UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Uso de formatos de almacenamiento y algoritmos a bloques en álgebra dispersa en dispositivos masivamente paralelos

Gonzalo Berger Álvarez

Programa de Posgrado en Ingeniería Informática
Facultad de Ingeniería
Universidad de la República

Montevideo – Uruguay
Mayo de 2024

UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Uso de formatos de almacenamiento y algoritmos a bloques en álgebra dispersa en dispositivos masivamente paralelos

Gonzalo Berger Álvarez

Tesis de Maestría presentada al Programa de Posgrado en Ingeniería Informática, Facultad de Ingeniería de la Universidad de la República, como parte de los requisitos necesarios para la obtención del título de Magister en Ingeniería Informática.

Director de tesis:
Dr. Ing. Ernesto Dufrechou

Codirector:
Dr. Ing. Pablo Ezzatti

Director académico:
Dr. Ing. Ernesto Dufrechou

Montevideo – Uruguay
Mayo de 2024

Berger Álvarez, Gonzalo

Uso de formatos de almacenamiento y algoritmos a bloques en álgebra dispersa en dispositivos masivamente paralelos / Gonzalo Berger Álvarez. - Montevideo: Universidad de la República, Facultad de Ingeniería, 2024.

XII, 112 p.: il.; 29, 7cm.

Director de tesis:

Ernesto Dufrechou

Codirector:

Pablo Ezzatti

Director académico:

Ernesto Dufrechou

Tesis de Maestría – Universidad de la República, Programa de Ingeniería Informática, 2024.

Referencias bibliográficas: p. 87 – 98.

1. Álgebra dispersa, 2. Almacenamiento a bloques, 3. Computación de alta performance, 4. CUDA, 5. GPU. I. Dufrechou, Ernesto *et al.* II. Universidad de la República, Programa de Posgrado en Ingeniería Informática. III. Título.

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE TESIS

Dr. Enzo Rucci

Dr. Javier Baliosian

Dr. Adrián Pérez Dieguez

Montevideo – Uruguay
Mayo de 2024

RESUMEN

La notoria evolución de disciplinas como la ciencia de datos y el aprendizaje automático en los últimos años ha despertado un gran interés en operaciones del álgebra lineal dispersa tales como la multiplicación de matrices dispersas generales (SpGeMM). Esta operación anteriormente no gozaban de la misma atención dedicada como por ejemplo a la multiplicación de matriz dispersa-vector (SpMV) por parte de la comunidad científica. Consecuentemente, hoy día son comunes los esfuerzos de investigación volcados al desarrollo eficiente de ambas rutinas en plataformas paralelas. La mayoría de las operaciones sobre matrices dispersas son caracterizadas por implicar una cantidad baja de cómputo en relación a los accesos a memoria, algo que dificulta explotar la gran capacidad de cómputo que tienen dispositivos masivamente paralelos como las GPUs. Estos accesos, además, son altamente irregulares, dado que dependen de la distribución de los valores no nulos y, en el caso de SpGeMM, dependen también de como se vinculen los elementos no nulos de las dos matrices de entrada. Para mitigar esta irregularidad, una posibilidad es el uso de formatos a bloques para el almacenamiento de las matrices dispersas.

En este trabajo se explora el uso de formatos de almacenamiento a bloques para las operaciones SpGeMM y SpMV. En este sentido, se profundiza en el formato *bmSparse*, que permite potenciales ahorros importantes tanto en espacio de almacenamiento como en accesos a memoria. En primera instancia, se busca atacar distintos cuellos de botella de una implementación base de la operación SpGeMM a través de distintas propuestas, que incluyen cambios en la representación del formato, mejoras en el paso de ordenamiento y el uso de Tensor Cores para la multiplicación de bloques. Por otro lado, se implementa una rutina para la operación SpMV con el fin de explorar el potencial del formato más allá del producto de matrices dispersas. Los resultados obtenidos al evaluar estas dos implementaciones en matrices de distintos tamaños de la *Suite Sparse Matrix Collection* muestran importantes mejoras del tiempo de ejecución para varias matrices. En el caso de SpGeMM, la implementación propuesta obtiene resultados considerablemente mejores que las variantes para CSR presentes en bibliotecas conocidas como cuSPARSE de NVIDIA y MKL

de Intel, y es competitiva con la implementación para BCSR de esta última. En el caso de SpMV, se alcanzan speedups de más de 4× en algunas matrices en comparación con la implementación de cuSPARSE para CSR, lo que sugiere que el formato es una alternativa interesante para distintas aplicaciones que involucran el uso de matrices dispersas.

Palabras claves:
Álgebra dispersa, Almacenamiento a bloques, Computación de alta performance, CUDA, GPU.

Lista de figuras

Lista de tablas

Tabla de contenidos

Capítulo 1

Introducción

Desde la aparición del primer microprocesador [15], el siempre creciente mercado ha ido aumentando la demanda de chips cada vez más rápidos [103]. En 1965, Moore [81] predijo que la cantidad de transistores de los chips se duplicarían cada 12 meses[1]. Si bien con el correr de los años este período se ha relajado a unos 18 o 24 meses [89], la tendencia exponencial se ha mantenido. Esto ha llevado a un incremento consistente de la frecuencia de reloj de los chips, al punto de que el consumo energético comenzó a crecer a un ritmo que opacaba las ganancias de velocidad obtenidas por una mayor frecuencia [25]. Por esta razón, los fabricantes de chips dejaron de incrementar la frecuencia de reloj de forma significativa [69] y comenzaron la búsqueda por una alternativa que permita seguir incrementando la capacidad de cómputo. Así, la tendencia se ha movido hacia la creación de plataformas con diseño orientado al paralelismo [54], en lugar de seguir llevando al límite la capacidad de procesadores de un único núcleo. En este contexto es que los procesadores paralelos comienzan a despertar interés. Almasi y Gottlieb [13] definen los procesadores paralelos como "una gran colección de unidades de procesamiento que pueden cooperar y comunicarse entre sí para resolver problemas de gran escala rápidamente". Un ejemplo son los procesadores orientados al *throughput* [52], como las GPUs (del inglés *Graphics Processing Units*), que se han convertido en plataformas atractivas para el cómputo científico debido a su moderado costo [88].

Muchas aplicaciones científicas, incluyendo el conocido algoritmo *Page Rank* de Google [27], se componen de operaciones del álgebra lineal. En 1973, Hanson et al. [60] proponen la creación de un conjunto de subprogramas que

[1] Concepto comúnmente conocido como Ley de Moore.

implementan algunas de estas operaciones y luego, en 1978, se presenta una implementación de 38 de estas operaciones bajo el nombre de BLAS (*Basic Linear Algebra Subprograms*). Posteriormente, se realizaron extensiones para incluir operaciones matriz-vector [43] y operaciones matriz-matriz [41], definidas en estos trabajos como *Level 2 BLAS* y *Level 3 BLAS*, respectivamente (siendo el trabajo original *Level 1 BLAS*). Existen implementaciones para la mayoría de las plataformas paralelas relevantes. En el caso de las GPUs se tiene a la biblioteca cuBLAS [2] de NVIDIA, mientras que para CPUs existen OpenBLAS [8] y MKL (Intel)[5][2]. En general, las implementaciones de estas bibliotecas son maduras y obtienen desempeños cercanos a su pico teórico.

Por el lado del álgebra lineal dispersa, es decir aquella donde las matrices presentan una gran cantidad de sus entradas iguales a cero, existe la biblioteca cuSPARSE [4], originalmente basada en una variante dispersa BLAS denominada *Sparse BLAS* [45]. Actualmente, cuSPARSE utiliza otro estándar, aunque mantiene implementaciones legadas de Sparse BLAS [4]. Esta especificación surge debido a que las variantes densas de BLAS ofrecían un soporte parcial para matrices dispersas. Las matrices dispersas son parte fundamental de innumerables problemas de cómputo eficiente e ingenieril. Ejemplos de esto son la resolución de ecuaciones diferenciales parciales mediante el Método de Elementos Finitos (o *FEM* por sus siglas en inglés) [96], aplicaciones de grafos [79] [56] y algunos problemas de optimización [57]. Además permiten resolver problemas de gran escala donde el uso de matrices densas no es factible debido a los enormes requerimientos de espacio de almacenamiento que ello implicaría, ya que es posible explotar la gran proporción de valores nulos a través de formatos de almacenamiento y rutinas que permiten trabajar con ellos de forma eficiente.

1.1. Motivación

La principal limitante a la hora de implementar operaciones dispersas se da en los accesos a memoria. Al procesar los elementos no nulos de la matriz, dichos accesos típicamente suceden en posiciones lejanas entre sí, potencialmente incrementando la tasa de *cache miss*. Además, los problemas dispersos realizan una baja cantidad de cómputo en comparación a los accesos a me-

[2]MKL permite delegar operaciones para que se ejecuten en GPUs [6]

moria considerando que, para poder ubicar los valores no nulos dentro de la matriz, es necesario almacenar y acceder a valores enteros que no forman parte del cómputo. Esto es particularmente problemático para las GPUs, ya que dificulta el aprovechamiento de su alta capacidad de procesamiento.

Una estrategia para abordar este obstáculo es el diseño de formatos de almacenamiento que permitan regularizar los accesos a memoria, o incluso disminuir su cantidad. Dentro de estos formatos se encuentran los formatos a bloques, donde se agrupan distintos elementos de la matriz en bloques, que pueden ser densos o dispersos. Esta distribución de los elementos, permite una indexación a nivel de bloque, en lugar de almacenar los índices de cada valor no nulo. De esta forma, no solamente se obtienen beneficios en cuanto al espacio de almacenamiento, sino que se reduce la cantidad necesaria de accesos a memoria, dado que no es necesario recuperar valores de índices para cada elemento de la matriz.

En el caso en que los bloques en los que se divide la matriz sean densos, aparece el inconveniente de que se estarán almacenando valores nulos de forma explícita. Esto no sucede si los bloques se almacenan de forma dispersa, pero de esta manera se pierde el orden relativo de los elementos dentro del bloque, por lo que es necesario un segundo nivel de indexación.

Una posibilidad es asociar un mapa de bits a cada bloque, donde cada bit representa un elemento y es 0 si este es nulo, y 1 en otro caso. Si bien se vuelve a mantener un índice por elemento, como son relativos al bloque solamente es necesario 1 bit por cada uno de ellos, evitando usar uno o más enteros.

El uso de un mapa de bits, además, puede resultar beneficioso en otros aspectos de las operaciones dispersas. Por ejemplo, en la multiplicación matriz-matriz, estas estructuras permiten calcular características de la matriz resultado utilizando valores binarios en lugar de los valores de entrada reales.

1.2. Objetivos y metodología de investigación

Este trabajo busca alcanzar el estado del arte en problemas dispersos usando formatos o estrategias a bloques. Para esto, se realiza un relevamiento completo de la literatura relacionada, para luego profundizar sobre *bmSparse*, un formato a bloques basado en indexación con mapas de bits. El interés por este tipo de formatos en el contexto de la investigación nace de la hipótesis de que sus ventajas en cuanto al uso de la memoria permitirán un mayor aprovecha-

miento de los recursos de cómputo de las GPU. Si bien el formato *bmSparse* ha sido estudiado anteriormente, en este trabajo se aportan optimizaciones que mejoran notablemente su rendimiento en la operación SpGeMM y se explora por primera vez su uso en la operación SpMV.

Para alcanzar este objetivo, en primer lugar, se realizará un relevamiento completo y sistemático de la literatura referente a la resolución de problemas de álgebra lineal dispersa en GPUs y, en particular, sobre el uso de formatos dispersos a bloques.

Teniendo en cuenta este relevamiento, se profundizará en algunas de las técnicas y se estudiará de qué manera puede ser aplicadas a distintas operaciones, como por ejemplo el producto matriz dispersa-vector (SpMV) o el producto de matrices dispersas (spGEMM).

En la siguiente etapa, para aquellas operaciones elegidas que cuenten con implementaciones existentes se propondrán modificaciones con el fin de mejorar su desempeño en GPUs modernas. En caso de no poder acceder a la implementación, se intentará reproducir los resultados de investigaciones previas desarrollando una implementación base a partir de la literatura relacionada.

Esta etapa es seguida por una evaluación experimental de las implementaciones, y el refinamiento iterativo del prototipo basado en los resultados experimentales.

Por último, se propondrán nuevas rutinas para operaciones dispersas que no hayan sido abordadas aún utilizando los formatos a bloques de la etapa anterior.

1.3. Organización del documento

La estructura del documento se organiza de la siguiente manera. El Capítulo 2 introduce nociones básicas relacionadas al paralelismo, GPUs y matrices dispersas. Se describe de forma detallada el estado del arte en cuanto a formatos de almacenamiento disperso, con énfasis en los formatos a bloques y el formato *bmSparse*, que es el foco central del trabajo. También se presentan las operaciones SpGeMM y SpMV y un relevamiento de trabajos relacionados. El Capítulo 3 presenta las propuestas realizadas para la operación SpGeMM, entre las que se encuentran modificaciones a la indexación del formato y a la etapa de ordenamiento de la rutina. También se adaptan las implementaciones previas [78] de la etapa de multiplicación con Tensor Cores a una nueva

arquitectura de GPU. El Capítulo 4 describe dos implementaciones para la operación SpMV utilizando el formato *bmSparse*. Por último, el Capítulo 5 resume los resultados obtenidos y detalla distintas líneas de trabajo futuro.

Capítulo 2

Marco Referencial

En este capítulo se definen algunos conceptos esenciales de computación paralela, GPUs y matrices dispersas. La Sección 2.1 describe algunas de las técnicas usuales de paralelismo, así como la arquitectura y la metodología de programación sobre GPUs. La Sección 2.2 presenta el concepto de matrices dispersas y detalla algunos de los formatos de almacenamiento más comunes. Por último, la Sección 2.3 incluye un listado de distintos formatos de almacenamiento a bloques de matrices dispersas, poniendo especial énfasis en el formato *bmSparse*.

2.1. Computación paralela y procesadores gráficos

Como se mencionó en el Capítulo 1, la continua necesidad por una mayor capacidad de cómputo en combinación con la limitante de poder seguir aumentando la frecuencia de reloj de los chips de manera significativa derivó en la necesidad de realizar cómputo paralelo. Una de las primeras estrategias es la de *pipelining*, que consiste en dividir una instrucción en n etapas, donde cada una de ellas es ejecutada en distintas unidades de hardware [100]. Este mecanismo permite la ejecución de distintas instrucciones en paralelo, dado que evita la necesidad de esperar el término de la ejecución de una instrucción para comenzar a ejecutar otra. Al terminar una etapa, la unidad de hardware ejecuta la misma etapa pero de otra instrucción. Esto es un ejemplo de lo que se conoce como paralelismo a nivel de instrucciones (*instruction level parallelism* o ILP por sus siglas). Este tipo de paralelismo consiste en ejecutar múltiples

instrucciones en paralelo, aunque el sistema ejecute un programa secuencial [48], por lo que es crucial encontrar un conjunto de instrucciones que pueda ejecutarse de forma independiente. Bernstein [24] plantea tres condiciones para que dos bloques de instrucciones secuenciales, S_1 y S_2, puedan ejecutarse en cualquier orden sin alterar el resultado:

$$L_1 \cap E_2 = \emptyset \tag{2.1}$$

$$E_1 \cap L_2 = \emptyset \tag{2.2}$$

$$E_1 \cap E_2 = \emptyset \tag{2.3}$$

donde L_i y E_i son el conjunto de direcciones de memoria que el bloque S_i accede para lectura y escritura, respectivamente. El incumplimiento de una o más de estas ecuaciones implica una cierta dependencia entre dos bloques de instrucciones. Bacon et al. [18] definen dos tipos de dependencias entre bloques de instrucciones. La dependencia de control ocurre cuando un bloque de instrucciones S_1 determina si el bloque S_2 debe ser ejecutado. Por ejemplo, el cuerpo de un if no puede ejecutarse previo a la evaluación de la condición. Por otro lado, la dependencia de datos sucede cuando dos bloques acceden al mismo fragmento de memoria, y al menos uno de esos accesos es una escritura. Este tipo de dependencia se divide en tres casos:

- Dependencia de flujo (*flow dependence* o *true dependence*), que ocurre cuando un bloque S_1 genera un resultado utilizado por otro bloque S_2. Si S_2 lee la memoria antes de que S_1 realice la escritura, el valor que leerá será un valor desactualizado, lo que es conocido como *Read After Write* (RAW).
- Antidependencia, que se da cuando un bloque S_1 escribe en una dirección de memoria que es leída por S_2 (S_2 tiene una antidependencia con S_1). Si S_2 lee la memoria luego de que S_1 realice la escritura, leerá un valor modificado en lugar del que esperaba, lo que es conocido como *Write After Read* (WAR).
- Dependencia de salida, que se da cuando dos bloques S_1 y S_2 escriben en la misma dirección de memoria. Si las escrituras se hacen en el orden incorrecto, en la dirección de memoria quedará almacenado un valor viejo, lo que es conocido como *Write After Write* (WAW).

Si consideramos que dos bloques de instrucciones dependientes entre sí (o que incumplan las ecuaciones 2.1, 2.2 o 2.3) no pueden ser ejecutados en paralelo, puede que no sea posible encontrar un conjunto de instrucciones independientes lo suficientemente grande para realmente aprovechar el paralelismo a nivel de instrucción. Existen múltiples estrategias que permiten aumentar el número de instrucciones independientes. *Loop unrolling* [77] consiste en replicar el cuerpo de una iteración en distintas instrucciones. *Register renaming* se utiliza para eliminar *hazards* WAR y WAW [61], asignando distintos nombres a distintos usos de la misma variable [72]. La ejecución fuera de orden o ejecución especulativa [100] consiste en adelantar la ejecución de instrucciones antes de saber si es necesaria su ejecución, siempre y cuando pueda revertirse cualquier cambio que genere. Además, la correcta sincronización de las lecturas y escrituras permitiría la ejecución paralela de dos bloques dependientes, pero los fragmentos de código que acceden a recursos compartidos (regiones críticas) deben seguir siendo ejecutados de forma secuencial.

En contraste con el paralelismo a nivel de instrucción, el paralelismo a nivel de datos sucede cuando se ejecuta una misma secuencia de instrucciones sobre distintos conjuntos de datos. En conjunto con el paralelismo a nivel de instrucción forman las bases de una metodología para clasificar distintas formas de paralelismo propuesta por Flynn [50], apropiadamente denominada Taxonomía de Flynn. Esta taxonomía clasifica las técnicas de paralelismo dependiendo de dos factores: la secuencia de instrucciones que ejecuta el procesador, definido como *Instruction Stream*, y la secuencia de datos sobre la que actúan estas instrucciones, denominado *Data Stream*. A partir de estos dos conceptos se derivan cuatro categorías que describen las arquitecturas paralelas:

1. **SISD (Single Instruction - Single Data Stream)**

   Esta categoría incluye los procesadores que ejecutan instrucciones de manera secuencial sobre un mismo stream de datos. De esta forma, parecería que no explota paralelismo a nivel de instrucción ni a nivel de datos. Sin embargo, estos sistemas pueden hacer uso de paralelismo a nivel de instrucción, como el *pipelining* o los procesadores superescalares [30], que pueden ejecutar más de una instrucción por ciclo de reloj dado que contienen múltiples unidades funcionales [100].

2. **SIMD (Single Instruction - Multiple Data Stream)**

   Al igual que SISD, la arquitectura SIMD maneja una única secuencia de instrucciones, pero esta se replica sobre distintos conjuntos de datos. Ejemplos de esto son los procesadores de arreglos, que consiste de múltiples procesadores con su propia memoria local, y los procesadores vectoriales, que son procesadores de un solo núcleo con unidades funcionales diseñadas específicamente para el manejo de vectores [49].

3. **MISD (Multiple Instruction - Single Data Stream)**

   La arquitectura MISD ejecuta múltiples instrucciones sobre el mismo stream de datos. No existen ejemplos generalmente aceptados de sistemas que utilicen esta arquitectura. Algunos autores [49] consideran las GPUs como ejemplo de un dispositivo que utiliza una arquitectura MISD, pero otros consideran que no hay ejemplos claros [100] [61].

4. **MIMD (Multiple Instruction - Multiple Data Stream)**

   La arquitectura MIMD consiste de sistemas multiprocesadores con alguna forma de interconexión. En estos sistemas se realiza paralelización tanto a nivel de instrucción como a nivel de datos. Si bien existen sistemas MIMD heterogéneos, usualmente todas las unidades de proceso suelen ser iguales [49]. MIMD frece una mayor flexibilidad que SIMD, pero a su vez resulta más costoso [61].

Una forma de combinar el paralelismo de datos y de instrucción es el paralelismo a nivel de hilos (*thread level parallelism*), que explota el paralelismo a nivel de datos a través de múltiples hilos cooperando en paralelo [61], y donde cada hilo o *thread* es una secuentia de instrucciones y datos ejecutada usando ILP.

Una forma de implementar el paralelismo a nivel de hilos es a través de la ejecución de múltiples hilos en paralelo dentro del mismo multiprocesador, para lo que es necesario duplicar todo el procesador. Multithreading es una técnica que evita esto haciendo que cada hilo mantenga una copia distinta del archivo de registros y del contador de programa (*program counter*), solamente

duplicando el estado de cada thread [90].

Existen diversas variantes de multithreading. *Fine-grained multithreading* ejecuta los hilos de forma intercalada, alternando su ejecución en cada ciclo de reloj de manera *round robin*. Este intercalado permite que si un hilo se encuentra parado por una dependencia (en un *stall*), se ejecuten instrucciones de otros hilos, y quizá para cuando el hilo retome su ejecución ya no se encuentre en stall. Una segunda variante, denominada *coarse-grained multithreading*, solamente detiene la ejecución de un hilo ante un *stall*. Una desventaja de esta variante es que, como solamente se ejecutan instrucciones de un hilo a la vez, al alternar la ejecución es necesario vaciar el pipeline y que el nuevo hilo lo vuelva a poblar [90]. Puede valer la pena pagar por este overhead en caso de que los stalls sean duraderos, pero si estos son cortos es probable que termine siendo una limitante. La última variante, denominada *simultaneous multithreading* puede verse como una extensión de coarse-grained multithreading para procesadores superescalares [100], en la que un mismo hilo puede ejecutar más de una instrucción por ciclo de reloj, y el hilo que se encuentra ejecutando se cambia ante un stall.

Las GPUs son dispositivos masivamente paralelos que reúnen varios de los tipos de paralelismo presentados a lo largo de esta sección: paralelismo a nivel de instrucción, multithreading, SIMD y MIMD [61]. En 2009, con el lanzamiento de la arquitectura de GPUs Fermi [85], NVIDIA acuña el término SIMT (*Single Instruction Multiple Threads*, para describir la arquitectura de estos dispositivos. La influencia de la industria de los videojuegos ha impulsado a incorporar enorme cantidad de unidades de cómputo capaces de ejecutar hasta miles de hilos de forma simultánea. Esta considerable capacidad de cómputo, disponible a un precio muy inferior que el de las plataformas convencionales de HPC, generó el interés de utilizarlas para un procesamiento de propósito general, es decir, no relacionado con la computación gráfica. Sin embargo, dado que en sus inicios estaban diseñadas específicamente para la presentación de gráficos, su programación para otros propósitos implicaba el mapeo de los cómputos a una API del pipeline gráfico [70][73] y por lo tanto una gran curva de aprendizaje para la programación sobre estos dispositivos, lo cual reducía su uso a cómputos realizados por expertos.

2.1.1. CUDA

En noviembre de 2006, NVIDIA anunció la GeForce 8800 [74], una tarjeta basada en una entonces nueva arquitectura denominada G80, y CUDA (del inglés *Compute Unified Device Architecture*), que permite ejecutar las etapas del pipeline gráfico en un mismo tipo de procesador, lo que da lugar a una programación más sencilla sobre GPUs a través de una API y distintas bibliotecas. Mediante la programación con CUDA, las GPUs adquirieron la capacidad de sustituir a la CPU en operaciones de alto nivel de paralelismo e intensidad computacional, como algunas operaciones matriciales. De esta forma, la CPU podía delegar el proceso de las porciones de aplicación computacionalmente intensas a la GPU a través de programas llamados *kernels*. Las características soportadas por cada GPU de NVIDIA se definen como *Compute Capability*. Las Compute Capabilities se representan con un número $X.Y$, donde X indica la arquitectura base del dispositivo e Y representa una iteración de ella. Al compilar aplicaciones con CUDA es importante especificar la Compute Capability correcta del dispositivo, dado que con esto las aplicaciones en tiempo de ejecución determinan las tecnologías disponibles en la GPU sobre la que ejecutan.

A. Modelo de programación

En el modelo de programación de CUDA, la ejecución de los programas (kernels) es dividida en hilos (*threads*), que a su vez se organizan en bloques (*thread blocks*). Estos bloques pueden tener hasta tres dimensiones. Los hilos de un bloque tienen un identificador, que representa el número del hilo dentro del bloque. Para un bloque de tamaño (S_x, S_y, S_z), el identificador del hilo en la posición (x, y, z) es igual a $z \cdot S_x \cdot S_y + y \cdot S_x + x$. Los hilos dentro de un mismo bloque pueden cooperar de manera sencilla a través de la memoria compartida y primitivas de sincronización.

De manera similar, los bloques también son agrupados en una grilla (*grid*). La existencia de la grilla se debe a que la cantidad de hilos que puede tener cada bloque es acotada, por lo que si se desea una mayor cantidad de hilos para la ejecución de un kernel, se deben organizar en múltiples bloques. Esto, sin embargo, significa que dos hilos que se encuentren en bloques distintos no puedan cooperar entre sí de forma sencilla, dado que tienen distinta memoria compartida y las primitivas de sincronización actúan de forma individual

sobre cada bloque[1]. Todos los bloques dentro de una grilla tienen las mismas dimensiones. Los bloques tienen un identificador, que representa el número del bloque dentro de la grilla. La grilla puede tener hasta tres dimensiones. En el caso de una grilla de tamaño (S_x, S_y, S_z), el identificador del bloque en la posición (x, y, z) se calcula como $z \cdot S_x \cdot S_y + y \cdot S_x + x$. En la Figura 2.1 se puede ver un esquema de ejemplo con grillas y bloques de dos dimensiones.

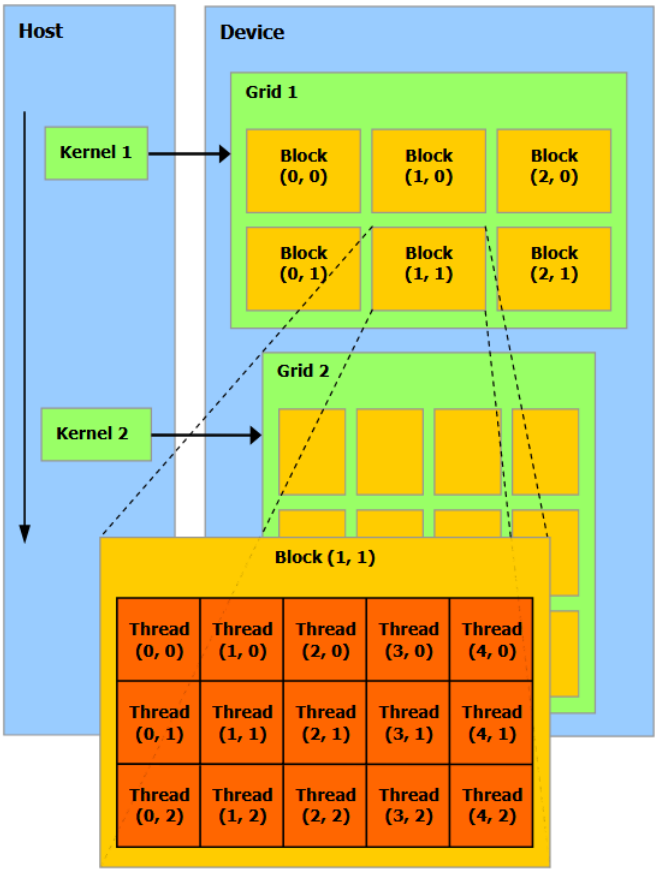

Figura 2.1: Esquema de ejecución de dos kernels. Cada kernel se ejecuta como una grilla de bloques de hilos.

En la extensión de C++ de CUDA, la definición de un kernel se realiza mediante la directiva `__global__`. Las funciones declaradas con esta directiva son ejecutadas en GPU (device) y solamente pueden ser invocadas desde el host. Las invocaciones a este tipo de funciones deben especificar la configuración para la ejecución a través de una expresión de la forma $\lll GridDims, BlockDims, Ns \ggg$ entre el nombre de la función y la lis-

[1]Esto cambia con la introducción de los *Thread Block Clusters* en la *Compute Capability 9.0*.

ta de parámetros. *GridDims* es una variable de tipo *dim3*[2] que contiene las dimensiones de la grilla, $(GridDims.x, GridDims.y, GridDims.z)$, donde $GridDims.x \times GridDims.y \times GridDims.z$ es igual al número de bloques con el que se lanza el kernel. *BlockDims*, también de tipo *dim3*, contiene las dimensiones de los bloques, $(BlockDims.x, BlockDims.y, BlockDims.z)$, donde $BlockDims.x \cdot BlockDims.y \cdot BlockDims.z$ es igual al número de hilos que contiene cada bloque.

