen la escena o sobre todos los píxeles de la imagen[32]. Este tipo de problema (conocido en inglés como *embarrassingly parallel*) se caracteriza por ser altamente paralelizable ya que se puede resolver en simultáneo por múltiples procesadores con pocas o ninguna dependencia entre ellos. La capacidad de cómputo de las GPUs, que excedía en órdenes de magnitud la de las CPUs para problemas altamente paralelizables, hizo que a comienzos de la década del 2000 desarrolladores de software científico se interesaran por utilizar el hardware para resolver problemas no relacionados a gráficos[33]. Sin embargo, para acceder a los recursos de la GPU, el programador debía mapear su problema a un problema de gráficos, lo cuál podría llegar a ser una tarea extremadamente difícil.

Dado que las GPUs típicamente procesan más píxeles que vértices, las arquitecturas antiguas solían incluir una mayor cantidad de procesadores de píxeles. El problema principal de esta clase de diseño era que para algunos flujos de trabajo, un tipo de shader era más utilizado que el otro, lo cual significaba un mal aprovechamiento de los recursos. Esto dio lugar al concepto de shader unificado, en donde un mismo procesador ejecuta todas las operaciones de shader[34]. El primer precedente de procesador unificado ocurrió en 2005 con la llegada de la consola de videojuegos XBox 360. Al año siguiente, NVIDIA incorpora este diseño en su nueva arquitectura Tesla, con el lanzamiento de la GeForce 8800. Poco tiempo después, se lanzó CUDA (*Compute Unified Device Architecture*), una plataforma de desarrollo y una API que permite que los desarrolladores utilicen los nuevos shaders unificados (denominados *CUDA cores*) para el procesamiento de propósito general. Los CUDA cores incorporaron memoria de instrucciones, caches y lógica de control. Para balancear el costo adicional asociado a estos recursos de hardware, se adoptó la estrategia de hacer que múltiples cores compartan un mismo cachéde instrucciones y lógica de control. Estos conjuntos de cores, junto al hardware adicional compartido, son denominados *Streaming Multiprocessor* (SM) en la terminología de NVIDIA. Los SMs utilizan el modelo de ejecución *SIMT* (*Single instruction, multiple threads*). SIMT es similar

a *SIMD* (*Single Instruction, Multiple Data*) en la taxonomía de Flynn en cuanto a que una única instrucción manipula datos distintos, pero no expone el ancho SIMD al programador. Para lograr esto, se introduce el modelo de threads, permitiendo que el desarrollador pueda especificar flujos de ejecución y direcciones de memoria relativas a threads particulares[6].

2.3.1 Hardware

Considerando la evolución de las arquitecturas, NVIDIA estructura el lanzamiento de las GPUs en generaciones. Las generaciones nuevas introducen grandes mejoras en la funcionalidad o la arquitectura del chip, mientras que los modelos dentro de una misma generación incorporan cambios pequeños que sólo afectan moderadamente la funcionalidad o el rendimiento. El conjunto de características que brinda un modelo de GPU está asociado a su Compute Capability[6], una especificación que describe el hardware y las instrucciones de una familia de arquitecturas de GPUs de NVIDIA. En el contexto de la jerarquía de Flynn, en ocasiones son clasificadas como MIMD[26], SPMD[35] o hasta MISD[20]. Dado que en el modo de ejecución más representativo de la GPU conceptualmente puede pensarse que las unidades funcionales ejecutan la misma instrucción en modo *lock-step*, NVIDIA describe a la arquitectura de las GPUs como SIMT. Si bien el conjunto de instrucciones es SIMD, el ancho de las instrucciones no se expone al programador[6].

A nivel de software, el cómputo en CUDA está estructurado jerárquicamente: la unidad más pequeña de paralelismo es el CUDA Thread, grupos de threads forman bloques y finalmente grupos de bloques forman un grid. Para procesar cada uno de estos elementos en paralelo, la GPU utiliza distintos niveles de planificación a nivel de hardware[36]. En un primer nivel, los bloques que componen el grid son distribuídos sobre distintos cores, denominados Streaming Multiprocessors (SMs), los cuales procesan bloques de manera independiente. La unidad encargada de la planificación central se denomina *GigaThread engine*[36]. Dentro de un mismo SM, los bloques son vistos como grupos de *warps*, que consisten en threads de instrucciones SIMD (conocidas como instrucciones *PTX* en el contexto de las arquitecturas de NVIDIA). El siguiente nivel de planificación es llevado a cabo mediante una unidad denominada *warp scheduler*, la cual se encarga de asignar la ejecución de cada warp a una de las particiones del SM. Las particiones, o bloques de procesamiento, se encargan de ejecutar warps de manera independiente. Para lograr esto, la ejecución de instrucciones PTX asociadas a un warp se realiza a través de múltiples *lanes*, o

CUDA cores, que actúan sobre elementos de datos individuales[37]. En la Figura 2.1 se presenta la organización global de un SM.

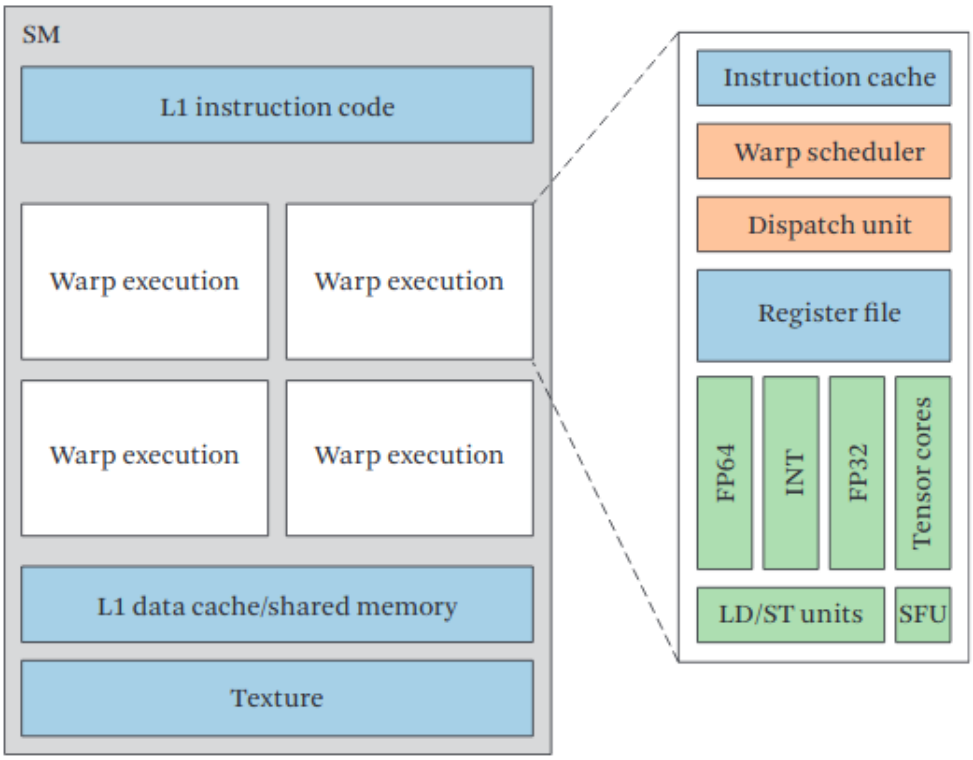


Figura 2.1: Visión simplificada del Streaming Multiprocessor de la GPU NVIDIA V100. Los bloques verdes representan múltiples unidades funcionales del mismo tipo. Extraído de *Heterogeneous Computing: Hardware and Software Perspectives*[38]

La GPU implementa varias técnicas de paralelismo mencionadas en la Sección 2.1. Si se ve a los warps como threads de instrucciones SIMD, los SMs implementan multithreading al permitir que los warp schedulers decidan qué warp ejecutar en cada ciclo de reloj. A diferencia de las CPUs, que ocultan la latencia mediante una jerarquía de caches grande, la principal estrategia que utiliza la GPU para ocultar la latencia es el multithreading[35]. Utilizar esta técnica requiere, como mínimo, que cada warp tenga su propio *program counter* asociado. Las microarquitecturas anteriores a Volta[5], utilizaban un único program counter para mantener el estado de cada warp, junto a un registro *active mask* que indicaba los threads activos en un momento dado. Esto significaba que ante la presencia de código divergente, grupos de threads divergentes ejecutaban secuencialmente hasta converger. Volta introdujo la planificación de threads independiente, una estrategia en la que se dispone de un program counter distinto para cada thread. Si bien el modelo de ejecución sigue siendo SIMT para los distintos subgrupos de threads divergentes, es-

ta técnica permite que threads puedan diverger y sincronizarse a una granularidad más pequeña que la del warp.

Otra de las técnicas de paralelismo implentada en la GPU, a nivel de CUDA core, es el pipelining, lo cual permite que cada CUDA core sea capaz de ejecutar una instrucción distinta en cada ciclo de reloj[36]. Sin embargo, estrategias como *branch prediction* no son incorporadas, lo cual se debe a que este tipo de diseño suele tenerse en cuenta en architecturas orientadas a latencia, mientras que la filosofía de diseño de la GPU es orientada al throughput[35].

La memoria principal de la GPU consiste en una memoria DRAM *off-chip*, la cual NVIDIA denomina memoria global. Cada CUDA Core tiene asignado una sección privada de la memoria global denominada *memoria local*. Todo el contenido del *stack* que no puede ser alojado en registros se almacena en memoria local, así como variables privadas que exceden el tamaño de los mismos[14]. Si bien la principal técnica para ocultar la latencia es el multithreading, las GPUs modernas también disponen de caches tradicionales para reducir el costo asociado a los accesos a memoria global. El comportamiento de los niveles de cachédepende de la Compute Capability del dispositivo[6]. Adicionalmente, cada bloque tiene acceso a cierta cantidad de memoria *scratchpad*, conocida como memoria compartida. A diferencia de las memorias cache, que utilizan un controlador de hardware complejo para decidir qué datos deben cargarse, las memorias scratchpad deben ser controladas por software y se utilizan para almacenar resultados intermedios o compartir datos entre hilos de un mismo bloque[37].

Tanto en memoria global como en memoria compartida se penalizan ciertos patrones de acceso. En el primer caso, los accesos a memoria global realizados por un mismo warp se subdividen en transacciones de 32, 64 o 128 bytes[6]. Mientras más distantes sean las regiones de memoria a la que acceden los threads de un warp, mayor es la cantidad de transacciones necesarias para satisfacer la petición, lo cual provoca un menor throughput de instrucciones. Por esta razón, es conveniente que a nivel de warp se acceda a regiones de memoria contiguas siempre que sea posible, lo cual se conoce como un acceso *coalesced*. El Código 2.1 presenta un ejemplo de acceso coalesced a memoria global. Por otro lado, la memoria compartida está dividida en bancos, que son módulos de memoria del mismo tamaño que pueden ser accedidos en paralelo. Si dentro de un warp se accede a dos direcciones pertenecientes al mismo banco, los accesos deben ser serializados[1], lo cual se conoce como un

[1]Existen algunas excepciones. Una de ellas ocurre cuando todo el warp lee la misma dirección, en cuyo caso se realiza un broadcast.

conflicto de bancos[6].

Las GPUs pueden estar presentes en una tarjeta de video o integrada en la placa madre. Típicamente, la comunicación entre la CPU y la GPU se realiza mediante un bus PCI Express. Sin embargo, algunas tarjetas modernas de alta gama de NVIDIA utilizan *NVLink*[39], un protocolo creado con el propósito de acelerar significativamente la comunicación basado en tecnología propietaria.

2.3.2 Modelo de programación

En un comienzo, utilizar la GPU para resolver problemas de propósito general implicaba interpretar ese problema como una realidad de gráficos, dado que las bibliotecas disponibles habían sido diseñadas con ese objetivo en mente. Para incrementar la productividad de los programadores, NVIDIA decidió crear una extensión de *C* a la cual le llamó CUDA[6], del inglés *Compute Unified Device Architecture*. Un programa escrito en CUDA puede estar compuesto tanto por código que ejecuta en la CPU (*host* en la terminología de NVIDIA) como en la GPU (*device*).

La unidad atómica de paralelismo en la GPU es el *CUDA thread*. Los threads ejecutan código definido mediante funciones especiales denominadas *kernels*. Para facilitar la asignación de cómputo y de datos a distintos recursos de hardware, la abstracción de *thread* se combina jerárquicamente con la de *bloque*, que corresponde a un grupo de threads, y *grid*, compuesto por un grupo de bloques. El tamaño de los bloques y la cantidad de bloques son parámetros configurables al momento de ejecutar el kernel[6].

Dependiendo del nivel en la jerarquía de threads, se puede acceder a distintos espacios de memoria, ilustrados en la Figura 2.2. La comunicación entre threads pertenecientes a un mismo warp puede realizarse mediante primitivas de comunicación entre warps desde Compute Capability 3.0[40]. Adicionalmente, la arquitectura Volta introdujo la posibilidad de sincronizar threads pertenecientes a un mismo warp[5]. Dentro de un mismo bloque, la comunicación y sincronización de threads se realiza mediante memoria compartida y la primitiva *__syncthreads()*, respectivamente. La comunicación entre bloques se realiza mediante memoria global, mientras que la sincronización admite dos maneras. La primera utiliza implícitamente el hecho de que las ejecuciones de kernels dentro de un mismo *stream* se realizan de manera secuencial[41], por lo que la sincronización se logra ejecutando kernels consecutivos. La segunda manera consiste en utilizar grupos cooperativos, una característica ofrecida a partir de CUDA 9.0 que permite la sincronización a nivel de grid[6].

La sintaxis que permite señalar que una función es un kernel consiste en anteceder la firma de función con el especificador *__global__*. Para identificar un thread dentro de la jerarquía de threads, se proveen distintas variables que almacenan vectores de tres componentes. Dentro del cuerpo del kernel, cada thread puede identificarse dentro de un bloque a través de la variable *threadIdx*. En el caso que se lance un único bloque, la misma es suficiente para identificar globalmente los threads. En el caso general, también se requiere del identificador de bloque, *blockIdx*, y la cantidad de elementos por bloque, *blockDim*. Al momento de realizar la llamada al kernel, se configuran parámetros de ejecución como la cantidad de bloques y de threads por bloque que se encargarán de ejecutarlo. El Código 2.1 presenta una implementación *naive* de multiplicación de matrices densas haciendo uso del modelo de programación CUDA. El código de la función *main* es ejecutado en la CPU y se encarga de transferir las matrices a la memoria global de la GPU y establecer los parámetros necesarios para la ejecución del kernel. Luego, cada thread calcula un elemento distinto de la matriz resultante en forma paralela manteniendo resultados parciales en una variable local. Finalmente, el resultado es almacenado en memoria global.

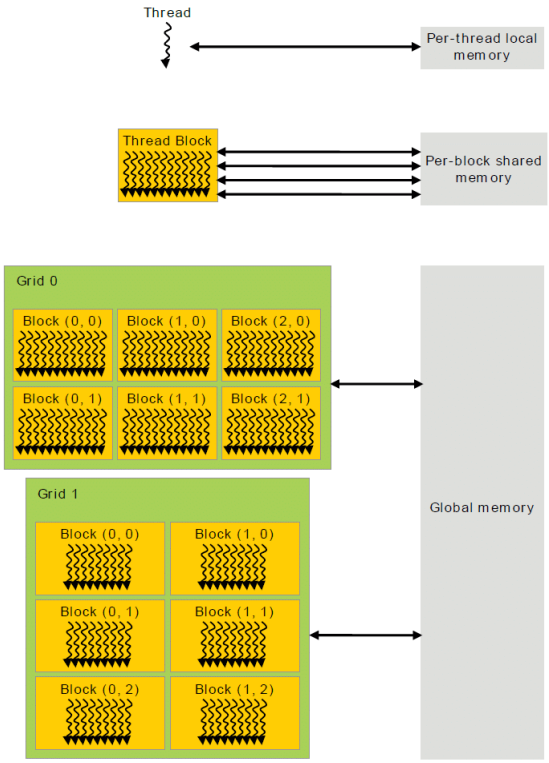


Figura 2.2: Jerarquías de threads y memoria. Extraído de *CUDA Programming Guide*[6]

```
1  __global__ void multiply(float* A, float* B, float* C, int dim) {
2      // El thread que ejecuta el kernel calcula la posición (i, j) de C
3      int i = blockIdx.y * blockDim.y + threadIdx.y;
4      int j = blockIdx.x * blockDim.x + threadIdx.x;
5
6      if ((i < dim) && (j < dim)) {
7          // Los resultados parciales son acumulados en la variable result
8          float result = 0;
9          for (int k = 0; k < dim; k++) {
10             result += A[i * dim + k] * B[k * dim + j];
11         }
12         C[i * dim + j] = result;
13     }
14 }
15
16 int main()
17 {
18     // Transferencia de las matrices de entrada a la memoria global de la GPU
19     ...
20     // Invocación del kernel
21     dim3 threadsPerBlock(16, 16);
22     dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
23     multiply<<<numBlocks, threadsPerBlock>>>(A, B, C, N);
24     ...
25 }
```

Algorítmo 2.1: Multiplicación de las matrices cuadradas A y B, de dimensión N.

2.3.3 Tensor Cores

En el año 2017, NVIDIA introduce su nueva microarquitectura Volta como sucesora de Pascal[5]. Uno de los cambios principales con respecto a microarquitecturas anteriores es la incorporación de una nueva unidad funcional denominada *Tensor Core*. Los Tensor Cores son unidades de punto flotante de precisión mixta optimizados para realizar la operación $D = A \times B + C$, donde A, B, C y D son matrices densas de un tamaño determinado por la arquitectura. Las matrices A y B almacenan números en punto flotante de media precisión (16 bits), mientras que en la matriz de aculumación, C, pueden utilizarse números en punto flotante de precisión

simple o media. Lo mismo aplica para la matriz D. La motivación principal detrás de la creación de esta nueva unidad funcional fue acelerar el entrenamiento de redes neuronales de gran tamaño, dado que la multiplicación de matrices de datos de entrada y pesos en las capas conectadas de la red es una de las principales operaciones en este contexto[5]. En Volta, cada Tensor Core es capaz de realizar 64 operaciones FMA (*fused multiply–accumulate*) de precisión mixta por ciclo de reloj.

WMMA API

Los Tensor Cores son expuestos al programador mediante el API WMMA, incluída a partir de CUDA 9. Toda la manipulación de matrices a través del API se realiza sobre la clase *fragment*, creada con el propósito de almacenar un fragmento de alguna de las matrices que participarán en la operación de multiplicación y acumulación, descrita anteriormente. Los datos contenidos dentro de un mismo fragmento son distribuidos en todo el warp de manera no especificada. Es decir, el fragmento es compartido por todo el warp, pero la asociación entre elementos del fragmento y la memoria privada de cada thread es desconocida.

Las operaciones que pueden realizarse a través del API son *load_matrix_sync*, *store_matrix_sync*, *fill_fragment* y *mma_sync*. Las primeras dos se utilizan para cargar y traer de memoria un fragmento, respectivamente. *fill_fragment* se utiliza para cargar un fragmento con un valor constante. *mma_sync* realiza la multiplicación entre fragmentos.

Si bien NVIDIA implementa la multiplicación de matrices de 4×4 en hardware[1], CUDA sólo permite operar sobre matrices de mayor dimensión, por lo que la división de las matrices de entrada en bloques de menor tamaño se realiza de manera transparente al programador. Existen múltiples combinaciones válidas para las dimensiones de las matrices de entrada, las cuales no necesariamente deben ser cuadradas. Sin embargo, en todas se cumple que la matriz resultante es de 16×16 elementos[6].

2.4 Álgebra lineal numérica dispersa

Existe una gran cantidad de problemas que pueden ser expresados en términos de operaciones sobre matrices y vectores. Por este motivo se desarrollaron implementa-

[1] Válido para la arquitectura Volta[42], pero no necesariamente para arquitecturas posteriores[43].

ciones altamente optimizadas de los algoritmos de álgebra lineal, dentro de las cuales se encuentran BLAS y LAPACK. Una subclase importante de estos problemas está compuesta por los que se representan mediante matrices con valores mayoritariamente nulos, conocidas como matrices dispersas. En esos casos, es conveniente que los métodos numéricos utilizados para encontrar soluciones tengan en consideración la estructura dispersa del sistema, dando lugar al álgebra lineal numŕica dispersa. Uno de los orígenes más comunes de los sistemas dispersos ocurre en la resolución de ecuaciones diferenciales en derivadas parciales (EDP), las cuales tienen un rol importante en muchas aplicaciones científicas e ingenieriles, como la dinámica de fluídos, la mecánica cuántica y el análisis estructural. Las matrices dispersas suelen aparecer en uno de los métodos númericos para resolver EDPs más populares, conocido como método de los elementos finitos[44]. Otros ejemplos de problemas de la disciplina son resolver sistemas lineales dispersos en el campo de la investigación de operaciones y la transformación de representaciones de un sistema en el procesamiento de imágenes[45].

2.4.1 Formatos

Los formatos de almacenamiento dispersos son estrategias que evitan almacenar información redundante de las matrices, lo cual en la práctica se traduce a almacenar los valores no nulos de una matriz en regiones de memoria contiguas, junto a una combinación de estructuras de datos adicionales que permiten determinar las coordenadas asociadas a cada elemento. Una motivación para utilizar formatos dispersos es que representar matrices con un número elevado de coeficientes nulos[1] mediante arreglos densos de dos dimensiones haría que algunos problemas no puedan representarse por los enormes requerimientos de almacenamiento. Por otro lado, el desempeño de operaciones sobre matrices dispersas como SpMV o SpMM está limitado por los accesos de memoria[46][47]. El primer factor que explica este fenómeno es que en estas operaciones, la cantidad de operaciones aritméticas es baja en relación a la cantidad de accesos a memoria (concepto conocido como intensidad computacional). El segundo es lo que se conoce como *memory wall*[48], que hace referencia al hecho de que la evolución del desempeño de la memoria, tanto en latencia como en bandwidth, no ha evolucionado a la par con el desempeño computacional de los procesadores.

