Aplicación de estrategias Synchornization Free a la resolución de operaciones de Álgebra Lineal Dispersa

Manuel Freire Picón

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

Montevideo – Uruguay
Mayo de 2025

Aplicación de estrategias Synchornization Free a la resolución de operaciones de Álgebra Lineal Dispersa

Manuel Freire Picón

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

Director de tesis:
Dr. Ing. Ernesto Dufrechou

Codirector:
Dr. Ing. Pablo Ezzatti

Director académico:
Dr. Ing. Ernesto Dufrechou

Montevideo – Uruguay
Mayo de 2025

Freire Picón, Manuel

Aplicación de estrategias Synchornization Free a la resolución de operaciones de Álgebra Lineal Dispersa< / Manuel Freire Picón. - Montevideo: Universidad de la República, Facultad de Ingeniería, 2025.

xxii, 127 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 Informática, 2025.

Referencias bibliográficas: p. 117 – 127.

1. Álgebra dispersa, 2. Synchronization Free, 3. Computación de alta performance, 4. CUDA, 5. GPU.

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE TESIS

Dra. Lorena Etcheverry

Dr. Gustavo Betarte

Dr. Francisco Igual

Montevideo – Uruguay
Mayo de 2025

RESUMEN

El Álgebra Lineal Numérica dispersa es fundamental en muchas áreas de la computación científica, desde la simulación numérica hasta el aprendizaje automático. Sin embargo, su implementación eficiente en hardware paralelo enfrenta desafíos debido a la baja intensidad computacional y la irregularidad en el acceso a memoria. En el contexto de los métodos iterativos para la resolución de sistemas lineales de gran tamaño, la resolución de sistemas triangulares dispersos (SpTRSV) y la factorización LU incompleta (ILU) representan gran parte del costo computacional.

Esta tesis explora el uso de estrategias synchronization-free para optimizar la ejecución de SpTRSV e ILU en GPUs. En SpTRSV, se estudian las limitaciones del enfoque tradicional basado en la generación de una estructura de niveles y se optimizan implementaciones que siguen el paradigma synchronization-free. Se propone una nueva estrategia para el ordenamiento del cómputo de las filas, así como un formato de almacenamiento diseñado específicamente para acelerar esta rutina. De manera similar, se desarrollan diversas implementaciones de ILU bajo el paradigma synchronization-free, empleando distintas estrategias para la etapa de actualización, el ordenamiento de filas y la distribución de la carga de trabajo entre los hilos.

Los métodos desarrollados son evaluados experimentalmente y comparados con soluciones del estado del arte en diferentes conjuntos de matrices dispersas. Los resultados muestran que las estrategias propuestas logran una reducción significativa en el tiempo de ejecución en SpTRSV y ofrecen resultados competitivos en la ILU.

En resumen, las técnicas synchronization-free demuestran ser una alternativa viable para optimizar el rendimiento en operaciones de álgebra dispersa que tienen dependencias de cómputo, evitando las penalizaciones asociadas a la sincronización explícita y al sobrecosto de lanzamiento de kernels.

Palabras claves:

Álgebra dispersa, Synchronization Free, Computación de alta performance, CUDA, GPU.

ABSTRACT

Sparse Numerical Linear Algebra is fundamental in many areas of scientific computing, from numerical simulation to machine learning. However, its efficient implementation on parallel hardware faces challenges due to its low computational intensity and irregular memory access. In the context of iterative methods for solving large linear systems, sparse triangular system solving (SpTRSV) and incomplete LU factorization (ILU) account for a large part of the computational cost.

This thesis explores the use of synchronization-free strategies to optimize the execution of SpTRSV and ILU on GPUs. In SpTRSV, the limitations of the traditional approach based on generating a level structure were studied, and implementations that follow the synchronization-free paradigm were optimized. A new strategy for ordering row computations was proposed, as well as a storage format specifically designed to accelerate this routine. Similarly, several ILU implementations were developed under the synchronization-free paradigm, employing different strategies for the update phase, row ordering, and workload distribution among threads.

The developed methods were experimentally evaluated and compared with state-of-the-art solutions on different sets of sparse matrices. The results show that the proposed strategies achieve a significant reduction in execution time in SpTRSV and offer competitive results on the ILU.

In summary, synchronization-free techniques prove to be a viable alternative for optimizing performance in sparse algebra operations that have computational dependencies, avoiding the penalties associated with explicit synchronization and the overhead of kernel launches.

Keywords:
Sparse algebra, Synchronization Free, High performance computing, CUDA, GPU.

Lista de figuras

Lista de tablas

Tabla de contenidos

Capítulo 1

Introducción

El Álgebra Lineal Numérica es una piedra angular de la computación científica, teniendo un rango de aplicación en campos que van desde la física a la ciencia de datos [10, 11]. Muchas tareas como simulaciones o redes neuronales tienen su cuello de botella en rutinas de álgebra lineal. El álgebra dispersa añade desafíos no presentes en el álgebra densa, en especial relativos a su baja intensidad computacional (ratio de operaciones de punto flotante por byte accedido a memoria) y de no localidad de datos (los datos requeridos en cada punto no están necesariamente cercanos en memoria) [12].

Siguiendo la metodología empleada en el álgebra densa, en la que los métodos numéricos se construyen en términos de un conjunto acotado de operaciones básicas determinado por la especificación BLAS (Basic Linear Algebra Subprograms) [13], el álgebra dispersa también cuenta con su juego de operaciones fundamentales. Debido a las características de los algoritmos sobre matrices, este conjunto es bastante más reducido que el determinado por BLAS. Del análisis de los métodos más utilizados para resolver sistemas de ecuaciones lineales, así como problemas de vectores y valores propios sobre matrices dispersas, surge que los principales cuellos de botella desde el punto de vista computacional ocurren en la solución de sistemas de ecuaciones triangulares (operación denominada $SpTRSV$) y, en menor medida, en el producto matriz-vector ($SpMV$) [14]. Además, una importante familia de métodos para resolver problemas dispersos de gran tamaño (los métodos llamados *iterativos*), frecuentemente se valen de factorizaciones aproximadas o incompletas de la matriz dispersa para acelerar su convergencia, lo que agrega operaciones como la Factorización LU Incompleta (ILU) al conjunto de rutinas fundamen-

tales sobre matrices dispersas.

Si bien las operaciones $SpMV$, $SpTRSV$ e ILU, no son las únicas operaciones básicas definidas sobre matrices dispersas, sí gozan de una relevancia para la ciencia y la ingeniería que ha atraído la atención de la comunidad dedicada a la computación de alto desempeño durante un largo tiempo. Como consecuencia, la implementación paralela de la $SpTRSV$ y la ILU ha sido estudiada para varias plataformas de hardware. En particular, el desarrollo de las GPU en la última década ha impulsado los trabajos sobre esta plataforma. Las GPUs son plataformas muy utilizadas en el álgebra lineal debido a su gran poder de cómputo y, especialmente en el caso del álgebra lineal dispersa, su gran ancho de banda de memoria. Sin embargo, para sacar provecho de las GPUs es necesario mantener una gran ocupación (tener suficiente paralelismo para mantener una gran cantidad de procesadores ocupados) y explotar eficientemente el ancho de banda de acceso a memoria.