Algoritmo 2.1: Ejemplo de declaración e invocación de un kernel.

```
1   __global__ void VecAdd(float* A, float* B, float* C){
2       int i = threadIdx.x;
3       C[i] = A[i] + B[i];
4   }

7   int main(){
8       ...
9       VecAdd<<<1, N>>>(A, B, C);
10      ...
11  }
```

En el ejemplo del Algoritmo 2.1 se declara un kernel que computa la suma de dos vectores A y B y acumula el resultado en un vector C. El kernel es invocado con una grilla de un único bloque (indicado por el parámetro 1) unidimensional de tamaño N (indicado por el parámtero N). Dentro del kernel se tiene acceso a la variable *threadIdx*, de tipo *uint3*, que contiene el índice del hilo dentro del bloque. Como el bloque es unidimensional, el número del hilo, i, es simplemente el valor del primer campo de *threadIdx*, *threadIdx.x*. Luego, cada hilo realiza una única operación de adición en la posición calculada.

En caso de que el bloque tuviera más de una dimensión, el cálculo del número de hilo también depende de los otros campos de la variable, *threadIdx.y* y *threadIdx.z* (este último solo para bloques tridimensionales). En el caso de los bloques, dentro del kernel se tiene acceso a las variables *blockIdx* y *blockDim*. La primera contiene el índice del bloque dentro de la grilla y la segunda contiene las dimensiones del bloque. En cuanto a la grilla, existe la variable *gridDim*, que contiene las dimensiones de la grilla.

A diferencia de las CPUs, las GPUs dedican considerablemente más transistores al procesamiento de datos que al manejo de cachés y al control de flujos. La arquitectura de GPUs de NVIDIA se centra en el concepto de *Streaming Multiprocessor* (*SM*), multiprocesadores paralelos diseñados para ejecu-

[2]Vector entero basado en *uint3* para especificar dimensiones. Las dimensiones omitidas se establecen en 1.

tar cientos de hilos en simultáneo, organizados en un arreglo escalable. Cada SM contiene una cantidad de núcleos que depende de la generación de la tarjeta. Al ejecutar un kernel, cada multiprocesador se encarga de la ejecución de un grupo de bloques de la grilla. Los bloques asignados a un multiprocesador pueden ejecutarse concurrentemente dentro de los mismos. Al terminar la ejecución de los bloques, en caso de haber más bloques a procesar, se asignan a aquellos multiprocesadores que hayan quedado disponibles. Los multiprocesadores dividen, planifican y ejecutan los hilos de estos bloques en grupos de 32 hilos denominados *warps*. Los hilos de cada warp poseen un identificador incremental, siendo el primer hilo del primer warp el hilo con identificador 0. De esta forma, el primer warp contiene los índices de 0 a 31, el segundo de 32 a 63, y así sucesivamente. Los warps ejecutan una única instrucción a la vez. Si bien dos hilos del mismo warp pueden diverger y ejecutar distintas instrucciones, estas se realizan de forma serial, es decir, los hilos que estén ejecutando la instrucción actual son llamados hilos *activos*, mientras que aquellos que se encuentren ociosos son denominados hilos *inactivos*. Los hilos también pueden ser inactivos si ya han terminado su ejecución o si se encuentran en un warp que no utiliza los 32 hilos (por ejemplo, warps de un bloque con una cantidad de hilos que no es múltiplo de 32). La máxima eficiencia es entonces alcanzada cuando todos los hilos ejecutan la misma secuencia de instrucciones, aunque se haga sobre un conjunto distinto de datos. A partir de la arquitectura Volta, se introduce el concepto de *Independent Thread Scheduling*, que permite que distintos hilos del mismo warp ejecuten instrucciones distintas de forma concurrente. Para ello, la GPU mantiene el estado de ejcución de cada hilo, y un planificador determina como distribuir los distintos grupos de hilos activos de cada warp en las unidades de cómputo.

La memoria de los dispositivos se divide de forma jerárquica, tal como se muestra en la Figura 2.2. En un primer nivel, se tiene una memoria global. Esta es una memoria *off-chip*, por lo que tiene una alta latencia. Es compartida por todos los kernels que se encuentren ejecutando en la GPU. Para cada kernel, todos los hilos de la grilla tienen acceso a esta memoria. CUDA ofrece funciones para el manejo de esta memoria tales como *cudaMalloc* y *cudaFree* para reservar y liberar memoria, respectivamente, y *cudaMemcpy* para transferencias de datos entre el host y el dispositivo (también host a host y dispositivo a dispositivo).

Cuando múltiples hilos de un warp acceden a esta memoria, el dispositivo

combina (*coalesce*) los accesos en una o más transacciones de memoria dependiendo de la distribución de las direcciones de memoria accedidas[3]. Si todos los hilos del warp acceden a posiciones consecutivas que se encuentran dentro del mismo bloque de memoria se consigue un acceso *coalesced*, donde los accesos pueden ser satisfechos con una sola transacción de memoria. En este sentido, también es importante almacenar las estructuras de forma alineada en memoria global[4]. Que una estructura de tamaño 2^N se encuentre alineada quiere decir que la primera posición en donde se encuentra en la memoria es múltiplo de 2^N [35]. En caso de que los hilos del warp necesiten acceder a posiciones de memoria distanciadas entre sí (conocido como *strided memory access*), una posibilidad es copiar los fragmentos de memoria global a memoria compartida, en la que no hay penalización para este tipo de accesos [3]. En dispositivos con Compute Capability 5.0 en adelante, los accesos a memoria global son siempre almacenados en una memoria caché L2, lo que mitiga la penalización de los accesos no coalesced [3].

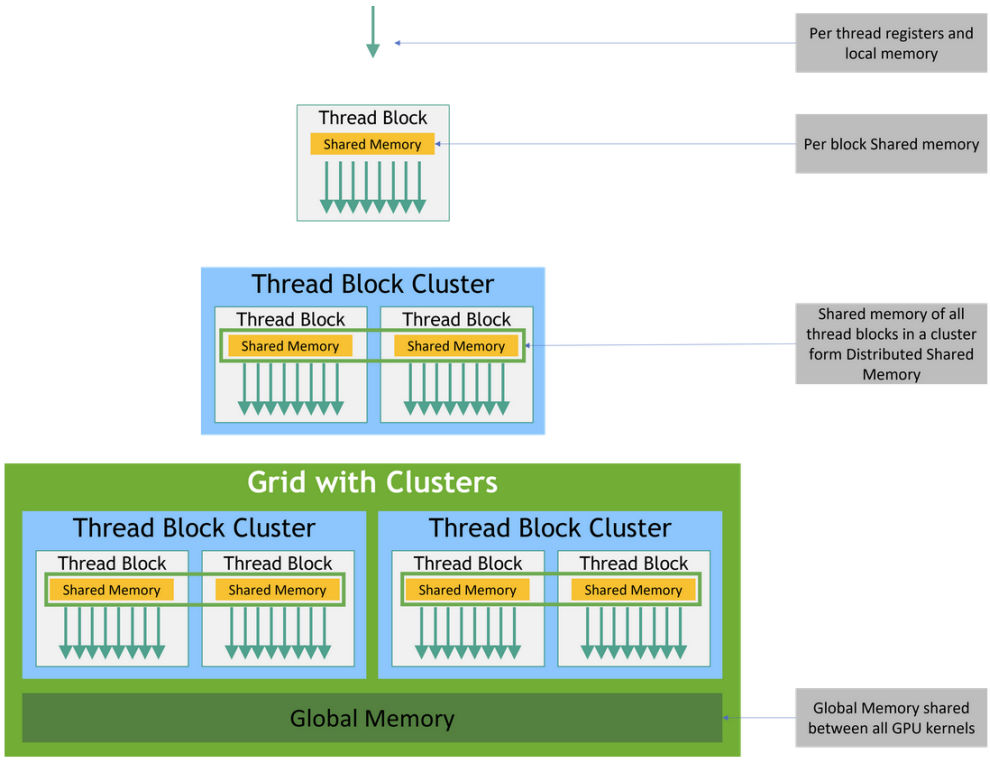

Figura 2.2: Jerarqúia de memoria en CUDA. Extraída de [3].

[3]También depende del tamaño de la palabra almacenada en memoria.

[4]Los vectores de tipo nativos se alinean automáticamente [3]. Para otras estructuras existe la función *align*.

En el mismo nivel que la memoria global se tiene una memoria constante y una memoria de texturas. Estas memorias son de solo lectura para el dispositivo y solo pueden ser escritas por el host.

Cada SM tiene una memoria denominada memoria compartida. Esta es una memoria *on-chip* con una latencia mucho menor y un ancho de banda mucho mayor que el de la memoria global. Los bloques que ejecuten dentro de cada multiprocesador utilizan un fragmento de esta memoria. Con la introducción de los *Thread Block Clusters*, un bloque puede realizar lecturas y escrituras en el fragmento de memoria compartida asignada a los otros bloques dentro del mismo cluster. Esta memoria es dividida en módulos de igual tamaño llamados *bancos de memoria*, que pueden ser accedidos a la vez. Esto implica en una mejora importante del ancho de banda, dado que n accesos a memoria que caigan en n bancos distintos se satisfacen en simultáneo. En contraparte, dos accesos al mismo banco de memoria se satisfacen de forma serial. A esto se le llama *conflicto de bancos*. A partir de la Compute Capability 5.0, dos hilos de un warp que accedan al mismo banco de memoria no generan un conflicto de bancos, siempre y cuando accedan a la misma palabra de 32 bits [3]. Para lecturas, se realiza un *broadcast* del valor sobre los hilos correspondientes. El caso de las escrituras sigue siendo problemático, dado que solamente uno de los hilos (no está definido cual) escribe sobre la memoria, lo cual puede generar un comportamiento no esperado en la ejecución.

En un último nivel, los multiprocesadores contienen un archivo de registros. Estos registros son asignados entre los hilos que se encuentren ejecutando en el multiprocesador.

La evaluación experimental de este trabajo fue realizada utilizando dos modelos de tarjetas gráficas, la RTX 2080 Ti de la arquitectura Turing y la RTX 3090 Ti de la arquitectura Ampere. En el Anexo A se describen estas arquitecturas.

2.1.2. Thrust

Thrust es una biblioteca de CUDA basada en el estándar *Standard Template Library* (STL) [10]. Introducida en 2009 [62], Thrust provee una interfaz de alto nivel para la programación en GPUs que permite el uso de operaciones como ordenamiento, transformaciones y reducciones, entre otras, aceleradas con dichos dispositivos. En el contexto de este trabajo se utiliza Thrust para

acelerar de forma sencilla el procesamiento fuera de los kernels desarrollados. Además, el algoritmo presentado en la Sección 2.4.4 sigue una idea similar a la formulación ESC [21] [34], que utiliza primitivas como *scan*, *gather* y *scatter*, de las cuales Thrust ofrece implementaciones. Las operaciones utilizadas de esta biblioteca son presentadas en el Anexo B.

2.1.3. Tensor Cores

Con el lanzamiento de la arquitectura Volta [84], NVIDIA introdujo una nueva unidad de procesamiento denominada *Tensor Cores*, motivado por las fuertes demandas generadas por las redes neuronales. Cada Tensor Cores de primera generación permite realizar 64 operaciones *fma*[5] por ciclo de reloj. Estas unidades realizan operaciones del tipo $D = A + C$, donde A, B, C y D son matrices de 4×4. Por esto, los Tensor Cores resultan de especial interés para acelerar la obtención de resultados intermedios en el producto de matrices (tanto GeMM como SpGeMM). Si bien existen limitantes en cuanto a la precisión de los valores almacenados (A y B solamente mantienen valores de precisión mixta, FP16), nuevas generaciones de Tensor Cores agregan soporte para más precisiones. En CUDA, la programación sobre Tensor Cores se hace a través de la API de operaciones a nivel de warp, *wmma*, que contiene operaciones para cargar, multiplicar y acumular matrices utilizando estas unidades.

2.2. Matrices dispersas

Una matriz es llamada dispersa cuando sus valores son mayoritariamente nulos. Los valores se dicen nulos dado que no representan información útil. El valor nulo suele ser el 0, aunque esto puede variar dependiendo del campo de aplicación. Esta definición presenta un cierto grado de ambigüedad, no explicita qué significa que los valores sean mayoritariamente nulos. Un posible criterio para la cantidad de valores no nulos de una matriz dispersa es $O(n)$, con entre 2 a 10 valores no nulos por fila de la matriz [26]. J. H. Wilkinson define una matriz dispersa como "una matriz con una cantidad de ceros lo suficientemente

[5]Del inglés *fused multiply-add*. Operación que computa el producto de dos números y los suma a un acumulador. Se realiza en un único paso de redondeo al final, por lo que permite evitar pérdidas de precisión en la adición.

grande para poder ser aprovechada" [105][6]. Si bien esta definición sigue siendo igual de ambigua que la anterior, ayuda a ilustrar un rasgo importante de este tipo de matrices, que es la explotación de la dispersidad.

En este sentido, es fundamental la utilización de formatos adecuados para el almacenamiento de las matrices dispersas. Las matrices densas se suelen almacenar en arreglos de $n \times m$ elementos, siendo n la cantidad de filas y m la cantidad de columnas. El orden de los elementos puede ser por filas (*row major order*) o por columnas (*column major order*). Las ventajas principales de este formato es el acceso en $O(1)$ a todos los elementos de la matriz, así como el aprovechamiento de la localidad espacial de los datos, dado que dos valores contiguos de una fila (si es *row major*) o de una columna (si es *column major*) en la matriz, también están contiguos en la memoria. Esta idea no es adecuada para las matrices dispersas, dado que se está almacenando de forma explícita los valores nulos. Lo que se busca entonces con los formatos de almacenamiento de matrices dispersas es evitar el almacenamiento de los valores nulos. De esta manera, el espacio de almacenamiento requerido para las estructuras es proporcional a la cantidad de valores no nulos de la matriz, y menor al de una matriz densa de mismas dimensiones.

El formato *COO* (del inglés *COOrdinate Format*) es posiblemente la estrategia más intuitiva para almacenar una matriz dispersa [95]. La misma consiste en almacenar únicamente los elementos no nulos de la matriz, así como los índices de fila y de columna de los mismos. Entonces el formato es conformado por tres arreglos: *vals*, que contiene los valores no nulos de la matriz, *row_ids*, que contiene los índices de fila, y *col_ids* que contiene los índices de columna. Un elemento de la matriz está entonces representado por la tripleta *(vals[i],row_ids[i],col_ids[i])*, donde $i \in [1, nnz]$. En la Figura 2.3 se puede ver un esquema de ejemplo.

Es claro que con este formato se hace un uso mucho más adecuado de la memoria que si se almacenara con el formato típico utilizado para matrices densas. Considerando valores de punto flotante de 8 bytes e índices enteros de 4 bytes (*int*), el arreglo de $n \times n$ utiliza $8n^2$ ($O(n^2)$) bytes, mientras que con el formato COO se utilizan $16 \times nnz$ bytes ($O(n)$ si se toma el valor de nnz como $O(n)$).

[6]En verdad, la definición dada por Wilkinson es para matrices densas: *"We shall refer to a matrix as dense if the percentage of zero elements or its distribution is such as to make it uneconomic to take advantage of their presence."* La definición dada de matriz dispersa está dada por la negativa de esta sentencia.

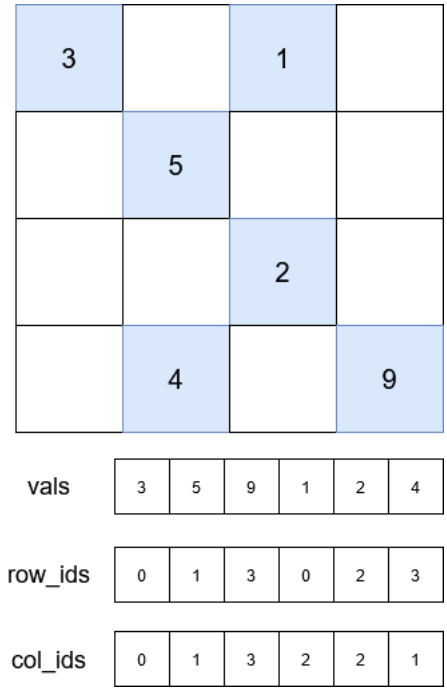

Figura 2.3: Matriz representada en formato COO.

En comparación a otros formatos de almacenamiento de matrices dispersas, COO utiliza estructuras que poseen el mismo tamaño. Otra ventaja es que es conceptualmente sencillo impactar cambios en la matriz, simplemente se agregan elementos con sus respectivos índices de fila y columna. En la práctica esto dependerá de las estructuras elegidas para representar la matriz. Será sencillo impactar estos cambios si por ejemplo se utilizan listas enlazadas. Lo contrario sucede si se utilizan arreglos de tamaño fijo. Por otro lado, la gran desventaja de este formato es el acceso irregular a los datos. Se pierde la localidad espacial del arreglo $n \times m$ y el acceso $O(1)$ ahora es $O(nnz)$, o $O(log(nnz)$ si los índices se encuentran ordenados. El acceso por filas o columnas es complejo. Si los elementos no están ordenados, acceder a una fila o columna implica recorrer todos los elementos. Si se encuentran ordenados por ejemplo, por filas, el acceso por columnas sigue requiriendo dicha recorrida (y viceversa). Además, si bien se evita el almacenamiento de valores nulos, se sigue teniendo redundancia, ahora en el almacenamiento de los índices. Dos elementos que se encuentren en la misma fila (columna) almacenan el mismo valor en el arreglo row_ids (col_ids). En este sentido, el formato COO no es óptimo en cuanto al costo del almacenamiento.

Si se ordenan los elementos almacenados en COO según el índice de la

fila, se tendrán los elementos con el mismo índice de forma contigua. Para eliminar esta redundancia presente en COO se pueden juntar los elementos pertenecientes a la misma fila, y guardar un solo índice en el arreglo de índices de fila. En esta idea se basa el formato CSR (del inglés *Compressed Sparse Row*) [46]. Los arreglos *vals* y *col_ ids* mantienen la misma información que en COO, aunque ahora deben estar ordenados por filas. El orden entre elementos de la misma fila en principio no es importante. El tercer arreglo, *row_ off*, mantiene los índices del comienzo de cada fila en el arreglo *vals*. En la matriz de ejemplo de la Figura 2.4, la primera fila comienza en el índice 0 y tiene dos elementos, por lo que la segunda fila comienza en el índice 2, y así sucesivamente. Al final, se agrega un índice más que indica el final de la matriz, y que no corresponde a ninguna fila. Para arreglos indexados a partir de cero, este índice corresponde a la cantidad de valores no nulos de la matriz.

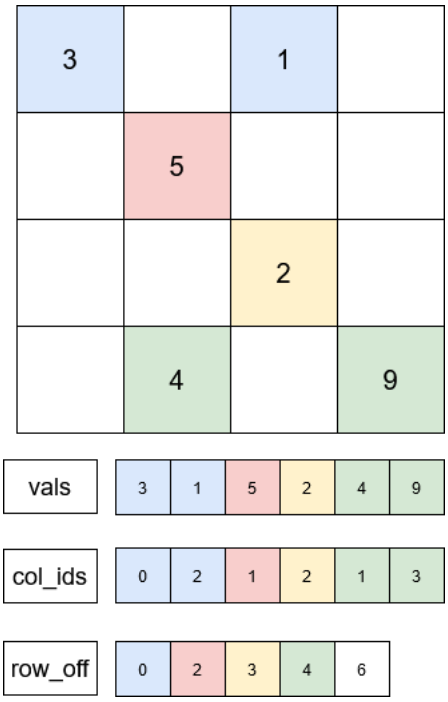

Figura 2.4: Matriz representada en formato CSR.

Para elementos de punto flotante de 8 bytes, la memoria utilizada por este formato es $12 \times nnz + (n + 1) \times 4$ bytes, donde n es la cantidad de filas de la matriz. Si hay más de un valor no nulo por fila, esto es menor que la memoria requerida por el formato COO, que es de $16 \times nnz$ bytes. Por otro lado, con este formato se simplifica el acceso por fila. Para una fila i, sus elementos se encuentran entre $vals[i]$ y $vals[i+1] - 1$. El acceso por columnas

sigue implicando una búsqueda. También existe el formato CSC (del inglés *Compressed Sparse Column*), que es el equivalente de CSR pero ordenando los valores no nulos y comprimiendo los índices por columnas en lugar de por filas.

De manera similar a CSR, se define CSC [44] (del inglés *Compressed Sparse Column*), que en lugar de almacenar índices de columnas e índices de comienzo de cada fila, almacena índices de filas e índices de comienzo de cada columna. Este formato es utilizado por lenguajes de programación en los que las matrices se almacenan por columnas, como MATLAB [55].

También existen formatos enfocados a matrices con estructuras específicas. Un ejemplo de esto es el formato CSD (del inglés *Compressed Sparse Diagonal*), para almacenar matrices de banda. Una matriz de banda contiene sus elementos no nulos en una banda diagonal, esto es, la diagonal de la matriz y alguna sub-diagonal adyacente. Formalmente, una matriz es de banda si existen dos enteros p y q tal que un elemento de la matriz a_{ij} es no nulo si y solo si $i - p \leq j \leq i + q$ [42]. CSD mantiene un arreglo de valores por diagonal y otro arreglo con los desplazamientos de cada diagonal respecto a la diagonal principal. La Figura 2.5 muestra un ejemplo de una matriz almacenada en este formato.

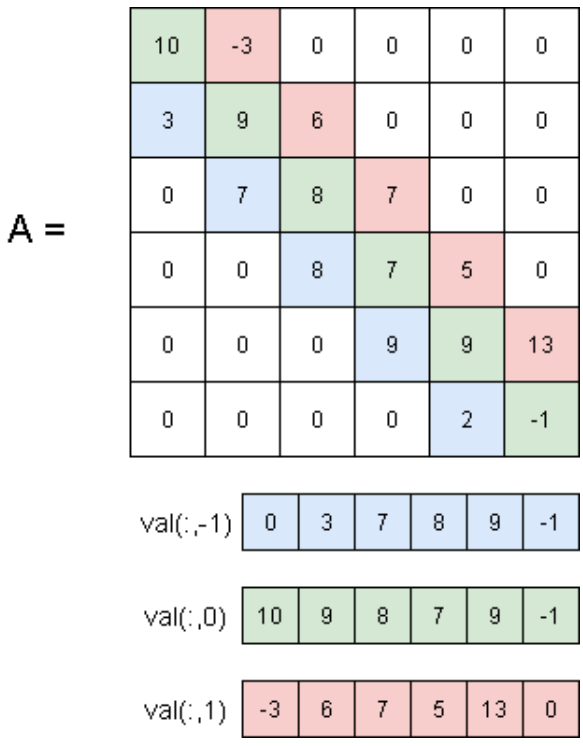

Figura 2.5: Matriz representada en formato CSD. Ejemplo extraído de [98].

El formato ELLPACK [94] [71] utiliza dos arreglos de tamaño $N \times maxNZ$, donde N es la cantidad de filas de la matriz y $maxNZ$ es la cantidad máxima de valores no nulos en una misma fila de la matriz. El primer arreglo, $COEF$, almacena únicamente los valores no nulos de la matriz, y cada fila de $COEF$ corresponde a una fila de la matriz original. Si alguna fila contiene menos de $maxNZ$ valores no nulos, se rellena con ceros a la derecha. El segundo arreglo, $JCOEF$, contiene los índices de columna de la matriz original para los valores almacenados en $COEF$, utilizando la misma correspondencia de filas y política de relleno de ceros. En [22] se utiliza una variante híbrida de ELLPACK y COO en la que las filas cuya cantidad de valores no nulos se desvíe mucho del promedio son almacenadas en COO y el resto en ELLPACK.

Muchas matrices tienen una diagonal completa de elementos no nulos, pero con otros elementos no nulos lo suficientemente lejos de la diagonal como para que no sea conveniente representarla como matriz de banda y almacenarla en formato CSD. Para estos casos existe el formato MSR (del inglés *Modified Sparse Row*) [95], una variante de CSR que almacena la diagonal principal con un vector de largo N, donde N es el tamaño de la diagonal. En este formato se mantienen dos arreglos, *vals* e *indexes*. Las primeras N posiciones del arreglo *vals* contienen los elementos de la diagonal de la matriz, en orden. La posición $N + 1$ no es utilizada. A partir del elemento $N + 2$, se almacenan los elementos no nulos de la matriz por fila, excepto los de la diagonal. En el arreglo *indexes*, se almacenan los índices de columnas de los elementos no diagonales de la matriz almacenados en *vals*, de forma que *indexes[i]* contiene el número de columna del elemento *vals[i]*, con $i > N + 1$. Las primeras N posiciones de *indexes* almacenan los índices donde comienza cada fila dentro de los valores no diagonales de la matriz almacenados en *vals* y también dentro de los índices de columnas almacenados en el propio vector *indexes*. Este formato es particularmente útil para matrices triangulares con diagonales no nulas. Es común que los elementos de la diagonal, a_{ii}, se almacenen de la forma $\frac{1}{a_{ii}}$, dado que es un valor comúnmente utilizado en la resolución del sistema de ecuaciones lineales representado por la matriz.

También derivado de CSR, el formato JDS (del inglés *Jagged Diagonal Storage* almacena diagonales "desplazadas" (*jagged diagonals*). Estas diagonales se crean partiendo de los tres vectores de CSR, ordenando las filas de manera descendente según su cantidad de valores no nulos. La primera diagonal es comprendida por los primeros elementos de cada una de las filas, la segunda

por los segundos, y así sucesivamente. Otra forma de visualizar estas diagonales es, partiendo de la matriz original, mover todos los elementos no nulos a la izquierda, y luego las diagonales serán las columnas de la matriz resultado. El formato JDS almacena cuatro vectores. El primero, *vals*, almacena los valores no nulos, desde la diagonal más a la menos numerosa. El vector *cols* almacena los índices de columna de cada elemento de las diagonales en la matriz original. Un vector *perm* almacena la permutación realizada para ordenar las filas, para poder volver al orden original. El último vector, *offsets*, contiene el índice del comienzo de cada diagonal en los vectores *vals* y *cols*. Un ejemplo se puede ver en la Figura 2.6. De manera similar, Ekambaram et al. [47] proponen TJDS (del inglés *Transposed Jagged Diagonal Storage*), donde las diagonales son comprendidas por los primeros elementos de cada columna de la matriz. Así como JDS parte del formato CSR, TJDS parte de CSC.

2.3. Formatos a bloques

Un problema de los formatos de almacenamiento disperso es la gran cantidad de espacio de memoria dedicada al almacenamiento de índices para los elementos no nulos de la matriz. En el formato COO, por ejemplo, la mitad del espacio de memoria requerido corresponde al almacenamiento de índices. Los formatos de almacenamiento a bloques buscan mitigar este problema dividiendo la matriz en sub-bloques, que pueden ser de tamaño fijo o variable, y luego almacenando índices para dichos bloques, en lugar de para cada elemento de forma individual. Los bloques almacenados tienen al menos un elemento no nulo.

Una forma de no almacenar índices para acceder a los elementos internos de cada bloque, es almacenar los bloques en formato denso. Esta estrategia tiene el inconveniente de que almacena valores nulos de forma explícita, ya que pueden haber valores nulos dentro de los bloques. La efectividad de este tipo de formatos se ve fuertemente influenciada por el patrón de dispersión de la matriz. Mientras más valores no nulos se almacenen dentro del mismo bloque, mayor es el ahorro en cuanto a índices, y menor es la cantidad de valores nulos almacenados. Es deseable entonces que los elementos no nulos de la matriz se encuentren concentrados en *clusters*.

Posiblemente el formato a bloques más ampliamente reconocido es BCSR (del inglés *Blocked Compressed Sparse Row*) [28] [32]. Como su nombre indica,

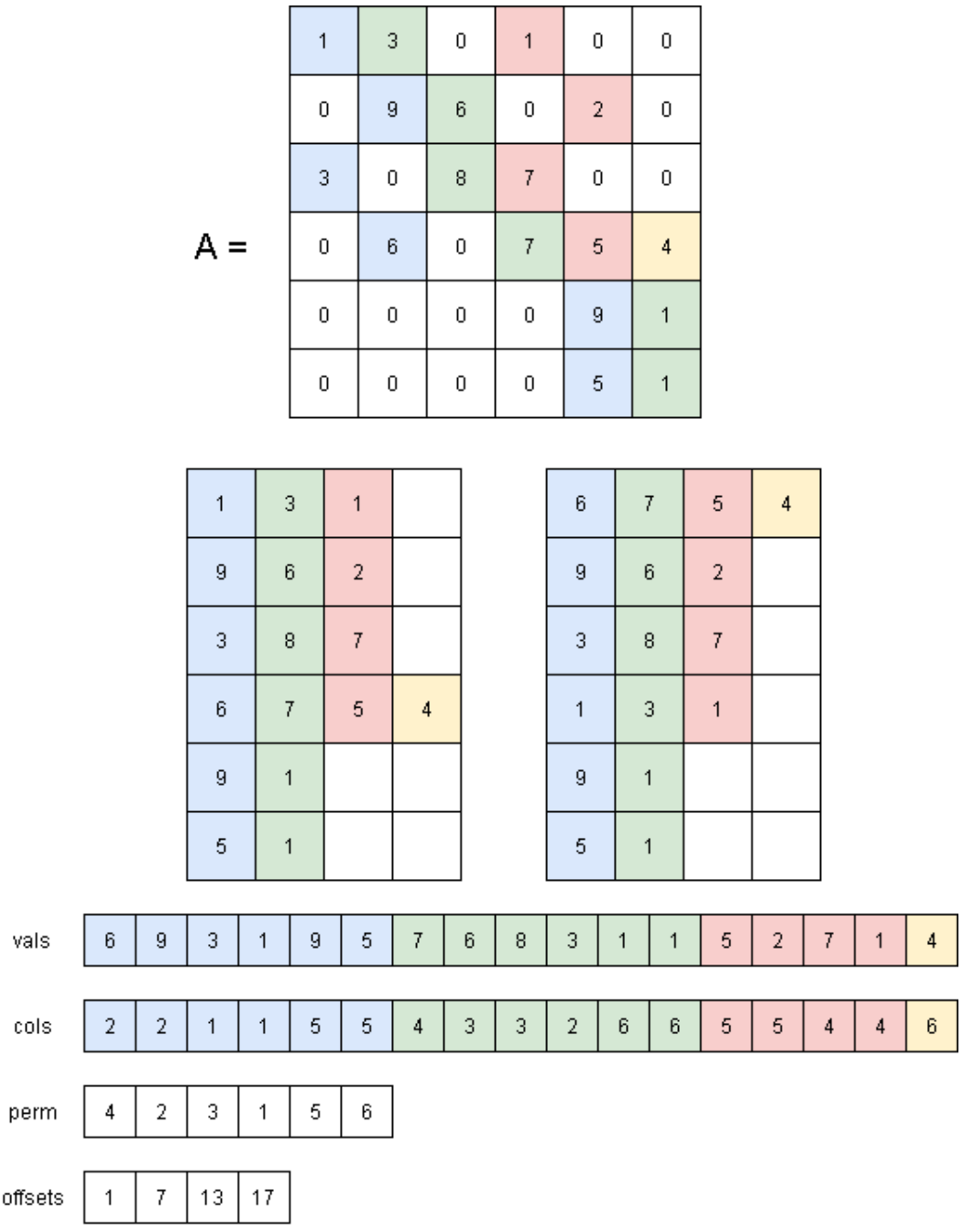

Figura 2.6: Matriz representada en formato JDS. Ejemplo extraído de [98].

es una versión a bloques de CSR. En BCSR, la matriz se divide en bloques de un tamaño cuadrado fijo $K \times K$ (si el tamaño es 1, BCSR y CSR son equivalentes). Si un bloque tiene menos de K^2 no ceros, las posiciones vacías dentro del bloque se rellenan con ceros.