[1] El concepto de nulo puede cambiar en distintos campos de aplicación, el más común es coeficientes con valor 0

$$A = \begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 0 \\ 3 & 1 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

Figura 2.3: Ejemplo de matriz densa

El formato más simple para almacenar matrices dispersas es el formato de coordenadas (conocido como COO)[49]. En este formato, se almacenan únicamente los valores no nulos de la matriz y las coordenadas son almacenadas en forma explícita. Típicamente, se utiliza un arreglo *val* para almacenar los valores, otro para almacenar las filas *row_ind* y otro para las columnas *col_ind*. Es deseable que las tuplas estén ordenadas por fila y columna, pero no es una condición necesaria. La matriz A, presentada en la Figura 2.3, puede representarse en formato COO de la siguiente forma:

$$val = \begin{bmatrix} 1 & 4 & 1 & 3 & 1 & 1 & 2 \end{bmatrix}$$

$$row\_ind = \begin{bmatrix} 0 & 0 & 1 & 2 & 2 & 2 & 3 \end{bmatrix}$$

$$col\_ind = \begin{bmatrix} 0 & 3 & 1 & 0 & 1 & 2 & 3 \end{bmatrix}$$

Si los elementos de COO se ordenan por fila, tal como sucede en el ejemplo anterior, entonces puede afirmarse que el arreglo *row_ind* almacena información redundante, ya que bastaría con almacenar el comienzo de cada fila. Una alternativa similar a COO que atiende este problema es el formato comprimido por fila (CSR)[49]. Ambas representaciones comparten el arreglo de valores y de columnas, pero esta última comprime el arreglo de filas. Es decir, se utilizan tres arreglos: *val*, *col_ind* y *row_ptr*, en donde el elemento i de *row_ptr* indica la posición en la que comienza la i-ésima fila en el resto de los arreglos. Para evitar que las implementaciones basadas en este formato tengan que realizar distintos flujos de ejecución según si se trata de la última fila u otra fila cualquiera, se sigue la convención de definir $row\_ptr(n+1) = nnz+1$, en donde nnz es la cantidad de elementos no nulos de la matriz. Una ventaja de CSR es que no almacena elementos innecesarios. Por otro lado, requiere que se realicen operaciones adicionales cada vez que se quiere acceder a un elemento. También, como principal ventaja, permite acceder a todos los elementos de una fila de forma muy sencilla. La representación de la matriz A en formato CSR se presenta a continuación:

$$val = \begin{bmatrix} 1 & 4 & 1 & 3 & 1 & 1 & 2 \end{bmatrix}$$

$$col\_ind = \begin{bmatrix} 0 & 3 & 1 & 0 & 1 & 2 & 3 \end{bmatrix}$$

$$row\_ptr = \begin{bmatrix} 0 & 2 & 3 & 6 & 7 \end{bmatrix}$$

El espacio necesario para almacenar una matriz depende del tamaño de los arreglos que forman parte de la representación. Para una matriz en formato CSR de nnz elementos no nulos y dimensión dim, los arreglos de valores y columnas son de tamaño nnz, mientras que el de filas es de tamaño $dim + 1$. Asumiendo que se utilizan 4 bytes para representar tanto los elementos de cada arreglo como los índices, el tamaño total es de $4(2nnz + dim + 1)$ bytes. Esto representa una diferencia de $4(nnz - dim - 1)$ bytes con respecto a COO.

Se han propuesto una variedad de formatos similares a CSR, dentro de los cuales se encuentra el formato comprimido por columnas (CSC), que es análogo a CSR pero comprime el arreglo de columnas. Una de las alternativas es *Compressed Diagonal Storage* (CDS)[50], que aprovecha la estructura de matrices banda para representar la matriz eliminando los arreglos de filas y columnas. Otra variante posible consiste en dividir la matriz original en bloques densos de igual tamaño y hacer que el arreglo de columnas represente posiciones de bloques. En este esquema, denominado *Block Compressed Row Format* (BSR)[50], el arreglo comprimido de filas es análogo al arreglo correspondiente de CSR, pero considera filas de bloques y no escalares. Este formato requiere que los elementos de un mismo bloque sean contiguos en el arreglo de valores y utiliza *padding* en dicho arreglo para completar los bloques con ceros, ya que asume que son del mismo tamaño. Otros formatos utilizan bloques de tamaño variable. Entre ellos, dos de los más destacados son 1D-VBL (*Varibable Block Length*)[51], que utiliza bloques unidimensionales, y VBR (*Variable Block Row*)[52], en el que los bloques son de dos dimensiones.

Existen múltiples criterios posibles con los cuales pueden evaluarse formatos dispersos. Langr y Tvrdík[53] hacen una revisión de la literatura disponible sobre el tema y observan que suelen utilizarse criterios de evaluación que dependen de una variedad de parámetros, incluyendo dependencias de arquitectura, calidad de implementación y representación utilizada para los valores de la matriz. Por último, sugieren criterios adicionales que no presentan estas dificultades.

2.5 Multiplicación de matrices dispersas (SpMM)

La multiplicación general de matrices (GEMM según la notación de BLAS) tiene la forma:

$$D = A \times B + C, \tag{2.1}$$

donde A, B y C son matrices de entrada y D es la salida. La multiplicación de matrices dispersas (SpMM) es una variante de GEMM en la que las matrices involucradas están almacenadas en un formato disperso. Es una operación de gran utilidad en diversos contextos del álgebra lineal y el análisis de grafos, con aplicaciones como solvers para métodos multigrid algebraicos (AMG)[8], conteo de triángulos[9] y búsqueda en anchura (BFS) con múltiples orígenes[10]. La diferencia principal con la variante densa de la operación es que los algoritmos de SpMM buscan explotar el hecho de que la mayor parte de los elementos de las matrices de entrada son nulos, ya que no hacerlo implica malgastar recursos de cómputo en calcular elementos nulos de la matriz resultante. Algoritmo 1 presenta un pseudocódigo del algoritmo fila por fila de SpMM presentado por Gustavson[54]. En el caso de otras operaciones típicas sobre estructuras dispersas, como la multiplicación de matriz dispersa por vector (SpMV), una estrategia común para mejorar el desempeño es explotar conocimiento previo sobre el patrón de dispersión de la matriz dispersa para minimizar operaciones de memoria sobre memoria global[55]. Sin embargo, SpMM agrega dificultades adicionales, ya que la complejidad computacional del problema no depende únicamente de la estructura de dispersión de las entradas por separado, sino de cómo interactúan entre sí. Por otro lado, en la mayoría de las aplicaciones, las técnicas de optimización que se basan en analizar patrones de dispersión son menos efectivas para SpMM ya que, a diferencia de SpMV, que es una operación que suele formar parte del loop más interno de solvers iterativos, SpMM suele ejecutarse una única vez para un par de matrices.

2.5.1 Diseño de SpMM

Las principales decisiones de diseño que deben considerarse a la hora de abordar esta operación son en relación a cómo determinar la cantidad de memoria que debe reservarse para la matriz resultado, cómo acumular los resultados parciales y cómo particionar el trabajo a realizar entre distintas unidades de procesamiento (en el caso paralelo). Si bien algunos enfoques y decisiones aplican para distintas arquitecturas, por ejemplo multi-core, en esta sección se hace énfasis en trabajos previos realizados

Algoritmo 1 SpMM propuesto por Gustavson

```
1: for a_{i*} ∈ A do
2:    for a_{ij} ∈ a_{i*} and a_{ij} ≠ 0 do
3:       for b_{jk} ∈ b_{j*} and b_{jk} ≠ 0 do
4:          value = a_{ij} * b_{jk}
5:          if c_{ik} ∉ c_{i*} then
6:             c_{ik} = 0
7:          end if
8:          c_{ik} = c_{ik} + value
9:       end for
10:   end for
11: end for
```

en GPU, con especial atención en implementaciones sobre hardware de NVIDIA.

Reservar memoria para la matriz de salida

A diferencia de operaciones que admiten matrices o vectores densos, en donde se puede saber con facilidad la cantidad de memoria requerida para almacenar el resultado, en el caso de SpMM esto no es posible sin realizar cálculos previos. Esto se debe a que la distribución de elementos no nulos del resultado no es conocida con precisión hasta que se ejecuta la operación. Para abordar este problema, se distinguen cuatro estrategias distintas: método preciso, método probabilístico, método del límite superior y método progresivo.

El primer método, denominado método preciso, divide SpMM en dos etapas: simbólica y numérica. En la etapa simbólica, se computa la cantidad de elementos no nulos de la matriz resultante mediante la ejecución de una versión simplificada de SpMM que opera sobre la estructura de la matriz y no sobre los valores. Una implementación típica de la etapa simbólica utiliza el mismo patrón computacional que la etapa numérica, pero sobre una versión booleana de las matrices de entrada, lo cual permite utilizar operaciones menos costosas que las de punto flotante. Luego, la etapa numérica se encarga de determinar los valores reales. Las implementaciones de SpMM encontradas en cuSPARSE e Intel MKL son representativas de este método[56][57]. Una posible forma de optimizar la etapa simbólica es mediante técnicas de compresión de las matrices de entrada, en las cuales se representan los elementos como bits y se utilizan operadores de bits para acelerar los cálculos[11][58]. Si bien este método es relativamente caro ya que ejecuta el mismo patrón de cómputo dos veces, posee la ventaja que permite reutilizar los resultados de la etapa simbólica en

multiplicaciones de matrices con la misma estructura.

El método probabilístico busca estimar la cantidad de elementos no nulos de la matriz resultante a partir de muestreos aleatorios y análisis probabilístico de las matrices de entrada. La motivación de este enfoque es reducir los costos asociados a la etapa del cálculo simbólico. Sin embargo, es necesario reservar memoria adicional en caso de que la estimación inicial falle.

El método del límite superior calcula la máxima cantidad de valores que puede tener la matriz resultante y utiliza este cálculo al momento de reservar memoria. La ventaja de este método es que suele ser eficiente computacionalmente y simple de implementar. Sin embargo, se suele reservar una cantidad de memoria mucho mayor de la necesaria.

El método progresivo reserva memoria dinámicamente. Comienza con cierta cantidad de memoria y reserva memoria adicional cada vez que la memoria actual no es suficiente. Si la primera estimación falla, este enfoque tiende a ser ineficiente en GPUs.

Partición de matrices y balanceo de carga

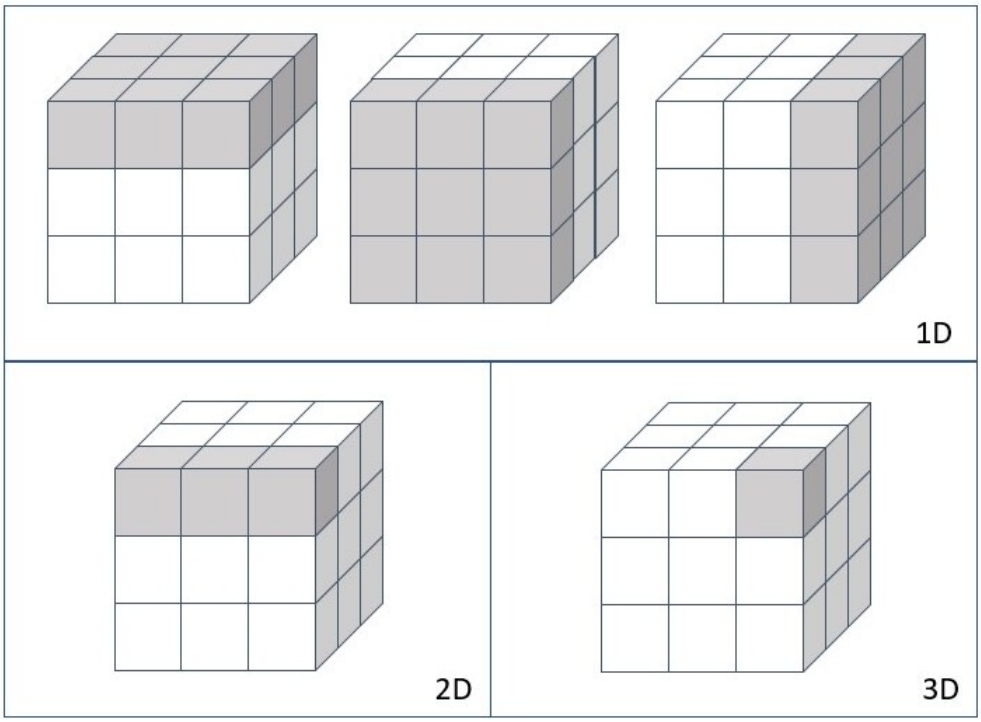


Figura 2.4: Ejemplos de particiones del cubo computacional, agrupados según la dimensión (1D, 2D o 3D) de la familia de algoritmos asociada

Este problema consiste en decidir cómo asignar el trabajo a realizar a las distintas unidades de procesamiento. La totalidad de las operaciones que deben realizarse para multiplicar dos matrices puede representarse mediante un cubo computacional. Un cubo computacional se define como un retículo de $n \times n$[1] en donde el *voxel* de la posición (i, j, k) corresponde con la multiplicación $A(i, k)B(k, j)$. Elegir una forma de asignar trabajo a cada procesador puede traducirse en definir una partición del cubo computacional. En este contexto, una posible primera clasificación de los algoritmos de SpMM consiste en distinguirlos según si la partición tiene en cuenta el patrón de dispersión de las matrices de entrada o no. Dentro del segundo caso, una familia importante de algoritmos son los que dividen el cubo en conjuntos de voxels que forman cuboides. Ballard et al.[59] distinguen estos algoritmos según la dimensión del cuboide: los algoritmos 1D corresponden con los que el cuboide tiene dos dimensiones de largo n, los algoritmos 2D corresponden con los que el cuboide tiene una dimensión de largo n y, por último, en los algoritmos 3D todas las dimensiones del cuboide tienen largo menor a n. La Figura 2.4 presenta gráficamente las distintas particiones mencionadas.

Cada manera de particionar el cubo computacional típicamente está relacionada con una formulación de SpMM para la Ecuación (2.1) distinta. A continuación, se presentan las distintas formulaciones posibles:

1. *Fila por fila:* Se obtienen las filas de C a partir de sumas de productos entre elementos pertenecientes una misma fila de A con una fila de B:

$$c_{i*} = \sum_k a_{ik} b_{k*} \tag{2.2}$$

   La mayor parte de la investigación sobre SpMM se basa en el algoritmo propuesto por Gustavson, que utiliza esta formulación.

2. *Columna por columna:* Análoga a la formulación 1 pero con columnas en vez de filas:

$$c_{*j} = \sum_k a_{*k} b_{kj} \tag{2.3}$$

3. *Producto exterior:* Se obtiene la matriz C como una suma de productos exte-

[1] La definición original asume matrices cuadradas por simplicidad, pero en el caso general en donde las dimensiones no son necesariamente iguales, podría utilizarse un ortoedro en vez de un cubo.

riores entre columnas de A y filas de B:

$$C = \sum_i a_{*i} \otimes b_{i*} \tag{2.4}$$

4. *Producto interno:* Se obtiene cada elemento de la matriz C como el producto interno de una fila de A con una columna de B:

$$c_{ij} = \sum_k a_{ik} b_{kj} \tag{2.5}$$

Las tres particiones 1D presentadas en la Figura 2.4 típicamente corresponden a las formulaciones 1, 2 y 3, respectivamente, mientras que la partición 2D corresponde a la formulación 4.

Deveci et al.[58] diseñaron un algoritmo de SpMM, *kkmem*, que utiliza una partición 1D por filas para distribuir el cómputo entre bloques de CUDA o warps. Posteriormente, presentaron tres algoritmos basados en *kkmem*[60], implementados tanto en procesadores *Knights Landing* de Intel (KNL)[61] como en GPU, que no requieren que la totalidad de las matrices de entrada estén simultáneamente en memoria del dispositivo. Para lograr esto, las variantes particionan las matrices en fragmentos que pueden estar distribuídos entre la memoria de la CPU y la de la GPU. El acceso a los datos se realiza mediante operaciones que exponen el hardware de almacenamiento asociado, a diferencia de operaciones sobre memoria unificada[1], que en arquitecturas recientes permiten que el programador trabaje de manera transparente con datos que exceden el tamaño máximo disponible en cualquiera de las memorias del sistema por separado. Todos los algoritmos presentados dividen las matrices de entrada en conjuntos de filas. En el caso de la matriz A, también se hace una partición por columnas, formando bloques de dos dimensiones. La Figura 2.5 presenta gráficamente la partición definida. Una de las variantes propuestas consiste en un loop que copia los fragmentos de una misma fila de A y C desde memoria de la CPU a memoria de la GPU. Luego, iterativamente se cargan los fragmentos de B, ejecutando el algoritmo base *kkmem* para realizar la multiplicación en cada paso. Finalmente, se carga el fragmento de C en memoria de la CPU y se procesa la siguiente fila de A y C. El siguiente método es análogo pero carga los fragmentos de B una única vez y luego carga iterativamente los de A y C. Para hacer un mejor uso de los recur-

[1] La memoria unificada es un componente del modelo de programación de CUDA que define un espacio de memoria administrada que permite que el programador acceda a los distintos dispositivos físicos mediante un mismo espacio de direccionamiento[6].

sos de cómputo, la tercer variante utiliza heurísticas para determinar cantidad de fragmentos que deben copiarse a memoria del dispositivo sin afectar el desempeño y posteriormente elige dentro de los métodos previos el que tiene menores costos de memoria asociados. En el caso particular que el almacenamiento del dispositivo sea suficientemente grande, los datos de entrada se copian por completo sin fragmentar. El desempeño de las variantes presentadas se comparó con el desempeño de *kkmem* asumiendo distintas formas de acceder a los datos: memoria de la CPU con el mecanismo de reemplazo de páginas desactivado (*pinned memory*), memoria de la GPU y memoria unificada. Se observó un desempeño, medido en base a GFLOPS, considerablemente mayor en las implementaciones basadas en fragmentos cuando las matrices de entrada no pueden ser almacenadas en su totalidad en memoria de la GPU.

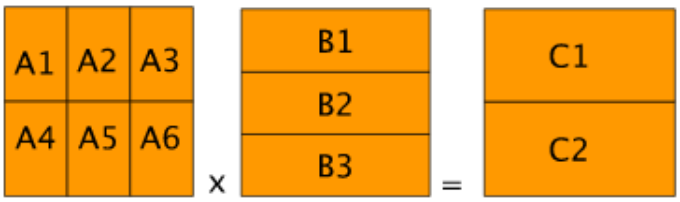


Figura 2.5: Partición de las matrices de entrada en el algoritmo basado en fragmentos propuesto por Deveci et al. Extraído de la publicación original[60]

Dado que los patrones de valores no nulos en las matrices de entrada pueden ser muy diversos, las implementaciones que dividen el trabajo en base a particiones estáticas del cubo computacional corren el riesgo de generar cargas desbalanceadas. Una posible excepción sucede con algunos algoritmos 2D y 3D en sistemas de memoria distribuida de gran escala[62][59]. En el contexto de la GPU, una estrategia para dividir el trabajo entre las unidades computacionales de forma balanceada es implementar la totalidad del algoritmo mediante primitivas paralelas. Un ejemplo de este enfoque es el algoritmo *Expansion-Sorting-Compression* (ESC)[63] y derivados. Para explicar el funcionamiento del algoritmo ESC, se parte de las matrices de entradas vistas como tuplas en formato de coordenadas (aunque en una implementación real típicamente se utilizaría un formato comprimido) y se realizan tres etapas:

1. *Expansión:* genera una lista de tuplas que representan las filas de B escaladas por los elementos de A. Es decir, dada la tupla (i, j, v_{in_1}) de A y la tuplas (j, k, v_{in_2}), para los distintos valores k de la fila j de B, se generan tuplas (i, k, v_{out}), con $v_{out} = v_{in_1} v_{in_2}$.

2. *Ordenamiento:* consiste en ordenar la lista de tuplas generadas por (i, k) para que tuplas contiguas correspondan a sumandos de una misma posición de la matriz de salida.

3. *Compresión:* se comprimen las tuplas correspondientes a la misma posición de la matriz de salida, sumando todos los valores asociados

Si bien las etapas de ESC pueden ser implementadas mediante un conjunto reducido de primitivas paralelas (*gather*, *scatter*, *scan* y *stable_sort_by_key*) que distribuyen el trabajo de forma balanceada entre procesadores, el desempeño de este enfoque está limitado por la cantidad de operaciones sobre memoria global. Dalton et al.[64] proponen una mejora a ESC que hace un mejor uso de los registros y la memoria compartida.

Otro algoritmo parcialmente basado en ESC es el propuesto por Liu y Vinter[65]. Este algoritmo propone una técnica de balanceo de carga que consiste en heurísticas. Primero, se distribuyen las filas de C entre 38 *bins* según la cantidad máxima de entradas de cada fila. Luego, se agrupan los bins en 5 grupos distintos. El primer grupo contiene un único bin con las filas con 0 elementos, mientras que, en el extremo opuesto, el quinto grupo contiene el último bin, que contiene las filas con más de 512 elementos. El resto de los bins están distruibidos entre grupos intermedios. Para computar las filas de cada grupo se utilizan tres métodos distintos, entre ellos ESC. El cómputo de cada grupo es asignado a distintos recursos de hardware, lo cual permite que se aprovechen mejor los recursos. Utilizando la terminología de NVIDIA, el tercer grupo se calcula utilizando un único warp, el cuarto se calcula utilizando un único bloque de CUDA y el quinto se calcula mediante múltiples bloques. Los primeros dos no se calculan en la GPU porque requieren operaciones triviales, lo cual se espera que mejore el balanceo de carga.

Nagasaka et al.[66] proponen un algoritmo de dos etapas que divide las filas de la matriz de salida en grupos según la máxima cantidad de operaciones necesarias para computarlas y luego según la cantidad de elementos no nulos de dichas filas. La primera división se utiliza en la etapa simbólica, mientras que la segunda se utiliza en la etapa numérica. La implementación consiste en múltiples kernels en paralelo con parámetros configurables, como la cantidad y el tamaño de bloques lanzados y la estrategia de cómputo asignada para el grupo. En cuanto a este último aspecto, las filas de algunos grupos son procesadas mediante un número fijo de threads de un mismo warp, mientras que otras son calculadas mediante un bloque de CUDA entero.

Acumulación de resultados parciales

La operación SpMM puede descomponerse como una suma (o acumulación) de resultados parciales, tal como se discutió en la sección anterior. La estructura de datos que almacena los resultados parciales, que puede ser densa o dispersa, y generalmente se denomina acumulador. La mayor parte de las implementaciones paralelas de SpMM se basan en la estrategia fila por fila presentada en el Algoritmo 1, en donde la etapa de acumulación corresponde con la línea 8. Las técnicas de acumulación pueden dividirse en tres categorías: acumulación densa, basada en hashing y basada en ordenamientos.

1. Acumulación densa

la primera clase de métodos consiste en utilizar arreglos densos para almacenar y acumular resultados intermedios. El acceso a los elementos se realiza de forma directa, a diferencia de los métodos basados en hashing, que deben calcular el hash y manejar colisiones. La principal ventaja de este tipo de acumuladores es que el acceso a los elementos es más eficiente. Sin embargo, esto implica requerimientos de memoria mayores. Deveci et al.[67] presentan un algoritmo implementado en CPU y KNL que utiliza un acumulador denso y argumentan que en estas arquitecturas los acumuladores dispersos son preferibles cuando los accesos a memoria son el cuello de botella, mientras que los acumuladores densos son preferibles en la medida en que los sistemas de memoria proveen más ancho de banda, asumiendo que la dimensión de las filas de la matriz de salida es suficientemente chica. También ofrecen una variante en GPU de su algoritmo pero la misma utiliza otro tipo de acumulador debido a altos requerimientos de memoria asociados. Patwary et al.[68] implementaron una versión paralela del algoritmo de Gustavson en procesadores Intel®Xeon® que utiliza un acumulador denso y compararon el desempeño de la implementación mencionada con implementaciones que acumulan utilizando técnicas de hashing, tanto en CPU como en GPU, obteniendo un mejor desempeño.