En el caso de las operaciones $SpTRSV$ e ILU, un obstáculo importante para lograr lo anterior es el manejo de las dependencias entre el procesamiento de distintas filas de la matriz. Esto implica el empleo de distintos mecanismos de sincronización entre hilos, lo cual agrega un costo que se no se encuentra en otras operaciones como la $SpMV$. Durante años, el paradigma dominante para el manejo de esta sincronización en diversos tipos de hardware consistió en el preprocesamiento de la matriz para determinar un esquema fijo de ejecución de los hilos donde se respetaran estas dependencias [15, 16]. En los últimos tiempos, un nuevo conjunto de algoritmos, denominados Synchronization-Free (*sync-free*), ha ganado relevancia. A diferencia de los anteriores, estos se basan en planificar la ejecución de los hilos conforme avanza el procesamiento de la matriz, obligando a algunos hilos a esperar hasta que otros resuelven sus dependencias. En estos últimos se centra la temática de la tesis.

1.1. SpTRSV e ILU en el ALN dispersa

Uno de los ejemplos más relevantes del ALN dispersa es la resolución de ecuaciones en derivadas parciales (PDEs por sus siglas en inglés -Partial Differential Equations-). Estas ecuaciones son una herramienta muy importante para modelar diversos problemas que abarcan áreas tan dispares como la física o la economía [17, 18]. Sin embargo, la complejidad y el número de parámetros involucrados suelen hacer inviables las soluciones analíticas. Por ello, la estra-

tegia más común es discretizar el dominio del problema en una malla finita de elementos o puntos, lo cual es la base de los métodos de diferencias finitas, volúmenes finitos o elementos finitos. Este tipo de estrategias genera grandes sistemas de ecuaciones lineales del tipo $Ax = b$, donde A es la matriz de coeficientes, x el vector de las incógnitas, y b el de los valores independientes. Debido a que los coeficientes expresan relaciones entre partes cercanas del dominio, la matriz A tiende a tener grandes dimensiones y una baja proporción de coeficientes distintos de cero ya que cada punto de la malla se relaciona con los elementos cercanos.

La resolución de estos sistemas es, en general, la parte más costosa de estos métodos, tanto en cuanto a tiempo de cómputo como a consumo de memoria [14]. Para resolverlos, existen dos grandes enfoques: directo e iterativo. El primero consiste en aplicar una serie de transformaciones a la matriz para llegar a un sistema equivalente pero más fácil de resolver (por ejemplo, las estrategias basadas en la eliminación Gaussiana llevan a una matriz diagonal) [19]. La gran ventaja de estos métodos es que obtienen resultados exactos (a menos de los errores inevitables debido al uso de punto flotante). Los métodos iterativos, por otro lado, parten de una solución inicial y la mejoran en iteraciones sucesivas hasta cumplir cierto criterio de calidad de la solución (o exceder un máximo número de iteraciones). Estos métodos se convierten en la alternativa viable con sistemas de gran escala, ya que los métodos directos no escalan bien con el tamaño del problema y sus requerimientos (cómputo y memoria) se vuelven prohibitivos en sistemas de gran tamaño [14].

Existen dos grandes grupos de métodos iterativos: estacionarios y métodos de Krylov. El primer grupo son métodos que reformulan el sistema $Ax = b$ sustituyendo A por $M - N$, con M siendo una matriz sencilla de invertir. Esto permite reescribir el sistema como $(M-N)x = b$ generando la iteración $x^{k+1} = x^k + M^{-1}(b - Ax^k)$. Ejemplos de estos grupos incluyen Jacobi y Gauss-Seidel. La principal desventaja de estas estrategias es que convergen únicamente en casos específicos.

Por otro lado, los métodos de Krylov, como el método del Gradiente Conjugado (CG) y GMRES, construyen un subespacio a partir de las potencias sucesivas del sistema aplicadas al vector b. El objetivo es aproximar la solución del sistema dentro de estos subespacios, minimizando el error ($e^k = x - x^k$) o el residuo ($r^k = b - Ax^k$) según el criterio específico del método. Este enfoque permite una adaptación iterativa que, en términos teóricos, puede garantizar la

convergencia bajo ciertas condiciones [20]. Sin embargo, en sistemas grandes la convergencia de estos métodos puede ser muy lenta o no ocurrir en la práctica. Es decir, el progreso que se hace en cada iteración es, en el mejor de los casos, muy pequeño. La convergencia depende sustancialmente de la distribución de los valores propios, lo que se refleja en el número de condición (el coeficiente de los valores propios de mayor y menor magnitud de la matriz). Si este número es muy alto, se dice que el problema está mal condicionado, lo que implica que pequeñas variaciones (o errores) en los datos de entrada generan grandes diferencias en los resultados [21, 22].

Para mejorar esta convergencia se utiliza el precondicionamiento. En algunos casos se puede aprovechar información específica del problema para diseñar precondicionadores adecuados al mismo. Este conjunto de precondicionadores, conocidos como geométricos, utiliza información de la malla o discretización del dominio para mejorar la convergencia del método iterativo. Sin embargo, en muchos casos no se cuenta con información de la geometría subyacente del problema. En este contexto, se utilizan los precondicionadores algebraicos, los cuales se construyen directamente a partir de la matriz del sistema [14].

Una forma de ver el precondicionamiento consiste en multiplicar al sistema $Ax = b$ de los dos lados por una matriz M^{-1} tal que el número de condición de $M^{-1}A$ mejore, y resolver el sistema equivalente $M^{-1}Ax = M^{-1}b$. En este sentido, idealmente la matriz M^{-1} es "relativamente similar" a A^{-1} o, lo que es lo mismo, se quiere hallar una matriz M similar a A pero fácilmente invertible. Por lo anterior, una buena elección de M tiene implicancia directa en la cantidad de iteraciones que requerirá el método iterativo. Si bien la multiplicación del sistema por M^{-1} reduce el número de iteraciones, implica la necesidad de invertir la matriz M, algo potencialmente mucho más costoso, ya que generalmente M^{-1} será una matriz densa. En general, la multiplicación del sistema por M^{-1} no se hace de manera explícita sino que en su lugar se obtiene una aproximación dispersa $M^{-1} \approx A^{-1}$ y se realiza el producto matriz-vector $z^k = M^{-1}r^k$ en cada iteración del método, o se obtiene una factorización de $M \approx A$ y resuelve el sistema equivalente $Mz^k = r^k$, donde r^k sería el residuo de cada iteración. Este comportamiento puede observarse en la línea 7 del Algoritmo 1, donde se muestra un pseudocódigo del método de Gradiente Conjugado Precondicionado (PCG por sus siglas en inglés -Preconditioned Conjugate Gradient-).

Uno de los precondicionadores algebraicos más utilizados es la factorización

Algoritmo 1 Pseudocódigo del método del gradiente conjugado precondicionado.

```
// In: A: Matriz simétrica y definida positiva, b: Vector de resultados
// In: M: Precondicionador, x0: Solución inicial, tol: Tolerancia, max_iter: Máximo número de iteraciones
// Out: x: Solución aproximada

//Inicializaciones
r_0 = b - A × x_0
Resolver M z_0 = r_0 // Solución de sistema triangular en vez de M^{-1} r_0
p_0 = z_0
x = x_0
for k = 0  to  max_iter do:
    α_k = (r_0^T × z_0)/(p_0^T × A × p_0) // Cálculo de α_k
    x = x + α_k × p_0 // Actualización de la solución
    r_1 = r_0 - α_k × A × p_0 // Cálculo del nuevo residuo

    // Si converge retornar
    if ||r_1|| < tol do:
        return x
    endif

    Resolver M z_1 = r_1 // Solución del sistema en lugar de multiplicar por M^{-1}
    β_k = (r_1^T × z_1)/(r_0^T × z_0) // Cálculo de β_k
    p_1 = z_1 + β_k × p_0 // Actualización de la dirección de búsqueda

    // Actualizaciones
    r_0 = r_1
    z_0 = z_1
    p_0 = p_1
endfor
return x
```

LU incompleta (ILU). Esta operación consiste en hallar dos matrices dispersas (L, triangular inferior y U, triangular superior) que cumplan que $LU \approx A$. La ventaja de definir $M = LU$ es que la resolución del sistema $Mz^k = r^k$ es equivalente a hacer dos resoluciones triangulares: $Ly = r^k$ y $Uz^k = y$.