Al igual que CSR, se almacenan los datos en tres arreglos. Los bloques que contengan valores no nulos son almacenados en un arreglo de valores *vals*. Los

elementos de un mismo bloque son contiguos en el arreglo de valores. El orden de los bloques dentro del arreglo de valores es por fila de bloques, mientras que el orden entre bloques de la misma fila de bloques no es importante. Luego, en un arreglo *col_ids* se almacenan los índices de las columnas de bloque para cada bloque almacenado en *vals*. Por último, en un arreglo *row_off* se almacena el comienzo de cada fila de bloques dentro del arreglo de valores *vals*, con un último elemento adicional para indicar el fin de la matriz. En la Figura 2.7 se puede ver un ejemplo de una matriz representada en formato BCSR con bloques de tamaño 2×2.

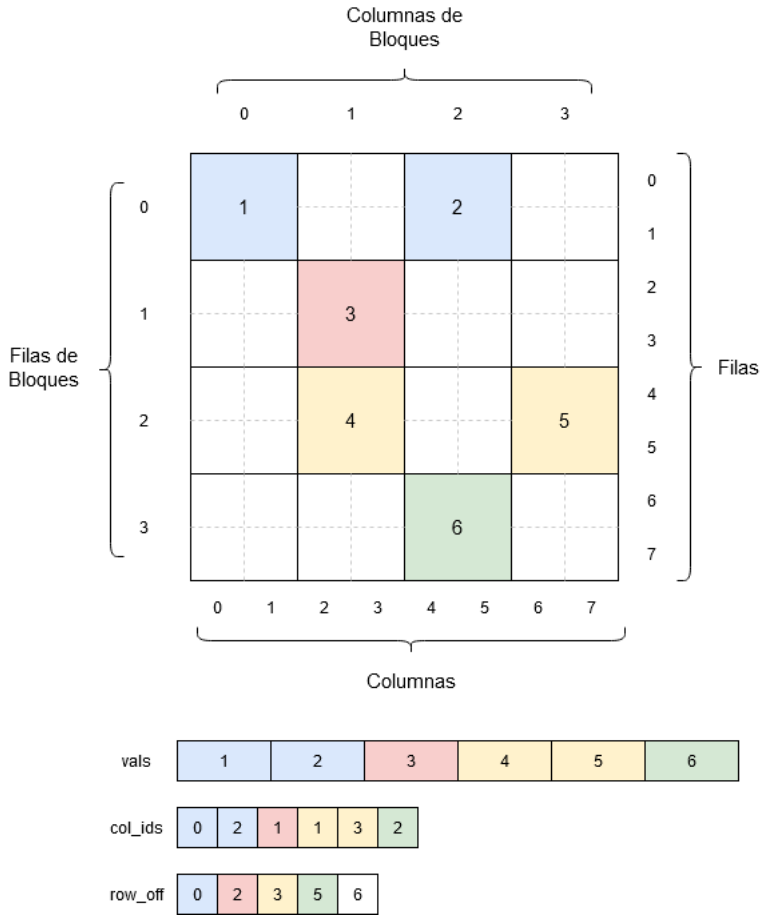

Figura 2.7: Matriz representada en formato BCSR con bloques de 2×2.

El espacio de almacenamiento requerido por BCSR es de $8 \times tamBloque \times numBloques + 4 \times numBloques + 4 \times numFilasBloques$. Se puede ver como el patrón de dispersión de la matriz influye fuertemente en el espacio de almacenamiento para BCSR, cosa que no sucede para formatos como CSR dado que almacenan cada elemento de forma individual (en CSR influye levemente

debido a los índices de comienzo de las filas). Si los elementos no nulos se encuentran concentrados en áreas específicas de la matriz, la cantidad de bloques requeridos para almacenarlos a todos disminuye, por lo que los primeros dos términos de la expresión anterior también disminuye. Por otro lado, el término $4 \times numFilasBloques$ es estrictamente menor (para bloques que no sean 1×1) a su contraparte en CSR, dado que la cantidad de filas de bloques es menor a la cantidad de filas de elementos. En el segundo sumando sucede algo similar, dado que el número de bloques es menor al número de elementos, excepto para el caso en que cada bloque contiene un único valor no nulo. En el caso ideal, que es cuando los bloques solo almacenan elementos no nulos, el primer sumando es igual a $8 \times nnz$, que es igual al sumando equivalente en CSR, mientras que los otros dos sumandos son significativamente menores a sus equivalentes.

Smailbegovic et al. [99] plantean una extensión de BCSR, basada en la idea de juntar los vectores que almacenan los valores de la matriz con los que almacenan la información posicional en una sola lista, con el objetivo de una mayor reutilización espacial de los recursos *on-chip*. De esta forma, cada elemento no nulo de la matriz se representa como una tupla que contiene su valor y su información posicional. Luego, los elementos que pertenezcan a la misma fila se agrupan en *chunks*. Como cada *chunk* tiene un distinto largo y solamente se puede cargar una determinada cantidad de chunks en los recursos *on-chip*, se le agrega un campo a la lista de tuplas denominado *chaining bit*, que indica si hay más elementos no nulos dentro de la misma fila.

De forma análoga a BCSR, BCSC (del inglés *Blocked Compressed Sparse Column*) adapta el formato CSC a bloques. La diferencia con BCSR es que en lugar de mantener los índices de columnas de los elementos y del comienzo de las filas de bloques, se almacenan los índices de filas de los elementos y del comienzo de las columnas de cada bloque.

Otra adaptación de uno de los formatos más comunes es BCOO (del inglés *Blocked COO*) [110] [112]. Como su nombre indica, adapta el formato COO a bloques, que es el equivalente a BCSR, pero almacenando los índices de la matriz de bloques en formato COO. Los autores de [110] y [112] también proponen una variante comprimida del formato que llaman BCCOO (del inglés *Blocked Compressed COO*). La modificación primordial que tiene BCCOO sobre BCOO es la transformación del arreglo de índices de filas a un arreglo llamado *bit flag* de unos y ceros. Esta transformación se hace restando elementos contiguos del arreglo de índices de filas. Si esta resta es 0 o 1, se agrega

una entrada con ese valor al arreglo de bits. Si la resta es mayor a 1, se agrega la misma cantidad de unos al arreglo de bits. Luego, se pasan los ceros a unos y viceversa. El resultado es un arreglo de bits donde un 1 indica que aún hay más elementos en la fila del elemento actual, mientras que un cero indica el fin de una fila. El arreglo de bits almacena un valor por cada bloque no nulo, y un valor por fila que indica el fin de la misma. Para algunos casos, el mismo bit puede corresponder a un bloque y al fin de la fila, pero de cualquier forma el espacio de almacenamiento requerido por el arreglo de bits está acotado superiormente por $(numBloques + numFilasBloques)/8$ bytes. Este número es considerablemente menor que el espacio requerido por el arreglo de índices de filas de bloques en BCOO, que es $4 \times numBloques$. Si bien en COO no importa el orden de los elementos almacenados, para poder utilizar BCCOO los elementos deben estar ordenados por filas (el orden entre elementos de la misma fila puede ser cualquiera).

Yan et al. [109] proponen BCSR&BCOO, un híbrido entre estos dos formatos. La idea principal es dividir la matriz en sub-bloques de un tamaño tal que entren en la caché principal, de manera de reducir el tiempo de ejecución. Luego de determinar los bloques, se define un umbral tal que aquellos bloques con una cantidad de valores no nulos que supere dicho umbral se almacenan en formato BCSR (mejor para bloques más densos) y los que estén por debajo se almacenan en formato BCOO (mejor para bloques más dispersos).

El formato BCSD (del inglés *Blocked Compressed Sparse Diagonal*) es similar a BCSR, pero cada bloque almacena diagonales parciales de la matriz. La idea de la que deriva este formato, presentada por Agarwal et al. [11], consistía en dividir una matriz de tamaño $m \times n$ en sub-matrices de tamaño $md \times n$, donde md es un parámetro configurable, y luego extraer diagonales completas (de tamaño md) de cada una de estas sub-matrices. De manera similar, en BCSD se divide la matriz en franjas de k filas, donde k es el tamaño de bloque (cada bloque es un vector que almacena una diagonal). Esto es equivalente a decir que cada bloque de tamaño k debe cumplir que su primer índice (i, j) es tal que $mod(i, k) = 0$ [67]. Luego, cada diagonal de estas franjas se almacena en bloques que son representados a través de 3 arreglos. Un arreglo *vals* almacena los elementos no nulos, ordenados por bloque, y luego por franja. Otro arreglo, *col_ids* almacena los índices de columna de cada bloque dentro de la matriz original. Por último, un arreglo *row_off* almacena el índice de comienzo de cada franja dentro del arreglo *vals*. Vuduc [104] presenta una extensión

de este formato, denominada RSDIAG, en la que además de los tres arreglos mencionados, mantiene otro que contiene la cantidad de diagonales en cada franja.

Una variante a bloques de ELLPACK, BELLPACK, es presentada en [32]. Busca combinar las ventajas de BCSR y ELLPACK, siguiendo la idea del formato híbrido presentado en [22]. BELLPACK reorganiza la matriz en bloques densos de tamaño fijo y luego ordena las filas de bloques de forma descendente según la cantidad de bloques por fila, similar al ordenamiento realizado en JDS. Luego, divide las filas de bloques en sub-matrices de igual tamaño y las almacena en formato ELLPACK.

Karakasis et al. [67] proponen versiones alternativas de BCSR y BCSD denominadas BCSR-DEC y BCSD-DEC, respectivamente. Estos formatos descomponen la matriz original en dos sub-matrices, buscando eliminar el *padding* de ceros. La primera contiene solamente bloques completos de BCSR o BCSD y es almacenada en estos formatos. La segunda sub-matriz contiene los elementos restantes, que no pueden ser agrupados en bloques del tamaño establecido, y es almacenada en formato CSR.

Vassiliadis et al. [102] proponen BBCS (del inglés *Block Based Compressed Storage*, formato en el cual la matriz se divide en Bloques Verticales (*Vertical Blocks* o *VB*), sub-matrices que abarcan todas las filas de la matriz original, y hasta *vbSize* columnas. BBCS almacena las matrices como una secuencia de elementos con seis valores distintos:

- *Value*, que guarda los elementos no nulos de la matriz si la flag *ZR* es igual a 0. En caso contrario, indica la cantidad de filas siguientes del VB que no contienen valores no nulos.
- *CP* (*Column Position*), que indica el índice de la columna dentro del VB del elemento no nulo guardado en *values* (si es un elemento nulo este valor no representa nada).
- *EOR* (End of Row) que es 1 si el elemento actual es el último de la fila actual del VB, y 0 en otro caso.
- *ZR* (Zero-Row) que es 1 si la fila actual del VB no contiene ningún elemento no nulo y 0 en otro caso.
- *EOB* (End of Block) que es 1 si el elemento actual es el último no nulo dentro del VB.
- *EOM* (End of Matrix) que es 1 si el elemento actual es el último no nulo

de toda la matriz.

Una ventaja de este formato es que los índices CP, al ser relativos al bloque, es posible que puedan ser representados con menos bits que en formatos donde el mismo índice es relativo a la matriz. También, a diferencia de otros formatos a bloques, no se almacenan valores no nulos explícitos, que son reemplazados por un único elemento en la secuencia por cada conjunto consecutivo de filas de valores nulos del VB. Por otro lado, si bien representan solamente un bit por elemento de la secuencia, EOB y EOR implican una gran redundancia de datos a nivel de VB y de matriz, respectivamente. Además, el uso de las cuatro banderas dificulta posibles modificaciones a la matriz, dado que se podrían tener que actualizar con cada elemento agregado.

Los formatos hasta ahora presentados utilizan un tamaño fijo para los bloques. Pinar et al. [92] presentan un formato en el que se mantienen bloques de tamaño variable, donde cada bloque es unidimensional y almacena elementos contiguos de una fila. Si bien en el trabajo original se refiere a este formato como BCRS (del inglés *Blocked Compressed Row Storage*), se referirá al mismo con el nombre que se le da en [67], 1D-VBL (del inglés *1-Dimensional Variable Block Length*). 1D-VBL utiliza los mismos tres vectores que BCSR, pero los índices de columna almacenados refieren al índice de columna del primer elemento del bloque. Además, se mantiene un cuarto vector que guarda los tamaños de cada bloque.

El formato VBR [95] (del inglés *Variable Block Row*) utiliza bloques de dos dimensiones con tamaño variable. En este formato se particiona la matriz tanto por filas como por columnas. Dichas particiones son almacenadas en dos vectores, *KVSTR* y *KVSTC*, que contienen los índices de la primer fila de cada fila de bloques y los índices de la primer columna de cada columna de bloques, respectivamente. También se utilizan los mismos tres vectores que CSR, salvo que los elementos no nulos son ordenados por bloque y luego por columna. Un último vector almacena los índices del comienzo de cada bloque. Ahrens et al. [12] proponen una versión unidimensional de VBR, denominada 1D-VBR, en la que solamente se particiona la matriz por sus filas.

Jain-Mendon et al. [65] proponen VDCB (del inglés *Variable Dual Compressed Blocks*, que divide la matriz original en sub-matrices de tamaño variable. Cada una de estas sub-matrices es representada con tres componentes: un *header*, que contiene toda la información posicional para ubicar cada submatriz

dentro de la matriz original, un *bitmap*, que contiene información posicional para los elementos no nulos dentro de cada submatriz, y un vector que contiene los valores no nulos de la submatriz, ordenados por columnas.

2.3.1. Formato bmSparse

bmSparse es un formato de almacenamiento a bloques para matrices dispersas basado en *bitmaps* propuesto por Zhang y Gruenwald en 2018 [111]. En dicho trabajo se menciona que en ese entonces la mayoría de los formatos de almacenamiento propuestos para matrices dispersas, incluidos aquellos que utilizan bitmaps, suelen ser evaluados en la operación SpMV. El objetivo entonces es el desarrollo de un formato que se adecue a la operación de SpGeMM.

En este formato, las matrices son divididas en bloques de 8×8. Cada bloque tiene asociada una posición dentro de la matriz, representada por un índice que indica la fila de bloque y un índice que indica la columna del bloque. En *bmSparse*, ambos índices de un bloque son almacenados en un único entero uint64_t, donde los primeros 32 bits representan el índice de la fila y los últimos 32 representan el índice de la columna. De esta forma es posible representar hasta 2^{32} filas y columnas de bloques, es decir, hasta 2^{64} bloques. Dado que los bloques son de tamaño 8×8, esto permite representar matrices de tamaño hasta $2^{35} \times 2^{35}$.

El formato *bmSparse* utiliza tres vectores para almacenar las matrices:

- *keys*: Arreglo de enteros (uint64_t) que almacena la posición de los bloques no nulos de la matriz a través de un índice de columna y otro de fila. El orden dentro del arreglo es por filas y luego por columnas.
- *bmps*: Arreglo de enteros (uint64_t) que almacena el bitmap asociado a cada bloque. Si el bit i del bitmap es un 1, quiere decir que el elemento i dentro del bloque (por filas y luego por columnas) es un valor no nulo. Si es un cero, el elemento es nulo.
- *values*: Arreglo con los valores no nulos de la matriz. Los bloques mantienen el orden en el que se encuentran en el arreglo de *keys*. Dentro del mismo bloque, los valores se ordenan por fila y luego por columna.

Si bien esta es la representación base del formato, para optimizar la operación de SpGeMM se agrega un vector adicional, *offsets*, que almacena las posiciones de comienzo de cada bloque dentro del arreglo *values*. Para un bloque

i, sus valores están almacenados entre *values[offsets[i-1]]* y *values[offsets[i]]*, salvo para el primer bloque, cuyos valores están almacenados entre *values[0]* y *values[[offsets[0]]]*. Sin el uso de este vector, el acceso a los valores de un bloque requiere cómputos adicionales cada vez que el bloque se utilice en una multiplicación. Por esta razón, resulta beneficioso el previo cálculo de este vector. En la Figura 2.8 se muestra un ejemplo de una matriz almacenada en formato bmSparse (sin el vector de *offsets*).

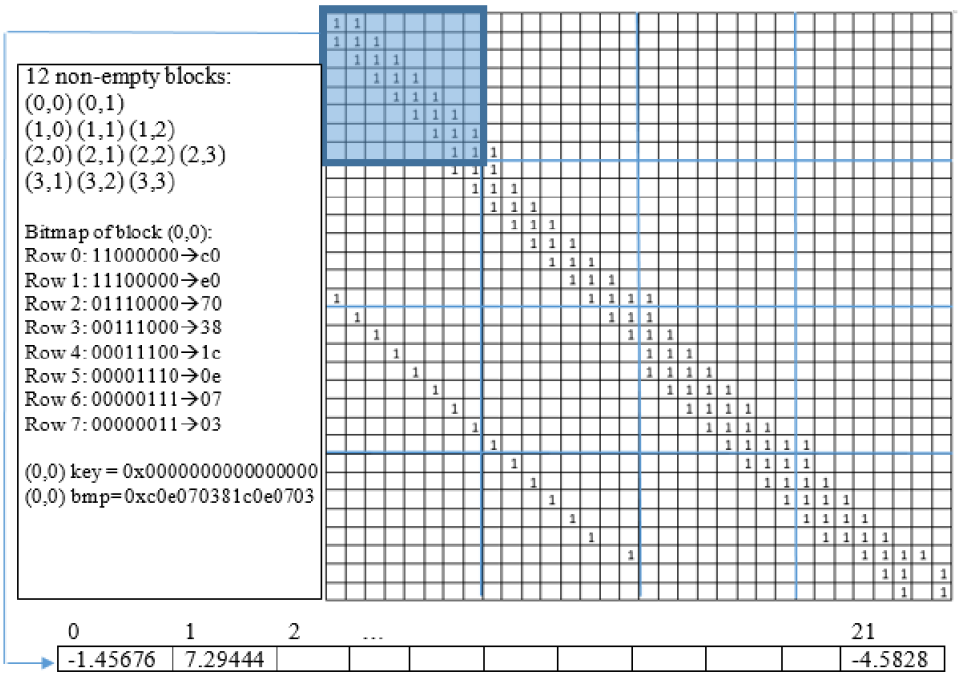

Figura 2.8: Matriz representada en formato bmSparse. Imagen extraída de [111].

2.4. SpGEMM

La multiplicación general de matrices dispersas, SpGeMM, tiene la forma:

$$D = A \times B + C$$

donde las matrices de entrada A, B y C y la matriz de salida D están almacenadas en algún formato disperso. La diferencia principal con la variante densa radica en la importancia de explotar la dispersidad de las matrices involucradas, dado que no hacerlo implica el desaprovechamiento de recursos de cómputo en cálculos con valores nulos. En el trabajo, no se considera la matriz

de entrada que solamente es sumada y la matriz de salida será llamada C. En el Algoritmo 1 se presenta el pseudocódigo planteado por Gustavson en 1978 para esta operación [59]. Esta operación está presente en numerosos contextos del álgebra lineal y algoritmos de grafos, como solvers de métodos multigrilla algebraicos [53] [21] [20], conteo de triángulos [33] [17] [107] [36], búsqueda en anchura (BFS) con múltiples orígenes [56] [29] [101], búsqueda de camino más corto [31] y coloreado de grafos [66] [38].

Esta operación posee una baja intensidad computacional. Esto es, la cantidad de operaciones de punto flotante es considerablemente más baja que los accesos a memoria. Este es uno de los cuellos de botella más importantes de la operación. Otra complejidad es que al multiplicar dos matrices dispersas el patrón de no ceros de la salida no solo depende de los patrones de dispersión de cada una de las matrices, sino también de como se vinculan entre sí. Esto genera que sea complejo estimar con antelación la memoria necesaria para almacenar la matriz de salida.

Algorithm 1: SpGeMM propuesto por Gustavson (1978)

```
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
```

La operación SpGeMM típicamente se divide en 5 etapas, presentadas en la Figura 2.9. La etapa de *size prediction* se encarga de predecir la memoria necesaria para almacenar la matriz resultado, basándose en las matrices de entrada. La etapa de *memory allocation* reserva la memoria que se predijo en la etapa anterior para almacenar la matriz de salida en el dispositivo deseado. En la etapa *work partition and load balance* se busca dividir el trabajo realizado por la operación de manera de explotar lo más posible el paralelismo. En la etapa *numeric multiplication* se realizan las operaciones aritméticas necesarias para obtener los resultados parciales. Finalmente, la etapa de *result*

accumulation reduce estos resultados parciales para obtener el resultado final.

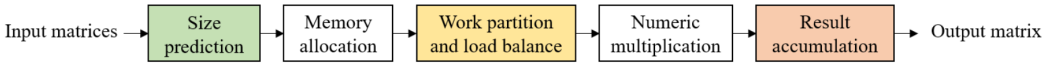

Figura 2.9: Etapas típicas de la operación SpGeMM. Imagen extraída de [51].

2.4.1. Predicción de tamaño y reserva de memoria de la matriz de salida

En la multiplicación de matrices densas generales (GeMM) es sencillo calcular el espacio requerido por la matriz de salida. Solamente es necesario saber sus dimensiones, calculadas a partir de las dimensiones de las matrices de entrada, y el tipo de dato que almacena. En SpGeMM, la matriz de salida se almacena en algún formato disperso, para los cuales el orden de memoria requerido depende fuertemente de la cantidad de valores no nulos y su distribución en la matriz, datos que no son conocidos hasta luego de realizada la operación. Por esto, es necesario realizar una predicción lo más precisa posible del espacio de almacenamiento requerido por la matriz resultado antes de ejecutar la operación. A continuación se presentan las cuatro estrategias principales para realizar esta predicción: método preciso, método probabilístico, método del límite superior y método progresivo.

A. Método preciso

Este método divide la operación de SpGeMM en una etapa simbólica y una numérica. En la etapa simbólica se calcula la cantidad de valores no nulos de la matriz resultante de forma precisa en base a los valores no nulos de las matrices de entrada. Usualmente, se utilizan versiones simplificadas de las matrices de entrada, las cuales almacenan valores binarios, que indican si un elemento es nulo o no en la matriz original. De esta forma, es posible realizar una multiplicación simplificada entre estas dos matrices, donde las operaciones aritméticas son menos costosas al tratarse de valores binarios. Los valores de las entradas no nulas de la matriz son luego calculados en la etapa numérica.

Una vez realizada la etapa simbólica, se puede utilizar el resultado para consencuentes operaciones cuyas matrices de entrada posean la misma estructura. Por otro lado, este método presenta la desventaja evidente de requerir

una etapa simbólica que puede ser casi tan costosa como la numérica. Trabajos previos realizan modificaciones en las matrices simbólicas de entrada, comprimiendo bits en enteros y utilizando operaciones lógicas de bits para acelerar los cómputos [40] [39].

Deveci et al. [39] utilizan este método en su algoritmo de SpGeMM por filas, KKMEM. En la etapa simbólica, además, se efectúa una etapa de compresión sobre la matriz del lado derecho de la multiplicación (B), de manera de codificar múltiples columnas dentro de un mismo entero. En la fase numérica, cada fila es procesada por un único thread. Las implementaciones de SpGeMM de cuSparse [4], MKL [5], Kokkos Kernels [93] y RMerge [58] también utilizan este método.

B. Método probabilístico

Este método propone transformar la predicción de la cantidad de valores no nulos de la matriz en un problema de grafos, utilizando un algoritmo basado en métodos de Monte Carlo para estimar el tamaño de los conjuntos alcanzables. Este enfoque ofrece menores costos computacionales en comparación con la multiplicación simbólica del método preciso. La eficacia de este método radica fuertemente en la precisión del método probabilístico, dado que una estimación inicial fallida implica la necesidad de reservar memoria adicional.

C. Método del límite superior

El método del límite superior acota superiormente la cantidad de valores no nulos que puede tener la matriz resultado, y utiliza esta cota al reservar la memoria. La cota más comúnmente utilizada es la resultante de contar la cantidad de valores no nulos en las filas de B correspondientes a cada valor no nulo de A. Este método es el más sencillo de implementar y el menos costoso computacionalmente, pero tiene la evidente desventaja de que se suele reservar cantidades de memoria mucho mayores a las necesarias. El algoritmo ESC (por sus siglas en inglés *Expansion*, *Sorting* y *Compression*) [21] utiliza este método.

D. Método progresivo

Este método reserva la memoria de manera dinámica según sea necesaria. Primero se reserva una cierta cantidad de memoria y se comienza la multiplicación. Si la memoria resulta insuficiente, se reserva un bloque más grande de

memoria. La implementación de SpGeMM de MATLAB [64] utiliza este método, multiplicando el tamaño del bloque de memoria reservado por un factor constante (típicamente 1,5) cada vez que se necesite más memoria [55].

2.4.2. Partición de matrices y balance de carga

Esta etapa busca dividir el trabajo de forma de maximizar el paralelismo entre las unidades computacionales. Las estrategias más comunes para particionar el trabajo de la SpGeMM son presentadas por Ballard [19] mediante el concepto de cubo computacional o *computational cube* o *workcube* en inglés (ver Figura 2.10).

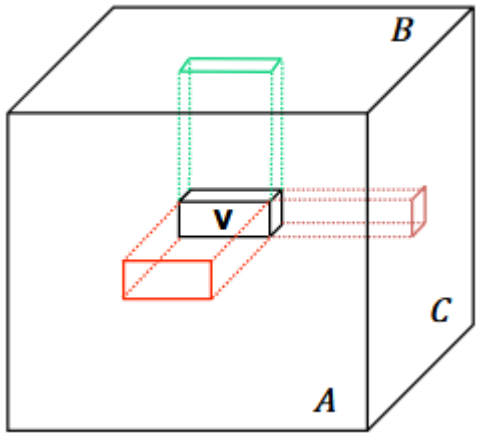

Figura 2.10: Cubo computacional para la multiplicación de matrices cuadradas. La proyección de un voxel en la capa frontal corresponde a un valor de la matriz A, mientras que la proyección en la capa superior corresponde a un elemento de la matriz B. Cuando estos dos elementos no son nulos, su multiplicación es acumulada en la entrada de C correspondiente a la proyección lateral del mismo voxel. Imagen extraída de [19].

El algoritmo ESC [21] [34] busca realizar un balanceo de carga dividiendo la operación SpGeMM en tres etapas, implementadas utilizando primitivas de paralelismo de datos como *gather*, *scatter* y *scan*, entre otras. ESC divide el conjunto de filas de la matriz de entrada A en múltiples sub-matrices, de igual manera que las particiones de tipo T del cubo computacional, en las que un conjunto de filas de A se vincula con la matriz B completa para generar resultados finales de una parte de la matriz resultado C. En la etapa de expansión, cada una de estas sub-matrices es agrupada con la matriz B para generar una

lista de coordenadas que indican productos que se deben hacer para generar la matriz resultado. Posteriormente, se ordena la lista de coordenadas de manera que queden contiguas aquellas que contribuyen al mismo elemento de la salida. Finalmente, se comprimen las coordenadas contiguas correspondientes al mismo valor de la salida, reduciendo los valores en uno solo.

Anh et al. [14] distribuyen equitativamente el trabajo de cada thread sobre la *work list*, listas que se almacenan en memoria global generadas a partir de los índices de A y B que son accedidos al computar productos intermedios.

Liu y Vinter [76] presentan un algoritmo de SpGeMM dividido en cuatro etapas. La primera consiste en calcular una estimación de la cantidad de valores no nulos de la matriz resultado mediante el método *upper bound*. Para ello, se genera un arreglo U de tamaño m, donde m es la cantidad de filas de la matriz C, y se guarda en cada entrada del arreglo una cota superior de la cantidad de multiplicaciones necesarias para la fila correspondiende de C. La segunda etapa se centra en la tarea de la distribución y balance de carga así como la reserva de memoria. Se definen 38 *bins* que están divididos en 5 grupos. Los bins se representan como un arreglo de tamaño m, que contiene los índices del arreglo U y está dividido en 38 segmentos, donde cada segmento guarda índices de filas según su tamaño.