2. Acumulación en tablas de hash

Las técnicas basadas en hashing acumulan los resultados de una fila en tablas de hash, que utilizan como keys índices de columna de elementos de la matriz de salida asociados. Esto permite que se utilice menos cantidad de memoria que con los acumuladores densos, pero con la desventaja que se debe ejecutar una función de hash y posiblemente resolver colisiones para acceder a los elementos. El primer problema que surge al implementar esta estrategia es determinar el tamaño de memoria que

debe reservarse para la tabla. Dado que el tamaño de la tabla depende de la cantidad de elementos no nulos de la matriz de salida, las alternativas para determinar el tamaño de la tabla son las mismas que para determinar la cantidad de memoria necesaria para almacenar la matriz de salida, discutidas previamente. El siguiente problema es que para tablas suficientemente grandes, el único almacenamiento posible es en la memoria global de la GPU, lo cual conlleva problemas de desempeño ya que los accesos a la tabla de hash son sobre direcciones de memoria aleatorias y por lo tanto no son coalesced. Por esta razón, en lugar de definir una tabla única en memoria global para almacenar todos los valores de C, suelen utilizarse múltiples tablas más pequeñas que pueden ser alojadas en memoria compartida de cada SM, la cual es más rápida y permite patrones de acceso más flexibles. El algoritmo de dos etapas propuesto por Nagasaka et al.[66], que se basa en la estrategia mencionada, utiliza una tabla de hash tanto en la etapa simbólica como en la numérica. En el primer caso, calcula una cota máxima para el tamaño de las filas basada en la cantidad de productos intermedios. Si este número es superior al máximo tamaño permitido, se utiliza toda la memoria compartida disponible para almacenar la tabla. Si durante la ejecución del algoritmo se detecta que el tamaño de la tabla no es suficiente para una fila, se vuelve a realizar el procedimiento pero en memoria global. Para la etapa numérica, se utiliza la cantidad de elementos no nulos de la fila para reservar espacio para la tabla en memoria compartida. Luego de que los resultados de la fila fueron acumulados en la tabla, se organizan los valores de la tabla de forma contigua en memoria y se ordenan según su número de columna en forma ascendiente. Finalmente, se guardan los resultados en memoria global. Liu et al.[69] proponen una optimización que permite que se utilicen registros para disminuir la cantidad de accesos a memoria compartida en esta etapa.

Deveci et al.[60] utilizan un acumulador basado en tablas de hash que soporta inserciones y acumulaciones paralelas por parte de múltiples threads. Se almacenan en una estructura de lista encadenada y consisten en 4 arreglos paralelos: Ids, $Values$, $Begins$ y $Nexts$. Ids y $Values$ almacenan pares $(clave, valor)$, que tienen una semántica distinta según si se ejecuta la etapa simbólica o la numérica. $Begins$ almacena el índice del comienzo de la lista encadenada para cada valor de la función de hash. Por último, el arreglo $Nexts$ contiene los índices de los siguientes elementos de la lista. Posteriormente, los mismos autores implementaron un acumulador basado en tablas de hash pero con distinta estructura de datos asociada[67]. En este caso, la tabla se implementó mediante dos arreglos y *linear probing* como método para resolver colisiones. Además, la acumulación se realiza en dos niveles: el primer

nivel consiste en un acumulador en memoria compartida y el segundo nivel, que se crea dinámicamente cuando no hay espacio suficiente en memoria compartida, utiliza memoria global. Dado que a medida que la tabla se llena las búsquedas se vuelven muy costosas, definieron un parámetro que limita la máxima ocupación permitida. Si la ocupación del acumulador de primer nivel es mayor que el parámetro mencionado, las siguientes inserciones se hacen directamente en el acumulador de segundo nivel.

Anh et al.[70] diseñaron un algoritmo fila por fila que asigna a cada bloque de CUDA un conjunto de filas para calcular, éstas se acumulan utilizando una tabla de hash. A diferencia de los algoritmos descritos anteriormente, que estiman el tamaño de la tabla de hash, este algoritmo parte de un tamaño de tabla H y estima la cantidad de filas que debe ser asignada a cada bloque para que en promedio se aproveche todo el espacio de la tabla. Para lograr esto se utiliza el cociente de compresión δ de la matriz, que se define como la cantidad promedio de productos intermedios que son necesarios para generar un elemento no nulo de la matriz C. Para estimar δ, primero se calcula la cantidad total de productos intermedios y luego se aplica una técnica de muestreo para estimar la cantidad de elementos no nulos de C. Luego de haber estimado δ, se utiliza como estimación que insertar δH elementos en la tabla producirá H elementos no nulos, lo cual coincide con el tamaño de la tabla. Si bien esta heurística se basa en suposiciones que no siempre se cumplen, los investigadores afirman que en la práctica tiene un muy buen desempeño.

3. Acumulación basada en ordenamientos

Esta clase de métodos consiste en ordenar los resultados intermedios según los índices de columna y luego sumar los valores asociados al mismo índice. La diferencia entre los algoritmos propuestos usualmente se encuentra en la elección del método de ordenamiento utilizado, dentro de los cuales se destacan los algoritmos radix sort, merge sort y heap sort.

Una de las propuestas más influyentes dentro de esta categoría es ESC[63], el cual fue descrito en la sección anterior. La implementación original utiliza las primitivas de ordenamiento de la biblioteca Thrust, que están basadas en algoritmo radix sort[71]. Dalton et al.[64] utilizan el algoritmo radix sort B40C[72], que permite especificar el número y la ubicación de los bits de ordenamiento.

Los algoritmos basados en merge sort utilizan listas ordenadas como resultados intermedios y luego las combinan generando otra lista ordenada. Gremse et al.[73]

utilizan un algoritmo fila por fila que acumula los resultados mediante una operación similar a merge sort, pero que combina los elementos con el mismo índice de columna, logrando un efecto de compresión. Winter et al.[74] proponen un algoritmo de SpMM que adapta ESC para trabajar con memoria compartida. A diferencia de algoritmos similares que distribuyen filas enteras o una cantidad fija de operaciones intermedias, el enfoque propuesto distribuye entre bloques de CUDA una cantidad fija de elementos de A, ignorando los límites de las filas de matrices. El ordenamiento de los fragmentos resultantes se realiza en base a un orden global en la etapa de merge, y luego se ejecuta la operación prefix scan para combinar las filas compartidas por fragmentos, generando el resultado final.

Azad et al.[75] representan resultados intermedios mediante una lista de tuplas (i, j, val) que almacenan el índice de fila, el índice de columna y el valor no nulo, respectivamente. El algoritmo primero ordena cada lista de tuplas según (j, i) y luego se combinan las listas, generando los resultados finales. Para realizar un merge de k listas de tuplas, se utiliza un heap de tamaño k que almacena la entrada menor actual en cada lista de tuplas. Luego, una rutina de merge *multi-way* encuentra la tupla más chica del heap y la combina con el resultado actual.

Capítulo 3

Uso del formato bmSPARSE

En este capítulo se introduce el formato de almacenamiento de matrices dispersas bmSPARSE y un algoritmo de SpMM basado en dicho formato. En este último caso, se realiza un enfoque conceptual, en donde se detallan las distintas etapas comprendidas por el algoritmo y cómo están relacionadas entre sí, y un enfoque práctico, en donde se profundiza sobre detalles de implementación de las mismas

La Sección 3.1 describe el formato bmSPARSE. La Sección 3.2 detalla el funcionamiento de un algoritmo de multiplicación de matrices bmSPARSE. Finalmente, en la Sección 3.3 se describen las implementaciones de cada etapa del algoritmo.

En todas las secciones se adopta la convención de anteponer el identificador de una matriz al nombre de una estructura para referenciar las estructuras del formato *bmSPARSE*. A modo de ejemplo, se utiliza el identificador A_keys para hacer referencia al arreglo $keys$ de la matriz A.

3.1 Formato bmSPARSE

El formato bmSPARSE representa matrices dispersas mediante bloques de tamaño 8×8[1]. Por esta razón, en el resto del documento se toma la convención de que todas las coordenadas son para referenciar bloques, a menos que se indique lo contrario. A modo de ejemplo, A_{ij} hace referencia al bloque comprendido entre las posiciones $(8 \times i, 8 \times j)$ y $(8 \times (i+1), 8 \times (j+1))$ de la matriz original, y no al elemento (i, j). Algo análogo sucede cuando se habla de filas o columnas de una matriz, ya que las referencias son a filas o columnas de bloques.

[1] Podría ser otro tamaño.

El conjunto de bloques que resulta de la partición es descrito mediante las siguientes estructuras:

- *keys*: Arreglo de enteros (uint64_t) que representan la posición de un bloque en la matriz de bloques. Los primeros 32 bits se usan para codificar el número de fila, mientras que los últimos 32 bits se usan para codificar el número de columna. Las keys aparecen ordenadas por fila y luego por columna. La elección de uint64_t para representar las keys hace que sea posible representar matrices de hasta 2^{32} bloques de columnas y filas. En el caso de matrices de menor tamaño, se podría reducir el uso de memoria adaptando el formato para que permita utilizar 32 bits para representar las keys.

- *bmps*: Arreglo de enteros (uin64_t) que almacena en la posición i, un bitmap asociado al bloque en la posición $keys[i]$. Cada elemento del bloque se mapea a un bit del bitmap. El bit es 0 si el elemento del bloque es nulo y 1 en caso contrario.

- *values*: Arreglo con los valores no nulos de la matriz. Los valores de un mismo bloque se ingresan por orden de fila y luego por columna. Entre bloques distintos, los conjuntos de valores asociados a cada uno mantienen entre sí el orden determinado por el arreglo keys.

- $offsets$: Arreglo con los desplazamientos de los bloques en values. Los valores del primer bloque son los comprendidos entre $values[0]$ y $values[offsets[0]]$. Los valores del bloque $keys[i]$, para $i > 0$, son los comprendidos entre $values[offsets[i-1]]$ y $values[offsets[i]]$

A modo de ejemplo, la Figura 3.1 muestra cómo obtener los valores, la key y el bitmap de un bloque. Representar una matriz densa entera implica realizar dicho procedimiento para cada bloque y combinar los resultados en los arreglos descritos previamente.

3.2 SpMM para bmSPARSE

Dadas dos matrices de entrada, A y B, almacenadas en formato *bmSPARSE*, el algoritmo de multiplicación para este formato realiza dos grandes tareas. La primera es construir una lista, denominada *task list*, que determina qué conjunto de pares de bloques de A y de B deben multiplicarse para generar un bloque de la matriz

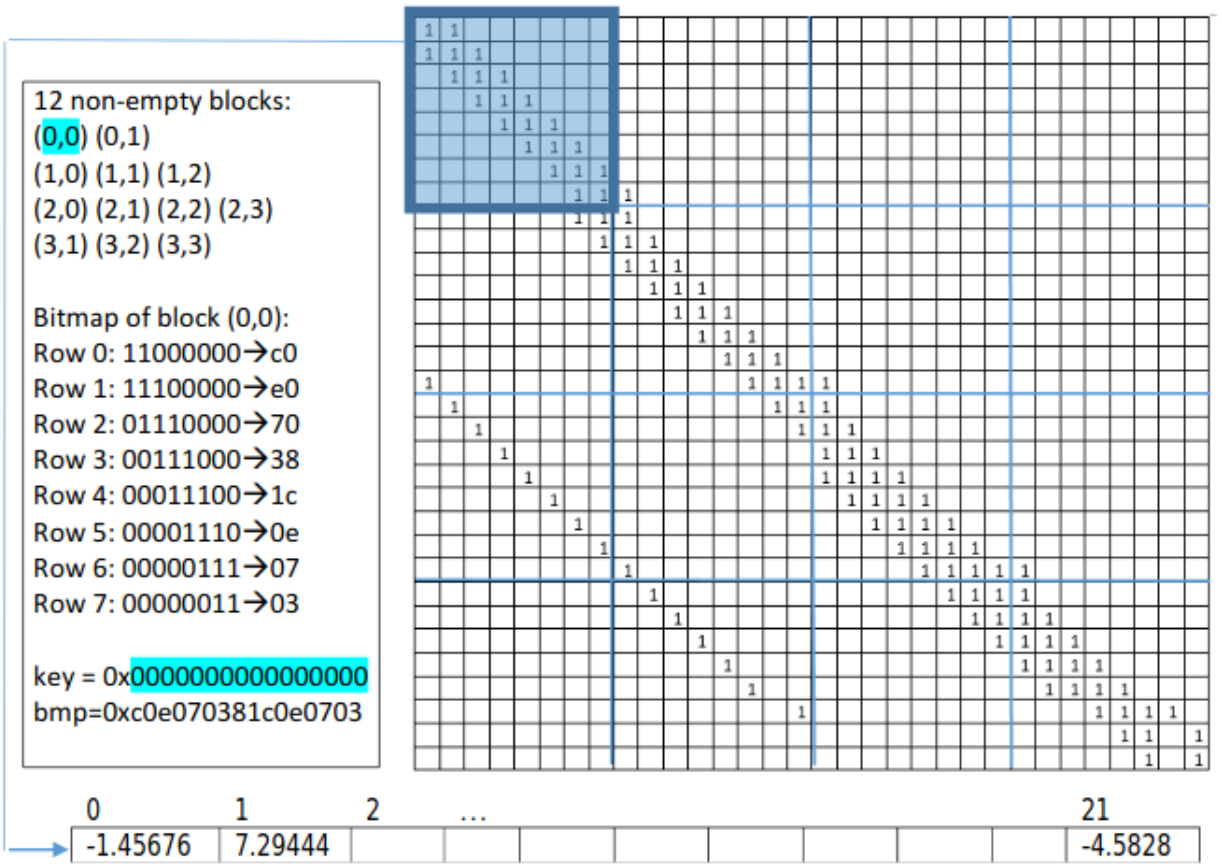


Figura 3.1: Ejemplo de representación en formato bmSPARSE. Extraído de "Regularizing Irregularity: Bitmap-Based and Portable Sparse Matrix Multiplication for Graph Data on GPUs"[11]

resultante C. La task list puede verse como una lista de tuplas (i, j, k), denominadas *tasks*, obtenidas a partir de la regla:

$$C_{ik} = \sum_j A_{ij} \times B_{jk}. \tag{3.1}$$

La segunda tarea es procesar las tasks para generar la matriz resultante.

El algoritmo está dividido en distintas etapas, que se identifican como $T_1, ..., T_9$. Existen diferentes versiones del algoritmo, cada una asociada a una secuencia distinta de etapas. Las secuencias válidas están representadas mediante caminos en el grafo dirigido de la Figura 3.2.

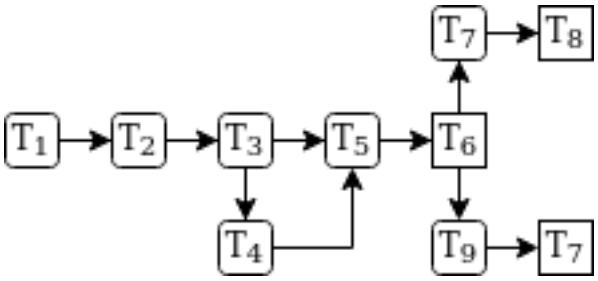


Figura 3.2: Alternativas de flujo de ejecución.

Las etapas T_1, T_2 y T_3 se encargan de crear la task list. Asumiendo que $A\_keys[n]$ almacena la key (i, j), para lograr el objetivo mencionado se considera a la task list como la unión de los conjuntos $t'_n, n \in [0, size(A\_keys))$, en donde t'_n representa el conjunto de todas las tuplas en las que participa la key (i, j) de A_keys. Si se define $B\_row_j = \{\ x \mid x \in B\_keys \wedge row(x) = j\}$, el conjunto de *keys* de la fila j de B, t'_n puede expresarse formalmente como:

$$t'_n = \{\ (i, j, k) \mid A\_keys[n] = (i, j) \wedge \exists x \in B\_row_j : col(x) = k\}. \tag{3.2}$$

Notar que en la definición 3.2, las tasks se representan como tuplas de coordenadas (i, j, k). Sin embargo, si $(i, j) = A\_keys[n]$ y $(j, k) = B\_keys[m]$, las tasks también pueden representarse mediante la tupla de posiciones (n, m). Si bien las representaciones son equivalentes y el algoritmo puede describirse en términos de ambas, se utilizará la segunda porque es en la que se basa la implementación. En la Ecuación (3.3) se define el conjunto t_n, que utiliza la representación de tasks como tupla de posiciones. Hay una correspondencia uno a uno entre los elementos de t'_n y t_n.

$$t_n = \{\ (n, m) \mid A\_keys[n] = (i, j) \wedge B\_keys[m] \in B\_row_j\} \tag{3.3}$$

La ventaja principal de definir la task list como la unión de conjuntos t_n es que estos conjuntos pueden ser calculados fácilmente gracias a las características del formato. Por un lado, observar que la segunda componente de las tasks de t_n son posiciones de una misma fila de bloques de la matriz B. Además, por cada elemento de la misma fila habrá una task en t_n. Por otro lado, el formato bmSPARSE garantiza que las keys están ordenadas por fila y luego por columna, por lo que las keys asociadas a una misma fila de bloques se almacenan en forma contigua. Esto permite que los conjuntos t_n puedan generarse en base a las posiciones en las que comienzan las filas de B y el tamaño de las mismas.

La etapa T_1 se encarga de calcular la cantidad de elementos de t_n, para todo n en el rango $[0, size(A\_keys))$. Por lo visto en la Ecuación (3.3), esto es lo mismo que calcular la cantidad de elementos no nulos en la fila j de B. La salida de esta etapa es B_count, un arreglo definido por la relación $B\_count[j] = |B\_row_j|$.

T_2 consiste en asociar el n-ésimo elemento de A_keys con $|t_n|$, calculado en T_1. La asociación se representa mediante el arreglo col_count, definido por la relación $col\_count[n] = B\_count[col(A\_keys[n])]$.

Fijando n como la primera coordenada de las tasks, la segunda coordenada toma valores en el rango $(pos_j, pos_j + col\_count[n])$, en donde pos_j es la posición del comienzo de la fila j en B_keys. Por lo visto anteriormente, la definición de t_n dada en la Ecuación (3.3) es equivalente con la de la Ecuación (3.4).

$$t_n = \{(n, pos[j] + idx) \mid idx \in [0, col\_count[n]])\}. \tag{3.4}$$

La etapa T_3 se encarga de crear el arreglo de desplazamientos, llamado pos, y posteriormente generar la task list en base a los arreglos construídos hasta el momento. La task list se representa mediante un arreglo, $task\_list$, compuesto por las tasks de todos los conjuntos t_n, ordenadas de forma ascendiente por n y luego por idx. Para generar el arreglo $task\_list$ se construyen dos arreglos auxiliares. El primero, $task\_keys$, se utiliza para determinar a qué conjunto t estará asociada una posición de $task\_list$. Es decir, $task\_keys[m] = n \iff task\_list[m] \in t_n$. El segundo, idx, indica en la m-ésima posición, cuál es el desplazamiento idx que forma la task $task\_list[m]$, de acuerdo a la Ecuación (3.3). A partir de estos dos vectores, la $task\_list$ queda determinada por la Ecuación (3.5).

$$task\_list[m] = (task\_key[m], idx[m] + pos[col(A\_keys[task\_key[m]])]) \tag{3.5}$$

El resultado de ejecutar una task es un bloque que se utiliza como un sumando en el cálculo de alguno de los bloques de la matriz resultante, como se muestra en la Ecuación (3.1). El bitmap asociado al bloque de salida puede calcularse a partir de los bitmaps de ambas matrices de entrada. Si el bitmap es nulo, se puede saber de antemano que los valores del bloque son todos nulos. En dicho caso, la task puede ignorarse sin que el resultado final cambie. En este trabajo se extiende el algoritmo original[11] mediante la etapa T_4, que consiste en eliminar de la task list las tasks que no contribuyen a los valores de la matriz resultante. Es una etapa opcional, en el sentido de que la correctitud del algoritmo no depende de la misma. La motivación de este paso es lograr un mayor desempeño en las etapas posteriores.

En la etapa T_5 se ordena la task list de modo que tasks asociadas a un mismo bloque de salida sean contiguas. Utilizando la representación de tasks como tuplas de coordenadas (i, j, k), lo anterior equivale a ordenar según (i, k). Como el arreglo $task\_list$ utiliza otra representación de tasks, para determinar el orden relativo entre dos elementos, primero se realiza la conversión y luego se comparan.

La etapa T_6 se encarga de determinar qué bloques de la matriz resultante serán no nulos, lo cual se corresponde con el arreglo keys del formato bmSPARSE. Para lograr esto, se interpreta el arreglo $task\_list$ como un arreglo de tasks representadas como (i, j, k). Luego se elimina la segunda coordenada, obteniendo un arreglo de tuplas (i, k). Finalmente, si $m > 1$ elementos consecutivos son iguales, entonces se eliminan $m - 1$ de ellos, dejando un único representante por conjunto de tasks asociadas a un mismo bloque. El resultado de este proceso es el arreglo de keys, ordenado por fila y luego por columna.

Se profundiza en dos de las cuatro posibilidades para reservar memoria para el arreglo que almacena los valores de la matriz de salida vistas en el capítulo anterior. La primera consiste en asumir el peor caso y reservar espacio para la máxima cantidad de valores posibles, determinada por la cantidad de keys generadas en la etapa T_6 y el tamaño de bloque de bmSPARSE. En este caso, el arreglo de valores es de tamaño $64 \times \#keys$. La segunda posibilidad es calcular el arreglo de bitmaps de la matriz resultante antes de realizar la multiplicación. Esto permite calcular el largo que debe tener el arreglo de valores antes de que se utilice. la primera alternativa mencionada corresponde con la versión del algoritmo que ejecuta T_7 luego de T_6, mientras que la segunda corresponde con la que ejecuta T_9 luego de T_6.

La etapa T_7 procesa las tasks para generar los bloques resultantes. Procesar una task implica construir una versión densa de los bloques de entrada, realizar la multiplicación y sumar el bloque parcial de resultados al bloque de salida correspondiente.

Dada una task (n, m), las posiciones no nulas de los bloques de A y B pueden obtenerse a partir de los bitmaps $A\_bmp[n]$ y $B\_bmp[m]$, respectivamente. Por definición del formato bmSPARSE, los valores de un mismo bloque son contiguos en el arreglo *values*. En el caso de la matriz A, para $n > 0$, los valores del bloque asociado se encuentran comprendidos entre las posiciones $A\_offset[n-1]$ y $A\_offset[n]$ de A_values, mientras que para $n = 0$, el rango de posiciones es $[0, A\_offset[0]]$. El procedimiento para la matriz B es análogo.

En la etapa T_8, se toma como entrada el arreglo de valores generado en la etapa T_7 y se crea un nuevo arreglo que contiene únicamente los valores no nulos del arreglo original. Los órdenes relativos se mantienen.

La etapa T_9 se encarga de, a partir de los bitmaps de las matrices de entrada, calcular el arreglo de bitmaps de la matriz resultante.

3.3 Implementación de multiplicación de matrices en bmSPARSE

Todas las etapas se implementan en C++ en base a algoritmos disponibles en *Thrust*[76], una biblioteca de algoritmos paralelos inspirada en la biblioteca estándar de C++ (STL). La única excepción es la etapa de multiplicación de bloques, que se realiza sobre matrices densas utilizando un *kernel* de *CUDA*.

Una misma etapa del algoritmo estudiado en la sección anterior puede ser implementada de distintas formas. En el grafo de la Figura 3.2, las implementaciones de una misma etapa se agrupan en un único nodo, por lo que en la práctica, la cantidad de variantes posibles es mayor.