Gran parte del costo de las iteraciones (y por ende de la resolución de $Ax = b$) queda definido por la aplicación del precondicionador sobre el residuo. Por esto, los *kernels* para computar el precondicionador (ILU) y la resolución

del sistema triangular disperso ($SpTRSV$) tienen una gran importancia[1].

1.2. Plataformas heterogéneas de Hardware y GPUs

La evolución de la computación ha estado guiada por el objetivo de aumentar la velocidad y capacidad de procesamiento o, en otras palabras, el poder de cómputo. Los primeros sistemas de computación contaban con una única unidad central de procesamiento (CPU, por sus siglas en inglés -Central Processing Unit-), lo que implicaba que el procesamiento de las tareas debía realizarse de manera secuencial. En este contexto, la estrategia predominante para mejorar el desempeño consistió en aumentar la frecuencia de reloj de los procesadores. Sin embargo, este enfoque encontró límites físicos debido al incremento del consumo energético y la generación de calor, los cuales comenzaron a contrarrestar las ganancias en velocidad de cómputo. Ante estas limitaciones, la industria viró hacia arquitecturas capaces de ejecutar múltiples tareas en paralelo, lo que marcó el inicio de la computación paralela como un paradigma dominante [23].

A principios de la década de 2000, comenzaron a surgir distintos dispositivos de hardware especializados, como las GPUs, que revolucionaron el campo del cómputo de alto desempeño. Originalmente diseñadas para acelerar el procesamiento gráfico en videojuegos, las GPUs se caracterizan por contar con una gran cantidad de unidades de procesamiento dedicadas a realizar cálculos en paralelo. Esta arquitectura motivó su exploración y eventual adopción en el cálculo científico y otras aplicaciones generales, inicialmente mediante la adaptación de implementaciones a las operaciones gráficas predefinidas. Posteriormente, con el desarrollo de entornos de programación como CUDA y OpenCL, se amplió su uso para resolver problemas de cómputo intensivo en diversas disciplinas, desde la simulación numérica hasta la inteligencia artificial. En particular, las GPUs son altamente eficientes cuando se requiere realizar la misma operación sobre un conjunto masivo de datos relativamente independientes, una característica que las hace idóneas para algoritmos con alto grado de paralelismo. En estos dispositivos, las tareas se organizan en *kernels* (funciones que ejecuta en la GPU), que son despachados para ejecutarse en

[1] Junto con la multiplicación matriz dispersa - vector ($SpMV$).

paralelo por grandes cantidades de hilos que repiten el mismo cómputo sobre distintos datos.

El crecimiento y diversificación de las plataformas de hardware propició el advenimiento del cómputo heterogéneo, donde distintos tipos de procesadores con capacidades especializadas trabajan de manera conjunta dentro de un mismo sistema [24]. Esta estrategia permite aprovechar las ventajas de cada arquitectura: por ejemplo, las CPUs son adecuadas para la ejecución de tareas secuenciales y operaciones con alta dependencia de control, mientras que las GPUs destacan en cargas de trabajo altamente paralelizables y de gran volumen de datos. Además de las GPUs, han emergido otros aceleradores especializados como las *Field-Programmable Gate Arrays* (FPGAs) y las *Tensor Processing Units* (TPUs), optimizados para aplicaciones específicas como el procesamiento de señales o el aprendizaje profundo.

No obstante, el uso eficiente de plataformas heterogéneas presenta desafíos significativos. La comunicación y coordinación entre los distintos procesadores introduce costos de sincronización, los cuales pueden ser significativos en aplicaciones con alta interdependencia entre hilos o con baja intensidad computacional. En sistemas masivamente paralelos, la sincronización entre hilos puede convertirse en un cuello de botella, afectando el rendimiento global si no se emplean estrategias específicas para mitigar su costo. Por otro lado, la transferencia de datos entre dispositivos de diferentes arquitecturas introduce latencias adicionales. Para abordar estos desafíos, debe ponerse foco en el desarrollo de estrategias que reduzcan el costo de la sincronización y la comunicación en general.

1.3. Métodos Synchronization-Free

Para resolver un sistema triangular de manera exacta, el procedimiento que se emplea es el de sustitución hacia adelante (cuando es una matriz triangular inferior) o hacia atrás (cuando es una matriz triangular superior). Este procedimiento consiste en recorrer las ecuaciones en orden, sustituyendo todas las incógnitas correspondientes a las ecuaciones previas por los valores ya calculados.

En el caso de la factorización LU (tanto completa como no), se debe transformar A usando la eliminación Gaussiana, es decir, transformando en 0 los elementos debajo de la diagonal mediante la resta de múltiplos de las filas

anteriores. Los multiplicadores usados para crear estos ceros se almacenan en L, mientras que U guarda los coeficientes restantes. La diferencia entre la factorización LU e ILU es que en la segunda se garantiza que las matrices L y U sean dispersas[2] aunque la multiplicación de estas dos matrices no resulta exactamente en la obtención de la matriz original, sino una "relativamente parecida". Al igual que en la $SpTRSV$, las filas son recorridas en orden y de izquierda a derecha.

Ambas operaciones tienen características que hacen que no sea trivial generar implementaciones paralelas de dichas rutinas. Por ejemplo, su baja intensidad aritmética (son problemas *memory bound*), la falta de equilibrio en la carga (el procesamiento de algunas filas requiere significativamente más tiempo y cómputo que otras) y las dependencias entre las distintas tareas. En particular, estas dependencias son una gran limitante del paralelismo. En matrices densas, el paralelismo puede darse al impactar el resultado de una fila i que se ha terminado de procesar, en múltiples filas a la vez, ya sea para sustituir la incógnita ($TRSV$) o hacer 0 el valor (LU). Sin embargo, las filas no pueden procesarse completamente en paralelo ya que, para impactar los valores de una fila, esta debe haberse terminado de procesar. Por el contrario, cuando se trabaja con operaciones de matrices dispersas, la mayoría de las filas dependen solamente de un conjunto reducido de filas, por lo que existen conjuntos de filas independientes que pueden ser procesadas en paralelo. En concreto, en ambas rutinas toda fila i depende de una fila j ($j < i$) si tiene una entrada no-cero en la columna j, ya que requiere el valor de la incógnita x_j (o del pivot, en ILU) para poder calcular sus propios valores. En matrices en las que existen muchas dependencias, o dependencias transitivas que serializan el cálculo (como por ejemplo, matrices tri-diagonales), el costo de sincronización puede ser relevante en el total de la rutina, contrarrestando las ganancias producto del paralelismo.