Para la etapa de cómputo, se utilizan distintas estrategias (heap, ESC con sort bitónico y un método de merge) dependiendo del grupo de bins al que pertenezca cada fila. La cuarta etapa consiste en definir y reservar el espacio de memoria final para la matriz resultado C y copiar entradas no nulas de la matriz temporal a la final.

2.4.3. Acumulación de resultados intermedios

La operación SpGeMM conlleva múltiples acumulaciones de resultados parciales antes de obtener el resultado final. Estos resultados parciales son acumulados en una estructura de datos comúnmente denominada acumulador, que puede ser densa o dispersa. Existen tres tipos de estrategias de acumulación de resultados intermedios: acumulación densa, acumulación basada en listas (o basada en ordenamientos) y acumulación basada en hash.

A. Acumulación densa

Esta técnica utiliza vectores densos como acumuladores para almacenar los resultados intermedios. El *SPA* (*sparse accumulator*) de MATLAB [55] utiliza el acumulador denso propuesto por Gustavson [59], aunque el vector que almacena índices de columnas es representado por una lista no ordenada. Los acumuladores densos ofrecen accesos directos a los resultados intermedios. Sin embargo, esto es a expensas de un un alto costo de almacenamiento, lo cual no es lo más adecuado para GPUs [39]. Patwary et al. [91] mencionan que los mejores resultados de su implementación de multiplicación de matrices en CPU se obtuvieron utilizando acumuladores densos.

B. Acumulación basada en tablas de hash

Este tipo de acumuladores almacena los resultados intermedios en tablas de hash y utiliza sus índices de columna como la clave para el acceso. De igual manera que sucede para la matriz de salida en SpGeMM, el principal problema con el uso de tablas de hash es la incertidumbre en relación a la memoria necesaria para almacenar los resultados intermedios. Para atacar este problema, se utilizan las alternativas presentadas en la Sección 2.4.1. Por otro lado, si el tamaño requerido es muy grande, es posible que las tablas de hash no puedan ser almacenadas en la memoria compartida de la GPU. Por ejemplo, Nagasaka et al. [82] utilizan tablas de hash para una etapa simbólica de cálculo de la cantidad de valores no nulos por fila y para una etapa numérica donde se calculan los valores de la matriz de salida. Anh et al. [14] utilizan tablas de hash en memoria compartida para la adición de resultados intermedios. Para un resultado intermedio c_{ijk} se utiliza la key (i, k) para acceder a la tabla de hash. Dos resultados que tengan la misma key deben ser sumados y, cuando hayan sido procesados todos los resultados intermedios, se pasa el contenido de la tabla de hash a memoria global. Dado que es difícil que la tabla de hash de toda la matriz C pueda ser almacenada en memoria compartida, se propone dividir la matriz A en *particiones*, definidas en el artículo como grupos de filas consecutivas. Como en este algoritmo iterar por una fila de A genera una fila de C, el procesar una partición completa genera todos los valores no nulos de la matriz C para las mismas filas que la partición. La idea es entonces procesar cada partición de a una, y al llegar al final de ella, pasar los contenidos de la tabla de hash a memoria global, vaciar la tabla y repetir el proceso con la

siguiente partición.

C. Acumulación basada en ordenamientos

En este método se ordenan los resultados intermedios de manera que los elementos que tengan el mismo índice de columna se almacenen de forma contigua. Existen distintas estrategias que dependen del algoritmo de ordenamiento utilizado. El *radix sort* considera los elementos a ordenar como una secuencia de dígitos y los ordena un dígito a la vez, comenzando por las unidades. Un ejemplo de esto es el algoritmo ESC [21], que utiliza la primitiva *stable_sort_by_key* de Thrust, basada en radix sort [97]. El *merge sort* divide el conjunto de resultados intermedios en distintas secuencias, las ordena y las combina de forma ordenada. Gremse et al. [58] proponen RMerge, que utiliza un algoritmo fila por fila que toma un conjunto de filas de B, multiplicado por una fila de A, y combina los resultados utilizando subwarps, comprimiendo los elementos con mismo índice de columna. Winter et al. [106] utilizan bloques de hilos en porciones de igual tamaño de la matriz A. Cada bloque computa SpGeMM para dicha porción de A y la matriz B, y genera distintos fragmentos de la matriz resultado C. Luego, se utilizan tres algoritmos basados en merge sort para combinar estos fragmentos, dependiendo de en cuantos de ellos se haya dividido cada fila. Liu et al. [75] proponen dos implementaciones de acumuladores dispersos, basados en los algoritmos de ESC y merge sort, en los que se utilizan registros de GPU para recuperar datos y almacenar resultados.

Azad et al. [16] utilizan una formulación columna por columna en la que se calculan matrices intermedias con resultados parciales. Estas matrices se almacenan a través de una lista de ternas (i, j, val) que almacenan los índices de fila y columna y el valor de cada elemento no nulo, ordenados por (j, i). Luego, para ordenar k de estas listas se utiliza un heap de tamaño k, en el que se almacenan los mínimos de cada una de estas listas, según el orden lexicográfico (j, i). El ordenamiento consiste en encontrar el mínimo elemento del heap y combinarlo en el resultado. En la lista combinada se reducen los valores que correspondan a los mismos índices. Nagasaka et al. [82] utilizan un acumulador basado en heaps para implementar SpGeMM en arquitecturas many-core.

2.4.4. SpGEMM para bmSparse

La propuesta para la implementación de SpGeMM para *bmSparse* [111] se basa en dos grandes etapas. La primera es la creación de una lista de tareas, denominada *task list*, que contiene los bloques de las matrices de entrada A y B que se deben multiplicar entre sí para generar los bloques de la matriz de salida C. La segunda etapa consiste en procesar cada tarea para construir la matriz resultado.

En [78] se realiza una implementación de SpGeMM para *bmSparse*, dividida en nueve etapas identificadas como T_1,...,T_9. Esta implementación presenta diversas optimizaciones respecto a la versión original de [111]. En el marco de este trabajo, se desarrolló una versión de SpGeMM que sigue los lineamientos originales de [111], apoyándose en el trabajo presentado en [78]. El algoritmo propuesto se divide en siete etapas identificadas como T_1,...,T_7. Las etapas T_1, T_2 y T_3 se encargan de crear la task list, la cual se representa como lista de ternas (i,j,k), llamadas *tasks*, obtenidas siguiendo la regla:

$$C_{ik} = \sum_j A_{ij} \times B_{jk}. \tag{2.4}$$

Si $A\_keys[n]$ almacena la key (i,j), la task list puede ser considerada como la unión de los conjuntos t_n, $n \in [0, size(A\_keys)]$, donde t_n representa el conjunto de todas las ternas en las cuales participa la key (i,j) de A_keys. Si el conjunto de *keys* de la fila j se define como $B\_row_j = \{x \mid x \in B\_keys \wedge row(x) = j\}$ entonces t_n puede definirse como:

$$\begin{aligned} &t_n = \{(i,j,k) \mid \\ &A\_keys[n] = (i,j) \wedge \exists x \in B\_row_j : col(x) = k\}. \end{aligned} \tag{2.5}$$

La principal ventaja de definir la task list como la unión de los conjuntos t_n es que estos conjuntos pueden calcularse de manera sencilla dadas las características del formato. De hecho, existe un conjunto t_n por cada key de A, y los bloques en cada conjunto pertenecen a la misma fila de bloques de B. Debido a que el formato *bmSparse* garantiza que las keys están ordenadas por fila y luego por columna, las keys asociadas a la misma fila de bloque se almacenan de forma contigua. Esto permite que los conjuntos t_n sean generados utilizando las posiciones donde comienzan las filas de B y su tamaño.

La etapa T_1 se encarga de calcular el número de elementos de los conjuntos t_n, para cada key de A. En la ecuación 2.5, esto es lo mismo que calcular el

número de bloques no nulos en la fila j de B. La salida generada por esta etapa es un arreglo B_count, definido por la relación $B\_count[j] = |B\_row_j|$.

En la etapa T_2 se asocia el n-ésimo elemento de A_keys con $|t_n|$, calculado en T_1.

La etapa T_3 consiste en crear un arreglo de desplazamientos, denominado *pos*, y luego generar la task list. La task list es representada con un arreglo tridimensional, *task list*, que es el resultado de la unión de todos los conjuntos t_n.

En la etapa T_4, la task list es ordenada de manera que las tareas asociadas al mismo bloque de salida sean contiguas, lo cual es equivalente a ordenar las ternas (i,j,k) por (i,k).

La etapa T_5 es responsable de determinar qué bloques de la matriz resultante serán no nulos, lo que corresponde a computar el arreglo de keys de la matriz de salida. Esto se logra obteniendo una tupla (i,k) por cada terna (i,j,k) y manteniendo un único representante por conjunto de tareas asociado al mismo bloque de salida.

La etapa T_6 procesa las tareas para generar la salida. Procesar una tarea involucra crear una versión densa de los bloques de entrada, multiplicarlos, y acumular los valores del bloque resultante en el bloque correspondiente de la salida. Dada una tarea (i,j,k), los valores no nulos de los bloques de A y B pueden ser obtenidos por los bitmaps $A\_bmp(i,j)$ y $B\_bmp(j,k)$, respectivamente. En esta etapa también se obtiene el bitmap resultante de la multiplicación de los bloques. Por último, los bitmaps que contribuyen al mismo bloque de salida son reducidos.

En la etapa T_7, el arreglo de valores generado en la etapa T_6 es tomado como entrada, para generar un nuevo arreglo que solamente contiene los valores no nulos del arreglo original. Esto es porque se pueden haber generado ceros en la etapa de multiplicación. El orden relativo de los valores se mantiene.

2.4.4.1. Implementación en GPU

En esta sección, se presenta la implementación de la propuesta base de SpGeMM para bmSparse, descrita en la Sección 2.4.4. Todas las etapas son implementadas en C++, utilizando algoritmos de la librería *Thrust* 2.1.2. La única excepción es la etapa de multiplicación de bloques, que es implementada por un kernel de CUDA.

$\mathbf{T_1}$. Cálculo de $|t_n|$

El objetivo de esta etapa es construir un arreglo B_count determinado por $|t_n| = B\_count[j] = |B\_row_j|$. Es posible calcular $|B\_row_j|$, con $j \in [0, size(B\_keys))$, a través de la primitiva *thrust::reduce_by_key*, utilizando B_keys como arreglo de keys y el número de fila del bloque como criterio de comparación. El orden de los elementos en el vector resultante B_count es el deseado dado que las keys en B_keys se encuentran ordenadas por fila y luego por columna.

$\mathbf{T_2}$. Asociar $A\_keys[n]$ con $|t_n|$

En esta etapa se busca construir un arreglo col_count, definido por la relación $col\_count[n] = B\_count[col(A\_keys[n])]$. Esto puede lograrse con un llamado a la primitiva *thrust::gather*, utilizando un arreglo con las columnas de A_keys como arreglo de índices y B_count como arreglo de datos. El valor de las columnas puede ser obtenido sin tener que almacenar un arreglo en memoria utilizando un iterador *thrust::transform_iterator* y un functor que retorna la columna de una key.

$\mathbf{T_3}$. Construir la task list

Para crear la task list, primero se calculan las posiciones iniciales de cada conjunto t_n en el arreglo *task list*, para todo n. Esto se hace con un llamado a la primitiva *thrust::exclusive_scan* sobre el arreglo B_count. Luego se utiliza *thrust::scatter* y un iterador *thrust::counting_iterator* para almacenar cada key en la posición asociada, calculada en el paso anterior. Finalmente, se ejecuta *thrust::inclusive_scan* en el arreglo, utilizando *thrust::maximum* como el operador binario. Para obtener la tercera coordenada de las tasks de los arreglos intermedios se utiliza 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.

$\mathbf{T_4}$. Ordenamiento de la task list

Para ordenar la task list, simplemente se utiliza la primitiva *thrust::sort*, ordenando las tareas con respecto al valor (i,k).

$\mathbf{T_5}$. Determinar el arreglo de keys de C

Siendo la task list un conjunto de ternas (i,j,k) de coordenadas de matriz, determinar la disposición de C involucra agrupar las tareas por (i,k), obteniendo una key por cada grupo a través de la primitiva *thrust::reduce_by_key*, utilizando el arreglo *task list* como el arreglo de keys.

$\mathbf{T_6}$. Multiplicación

Cada bloque de CUDA computa un bloque diferente de la matriz resultado. Calcular un bloque de la salida implica procesar todas las tareas asociadas a dicho bloque, por lo que el primer paso que debe ejecutar cada bloque de CUDA es identificar las tareas asociadas al bloque que debe procesar. Luego de la etapa T_5, las tareas asignadas al mismo bloque de la salida se encuentran de forma contigua en el arreglo *task list*. Con esto, es posible determinar las posiciones de la primera y de la última tarea que debe procesar cada bloque de CUDA.

Una vez que se halla el intervalo de la task list a procesar, se itera sobre las tareas, cargando los bloques de las matrices de entrada en memoria compartida, realizando la multiplicación y almacenando los resultados parciales en memoria compartida.

Para calcular las posiciones en el arreglo de bitmaps y en el de desplazamientos, es necesario tener un mapeo entre las ternas (i,j,k) que representan cada tarea y los índices correspondientes en los arreglos de keys de las matrices de entrada. Este mapeo es computado al final de la etapa T_6.

En la función que carga los bloques a memoria compartida, *shmem_load*, los hilos cargan un 0 o un valor no nulo del bloque de entrada de acuerdo al bit que se encuentra en la posición *threadIdx.x* del bitmap. Si el bit es 1, se debe calcular la cantidad de unos previos al mismo en el bitmap, para de esta manera poder calcular el desplazamiento interno al bloque del valor no nulo dentro del arreglo de valores. Esto se implementa utilizando la primitiva *_popcll*, que cuenta el número de bits con valor 1 en una variable de tipo *uint64_t*.

Una vez que los bloques de la entrada son cargados en memoria compartida, cada posición del bloque de la salida es computada por un único hilo, que multiplica la fila y columna correspondiente.

Luego de que todas las tareas hayan sido procesadas, el bitmap del bloque

del resultado se calcula utilizando la primitiva *_ballot_sync*. Cada warp del bloque genera 32 bits del bitmap final. El bitmap generado es utilizado para contar la cantidad de valores no nulos en el bloque, nuevamente utilizando la primitiva *_popcll*. Finalmente, se escribe en memoria global el bitmap, el bloque calculado y su número de elementos.

T_7. Compactación

En la etapa de multiplicación se pueden haber generado valores nulos que luego se escriben en el arreglo de valores de la matriz resultado. Para eliminar estos valores, se utiliza la primitiva *thrust::copy_if* sobre el arreglo de valores generado, que copia a un nuevo arreglo solamente los valores no nulos.

2.4.4.2. Uso de Tensor Cores para la etapa de multiplicación

Hay tres alternativas para programar con Tensor Cores. La más conocida y documentada es la API *WMMA* [3], aunque los desarrolladores también tienen acceso a las instrucciones *wmma* y *mma* del conjunto de instrucciones virtual PTX [9], y a las instrucciones *HMMA* del lenguaje de ensamblaje nativo SASS [108].

La multiplicación de dos matrices usando Tensor Cores requiere de la creación de estructuras de datos especiales, llamadas *fragmentos*, para almacenar las matrices. La interfaz WMMA admite un conjunto limitado de dimensiones para los fragmentos de entrada y salida (llamados acumuladores), dependiendo del tipo de dato de los valores multiplicados. Al momento de realizar este trabajo, los Tensor Cores se encontraban en su tercera generación. Sin embargo, solamente una de las dos plataformas de evaluación posee esta tecnología, por lo que para simplificar la comparación entre ambas plataformas, se utilizan tensor cores de segunda generación. Idealmente se busca utilizar fragmentos con dimensiones compatibles con el tamaño del bloque utilizado por el algoritmo (8×8), pero fragmentos de este tamaño no son soportados en tensor cores de segunda generación. Experimentos no formalizados muestran que la diferencia de rendimiento entre fragmentos de distintos tamaños no es significativa para este problema. En cuanto al tipo de dato utilizado, se utiliza media precisión, dado que los tensor cores de segunda generación no poseen soporte para precisión simple. En la arquitectura Ampere, con los tensor cores de tercera generación, se introdujo soporte para una precisión, llamada *tf32*, que

mantiene los 8 bits de exponente de precisión simple, pero utiliza 10 bits de mantisa, como media precisión. La utilización de esta precisión queda como trabajo futuro. Por estas razones, se utilizan fragmentos de entrada de media precisión de 16×16, con un acumulador de simple precisión del mismo tamaño, una estrategia comúnmente llamada *precisión mixta*.

Las variantes presentadas a continuación, que fueron presentadas e implementadas en un trabajo previo a esta tesis [78], son las que luego se utilizarán en la Sección 3.4.

3. Variante 1, Tensor Core Naive ($\mathbf{TC}_{naive}$)

Esta implementación utiliza el primer sub-bloque de 8×8 de cada fragmento e ignora el resto. A diferencia de la implementación que utiliza CUDA Cores, en la que luego de cargar las matrices en memoria compartida, cada hilo acumula resultados parciales asociados a dos elementos de la salida, en esta versión, esa parte del código es reemplazada por dos llamadas a *load_matrix_sync* y una a *mma_sync*. La primera función carga los datos en los fragmentos asociados a los bloques de la entrada, mientras que la segunda computa la multiplicación. Después de procesar cada task, la función *store_matrix_sync* transfiere los resultados a memoria compartida para luego cargarlos a memoria global.

En el hardware utilizado, los elementos asignados al mismo *lane* pueden estar en el mismo banco de memoria, causando un conflicto de banco en las llamadas a esta función. Para evitar que esto suceda, se agrega un cero (*padding*) a los bloques en memoria compartida para que los valores queden en distintos bancos, evitando conflictos de banco. La API requiere que el *stride* utilizado para cargar valores sea múltiplo de 8, por lo que el mínimo número de elementos que necesitan ser añadidos a cada fila del bloque es 8 (cada fila pasa de 16 a 24 elementos). Un esquema de esta variante se puede ver en la Figura 2.11.

Es importante notar que las funciones accesibles de la interfaz WMMA requieren la colaboración de cada hilo dentro del mismo *warp*, pero la API esconde el mapeo explícito entre los hilos y los valores de la salida.

4. Variante 2, Dos bloques por fragmento ($\mathbf{TC}_{2blk}$)

La variante anterior presenta la desventaja evidente de desperdiciar el 75 % de cada fragmento. Esta variante ataca este problema procesando dos bloques

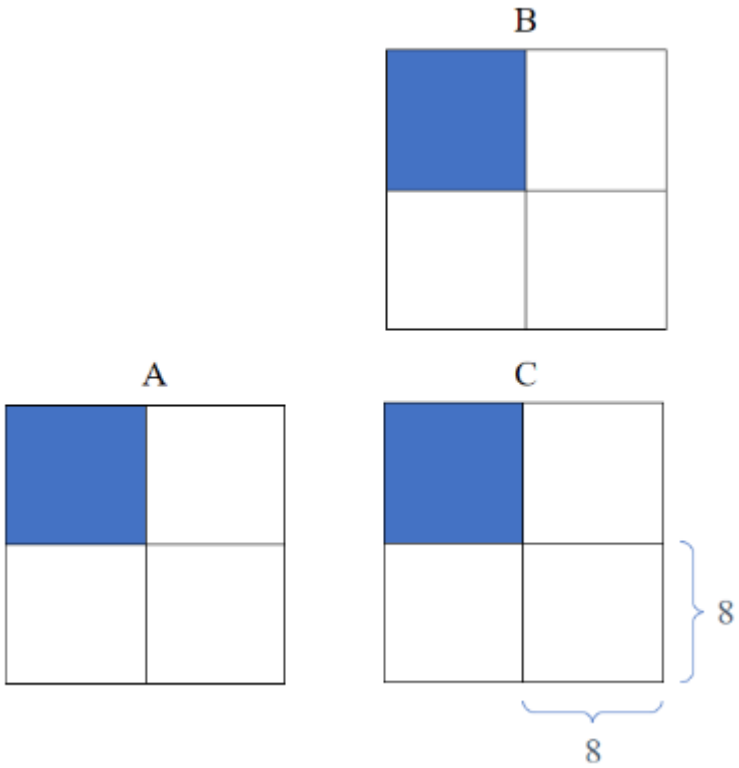

Figura 2.11: Fragmentos y acumulador de tamaño 16×16 involucrados en la multiplicación. Solamente el primer cuadrante de los fragmentos es cargado con valores de un bloque de las matrices A y B, y generan un resultado parcial en el primer cuadrante del acumulador C.

de la salida en paralelo. Los bloques de entrada asociados con el primer bloque de la salida son cargados en el sub-bloque superior izquierdo de 8×8 del fragmento, mientras que el otro par de bloques de entrada son cargados en el sub-bloque inferior derecho, también de 8×8. Esto puede verse en la Figura 2.12. En el caso en que las tasks de uno de los bloques de la salida sean computadas antes que las tasks de otro, las secciones asociadas al bloque que ya terminó su cómputo se llenan con ceros. De esta forma, los resultados parciales del segundo bloque pueden continuar acumulándose sin alterar los resultados del primer bloque.

Además del procesamiento de dos bloques, esta variante se diferencia de TC_{naive} en el uso de la memoria compartida. La única función de la API WMMA para cargar los valores de un bloque en un fragmento requiere que los bloques sean almacenados en un formato denso en memoria compartida o

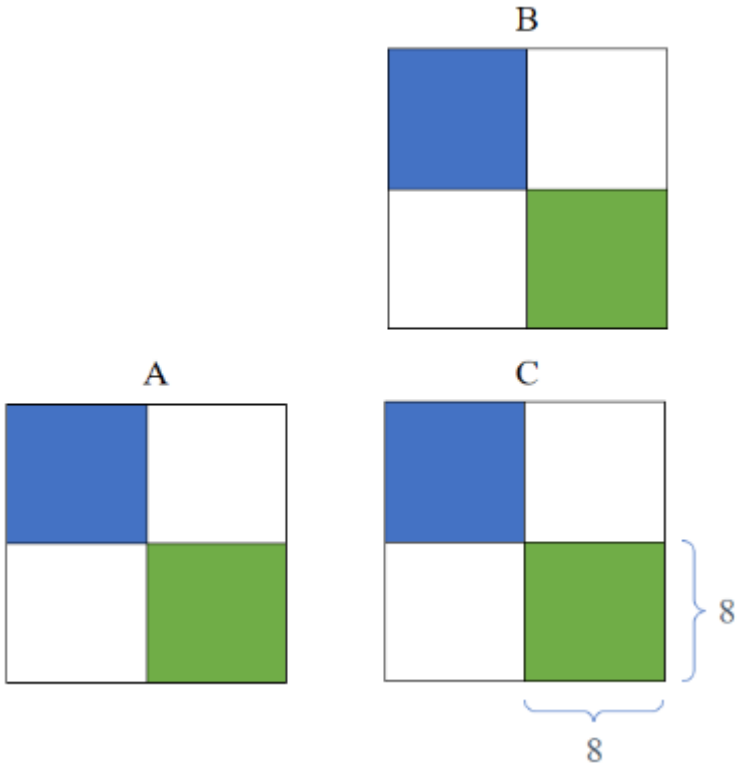

Figura 2.12: Fragmentos y acumulador de tamaño 16×16 involucrados en la multiplicación. El primer y último cuadrante de los fragmentos es cargado con valores de dos bloques distintos de las matrices A y B, y generan un resultado parcial en el primer y último cuadrante del acumulador C. En los fragmentos A y B sub-bloques de colores distintos refieren a bloques distintos dentro de la matriz. Los sub-bloques del mismo color contribuyen el mismo resultado parcial del acumulador C.

global [3]. En este caso, dado que los bloques están en un formato disperso, necesitan ser reconstruidos en un formato denso en memoria compartida antes de la llamada a *load_matrix_sync*. Como el tamaño del fragmento es más grande que el tamaño del bloque utilizado por el algoritmo, la variante anterior malgasta una cantidad importante de ancho de banda de memoria compartida al cargar valores que no son relevantes para el cómputo. Aún sabiendo que la API permite que cada hilo acceda a elementos de su fragmento asociado de forma directa, el principal obstáculo para cargar los valores de forma eficiente es que el mapeo entre los elementos del fragmento y los hilos no está especificado y podría cambiar en futuras arquitecturas de GPUs [3].

Para lograr una forma más eficiente de cargar el bloque disperso al fragmento de WMMA, se determina el mapeo entre hilos y fragmentos de forma

experimental. Este conocimiento hace posible cargar los valores sin necesidad de acceder a memoria compartida y sin malgastar ancho de banda. Es importante notar que el mapeo utilizado en las implementaciones puede depender de la arquitectura.

El procedimiento para determinar el mapeo mencionado consiste en crear un fragmento y almacenar en cada uno de sus elementos (usando accesos directos) el identificador del hilo que hizo la asignación. Posteriormente, se imprime el contenido del fragmento en salida estándar, para poder ver la correspondencia entre las posiciones en memoria compartida y el fragmento generado. Este procedimiento revela que el mapeo para el fragmento que almacena la matriz A (inicializada por el parámetro *matrix_a*) es el mismo que el mapeo para la matriz C (acumulador), pero no el mismo que para la matriz B (*matrix_b*).

Una vez que se conocen las posiciones asignadas a cada hilo, los valores pueden ser obtenidos directamente desde memoria global sin cargar el bloque denso en memoria compartida. Por esta razón, en contraste a la variante TC_{naive}, en esta variante no es necesaria la utilización de las funciones *load_matrix_sync* y *store_matrix_sync*.

5. Variante 3, Dos *tasks* en paralelo (TC_{2tsk})

Esta variante es similar a la anterior, con la diferencia de que los warps no procesan únicamente dos bloques sino también dos tasks de forma paralela para cada bloque, como puede verse en la Figura 2.13. El hecho de procesar dos tasks permite la utilización por completo de los fragmentos de la entrada, y la mitad de los fragmentos de salida, reduciendo de forma significativa el desaprovechamiento del ancho de banda. Esta variante evita el uso de memoria compartida igual que TC_{2blk}.

Cabe destacar que dos sub-bloques de 8×8 (tanto vertical como horizontalmente) dentro del mismo fragmento no son necesariamente contiguos en la matriz original. Esto no afecta la correctitud del algoritmo.

2.5. SpMV

El producto entre una matriz dispersa y un vector, comúnmente denominado SpMV (del inglés *Sparse Matrix Vector*) tiene la siguiente forma:

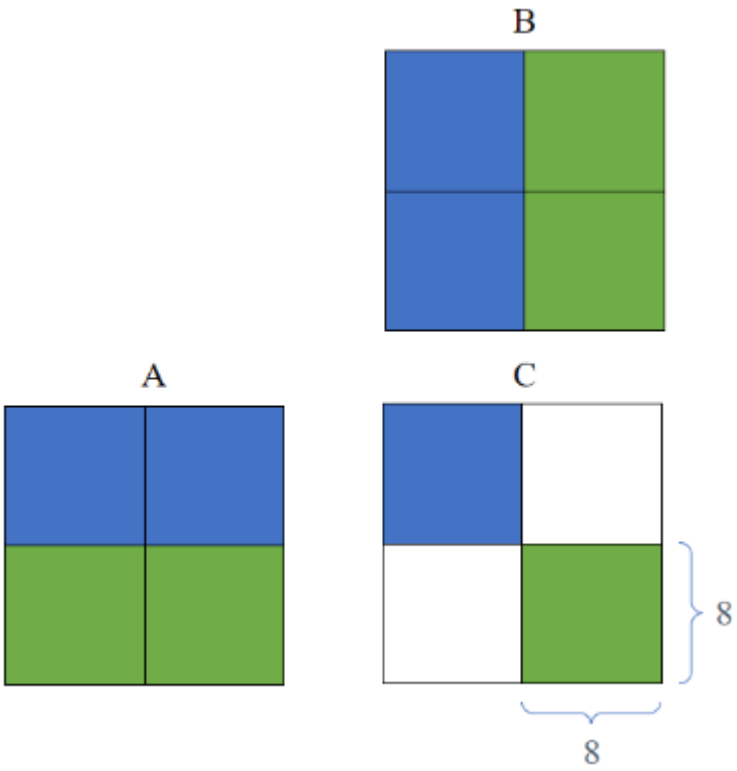

Figura 2.13: Estrategia similar a la presentada en la Figura 2.12. Dos bloques del mismo color en el mismo fragmento (A o B) contribuyen al mismo bloque de salida del acumulador C.

$$y = Ax$$

donde A es una matriz dispersa y los vectores de entrada (x) y de salida (y), son densos.

Esta operación es, posiblemente, la operación fundamental dentro del álgebra dispersa. Esto se debe a que representa gran parte del esfuerzo computacional de implementaciones de métodos iterativos para resolver sistemas de ecuaciones lineales o problemas de valores y vectores propios. Esta operación tiene una baja intensidad computacional, por lo que el desempeño de esta operación se ve limitado por los accesos a memoria, y entonces es crucial la elección de un formato de almacenamiento adecuado. De esta forma, se han propuesto diversos formatos de almacenamiento para optimizar esta operación dentro de contextos específicos. También es importante el diseño inteligente de la implementación sobre el formato elegido de manera de atacar este cuello

Figura 2.14: Kernel SpMV para CSR propuesto por Bell y Garland. Esta variante utiliza un hilo por fila de la matriz.

```
1  __global__ void
2  spmv_csr_scalar_kernel (const int num_rows,
3                          const int* ptr,
4                          const int* indices,
5                          const float* data,
6                          const float* x,
7                                float* y)
8  {
9      int row = blockDim.x*blockIdx.x + threadIdx.x;
10     if (row < num_rows){
11         float dot = 0;
12
13         int row_start = ptr[row];
14         int row_end = ptr[row+1];
15
16         for (int jj = row_start; jj < row_end; jj++)
17             dot += data [jj]*x[indices[jj]];
18
19         y [row] += dot;
20     }
21 }
```

de botella. Por ejemplo, se puede buscar minimizar los accesos de memoria analizando el patrón de dispersión de la matriz.