La propuesta de implementación de este trabajo difiere de la original en las siguientes formas:

- T_3 utiliza *thrust::scatter* en vez de *thrust::gather* al expandir.
- T_4 no es parte del algoritmo original.
- Se utiliza una representación distinta de las tasks. En la implementación original, cada task es representada mediante tuplas (i, j, k), mientras que en las implementaciones realizadas en este trabajo, las tasks se representan mediante tuplas de posiciones (pos_A, pos_B), en donde cada elemento se refiere a posiciones de los arreglos de keys correspondientes. Esto requiere cambios en las

etapas T_3, T_5 y T_6, en donde deben utilizarse functors que contemplen esta representación.

- En la propuesta original, luego de la etapa de ordenamiento debe ejecutarse una búsqueda para generar posiciones, mientras que en las variantes implementadas éstas se obtienen por defecto y se realizan distintas variantes de T_6, que se diferencian en cómo generar las keys de la matriz final.
- En T_8 se implementa la idea original (con *thrust::copy_if*) y otra alternativa.
- El código de la etapa del cálculo simbólico no figura en la propuesta original, pero su descripción sugiere que se basa en una versión simplificada del kernel de multiplicación. En la propuesta de este trabajo, esta etapa es implementada mediante primitivas paralelas.
- Las variantes del kernel multiplicación difieren sustancialmente de la implementación original, en donde sólo se contempla una estrategia similar a la primera variante.

A continuación, se describe con mayor profundidad las opciones exploradas.

T1 Calcular $|t_n|$

El objetivo de esta etapa es construir un arreglo *B_count* determinado por $|t_n| = B\_count[j] = |B\_row_j|$. Es posible calcular $|B\_row_j|$, para $j \in [0, size(B\_keys))$, mediante la primitiva *thrust::reduce_by_key*, utilizando como arreglo de keys a B_keys y el número de fila de los bloques como criterio de comparación. El orden de los elementos en *B_count* es el esperado gracias a que las keys en B_keys están ordenadas por fila y luego por columna.

T2 Asociar $A\_keys[n]$ con $|t_n|$

Esta etapa busca construir un arreglo *col_count*, definido por la relación $col\_count[n] = B\_count[col(A\_keys[n])]$. Esto puede lograrse a partir de una llamada a *thrust::gather*, utilizando un arreglo con columnas extraídas de las keys de A_keys como arreglo de índices y *B_count* como arreglo de datos. Los valores de las columnas pueden obtenerse sin necesidad de almacenar el arreglo en memoria a través de un iterador *thrust::transform_iterator* y un *functor* que devuelva la columna de una key.

T3 Construir la task list

Implementar esta etapa requiere que se construyan los arreglos *pos*, *task_key*, *idx* y finalmente *task_list*. La forma en la que se determina *task_list* está dada por la Ecuación (3.5).

El arreglo de desplazamientos *pos*, puede generarse mediante una llamada a *thrust::exclusive_scan* sobre *B_count*.

Para crear el arreglo *task_keys*, primero se calculan las posiciones en donde empezará t_n en *task_list*, para todo n. Esto se logra mediante una llamada a *thrust::exclusive_scan* sobre *B_count*. Luego se utiliza *thrust::scatter* y un *thrust::counting_iterator* para almacenar cada n en la posición asociada, calculada en el paso anterior. Finalmente, se ejecuta *thrust::inclusive_scan* sobre el arreglo, utilizando *thrust::maximum* como el operador binario.

El arreglo *task_keys* puede construirse mediante la primitiva *thrust::exclusive_scan_by_key*, utilizando el arreglo *task_keys* como arreglo de keys y un iterador *thrust::constant_iterator* inicializado en 1 como iterador de valores.

La combinación de *task_keys* e *idx* para formar *task_list* se hace a través de la primitiva $thrust :: transform$ y un functor que crea las tasks según la Ecuación (3.5).

T4 Remover tasks innecesarias

El filtrado de tasks se realiza mediante la primitiva *thrust::remove_if*, que toma como entrada el arreglo de tasks y un functor que determina si el bitmap que se obtendría al ejecutar la task es nulo.

Se implementan tres versiones de esta etapa, que surgen de distintas implementaciones del functor mencionado:

Versión 1 (naive): Se itera sobre cada dimensión del bitmap resultante, chequeando posibles intersecciones entre los bits de la fila de A y de la columna de B correspondientes. La función retorna si se encuentra un bit no nulo.

Versión 2 (transpose-bmps): En la implementación anterior los bits de una columna de B no forman parte del mismo byte, por lo que hay que hacer cálculos adicionales para determinar el byte correspondiente a cada elemento de la columna. En esta versión del functor, primero se transpone el bitmap de la matriz T_2. Para transponer eficientemente el bitmap se utiliza el algoritmo detallado en *Hacker's Delight*[77].

Finalmente, cada bit del bitmap de salida puede calcularse mediante un *and* bit a bit entre el byte de fila y el byte de columna correspondiente.

Versión 3 (column-major-bmps): Esta versión se basa en la misma idea que la anterior, pero los bitmaps de T_2 están organizados de manera *column-major* a nivel de bits. Para lograr esto, se utiliza la *flag* del constructor de la clase *bmSpMatrix* que permite construir el arreglo de bitmaps de esta forma.

T5 Ordenar

En la etapa T_7, un bloque de CUDA se encarga de procesar todas las tasks asociadas a un bloque de la matriz de salida, por lo que es conveniente ordenar la task list para que las tasks asociadas a un mismo bloque de salida sean contiguas. Para ordenar el arreglo *task_list* se utiliza la primitiva *thrust::sort*. El *functor* que se utiliza para comparar elementos primero realiza la transformación de representación de la task y luego determina un mínimo entre dos elementos según el valor de (i, k).

T6 Determinar el arreglo de keys de C

La etapa se implementa de tres formas:

Versión 1 (two-transformations): Si se ve a la task list como tuplas (i, j, k) compuestas por coordenadas de matrices, determinar el *layout* de C implica agrupar las tasks por (i, k) y cada representante de grupo obtenido como una key de C. Lo primero se hizo en la etapa T_5, mientras que lo segundo puede lograrse mediante la primitiva *thrust::reduce_by_key* utilizando *task_list* como arreglo de keys. Sin embargo, surgen dos inconvenientes que deben resolverse para implementar esta estrategia. El primero es que en las etapas previas se representan las tasks mediante tuplas (n, pos_k), por lo que el chequeo de igualdad de keys no es directo. Por otro lado, por defecto, el arreglo de keys de salida también utilizará esta representación, la cual sólo es útil en la etapa T_7. Dado que esta representación sólo es válida durante la ejecución del algoritmo, se necesita convertir el arreglo de keys de C a la representación utilizada por el formato *bmSPARSE*.

El primer problema puede solucionarse permitiendo que el functor tenga acceso a los arreglos de keys de ambas matrices, lo cual le permite generar la representación necesaria para comparar las keys. Una forma de solucionar el segundo problema es realizar un cambio de representación mediante una llamada a la primitiva *thrust::transform* sobre el arreglo de keys reducidas.

Hasta ahora sólo se ha contemplado el arreglo de keys que se obtiene de *thrust::reduce_by_key*, pero el arreglo de valores también es de utilidad. Si se usa un iterador constante con el valor 1 como arreglo de valores de entrada, el arreglo de valores de salida puede utilizarse como entrada de *trust::inclusive_scan* para calcular las posiciones de *task_list* en donde comienzan las tasks asociadas a cada bloque de la matriz de salida. Esto permite que cada bloque de cómputo en la etapa T_7 determine qué tasks le corresponde procesar.

Versión 2 (two-transformations-transform-output-it): Implementación análoga a la primera, pero se utiliza un iterador de tipo *thrust::transform_output_iterator* como iterador de keys de salida, lo cual hace que sea innecesaria la llamada posterior a *thrust::transform*.

Versión 3 (one-transformation): En la primera implementación, se realizan dos conversiones de representación: una cuando se chequea por la igualdad en el arreglo de keys y otra al transformar el arreglo de keys de salida. Una forma de realizar una única transformación es asegurarse que no se use *task_list* como arreglo de keys de entrada, sino un arreglo con las tasks previamente representadas como tuplas (i, k), ignorando j. Esto permitiría que el functor pueda hacer una comparación directa entre elementos y que el arreglo de keys de salida utilice la representación deseada.

Para evitar reservar memoria adicional que almacene el nuevo arreglo, es posible utilizar *thrust::make_transform_iterator* para crear un iterador que genere la representación deseada a medida que se accede a elementos de *task_list*.

T7 Multiplicar

Todas las versiones implementadas de esta etapa consisten en un *kernel* de *BLAS* que se encarga de procesar las tasks del arreglo *task_list* para determinar la matriz resultante. Todas las versiones, con la excepción de la bmp-generation, asumen que se ejecutó T_9 previamente. En especial se consideran variantes basadas en CUDA[6], a continuación se detalla cada variante:

Versión 1 (bmp-generation): Cada bloque de *CUDA* calcula un bloque distinto de la matriz saliente. Calcular un bloque de salida implica ejecutar todas las tasks asociadas, por lo que la primera tarea del bloque de *CUDA* es identificar las tasks que tiene asignadas. Gracias a la etapa T_5, las tasks asignadas a un mismo bloque son contiguas en el arreglo *task_list*. Este hecho puede utilizarse, en conjunto al arreglo

construído al final de la etapa T_6, para determinar las posiciones de la primera y última task que le corresponden a un bloque de *CUDA*.

Una vez que fueron determinados los límites sobre *task_list*, se itera sobre las tasks, cargando los bloques de la matrices de entrada en memoria compartida, realizando la multiplicación y guardando los resultados parciales en memoria compartida. El código asociado al loop mencionado se presenta en el Código 3.1.

Al momento de calcular las posiciones sobre el arreglo de bitmaps y desplazamientos, se vuelve especialmente relevante la representación de tasks utilizada. En la Sección 3.2 se discutieron dos alternativas: la primera representa las tasks mediante una tupla de coordenadas de matrices, como se muestra en la Ecuación (3.2), mientras que la segunda, presentada en la Ecuación (3.3), se basa en posiciones de los arreglos de keys de las matrices de entrada. La implementación utiliza la segunda ya que ésta permite obtener tanto los bitmaps asociados a los bloques entrantes, como los desplazamientos en los vectores de valores, a partir de las mismas posiciones que definen la task, por definición del formato *bmSPARSE*.

En la función que carga los bloques en memoria, *shmem_load*, los threads cargan un 0 o un valor no nulo del bloque de entrada según si el bit en la posición $threadIdx.x$ del bitmap *bmp* es 0 o 1. En caso de que el bit sea 1, se debe contar la cantidad de bits en 1 en posiciones anteriores del bitmap para calcular el desplazamiento que debe utilizarse para acceder al valor correspondiente de *values*. Lo anterior se implementa mediante la primitiva *__popcll*, que cuenta la cantidad de bits en 1 en una variable de tipo $uint64\_t$. Como entrada se le hace un shift a la derecha a *bmp* para que no tenga en cuenta los 1 a partir de la posición $threadIdx.x$: $bmp >> (64 - threadIdx.x)$.

Algorítmo 3.1: Loop principal en kernel de multiplicación.

```
1  ...
2  for (; cur_task < last_task; cur_task++) {
3      /* Se obtienen los bitmaps y los desplazamientos en los vectores de valores*/
4      uint64_t bmp_A = A_bmps[task_list[2 * cur_task].first];
5      uint64_t bmp_B = B_bmps[task_list[2 * cur_task].second];
6      float *A_block_values = A_values + A_offsets[task_list[2 * cur_task].first]];
7      float *B_block_values = B_values + B_offsets[task_list[2 * cur_task].second];
8
9      /* Se cargan los bloques A y B en las direcciones de memoria compartida apuntadas
           por block_A y block_B, respectivamente */
10     __syncthreads();
11     shmem_load(bmp_A, block_A, A_block_values);
12     shmem_load(bmp_B, block_B, B_block_values);
13     __syncthreads();
14
15     /* Se multiplican los bloques A y B y se almacenan los resultados en out */
16     for (int t = 0; t < 8; t++) {
17         out[threadIdx.x] += block_A[(threadIdx.x / 8) * 8 + t] * block_B[t * 8 + (
               threadIdx.x % 8)];
18     }
19 }
```

Una vez que los bloques de entrada fueron cargados en memoria compartida, cada posición del bloque de salida es calculada por un único thread, que multiplica la fila y columna correspondiente y almacena el resultado en $out[threadIdx.x]$.

Luego de haber procesado todas las tasks, se calcula el bitmap del bloque saliente a partir de la primitiva *__ballot_sync*, que puede utilizarse para que cada warp del bloque genere 32 bits del bitmap final. El bitmap generado se utiliza para contar la cantidad de elementos no nulos del bloque, utilizando nuevamente la primitiva *__popcll*. Finalmente, se escribe en memoria global el bitmap, el bloque calculado y la cantidad de elementos del mismo.

Versión 2 (no-bmp-generation): Esta versión es igual a bmp-generation, salvo que no realiza el cálculo del arreglo de bitmaps sobre el final, dado que asume que se ejecutó la variante $T_{9.1}$ previamente.

Versión 3 (tr-indices-shmem): Esta variante asume que se ejecutó la variante de $T_{9.2}$ pre-

viamente, lo cual significa que los bitmaps de la matriz B estén dispuestos en formato column major a nivel de bits. Para contemplar este cambio, modifica la posición que cada thread tiene asignada en memoria compartida a la hora de cargar los bloques de las matrices entrantes. La posición pasa de ser $threadIdx.x$ a $(threadIdx.x\,\%8) \times 8 + (threadIdx.x/8)$, lo cual transpone nuevamente el bloque, permitiendo que las posiciones del loop original vuelvan a ser correctas.

El resto de la implementación es idéntica a la variante no-bmp-generation.

Versión 4 (tr-indices-global): Esta implementación utiliza la asignación de un thread al elemento transpuesto de la matriz, como se describió en tr-indices-shmem. La diferencia es que en vez de utilizar dicho valor para acceder a memoria compartida, se utiliza como un desplazamiento sobre memoria global, manteniendo el resto de las direcciones como no-bmp-generation.

Otro cambio menor introducido en esta versión es que se eliminan ciertos accesos a memoria compartida innecesarios. En versiones previas, se escriben ceros en las regiones de memoria compartida asociadas a ambos bloques de entrada y el bloque de salida. Sin embargo, la función *shmem_load* fue implementada de manera que los threads que no tienen asociado un elemento no nulo, escriben un cero en la posición que les corresponde, por lo que sólo es necesario inicializar el bloque de salida.

Versión 5 (output-register): En esta versión se utilizan registros para almacenar los resultados de las operaciones intermedias y no se escriben en memoria compartida los resultados parciales. En el Código 3.2 se presenta la modificación incorporada. Luego de procesar todas las tasks, la variable *result* almacena el resultado final.

Algorítmo 3.2: Multiplicación de bloques con registros.

```
1  ...
2  float result = 0;
3  for (; cur_task < last_task; cur_task++) {
4  ...
5      for (int t = 0; t < 8; t++) {
6          result +=
7              block_A[(threadIdx.x / 8) * 8 + t]
8              * block_B[t * 8 + (threadIdx.x % 8)];
9      }
10 }
```

El resto de la implementación es idéntica a la versión tr-indices-shmem, con la excepción de que esta variante escribe la variable *result* en memoria global al momento de almacenar los valores de la matriz resultante, en lugar de escribir el resultado acumulado en memoria compartida. Como consecuencia de esto, al escribir los resultados finales en memoria global, los valores se leen directamente de *result*, en lugar de la región de memoria a la que apunta *out* en versiones previas (la cual se deja de utilizar).

Versión 6 (shmem-prefetching): En cada iteración del loop principal del kernel, presentado en el Código 3.1, todos los threads del bloque leen los mismos valores de los arreglos de bitmaps y el arreglo de tasks. Esta versión de la etapa busca hacer un mejor uso del ancho de banda de memoria global haciendo que se carguen en memoria compartida los bitmaps y los offsets de varias tasks en forma paralela, antes de realizar la multiplicación de bloques. Para lograr esto se definen dos arreglos sobre memoria compartida: $bmps$ y $offsets$. En ambos arreglos, las posiciones pares almacenan las variables de la matriz A y las impares las de B. Además, los datos están ordenados de manera que la posición $2i$ de uno de los arreglos, almacena ambos datos de la task i en forma consecutiva. Luego, se adapta el loop principal para que se itere sobre la cantidad de tasks cargadas en memoria compartida y se lean los valores de bitmaps y offsets de memoria compartida.

El resto de la implementación es idéntica a la variante output-register.

Versión 7 (warp-multiply): En las variantes previas, cada bloque de la matriz de salida es calculado por un bloque de CUDA distinto. Para lograr esto, se realiza un mapeo uno a uno entre threads y elementos de un bloque de salida, por lo que se lanzan bloques de CUDA con *64* elementos. En esta versión cada bloque de salida es calculado por un warp, por lo que cada bloque de CUDA pasa a calcular dos bloques de salida. Esta modificación requiere que cada thread calcule dos posiciones del bloque final, por lo que, a diferencia de la variante output-register, *result* pasa a ser un arreglo de float de dos elementos. La Figura 3.3 representa gráficamente la asignación del cálculo de elementos a threads. Observar que cada elemento de B es utilizado dos veces por el mismo thread.

Un cambio importante incorporado es que se eliminan las llamadas a *__syncthreads()*, ya que la misma es utilizada para sincronizar threads a nivel de bloques, pero en esta versión los warps pasan a trabajar sobre datos distintos por lo que no necesitan sincronizarse entre sí. Sin embargo, es importante notar que sí es necesario

alguna forma de sincronización a nivel de warps, ya que las suposiciones de que los warps ejecutan en modo *lock-step* o que las lecturas/escrituras son visibles a nivel de warp dejaron de ser válidas a partir de la arquitectura Volta[78]. En este caso, la sincronización se realiza mediante la primitiva *__syncwarp()*.

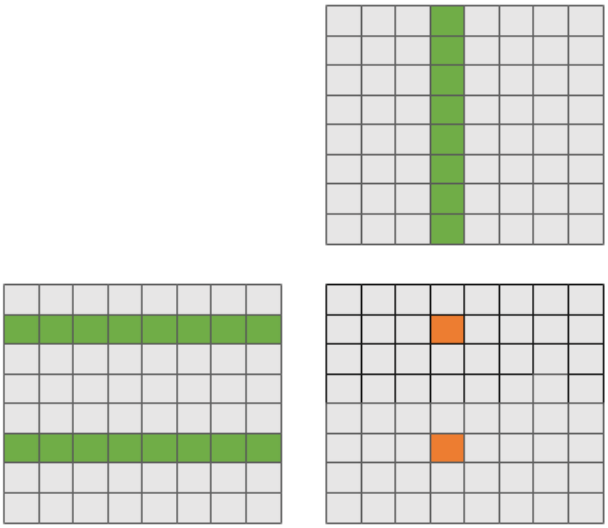


Figura 3.3: Las celdas anaranjadas indican posiciones del bloque de salida asignadas a un mismo thread. El resto de las celdas coloreadas representan elementos de las matrices de entrada que están involucrados en el cálculo de los elementos señalados previamente.

T8 Compactar

Se implementan dos variantes de esta etapa:

Versión 1 (for-each): Esta variante se implementa a través de la primitiva *thrust::for_each*. Cada thread ejecuta un functor que se encarga de compactar los valores de un mismo bloque. Para determinar las posiciones del arreglo que almacena los valores compactados se utiliza el arreglo *C_offsets*, generado en la etapa T6 . Los valores del bloque n en el arreglo a compactar se encuentran a partir de la posición $64 \times n$

Versión 2 (copy-if): Esta versión utiliza la primitiva *thrust::copy_if* para copiar a un arreglo nuevo los valores no nulos del arreglo de valores inicial.

T9 Calcular bitmaps de C en función de bitmaps de A y B

La etapa se implementa de dos formas:

Versión 1 (3-nested-loops): la primera tarea que se realiza es definir un iterador de tipo *thrust::make_transform_iterator* para generar el bitmap resultante de ejecutar una task a partir de los bitmaps entrantes. Para calcular el bit (i, j) de salida, el functor asociado itera sobre la dimensión k de ambos bitmaps de entrada. Lo anterior puede verse en el Código 3.3. Al momento de determinar qué bit debe estar en 1 en cada bitmap, se parte de un bitmap con 1 en la primera posición ($0x8000000000000000$) y se realiza un shift a la derecha según el número de fila y columna. En caso de que ambos bits correspondientes estén en 1, los resultados se acumulan en el bitmap *result* mediante un *or* bit a bit.

Algorítmo 3.3: Cálculo de bitmap resultante.

```
1   ...
2   uint64_t result = 0;
3   for (int i = 0; i < 8; i++) {
4       for (int j = 0; j < 8; j++) {
5           for (int k = 0; k < 8; k++) {
6               const bool A_bit_set =
7                   A_bmp & (0x8000000000000000 >> (i * 8 + k));
8               const bool B_bit_set =
9                   B_bmp & (0x8000000000000000 >> (k * 8 + j));
10              if (A_bit_set && B_bit_set) {
11                  result |= 0x8000000000000000 >> (i * 8 + j);
12              }
13          }
14      }
15  }
```

Una vez que se tiene todos los bitmaps asociados a las multiplicaciones de bloques, se utiliza *thrust::reduce_by_key* para realizar un *or* bit a bit entre los bitmaps de un mismo bloque de salida. Dado que sólo se tienen en cuenta las posiciones de los elementos no nulos, y los valores, es posible que alguno de los elementos resulte ser nulo y se almacenen ceros explícitos.

Versión 2 (2-nested-loops): En la versión 3-nested-loops, se necesitan 8 iteraciones sobre k para determinar un bit de la matriz resultante. En cada una de esas iteraciones, se realizan cálculos adicionales para determinar la posición del bit buscado. Finalmente, se realiza un *and* bit a bit para determinar el bit resultante debería estar encendido. Una forma de reducir la cantidad de operaciones es realizar un *and* bit a bit entre el

byte que almacena la fila i del primer bitmap, con el byte que almacena la columna j del segundo. De esta manera, se puede determinar si el valor del bit resultante en una única operación, ya que éste será 1 cuando alguno de los bits del resultado de la operación anterior sea 1. Sin embargo, esto no es posible a menos que los bitmaps de la matriz B estén dispuestos en formato column major a nivel de bits, lo cual permite que los bits de una misma columna estén contenidos en el mismo byte. Por esta razón, se asume que la matriz B fue generada con esto en mente. El Código 3.4 implementa la estrategia mencionada.

Algorítmo 3.4: Cálculo de bitmap resultante.

```
1   ...
2   uint64_t result = 0;
3   for (int i = 0; i < 8; i++) {
4       uint64_t first_bmp_row = (first_bmp << (8 * i)) & 0xFF00000000000000;
5       uint64_t row_position = 0x8000000000000000 >> i * 8;
6       for (int j = 0; j < 8; j++) {
7           const uint64_t second_bmp_col =
8               (second_bmp << (8 * j)) & 0xFF00000000000000;
9           if (first_bmp_row & second_bmp_col) {
10              result |= row_position >> j;
11          }
12      }
13  }
```

El resto de la implementación es idéntica a la variante 3-nested-loops.