En GPU existen dos grandes paradigmas para implementar la $SpTRSV$ y ordenar el cómputo de las filas según sus dependencias: *level-set* [16] y *self-scheduled* (o *sync-free*) [25]. El primero se compone de dos etapas: primero se ejecuta un preprocesamiento (análisis) y, posteriormente, se procesa el cálculo dividido en varios pasos, cada uno de los cuales procesa un conjunto de filas en

[2]En el caso de la ILU-0 el patrón de dispersión se mantiene exactamente igual a A mientras que en ILU-k se permite cierto grado de *fill-in* (es decir, de que algunas entradas que eran 0 en la matriz A tengan un no-cero en L o U).

paralelo (resolución). La etapa de análisis genera conjuntos llamados *niveles* de filas que pueden ejecutarse en paralelo (no dependen entre ellas) y únicamente dependen de filas en niveles inferiores. Luego, la etapa de resolución lanza un *kernel* que procesa las filas del primer nivel; una vez terminado, se lanza otro para el segundo y así sucesivamente. Este orden de ejecución garantiza que cuando una fila es procesada, todas sus dependencias están resueltas, y permite evitar *deadlocks*[3]. Si bien el preprocesamiento puede ser costoso, este puede ser reutilizado para sucesivas resoluciones con distintos valores de b u otras operaciones sobre una matriz con el mismo patrón de dispersión (variación de los coeficientes) generando que el costo sea compensado. En este paradigma, el costo de sincronización está dado por el de lanzamiento de cada uno de los *kernels* que, si bien individualmente es insignificante, en contextos con muchos niveles y pocas filas por nivel, puede acumularse y convertirse en una parte significativa del tiempo total.

El paradigma *sync-free* busca resolver la problemática mencionada anteriormente y eliminar el costo de sincronización entre los niveles. Para esto, se elimina el preprocesamiento y se procesan las filas en el orden dado por la matriz. Si bien el orden de ejecución también evita los *deadlocks* (cuando una fila i comienza a ejecutar, ya lo hicieron todas las filas j con $j < i$ de las que podría depender) a priori podría generar condiciones de carrera[4]. Para evitar esto, los hilos que ejecutan cada fila chequean si las dependencias están cumplidas mediante *polling* (o espera activa) antes de procesarla. Las estrategias de espera activa (*busy waiting*) generalmente no son recomendables, ya que implican un gran costo al utilizar tiempo de cómputo en consultar constantemente por la disponibilidad de un recurso. Sin embargo, gracias a la forma en que el hardware de la GPU planifica la ejecución de los hilos, en la práctica se generan bloqueos implícitos al hacer estas consultas y otros hilos pueden ejecutar mientras unos preguntan por sus dependencias.

[3]Situación dentro de un sistema informático en la que dos o más tareas por ejemplo: hilos o procesos se esperan mutuamente para terminar o continuar su trabajo. Esta situación detiene las tareas y les impide continuar pudiendo bloquear otras tareas o potencialmente el sistema entero. En el caso concreto del *SpTRSV* en GPU esto puede darse si todos los procesadores son utilizados para procesar filas que dependen de otras que están esperando para ejecutar

[4]Una condición de carrera es la condición de un sistema electrónico, software u otro sistema en el que el comportamiento sustantivo del sistema depende de la secuencia o el tiempo en el que suceden otros eventos incontrolables.

1.4. Objetivos y organización de la tesis

En este trabajo se aborda el uso de métodos *sync-free* en arquitecturas de hardware masivamente paralelas para acelerar operaciones de álgebra lineal dispersa con un importante costo de sincronización, como la $SpTRSV$ y la ILU. Concretamente, en el caso de la operación $SpTRSV$ se continúan esfuerzos presentados en [9, 26] para mejorar el desempeño computacional de la resolución de sistemas triangulares dispersos explorando nuevos patrones de cómputo y formatos de almacenamiento especializados. Se presentan dos contribuciones principales: un formato de almacenamiento especialmente diseñado para mejorar el acceso a memoria de la operación y una nueva estrategia para ordenar el cómputo de las filas. Los resultados experimentales muestran que estos cambios mejoran el tiempo de cómputo de las rutinas.

Posteriormente, producto de su patrón de dependencias similar, se explora la aplicación de técnicas *sync-free* a la factorización LU incompleta. Contrario a lo que ocurre con la $SpTRSV$, no se encontraron otros trabajos que apliquen el paradigma *sync-free* a esta operación, por lo que se dedicaron los esfuerzos a proponer una suite de rutinas con distintas estrategias y compararlas entre sí.

La estructura de este manuscrito es la siguiente. En el Capítulo 2 se presentan los conceptos básicos para la comprensión de esta tesis. Se explican las matrices dispersas con especial énfasis en estrategias para almacenarlas en memoria. También se introducen las características fundamentales de las GPUs, las principales operaciones del álgebra lineal dispersa y el paradigma de cómputo utilizado en este trabajo (*sync-free*). Posteriormente, en el Capítulo 3 se presenta una revisión del estado del arte en lo relativo a la aplicación del paradigma *sync-free* a distintas rutinas del álgebra lineal. Se constata que gran parte de los esfuerzos fueron puestos en la aceleración de la $SpTRSV$ pero también existen trabajos sobre otras rutinas. En el Capítulo 4 se describen los aportes relativos a la resolución de sistemas triangulares dispersos. A continuación, en el Capítulo 5 se compendian los aportes relativos a la ILU. Finalmente, en el Capítulo 6 se presentan las conclusiones de la tesis y posibles líneas de trabajo futuro.

Capítulo 2

Conceptos preliminares

En este capítulo se presentan los conceptos preliminares relevantes para la comprensión del trabajo de maestría. En la Sección 2.1 se presentan las matrices dispersas y, en especial, las estrategias para almacenarlas en memoria. En la Sección 2.2 se comentan las características fundamentales de las tarjetas de procesamiento gráfico (conocidas como GPU por sus siglas en inglés -Grafics Processing Units-), la principal plataforma de cómputo utilizada en esta tesis. Posteriormente, en la Sección 2.3 se describen las principales operaciones de álgebra dispersa [14, 19, 27]. Finalmente, en la Sección 2.4 se explica el paradigma de cómputo *self scheduled*.

2.1. Matrices Dispersas

No existe una definición única, o de consenso, para las matrices dispersas. Una idea intuitiva es que las matrices dispersas son matrices en las que la mayoría de sus coeficientes tienen valor 0[1]. Dado que la baja proporción de no-ceros es la única condición que debe cumplir una matriz para ser dispersa, esto genera que exista una amplia variedad de tipos de matrices según sus características y el campo de aplicación del que provienen [12]. Algunos ejemplos de los diversos campos de aplicación son la resolución de ecuaciones en derivadas parciales utilizando métodos de elementos finitos (FEM por sus siglas en inglés -Finite Element Methods-), la simulación de circuitos o las operaciones de grafos que se usan en redes sociales. La Figura 2.1 presenta un ejemplo de una matriz dispersa que cuenta con no-ceros en toda la diagonal.

[1]Puede ser dispersa respecto a otro valor.