Dos de los primeros trabajos en proponer implementaciones de esta operación en GPUs fueron presentados por Bell y Garland [22] [23]. En estos se desarrollan implementaciones para algunos de los formatos más comunes, como COO, CSR y ELL, entre otros. La primera implementación descrita para CSR, mostrada en el Algoritmo 2.14, utiliza una estrategia denominada *scalar kernel* por los autores. En este kernel, un único hilo se encarga de procesar una fila entera de la matriz. De esta forma, cada hilo es responsable por un elemento del vector resultado, por lo que todos las escrituras en él serán en posiciones distintas. En esta implementación, dada la asignación de hilos, el paralelismo se ve acotado por la cantidad de filas de la matriz y la fila con mayor cantidad de elementos no nulos se convierte en un cuello de botella para el tiempo de ejecución. Además, dado que cada hilo procesa elementos de filas distintas, se disminuye la cantidad de accesos coalesced a memoria.

En el mismo trabajo se propone otro kernel para CSR, siguiendo una estrategia que denominan *vector kernel*. En este caso, cada fila es procesada por un warp en lugar de un hilo. Esta implementación puede verse en el Algoritmo 2.15. Al terminar de procesar la fila, cada hilo del warp posee resultados parciales del mismo elemento de la salida. Es necesario entonces combinar estos valores a través de una reducción, lo que se puede ver en las líneas 26 a 30, donde la suma de todos los resultados parciales se acumula en el hilo 0 de cada warp. Ese hilo es el que luego, en las líneas 32 y 33 escribe el valor en el vector

Figura 2.15: Kernel SpMV para CSR propuesto por Bell y Garland. Esta variante utiliza un warp por fila de la matriz.

```
1  __global__ void
2  spmv_csr_vector_kernel (const int num_rows,
3                          const int* ptr,
4                          const int* indices,
5                          const float* data,
6                          const float* x,
7                                float* y)
8  {
9      __shared__ float vals[]
10
11     int thread_id = blockDim.x*blockIdx.x + threadIdx.x;
12     int warp_id = thread_id / 32;
13     int lane_id = thread_id & (32-1);
14
15     int row = warp_id;
16
17     if (row < num_rows){
18         int row_start = ptr[row];
19         int row_end = ptr[row+1];
20
21         vals [threadIdx.x] = 0;
22
23         for(int jj = row_start + lane; jj < row_end; jj += 32)
24             vals[threadIdx.x] += data[jj]*x[indices[jj]];
25
26         if (lane < 16) vals[threadIdx.x] += vals[ threadIdx.x + 16];
27         if (lane < 8) vals[threadIdx.x] += vals[threadIdx.x + 8];
28         if (lane < 4) vals[threadIdx.x] += vals[threadIdx.x + 4];
29         if (lane < 2) vals[threadIdx.x] += vals[threadIdx.x + 2];
30         if (lane < 1) vals[threadIdx.x] += vals[threadIdx.x + 1];
31
32         if (lane == 0)
33             y [row] += vals[threadIdx.x];
34     }
35 }
```

de salida. Si bien perdura la limitante en cuanto a la fila con mayor cantidad de elementos, se soluciona el acceso no coalesced que presenta la variante *scalar*. Los kernels SpMV propuestos en el Capítulo 4 siguen una estrategia similar a esta.

Como la principal limitante a la hora de implementar esta operación son los accesos a memoria, los formatos de almacenamiento a bloques para las matrices dispersas pueden usarse para reducir el *working set*, es decir, el conjunto de datos que se manipulan en el contexto de la operación, y así reducir la cantidad de accesos a memoria.

La implementación de SpMV para estos formatos introduce distintas problemáticas. En el caso de formatos con bloques de tamaño fijo, la elección de este tamaño influye fuertemente en el tamaño del working set. La elección de un tamaño no adecuado puede resultar en una enorme cantidad de bloques necesarios para contener todos los elementos no nulos de la matriz. Estos bloques además, es probable que sean muy dispersos, derivando en padding excesivo. Existen diversos trabajos sobre la elección del tamaño óptimo de bloque [104] [68].

Por el lado de los formatos con tamaño de bloque variable, la necesidad de estructuras adicionales para referenciar los valores agrega complejidad a la hora de recuperarlos. En formatos que dividen la matriz en sub-matrices almacenados de distintas formas, se pierde la localidad espacial y temporal entre estas sub-matrices y son necesarias más operaciones para acumular los resultados parciales [67].

La implementación intuitiva de SpMV para BCSR consiste en seguir la misma línea que Bell et al. [22][23] presentaron para CSR. La diferencia es que ahora un hilo (*CSR-scalar*) o warp (*CSR-vector*) procesa una fila de bloques, en lugar de una fila de elementos. Esta idea es explorada por Choi et al. [32], que obtienen un mal desempeño, atribuyéndolo al acceso no coalesced a memoria. Para mitigar esta limitante, los mismos autores proponen optimizaciones como agrupar múltiples bloques de tamaño pequeño en tipos de datos de vectores nativos de CUDA (*float2*, *float3* y *float4*) y agregar padding en las filas de bloques utilizando bloques de ceros.

Una línea similar es transitada por Buatois et al. [28], que también implementan la operación SpMV para BCSR en GPUs NVIDIA y AMD. Para reducir la cantidad de accesos a memoria, los valores del vector de entrada x recuperados son almacenados en registros y reutilizados por las demás filas del mismo bloque. Los vectores de indexación de BCSR son almacenados como arreglos unidimensionales, mientras que el almacenamiento del vector de valores depende del tamaño del bloque. Para bloques de 2×2, también se almacena en un arreglo unidimensional, mientras que para bloques de 4×4, el vector es almacenado en 4 arreglos unidimensionales, cada uno de ellos con un sub-bloque distinto de 2×2. Para recuperar un bloque, entonces, es necesario un acceso en el mismo índice a cada uno de estos 4 arreglos.

En [32] también se presenta una implementación de SpMV para BELL-PACK. Los bloques son almacenados en memoria de forma que asegura que los accesos sean coalesced y el vector de entrada x es almacenado en caché de memoria de textura. Además, se desarrolla un framework para elegir el tamaño de bloque en tiempo de ejecución. Se estima la cantidad de datos generados por un tamaño de bloque específico y se mantienen solamente aquellos tamaños que generen las menores cantidades. Luego, se estima el tiempo de ejecución para los tamaños restantes y se utiliza aquel que obtenga el menor de estos valores.

Para BCCOO/BCCOO+, Yan et al. [110] [112] implementan SpMV a tra-

vés de un kernel con tres etapas. Primero, se leen los arreglos de valores de la matriz y se multiplican con los valores correspondientes del vector de entrada. Luego, se realiza un *segmented scan/sum* utilizando el arreglo de bits de bandera. Finalmente, se reducen los resultados parciales y se escriben en memoria global. En la segunda etapa se busca realizar un balance de carga adecuado haciendo que cada hilo procese la misma cantidad de bloques no nulos consecutivos.

Niu et al. [83] presentan TileSPMV, un algoritmo que divide la matriz dispersa en *tiles* de tamaño fijo (en el trabajo se utiliza 16×16) para luego almacenar cada una de ellas en un formato distinto, de acuerdo a la estructura de la propia *tile*.

Para almacenar una matriz se utiliza una estructura de almacenamiento de dos niveles.

En el primer nivel se guardan tres arreglos:

- Un arreglo *tilePtr*, de tamaño *tilemA*+1, donde *tilemA* es la cantidad de filas de *tiles* de la matriz. En este arreglo se guardan los *offsets* de las distintas filas de *tiles* (índices del primer tile de cada fila).
- Un arreglo *tileColIdx*, de tamaño *numtileA*, donde *numtileA* es la cantidad de *tiles* de la matriz. En este arreglo se guarda el índice de columna para cada *tile*.
- Un arreglo *tileNnz*, de tamaño *numtileA*+1, que guarda el *offset* de la cantidad de valores no nulos de cada *tile* (*tileNnz[i]* es la cantidad de valores no nulos antes del *tile* i).

En el segundo nivel se almacenan los valores no nulos y sus respectivos índices dentro de cada *tile* en distintos formatos. Los formatos utilizados en este trabajo son: CSR, COO, ELL, HYB, *dense*, *dense row* y *dense column*.

Para la multiplicación posterior, se detallan siete implementaciones de *SpMV* a nivel de *warp*, una para cada uno de los formatos utilizados para el almacenamiento de las *tiles*.

Para la elección del formato a utilizar para cada *tile*, se consideran aspectos como la cantidad y la distribución de los valores no nulos del *tile*. En la figura 2.16, extraída del artículo, se muestra un diagrama de flujo que muestra el camino a seguir para la elección del formato. Dado que el artículo menciona que utilizan *tiles* de 16×16, se asume que ese es el tamaño considerado para establecer los números presentes en el diagrama.

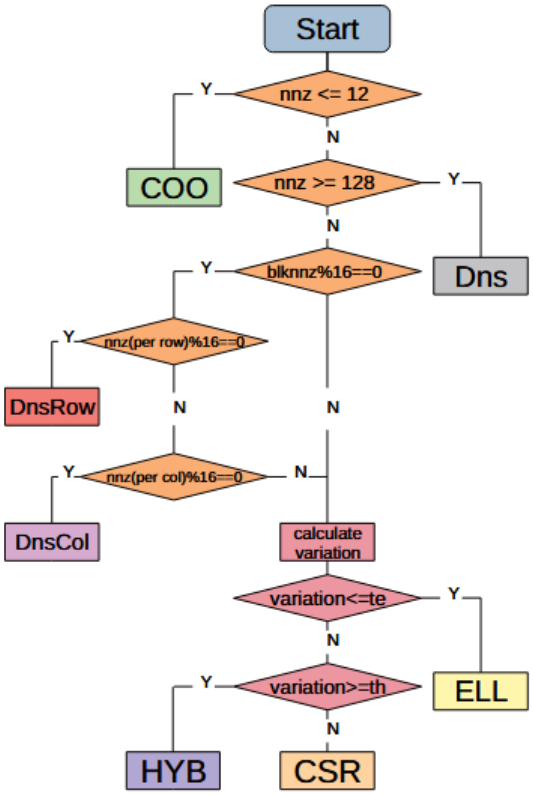

Figura 2.16: Diagrama de flujo para la elección de formato del *tile*. Extraído de [83]

.

Se describen dos métodos adicionales de selección: *TileSpMV_CSR* y *TileSpMV_deferredCOO*. El primero es puramente para comparación en la etapa de evaluación experimental y consiste en almacenar todos los *tiles* en formato *CSR*. El segundo surge de observar que el método *TileSpMV_ADPT* podría no funcionar muy bien para matrices extremadamente dispersas, donde hayan muchas *tiles* almacenadas con el formato *COO*. Este método consiste en tomar todos los *tiles* almacenados en formato *COO* así como la parte almacenada en *COO* de las *tiles* en formato *HYB* y extraerlos para formar una nueva matriz almacenada en formato *CSR* clásico que luego utilizará el método *CSR5-SpMV* para la multiplicación.

Mohammed et al. [80] presentan una taxonomía para la clasificación de estrategias de optimización para SpMV. El uso de formatos, entre ellos los formatos a bloques, para reducir los requerimientos de almacenamiento y los accesos a memoria es descrito como técnicas orientadas al almacenamiento

(*Storage-focused techniques*). Esta categoría también incluye técnicas como balanceo de carga, reutilización de datos y compresión. El uso adecuado de este tipo de técnicas permite mitigar el impacto de los accesos a memoria. Esto implica un peso más importante de la fase de cómputo dentro de la operación SpMV, por lo que toma relevancia la optimización de esta etapa mediante, tal como son llamadas por los mismos autores, técnicas orientadas al cómputo (*Computation-focused techniques*). Se refiere a este trabajo para un listado extensivo en técnicas de optimización de la operación SpMV.

Capítulo 3

Propuestas para SpGeMM

En este capítulo se proponen variantes para mejorar el desempeño de las rutinas existentes de SpGeMM sobre el formato *bmSparse*.

La evaluación preliminar de la implementación *baseline*, presentada en la Sección 2.4.4, indica que las dos etapas que llevan más tiempo de cómputo son el ordenamiento de la *task list* (T_4) y la multiplicación o procesamiento de *tasks* (T_6). Para atacar estos cuellos de botella se realizan tres propuestas. La primera, detallada en el apartado 3.1, trata de una nueva forma de representar la *task list* que permite ahorrar accesos de memoria en la etapa de multiplicación. Luego, en el apartado 3.2, se presenta el uso de un algoritmo alternativo para la etapa de ordenamiento, que se adecua más a la tarea de ordenar la *task list*. Una versión adaptativa que selecciona el algoritmo de ordenamiento a ejecutar en base a las características de la task list es descrita en el apartado 3.3. Por último, en el apartado 3.4 se utiliza la tecnología de *Tensor Cores* de NVIDIA [7] para acelerar la etapa de multiplicación.

3.1. Nueva representación de la task list

En la nomenclatura de *bmSparse*, una *task* se refiere a un par de bloques, uno de la matriz A, y otro de la matriz B, que deben ser multiplicados en el contexto del producto de matrices entre A y B. Hay dos formas de representar una *task*. La primera es utilizando los índices de los dos bloques dentro de sus respectivas matrices. Esto corresponde a la representación que usa ternas (i, j, k). Por otro lado, las *tasks* también pueden ser representadas utilizando los índices de dichos bloques dentro del arreglo de *keys* de *bmSparse*. Esto

corresponde a un par (n, m) donde $(i, j) = A\_keys[n]$ y $(j, k) = B\_keys[m]$.

En la implementación original del algoritmo se utiliza la representación (i, j, k). Esta representación es ventajosa para las etapas que operan con las posiciones de los bloques dentro de sus respectivas matrices, tales como la etapa de ordenamiento (T_4). Sin embargo, en la etapa de multiplicación se necesita acceso a los *bitmaps* y valores correspondientes a los bloques que se está multiplicando, por lo que es necesario recuperar los valores indexados por (n, m) (tales que $(i, j) = A\_keys[n]$ y $(j, k) = B\_keys[m]$) para poder acceder a $A\_bmps[n]$, $B\_bmps[m]$, $A\_values[n]$ y $B\_values[m]$.

Para evitar el *overhead* de estas búsquedas, se propone almacenar la *task list* como los índices n y m desde el principio del algoritmo. Esto implica una implementación menos eficiente de T_4, a favor de ganancias importantes en T_6. En la nueva implementación, se necesita transformar la *task list* al formato (i, j, k) para realizar el ordenamiento de la misma. Este cambio solo implica el acceso a los arreglos A_keys y B_keys antes de comparar dos elementos, y se realiza de forma sencilla utilizando un functor. Una transformación similar es necesaria para la operación $reduce\_by\_key$ de la etapa T_5.

3.2. Segmented sort

El ordenamiento de la etapa T_4 en la versión original se realiza con la función *sort* de la biblioteca *Thrust*. Por como es construida la *task list*, ya se tiene un ordenamiento parcial según los índices i, j y k, en ese orden. Este orden parcial puede ser aprovechado utilizando un ordenamiento por segmento: en lugar de ordenar la *task list* en su totalidad se ordenan independientemente los segmentos de *tasks* contiguas con el mismo valor i. Es importante tener en cuenta que los datos a ordenar se encuentran en memoria de la GPU, por lo que la rutina de ordenamiento tiene que ser ejecutada en dicho dispositivo. De otra forma, el costo de la transferencia de datos entre la CPU y la GPU puede ser mayor que el ahorro relacionado con la utilización de un mejor método de sort.

En este contexto, se propone utilizar la implementación de *segmented sort* para GPUs de [63], dado que es implementada en GPU y maneja de forma distinta segmentos de distinto largo, balanceando la carga de trabajo de cada warp. La utilización de esta rutina requiere crear dos vectores adicionales. Uno, llamado $keys\_d$, contiene los valores según los que se realizará el ordenamiento.

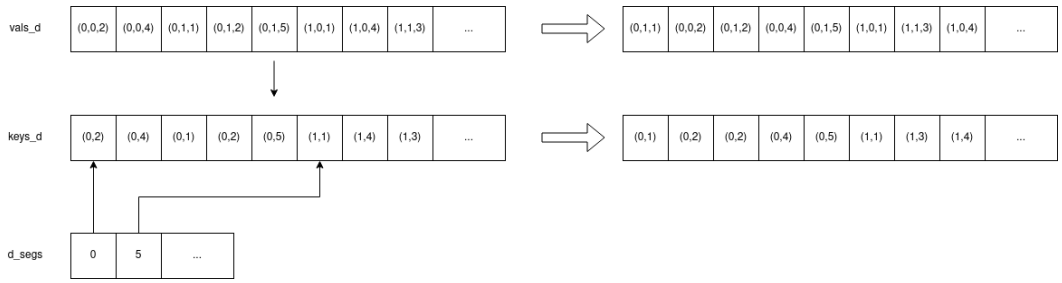

Figura 3.1: Ejemplo de ordenamiento por segmento de la *task list*.

El otro, *d_segs*, contiene los índices del comienzo de cada segmento dentro del vector de valores (la *task list*). El primero es necesario dado que se quiere ordenar la *task list* según (i, k), ignorando el valor de j. Además de los vectores *keys_d* y *d_segs* ya mencionados, la función utilizada, *bb_segsort*, recibe otros tres parámetros. *vals_d* es el vector que se quiere ordenar (en este caso, la *task list*), n, que es el largo de los vectores *keys_d* y *vals_d* y *length*, que es el tamaño del vector de segmentos.

Para *d_segs*, se crea un vector de *Thrust* y se utliza la operación *transform* para almacenar en este solamente los valores i de las ternas (i, j, k) de la *task list*. Luego, se utiliza la operación *reduce_by_key* para obtener la cantidad de veces que cada valor de i aparece en el vector. Finalmente, con una operación *exclusive scan* se obtiene el índice en el que se da la primera ocurrencia de cada valor de i dentro de la *task list*. Las operaciones *transform*, *reduce_by_key* y *exclusive scan* utilizadas son de la biblioteca *thrust*.

Para *keys_d*, se crea otro vector de *thrust* y con una operación *transform* se almacena en él los valores de i y k de cada elemento de la *task list*.

Como la implementación de *segmented sort* utilizada no soporta vectores de *thrust*, es necesario transformar los vectores generados en arreglos estándar de C. Esto añade un *overhead* que igualmente es preferible debido al uso de las operaciones de *thrust* en lugar de operaciones más costosas sobre arreglos estándar de $\mathbb{C}$.

En la Figura 3.1 se puede ver un ejemplo del ordenamiento utilizando este algoritmo. Notar como el vector que se ordena, *keys_d*, es el que marca el orden final del vector de valores *vals_d*. De esta manera, la *task list* queda ordenada según el valor (i, k), por lo que al procesar las *tasks*, se accederá a los bloques de la salida en orden.

Luego de la llamada a la función de ordenamiento, es necesario copiar los valores ordenados a la *task list*. Esto es porque los elementos ordenados están

en el formato (i, j, k), mientras que la *task list* fue transformada en elementos (n, m). Este *overhead* puede ser evitado si se adapta la implementación del *segmented sort* para que trabaje con la representación (n, m) de la *task list*, aunque de esta manera es necesario obtener los valores i y k previo a cada comparación. La etapa T_4 está detallada en el Algoritmo 3.1.

Algoritmo 3.1: Código de la etapa de ordenamiento T_4

```
thrust::device_vector<uint64_t> i_task_list(task_list_size);

auto getFirstElemFromStruct = [=] __device__ (task_list_elem
↪  x)b{return x.first;};

auto getI = [=] __device__ (task_list_elem x) {return (x.first &
↪  0xFFFFFFFF00000000) >> 32;};

thrust::transform(task_list.begin(), task_list.end(),
↪  i_task_list.begin(), getI);

auto end_i = thrust::reduce_by_key(thrust::device,
↪  i_task_list.begin(), i_task_list.begin() + task_list_size,
↪  ones_it, task_keys.begin(), i_task_list.begin());

thrust::transform(task_list.begin(), task_list.end(),
↪  task_keys.begin(), getFirstElemFromStruct);

thrust::exclusive_scan(i_task_list.begin(), end_i.second +
↪  1,        i_task_list.begin());

int m = std::distance(i_task_list.begin(), end_i.second + 1);

uint64_t *key_d;
task_list_elem *val_d;
int *seg_d;

cudaMalloc((void**)&key_d, sizeof(uint64_t)*task_list_size);
```

```
22  cudaMalloc((void**)&val_d,
    ↪  sizeof(task_list_elem)*task_list_size);
23  cudaMalloc((void**)&seg_d, sizeof(int)*m);
24
25  thrust::device_ptr<uint64_t> key_ptr(key_d);
26  thrust::copy(task_keys.begin(), task_keys.end(), key_ptr);
27
28  thrust::device_ptr<task_list_elem> val_ptr(val_d);
29  thrust::copy(task_list.begin(), task_list.end(), val_ptr);
30
31  thrust::device_ptr<int> seg_ptr(seg_d);
32  thrust::copy(i_task_list.begin(), i_task_list.begin() + m,
    ↪  seg_ptr);
33
34  bb_segsort(key_d, val_d, task_list_size, seg_d, m);
35
36  thrust::copy(val_ptr, val_ptr + task_list_size,
    ↪  task_list.begin());
37
38  }
```

La implementación detallada en el Algoritmo 3.1 fue la utilizada en la evaluación experimental. Desde entonces, se ha logrado evitar el uso de las estructuras *key_ d*, *val_ d* y *seg_ d* (líneas 67, 68 y 69), y por lo tanto también la necesidad de reservas de memoria y copias de vectores, a través del uso de la función *raw_ pointer_ cast* de Thrust. La evaluación del impacto de este cambio en el tiempo de ejecución del algoritmo queda pendiente. Sin embargo, es esperable que sea considerablemente pequeño y puede estimarse observando los tiempos de las etapas de copia en la Tabla 3.5.

3.3. Sort adaptativo

Los resultados obtenidos de evaluar el uso de segmented sort para el ordenamiento de la task list (ver Sección 3.5.1) indican que la nueva estrategia de ordenamiento no siempre resulta en un mejor desempeño que si se hubiera ordenado la *task list* con un ordenamiento convencional. Concretamente, el

segmented sort obtiene mejores resultados para matrices grandes, mientras que el método *sort* de la biblioteca *Thrust* es el de mejor desempeño en matrices pequeñas (por ejemplo, *cryg10000*). Este comportamiento es esperado dado que, para matrices grandes, la ganancia obtenida al explotar el orden parcial de la *task list* compensa el *overhead* causado por las operaciones preliminares necesarias para el uso del *segmented sort*. Para matrices en las que la etapa de ordenamiento tiene bajo costo computacional, el costo de las reservas y copias de memoria requerido son superiores a la mejora en el ordenamiento.

Considerando esto, se busca optimizar aún más esta etapa haciendo una elección adaptativa del algoritmo de ordenamiento a utilizar, dependiendo de las características del problema. Es necesario entonces caracterizar el costo computacional de esta etapa, analizando las matrices y estructuras intermedias para determinar qué características influyen de mayor manera en el tiempo de ejecución.

El análisis de la etapa de ordenamiento reveló que el número de filas, bloques y valores no nulos se correlaciona levemente con la cantidad de trabajo. En cambio, por más que esta etapa puede también estar influenciada por accesos a caché y el ordenamiento de las tareas previo al *sort*, el tamaño de la *task list* está directamente relacionado a la cantidad de trabajo a realizar.

La estrategia utilizada consiste en definir un umbral para el tamaño de la *task list*, ordenando las *task lists* que superen dicho umbral con el sort por segmentos y el sort convencional para el resto.

3.4. Tensor Cores

Para las distintas implementaciones de SpGeMM para *bmSparse* se utilizaron CUDA cores para computar las operaciones aritméticas requeridas en la multiplicación de dos bloques. En esta sección se presenta el uso de *Tensor Cores* para acelerar la multiplicación de bloques.

Se toman las variantes presentadas en la Sección 2.4.4.2 y se adapta el kernel de multiplicación de la etapa T_4 de la versión base de SpGeMM (SpGeMM$_{BL}$) para el uso de Tensor Cores. El resto de las etapas y sus respectivas implementaciones se mantienen sin cambios. También se unificó las implementaciones adaptadas en una misma aplicación parametrizable.

La variante que integra todas las mejoras previas pero utiliza CUDA Cores para la multiplicación se denomina CC$_{final}$, para distinguirla de las variantes

que utilizan Tensor Cores.

3.5. Evaluación experimental

La Tabla 3.1 detalla las plataformas utilizadas en la evaluación experimental efectuada. Los experimentos de la Sección 3.5.1 fueron realizados utilizando la Plataforma 1, que está conectada a un acelerador NVIDIA GeForce RTX 2080 Ti, de arquitectura Turing, con 4352 CUDA Cores, 11GB de RAM y un bus de memoria de 352 bits. A partir de la Sección 3.5.2 se utilizan las Plataformas 2 y 3. La Plataforma 2 contiene una GPU GeForce RTX 2080 Ti, de la arquitectura Turing, mientras que la Plataforma 3 tiene una GPU GeForce RTX 3090 Ti, de la arquitectura Ampere. En todos los casos se utilizó la versión 11.4 de CUDA *runtime* y Toolkit (que incluye las bibliotecas *thrust* y cuSparse). El uso de distintas plataformas de evaluación para las distintas secciones se debe a que parte del hardware, como la tarjeta RTX 3090 Ti, fue adquirido en el transcurso del trabajo.

Tabla 3.1: Plataformas de hardware utilizadas en la evaluación.

	Plataforma 1	Plataforma 2	Plataforma 3
Procesador	Intel Core i7-6700@3.40GHz	Intel Core i7-9750H@2.60Hz	Intel Core i7-9750H@2.60GHz
RAM	64 GB	64 GB	64 GB
GPU	NVIDIA GeForce RTX 2080 Ti	NVIDIA GeForce RTX 2080 Ti	NVIDIA GeForce RTX 3090 Ti
CUDA	11.4	11.4	11.4

Las matrices dispersas utilizadas en la evaluación fueron obtenidas de la *Suite Sparse Matrix Collection* [37]. Se eligieron 9 matrices cuadradas de diferentes características. Particularmente, se seleccionaron tres grupos de matrices de pequeño, mediano y gran tamaño: las primeras 3 tienen una dimensión cercana a 10^4, las matrices de la 4 a la 6 tienen una dimensión cercana a 10^5 y las últimas 3 tienen dimensión cercana a 10^6. Las matrices seleccionadas almacenan valores en punto flotante de precisión simple (*floats*). Las características de cada matriz son presentadas en la Tabla 3.2. En la Sección 3.5.4 se extiende este conjunto con más matrices de la colección de *Suite Sparse* para una evaluación más exhaustiva al comparar con implementaciones de SpGeMM de bibliotecas conocidas.

Tabla 3.2: Características principales de las matrices seleccionadas para la evaluación experimental.

Nombre	Id.	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

3.5.1. Evaluación de la nueva representación y nuevo ordenamiento

El primer experimento se centra en evaluar el efecto de cambiar la representación original de la *task list* por la descrita en la Sección 2.4.4. Para esto, se realiza el cuadrado de cada matriz seleccionada a través de la implementación base (SpGeMM$_{BL}$) y la que utiliza la nueva representación de la *task list* (SpGeMM$_{NI}$). En la Tabla 3.3 se presenta un resumen de los resultados, incluyendo como referencia el tiempo de ejecución obtenido por el *kernel* correspondiente de la biblioteca cuSparse (csrgemm), que utiliza el formato CSR para almacenar las matrices. Los tiempos presentados son el promedio de 5 ejecuciones.

Tabla 3.3: Tiempo de ejecución (en ms) de la operación para las variantes SpGeMM$_{BL}$, SpGeMM$_{NI}$ y csrgemm.

Matriz	SpGeMM$_{BL}$	SpGeMM$_{NI}$	csrgemm
cryg10000	2.456	1.353	160.760
Goodwin_030	7.209	3.646	161.272
ted_A_unscaled	4.505	2.608	160.542
Goodwin_095	46.048	24.468	171.225
matrix_9	47.017	20.604	275.346
hcircuit	33.566	18.346	183.093
webbase-1M	102.830	54.593	572.239
t2em	45.829	23.388	176.839
atmosmodd	157.942	82.081	238.031

Los tiempos obtenidos indican que ambas variantes basadas en *bmSpar-*

se presentan una mejoría importante en comparación a la implementación de cuSparse. Adicionalmente, comparando ambas versiones de *bmSparse*, se puede notar que la nueva propuesta ofrece reducciones importantes en cuanto a tiempo de ejecución. Particularmente, el kernel SpGeMM$_{NI}$ alcanza *speedups* de hasta 2× sobre SpGeMM$_{BL}$.

Siguiendo con la evaluación, la Tabla 3.4 presenta el porcentaje de tiempo de ejecución que conlleva cada una de las partes destacables de la variante SpGeMM$_{NI}$. Los resultados confirman que las etapas de multiplicación y ordenamiento resultan los dos cuellos de botella más grandes a nivel de cómputo. La etapa de multiplicación compone más de un 50 % del tiempo de ejecución para la mayoría de las matrices. Por otra parte, para varias de las matrices, la etapa de ordenamiento conlleva más tiempo que todas las demás etapas juntas. El costo de esta etapa parece ser especialmente crítico para casos en los que la *task list* es grande, lo cual es esperable para las matrices más grandes del conjunto seleccionado. Estos resultados enfatizan la importancia de mejorar el rendimiento de esta etapa.

Tabla 3.4: Porcentaje de tiempo de ejecución de cada una de las etapas principales de la variante SpGeMM$_{NI}$.