Capítulo 4

Evaluación experimental

En este capítulo se hace una evaluación experimental de distintas variantes del algoritmo de SpMM basado en bmSPARSE descrito en el capítulo anterior, así como una evaluación del formato.

La Sección 4.1 describe tanto el hardware como el software utilizado para realizar las pruebas. Además, presenta las matrices que se utilizan como entrada de los experimentos. La Sección 4.2 analiza el consumo de memoria asociado al formato bmSPARSE y lo compara con el de CSR. Por último, en la Sección 4.3 se presentan y discuten los resultados de ejecutar experimentos con múltiples variantes del algoritmo de SpMM basado en el formato bmSPARSE.

4.1 Entorno de evaluación

Todas las pruebas se realizan en un sistema compuesto por una CPU Intel i7-9750H@2.60GHz y una GPU NVIDIA GeForce GTX 1660 Ti, la cual corresponde a la arquitectura Turing. La programación en GPU se realiza mediante CUDA 10.2 y la biblioteca de algoritmos paralelos Thrust[56] asociada.

Tabla 4.1: Desempeño teórico en punto flotante para la GPU GTX 1660 Ti[79].

Precisión	Desempeño
FP16	10.87 TFLOPS
FP32	5.44 TFLOPS
FP64	169.9 TFLOPS

Se evalúa tanto el consumo de memoria del formato como el desempeño de distintas implementaciones del algoritmo bmSPARSE. En ambos casos, se utilizan matrices dispersas cuadradas obtenidas de la colección de matrices *SuiteSparse Matrix*

Collection[80], identificadas con un número del 1 al 9. Las características de cada matriz son presentadas en la Tabla 4.2.

Tabla 4.2: Principales características de las matrices utilizadas. Las matrices 1-3 tienen dimensión cercana a 10^4, las matrices 4-6 una dimensión cercana a 10^5 y el resto a 10^6.

Nombre	Identificador	Bloques	NNZ	Dimensión
cryg10000	1	8613	49699	10000
Goodwin_030	2	20728	312814	10142
ted_A_unscaled	3	13761	424587	10605
Goodwin_095	4	203725	3226066	100037
matrix_9	5	148928	2121550	103430
hcircuit	6	90082	513072	105676
webbase-1M	7	550761	3105536	1000005
t2em	8	572656	4590832	921632
atmosmodd	9	1410884	8814880	1270432

El desempeño medido según tiempo de ejecución se compara con el de la biblioteca cuSPARSE[56], también parte de CUDA Toolkit.

4.2 Consumo de memoria del formato

El primer experimento refiere al estudio del consumo de memoria por el método estudiado. Con este objetivo, la Tabla 4.3 detalla los requerimientos de memoria para almacenar las matrices presentadas en la Sección 4.1 en formato bmSPARSE y CSR, utilizando MBs como unidad de medida.

Tabla 4.3: Espacio requerido (en MB) para almacenar las matrices en formato bmSPARSE y CSR.

Formato	Espacio requerido según matriz								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
bmSPARSE	0,3	1,5	1,9	15,3	10,3	3,1	19,0	25,2	52,2
CSR	0,4	2,5	3,4	26,2	17,4	4,5	28,8	40,4	75,6

Tal como se discutió en el capítulo anterior, asumiendo que utilizan elementos de 4 bytes, representar una matriz en formato CSR requiere de $4(2nnz + dim)$ bytes. En el caso de las matrices analizadas (y en general), nnz es significativamente mayor que dim, por lo que es el factor que más contribuye al tamaño final. Esto explica por qué matrices con la misma dimensión tienen requerimientos de memoria tan

distintos, así como el hecho de que las matrices 4 y 7 ocupan un espacio similar, a pesar de que la dimensión de la matriz 7 es 10 veces mayor que la de la matriz 4.

En el formato bmSPARSE, la cantidad de memoria necesaria para representar una matriz, depende principalmente de nnz y la cantidad de bloques $block\_num$. La cantidad de elementos del vector de valores es idéntica en ambas representaciones. Luego, los vectores $keys$, $offsets$ y $bmps$ tienen $block\_num$ elementos, por lo que el tamaño total de la representación de una matriz en el formato es de $4(nnz + 3block\_num)$ bytes. Lo anterior implica que la diferencia entre espacio requerido por bmSPARSE y CSR es de $4(3block\_num - nnz - dim)$ bytes, por lo tanto, para que bmSPARSE requiera más memoria, es necesario que se cumpla la desigualdad (4.1).

$$3block\_num > nnz + dim \tag{4.1}$$

Si se conoce la cantidad promedio de elementos no nulos por bloque, $block\_nnz$, se puede expresar la cantidad de bloques en función de nnz y $block\_nnz$ como $block\_num = \frac{nnz}{block\_nnz}$. Esto permite reformular la desigualdad (4.1) de la siguiente manera:

$$3block\_num = 3\frac{nnz}{block\_nnz} > nnz + dim \iff (\frac{3}{block\_nnz} - 1)nnz < dim \tag{4.2}$$

A partir de la expresión (4.2), se observa que una condición suficiente para garantizar que el formato bmSPARSE requiera menos memoria es que $block\_nnz$ sea mayor o igual a 3, puesto que dim es siempre positivo. Las matrices presentadas en la Tabla 4.2 cumplen con la condición suficiente mencionada anteriormente. En el caso general, no se conoce el valor $block\_nnz$ antes de generar la representación de una matriz en formato bmSPARSE, lo cual permitiría que la comparación de la memoria requerida por ambos formatos pueda hacerse en forma directa. Si se asume el peor caso ($block\_nnz = 1$), se llega a otra condición suficiente para determinar que el bmSPARSE requiere menos memoria: $2nnz < dim$[1]. En el escenario en que la desigualdad anterior es falsa, no es posible sacar conclusiones sin tener más información acerca de $block\_nnz$.

[1]Esta condición es completamente inusual.

4.3 Desempeño del algoritmo SpMM

Las variantes del algoritmo de multiplicación evaluadas se identifican con un número asociado a la secuencia de etapas ejecutada. En todos los casos, se ejecutan las etapas T_1, T_2, T_3, T_5 y T_6 utilizando una implementación común para cada una de ellas. El resto de las etapas pueden estar implementadas de distintas maneras o no incluirse en caso de ser opcionales, según lo detallado en las secciones 3.2 y 3.3 del Capítulo 3. La Tabla 4.4 presenta la versión de las etapas utilizadas por cada variante.

Tabla 4.4: Versiones de los kernels T_4, T_7, T_8 y T_9 de las variantes SpMM consideradas. Para identificar la implementación asociada a una etapa se utiliza una cruz en la celda correspondiente. Para representar el hecho de que una etapa opcional no es ejecutada, se dejan todas las celdas asociadas a la misma sin marcar.

Etapa	**Versión del kernel**	**Variante SpMM**									
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
T_4	transpose-bmps	X		X		X					
	column-major-bmps						X	X	X	X	X
T_7	bmp-generation	X	X	X	X						
	no-bmp-generation					X					
	tr-indices-shmem						X				
	tr-indices-global							X			
	output-register								X		
	shmem-prefetching									X	
	warp-multiply										X
T_8	for-each	X	X								
	copy-if			X	X						
T_9	3-nested-loops					X					
	2-nested-loops						X	X	X	X	X

A continuación se da una breve descripción de los cambios incorporados en cada variante del algoritmo:

1. Implementación base.
2. Se elimina la etapa encargada de remover tasks de la task list que no contribuyen al resultado final (T_4).
3. Similar a la implementación base, con la diferencia de que en la etapa T_8 se utiliza la primitiva *thrust::copy_if* y no *thrust::for_each*.

4. Se combinan los cambios introducidos en las variantes 2 y 3.

5. El cálculo del arreglo de bitmaps de la matriz resultante (T_9) se realiza antes de la multiplicación. Como consecuencia, no se ejecuta la etapa T_8 y la etapa T_7 debe ser ligeramente modificada.

6. Se asume que la matriz B fue cargada en formato column-major. Como consecuencia, en el kernel utilizado en T_4 deja de ser necesario transponer bitmaps. En la etapa T_7 debe modificarse ligeramente la sección encargada de cargar valores en memoria compartida. Por último, se implementa el loop de la etapa T_9 mediante menos operaciones.

7. Se modifica la etapa T_7 para eliminar conflictos de banco. También se elimina un acceso innecesario a memoria compartida.

8. Se modifica la etapa T_7 para acumular resultados en registros en lugar de memoria compartida.

9. Se modifica la etapa T_7 para hacer un mejor uso del ancho de banda de memoria global al momento de cargar bitmaps y offsets de tasks.

10. Cada bloque de salida pasa a ser calculado por un único warp, en lugar de un bloque de CUDA entero.

A menos que se indique lo contrario, las tablas fueron generadas asumiendo que tanto las matrices de entrada como las de salida almacenan números en punto flotante de simple precisión (*floats*).

4.3.1 Implementación base

En primer término, se profundiza el análisis del tiempo de ejecución de cada etapa de la variante 1 del método basado en bmSPARSE. Notar que, a diferencia de lo que sucede en el álgebra densa, en el álgebra dispersa no se supone que existe una relación directa entre la dimensión de la matriz y el costo computacional. En muchos casos, se asume que esta dependencia está relacionada con la cantidad de valores no-nulos u otras características.

La Tabla 4.5 detalla los tiempos de ejecución de la implementación base[1] al multiplicar cada matriz por sí misma. Adicionalmente, los mismos se desglosan según cada una de las etapas ejecutadas.

[1]Se toma la variante 1 como la línea base.

Tabla 4.5: Tiempo de ejecución (en µs) de multiplicar matrices dispersas mediante la variante 1 del algoritmo de SpMM basado en bmSPARSE. Se asume que las matrices están en memoria del dispositivo, es decir, no se incluye el tiempo de transferencia.

Etapa	**Tiempo según matriz**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T_1	53	58	56	731	185	391	693	1084	2000
T_2	39	43	41	269	172	110	251	436	1858
T_3	202	536	339	1808	1668	1255	3788	2122	6238
T_4	42	91	58	592	492	385	1310	603	2249
T_5	167	1080	630	8947	7141	4199	14626	7944	33276
T_6	174	397	303	2106	1997	1391	3519	2387	6002
T_7	257	1731	949	16711	15938	10648	31330	17783	61583
T_8	57	1619	2635	17340	1017	3254	50965	4998	7945
Total	994	5557	5015	48509	28613	21636	106485	37359	121155
Std Dev	42	41	27	149	1582	68	247	605	2983

En esta implementación, la etapa T_1 realiza una llamada a la primitiva *thrust::reduce_by_key* sobre el arreglo de keys, por lo que es esperable que la duración de T_1 dependa mayoritariamente del tamaño de ese arreglo, que se corresponde con la tercer columna de la Tabla 4.2. Esta hipótesis puede corroborarse a partir de las tablas 4.2 y 4.5, en donde puede observarse que a mayor cantidad de bloques, mayor es la duración de T_1.

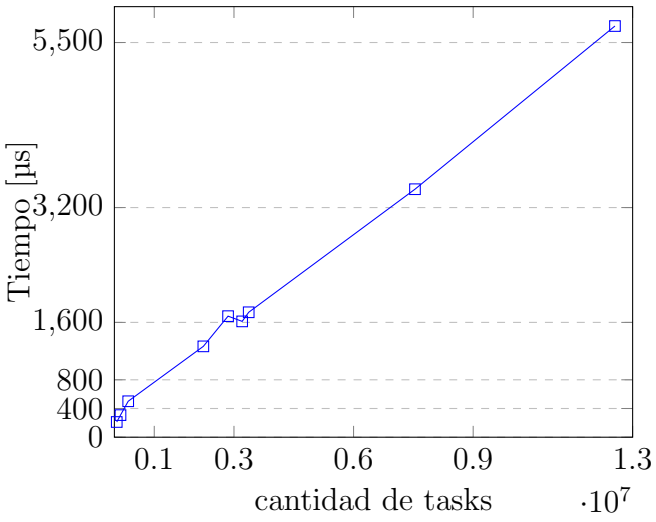
En la etapa T_2, se hace un acceso al vector *B_count* por cada elemento de *A_keys*. Por un razonamiento análogo al anterior, puede suponerse que la duración de la etapa depende mayoritariamente del tamaño de *A_keys*. Al igual que antes, este hecho puede verificarse comparando la cantidad de bloques y los tiempos de ejecución de las matrices.

El objetivo principal de la etapa T_3 es crear el arreglo *task_list*, por lo que se puede descartar el parámetro *nnz* para intentar explicar los tiempos de ejecución. Las tasks se forman de a bloques y los valores no nulos de la matriz sólo son relevantes en la medida que forman nuevos bloques. La densidad de los bloques no se tiene en cuenta en el procedimiento para determinar qué bloques formarán una task. Por otro lado, notar que el tiempo de ejecución de T_3 con la matriz 7 es casi el doble del tiempo que con la matriz 8. Ninguno del resto de los parámetros con los que se describen las matrices parece explicar esta diferencia.

Aplicando un argumento similar a los expuestos anteriormente, los tiempos de T_3 deberían depender mayoritariamente de la cantidad de tasks que forman parte de la task list. Si esto fuese así, debería suceder que al aumentar la cantidad de

Figura 4.1: Tiempo promedio de ejecución (en µs) de la etapa T_3 en función de la cantidad de tasks que forman parte de la task list. Se consideran todas las matrices de prueba.

Figura 4.1



tasks aumenten los tiempos de ejecución en todos los casos. Esta hipótesis puede corroborarse en la Figura 4.1, en la cual puede observarse una dependencia lineal entre ambas variables.

Si bien la cantidad de tasks es la variable que mejor predice la duración de T_3, típicamente no se cuenta con este dato antes de realizar la multiplicación. En ese caso, podría utilizarse la cantidad de bloques y la dimensión de una matriz para estimar la duración de T_3. Por un lado, mientras más bloques, más probable es que se forme una task entre dos de ellos. Sin embargo, a medida que aumenta la dimensión, hay más posibles posiciones para los bloques, lo cual disminuye la probabilidad de que formen una task. Esto explica por qué, salvo algunas excepciones, los tiempos de T_3 van aumentando a medida que la cantidad de bloques aumenta.

La Tabla 4.6 detalla, para cada matriz, la cantidad de tasks que forman parte del vector *task_list*, construido en la etapa T_3, y el porcentaje de tasks que posteriormente son descartadas en la etapa T_4.

En la etapa T_4, sería razonable que los tiempos de ejecución dependan del tamaño de la task list, al igual que como sucede en la etapa T_3. Esto puede corroborarse a partir de los datos de las tablas 4.5 y 4.6, o simplemente observando que los cocientes entre datos consecutivos de la Tabla 4.5 son similares en las filas de T_3 y T_4.

Tabla 4.6: Cantidad de tasks generadas en T_3 y porcentaje de tasks eliminadas en la etapa T_4. Datos válidos para cualquier versión del algoritmo que ejecute T_4

Matriz	# tasks iniciales	% tasks eliminadas
1	59512	36,3
2	346798	14,4
3	153486	1,6
4	3369528	13,4
5	3205366	20,0
6	2231426	38,5
7	7540274	42,6
8	2851764	7,8
9	12556668	22,1

Otra cuestión que puede explorarse es la medida en que el porcentaje de task eliminadas, también presentes en la Tabla 4.6, afecta el tiempo de ejecución de T_4. Hay dos maneras en las que el desempeño del algoritmo en esta etapa podría ser afectado por esta variable. La primera está relacionada con el comportamiento global de la primitiva *thrust::remove_if*, ya que, dependiendo de la implementación de la misma, es posible que el costo de realizar la eliminación varíe según la cantidad de elementos a eliminar. Por otro lado, el trabajo que debe realizar el *functor* que se ejecuta para chequear si una task debe ser eliminada, varía según la task. Para descartar una task, el *functor* debe asegurarse de que todos los bits del bitmap que se obtiene al ejecutar la task son nulos, mientras que si encuentra un bit no nulo, puede retornar de forma inmediata sin realizar el cálculo para el resto de los bits.

Buscando responder los cuestionamientos anteriores, se modificó la implementación de bmSPARSE para poder evaluar el impacto de cada factor mencionado sobre el tiempo de T_4. Para medir el impacto del *functor* sobre los tiempos, se hicieron cambios en los vectores de bitmaps de cada matriz de manera que el *functor* tarde más en ejecutar, pero la cantidad de bitmaps a eliminar sea la misma. De esta forma, el trabajo que realiza la primitiva *thrust::remove_if* por fuera de la ejecución del *functor* se mantiene constante. En el mejor caso, una task no se elimina cuando el bitmap resultante tiene un 1 en su primer bit y no es necesario chequear los valores del resto de los bytes el bitmap. En el peor caso, el bitmap resultante es nulo, pero de todas formas debe iterarse sobre cada byte del bitmap para corroborarlo. No se notó una diferencia significativa en los tiempos de la etapa T_4 al ejecutar esta versión modificada de implementación sobre ninguna de las matrices.

En el caso del primer factor mencionado, se realizaron dos cambios sobre la implementación. En el primero, se modificó el *functor* para que decida de forma

inmediata eliminar cualquier task que reciba. En el segundo, se modificó el *functor* para que no elimine ninguna task. De esta manera se puede comparar de forma aislada la relación entre la cantidad de tasks a eliminar y el comportamiento de *thrust::remove_if*. Lo que se observó en todos los casos, es que el tiempo de ejecución de la versión que elimina todas las tasks es aproximadamente 30 % menor que la versión que conserva todas las tasks. Es decir, es menos costoso eliminar valores en el arreglo que mantenerlos. En base a este resultado y los datos presentados en la Tabla 4.6, podría afirmarse que las multiplicaciones de las matrices 3 y 8 tienen el peor desempeño en relación a esta etapa, en términos relativos. Por otro lado, como se verá más adelante, la cantidad de tasks descartadas tiene un gran impacto sobre el desempeño de las etapas posteriores.

En las etapas T_5, T_6 y T_7 se repite un patrón similar entre los tiempos de ejecución de cada matriz, ya que en estas etapas también se procesa la task list. Sin embargo, es esperable que el orden entre los tiempos de ejecución de las matrices en etapas posteriores a T_4 sea distinto al de las etapas anteriores, producto de haber descartado una cantidad suficientemente elevada de tasks.

4.3.2 Impacto de ejecutar T_4

Para entender cómo la ejecución de la etapa T_4 afecta los tiempos de las etapas posteriores, se puede comparar el desempeño de las implementaciones 1 y 3 con las implementaciones 2 y 4, respectivamente. Las primeras ejecutan la etapa T_4 mientras que las últimas no lo hacen. La comparación de los tiempos de ejecución de ambas versiones se detalla en las tablas 4.7 y 4.8.

A partir de la Figura 4.2, puede observarse que en las etapas T_5, T_6 y T_7, se obtiene una reducción lineal en el tiempo de ejecución con respecto a la cantidad de tasks descartadas. Esta observación vale para todas las versiones del algoritmo consideradas en esta sección, ya que las implementaciones de T_5, T_6 y T_7 utilizadas son las mismas.

Una diferencia importante entre las implementaciones que no ejecutan T_4 y las implementaciones que sí lo hacen es que, en en el primer caso, es posible que se reserve memoria para bloques de la matriz resultante que habrían sido descartados en la etapa T_4 en caso de ejecutarse. Esto, a su vez, incrementa el tamaño de otras estructuras. Por esta razón, sería esperable que en la etapa T_8, las implementaciones que no ejecutan T_4 tengan un desempeño igual o peor que las que sí lo hacen. Sin embargo, sucede lo contrario en las variantes 1 y 2 del algoritmo. Mediante un análisis

de reportes generados por el profiler Nsight Compute[81], el cual brinda información sobre cómo los recursos del hardware son utilizados por un kernel, puede observarse que una posible explicación para la diferencia de tiempo es un peor uso de los *caches* por el kernel asociado a *thrust::for_each*. La cantidad de instrucciones de acceso a memoria tiende a ser mayor cuando no se ejecuta T_4 (por tener mayor cantidad de bloques), y los *hit rates* sobre *L1* y *L2* menores, lo cual hace que se transfieran más bytes entre las distintas unidades de memoria. Las implementaciones 3 y 4, basadas en *thrust::copy_if*, no sufren de este problema.

Tabla 4.7: Diferencias (en porcentaje y en µs) entre tiempos de ejecución de etapas posteriores a T_3 de la implementación 1 con respecto a la 2

Etapa	**Diferencias porcentuales según matriz**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T_5	52,5	89,9	98,1	86,4	79,4	60,8	57,6	93,4	78,2
T_6	57,5	94,8	98,4	92,2	80,1	65,3	58,8	94,9	85,5
T_7	64,8	85,0	98,3	85,8	77,3	61,5	56,8	90,3	79,0
T_8	98,3	128,1	99,1	122,8	118,7	145,4	110,3	158,4	140,3
TOTAL %	61,3	97,9	98,7	97,5	79,2	68,2	75,7	97,7	81,7
TOTAL µs	595	4814	4503	45219	25293	19501	100096	32821	106331
Std Dev	67	104	55	128	1813	87	85	416	4123

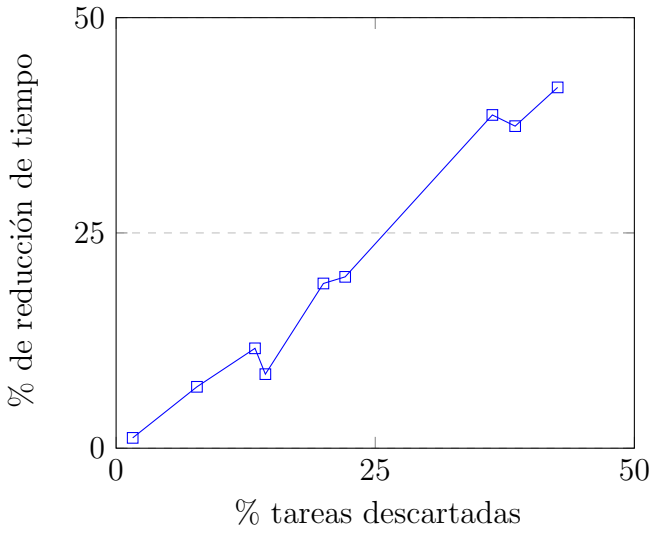
Tabla 4.8: Análoga a la Tabla 4.7, pero se compara la implementación 3 con respecto a la 4

Etapa	**Diferencias porcentuales según matriz**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T_5	54,8	89,8	100,5	86,5	80,9	61,1	57,7	93,4	78,3
T_6	63,3	99,3	98,0	92,9	81,5	65,2	59,6	94,8	85,1
T_7	65,8	85,1	97,9	85,7	80,2	61,5	57,0	90,4	76,9
T_8	79,4	93,4	97,4	85,0	76,9	65,3	62,8	87,8	75,2
TOTAL %	63,2	88,6	98,6	86,4	80,2	62,1	57,9	91,3	77,6
TOTAL µs	624	3475	2110	29115	27240	18247	54717	30381	105688
Std Dev	62	63	75	370	886	102	263	638	4260

El tiempo de ejecución total es menor en todas las implementaciones que ejecutan la etapa T_4, incluso en la matriz 3, en la que sólo se descarta el $1,6\,\%$ de tasks. Dentro de estas implementaciones, el desempeño de 3 tiende a ser el mejor. Particularmente con las matrices 3, 4 y 7, en donde se observan reducciones de hasta el $50\,\%$ del tiempo total.