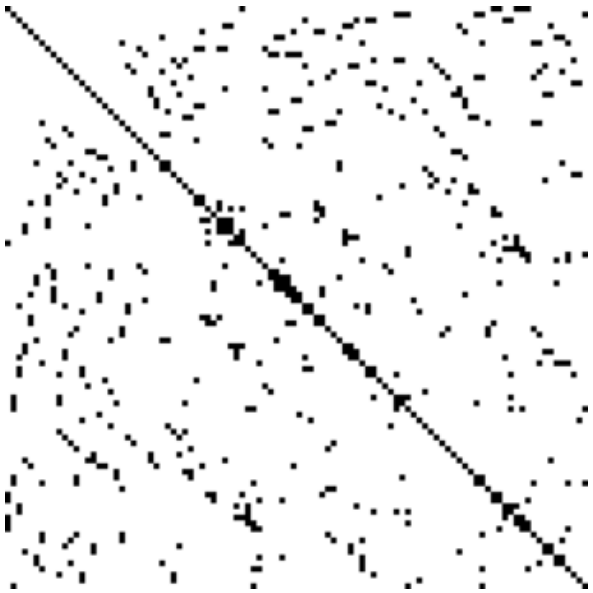


Figura 2.1: Matriz de ejemplo. Los puntos en negro representan no-ceros mientras que los puntos blancos representan ceros.

Las matrices dispersas pueden ser clasificadas según características como si son cuadradas o rectangulares, la cantidad de no-ceros, de filas o de columnas. Además, independientemente de estas métricas, existen características relativas al patrón de dispersión (cómo están organizados los coeficientes no-ceros) como, por ejemplo, si la matriz es simétrica, de banda, a bloques, etc. Finalmente, si se consideran los valores de los coeficientes y no únicamente el patrón de dispersión, se pueden caracterizar por ciertas propiedades numéricas como el número de condición, el rango, si es numéricamente simétrica o si es definida positiva. Las matrices de cierto tipo de aplicación en muchos casos comparten algunas características, por ejemplo, las matrices que surgen de ecuaciones en derivadas parciales con grillas regulares de dos dimensiones son esencialmente pentadiagonales debido a la forma en que se vinculan los elementos de la grilla [12].

2.1.1. Formatos de almacenamiento de matrices dispersas

Cuando se almacenan matrices densas no existen demasiadas alternativas. En general, se utiliza un arreglo de $n \times m$ elementos donde n y m con la cantidad de filas y columnas respectivamente. La única variable a considerar es si se almacenan contiguamente en memoria los elementos de una misma fila

(conocido como ordenamiento *row major*) o columna (*column major*). La gran ventaja de estas estrategias es que, una vez que se sabe el ordenamiento de los datos, los índices de fila y columna están implícitos en la posición de memoria, es decir, el elemento ij se encuentra en la posición $i \times m + j$ si es *row major* o $j \times n + i$ si no[2].

Esta estrategia de almacenamiento cuenta con dos principales ventajas. La primera y más evidente consiste en que es posible acceder en $O(1)$ a cualquier elemento si se tienen sus índices. Esto posibilita implementaciones sencillas para operaciones de álgebra numérica como la suma de matrices, multiplicación escalar o trasposición. La segunda ventaja es que este tipo de almacenamiento permite explotar la localidad espacial de los datos, ya que todos los elementos que están cercanos en la matriz están cercanos en memoria. Esto permite mejores patrones de acceso a memoria y uso de caché cuando se implementan las operaciones mencionadas anteriormente.

Por otro lado, las matrices dispersas generan desafíos adicionales a la hora de almacenarlas. Aunque la estrategia de utilizar un arreglo es eficiente en el caso denso, en el disperso generaría un desperdicio de memoria importante al almacenar una gran cantidad de entradas con valor cero. Incluso en el caso de que la cantidad de memoria no fuera un problema, esto también generaría dificultades en las operaciones; repetir las estrategias de álgebra densa implicaría una gran cantidad de cálculos inútiles al operar con ceros.

A diferencia del caso denso, el cómo almacenar matrices dispersas es un problema abierto y que está estrictamente relacionado con el patrón de matriz con el que se trabaja y la operación que se desea computar. A lo largo del tiempo han surgido distintos formatos de almacenamiento que buscan a la vez no almacenar los coeficientes con valor cero y tener patrones de acceso a datos que permitan implementaciones eficientes de las operaciones de álgebra. En otras palabras, la multiplicidad de factores a considerar (ahorro de memoria, desempeño de las operaciones, el patrón de dispersión de la matriz, características subyacentes del hardware en el que se ejecuta, entre otros) hace que no exista un formato de almacenamiento que sea mejor que los otros para todos los casos (ni todas las plataformas de hardware).

En este sentido, existen dos grandes enfoques. Por un lado, hay enfoques generalistas que buscan funcionar bien en un amplio abanico de matrices y operaciones en base a no explotar características particulares del patrón de

[2]Notación tipo C (o 0-based), en FORTRAN (1-based) sería $(i-1) \times m + j$

dispersión de las matrices en cuestión. En el otro extremo, es posible optimizar la estrategia de almacenamiento para aplicaciones o dispositivos específicos a costa de no ser transferibles a otros casos de uso.

A. Formato de Coordenadas (COO)

Quizá la estrategia más intuitiva entre las estrategias generalistas es el formato de coordenadas (COO). Este formato no almacena los ceros y mantiene los coeficientes en un arreglo unidimensional. Al no almacenar todos los coeficientes, es necesario asociar a cada uno los índices de fila y columna de manera explícita utilizando dos arreglos extra. En resumen el formato representa a una matriz como tres arreglos unidimensionales: $(data, row, col)$. Los dos últimos son arreglos de enteros que mantienen los índices de fila y columna respectivamente, mientras que el primero guarda los coeficientes distintos a cero, generalmente en punto flotante. Un esquema con una matriz de ejemplo puede verse en la Figura 2.2. Esta estrategia tiene dos grandes problemas. El primero es que se pierde la regularidad de los datos. Si se compara con el acceso directo que se tiene en las matrices densas se pasa de un acceso en $O(1)$ a uno $O(nnz)$[3] o, en caso de que esté ordenada $O(log(nnz))$. El segundo problema se muestra en la Figura 2.2 en donde los vectores de fila o columna se presentan entradas repetidas. En este caso, como los elementos están ordenados por fila, las primeras dos entradas guardan información repetida y lo mismo sucede con las últimas dos. Aunque se logra una importante reducción de memoria si se compara con la estrategia densa, si se utiliza punto flotante de doble precisión para los coeficientes distintos de cero y 4 bytes para almacenar los enteros, el 50 % de la memoria utilizada por COO es un sobrecosto de almacenar los índices, algo que en rutinas acotadas por memoria (como muchas operaciones de álgebra lineal dispersa) es un claro problema.

La reducción del *overhead* es una línea que ha motivado varios esfuerzos. Dang et al. propusieron el formato Sliced Coordinate format (SCOO) [28, 29] que subdivide la matriz en partes llamadas slices. Por otro lado Yan et al. propusieron el formato Block-based Compressed Common Coordinate (BCCOO) [30] que divide la matriz en bloques y luego almacena estos bloques utilizando COO lo que genera una reducción del sobrecosto de índices por elemento.

[3] Número de no-ceros en la matriz.