Matriz	Sort	Multiplicación	Otras etapas
cryg10000	23 %	25 %	51 %
Goodwin_030	24 %	42 %	33 %
ted_A_unscaled	21 %	33 %	44 %
Goodwin_095	24 %	56 %	19 %
matrix_9	24 %	54 %	21 %
hcircuit	22 %	56 %	20 %
webbase-1M	24 %	58 %	17 %
t2em	22 %	53 %	24 %
atmosmodd	25 %	55 %	19 %

La Figura 3.2 compara los tiempos de ejecución de la etapa de ordenamiento para la variante SpGeMM$_{NI}$ con la implementación de la misma etapa utilizando *segmented sort*. Se puede observar que la utilización del *segmented sort* resulta en una mejora considerable, especialmente para los casos donde esta etapa consume más tiempo de cómputo, alcanzando aceleraciones de hasta 2× para las instancias evaluadas.

Finalmente, se analizan las diferentes sub-etapas de T_4 de la implementación propuesta en la Tabla 3.5. Como es esperado, el *segmented sort* es la

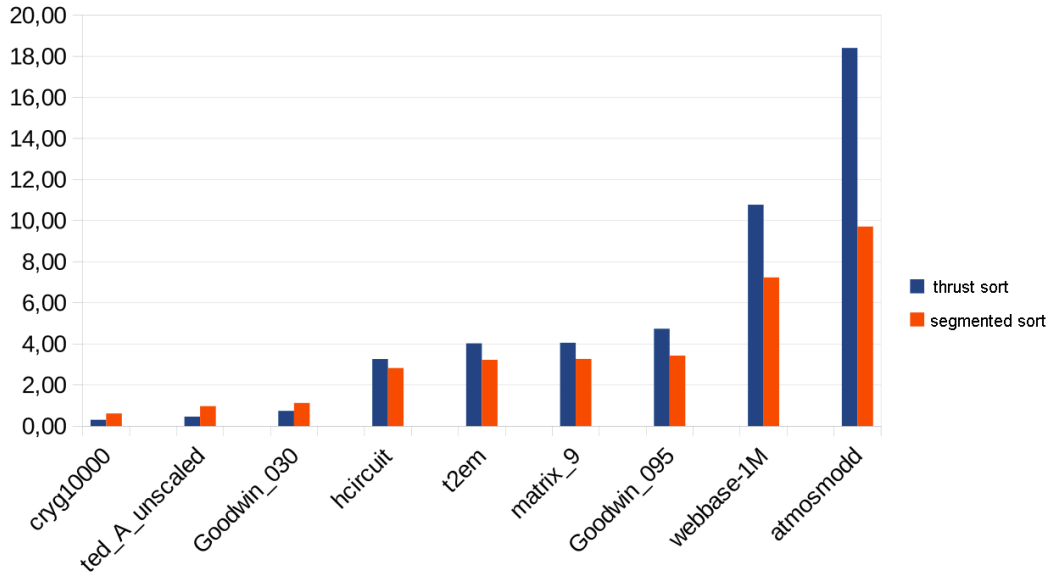

Figura 3.2: Tiempos de ejecución de la etapa T_4 (ordenamiento) usando *thrust sort* y la implementación con *segmented sort*. La desviación estándar es despreciable. Los casos de evaluación están en orden ascendente de tiempo de ejecución del *sort* original.

operación que concentra la mayor parte del tiempo de ejecución. Sin embargo, el tiempo requerido por las etapas de configuración no es despreciable, especialmente para reserva y copias de memoria. Por lo tanto, sería razonable optimizar la implementación del *segmented sort* buscando mitigar este *overhead*.

3.5.2. Evaluación de la estrategia adaptativa de ordenamiento

A continuación se presenta el estudio de la estrategia adaptativa de ordenamiento de la *task list*. En la Figura 3.3 se compara el tiempo de ejecución del *segmented sort* con la estrategia adaptativa. Específicamente, se computa la etapa de ordenamiento sobre ambas plataformas de evalaución para las nueve matrices, normalizando el tiempo de ejecución de ambas estrategias con respecto a la más lenta de las dos.

Tabla 3.5: Tiempo de ejecución (en ms) de las sub-etapas de T_4 en la nueva implementación.

Etapa	vect. creation	transform 1	reduce_by_key	transform 2	exclusive_scan	malloc	copy	seg. sort	copy	total
cryg10000	0.02	0.03	0.04	0.02	0.04	0.13	0.05	0.37	0.02	0.71
Goodwin_030	0.12	0.04	0.05	0.04	0.03	0.21	0.08	0.67	0.03	1.27
ted_A_unscaled	0.12	0.02	0.06	0.02	0.05	0.18	0.05	0.63	0.02	1.15
Goodwin_095	0.18	0.17	0.16	0.17	0.05	0.26	0.33	2.12	0.23	3.66
matrix_9	0.19	0.16	0.14	0.15	0.03	0.21	0.32	2.09	0.22	3.53
hcircuit	0.17	0.12	0.12	0.11	0.04	0.25	0.23	1.83	0.15	3.01
webbase-1M	0.29	0.35	0.39	0.36	0.18	0.41	0.71	4.51	0.49	7.69
t2em	0.16	0.16	0.29	0.15	0.22	0.29	0.29	1.77	0.20	3.53
atmosmodd	0.38	0.58	0.53	0.61	0.19	0.49	1.14	5.85	0.81	10.58

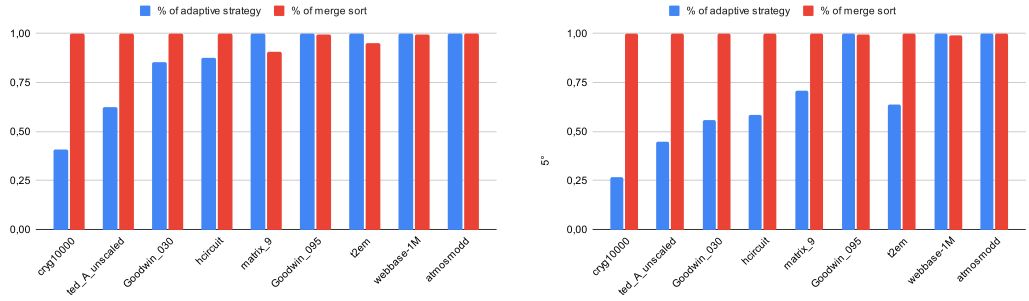

Figura 3.3: Comparación de rendimiento entre el *segmented sort* y la estrategia adaptativa. Para cada matriz las barras representan el porcentaje de tiempo de ejecución de la variante correspondiente respecto a la más lenta de las dos. El gráfico de la izquierda corresponde a la plataforma de evaluación con la GPU RTX 2080 Ti, mientras que el de la derecha corresponde a la plataforma con GPU RTX 3090 Ti.

Ambos gráficos en la Figura 3.3 muestran que la estrategia adaptativa obtiene, en general, menores tiempos de ejecución que la propuesta previa. La estrategia adaptativa obtiene mejores desempeños que el *segmented sort*, principalmente en matrices pequeñas. Además, los resultados de la estrategia adaptativa son generalmente mejores para la GPU RTX 3090 Ti. En la plataforma con este acelerador, la nueva estrategia es hasta 3× más rápida que utilizar *segmented sort* de forma directa.

Solamente en dos casos, para la tarjeta RTX 2080 Ti, se obtienen mejores resultados con la estrategia anterior, lo cual indica que el nuevo método falla en predecir cual es la estrategia de ordenamiento óptima en algunos casos. La diferencia en tiempos de ejecución en los demás casos en los que la estrategia adaptativa resulta más lenta es despreciable.

3.5.3. Evaluación del uso de Tensor Cores

Para la evaluación de las distintas versiones que utilizan Tensor Cores para acelerar la etapa de multiplicación, se ejecuta el kernel de SpGEMM para cada

implementación, para todas las matrices en ambas plataformas de evaluación. Se comparan las tres variantes (TC_{naive}, TC_{2blk} y TC_{2tsk}) con la mejor variante que utiliza CUDA Cores (CC_{final}).

En la Tabla 3.6 se presentan los resultados obtenidos, distinguiendo los tiempos de ejecución de la etapa de multiplicación. Se muestra también el tiempo total de ejecución de la operación.

Los resultados experimentales pueden ser analizados desde diferentes perspectivas. Como primera observación, las nuevas versiones basadas en Tensor Cores obtienen mejores resultados que la versión que utiliza CUDA Cores para todas las matrices y para ambas plataformas de evaluación. Luego, se puede observar que no es trivial identificar cuál variante basada en Tensor Cores es la mejor. Las diferencias en tiempos de ejecución para cada matriz es pequeña para estas variantes, pero no es posible identificar una que obtenga un mejor desempeño sobre las demás para todos los casos. Considerando el tamaño de las matrices, es notable que la variante TC_{naive} es mejor para los casos pequeños mientras que las variantes TC_{2blk} y TC_{2tsk} es mejor para los casos más grandes. Además, la variante TC_{2tsk} suele obtener mejores resultados que TC_{2blk}, especialmente para la plataforma con la tarjeta RTX 3090 Ti.

Centrándose en las plataformas de hardware sobre las que se realizó la evaluación, se puede concluir que la tarjeta NVIDIA RTX 3090 Ti obtiene mejores rendimientos al computar la etapa de multiplicación que la tarjeta NVIDIA RTX 2080 Ti para todos los casos de evaluación y para todas las variantes evaluadas. Otro aspecto notable es que la diferencia entre los tiempos de ejecución de la etapa T_6 en ambas GPUs es más significativa que la diferencia entre los tiempos totales de la operación SpGeMM. Por ejemplo, en la matriz *atmosmodd*, la relación entre ambas plataformas para la versión TC_{2blk} es $1,68$, comparado con solamente $1,22$ para la operación entera. Los resultados obtenidos están alineados con el desempeño esperado del hardware. Es posible que el trabajo realizado no abarque la totalidad de los Tensor Cores disponibles en la tarjeta RTX 2080 Ti, y debido a esto la implementación propuesta puede beneficiarse de los menos, pero más potentes, Tensor Cores de la arquitectura Ampere presente en la tarjeta RTX 3090 Ti. Más aún, la RTX 3090 Ti ofrece $1,5\times$ más ancho de banda de memoria que la RTX 2080 Ti, lo cual es un factor esencial en el desempeño de códigos dispersos en general.

Por otro lado, es interesante remarcar que la diferencia de tiempo de ejecución para la etapa S_{mult} entre la mejor variante de Tensor Cores y la versión

Tabla 3.6: Tiempo de ejecución (en ms) de la etapa de multiplicación y tiempo de ejecución total de SpGeMM de cada variante que utiliza Tensor Cores

Matriz	**Versión**	**RTX 2080**		**RTX 3090**	
		S_{mult}	**Total**	S_{mult}	**Total**
cryg10000	TC_{naive}	0,03	0,90	0,02	0,89
	TC_{2blk}	0,03	0,91	0,02	0,89
	TC_{2tsk}	0,03	0,92	0,02	0,91
	CC_{final}	0,04	0,94	0,03	0,90
Goodwin_030	TC_{naive}	0,10	1,68	0,06	1,53
	TC_{2blk}	0,11	1,68	0,07	1,52
	TC_{2tsk}	0,11	1,68	0,07	1,52
	CC_{final}	0,13	1,72	0,08	1,53
ted_A_unscaled	TC_{naive}	0,14	2,44	0,09	2,12
	TC_{2blk}	0,15	2,47	0,09	2,12
	TC_{2tsk}	0,16	2,46	0,09	2,11
	CC_{final}	0,21	2,51	0,13	2,16
Goodwin_095	TC_{naive}	1,43	8,58	0,96	6,83
	TC_{2blk}	1,25	8,41	0,77	6,67
	TC_{2tsk}	1,27	8,40	0,77	6,65
	CC_{final}	1,63	8,79	1,08	6,94
matrix_9	TC_{naive}	1,80	11,19	1,35	9,56
	TC_{2blk}	1,54	10,90	1,15	9,48
	TC_{2tsk}	1,56	10,96	1,14	9,49
	CC_{final}	2,07	11,46	1,56	9,85
hcircuit	TC_{naive}	1,34	11,46	0,77	9,88
	TC_{2blk}	1,43	11,60	0,81	9,91
	TC_{2tsk}	1,44	11,57	0,78	9,86
	CC_{final}	2,03	12,14	1,17	10,29
webbase-1M	TC_{naive}	1,96	14,68	1,26	11,89
	TC_{2blk}	1,74	14,48	1,07	11,67
	TC_{2tsk}	1,77	14,21	1,03	11,52
	CC_{final}	2,39	15,08	1,50	12,05
t2em	TC_{naive}	3,83	20,80	2,46	17,69
	TC_{2blk}	3,42	20,52	2,18	17,50
	TC_{2tsk}	3,46	20,78	2,12	17,42
	CC_{final}	4,56	21,62	2,85	18,20
atmosmodd	TC_{naive}	6,49	34,01	4,11	27,67
	TC_{2blk}	6,01	33,59	3,57	27,43
	TC_{2tsk}	6,10	33,64	3,47	27,28
	CC_{final}	8,25	35,80	5,09	28,97

CC_{final} crece con la dimensión de la matriz evaluada (o, más precisamente, con el costo computacional implícito). Por ejemplo, para la GPU RTX 3090 Ti, la brecha para la matriz *Goodwin_030* es de $1,33$, mientras que para la matriz *atmosmodd* este valor sube hasta casi $1,47$. Además, estas brechas son más notorias para la tarjeta RTX 3090 Ti, donde el tiempo de ejecución de S_{mult} se reduce casi a la mitad.

3.5.4. Evaluación general

Luego de estudiar cada propuesta de manera individual, se realiza una evaluación sobre una nueva variante, SpGeMM$_{FN}$, que incluye todas las mejoras presentadas en este capítulo. Primeramente, se compara esta versión con la implementación original presentada en la sección 2.4.4.1, que sigue los pasos presentados por Zhang y Gruenwald (SpGeMM$_{BL}$). En la Tabla 3.7 se presentan los tiempos de ejecución para estas versiones utilizando ambas plataformas de evaluación.

Tabla 3.7: Tiempos de ejecución (en ms) de la implementación del algoritmo original descrita en [111] (SpGeMM$_{BL}$), y la variante que integra todas las extensiones propuestas (SpGeMM$_{FN}$), en ambas plataformas de cómputo.

		Matriz								
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
RTX 2080 Ti	**Tiempo** SpGeMM$_{BL}$	2,34	4,47	6,82	33,17	38,44	45,99	45,26	102,73	154,94
	Tiempo SpGeMM$_{FN}$	0,92	1,68	2,45	8,39	10,93	11,59	14,38	20,43	33,55
	Rendimiento relativo	2,5×	2,7×	2,8×	4,0×	3,5×	4,0×	3,1×	5,0×	4,6×
RTX 3090 Ti	**Tiempo** SpGeMM$_{BL}$	2,54	4,29	6,03	25,78	33,59	34,99	35,21	75,19	115,46
	Tiempo SpGeMM$_{FN}$	0,89	1,54	2,15	6,67	9,48	9,88	11,66	17,50	27,22
	Rendimiento relativo	2,8×	2,8×	2,8×	3,9×	3,5×	3,5×	3,0×	4,3×	4,2×

Los resultados reafirman que la nueva propuesta obtiene resultados significativamente mejores que la versión original del algoritmo. Además, los beneficios obtenidos crecen con el costo computacional asociado a los casos de evaluación.

Las diferencias entre tiempos de ejecución son considerables incluso cuando se comparan con la versión que integra las mejoras presentadas en las Secciones 3.1 y 3.2 (SpGeMM$_{NI}$). En la Tabla 3.8 se presenta esta comparación. Nuevamente, la brecha entre ambas versiones es significativa. Más precisamente, estas diferencias van entre $1,4\times$ y $1,7\times$.

Luego de la evaluación de la nueva propuesta con las variantes SpGeMM$_{BL}$ y SpGeMM$_{NI}$, se compara el rendimiento computacional de SpGeMM$_{FN}$ con

Tabla 3.8: Tiempo de ejecución (en ms) de la versión que integra las mejoras de las Secciones 3.1 y 3.2 (SpGeMM$_{NI}$) y la variante que integra todas las extensiones propuestas (SpGeMM$_{FN}$), en ambas plataformas de cómputo.

		Matriz								
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
RTX 2080 Ti	**Tiempo** SpGeMM$_{NI}$	1,31	2,27	3,32	12,47	15,93	17,82	20,21	31,96	55,58
	Tiempo SpGeMM$_{FN}$	0,92	1,68	2,457	8,39	10,93	11,59	14,38	20,43	33,55
	Rendimiento relativo	1,4×	1,4×	1,4×	1,5×	1,5×	1,5×	1,4×	1,6×	1,7×
RTX 3090 Ti	**Tiempo** SpGeMM$_{NI}$	1,29	2,21	3,13	11,05	14,89	15,46	17,85	27,63	46,30
	Tiempo SpGeMM$_{FN}$	0,89	1,54	2,15	6,67	9,48	9,88	11,66	17,50	27,22
	Rendimiento relativo	1,4×	1,4×	1,5×	1,7×	1,6×	1,6×	1,5×	1,6×	1,7×

implementaciones de SpGeMM disponibles en otras bibliotecas. Particularmente, se utiliza la biblioteca MKL (con dos formatos distintos de almacenamiento de matrices dispersas, CSR y BSR con bloques de 2×2) y la biblioteca cuSparse de NVIDIA. Específicamente, la Tabla 3.9 incluye el tiempo de ejecución total de la operación SpGeMM para el conjunto de matrices de evaluación, solamente utilizando la plataforma que contiene la tarjeta RTX 2080 Ti, donde la CPU y la GPU presentan fechas de lanzamiento y gama más similares.

Tabla 3.9: Tiempos de ejecución (en ms) de la operación SpGeMM para la implementación propuesta (SpGeMM$_{FN}$), la implementación de MKL utilizando los formatos CSR y BSR, y la implementación de cuSparse para CSR.

	Matriz								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Time SpGeMM$_{FN}$	0.92	1.68	2.45	8.39	10.93	11.59	14.38	20.43	33.56
Time MKL - CSR	2.00	10.95	7.52	10.76	52.93	60.48	22.96	77.13	54.72
Time MKL - BSR	0.31	6.78	4.62	7.88	45.26	44.44	18.71	65.14	46.26
Time cuSparse	164.93	166.20	162.69	179.55	289.76	187.03	180.23	581.99	250.44

La variante SpGeMM$_{FN}$ obtiene un mejor desempeño que la implementación de MKL para CSR en todos los casos de evaluación, con aceleraciones entre $1,3\times$ y $7\times$. Los speedups de SpGeMM$_{FN}$ en comparación a la implementación de MKL para BSR son más moderados, pero superiores en la mayoría de los casos. Es importante mencionar que no se incluyeron resultados para otros tamaños de bloque para el formato BSR debido a que este formato presentó un desempeño considerablemente menor en experimentos cuyos resultados se omiten por brevedad. Finalmente, los resultados muestran que la implementación de cuSparse no es competitiva con ninguna de las demás variantes.

En la Figura 3.4 se presenta una comparación de la variante SpGeMM$_{FN}$ con las implementaciones de MKL para un conjunto más grande de matrices. Se tomaron al azar 112 matrices cuadradas con más de 100.000 filas y columnas de

la colección *Suite Sparse*. La variante SpGeMM$_{FN}$ obtiene valores positivos de aceleración en comparación a la implementación de MKL para CSR. En cuanto a MKL para BSR, SpGeMM$_{FN}$ logra resultados competitivos, especialmente para matrices grandes.

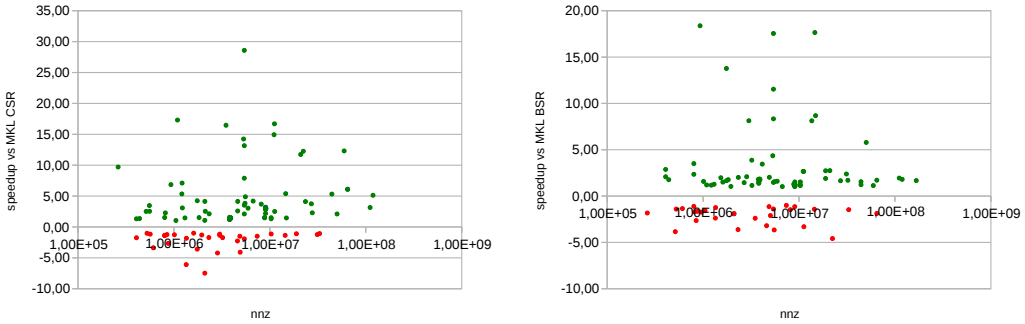

Figura 3.4: Aceleración obtenida por la variante SpGeMM$_{FN}$ con respecto a las implementaciones de MKL para CSR y BSR. Valores positivos indican que SpGeMM$_{FN}$ es favorable.

Capítulo 4

Propuestas para SpMV

Si bien el formato *bmSparse* fue diseñado para lograr una mayor eficiencia en la operación SpGeMM, tiene el potencial de ser un formato de almacenamiento disperso de propósito general. En particular, *bmSparse* puede ofrecer importantes ahorros en transferencias de datos y espacio de almacenamiento, lo cual es crucial para lograr un buen desempeño en operaciones limitadas por memoria, como lo son las operaciones con matrices dispersas.

Como se mencionó anteriormente, una de las principales operaciones de álgebra dispersa es la multiplicación de matriz dispersa por vector (SpMV por su sigla en inglés). Esta característica moriva a estudiar la implementación de la operación SpMV con formatos a bloques.

En los siguientes apartados se describen las propuestas para optimizar el uso de *bmSparse* en el marco de la operación SpMV. Específicamente se incluyen dos variantes y se resume la evaluación experimental asociada.

4.1. Primera variante, agrupamiento por filas (SpMV_{BASE})

Una forma intuitiva de paralelizar la operación SpMV usando el formato *bmSparse* es procesar cada bloque de manera independiente, multiplicándolo por el fragmento del vector de entrada correspondiente y sumando los resultados al vector de salida. Sin embargo, esto implica el uso de operaciones atómicas al escribir sobre el vector de salida para evitar condiciones de carrera. Esto puede resultar en un importante cuello de botella dado que una gran cantidad de hilos accederán a la misma posición del vector de salida, y su ejecución se

verá bloqueada hasta que pueda acceder a este recurso compartido.

Para evitar el uso de operaciones atómicas, una alternativa consiste en que cada fila de bloques de la matriz sea procesada por un único bloque de hilos, habiendo así una única escritura en cada entrada del vector de salida. Para llevar a cabo esta implementación es crucial tener un acceso eficiente al comienzo de cada fila de bloques. *bmSparse* no contiene una estructura que mantenga esta información, por lo que la propuesta incluye una etapa de preprocesamiento que, a partir del vector *keys* de *bmSparse*, genera un vector *bl_rw_ptr* que contiene el índice del primer bloque de cada fila.

Este proceso se realiza en tres pasos. Primero, se genera un vector de unos de igual tamaño que la cantidad total de bloques de la matriz. Luego, se realiza una operación *reduce_by_key* de *Thrust*, usando el índice de la fila del bloque como la *key*. Esto genera un vector donde cada entrada contiene la cantidad de bloques de cada fila de bloques. Por último, se realiza una operación *scan*, también de *Thrust* para obtener un vector donde cada entrada contiene el índice del comienzo de cada fila de bloques.

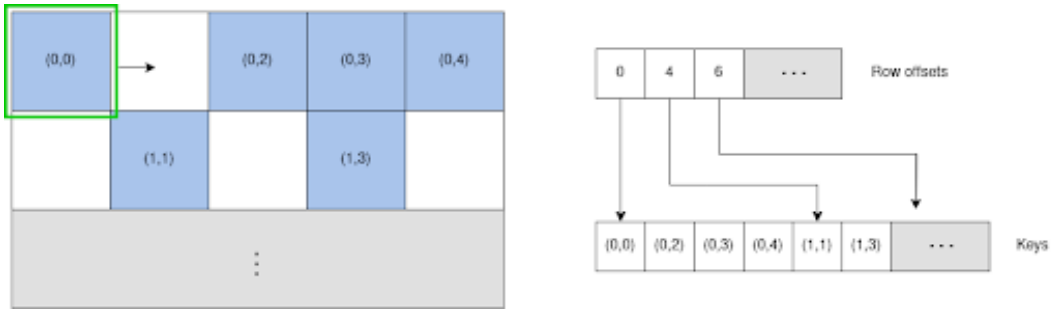

Figura 4.1: Primera implementación de SpMV para bmSparse

Luego de la etapa de preprocesamiento, cada bloque de hilos utiliza el vector *row_offsets* para obtener el primer y último bloque de la fila que debe procesar. Luego, de forma iterativa, recupera cada bloque de la fila desde memoria global, lo convierte en un bloque denso de 8×8 y lo multiplica por el segmento correspondiente del vector de salida. Este producto es realizado por dos *warps*, dado que el tamaño de los bloques es de 8×8. Cada *warp* es dividido en secciones de 8 hilos que procesan cada fila del bloque. Los hilos acumulan las sumas parciales de las entradas que multiplican (por ejemplo, el primer hilo del bloque procesa las entradas $(1, 1)$ de todos los bloques de la fila). El resultado final se obtiene realizando una reducción a través de la operación de tipo *warp shuffle* de CUDA. Esta implementación se representa

en la Figura 4.1.

El código de CUDA para esta variante se puede ver en el Algoritmo 4.1. La matriz es representada por los vectores *A_keys*, *A_vals*, *A_bmps* y *A_off*. El vector *bl_rw_ptr* es el computado en la etapa de preprocesamiento, que contiene la información sobre el comienzo de cada fila. La grilla de CUDA se ejecuta con una cantidad de bloques de hilos igual a la cantidad de filas de bloques de la matriz (aproximadamente $n/8$). Luego de las inicializaciones, desde la línea 16 a la 31 se procesa la fila de bloques correspondiente al bloque de CUDA determinado por *blockIdx.x*. En la línea 18 se recupera el *bitmap* correspondiente al bloque y en la línea 19 cada hilo obtiene el bit que le corresponde dentro del *bitmap*. Luego, en las líneas 23 a 28, si este bit es un 1, el hilo cuenta la cantidad de unos a su izquierda dentro del bitmap para obtener su posición dentro del bloque, y así recuperar su valor dentro del arreglo de valores *A_val*. También se obtiene el valor del vector de entrada correspondiente al hilo calculando el valor de la columna del elemento que está procesando el hilo.

El producto entre el coeficiente de la matriz y el del vector de entrada se acumula en un registro. Al final de la iteración, cada conjunto de 8 hilos contiguos contiene valores que corresponden a la misma entrada del vector de salida, ya que son hilos que procesaron elementos de una misma fila. Esta reducción se realiza en paralelo a través de la primitiva *shfl_down_sync* con un parámetro adicional que permite que grupos de 8 hilos se reduzcan de forma independiente.

Algoritmo 4.1: Código CUDA del kernel de la primera variante

```
1  template <class VIN, class VOUT> __global__
2  void spmv_kernel_new( uint64_t* bl_rw_ptr, uint64_t* A_keys,
3                        VIN* A_val, uint64_t* A_bmps, uint64_t*
                          ↪  A_off,
4                        VIN* v, VOUT* u, int n)
5  {
6      VOUT res = 0;
7
8      const int lane_id = threadIdx.x % 64,
9                row_idx = blockIdx.x*8+ threadIdx.x/8;
10
```

```
11      if (row_idx >= n) return;
12
13      uint64_t first = bl_rw_ptr[blockIdx.x],
14               last = bl_rw_ptr[blockIdx.x+1];
15
16      for(uint64_t bl_idx = first; bl_idx < last; bl_idx++ ){
17
18          uint64_t bmp = A_bmps[bl_idx];
19          uint64_t my_bit = bmp & (uint64_t(1) << 64 - 1 -
            ↪  lane_id);
20
21          VIN mat_coef = 0, valv = 0;
22
23          if (my_bit) {
24              int col_idx = ((A_keys[bl_idx] << 32) >>
                ↪  32)*8+threadIdx.x%8;
25              int pos = __popcll(bmp >> (64 - lane_id));
26              mat_coef = A_val[A_off[bl_idx]+pos];
27              valv = v[col_idx];
28          }
29
30          res+= (VOUT) (mat_coef*valv);
31      }
32
33      for (int i=4; i>=1; i/=2)
34              res += __shfl_down_sync(__activemask(), res, i, 8);
35
36      if((lane_id % 8) == 0)
37          u[row_idx]  = res;
38  }
```

4.2. Segunda variante, 1 warp por fila (SpMV$_{BATCH}$)

La primera variante resulta bastante sencilla de entender y el código del *kernel* es relativamente simple. Además, se logra un acceso *coalesced* a la matriz, ya que los elementos a los que accede cada *warp* son contiguos en memoria (el arreglo de valores se ordena por bloque, y dentro de cada bloque, por fila). Sin embargo, el procesamiento secuencial de las filas puede resultar en un gran desaprovechamiento de los recursos. La máxima cantidad posible de operaciones en paralelo es 64 para cada fila de bloques, y en muchos casos será menor, dado que los bloques con densidad alta suelen ser poco comunes. Además, puede haber un importante desbalance de carga entre distintos bloques de hilos dado que la cantidad de bloques de dos filas de la matriz de bloques puede ser muy distinta. Para atacar estos problemas, se propone una segunda variante que procesa cada fila en tandas de 4 bloques, mostrada en la Figura 4.2. Ahora los bloques de CUDA son de 8×32 hilos, lo que teóricamente incrementa el paralelismo en comparación con la primera variante.