Figura 4.2: Porcentaje de reducción de tiempo promedio de las etapas T_5, T_6 y T_7 (Tabla 4.8) en función del porcentaje de tasks eliminadas en la etapa T_4 (Tabla 4.6).

Figura 4.2



4.3.3 Impacto de ejecutar T_9

Para responder la pregunta de si es conveniente realizar el cálculo del arreglo de bitmaps y offsets antes de la multiplicación, se comparan implementaciones que realizan el cálculo previo con la versión 3, que se elige como la mejor representante de las que no lo hacen. La Tabla 4.9 detalla los tiempos de ejecución de las versiones 3, 5 y 6

En la Tabla 4.9 puede notarse que el tiempo de la etapa T_7 para la versión 5 es menor que el de la versión 3 en todos los casos. La razón principal es que, en la primera versión mencionada, el cálculo del arreglo offsets y bitmaps de salida se considera parte de T_9, mientras que en la segunda se considera parte de T_7. Sin embargo, al comparar el tiempo total de ambas versiones se da la relación inversa. Esto se debe a que la etapa T_8 en 3 ejecuta en menos tiempo que la etapa T_9 en 5. Con la versión 6 pasa algo similar en T_7, pero el mayor desempeño en la etapa T_9 hace que los tiempos totales también sean favorables a la misma. Dentro de la implementación de T_9 utilizada por 5, la mayor parte del tiempo transcurre en la ejecución del functor que genera bitmaps a partir de tasks. Para verificar la afirmación anterior, se sustituyó el functor original por un functor trivial y se observó una disminución de hasta el 90 % del tiempo de la etapa. La versión 6 incorpora la optimización de este functor detallada en la descripción de la implementación *2-nested-loops* de la etapa T_9.

Tabla 4.9: Tiempo promedio (en µs) de duración de las etapas posteriores a T_6 en las variantes 3, 5 y 6. Las últimas dos filas muestran el promedio de la suma de los tiempos de las etapas mencionadas y la desviación estándar asociada, respectivamente.

Etapa	Versión	Tiempo según matriz								
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T_7	3	260	1730	952	16793	15985	10635	31320	17782	60671
	5	226	1650	887	16099	14094	9381	28287	16105	53778
	6	236	1708	915	16516	14615	9764	29045	16585	58154
T_8	3	76	225	222	1304	2399	1995	5464	2452	6727
T_9	5	233	1002	621	8301	5723	3521	10750	5957	21081
	6	174	234	198	1023	1057	779	1847	1113	3454
Total	3	336	1955	1174	18097	18384	12630	36784	20234	67398
	5	459	2652	1508	24400	19818	12902	39038	22062	74860
	6	410	1942	1113	17539	15672	10544	30892	17698	61609
Std Dev	3	5	13	10	83	1205	18	33	62	4328
	5	8	10	14	110	628	27	174	43	5611
	6	29	2	16	9	934	57	58	46	3288

Una pequeña diferencia inesperada se da en los tiempos de la etapa T_7 en las variantes que ejecutan T_9, especialmente con la última matriz de prueba. La causa de este fenómeno es que utilizar la optimización *2-nested-loops* requiere que cada bitmap de la matriz B se represente mediante su transpuesto. Los bitmaps se utilizan a la hora de cargar la matriz en memoria, por lo que la implementación 6 debe implementar la etapa T_7 de forma distinta. Puntualmente, al momento de cargar la matriz B, cada thread se encarga de escribir el valor correspondiente en la posición transpuesta del bloque de memoria compartida. Esto genera un conflicto de bancos en memoria compartida, dado que distintos threads de un mismo warp acceden al mismo banco, debiendo serializar los accesos. Para alivianar este problema, la variante 7 utiliza la implementación *tr-indices-global* de T_7, que accede a memoria compartida de la misma forma que la variante 5, evitando los conflictos de banco. Sin embargo, la forma en la que accede a memoria global es distinta ya que asume que dentro de un mismo bloque los elementos están organizados en forma column-major. Como la matriz es dispersa y los threads que no tienen asignado un elemento no nulo no acceden a memoria global, es posible que la cantidad de accesos a memoria global varíe según la distribución de ceros de la matriz. A modo de ejemplo, un análisis obtenido mediante la herramienta Nsight Compute muestra que para la matriz 9, la variante 5 ejecuta 168M instrucciones de lectura, mientras que la variante 7 ejecuta 174M. Sin embargo, como puede observarse al comparar las tablas 4.9 y 4.10, la penalización en términos de tiempo de ejecución de los conflictos de banco mencionados previamente es ligeramente mayor, por lo que la versión *tr-indices-*

global del kernel, utilizada por la variante 7, es preferible a la *tr-indices-shmem*, utilizada por la variante 6.

Por lo visto anteriormente, el resto del análisis del rendimiento del algoritmo se centra en implementaciones que realizan el cálculo de los arreglos de bitmaps y offsets antes de realizar la multiplicación. Además, mantienen la estrategia para cargar elementos desde memoria global introducida en la versión *tr-indices-global* de la etapa T_7.

4.3.4 Optimización del kernel de multiplicación de bloques

La etapa T_7 se corresponde con el kernel de multiplicación y es la etapa de mayor costo en términos de tiempo de ejecución para las variantes analizadas hasta el momento. Los tiempos de ejecución de las las variantes 7, 8, 9 y 10, las cuales varían únicamente en la implementación de esta etapa, son detallados en la Tabla 4.10. Puede observarse que cada variante representa una reducción en el tiempo de ejecución con respecto a la anterior. En particular, la última ejecuta en menos de la mitad del tiempo que la primera. El resto de esta sección analiza reportes obtenidos de la herramienta Nsight Compute para explicar el desempeño de cada optimización. Por simplicidad, se elige la matriz 9 como representativa del resto, pero existen pequeñas variaciones según la elección de matriz.

Tabla 4.10: Tiempo promedio (en µs) de duración de la ejecución de la etapa T_7 en las variantes 7, 8, 9 y 10 del algoritmo. En el caso de esta última, se analizan dos instancias de la misma: la primera es la descrita previamente y la segunda hace uso del qualifier __launch_bounds__().

Etapa	**Versión**	**Inst**	**Tiempo según matriz**								
			1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T_7	7	-	227	1662	892	16189	14447	9378	28066	16049	56093
	8	-	162	1165	645	11356	11347	7371	21481	11824	41334
	9	-	139	955	550	9338	10987	7393	20609	10945	36690
	10	1	96	632	382	6196	7489	5303	14695	7659	25841
	10	2	89	604	349	5851	6310	4374	12480	6563	23314
Std Dev	7	-	0	6	1	69	1016	8	72	50	3748
	8	-	2	0	2	36	602	260	54	26	1857
	9	-	0	2	1	17	722	9	30	14	1926
	10	1	0	2	3	17	403	3	8	15	1043
	10	2	1	0	0	11	419	3	8	19	1297

El reporte asociado la variante 7 muestra que el porcentaje de utilización de los recursos de cómputo es significativamente mayor a los recursos de memoria, es decir, esta etapa puede considerarse *compute bound*.

Una lectura de la métrica *smsp__inst_issued.avg.per_cycle_active*, que cuenta la cantidad promedio de instrucciones emitidas por ciclo, permite ver que los *scheduler* pueden emitir una instrucción cada 3,7 ciclos, en contraste con el máximo de una instrucción por ciclo que admite la arquitectura. Esto puede suceder porque no hay suficientes warps activos o porque existen motivos que impiden que los warp activos ejecuten su próxima instrucción en un ciclo dado, escenario conocido como *stall*. la primera pregunta puede ser respondida analizando la *Occupancy*, que representa la razón entre la cantidad de warps activos por multiprocesador y la máxima cantidad de warps activos posibles. Las variantes de esta métrica, *Theorical Occupancy* y *Achieved Occupancy*, tienen valores de 100 % y 90 %, respectivamente, lo cual es un indicador que la capacidad del hardware para procesar warps concurrentemente es elevada y que probablemente sea necesario reducir la cantidad de *stalls* para mejorar el desempeño. Los motivos principales por los cuales suceden *stalls* en el kernel aparecen listados como *MIO Throttle*, *Long Scoreboard* y *Short Scoreboard*. la primera situación sucede cuando un warp ejecuta una instrucción de entrada/salida en momentos en donde las *pipelines* de *MIO (Memory Input Output)* están saturadas. Dentro de estas instrucciones se contemplan los accesos a memoria compartida, pero no los accesos a memoria del dispositivo[81]. El resto de los stalls suceden cuando una instrucción no puede ejecutarse porque aún no se ha resuelto una dependencia de datos asociada a una lectura sobre memoria del dispositivo o memoria compartida, respectivamente.

La cantidad de *stalls* ocasionadas por *MIO Throttle* es consistente con la distribución de las instrucciones de *assembler SASS* ejecutadas, ya que dos de las instrucciones ejecutadas con más frecuencia, *STS* y *LDS*, involucran accesos a memoria compartida. La sección de código en donde más ocurren esta clase de *stalls* es la que se encarga de multiplicar bloques asociados a la task que se está procesando en el momento. Este segmento se presenta al final del bloque de código 3.1. El *SASS* generado para esa sección muestra que en cada iteración se realizan dos lecturas de memoria compartida mediante la instrucción *LDS*, y una escritura mediante la instrucción *STS*. Dentro del muestreo realizado por el profiler, entre un 70 % y un 90 % de estas instrucciones no son ejecutadas a causa de *MIO Throttle*. Algo similar ocurre con la instrucción *FFMA*, que realiza la multiplicación y suma de los valores. En este caso, la instrucción no puede ejecutarse debido a que las dependencias de datos no están resueltas. Es decir, se espera por las instrucciones de acceso a memoria compartida (*Short Scoreboard*). Una estrategia para alivianar el problema descrito es reducir los accesos a memoria compartida mediante la utilización de registros.

Una modificación posible del código que incorpora esta optimización se presenta en el bloque de código 3.2 y es incorporada en la variante 8 del algoritmo. En esta implementación, la cantidad de instrucciones de acceso a memoria compartida se reducen a aproximadamente la mitad con respecto a la variante 7. En cuanto a la cantidad de *stalls*, la cantidad promedio de ciclos en los que un warp no puede ejecutar por *MIO Throttle* se reduce en un 45 %, mientras que ciclos perdidos por *Short Scoreboard* se reducen en un 80 %. La reducción de stalls permite que el *scheduler* despache una instrucción cada 3,1 ciclos, una mejora del 16 % con respecto al valor inicial. Un posible riesgo de esta clase de optimizaciones es que el beneficio que se obtiene al utilizar una cantidad mayor de registros por thread, no sea suficientemente elevado en relación a la pérdida de performance ocasionada por la necesidad de acceder a memoria local. Este fenómeno se conoce como *Register Spilling*. En el caso del kernel de multiplicación utilizado, no se detectaron accesos a memoria local.

La variante 9, la cual incorpora la versión *shmem-prefetching* del kernel de multiplicación, tiene como objetivo mejorar el aprovechamiento del ancho de banda de memoria global y reducir stalls. En la versión *output-register* del kernel, al procesar una task, todos los threads de un bloque leen simultáneamente los mismos bitmaps y offsets de memoria global, desaprovechando la mayor parte del ancho de banda. A continuación se procede a discutir las características principales del comportamiento de ambos kernels según los datos recopilados por Nsight Compute. En primer lugar, dado que el hit rate del caché L2 es similar, la cantidad de datos que se transfieren de memoria del dispositivo a la caché L2 debería ser esencialmente la misma, ya que los accesos se hacen sobre los mismos datos. Esto puede corroborarse mediante la columna *L2<->FB Sectors* de cada reporte. Luego, el hecho de que en el kernel *shmem-prefetching* los threads acceden a distintos elementos de memoria global debería traducirse en que se ejecuten menos instrucciones para obtener la misma cantidad de datos, ya que en *output-register* por cada instrucción se utiliza un sólo dato, mientras que en *shmem-prefetching* se aprovechan múltiples datos. Lo anterior puede notarse en que *output-register* ejecuta aproximadamente 180M instrucciones de acceso a memoria global, mientras que *shmem-prefetching* ejecuta cerca de 85M. Por otro lado, como al momento de reconstruir las matrices los bitmaps pasan a leerse de memoria compartida, las instrucciones que involucran memoria compartida se incrementan en aproximadamente 50M. Una observación que puede parecer contraintuitiva es que en el kernel optimizado se transfiere un número ligeramente superior de paquetes de 32 bytes desde el caché L2 a L1 (*L2->TEX Returns*), pero aún así los tiempos de ejecución son menores. Esto se debe a que los patrones de

acceso a memoria global en *shmem-prefetching* ocasionan una menor cantidad de stalls, lo cual se ve reflejado en que los stalls de tipo Long Scoreboard se ven reducidos en un 25 %. La reducción de stalls hace que se pueda ejecutar una instrucción cada 2,6 ciclos, una mejora del 16 % con respecto a la medición previa.

La variante 10 cambia la forma en la que se reparte el trabajo a realizar. En versiones previas, un bloque de salida era calculado por un mismo bloque de CUDA, mientras que en esta versión, dicho cálculo le corresponde a un único warp. Una ventaja de este enfoque es que deja de ser necesario sincronizar los threads a nivel de bloque, ya que warps distintos trabajan sobre datos distintos. Este cambio se ve reflejado en el hecho de que en esta variante desaparecen los stalls de tipo *Barrier*. Otra ventaja es que permite que los elementos de la matriz B sean reutilizados, lo cual reduce significativamente los accesos a memoria compartida. Finalmente, una posibilidad que se abre al hacer que el trabajo se divida por warps y no por bloque es la de poder variar libremente el tamaño de bloque, algo que en versiones previas habría requerido cambios en el código del kernel. Sin embargo, la única motivación para hacer esto sería aumentar la Occupancy del kernel, la cual no está limitada por el tamaño de bloque, sino por la cantidad de registros utilizados por thread. Esto se debe a que el estado que debe mantener cada thread en la variante 10 es mayor al de las variantes anteriores, principalmente porque un thread pasa a calcular dos elementos de la matriz final por bloque, en lugar de uno. Para decidir la cantidad de registros que deben ser asignados a cada thread el compilador se basa en heurísticas, que pueden ser guiadas mediante parámetros proporcionados por el *qualifier* *__launch_bounds__()* en la declaración del kernel[6]. En este caso, el limite impuesto por los registros es de 12 bloques por SM. Utilizando el qualifier mencionado para imponer que la menor cantidad de bloques sea 16, se obtiene una mejora del 20 % en Achieved Occupancy, la cual se traduce en que cada scheduler pase de despachar una instrucción cada 2,2 ciclos a despachar una instrucción cada 1,8 ciclos. No se observó cambios en la cantidad de accesos a memoria local como consecuencia.

4.3.5 Optimización de T_5

La Tabla 4.11 detalla los tiempos de ejecución de la implementación de la variante 10, con el uso de __launch_bounds__(), como se describió en la sección anterior. En relación a los tiempos de la implementación base (Tabla 4.5), puede observarse que la etapa T_7 optimizada ejecuta entre 2 y 3 veces más rápido que la versión sin

optimizar. Por otro lado, la etapa más costosa deja de ser T_7 y pasa a ser T_5, la cual se encarga de ordenar la task list.

Tabla 4.11: Tiempos de ejecución (en µs) de la variante 10 de bmSPAPRSE. Se asume que las matrices de entrada están cargadas en memoria del dispositivo previamente.

Etapa	**Tiempo según matriz**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T_1	52	56	64	530	321	399	1281	1607	1791
T_2	41	42	42	1009	179	108	251	579	2189
T_3	213	554	336	1801	1668	1255	3819	2139	6951
T_4	41	94	58	615	500	401	1361	623	2327
T_5	170	1087	633	8951	7111	4207	14648	8005	33283
T_6	185	396	305	2104	1985	1388	3532	2419	5979
T_7	89	604	349	5858	6211	4376	12471	6564	22121
T_9	113	240	200	1024	1057	776	2026	1290	3632

Debido a detalles de implementación discutidos en el capítulo anterior, el functor utilizado al comparar elementos de la task list no accede directamente a los elementos sino que existe un nivel de indirección. Para cuantificar el impacto de lo anterior, la Tabla 4.12 detalla los tiempos de ejecución de las llamadas a thrust::sort que se harían si se ordenara la task list mediante accesos directos. Salvo en el primer caso, se observa una mejora del 12 % al 17 % en los tiempos. Sin embargo, modificar esta etapa impactaría sobre los tiempos de etapas posteriores, por lo que este cambio no necesariamente representa una mejora. En particular, si se representa la task list de manera directa (sin un nivel de indirección), sería necesario agregar una etapa adicional que se encargue de realizar una búsqueda en los arreglos de keys para obtener las posiciones. La razón es el kernel de multiplicación necesita que las tasks estén representadas de esta forma para acceder al arreglo de bitmaps y offsets de manera eficiente.

Tabla 4.12: Tiempo de ejecución (en µs) de una llamada a thrust::sort utilizando como el arreglo de entrada la task list sin un nivel de indirección adicional.

	Tiempo según matriz								
	1	2	3	4	5	6	7	8	9
thrust::sort	239	911	542	7826	6291	3685	12237	6951	28048
Std Dev	15	35	27	41	434	48	82	65	207

4.3.6 Comparación con cuSPARSE

La Tabla 4.13 compara los tiempos de ejecución de bmSPARSE y cuSPARSE. Se observa que bmSPARSE tiene un mejor desempeño en todas las matrices salvo en la primera, en donde existe una pequeña diferencia que favorece a cuSPARSE. La matriz en donde se presenta la mayor diferencia es la matriz 7, en donde el tiempo de ejecución de cuSPARSE es aproximadamente 12 veces mayor que el de bmSPARSE.

Tabla 4.13: Tiempos de ejecución (en µs) de la variante 10 de bmSPAPRSE y de cuSPARSE. En ambos casos se asumen valores de tipo float y matrices de entrada previamente cargadas en memoria del dispositivo.

	Tiempo según matriz								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Total bmSPARSE	908	3076	1992	21895	19035	12914	39393	23231	78276
Std Dev	43	27	26	150	786	39	218	788	1440
Total cuSPARSE	882	3998	5841	46555	140496	22113	455073	48310	202372
Std Dev	22	33	37	61	2998	56	10373	68	4556

La Tabla 4.14 muestra la misma comparación pero utilizando elementos de doble precisión. Dado que la única etapa que manipula los valores de la matriz es T_7, la diferencia entre los valores reportados por ambas tablas pueden atribuirse a dicha etapa. Una observación que puede realizarse es que, si bien bmSPARSE sigue teniendo un mejor desempeño en la mayoría de los casos, la penalización por utilizar doble precisión es significativamente mayor. Una posible explicación para este fenómeno involucra dos factores. El primero es la sobreutilización del pipeline FP64, encargado de ejecutar instrucciones sobre números de doble precisión. La razón por la cual es más probable que ocurra esto con la implementación de bmSPARSE que con la de cuSPARSE, es que bmSPARSE multiplica a nivel de bloques (mientras que cuSPARSE utiliza CSR), lo cual significa que la cantidad de multiplicaciones es mayor. Esta hipótesis puede corroborarse al observar el porcentaje de utilización del pipeline mencionado (95 %) y los stalls de tipo Tex Throttle que ocurren en las instrucciones SASS DFMA (*FP64 Fused Multiply Add*). El segundo causante es específico al hardware utilizado: el desempeño máximo medido en operaciones de punto flotante por segundo (FLOPs) cuando se trata de precisión simple es muy inferior al de doble precisión, como se observa en la Tabla 4.1. Es posible que la diferencia en tiempos observada aumente en una GPU mejor equipada con unidades FP64.

Tabla 4.14: Análoga a la Tabla 4.13, con la excepción de que asume matrices con valores de tipo double.

	Tiempo según matriz								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Total bmSPARSE	1102	4806	2899	37547	29039	19233	59946	35917	125884
Std Dev	33	93	464	161	1718	626	108	597	5702
Total cuSPARSE	955	4278	6385	49722	143179	22420	456142	49705	207235
Std Dev	32	29	66	46	2350	703	11176	43	4424

Capítulo 5

Multiplicación de bloques en Tensor Cores

En todas las implementaciones para el algoritmo de SpMM descrito en la Sección 3.2, al igual que en la implementación original[11], se utilizan CUDA Cores para realizar las operaciones aritméticas requeridas por la multiplicación de bloques (etapa T_7). En este capítulo se presentan distintas implementaciones de dicha etapa basadas en Tensor Cores, en lugar de CUDA Cores. La Sección 5.1 describe las distintas variantes consideradas. La Sección 5.2 realiza una evaluación experimental y compara el desempeño obtenido con el de las variantes previas. La Sección 5.3 realiza la misma comparación pero con cuSPARSE.

5.1 Implementación

Existen tres maneras de programar Tensor Cores: el API de alto nivel WMMA[6], las instrucciones *wmma* y *mma* del conjunto de instrucciones virtuales PTX[82] y las instrucciones *HMMA* en el lenguaje ensamblador nativo SASS[43]. A la fecha, la última vía no está documentada oficialmente y tampoco existen herramientas oficiales para traducir instrucciones SASS a código binario, por lo que los trabajos realizados son en base a ingeniería inversa y herramientas construidas por terceros. Una posible ventaja de utilizar instrucciones PTX en la arquitectura *Ampere* es que se brinda soporte para multiplicación de matrices dispersas[82]. Sin embargo, este no es el caso de la arquitectura en el que se realizó la implementación (*Turing*), por lo que se optó por la primera vía.

La interfaz WMMA admite un conjunto limitado de dimensiones para los bloques

de entrada y salida (los cuales son cargados en el tipo de datos *wmma::fragment*). Idealmente, se utilizarían fragmentos compatibles con el tamaño de bloque utilizado por el algoritmo (8×8), pero no hay soporte para fragmentos de dicho tamaño. Dentro de las opciones disponibles[6], no parece haber una ventaja entre una y otra en términos de performance, por lo que se optó por utilizar fragmentos de tamaño 16×16 por simplicidad de programación.

Una restricción que se impone al momento de utilizar Tensor Cores está relacionada al tipo de datos asociado a los valores de las matrices. No es posible multiplicar fragmentos que almacenen números de precisión simple en ninguna Compute Capability disponible a la fecha y el soporte para precisión doble fue introducido junto a la arquitectura Ampere (posterior a Turing). Por esta razón, se optó utilizar fragmentos de precisión media en las matrices y precisión simple en el fragmento de salida, una estrategia conocida como precisión mixta.

Las implementaciones realizadas, detalladas a continuación, toman como punto de partida la variante 10 del algoritmo de SpMM, descrita en el capítulo anterior, y adaptan el kernel de multiplicación para que utilice Tensor Cores. El resto de las etapas, junto a sus respectivas implementaciones, son las mismas:

Versión 1 (tensor-naive): En esta implementación *naive*, se utiliza el primer sub-bloque de 8×8 de cada fragmento y se ignora el resto, como se muestra en la Figura 5.1. A diferencia de la variante *warp-multiply*, en donde luego de cargar las matrices en memoria compartida cada thread acumula resultados parciales asociados a dos elementos de salida, en esta versión esa sección de código es reemplazada por dos llamadas a *wmma::load_matrix_sync* y una a *wmma::mma_sync*. Las primeras se encargan de cargar los fragmentos asociados a los bloques de entrada con los datos correspondientes, mientras que la segunda efectúa la multiplicación. Luego de procesar todas las tasks, se realiza una llamada a *wmma::store_matrix_sync* para transferir los resultados a memoria compartida y luego cargarlos en memoria global.