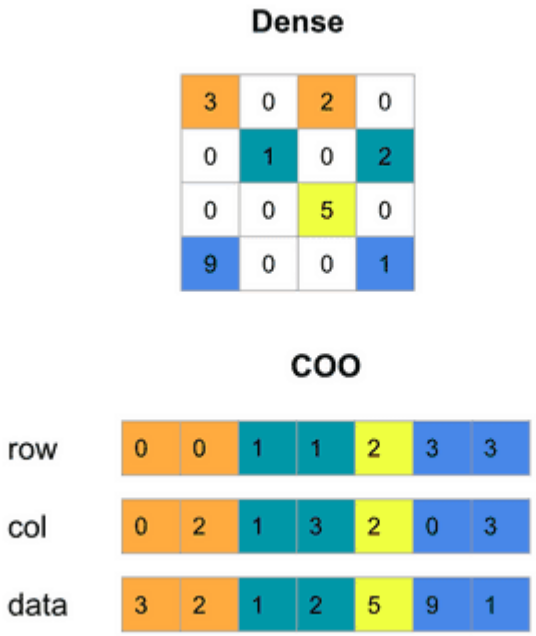


Figura 2.2: Esquema del formato Coordinate (COO) para una matriz de ejemplo. Extraído de [1].

B. Compressed Sparse Row (CSR)

El repetimiento de información en el vector de índices de fila en COO motiva el formato comprimido por fila o Compressed Sparse Row (CSR) [31]. CSR es uno de los formatos más utilizados en la práctica y, al igual que COO, sigue un enfoque generalista. Este formato sigue una idea similar a COO, ordenando los elementos según su índice de fila. Los dos primeros vectores, val y col_idx, son exactamente iguales que en COO y mantienen el valor del coeficiente e índice de columna respectivamente. Por el contrario, para eliminar la redundancia mencionada anteriormente, el tercer vector row_ptr mantiene el índice del primer valor de cada fila en los otros dos vectores. Es decir, el elemento i del vector row_ptr es el índice del primer valor en $data$ y col correspondiente a la fila i. Un ejemplo con la misma matriz se puede ver en la Figura 2.3. Esta estrategia logra una mayor compresión que COO ya que, para almacenar una matriz cuadrada de dimensión n con nnz coeficientes distintos de cero, en lugar de utilizar $16 \times nnz$ bytes, requiere $12 \times nnz + 4 \times n$ bytes. Este ahorro de memoria impacta directamente en la eficiencia de muchas rutinas de álgebra dispersa clasificados como *memory bound*. El ahorro aumenta cuanto mayor sea la cantidad de no-ceros promedio por fila. Por otro lado, CSR agrega

un nivel de indirección, algo que genera problemas especialmente cuando se quiere acceder por columna. En estos casos, dado que los valores no están contiguos en memoria en el peor de los casos puede recorrerse todo el arreglo *val*. Adicionalmente, si se quiere acceder a una entrada específica primero debe accederse al vector *row_ptr* y, una vez que se tiene el puntero a donde comienza la fila, se hace una búsqueda lineal por el índice de columna.

Al ser tan difundido CSR tiene muchas optimizaciones para distintas áreas. Por ejemplo, Feng et al. propusieron el formato SIC [32] que combina filas para optimizar la operación $SpMV$. Para esto agrupa de a c filas y alterna los elementos de éstas permitiendo un mejor acceso a memoria en las plataformas de hardware utilizadas. Cuando se utiliza un parámetro c que no es múltiplo de n se debe hacer *zero padding* al final. Otra estrategia en esta línea es el Compressed Multirow Storage format (CMRS) [33] de Koza et al. que también sigue una idea de agrupar las filas en una estructura que llaman *strip*. La ventaja de esta estrategia es que no usa *zero padding* ya que el *strip* no tiene un tamaño fijo.

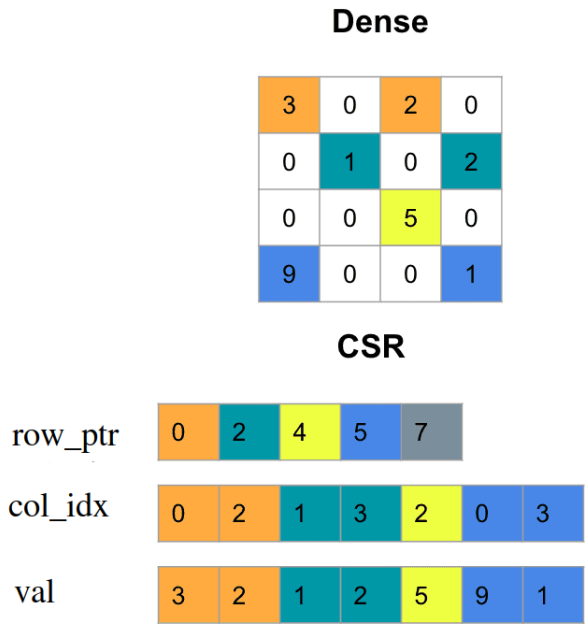


Figura 2.3: Esquema del formato Compressed Sparse Row (CSR) para una matriz de ejemplo. Extraído de [1].

Por último, CSC es un formato análogo a CSR, excepto que agrupa los no-ceros de la matriz por columnas y comprime el vector de índices de columnas

en lugar del vector de índices de fila.

C. ELLPACK-ITPAK (ELL)

Otra estrategia, que no tiene un enfoque tan generalista, es el formato ELLPACK-ITPAK (ELL). El formato almacena una matriz de tamaño $m \times n$ en un par de arreglos de tamaño $c \times n$ donde:

$$c = \max_{0 \leq j < m} \{nnz(j)\}, \tag{2.1}$$

y $nnz(j)$ es el número de elementos distintos a cero en la fila j. Esta estrategia busca regularizar la estructura a efectos de mejorar el rendimiento de distintas operaciones, ya que el acceso a los datos es similar al utilizado en álgebra densa. Como se verá más adelante en la tesis, esta técnica puede ser especialmente beneficiosa en GPUs NVIDIA para mejorar el acceso a memoria en operaciones como la $SpMV$.

El primer arreglo (val) mantiene los coeficientes de cada fila mientras que el otro (col) guarda los dos índices de columna. Como el objetivo es alinear los datos y generar que el índice de fila esté implícito las filas que tienen menos de c elementos son rellenadas con 0s al final en ambos arreglos. Al igual que en CSR los valores de uno y otro arreglo están relacionados por la posición en memoria, es decir que los elementos de la posición i en col y val tienen los índices de columna y valor del mismo coeficiente. Continuando con el mismo ejemplo, en la Figura 2.4 se muestra el formato ELL aplicada a la misma matriz de la Figura 2.2.

Es evidente que ELL tiene buenos resultados en casos, como el del ejemplo, donde la cantidad de no-ceros entre las distintas filas es similar. Por el contrario, en los casos de matrices que tienen unas pocas filas mucho más grandes que el resto el sobrecosto del *zero padding* se hace prohibitivo.

La poca adaptabilidad de ELL ha motivado varios trabajos. Una primera idea es el Sliced-Ell (SELL-C) [34] que divide la matriz en subpartes o *slices* para las que calcula un c_i local y el *zero padding* se hace hasta ese número y no hasta el c global. Esto reduce el número de ceros explícitos que se necesita guardar. Una mejora a SELL-C fue propuesta por Barbieri et al. quienes aplicaron reordenamientos previo al fraccionamiento y propusieron el formato hacked-ELL (HLL) [35]. Otra modificación (o generalización) a SELL-C fue propuesta por Kreutzer et al. [36]. Finalmente, Antz et al. propusieron el for-

Figura 2.4: Esquema del formato ELL para una matriz de ejemplo. Extraído y adaptado de [1]

mato Sliced ELLPACK with Padding (SELL-P) [37] que mejora el acceso a memoria de SELL-C. Este formato introduce *padding* adicional para permitir que la longitud de cada *slice* sea múltiplo del tamaño de *warp* permitiendo acceso alineados a memoria.