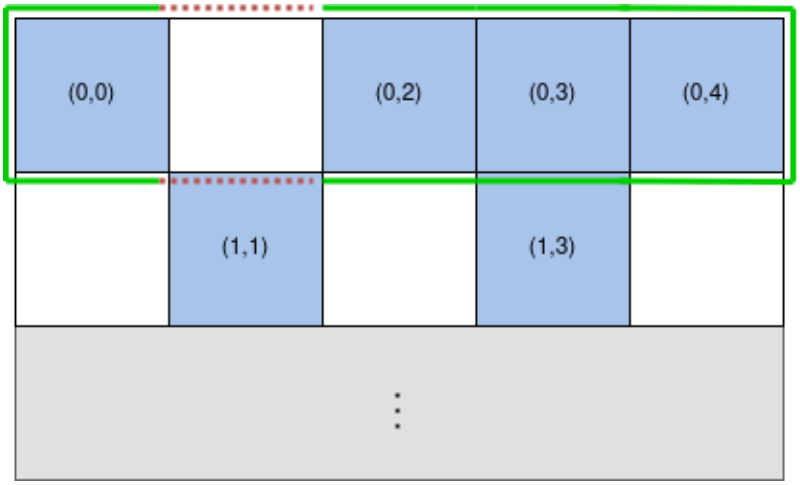

Figura 4.2: Implementación de SpMV procesando de a 4 bloques

En la primera variante, 2 *warps* procesan cada fila de bloques. Los hilos del mismo warp procesan distintas filas de la matriz pero siempre del mismo bloque. Como se puede ver en la Figura 4.3, en esta nueva variante son 8 los *warps* que procesan cada fila, y los hilos del mismo warp procesan la misma fila de la matriz, pero distintos bloques.

Esta variante no requiere grandes modificaciones de código en relación a

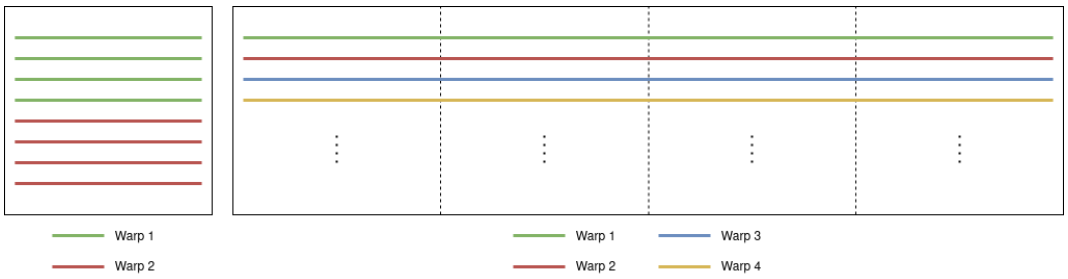

Figura 4.3: Diferencia de distribución de warps entre las variantes propuestas.

la primera. Los mayores cambios son en el *loop* principal, que ahora procesa de a 4 bloques, en el cómputo de los índices *lane_id*, *bl_idx*, *row_idx* y *col_idx* y en la reducción final que, dado que ahora los *warps* procesan elementos de la misma fila, se realiza sobre todo el warp, por lo que no se necesita el parámetro adicional para hacer reducciones independientes dentro del mismo *warp*. El código de CUDA para esta variante se puede ver en el Algoritmo 4.2

Algoritmo 4.2: Código CUDA del kernel de la segunda variante

```
1  template <class ValueIn, class ValueOut>
2  __global__
3  void spmv_kernel_new(uint64_t* first_blocks, uint64_t* A_keys,
4                       ValueIn* A_values, ValueIn* v, ValueOut* u,
5                       uint64_t* A_bmps, uint64_t* A_offsets, int
                        ↪  n){
6
7      ValueOut res = 0;
8
9      if(blockIdx.x*8+threadIdx.y >= n) return;
10
11     uint64_t first = first_blocks[blockIdx.x], last =
       ↪  first_blocks[blockIdx.x+1];
12
13     int blocksLeft = last - first;
14
15     int blk = threadIdx.x/8;
16     int col = threadIdx.x%8;
17     int row = threadIdx.y;
18
```

```
19      int lane_id = row*8 + col;
20
21      while(blocksLeft>0){
22
23          ValueIn aux, valv;
24          uint64_t bmp, key, off;
25          int blk_idx;
26
27          aux=0;
28          valv=0;
29
30          if(blk<blocksLeft){
31
32              blk_idx = last - blocksLeft + blk;
33              bmp = A_bmps[blk_idx];
34
35              uint64_t my_bit = bmp & (uint64_t(1) <<
                ↪  BMSP_BLOCK_SIZE - 1 - lane_id);
36
37              if(my_bit){
38
39                  off = A_offsets[blk_idx];
40                  key = (A_keys[blk_idx] << 32) >> 32;
41                  valv=v[key*8 + col];
42
43                  int pos = __popcll(bmp >> (BMSP_BLOCK_SIZE -
                    ↪  lane_id));
44                  aux = A_values[off+pos];
45              }else{
46                  aux = 0;
47                  valv = 0;
48              }
49
50          }
51
52          res += (ValueOut) aux * valv;
```

```
53          blocksLeft-=4;
54      }
55
56       __syncthreads();
57
58      for(int i=16; i>=1; i/=2){
59          res += __shfl_down_sync(__activemask(), res, i, 32);
60      }
61       __syncthreads();
62
63      if( threadIdx.x == 0 ){
64          u[blockIdx.x*8 + row] = res;
65      }
66  }
```

Una desventaja de esta variante es que se pierde el acceso *coalesced* sobre la matriz, dado que ahora cada 8 elementos del *warp* habrá un acceso no *coalesced* al pasar de un bloque a otro. Sin embargo, en los dispositivos CUDA modernos se mitiga el acceso no *coalesced* haciendo que una operación de memoria de un *warp* se divida en tantas transferencias de 32 *bytes* como sea necesario para satisfacer las solicitudes de cada hilo.

4.3. Evaluación experimental

La plataforma sobre la que se ejecutaron los experimentos consiste de un servidor con un procesador Intel(R) Core(TM) i7-6700 @ 3.40GHz, con 64GB de RAM, sistema operativo Linux y una GPU NVIDIA RTX 3090 Ti (arquitectura *Ampere*). Se utilizó la versión 11.4 del *toolkit* de CUDA (incluyendo las bibliotecas *thrust* y cuSparse).

Se evaluaron ambas variantes sobre 1455 matrices de la *SuiteSparse Matrix Collection*. El conjunto fue tomado de manera que fuera lo más diverso posible, conteniendo matrices de distintos tamaños y patrones de valores no nulos, como se puede ver en la izquierda de la Figura 4.4. Por otro lado, a la derecha de la Figura 4.4 se puede ver que, en promedio, en el *dataset* predominan matrices con bloques muy dispersos, lo que, en principio, no favorece el uso del formato *bmSparse*. Se utilizaron números en punto flotante de precisión simple (*floats*).

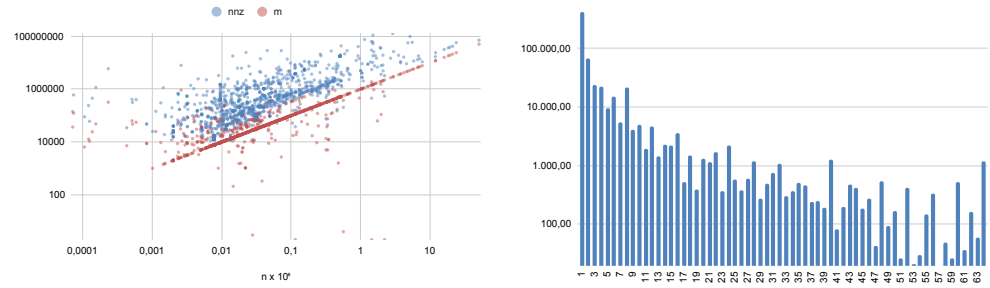

Figura 4.4: Características del conjunto de matrices utilizado para la evaluación. La cantidad de valores no nulos y columnas de las matrices vs. cantidad de filas (izquierda), y cantidad promedio de bloques de *bmSparse* con 1-64 elementos (derecha). En ambas gráficas el eje y está en escala logarítmica.

Para la evaluación de ambas variantes, se realiza una comparación con la implementación de SpMV de la biblioteca cuSparse para el formato *CSR*. De este punto en adelante, se le llamará SpMV$_{BASE}$ a la variante presentada en la Sección 4.1 y SpMV$_{BATCH}$ a la presentada en la Sección 4.2.

Tanto la implementación de cuSparse como las variantes propuestas requieren una etapa de preprocesamiento. En el caso de las rutinas de *bmSparse*, se requiere el cómputo del vector que contiene el inicio de cada fila. Para cuSparse, es necesario la creación de un *handle* y un *buffer*. La evaluación experimental se centra únicamente en los *kernels* de multiplicación ya que las etapas de preprocesamiento normalmente se hacen una única vez, para luego hacer múltiples llamados al *kernel* SpMV, por lo que el tiempo de esto último es lo que termina predominando en la práctica.

En la Figura 4.5 se pueden ver los tiempos de ejecución de las dos variantes de *bmSparse* y la implementación de cuSparse para SpMV. Es importante notar el contraste entre la regularidad de los tiempos de ejecución de cuSparse, que dependen mayoritariamente de la cantidad de valores no nulos, y los tiempos de ambas variantes de *bmSparse*, que son altamente oscilantes. Particularmente, esto se debe a que otros factores, tales como la dispersión individual de cada bloque y la cantidad de valores no nulos por fila de bloques influyen fuertemente en el desempeño de las variantes de *bmSparse*.

La Figura 4.6 muestra la relación entre los tiempos de ejecución de ambas variantes de SpMV para *bmSparse*. Como se puede ver en la figura, SpMV$_{BATCH}$ supera consistentemente a SpMV$_{BASE}$. La media armónica de la aceleración es 1.38×, y hay matrices donde SpMV$_{BATCH}$ obtiene *speedups* de

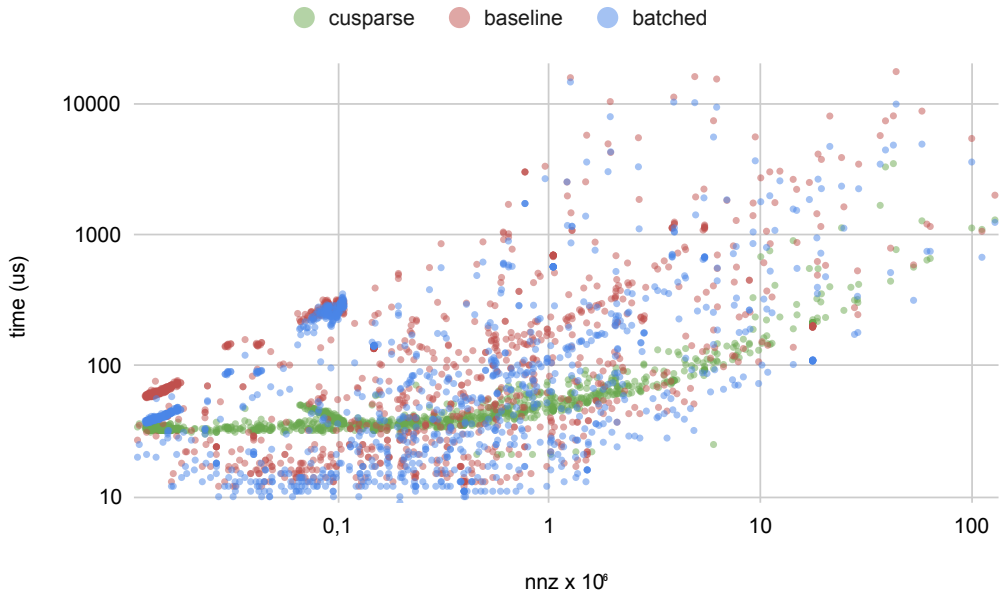

Figura 4.5: Tiempo de ejecución (en μs) de ambas variantes de *bmSparse* y la implementación de cuSparse. Las matrices están en orden creciente de nnz. El eje x está en escala logarítmica.

hasta 3× sobre SpMV$_{BASE}$. La variante SpMV$_{BASE}$ obtiene un mejor tiempo de ejecución en solamente 60 de las 1445 matrices, y la mayor diferencia obtenida en comparación a SpMV$_{BATCH}$ es de 72 %.

Dados los resultados obtenidos, se concluye que SpMV$_{BATCH}$ es la mejor de las implementaciones para *bmSparse*. Consecuentemente, es la elegida para seguir comparándola con la implementación de cuSparse. La Figura 4.7 muestra los tiempos de ejecución de cuSparse dividido por los tiempos de ejecución de SpMV$_{BATCH}$, presentando los resultados en orden creciente de nnz (izquierda) y en orden creciente de cantidad promedio de nnz por bloque de *bmSparse* (derecha). De las figuras se puede concluir que tanto la cantidad de valores no nulos como la cantidad promedio de valores no nulos por bloque tiene una influencia importante en la potencial ventaja de SpMV$_{BATCH}$ sobre cuSparse. Esto en cierta medida explica la variabilidad encontrada en los resultados presentados en la Figura 4.5. En particular, la variante de *bmSparse* es superada consistentemente para matrices con menos de 500.000 valores no nulos. Hay excepciones, como la matriz *bcsstk33*, que tiene 393.810 valores no nulos pero un promedio de 14,72 valores no nulos por bloque, para la que SpMV$_{BATCH}$ obtiene un *speedup* de más de 4× sobre cuSparse. La media

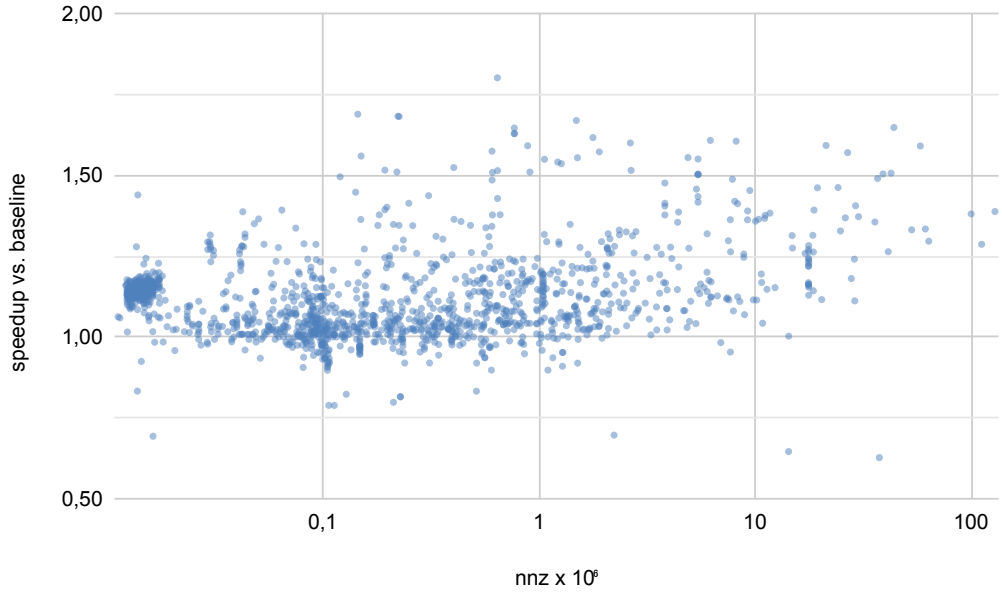

Figura 4.6: Tiempo de ejecución de SpMV$_{BASE}$ dividido por el tiempo de ejecución de la SpMV$_{BATCH}$ (los valores mayores a 1 favorecen a la SpMV$_{BATCH}$). Los resultados están en orden creciente de valores no nulos (nnz). El eje x está en escala logarítmica.

armónica de los *speedups* para matrices con un promedio de más de 31 valores no nulos por bloque es de 1,62 a favor de SpMV$_{BATCH}$.

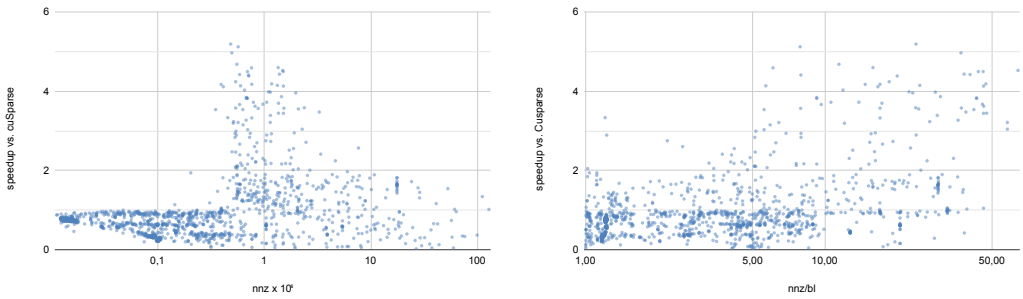

Figura 4.7: Aceleración obtenida por SpMV$_{BATCH}$ en comparación a cuSparse presentada en orden creciente de nnz (izquierda) y orden creciente de cantidad promedio de nnz por bloque de *bmSparse* (derecha). El eje x está en escala logarítmica.

Capítulo 5

Conclusiones y trabajo futuro

El trabajo realizado buscó alcanzar el estado del arte en la resolución de problemas dispersos utilizando formatos de almacenamiento y algoritmos a bloques. Luego de un relevamiento del estado del arte, se decidió profundizar sobre el formato *bmSparse* dado que su estructura a bloques e indexación a través de mapas de bits da lugar a potenciales ahorros importantes en cuanto al espacio de almacenamiento y a los accesos a memoria. Además, este formato presenta características interesantes en el contexto de la SpGeMM, una operación que ha ganado notoriedad en los últimos años gracias al desarrollo de las aplicaciones de ciencias de datos. Algunos trabajos previos a esta tesis que mostraron las potencialidades de este formato son [111] [78].

5.1. Conclusiones

Con el objetivo de realizar las evaluaciones sobre las mejoras propuestas, se realizó una implementación del algoritmo de SpGeMM propuesto en el artículo original que presenta el formato [111]. El análisis de esta implementación reveló que los dos mayores cuellos de botella se producen en las etapas de multiplicación, seguido de la etapa de ordenamiento de la *task list*. Con respecto a la etapa de multiplicación, en la Sección 3.1 se presentó una nueva representación para la *task list*. Si bien la representación original es ventajosa para etapas como la de ordenamiento, en las que la posición de los bloques dentro de la matriz es útil, representar los bloques utilizando sus índices en los vectores facilita el acceso a sus *bitmaps* y a los valores asociados, lo cual favorece el desempeño de la etapa de multiplicación. La evaluación experimental mostró

que esta nueva representación alcanza *speedups* de hasta $2,4\times$ en las matrices utilizadas para la evaluación.

En la Sección 3.4 se estudió el uso de la tecnología de Tensor Cores de NVIDIA para realizar el producto entre bloques dentro de la etapa de multiplicación, adaptando implementaciones de [78] para su funcionamiento con esta nueva versión. Estas implementaciones son unificadas en un mismo código para permitir la elección de la variante a utilizar. Los resultados obtenidos muestran que el uso de Tensor Cores ofrece aceleraciones importantes en comparación a versión de la multiplicación que utiliza CUDA Cores. No es claro distinguir cual de las variantes implementadas con Tensor Cores resulta ser la mejor; para matrices pequeñas TC_{padd} obtiene los mejores resultados, mientras que para las matrices de mayor tamaño se destacan las variantes TC_{2blk} y TC_{2tsk}.

En cuanto a la etapa de ordenamiento, se propuso aprovechar el orden parcial de los elementos de la *task list* utilizando un algoritmo de *sort* por segmentos. Para esto se utilizó la implementación de *segmented sort* de [63]. Las evaluaciones revelaron reducciones del tiempo de ejecución de esta etapa de hasta un 50 %. Aunque los resultados de este nuevo método de ordenamiento fueron en general positivos, se identificaron casos en los que el algoritmo de ordenamiento de *Thrust* (*thrust::sort*) obtuvo un mejor desempeño. Por esto, en la Sección 3.3 se planteó el uso de un ordenamiento adaptativo, que según las características de la matriz, utilizará el algoritmo de ordenamiento más adecuado.

Comparando una implementación que agrupa todas estas propuestas, los resultados obtenidos son considerablemente superiores en comparación con implementaciones de SpGeMM de dos de las bibliotecas más utilizadas en el ámbito, como lo son cuSparse y MKL.

Si bien *bmSparse* fue diseñado con el fin de ser un formato adecuado para la operación SpGeMM para muchas matrices, ofrece importantes ahorros en términos de espacio de almacenamiento y cantidad de accesos a memoria, aspectos que resultan de gran interés y que hacen que *bmSparse* tenga potencial para ser un formato de propósito general para el almacenamiento de matrices dispersas. Para explorar esta posibilidad, se propusieron dos implementaciones para GPUs de la operación SpMV, detallados en el Capítulo 4. Los resultados muestran que el kernel presentado en la Sección 4.2 supera al descrito en la Sección 4.1 en casi todos los casos de prueba. Una evaluación más profunda de

la mejor de estas variantes reveló que obtiene mayores desempeños a medida que la cantidad de valores no nulos por bloque de *bmSparse* crece. Más precisamente, la propuesta para *bmSparse* es competitiva con la implementación de SpMV de cuSparse para el formato CSR en matrices con más de nueve valores no nulos por bloque en promedio, alcanzando *speedups* de hasta 3×. Como parte del trabajo se dejan disponibles las implementaciones de SpGeMM y SpMV utilizando el formato *bmSparse*[1].

5.2. Difusión del trabajo realizado

A continuación se presentan las publicaciones de los resultados obtenidos en el marco de esta tesis en distintos foros de divulgación científica.

. Revistas

- G. Berger, M. Freire, R. Marini, E. Dufrechou and P. Ezzatti, Advancing on an efficient sparse matrix multiplication kernel for modern GPUs, *Concurrency and Computation: Practice and Experience*. 2023; 35(20):e7271. doi: 10.1002/CPE.7271. url: https://doi.org/10.1002/cpe.7271

. Conferencias internacionales

- G. Berger, M. Freire, R. Marini, E. Dufrechou and P. Ezzatti, Ünleashing the performance of bmSparse for the sparse matrix multiplication in GPUs", *2021 12th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, St. Louis, MN, USA, 2021, pp. 19-26, November 19, 2021. IEEE, 2021, pp. 19–26. doi: 10.1109/SCALA54577.2021.00008. url: https://doi.org/10.1109/ScalA54577.2021.00008
- G. Berger, E. Dufrechou and P. Ezzatti, "Sparse matrix-vector product for the bmSparse matrix format in GPUs", *21st International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*, Limassol, Cyprus, August 28, 2023 (aceptado para publicación)

[1] https://github.com/GonzaBerger/bmSparse-SPGEMM-SPMV

5.3. Trabajo futuro

Los resultados obtenidos invitan a continuar realizando esfuerzos, profundizando sobre el uso del formato *bmSparse* a través de las siguientes líneas de trabajo futuro:

- Implementar una variante del algoritmo de SpGeMM que omita la etapa de ordenamiento, desarrollando una variante de la etapa de multiplicación que pueda procesar las tasks fuera de orden. Esto implica realizar una búsqueda lineal en la etapa de multiplicación. Esta búsqueda no debe realizarse sobre la totalidad de la task list, sino solamente en el segmento que contiene el mismo índice i.
- Desarrollar una rutina que estime y seleccione, de acuerdo a las características de las matrices, el kernel de SpGeMM que obtendrá el mejor desempeño (entre, por ejemplo, *bmSparse*, cuSparse, MKL CSR, MKL BSR-2, etc.).
- Revisitar el uso de *Tensor Cores* considerando el lanzamiento de nuevas generaciones de esta tecnología (siendo la cuarta la más nueva al momento de escribir este trabajo). En este aspecto interesa explorar el uso de fragmentos de distinto tamaño, que pueden ser más adecuados de acuerdo al tamaño de bloque de *bmSparse*. Por otro lado, interesa evaluar el desempeño de otras precisiones disponibles, tales como TF32, bfloat16, entre otras.
- Extender el algoritmo SpGeMM propuesto con una selección automática de la variante de Tensor Cores a utilizar, teniendo en cuenta las características de las matrices.
- Explorar las consecuencias de modificar el formato *bmSparse*, como puede ser el uso de bloques de distintos tamaños.
- Reevaluar el diseño de las implementaciones de SpMV, poniendo especial enfoque a mitigar el impacto de bloques con pocos valores no nulos. Una posibilidad es realizar una evaluación previa de la matriz y asignar los hilos de manera distinta en bloques o filas de bloques altamente dispersas.
- Avanzar en implementaciones para otras operaciones matriciales, como por ejemplo un solver triangular disperso (SpTrSv).

Referencias bibliográficas

[1] Bfloat16. https://cloud.google.com/tpu/docs/bfloat16?hl=es-419. Fecha de acceso: 2024-31-01.

[2] cublas library. https://developer.nvidia.com/cublas. Last access: 2023-31-12.

[3] *CUDA C++ Programming Guide*.

[4] cusparse library. https://developer.nvidia.com/cusparse. Last access: 2023-31-12.

[5] Intel mkl. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html. Last access: 2023-31-12.

[6] Intel mkl gpu offloading. https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-0/offloading-onemkl-computations-onto-the-gpu.html. Last access: 2023-31-12.

[7] Offloading onemkl computations onto the gpu. https://www.nvidia.com/es-es/data-center/tensor-cores/. Access date: 2023-21-12.

[8] Openblas library. https://www.openblas.net/. Last access: 2023-31-12.

[9] Ptx documentation. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html. Fecha de acceso: 2021-21-02.

[10] Thrust. https://thrust.github.io/. Access date: 2024-18-01.

[11] R.C. Agarwal, F.G. Gustavson, and M. Zubair. A high performance algorithm using pre-processing for the sparse matrix-vector multiplication. In *Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 32–41, 1992.

[12] Willow Ahrens and Erik G. Boman. On optimal partitioning for sparse matrices in variable block row format. *CoRR*, abs/2005.12414, 2020.

[13] G. S. Almasi and A. Gottlieb. *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., USA, 1989.

[14] Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. Balanced hashing and efficient gpu sparse general matrix-matrix multiplication. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery.

[15] W. Aspray. The intel 4004 microprocessor: what constituted invention? *IEEE Annals of the History of Computing*, 19(3):4–15, 1997.

[16] Ariful Azad, Grey Ballard, Aydin Buluç, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, 38(6):C624–C651, 2016.

[17] Ariful Azad, Aydin Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811, 2015.

[18] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 12 1994.

[19] Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. Communication optimal parallel multiplication of sparse random matrices. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, page 222–231, New York, NY, USA, 2013. Association for Computing Machinery.

[20] Grey Ballard, Christopher Siefert, and Jonathan Hu. Reducing communication costs for sparse matrix multiplication within algebraic multigrid. *SIAM Journal on Scientific Computing*, 38(3):C203–C231, 2016.

[21] Nathan Bell, Steven Dalton, and Luke N. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing*, 34(4):C123–C152, 2012.

[22] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[23] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 2009.

[24] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, 1966.

[25] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, 2009.

[26] Robert K. Brayton, Fred G. Gustavson, and Ralph A. Willoughby. Some results on sparse matrices. *Mathematics of Computation*, 24(112):937–954, 1970.

[27] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1):107–117, 1998. Proceedings of the Seventh International World Wide Web Conference.

[28] Luc Buatois, Guillaume Caumon, and Bruno Levy. Concurrent number cruncher - a gpu implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems*, 24, 2009.

[29] Aydın Buluç and John R. Gilbert. The combinatorial blas: design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25:496 – 509, 2011.

[30] Thomas J. Watson IBM Research Center, T. Agerwala, and J. Cocke. *High Performance Reduced Instruction Set Processors*. IBM Thomas J. Watson Research Division, 1987.

[31] Timothy M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '07, page 590–598, New York, NY, USA, 2007. Association for Computing Machinery.

[32] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, page 115–126, New York, NY, USA, 2010. Association for Computing Machinery.

[33] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science Engineering*, 11(4):29–41, 2009.

[34] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Trans. Math. Softw.*, 41(4), 10 2015.

[35] Jack W. Davidson and Sanjay Jinturkar. Memory access coalescing: a technique for eliminating redundant memory accesses. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 186–195, New York, NY, USA, 1994. Association for Computing Machinery.

[36] Timothy A. Davis. Graph algorithms via suitesparse: Graphblas: triangle counting and k-truss. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6, 2018.

[37] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011.

[38] Mehmet Deveci, Erik G Boman, Karen D. Devine, and Sivasankaran Rajamanickam. Parallel graph coloring for manycore architectures. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 892–901, 2016.

[39] Mehmet Deveci, Simon D. Hammond, Michael M. Wolf, and Sivasankaran Rajamanickam. Sparse matrix-matrix multiplication on multilevel memory architectures : Algorithms and experiments. *CoRR*, abs/1804.00695, 2018.

[40] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. Performance-portable sparse matrix-matrix multiplication for many-core architectures. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 693–702, 2017.

[41] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 3 1990.

[42] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1979.

[43] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, 3 1988.

[44] I. S. Duff, Roger G. Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15(1):1–14, 3 1989.

[45] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Trans. Math. Softw.*, 28(2):239–267, 6 2002.

[46] Stanley C. Eisenstat, M. C. Gursky, Martin H. Schultz, and Andrew H. Sherman. Yale sparse matrix package i: The symmetric codes. *International Journal for Numerical Methods in Engineering*, 18:1145–1151, 1982.

[47] Anand Ekambaram and Eurípides Montagne. An alternative compressed storage format for sparse matrices. In Adnan Yazıcı and Cevat Şener, editors, *Computer and Information Sciences - ISCIS 2003*, pages 196–203, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[48] Joseph A. Fisher and Ramakrishna Rau. Instruction-level parallel processing. *Science*, 253(5025):1233–1241, 1991.