Notar que las funciones accesibles mediante la interfaz WMMA requieren la colaboración de todos los threads de un mismo warp, por lo que no es necesario asignarle a cada thread un elemento de salida a computar, como se hacía en implementaciones anteriores. Esta es una de las razones por las cuales se tomó como punto de partida la variante *warp-multiply* para realizar esta implementación, ya que es la única en la que los bloques de salida se calculan a nivel de warps (y no bloques de CUDA enteros).

Versión 2 (tensor-naive-padding): Antes de ejecutar la primitiva de multiplicación,

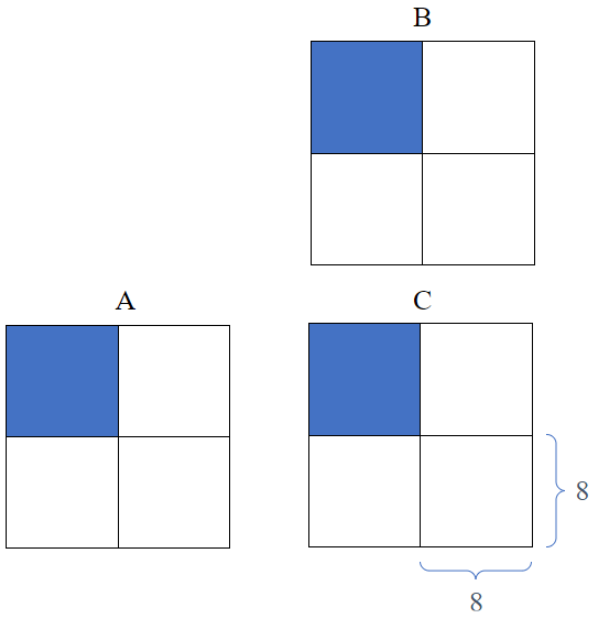


Figura 5.1: Fragmentos de tamaño 16×16 que están involucrados en la operación $C = A \times B + C$. Sólo se utiliza el primer sub-bloque de tamaño 8×8 de cada fragmento.

wmma::mma_sync, el warp debe cargar la matriz en memoria compartida mediante *wmma::load_matriz_sync*. En el hardware utilizado, es posible que elementos asignados a un mismo lane se encuentren en el mismo banco de memoria, ocasionando un conflicto de bancos. Por esta razón, se agregan bytes adicionales (*padding*) para que las lecturas se hagan sobre bancos distintos. El API requiere que el *stride* utilizado al cargar valores sea múltiplo de 8, por lo que el mínimo de elementos que deben agregarse por fila es 8 (cada fila pasa de tener 16 a 24 elementos).

Versión 3 (direct-frag-access): La única función expuesta por la interfaz WMMA para cargar los valores de un bloque en un fragmento obliga a que previamente el bloque esté almacenado en formato denso en memoria compartida o global[6]. En este caso, como los bloques están en un formato disperso, deben reconstruirse en forma densa en memoria compartida antes de llamar a *wmma::load_matrix_sync*, tal como se hacía en variantes previas. Dado que el tamaño de fragmento es mayor que el tamaño de bloque usado por el algoritmo, gran parte del ancho de banda de memoria compartida es malgastado en cargar valores que no son relevantes para el cálculo. Si bien el API permitie que cada thread acceda de manera directa a elementos del fragmento que tiene asignados, la principal dificultad para cargar los valores de manera más eficiente es que el mapeo entre elementos del fragmento y threads no está especificado y está sujeto a cambios en arquitecturas futuras[6]. La motivación

principal mencionada en la documentación de NVIDIA para brindar acceso directo al almacenamiento interno del fragmento es poder realizar operaciones que afectan a todo el fragmento de manera uniforme. Por ejemplo, si se desea multiplicar todos los valores por un mismo número, no es necesario conocer el mapeo de threads a elementos del fragmento ya que la misma operación es realizada sobre todos los elementos. Para implementar esta variante, se realizaron experimentos para determinar cómo se hace la asignación de elementos del fragmento a threads. Con este conocimiento, es posible cargar los valores sin necesidad de acceder a memoria compartida y sin desperdiciar ancho de banda. Es importante tener en cuenta que el mapeo utilizado en esta implementación es válido para la arquitectura Turing y no necesariamente para otras.

El procedimiento realizado para determinar el mapeo mencionado consiste en crear un fragmento y almacenar en cada elemento del mismo (mediante accesos directos) el identificador del thread que realizó la asignación. Posteriormente, se imprime el fragmento en la salida estándar. El resultado es una matriz de tamaño 16×16 (el tamaño de fragmento utilizado) que en la posición (i, j) tiene el identificador del thread que tiene asignado dicho elemento del fragmento. Un detalle que puede observarse al realizar este procedimiento es que el mapeo para el fragmento que almacena la matriz A (inicializado mediante el parámetro *wmma::matrix_a*) es el mismo que el mapeo para la matriz C (*wmma::accumulator*), pero no el mismo que el de la matriz B (*wmma::matrix_b*).

Una vez que se conocen las posiciones asignadas a un mismo thread, los valores pueden obtenerse directamente de memoria global, sin necesidad de cargar el bloque en formato denso en memoria compartida. Por esta razón, a diferencia de la variante tensor-naive, en esta variante no se hace uso de las funciones *wmma::load_matrix_sync* y *wmma::store_matrix_sync*.

Versión 4 (two-blocks): En las implementaciones con Tensor Cores previas, gran parte del fragmento es desperdiciado ya que sólo resulta útil una cuarta parte del mismo. Para atender ese problema, en esta variante cada warp procesa dos bloques en paralelo. Los bloques de entrada asociados al primer bloque de salida se cargan en el extremo superior izquierdo del fragmento correspondiente, mientras que el otro par de bloques es cargado en el extremo inferior derecho de cada fragmento, tal como muestra la Figura 5.2. En caso de que las tasks de un bloque de salida sean procesadas antes que las del otro, se rellenan con ceros las secciones de los fragmentos de entrada asociadas a dicho bloque, lo cual permite que se continúen acumulando los resultados parciales

vinculados al segundo bloque sin alterar los resultados del primero.

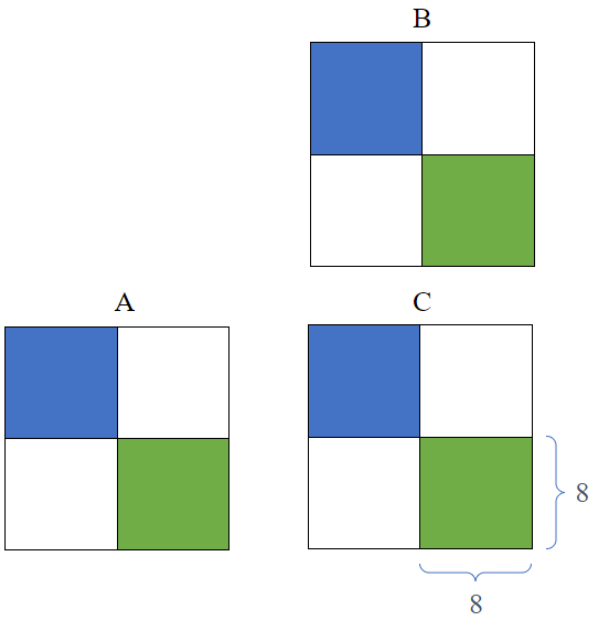


Figura 5.2: Fragmentos de tamaño 16×16 que están involucrados en la operación $C = A \times B + C$. Si dos sub-bloques de 8×8 en las matrices de entrada tienen el mismo color significa que contribuyen al resultado de un mismo sub-bloque del fragmento de salida.

Versión 5 (two-blocks-two-tasks): Esta variante es similar a la anterior, con la diferencia de que los warps no sólo procesan dos bloques en paralelo, sino que también procesan dos tasks en paralelo para cada bloque, como se muestra en la Figura 5.3. Esto permite que la totalidad de los fragmentos de entrada y la mitad de los fragmentos de salida sean utilizados. Los sub-bloques de color blanco del fragmento de salida en la figura mencionada no representan información útil, al igual que en implementaciones anteriores.

Notar que los sub-bloques de 8×8 contiguos (tanto vertical como horizontalmente) en un fragmento no necesariamente son contiguos en la matriz original. Este hecho no afecta la correctitud del algoritmo ya que los valores finales están determinados por qué pares de bloques de las matrices de entrada deben multiplicarse entre sí, y no la distribución de los mismos.

Versión 6 (hyb-tc-cuda): En esta implementación se combina la variante *warp-multiply*, en donde el cómputo de los resultados es realizado por CUDA Cores, con la variante *two-blocks-two-tasks*, en donde el cómputo es realizado por Tensor Cores. El primer warp de cada bloque de CUDA se encarga de procesar dos bloques de salida mediante

Tensor Cores, mientras que el segundo warp se encarga de procesar una cantidad configurable de bloques de salida mediate CUDA Cores.

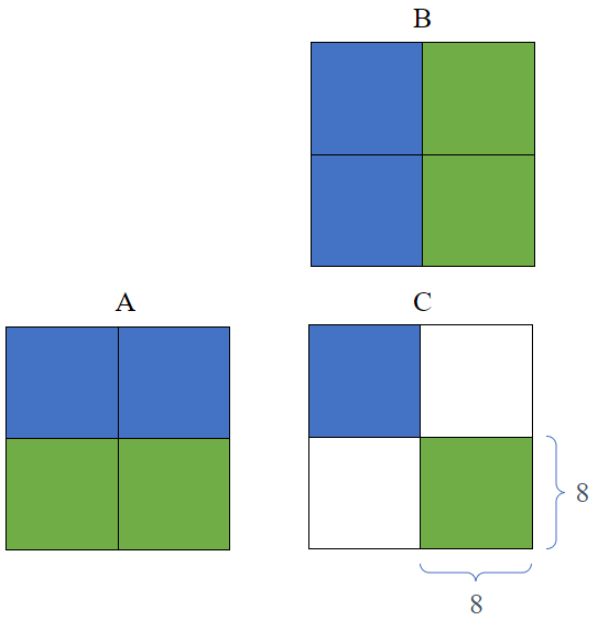


Figura 5.3: Estrategia simiar a la representada en la Figura 5.2, pero se con dos tasks asociadas a cada bloque de salida.

5.2 Evaluación experimental

Las pruebas realizadas se hacen sobre una CPU Intel i7-10750H@2.60GHz y una GPU NVIDIA GeForce RTX 2070, la cual corresponde a la arquitectura Turing. La programación en GPU se realiza mediante CUDA 11.2. El resto de las características del entorno son las mismas que en la Sección 4.1.

Tabla 5.1: Desempeño teórico en punto flotante para la GPU RTX 2070[83].

Precisión	Desempeño
FP16	14,93 TFLOPS
FP32	7,46 TFLOPS
FP64	233,3 TFLOPS

La Tabla 5.2 muestra los tiempos de las tres mejores variantes sin Tensor Cores. Se observa un patrón similar, con la excepción de que el uso de __launch_bounds__() en el kernel de la etapa T_7 de la variante 10 no representa una mejora de desempeño.

Esto se debe a que en esta GPU los registros no limitan la cantidad de warps por SM en la ejecución de dicho kernel. La Tabla 5.3 detalla los tiempos de ejecución de la etapa T_7 para las distintas variantes con Tensor Cores consideradas.

El kernel *tensor-naive* se caracteriza por ser el único kernel basado en Tensor Cores que utiliza memoria compartida para cargar las matrices de entrada y el resultado final (previo a la escritura en memoria global). Lo anterior es evidenciado por el hecho de que la cantidad de *requests* a memoria compartida es aproximadamente seis veces mayor a la del resto de las variantes consideradas. Casi la mitad de las instrucciones de acceso a memoria compartida resultan en conflictos de bancos, lo cual es una de las causas que explica la diferencia entre los tiempos de este kernel y los posteriores. Una forma de reducir el impacto de este problema es mediante la técnica de padding, utilizada en el kernel *tensor-naive-padding*. Esto representa un aumento significativo del uso de memoria compartida, lo cual disminuye el porcentaje de occupancy. Sin embargo, el beneficio que se obtiene de reducir los conflictos de bancos es mayor que la penalización en términos de occupancy.

Tabla 5.2: Tiempo promedio (en µs) de duración de la ejecución de la etapa T_7 y la totalidad del algoritmo en las tres mejores variantes sin Tensor Cores, pero ejecutadas en la GPU GeForce RTX 2070, en lugar de la GeForce GTX 1660 Ti. La segunda instancia de la variante 10 utiliza __launch_bounds(), mientras que la primera no lo hace.

Etapa	**Variante**	**Inst**	**Tiempo según matriz**								
			1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T_7	9	-	106	733	429	7107	7457	5625	15692	8327	28497
	10	**1**	**71**	**478**	**275**	**4554**	**4395**	**3327**	**9503**	**5020**	**17525**
	10	2	71	476	274	4527	4370	3356	9540	5057	17584
Total bmSPARSE	9	-	782	3006	1962	21482	18171	12946	38798	23627	76247
	10	1	745	2761	1814	18942	15192	10626	32587	20420	65236
	10	2	740	2755	1817	18890	15063	10645	32575	20205	65256
Std Dev	9	-	6	10	8	287	1201	63	298	585	487
	10	1	5	30	4	270	1162	17	269	568	184
	10	2	5	5	26	274	894	32	265	506	219

Tabla 5.3: Tiempo promedio (en µs) de duración de la ejecución de la etapa T_7 en las variantes tensor-naive, tensor-naive-padding, direct-frag-access, two-blocks y two-blocks-two-tasks del kernel.

Etapa	Kernel	Tiempo según matriz								
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T_7	tensor-naive	101	584	390	5715	8019	6892	17615	9108	29912
	tensor-naive-padding	89	531	350	5155	7138	5759	15033	7812	25842
	direct-frag-access	45	273	189	2664	3634	2847	7630	3901	13016
	two-blocks	**46**	**279**	**192**	**2685**	**3111**	**2375**	**6572**	**3334**	**11396**
	two-blocks-two-tasks	46	283	195	2734	3088	2425	6683	3374	11493
Std Dev	tensor-naive	1	1	0	22	624	26	2	14	417
	tensor-naive-padding	1	2	0	13	723	1	22	16	353
	direct-frag-access	0	1	0	3	225	4	4	10	24
	two-blocks	0	0	0	6	284	7	6	15	94
	two-blocks-two-tasks	0	1	1	15	300	11	69	7	94

El porcentaje de utilización tanto de los recursos de cómputo como los de memoria es del 52 % y 42 %, respectivamente. Kernels que exhiben este comportamiento típicamente son considerados *latency bound* (limitados por latencia). Otro indicador de que el kernel está limitado por la latencia es que schedulers logran despachar instrucciones cada 2,4 ciclos (cuando la arquitectura permite despachar una instrucción por ciclo). El análisis de los stalls más frecuentes muestra que los warps pasan un tiempo considerable en stalls de tipo *Long Scoreboard*, los cuales suceden cuando un warp no puede ejecutar una operación sobre L1TEX (que incluye memoria local y global, pero no compartida) a causa de una dependencia de datos[81]. El profiler indica que la mayor parte de estos stalls ocurren al momento de traer de memoria global los bitmaps y desplazamientos de las matrices de entrada. Un factor que también influye sobre la cantidad de stalls es que en este kernel los requerimientos de memoria compartida son mayores a los de las variantes previas, por lo que el porcentaje de occupancy es menor.

El reporte del kernel *direct-frag-access* muestra una reducción significativa en accesos a memoria compartida, lo cual se desprende del hecho de que no realiza llamadas a *wmma::load_matrix_sync*. Por otro lado, bajan los requerimientos de memoria compartida y uso de registros con respecto al kernel analizado previamente, lo cual se traduce en un porcentaje de occupancy mayor. Esto aumenta la frecuencia con la que se despachan instrucciones en casi un 17 %.

Mediante una inspección sobre el código SASS generado para los kernels *direct-frag-access* y *two-blocks* puede observarse que en el primer caso la llamada a *wmma::mma_sync*, la cual multiplica fragmentos a nivel de warp, se compila en dos instrucciones HMMA.1688.F32, mientras que en el segundo caso se utilizan cua-

tro. Si bien a la fecha no hay documentación oficial sobre estas instrucciones, Yan et al.[43] mostraron que, internamente, cada instrucción de este tipo contribuye a los valores de un bloque de 16×8 dentro del fragmento de 16×16. Adicionalmente, se necesitan dos instrucciones para calcular dicho bloque. En el kernel *direct-frag-access* sólo se accede al primer bloque de 16×8 del fragmento acumulador, lo cual probablemente es utilizado por el compilador como un indicador de que sólo es necesario generar las dos instrucciones HMMA.1688.F32 asociadas a dicho bloque. Para corroborar la afirmación anterior, puede observarse el SASS obtenido a partir de variaciones del kernel presentado en el Código 5.1. En dicho kernel, se realiza una multiplicación utilizando el API de WMMA y luego se accede de manera directa al almacenamiento interno del fragmento acumulador. Los experimentos realizados para obtener el mapeo de threads a elementos internos del fragmento indican que, para la arquitectura Turing, *pos* puede variar entre 0 y 7 y que $pos/2$ identifica a uno de los cuatro sub-bloques de 8×8 comprendidos en el fragmento. En todas las combinaciones posibles de lecturas sobre sub-bloques del fragmento acumulador, puede observarse que la granularidad con la cual el compilador genera instrucciones HMMA.1688.F32 es de dos instrucciones por cada bloque de 16×8 al que se accede. Esta hipótesis también es consistente con el resto de las implementaciones en base a Tensor Cores realizadas.

Algorítmo 5.1: Kernel utilizado para analizar el mecanismo mediante el cual se decide cuántas instrucciones HMMA.1688.F32 deben utilizarse por cada llamada a *wmma::mma_sync*

```
1  __global__
2  void test() {
3      wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::
           row_major> a_frag;
4      wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::
           row_major> b_frag;
5      wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;
6      wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
7      printf(" %f ", c_frag.x[pos1]); // pos1 entre 0 y 7
8      ...
9      printf(" %f ", c_frag.x[posn]); // posn entre 0 y 7
10 }
```

Como consecuencia de lo discutido previamente, los pipelines de Tensor Cores pasan de estar activos el 6, 8 % de los ciclos a el 9, 3 % de los ciclos. El bajo porcentaje,

en relación al porcentaje de utilización del pipeline LSU, se debe a dos razones. La primera es la baja intensidad computacional característica de la operación SpMM. La segunda es que el throughput de los Tensor Cores es mayor que el de otras unidades de cómputo, por lo que se requieren menos ciclos para obtener los resultados.

two-blocks tiene un desempeño ligeramente superior que *direct-frag-access*, lo cual se debe principalmente a que se reducen la cantidad de lecturas a memoria global, lo cual hacen que los stalls de tipo Long Scoreboard disminuyan. Ésta sigue siendo el principal causa de stalls, lo cual es esperable dada la baja intensidad computacional del kernel. Por último, algo que puede observarse es el aumento considerable de accesos a memoria local (Register Spilling). Esto se debe a que el estado que debe mantener cada warp es mayor, ya que debe llevar un registro de las tasks procesadas para dos bloques, en vez de uno. Sin embargo, la mayoría de estos accesos son de escritura, y los stalls son mayoritariamente ocasionado por lecturas.

El kernel *two-blocks-two-tasks* es similar a *two-blocks*, con la excepción de que se cargan dos tasks adicionales en los fragmentos de entrada. Esto ocasiona que los Tensor Cores sean utilizados con menos frecuencia, lo cual puede corroborarse notando que el porcentaje de ciclos en los que los pipelines de Tensor Cores están activos se reduce a la mitad. Si bien esto significa un mejor aprovechamiento de los recursos, los tiempos entre ambos kernels no presentan diferencias significativas. Esto se debe nuevamente a que el limitante principal no son los recursos de cómputo sino la latencia generada por accesos a memoria global.

El kernel *hyb-tc-cuda* se caracteriza por el hecho de que la mitad de los warps realizan la multiplicación mediante Tensor Cores y la otra mitad mediante CUDA Cores. La Tabla 5.4 muestra los tiempos de ejecución de tres instancias distintas del mismo, variando la cantidad de warps y registros utilizados por bloque, así como la proporción de warps que utilizan los Tensor Cores.

Tabla 5.4: Tiempo de ejecución (en µs) del tres instancias del kernel *hyb-tc-cuda*. la primera instancia lanza bloques de dos warps, uno utiliza Tensor Cores y el otro no. La segunda es idéntica pero se utiliza el qualifier __launch_bounds__() para reducir la cantidad de registros asignados a cada thread. Por último, la tercer variante, que también hace uso del qualifier mencionado, lanza bloques con tres warps, entre los cuales sólo el primero utiliza Tensor Cores.

Etapa	**Kernel**	**Inst**	**Tiempo según matriz**								
			1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T_7	hyb-tc-cuda	1	58	360	229	3384	4141	3386	9135	4666	15582
	hyb-tc-cuda	2	56	332	208	3082	3403	2782	7776	3960	13456
	hyb-tc-cuda	3	58	329	210	3076	3473	2745	7697	3926	13360
Std Dev	hyb-tc-cuda	1	0	1	0	8	167	18	33	25	83
	hyb-tc-cuda	2	0	4	0	2	256	1	5	8	102
	hyb-tc-cuda	3	0	5	1	2	339	1	3	1	13

El reporte obtenido del profiler indica que la cantidad de registros utilizados por thread en la primera instancia de *hyb-tc-cuda* es mayor al de los kernels previos, lo cual reduce el porcentaje de occupancy. Las instancias 2 y 3 del kernel, que tienen un desempeño muy similar, reducen el impacto de este problema mediante el uso del qualifier __launch_bounds__(). Esto aumenta tanto el porcentaje de occupancy como las instrucciones a memoria local. La combinación de estos factores resulta en una reducción de tiempo en todos los casos observados con respecto a la primera instancia, pero no con respecto a los kernels que se basan exclusivamente en Tensor Cores, en donde el desempeño es superior. En comparación con *two-blocks-two-tasks*, se observa un aumento en el uso de las unidades LSU. Esto se debe a que los warps que ejecutan las tasks mediante las unidades FMA utilizan operaciones de acceso a memoria compartida con mayor frecuencia.

5.3 Comparación con cuSPARSE

La Tabla 5.5 muestra los tiempos de ejecución de la mejor variante con Tensor Cores y la biblioteca cuSPARSE. Salvo en las primeras matrices, en donde el desempeño es similar, se observa que los tiempos de ejecución de bmSPARSE son menores. Por otro lado, se observa una mejora de desempeño con respecto a las variantes presentadas en la Tabla 5.2, que no utilizan Tensor Cores. Notar que a diferencia de las dichas variantes, en este caso no es posible comparar usando precisión simple o doble por limitaciones del hardware.

Tabla 5.5: Tiempo de ejecución (en µs) en la GPU NVIDIA GeForce RTX 2070 de la variante de SpMM que utiliza *two-blocks* en la etapa T_7.