Para reducir debilidades de algún formato específico es común intentar combinar formatos complementarios. Un ejemplo en esta línea es el formato HYB [38] propuesto por Bell y Garland que combina ELL y COO. En concreto se elige un valor k tal que 2/3 de las filas sean más cortas que k y se toma $c = k$. Por el contrario, para las filas más largas que k los últimos elementos son guardados en formato COO.

D. Formatos a bloques

Otra línea de trabajo popular para reducir el sobrecosto de almacenar los índices explícitamente es el uso de formatos a bloques. La idea general consiste en subdividir la matriz en bloques regulares y tratarlos como unidades de una matriz de bloques almacenada en un formato de almacenamiento disperso. Cada bloque, a su vez, se almacena en un formato denso (incluyendo los ceros

que contenga) o en un formato que permita acceder a sus elementos con un menor costo de almacenamiento de índices. Los formatos que utilizan esta premisa tienen buen rendimiento en matrices que tienen los coeficientes no-cero agrupados en ciertas partes de la matriz y muchas áreas sin ningún elemento. Por el contrario, si los coeficientes no-cero se encuentran repartidos de manera más equitativa a lo largo de la matriz el costo de almacenar ceros de manera explícita compensará la ganancias en los índices. Algunas ideas en esta línea fueron presentadas en los apartados anteriores, a continuación se presentan otros trabajos que usan esta estrategia.

Weber et al. propusieron BIN-CSR, un formato que agrupa las filas en bloques que se almacenan en ELL [39]. El formato funciona bien si el largo promedio de las filas de un mismo bin (grupo de filas) es similar al máximo largo. Por otro lado, Yan et al. presentaron el formato Block-based Compressed Common Coordinate [30]. El formato usa la idea de COO a nivel de bloques. Otra estrategia en este sentido es BCSR [40] que fue propuesto por Choi et al. en 2010. Similar al formato anterior aplica CSR a la matriz de bloques.

La revisión que se presenta en en este apartado no pretende ser un estado del arte exhaustivo de formatos de almacenamiento sino que busca explicar algunas de las propuestas más relevantes y que guardan relación con el trabajo. Existen distintos formatos de almacenamiento que buscan explotar distintas características específicas de un subconjunto de matrices. En esta línea, se han propuesto formatos como DIA o JAD [41] enfocados en matrices con estructuras diagonales. Una revisión más detallada se puede encontrar en [42].

2.2. Unidades de Procesamiento Gráfico (GPU)

Las unidades de procesamiento gráfico son una de las arquitecturas de hardware más utilizadas en la última década en el área de la computación de alto rendimiento (HPC por sus siglas en inglés -High Performance Computing-). El desarrollo de estos dispositivos, compuestos por una gran cantidad de procesadores capaces de ejecutar un elevado número de cálculos de forma paralela, fue impulsado, inicialmente, por la industria de los videojuegos. A diferencia de lo que ocurre con las CPU la gran mayoría de los transistores de una GPU son dedicados a cómputo en lugar de otras tareas (predicción de *branches*, cache

de datos, etc.). Esta capacidad de cómputo paralelo y su relativamente bajo costo económico en comparación con otras plataformas de HPC propiciaron que las GPUs se comenzaran a adoptar para el cómputo de propósito general, es decir, tareas no relacionadas con el despliegue de gráficos en la pantalla.

Originalmente, hacer cómputo general en GPU tenía una gran dificultad, ya que se debía mapear las implementaciones a operaciones definidas en el *pipeline* gráfico [43]. Esta situación generaba que solamente pequeños grupos de expertos fueran capaces de sacar provecho de la arquitectura de las GPUs. En el año 2006, NVIDIA presentó la arquitectura CUDA (*Arquitectura Unificada de Dispositivos de Cómputo* por sus siglas en inglés) que permitió desarrollar rutinas de cómputo general en GPU independientemente de las primitivas de *pipeline* gráfico. CUDA es a la vez una arquitectura de cómputo paralelo y cuenta con una API que permite desarrollar rutinas de cómputo general que se ejecuten en tarjetas gráficas de NVIDIA utilizando una variación del lenguaje C (CUDA C). Esto motivó su adopción masiva por parte de la comunidad de cómputo científico. NVIDIA presentó su paradigma SIMT (*Single Instruction Multiple Thread* con puntos en común con el SIMD de la taxonomía de Flynn [44]) para describir la arquitectura de estos dispositivos.

A nivel de hardware, los núcleos de cómputo de las GPUs se organizan en un conjunto de *Streaming Multiprocessors* (SM). Los SM están compuestos por un grupo de procesadores que comparten registros y distintas memorias (cache L1, memoria compartida, cache constante y cache de texturas). Dependiendo de la arquitectura, cuentan con uno o más *warp scheduler*[4] Los SMs se agrupan en GPU Processing Clusters (GPCs). Estos son, básicamente, un conjunto de Texture Processing Clusters (parejas de SMs, no tienen una correspondencia con el modelo de programación de CUDA) más una *raster engine* (una unidad responsable de convertir gráficos vectoriales -definidos por ecuaciones matemáticas- en imágenes compuestas de píxeles). Toda esta estructura es transparente al programador y, por lo tanto, con el correr de las diferentes arquitecturas NVIDIA ha ido variando tanto la cantidad de procesadores por SM como la cantidad de SMs por GPC. Por lo anterior, no existen números generales que se mantengan a lo largo del tiempo, como sí existen en algunos agrupamientos a nivel de *software* como el *warp* [5].

[4]Unidad que gestiona la ejecución de un *warp* como una unidad aunque ejecute en más de un ciclo de reloj. Similar a un *program counter* en CPU pero con la complejidad inherente del paralelismo.

[5]En CUDA, un *warp* es el conjunto básico de ejecución: 32 hilos que ejecutan de manera

Con respecto a la memoria, todas las tarjetas cuentan con una memoria RAM llamada memoria global compartida por todos los SM y cuya latencia es alta, ya que es una memoria *off-chip*. Para el manejo de esta memoria CUDA brinda ciertas funciones como *cudaMalloc*, *cudaFree* y *cudaMemcpy* para reservar, liberar y copiar memoria respectivamente. También *off-chip* se cuenta con las memorias constante y de texturas, ambas son de solo lectura, es decir, solo pueden ser escritas por el programa que ejecuta en la CPU. Pueden ser utilizadas para almacenar parámetros que vayan a ser utilizados en muchas ejecuciones. Finalmente, se cuenta con una caché L2, global para todas las SM.

Un segundo nivel en la jerarquía de memoria es la memoria compartida. Esta memoria es interna de cada SM y, al ser *on-chip*, tiene mucho mayor velocidad y ancho de banda que las memorias *off-chip*. La memoria compartida toma su nombre del hecho de que todos los hilos que se ejecutan en la misma SM tendrán acceso a ella y podrán cooperar utilizándola. A su vez, las SM cuentan con una caché L1 privada para cada una de ellas.

Finalmente, el último nivel de la jerarquía son los registros, que son la memoria más escasa y con menos latencia.