[49] Michael Flynn. *Flynn's Taxonomy*, pages 689–697. Springer US, Boston, MA, 2011.

[50] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[51] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. A systematic survey of general sparse matrix-matrix multiplication. *ACM Computing Surveys*, 55(12):1–36, March 2023.

[52] Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Commun. ACM*, 53(11):58–66, 11 2010.

[53] Joachim Georgii and Rüdiger Westermann. A streaming approach for sparse matrix products and its application in galerkin multigrid methods. *Electronic Transactions on Numerical Analysis*, 37, 2010.

[54] P. Gepner and M.F. Kowalik. Multi-core processors: New way to achieve high system performance. In *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, pages 9–13, 2006.

[55] John Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13, 1997.

[56] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. High-performance graph algorithms from parallel sparse matrices. In Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Waśniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 260–269, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[57] Philip E. Gill, Walter Murray, Michael A. Saunders, and Margaret H. Wright. Sparse matrix methods in optimization. *SIAM Journal on Scientific and Statistical Computing*, 5(3):562–589, 1984.

[58] Felix Gremse, Andreas Höfter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging. *SIAM Journal on Scientific Computing*, 37(1):C54–C71, 2015.

[59] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, 9 1978.

[60] Richard J Hanson, Fred T Krogh, and CL Lawson. A proposal for standard linear algebra subprograms. Technical report, 1973.

[61] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[62] Jared Hoberock and Nathan Bell. Thrust v1.0 release a high-level c++ template library for cuda, 5 2009.

[63] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. Fast segmented sort on gpus. In *Proceedings of the International Conference on Supercomputing*, ICS '17, New York, NY, USA, 2017. Association for Computing Machinery.

[64] The MathWorks Inc., 2024.

[65] Shweta Jain-Mendon and Ron Sass. A case study of streaming storage format for sparse matrices. In *2012 International Conference on Reconfigurable Computing and FPGAs*, pages 1–6. IEEE, 2012.

[66] Haim Kaplan, Micha Sharir, and Elad Verbin. Colored intersection searching via sparse rectangular matrix multiplication. In *Proceedings of the Twenty-Second Annual Symposium on Computational Geometry*, SCG '06, page 52–60, New York, NY, USA, 2006. Association for Computing Machinery.

[67] Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. A comparative study of blocking storage methods for sparse matrices on multicore architectures. In *2009 International Conference on Computational Science and Engineering*, volume 1, pages 247–256, 2009.

[68] Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. Exploring the effect of block shapes on the performance of sparse kernels. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, 2009.

[69] S.W. Keckler, K. Olukotun, and H.P. Hofstee. *Multicore Processors and Systems*. Integrated Circuits and Systems. Springer US, 2009.

[70] Emmett Kilgariff and Randima Fernando. The geforce 6 series gpu architecture. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, page 29–es, New York, NY, USA, 2005. Association for Computing Machinery.

[71] David R Kincaid, Thomas C Oppe, and David M Young. Itpackv 2d user's guide. Technical report, Texas Univ., Austin, TX (USA). Center for Numerical Analysis, 1989.

[72] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, page 207–218, New York, NY, USA, 1981. Association for Computing Machinery.

[73] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, page 149–158, New York, NY, USA, 2001. Association for Computing Machinery.

[74] Erik Lindholm and Stuart Oberman. The nvidia geforce 8800 gpu. In *2007 IEEE Hot Chips 19 Symposium (HCS)*, pages 1–17, 2007.

[75] Junhong Liu, Xin He, Weifeng Liu, and Guangming Tan. Register-aware optimizations for parallel sparse matrix–matrix multiplication. *International Journal of Parallel Programming*, 47(3):403–417, 6 2019.

[76] Weifeng Liu and Brian Vinter. An efficient gpu general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 370–381, 2014.

[77] S.A. Mahlke, W.Y. Chen, J.C. Gyllenhaal, W.W. Hwu, P.P. Chang, and T. Kiyohara. Compiler code transformations for superscalar-based high-performance systems. In *Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 808–817, 1992.

[78] R. Marini Salsamendi. Uso de formatos no convencionales para matrices dispersas en gpus. Tesis de grado, Universidad de la República (Uruguay), Facultad de Ingeniería, 2021.

[79] Tim Mattson, David A. Bader, Jonathan W. Berry, Aydin Buluç, Jack J. Dongarra, Christos Faloutsos, John Feo, John R. Gilbert, Joseph Gonzalez, Bruce Hendrickson, Jeremy Kepner, Charles E. Leiserson, Andrew Lumsdaine, David A. Padua, Stephen W. Poole, Steven P. Reinhardt, Mike Stonebraker, Steve Wallach, and Andrew Yoo. Standards for graph algorithm primitives. *CoRR*, abs/1408.0393, 2014.

[80] Thaha Mohammed and Rashid Mehmood. Performance enhancement strategies for sparse matrix-vector multiplication (spmv) and iterative linear solvers, 2022.

[81] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[82] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 101–110, 2017.

[83] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. Tilespmv: A tiled algorithm for sparse matrix-vector multiplication on gpus. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 68–78, 2021.

[84] NVIDIA. Nvidia tesla v100 gpu architecture. Technical report, 2009.

[85] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi. Technical report, 2009.

[86] NVIDIA. Nvidia turing gpu architecture. Technical report, 2018.

[87] NVIDIA. Nvidia ampere ga102 gpu architecture. Technical report, 2021.

[88] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-

purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[89] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: Preparing for a new exponential. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 67–72, 2006.

[90] David A Patterson and John L Hennessy. *Computer organization and Design*. Morgan Kaufmann, 1994.

[91] Md. Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Jongsoo Park, Michael Anderson, Satya Vadlamudi, Dipankar Das, Sergey Pudov, Vadim Pirogov, and Pradeep Dubey. Parallel efficient sparse matrix-matrix multiplication on multicore platforms. pages 48–57, 2015.

[92] A. Pinar and M.T. Heath. Improving performance of sparse matrix-vector multiplication. In *SC '99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, pages 30–30, 1999.

[93] Sivasankaran Rajamanickam, Seher Acer, Luc Berger-Vergiat, Vinh Dang, Nathan Ellingwood, Evan Harvey, Brian Kelley, Christian R. Trott, Jeremiah Wilke, and Ichitaro Yamazaki. Kokkos kernels: Performance portable sparse/dense linear algebra and graph kernels, 2021.

[94] John R. Rice and Ronald F. Boisvert. *Solving Elliptic Problems Using ELLPACK*. Springer New York, 1985.

[95] Yousef Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2. 1990.

[96] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[97] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–10, 2009.

[98] Rukhsana Shahnaz, Anila Usman, and Imran R. Chughtai. Review of storage techniques for sparse matrices. In *2005 Pakistan Section Multitopic Conference*, pages 1–7, 2005.

[99] FS Smailbegovic, Georgi N Gaydadjiev, and Stamatis Vassiliadis. Sparse matrix storage format. In *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing*, pages 445–448, 2005.

[100] A.S. Tanenbaum and T. Austin. *Structured Computer Organization.* Pearson, 2013.

[101] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. The more the merrier: Efficient multi-source graph traversal. *Proc. VLDB Endow.*, 8(4):449–460, 12 2014.

[102] Stamatis Vassiliadis, Sorin Cotofana, and Pyrrhos Stathis. Vector ISA extension for sparse matrix-vector multiplication. 2000.

[103] Balaji Venu. Multi-core processors - an overview. *CoRR*, abs/1110.3535, 2011.

[104] Richard Wilson Vuduc and James W. Demmel. *Automatic performance tuning of sparse matrix kernels.* PhD thesis, 2003. AAI3121741.

[105] J. H. Wilkinson and C. Reinsch. *The Algebraic Eigenvalue Problem.* Springer Berlin Heidelberg, 1971.

[106] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. Adaptive sparse matrix-matrix multiplication on the gpu. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 68–81, New York, NY, USA, 2019. Association for Computing Machinery.

[107] Michael M. Wolf, Mehmet Deveci, Jonathan W. Berry, Simon D. Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with kokkoskernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[108] D. Yan, W. Wang, and X. Chu. Demystifying tensor cores to optimize half-precision matrix multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 634–643, 2020.

[109] Di Yan, Tao Wu, Ying Liu, and Yang Gao. An efficient sparse-dense matrix multiplication on a multicore system. In *2017 IEEE 17th International Conference on Communication Technology (ICCT)*, pages 1880–1883. IEEE, 2017.

[110] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. Yaspmv: Yet another spmv framework on gpus. *SIGPLAN Not.*, 49(8):107–118, 2 2014.

[111] Jianting Zhang and Le Gruenwald. Regularizing irregularity: Bitmap-based and portable sparse matrix multiplication for graph data on gpus. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '18, New York, NY, USA, 2018. Association for Computing Machinery.

[112] Yunquan Zhang, Shigang Li, Shengen Yan, and Huiyang Zhou. A cross-platform spmv framework on many-core architectures. *ACM Trans. Archit. Code Optim.*, 13(4), 10 2016.

ANEXOS

Anexo A

Arquitecturas Turing y Ampere

En este anexo se describen las arquitecturas Turing y Ampere, más precisamente la de los chips TU102 y GA102, que son las principales GPUs de estas arquitecturas, y que son las que se encuentran en las tarjetas gráficas utilizadas en este trabajo.

A.1. Arquitectura Turing TU102

La tarjeta RTX 2080 Ti utiliza un chip TU102 de la arquitectura Turing [86] (presentada en la Figura 1), que presenta 68 multiprocesadores (SMs) dispuestos de a pares en 34 *Texture Processing Clusters* (TPCs), que a su vez se organizan en 6 *Graphics Processing Clusters* (GPCs). Cada SM incluye 64 CUDA Cores, 8 Tensor Cores, un archivo de registros de 256 KB, una memoria compartida de 96 KB y una nueva unidad de cómputo denominada *RT Core* (de *Ray Tracing*). En dispositivos con Compute Capability 7.5 (como la RTX 2080 Ti), cada SM contiene 64 núcleos para operaciones de punto flotante (FP32 Core), 64 núcleos para operaciones enteras (INT32 Core) y 2 núcleos para operaciones aritméticas de doble precisión (FP64 cores). En la Figura 2 se muestra un esquema del multiprocesador de esta arquitectura. En total, la tarjeta contiene 4.352 CUDA Cores, 544 Tensor Cores y 68 RT Cores. La tarjeta posee 12 GB de memoria RAM GDDR6 con una interfaz de memoria de 352 bits, siendo la primera arquitectura de GPU en utilizar este subsistema de memoria.

Una de las principales novedades de esta arquitectura es la capacidad de ejecutar operaciones de punto flotante con operaciones enteras en simultáneo.

Figura 1: Arquitectura Turing TU102. Extraída de [86].

En generaciones previas, la ejecución de una instrucción entera bloqueaba la instrucción de punto flotante. Por otro lado, se unifica la memoria compartida con la memoria de caché L1, permitiendo un tamaño variable para este caché según el uso de la memoria compartida. De los 96 KB de esta memoria unificada, el caché L1 puede ocupar hasta 64 KB cuando la memoria compartida no está siendo utilizada en su totalidad. En caso contrario, la capacidad del caché se reduce a 32 KB. Esta modificación implica un ancho de banda dos veces mayor en comparación a la previa arquitectura, Pascal. Además, con la introducción de los RT Cores se permite la aceleración de Ray Tracing en tiempo real, algo que no era posible en generaciones previas utilizando una única GPU.

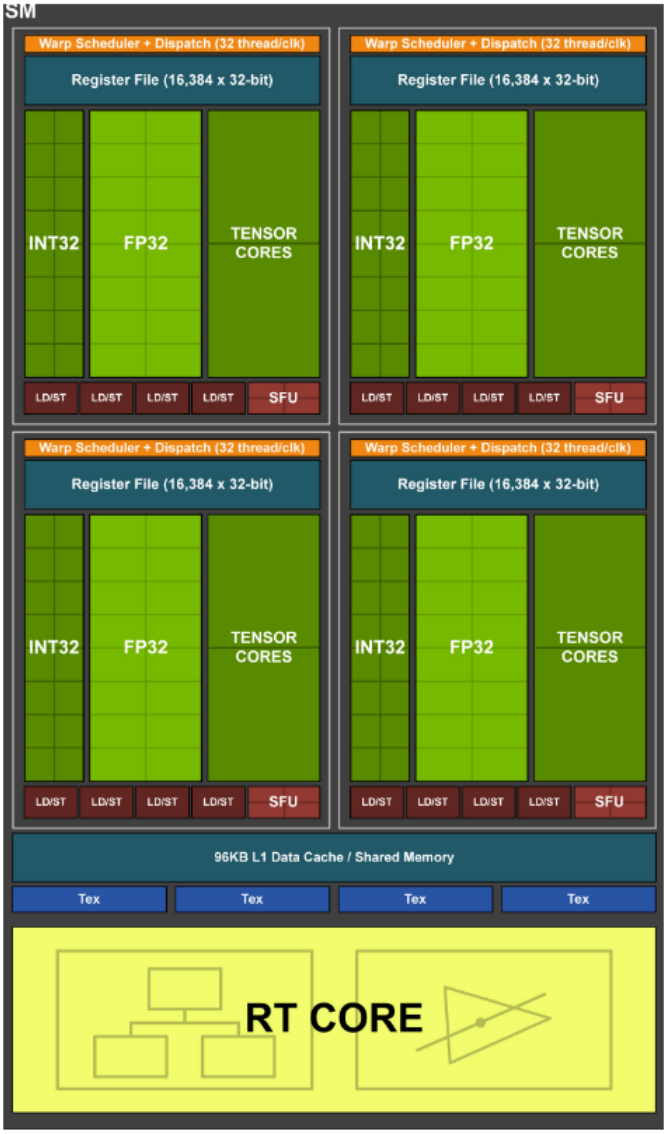

Figura 2: Multiprocesador (SM) de la arquitectura Turing. Esta versión, con 72 SMs, corresponde a la tarjeta NVIDIA Quadro RTX 6000. Los núcleos FP64 no se encuentran representados en la imagen. Imagen extraída de [86].

A.2. Arquitectura Ampere GA102

La arquitectura Ampere GA102 [87] (ilustrada en la Figura 3), presenta 84 multiprocesadores (SMs) dispuestos de a pares en 42 *Texture Processing Clusters* (TPCs), que a su vez se organizan en 7 *Graphics Processing Clusters* (GPCs). Cada SM incluye 128 CUDA Cores, 4 Tensor Cores, un archivo de registros de 256 KB, una memoria unificada (compartida y L1) de 128 KB y un RT Core. En dispositivos con Compute Capability 8.6 (como la RTX 3090

Ti), cada SM contiene 64 núcleos para operaciones de punto flotante (FP32 Core), 64 núcleos para operaciones enteras (INT32 Core) y 2 núcleos para operaciones aritméticas de doble precisión (FP64 cores). En la Figura 4 se muestra un esquema de el multiprocesador de esta arquitectura. En total, la arquitectura contiene 10.752 CUDA Cores, 336 Tensor Cores y 84 RT Cores. La tarjeta posee 24 GB de memoria RAM GDDR6 con una interfaz de memoria de 384 bits.

Figura 3: Arquitectura Ampere GA102. Los núcleos FP64 no se encuentran representados en la imagen. Imagen extraída de [87].

En la arquitectura Ampere, al igual que las anteriores, se dividen los SM en particiones denominadas bloques de procesamiento. Cada bloque puede manejar dos hilos de ejecución. En la arquitectura Turing, uno de estos hilos podía realizar operaciones de punto flotante de precisión simple (FP32), pero el otro quedaba limitado a operaciones enteras. Esto implicaba un gran desaprovechamiento de recursos en aplicaciones que no realizaran aritmética entera. A partir de la arquitectura Ampere, se agrega soporte para operaciones de punto flotante en ambos hilos de ejecución, duplicando la posible cantidad de este tipo de operaciones hechas en simultáneo. Otra novedad de la arquitectura son los núcleos de Ray Tracing de segunda generación, que alcanzan una mejoría de hasta 2× sobre el desempeño de los RT Cores de la arquitectura Turing. La arquitectura Ampere también introduce la tercera generación de Tensor Co-

res. Cada Tensor Core de tercera generación posee dos veces la potencia que sus antecesores de segunda generación. Para matrices densas, la capacidad de cómputo de los TC de tercera generación se duplica en comparación con los de segunda, pero dado que cada SM contiene 4 TC en Ampere y 8 TC en Turing, la capacidad de cómputo de un SM para procesamiento de matrices densas es la misma. Esto es diferente para el procesamiento de matrices dispersas, para las que los TC de tercera generación pueden realizar 256 operaciones *fma* FP16 en simultáneo, cuatro veces las 64 soportadas por los TC de segunda generación. Esto quiere decir que al procesar matrices dispersas, un SM en Ampere ofrece el doble de *throughput* que un SM en Turing. Además, los TC de tercera generación introducen dos nuevas precisiones: *BF16* y *TF32*. La precisión BF16 [1] (de *brain floating point*) incluye un exponente de 8 bits, una mantisa de 7 bits y 1 bit de signo, ofreciendo un rango de representación mayor pero precisión reducida en comparación a la media precisión (FP16). La precisión TF32 (de *tensor float*) es una combinación de la simple y la media precisión, dado que al igual que FP32 utiliza 8 bits de exponente, pero adopta los 10 bits de mantisa que caracterizan a FP16. También se utiliza 1 bit para el signo.

Figura 4: Multiprocesador (SM) de la arquitectura Ampere. Extraída de [87].

Anexo B

Operaciones de Thrust

En este anexo se presentan las operaciones de la biblioteca Thrust utilizadas en este trabajo, con el fin de facilitar la comprensión de la implementación de las distintas etapas de la operación SpGeMM.

B.1. Reduce by key

La operación *thrust::reduce_by_key* es una variante de la operación de reducción, que incluye un arreglo de *keys* que indica de que forma se debe hacer la reducción. Dos elementos son reducidos al mismo valor solamente si son consecutivos y su key es la misma.

Si bien en la biblioteca hay 6 variantes de esta misma operación (con distintos parámetros), la utilizada en este trabajo tiene la siguiente forma:

```
thrust::pair<OutputIterator1,OutputIterator2> thrust::reduce_by_key (

        InputIterator1          keys_first,
        InputIterator1          keys_last,
        InputIterator2          values_first,
        OutputIterator1         keys_output,
        OutputIterator2         values_output
)
```

donde *keys_first* y *keys_last* son punteros al inicio y al fin del arreglo de keys, respectivamente, *values_first* es un puntero al inicio del arreglo de valores a reducir, y *keys_output* y *values_output* son dos punteros al inicio de los arreglos donde se almacenarán las keys y los valores del resultado, res-

pectivamente. En la Figura B.1 se puede ver un ejemplo de ejecución de la operación.

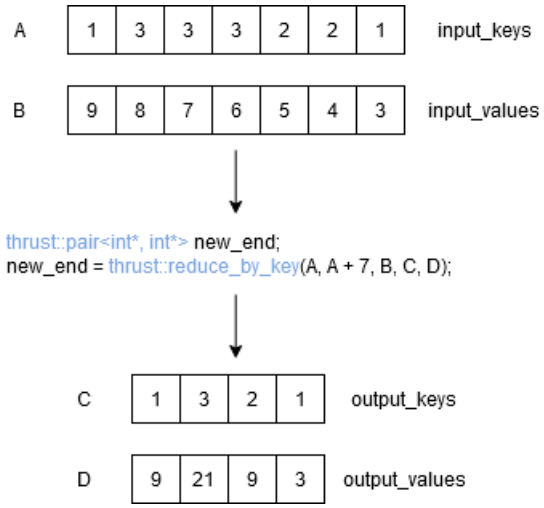

Figura B.1: Ejemplo de ejecución de operación *thrust::reduce_by_key*.

Es importante notar que aunque los elementos con valores 9 y 3 poseen la misma key (1), no son reducidos pues no son consecutivos (hay una key distinta entre ellos). El parámetro de salida, *new_end*, contiene un par de iteradores al final de los arreglos *keys_output* y *values_output*. En este ejemplo contendrá los valores $C + 4$ y $D + 4$.

B.2. Inclusive scan

La primitiva *thrust::inclusive_scan* realiza la operación de suma de prefijos de forma inclusiva. Esto es, la suma acumulada incluye el elemento actual.

Si bien en la biblioteca hay 4 variantes de esta misma operación (con distintos parámetros), la utilizada en este trabajo tiene la siguiente forma:

```
OutputIterator thrust::inclusive_scan (
        InputIterator   first,
        InputIterator   last,
        OutputIterator  result
)
```

donde *first* y *last* son punteros al inicio y al final del arreglo de entrada, respectivamente, y *result* es un puntero al inicio del arreglo donde se almacenará el resultado. El parámetro de salida contiene un iterador al final del arreglo

apuntado por *result*. En la Figura B.2 se puede ver un ejemplo de ejecución de la operación.

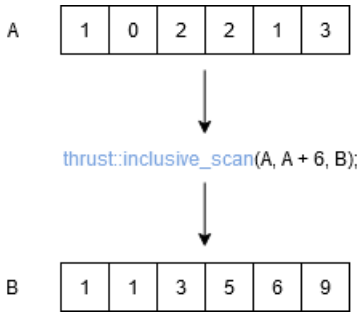

Figura B.2: Ejemplo de ejecución de operación *thrust::inclusive_scan*.

B.3. Exclusive scan

La primitiva *thrust::exclusive_scan* realiza la operación de suma de prefijos de forma exclusiva. Esto es, la suma acumulada no incluye el elemento actual.

Si bien en la biblioteca hay 6 variantes de esta misma operación (con distintos parámetros), la utilizada en este trabajo tiene la siguiente forma:

```
OutputIterator thrust::exclusive_scan (
        InputIterator   first,
        InputIterator   last,
        OutputIterator  result
)
```

donde *first* y *last* son punteros al inicio y al final del arreglo de entrada, respectivamente, y *result* es un puntero al inicio del arreglo donde se almacenará el resultado. El parámetro de salida contiene un iterador al final del arreglo apuntado por *result*. En la Figura B.3 se puede ver un ejemplo de ejecución de la operación.

B.4. Exclusive scan by key

La primitiva *thrust::exclusive_scan_by_key* realiza una operación de exclusive scan segmentada. Si bien en la biblioteca hay 8 variantes de esta misma operación (con distintos parámetros), la utilizada en este trabajo tiene la siguiente forma:

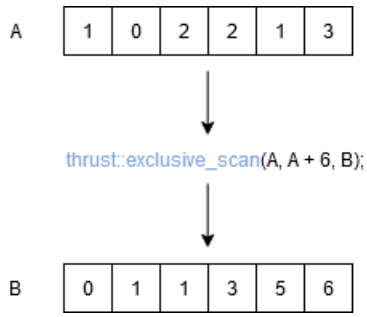

Figura B.3: Ejemplo de ejecución de operación *thrust::exclusive_scan*.

```
OutputIterator thrust::exclusive_scan_by_key (
        InputIterator1  first1,
        InputIterator1  last1,
        InputIterator2  first2,
        OutputIterator  result
)
```

donde *first1* y *last1* son punteros al inicio y al final del arreglo de keys, respectivamente, *first2* es un puntero al inicio del arreglo de valores y *result* es un puntero al inicio del arreglo donde se almacenará el resultado. El parámetro de salida contiene un iterador al final del arreglo apuntado por *result*. En la Figura B.4 se puede ver un ejemplo de ejecución de la operación.

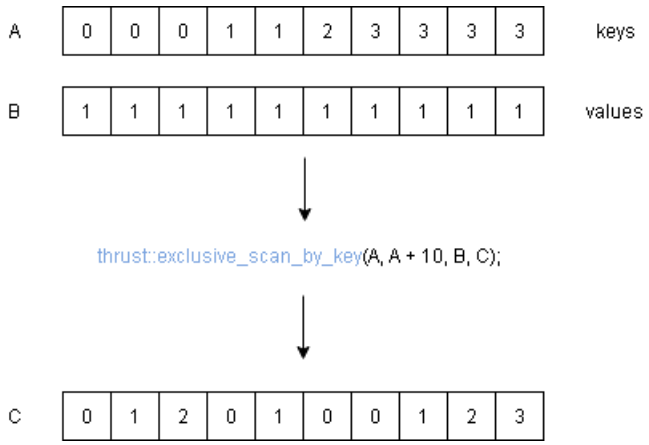

Figura B.4: Ejemplo de ejecución de operación *thrust::exclusive_scan_by_key*.

B.5. Gather

La primitiva *thrust::gather* copia los elementos de un vector a otro según un vector de índices *map*. Si bien en la biblioteca hay 2 variantes de esta

misma operación (con distintos parámetros), la utilizada en este trabajo tiene la siguiente forma:

```
OutputIterator thrust::gather (
        InputIterator          map_first,
        InputIterator          map_last,
        RandomAccessIterator   input_first,
        OutputIterator         result
)
```

donde *map_first* y *map_last* son punteros al inicio y al final del arreglo de índices, *input_first* es un puntero al principio del arreglo de valores y *result* es un puntero al inicio del vector donde se copiarán los valores. Los índices del vector *map* indican de qué manera se debe copiar el arreglo de valores a *result*. Si en la posición i de *map* se tiene un índice j, quiere decir que en la posición i de *result* deberá copiarse el valor contenido en la posición j de *input*. En la Figura B.5 se puede ver un ejemplo de ejecución de la operación.

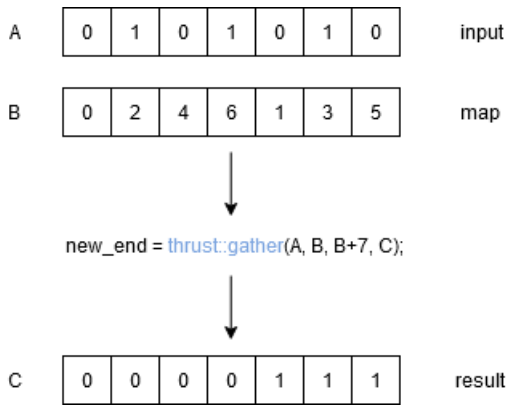

Figura B.5: Ejemplo de ejecución de operación *thrust::gather*.

B.6. Scatter

La primitiva *thrust::scatter* copia los elementos de un vector a otro según un vector de índices *map*. Si bien en la biblioteca hay 2 variantes de esta misma operación (con distintos parámetros), la utilizada en este trabajo tiene la siguiente forma:

donde *first* y *last* son punteros al inicio y al final del arreglo de valores, *map* es un puntero al principio del arreglo de índices y *result* es un puntero al inicio del vector donde se copiarán los valores. Los índices del vector *map* indican de

```
void thrust::scatter (
        InputIterator1          first,
        InputIterator1          last,
        InputIterator2   map,
        RandomAccessIterator         result
)
```

qué manera se debe copiar el arreglo de valores a *result*. Si en la posición i de *map* se tiene un índice j, quiere decir que el elemento en la posición i de *input* se copiará en la posición j de *result*. En la Figura B.6 se puede ver un ejemplo de ejecución de la operación.

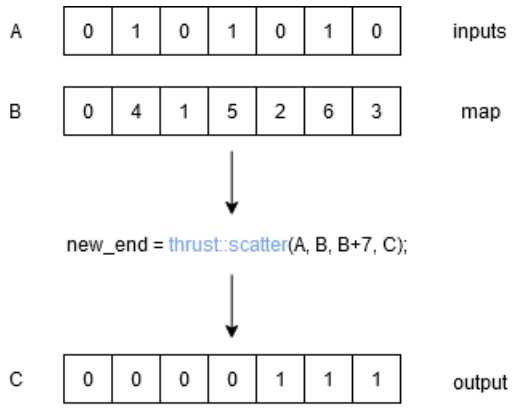

Figura B.6: Ejemplo de ejecución de operación *thrust::scatter*.

B.7. Sort

La primitiva *thrust::sort* ordena los elementos de un vector. Si bien en la biblioteca hay 4 variantes de esta misma operación (con distintos parámetros), la utilizada en este trabajo tiene la siguiente forma:

```
__host__ __device__ void thrust::sort (
        RandomAccessIterator     first,
        RandomAccessIterator     last,
        StrictWeakOrdering       comp
)
```

donde *first* y *last* son punteros al inicio y al final del arreglo de valores a ordenar y *comp* es la operación utilizada para la comparación entre los elementos.

B.8. Copy if

La primitiva *thrust::copy_if* copia los elementos de un vector *input* a un vector *result*, salvo los elementos que hagan falso el predicado *pred* pasado como parámetro. Si bien en la biblioteca hay 4 variantes de esta misma operación (con distintos parámetros), la utilizada en este trabajo tiene la siguiente forma:

```
OutputIterator thrust::copy_if (
        InputIterator     first,
        InputIterator     last,
        OutputIterator    result,
        Predicate         pred
)
```

donde *first* y *last* son punteros al inicio y al final del arreglo de valores a copiar, *result* es un puntero al inicio del vector donde se copiarán los valores y *pred* es el predicado utilizado para saber si copiar o no cada elemento. En la Figura B.7 se puede ver un ejemplo de ejecución de la operación.

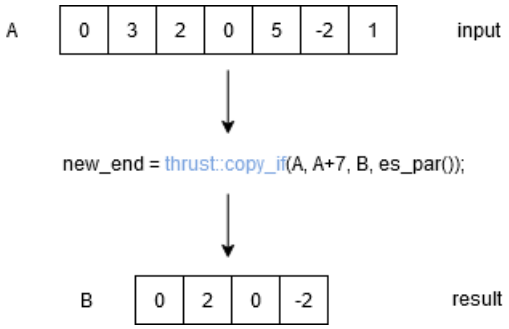

Figura B.7: Ejemplo de ejecución de operación *thrust::copy_if*.