	Tiempo según matriz								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
bmSPARSE	785	2522	1721	16938	13728	9562	29531	19461	59331
Std Dev	4	5	5	151	715	15	165	319	223
cuSPARSE	785	3148	4655	33216	144231	22505	447997	35584	149200
Std Dev	7	39	48	122	5024	31	15138	201	4038

Capítulo 6

Conclusiones y trabajo futuro

El trabajo realizado consistió en diseñar y evaluar distintas variantes del algoritmo para computar SpMM basado en el formato bmSPARSE. Las etapas del algoritmo son las mismas que las del algoritmo sugerido por Zhang y Gruenwald[11], con la excepción de la etapa T_4, que se agregó con el objetivo de remover de la task list las tasks que no afectan el resultado. Salvo la etapa T_7, todas las variantes se implementan mediante la biblioteca Thrust.

Los principales resultados obtenidos son los siguientes:

- El formato bmSPARSE requiere menos espacio de almacenamiento que CSR en todos los casos considerados.
- En la mayor parte de los casos de prueba, la operación SpMM basada en bmSPARSE tiene un mejor desempeño que cuSPARSE, tanto utilizando matrices compuestas por elementos de precisión simple como de doble precisión. En este último caso, la diferencia relativa es menor, aunque esto podría estar condicionado por el hardware en el que se realizaron las pruebas, el cual no es adecuado para trabajar en doble precisión.
- El beneficio de incorporar la etapa T_4 supera el costo asociado a su ejecución en todas las matrices, salvo en un caso (matriz 3) en donde no hay diferencia significativa. En otras palabras, resulta menos costoso descartar tasks a partir de operaciones bit a bit sobre los bitmaps de entrada que procesar dichas tasks en etapas posteriores.
- Las variantes que ejecutan la etapa T_9 tienen un mejor desempeño que las que ejecutan T_8. Por otro lado, contrario a lo observado en la implementación de

SpMM para bmSPARSE original[11], el cálculo simbólico (etapa T_9) es una de las etapas que requiere menos tiempo de ejecución. Una posible razón es que la implementación utilizada no se basa en adaptar el kernel de multiplicación (el cual utiliza memoria compartida), sino en operaciones bit a bit aplicadas sobre los bitmaps de las matrices de entrada, a través de primitivas de Thrust. Esto provoca que los accesos a memoria compartida sean sustituidos por accesos a registros. Adicionalmente, permite realizar la optimización asociada a la variante *2-nested-loops*, la cual es capaz de reducir los tiempos del kernel inicial hasta en un 85 %, como puede verse en la matriz 3 de la Tabla 4.9.

- Las implementaciones que utilizan Tensor Cores ejecutan en menor tiempo que las que no lo hacen. Hasta la arquitectura Turing inclusive, para aprovechar esta mejora se debe usar precisión media o mixta.
- La ejecución de la versión *naive* del kernel de multiplicación toma entre el doble y el triple del tiempo que la implementación *warp-multiply*. Esto se debe mayoritariamente a una serie de optimizaciones centradas en reemplazar el uso de memoria compartida por registros, reutilizar elementos leídos de memoria compartida y mejorar los patrones de acceso a memoria global.
- Luego de optimizar T_7, la etapa T_5 pasa a ser la etapa más costosa del algoritmo.

Algunas líneas de trabajo a futuro son las siguientes:

- Implementar la etapa T_5 mediante la operación *segmented sort*, lo cual permitiría explotar el hecho de que las tasks asociadas a una misma fila de la matriz de salida son contiguas en la task list previo a la etapa de ordenamiento. En otras palabras, puede obtenerse el mismo resultado final si se ordena cada sub-arreglo de tasks asociadas a una misma fila por separado.
- Realizar una implementación basada en Tensor Cores en una GPU con arquitectura Ampere, la cual provee soporte para fragmentos de menor tamaño que el tamaño de bloque utilizado por bmSPARSE[6]. Esto potencialmente podría significar un mejor aprovechamiento de los cores.
- Mejorar el uso de memoria del kernel basado en CUDA Cores. Una posible vía sería adaptar el kernel *10* para que exista una mayor reutilización de elementos.

- Evaluar el desempeño de las variantes desarrolladas en otras plataformas de hardware, en especial GPUs diseñadas para trabajar en doble precisión.
- Explorar alternativas que permitan minimizar la penalización que se obtiene al pasar de valores de precisión simple a doble precisión. Por ejemplo, mediante la reducción de operaciones que no influyen sobre el resultado, lo cual podría mitigar el problema de que el pipeline FP64 es sobreutilizado. Una posibilidad sería realizar una multiplicación únicamente cuando los bitmaps de entrada lo indican.
- Adaptar el algoritmo para que permita bloques de distinto tamaño. En particular, los bloques de tamaño 16×16 podrían ajustarse mejor al uso de Tensor Cores.
- Investigar el impacto que tienen las cargas desbalanceadas sobre el kernel de multiplicación de bloques y, en caso de ameritarlo, explorar formas de adaptar el algoritmo para que incorpore una estrategia de balanceo de carga.

Bibliografía

[1] G. Blake, R. G. Dreslinski y T. Mudge, "A survey of multicore processors", *IEEE Signal Processing Magazine*, vol. 26, n.º 6, págs. 26-37, 2009. DOI: 10.1109/MSP.2009.934110.

[2] M. Garland y D. B. Kirk, "Understanding Throughput-Oriented Architectures", *Commun. ACM*, vol. 53, n.º 11, págs. 58-66, nov. de 2010, ISSN: 0001-0782. DOI: 10.1145/1839676.1839694. dirección: https://doi.org/10.1145/1839676.1839694.

[3] *BLAS (Basic Linear Algebra Subprograms*, http://www.netlib.org/blas/, Fecha de acceso: 2021-21-02.

[4] V. Volkov y J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra", en *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, págs. 1-11. DOI: 10.1109/SC.2008.5214359.

[5] NVIDIA, *NVIDIA Tesla V100 GPU Architecture*, 2017. dirección: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[6] *CUDA Programming Guide*, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Fecha de acceso: 2021-21-02.

[7] *Cloud TPU — Google Cloud*, https://cloud.google.com/tpu, Fecha de acceso: 2021-21-02.

[8] J. Georgii y R. Westermann, "A Streaming Approach for Sparse Matrix Products and its Application in Galerkin Multigrid Methods", *Electronic Transactions on Numerical Analysis*, vol. 37, ene. de 2010.

[9] T. A. Davis, "Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and K-truss", en *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, págs. 1-6. DOI: 10.1109/HPEC.2018.8547538.

[10] J. R. Gilbert, S. Reinhardt y V. B. Shah, "High-Performance Graph Algorithms from Parallel Sparse Matrices", en *Applied Parallel Computing. State of the Art in Scientific Computing*, B. Kågström, E. Elmroth, J. Dongarra y J. Waśniewski, eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, págs. 260-269, ISBN: 978-3-540-75755-9.

[11] J. Zhang y L. Gruenwald, "Regularizing Irregularity: Bitmap-Based and Portable Sparse Matrix Multiplication for Graph Data on GPUs", en *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences Systems (GRADES) and Network Data Analytics (NDA)*, ép. GRADES-NDA '18, Houston, Texas: Association for Computing Machinery, 2018, ISBN: 9781450356954. DOI: 10.1145/3210259.3210263. dirección: https://doi.org/10.1145/3210259.3210263.

[12] R. Kannan, "Efficient sparse matrix multiple-vector multiplication using a bitmapped format", en *20th Annual International Conference on High Performance Computing*, 2013, págs. 286-294. DOI: 10.1109/HiPC.2013.6799135.

[13] A. Dakkak, C. Li, J. Xiong, I. Gelado y W.-m. Hwu, "Accelerating Reduction and Scan Using Tensor Core Units", en *Proceedings of the ACM International Conference on Supercomputing*, ép. ICS '19, Phoenix, Arizona: Association for Computing Machinery, 2019, págs. 46-57, ISBN: 9781450360791. DOI: 10.1145/3330345.3331057. dirección: https://doi.org/10.1145/3330345.3331057.

[14] J. L. Hennessy y D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017, ISBN: 0128119055.

[15] J. Shen y M. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, 2013, ISBN: 9781478610762. dirección: https://books.google.com.uy/books?id=ffQqAAAAQBAJ.

[16] A. Tanenbaum y T. Austin, *Structured Computer Organization*. Pearson, 2013, ISBN: 9780132916523.

[17] *Pentium Processor with MMX Technology*, http://datasheets.chipdb.org/Intel/x86/Pentium%20MMX/24318504.PDF, Fecha de acceso: 2021-21-02.

[18] T. Ungerer, B. Robič y J. Šilc, "A Survey of Processors with Explicit Multithreading", *ACM Comput. Surv.*, vol. 35, n.º 1, págs. 29-63, mar. de 2003, ISSN: 0360-0300. DOI: 10.1145/641865.641867. dirección: https://doi.org/10.1145/641865.641867.

[19] M. J. Flynn, "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*, vol. C-21, n.º 9, págs. 948-960, 1972. DOI: 10.1109/TC.1972.5009071.

[20] M. Flynn, "Flynn's Taxonomy", en *Encyclopedia of Parallel Computing*, D. Padua, ed. Boston, MA: Springer US, 2011, págs. 689-697, ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_2. dirección: https://doi.org/10.1007/978-0-387-09766-4_2.

[21] *MMX Instructions*, https://docs.oracle.com/cd/E18752_01/html/817-5477/eojdc.html, Fecha de acceso: 2021-21-02.

[22] M. McCool, J. Reinders y A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012, ISBN: 9780123914439.

[23] N. Adams, S. Kirby, P. Harris y D. B. Clegg, "A review of parallel processing for statistical computation", *Statistics and Computing*, vol. 6, págs. 37-49, 1996.

[24] F. Darema, "SPMD Computational Model", en *Encyclopedia of Parallel Computing*, D. Padua, ed. Boston, MA: Springer US, 2011, págs. 1933-1943, ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_26. dirección: https://doi.org/10.1007/978-0-387-09766-4_26.

[25] Chi-Chao Chang, G. Czajkowski, T. von Eicken y C. Kesselman, "Evaluating the Performance Limitations of MPMD Communication", en *SC '97: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, 1997, págs. 11-11. DOI: 10.1109/SC.1997.10040.

[26] D. B. Kirk y W.-m. W. Hwu, "Chapter 2 - History of GPU Computing", en *Programming Massively Parallel Processors (Second Edition)*, D. B. Kirk y W.-m. W. Hwu, eds., Second Edition, Boston: Morgan Kaufmann, 2013, págs. 23-39, ISBN: 978-0-12-415992-1. DOI: https://doi.org/10.1016/B978-0-12-415992-1.00002-X.

[27] J. Little, "OR FORUM—Little's Law as Viewed on Its 50th Anniversary", *Operations Research*, vol. 59, mayo de 2011. DOI: 10.1287/opre.1110.0940.

[28] R. Shoup, "SuperPaint: an early frame buffer graphics system", *IEEE Annals of the History of Computing*, vol. 23, n.º 2, págs. 32-37, 2001. DOI: 10.1109/85.929909.

[29] T. S. Crow, "Evolution of the graphical processing unit", *A professional paper submitted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science, University of Nevada, Reno*, 2004.

[30] T. Akenine-Möller, E. Haines y N. Hoffman, *Real-Time Rendering*. CRC Press, 2008, ISBN: 9781439865293.

[31] *The Cg Tutorial*, https://developer.download.nvidia.com/CgTutorial/cg_tutorial_chapter01.html, Fecha de acceso: 2021-21-02.

[32] A. R. Dennis, "An overview of rendering techniques", *Computers Graphics*, vol. 14, n.º 1, págs. 101-115, 1990, ISSN: 0097-8493. DOI: 10.1016/0097-8493(90)90014-O. dirección: http://www.sciencedirect.com/science/article/pii/0097849390900140.

[33] M. D. McCool, Z. Qin y T. S. Popa, "Shader Metaprogramming", en *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ép. HWWS '02, Saarbrucken, Germany: Eurographics Association, 2002, págs. 57-68, ISBN: 1581135807.

[34] E. Lindholm, J. Nickolls, S. Oberman y J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture", *IEEE Micro*, vol. 28, n.º 2, págs. 39-55, 2008. DOI: 10.1109/MM.2008.31.

[35] M. Khairy, A. G. Wassal y M. Zahran, "A survey of architectural approaches for improving GPGPU performance, programmability and heterogeneity", *Journal of Parallel and Distributed Computing*, vol. 127, págs. 65-88, 2019, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2018.11.012. dirección: http://www.sciencedirect.com/science/article/pii/S0743731518308669.

[36] NVIDIA, *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, Fecha de acceso: 2021-21-02, 2009.

[37] A. Ben Abdallah, "3 - Heterogeneous Computing: An Emerging Paradigm of Embedded Systems Design", en *Computational Frameworks*, M. K. Traoré, ed., Elsevier, 2017, págs. 61-93, ISBN: 978-1-78548-256-4. DOI: https://doi.org/10.1016/B978-1-78548-256-4.50003-X. dirección: http://www.sciencedirect.com/science/article/pii/B978178548256450003X.

[38] M. Zahran, *Heterogeneous Computing: Hardware and Software Perspectives*. New York, NY, USA: Association for Computing Machinery, 2019, ISBN: 9781450360975.

[39] NVIDIA, *NVIDIA Tesla P100*, http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper-v1.2.pdf, Fecha de acceso: 2021-21-02, 2017.

[40] *Using CUDA Warp-Level Primitives*, https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/, Fecha de acceso: 2021-21-02.

[41] *GPU Pro Tip: CUDA 7 Streams Simplify Concurrency*, https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/, Fecha de acceso: 2021-21-02.

[42] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng y J. S. Vetter, "NVIDIA Tensor Core Programmability, Performance Precision", en *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, págs. 522-531. DOI: 10.1109/IPDPSW.2018.00091.

[43] D. Yan, W. Wang y X. Chu, "Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply", en *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, págs. 634-643. DOI: 10.1109/IPDPS47924.2020.00071.

[44] J. J. Dongarra y V. Eijkhout, "Numerical linear algebra algorithms and software", *Journal of Computational and Applied Mathematics*, vol. 123, n.º 1, págs. 489-514, 2000, Numerical Analysis 2000. Vol. III: Linear Algebra, ISSN: 0377-0427. DOI: https://doi.org/10.1016/S0377-0427(00)00400-3. dirección: http://www.sciencedirect.com/science/article/pii/S0377042700004003.

[45] *Sparse Linear Algebra*, https://patterns.eecs.berkeley.edu/?page_id=202, Fecha de acceso: 2021-21-02.

[46] J. Davis y E. Chung, "SpMV: A Memory-Bound Application on the GPU Stuck Between a Rock and a Hard Place", inf. téc. MSR-TR-2012-95, sep. de 2012. dirección: https://www.microsoft.com/en-us/research/publication/spmv-a-memory-bound-application-on-the-gpu-stuck-between-a-rock-and-a-hard-place/.

[47] Z. Gu, J. Moreira, D. Edelsohn y A. Azad, "Bandwidth Optimized Parallel Algorithms for Sparse Matrix-Matrix Multiplication Using Propagation Blocking", en *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, ép. SPAA '20, Virtual Event, USA: Association for Computing Machinery, 2020, págs. 293-303, ISBN: 9781450369350. DOI: 10.1145/3350755.3400216. dirección: https://doi.org/10.1145/3350755.3400216.

[48] P. Kogge et al., "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems", *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Techinal Representative*, vol. 15, ene. de 2008.

[49] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Second. Society for Industrial y Applied Mathematics, 2003. DOI: 10.1137/1.9780898718003. dirección: https://epubs.siam.org/doi/abs/10.1137/1.9780898718003.

[50] R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial y Applied Mathematics, 1994. DOI: 10.1137/1.9781611971538. dirección: https://epubs.siam.org/doi/abs/10.1137/1.9781611971538.

[51] A. Pinar y M. T. Heath, "Improving Performance of Sparse Matrix-Vector Multiplication", en *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, ép. SC '99, Portland, Oregon, USA: Association for Computing Machinery, 1999, 30-es, ISBN: 1581130910. DOI: 10.1145/331532.331562. dirección: https://doi.org/10.1145/331532.331562.

[52] Y. Saad, *SPARSKIT: a basic tool kit for sparse matrix computations - Version 2*, 1994.

[53] D. Langr y P. Tvrdík, "Evaluation Criteria for Sparse Matrix Storage Formats", *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, n.º 2, págs. 428-440, 2016. DOI: 10.1109/TPDS.2015.2401575.

[54] F. G. Gustavson, "Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition", *ACM Trans. Math. Softw.*, vol. 4, n.º 3, págs. 250-269, sep. de 1978, ISSN: 0098-3500. DOI: 10.1145/355791.355796. dirección: https://doi.org/10.1145/355791.355796.

[55] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick y J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms", en *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007, págs. 1-12. DOI: 10.1145/1362622.1362674.

[56] *cuSPARSE Library*, https://developer.nvidia.com/cusparse, Fecha de acceso: 2021-21-02.

[57] *Math Kernel Library*, https://software.intel.com/en-us/mkl, Fecha de acceso: 2021-21-02.

[58] M. Deveci, C. Trott y S. Rajamanickam, "Performance-portable sparse matrix-matrix multiplication for many-core architectures", en *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, págs. 693-702. DOI: 10.1109/IPDPSW.2017.8.

[59] G. Ballard, A. Buluc, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz y S. Toledo, "Communication Optimal Parallel Multiplication of Sparse Random Matrices", en *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ép. SPAA '13, Montréal, Québec, Canada: Association for Computing Machinery, 2013, págs. 222-231, ISBN: 9781450315722. DOI: 10.1145/2486159.2486196. dirección: https://doi.org/10.1145/2486159.2486196.

[60] M. Deveci, S. Hammond, M. Wolf y S. Rajamanickam, "Sparse Matrix-Matrix Multiplication on Multilevel Memory Architectures : Algorithms and Experiments", *ArXiv*, vol. abs/1804.00695, 2018.

[61] A. Sodani, "Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor", en *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015, págs. 1-24. DOI: 10.1109/HOTCHIPS.2015.7477467.

[62] A. Buluç y J. Gilbert, "Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments", *SIAM Journal on Scientific Computing*, vol. abs/1109.3739, 2012.

[63] N. Bell, S. Dalton y L. Olson, "Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods", *SIAM Journal on Scientific Computing*, vol. 34, ene. de 2012. DOI: 10.1137/110838844.

[64] S. Dalton, L. Olson y N. Bell, "Optimizing Sparse Matrix—Matrix Multiplication for the GPU", *ACM Trans. Math. Softw.*, vol. 41, n.º 4, oct. de 2015, ISSN: 0098-3500. DOI: 10.1145/2699470. dirección: https://doi.org/10.1145/2699470.

[65] W. Liu y B. Vinter, "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data", en *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, págs. 370-381. DOI: 10.1109/IPDPS.2014.47.

[66] Y. Nagasaka, A. Nukada y S. Matsuoka, "High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU", en *2017 46th International Conference on Parallel Processing (ICPP)*, 2017, págs. 101-110. DOI: 10.1109/ICPP.2017.19.

[67] M. Deveci, C. Trott y S. Rajamanickam, "Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures", *Parallel Computing*, vol. 78, págs. 33-46, 2018, ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2018.06.009. dirección: http://www.sciencedirect.com/science/article/pii/S0167819118301923.

[68] M. M. A. Patwary et al., "Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms", en *High Performance Computing*, J. M. Kunkel y T. Ludwig, eds., Cham: Springer International Publishing, 2015, págs. 48-57, ISBN: 978-3-319-20119-1.

[69] J. Liu, X. He, W. Liu y G. Tan, "Register-Aware Optimizations for Parallel Sparse Matrix—Matrix Multiplication", *Int. J. Parallel Program.*, vol. 47, n.º 3, págs. 403-417, jun. de 2019, ISSN: 0885-7458. DOI: 10.1007/s10766-018-0604-8. dirección: https://doi.org/10.1007/s10766-018-0604-8.

[70] P. N. Q. Anh, R. Fan e Y. Wen, "Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication", en *Proceedings of the 2016 International Conference on Supercomputing*, ép. ICS '16, Istanbul, Turkey: Association for Computing Machinery, 2016, ISBN: 9781450343619. DOI: 10.1145/2925426.2926273. dirección: https://doi.org/10.1145/2925426.2926273.

[71] N. Satish, M. Harris y M. Garland, "Designing efficient sorting algorithms for manycore GPUs", en *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, págs. 1-10. DOI: 10.1109/IPDPS.2009.5161005.

[72] D. Merrill y A. Grimshaw, "High Performance and Scalable Radix Sorting: a Case Study of Implementing Dynamic Parallelism for GPU Computing", *Parallel Process. Lett.*, vol. 21, págs. 245-272, 2011.

[73] F. Gremse, A. Höfter, L. O. Schwen, F. Kiessling y U. Naumann, "GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging", *SIAM Journal on Scientific Computing*, vol. 37, n.º 1, págs. C54-C71, 2015. DOI: 10.1137/130948811. eprint: https://doi.org/10.1137/130948811. dirección: https://doi.org/10.1137/130948811.

[74] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel y M. Steinberger, "Adaptive Sparse Matrix-Matrix Multiplication on the GPU", en *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ép. PPoPP '19, Washington, District of Columbia: Association for Computing Machinery, 2019, págs. 68-81, ISBN: 9781450362252. DOI: 10.1145/3293883.3295701. dirección: https://doi.org/10.1145/3293883.3295701.

[75] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo y S. Williams, "Exploiting Multiple Levels of Parallelism in Sparse Matrix-Matrix Multiplication", *SIAM Journal on Scientific Computing*, vol. 38, n.º 6, págs. C624-C651, 2016. DOI: 10.1137/15M104253X. eprint: https://doi.org/10.1137/15M104253X. dirección: https://doi.org/10.1137/15M104253X.

[76] *Thrust: Algorithms*, http://thrust.github.io/doc/group__algorithms.html, Fecha de acceso: 2021-21-02.

[77] H. Warren, *Hacker's Delight*. Addison-Wesley, 2003, pág. 111, ISBN: 9780201914658.

[78] *Volta Tuning Guide*, https://docs.nvidia.com/cuda/volta-tuning-guide, Fecha de acceso: 2021-21-02.

[79] *NVIDIA GeForce GTX 1660 Ti Specs*, https://www.techpowerup.com/gpu-specs/geforce-gtx-1660-ti.c3364, Fecha de acceso: 2021-21-02.

[80] T. A. Davis e Y. Hu, "The University of Florida Sparse Matrix Collection", *ACM Trans. Math. Softw.*, vol. 38, n.º 1, dic. de 2011, ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. dirección: https://doi.org/10.1145/2049662.2049663.

[81] *Kernel Profiling Guide*, https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html, Fecha de acceso: 2021-21-02.

[82] *PTX Documentation*, https://docs.nvidia.com/cuda/parallel-thread-execution/index.html, Fecha de acceso: 2021-21-02.

[83] *NVIDIA GeForce RTX 2070 Specs*, https://www.techpowerup.com/gpu-specs/geforce-rtx-2070.c3252, Fecha de acceso: 2021-21-02.