2.2.1. Arquitectura de dispositivo unificada de cómputo (CUDA)

Los programas en CUDA cuentan con dos partes definidas: el código que ejecuta en la CPU (también llamado código *host*) y los *kernels* que ejecuta en la GPU (código *device*). Generalmente los *kernels* son invocados por código *host*, por ejemplo un *main* en CPU. Los *kernels* se definen mediante la directiva __global__. Al momento de invocar un *kernel* el código *host* explicita cuántos hilos (y con qué organización) ejecutarán dicha rutina, algo conocido popularmente como la grilla de ejecución. La grilla se organiza como un arreglo tridimensional de agrupaciones de hilos llamadas bloques. Estos bloques también son tridimensionales y están compuestos por hasta 1024 hilos. Finalmente, los hilos se agrupan en conjuntos de 32 hilos llamados *warps*.

Notar que los *kernels* son ejecutados por todos los hilos en paralelo; la diferencia de comportamiento entre los hilos se debe al uso de sus identificadores para acceder a memoria o evaluar *branches* lógicas. Por lo anterior, CUDA es una instancia específica del paradigma de programación de siste-

colaborativa.

mas paralelos SPMD (Un Programa Múltiples Datos por sus siglas en inglés) [45]. Esta instancia, como se dijo anteriormente, es conocida como *Single Program Multiple Threads* (SPMT) La cantidad y organización de los hilos que ejecutan un *kernel* se define al momento de la invocación, poniendo $\lll GridDims, BlockDims, Ns \ggg$ entre el nombre de la función y la lista de parámetros. La variable $GridDims$ es de tipo $dim3$[6] y se utiliza para especificar el tamaño de la grilla en bloques en las tres dimensiones $(GridDims.x, GridDims.y, GridDims.z)$. La variable $BlockDims$ es análoga para el tamaño del bloque en hilos.

Los hilos son identificados por dos ternas de índices, el primero (que se accede con la variable `blockIdx`) determina los índices en el eje x, y y z del bloque en el que se encuentra, el segundo (que se accede con la variable `threadIdx`) lo identifica dentro del bloque también en los índices en el eje x, y y z. Los hilos de un mismo bloque cuentan con una memoria común propia llamada *memoria compartida*. Por último, los hilos pueden acceder al tamaño del bloque utilizando la variable `blockDim`. En el Algoritmo 2 se presenta un código de ejemplo que suma dos vectores elemento a elemento utilizando N bloques de 256 hilos. En este ejemplo solamente se utiliza la dimensión del eje x. Para una visualización gráfica en la Figura 2.5 se presenta el esquema del modelo de ejecución del mismo código. Puede verse que la única diferencia que habrá entre los distintos hilos es el índice i que generan a partir de su id de bloque e hilo que les generará una posición única en el vector.

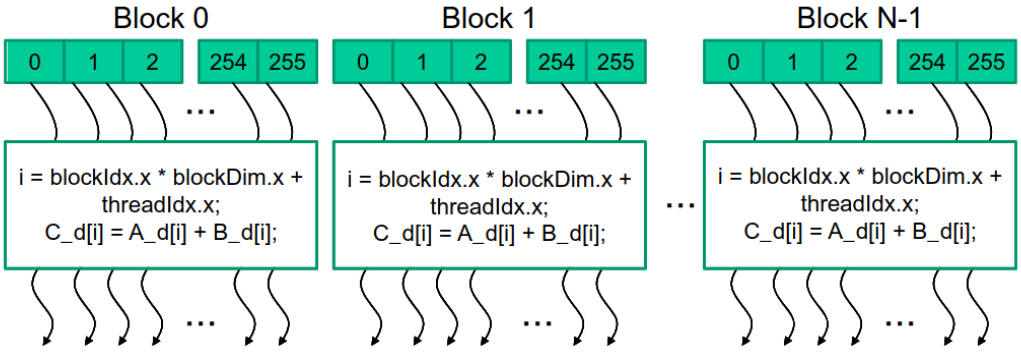


Figura 2.5: Modelo de ejecución de dos niveles de CUDA. Todos los hilos ejecutan el mismo código. Tomado de [2]

Si bien las GPUs cuentan con memorias rápidas como la memoria de textu-

[6]Struct de tres valores enteros

Algoritmo 2 Declaración de un kernel de ejemplo y su correspondiente programa principal

```
__global__ void VecAdd(float* A_d, float* B_d, float* C_d){

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}

    int main(){
    ...
    dim3 numThreads(256,1,1) //Los 1s son opcionales
    dim3 numBlocks(N,1,1)
    VecAdd<<<numBlocks, numThreads>>>(A_d, B_d, C_d);
    ...
}
```

ras, estas son pequeñas si se las compara con las cache de CPU. Esto se debe a que la gran mayoría de los transistores son utilizados para cómputo. Por lo anterior, las estrategias de utilizar las cachés para reducir el costo de acceso a memoria en general no funcionan a gran escala en estos dispositivos. En esta línea, son necesarias estrategias distintas a las utilizadas en las plataformas de hardware tradicionales para ocultar el costo del acceso a memoria a niveles razonables, en especial cuando se debe acceder a la memoria global. La principal estrategia que provee el hardware con este fin es suspender la ejecución del *warp* que hace acceso a memoria hasta que su dato esté disponible y pasar a ejecutar algún *warp* que estuviera en espera en la SM y esté listo para ejecutar. Para hacer uso de esta estrategia es necesario que existan virtualmente más hilos y bloques que los que están ejecutando en cada momento (es decir, mayor cantidad de hilos/bloques que capacidades de hardware como procesadores y SMs). Esta estrategia es viable porque los costos de cambio de contexto entre *warps* son muy bajos ya que la planificación está resuelta a nivel de hardware. Por lo tanto, la gran cantidad de procesadores con los que cuentan las GPUs modernas hace que estas plataformas tengan un gran potencial en la medida de que se logre ocultar la latencia y los procesadores estén realmente computando.

Además del soporte a nivel de hardware por el planificador existen estrategias a nivel de programación que también mejoran sustancialmente el costo de acceso a memoria. En concreto, es necesario que los accesos a memoria exploten todo el ancho de banda de las GPU, el cual es varias veces mayor al

ofrecido normalmente por las CPU. En dispositivos con Compute Capabilities[7] 5.0 en adelante (arquitecturas Maxwell y posteriores) para aprovechar el ancho de banda de acceso a memoria se debe combinar los accesos en bloques de 32 Bytes. Esto es posible, en parte, debido a la adición de la memoria caché L2, que reduce los requisitos de alineamiento en el acceso a memoria presentes en las primeras arquitecturas (pre-Fermi) además de mejorar los tiempos de acceso.

También es posible optimizar el acceso a memoria compartida. Esta memoria se particiona en 32 módulos llamados bancos. Estas particiones tienen un ancho de 4 u 8 bytes dependiendo del tipo de datos. Los accesos a distintos bancos de memoria pueden ser hechos en paralelo mientras que si dos hilos quieren acceder a direcciones distintas del mismo banco de memoria los accesos serán serializados. Este último comportamiento es llamado conflicto de bancos. En esta línea una estrategia de optimización del acceso a esta memoria consiste en que los *warps* accedan a distintos bancos en la misma iteración. A partir de las Compute Capabilities [46] 5.0 si dos hilos acceden al mismo banco pero lo hacen en la misma palabra de 4 Bytes para lectura no generan conflicto de bancos y se procederá a hacer un *broadcast* con el resultado. Por el contrario, la escritura sigue generando problemas ya que únicamente uno de los hilos escribirá y cuál de ellos lo hará es un comportamiento indefinido. En la Figura 2.6 se muestra un ejemplo con la memoria compartida dividida en bancos de 4 bytes.

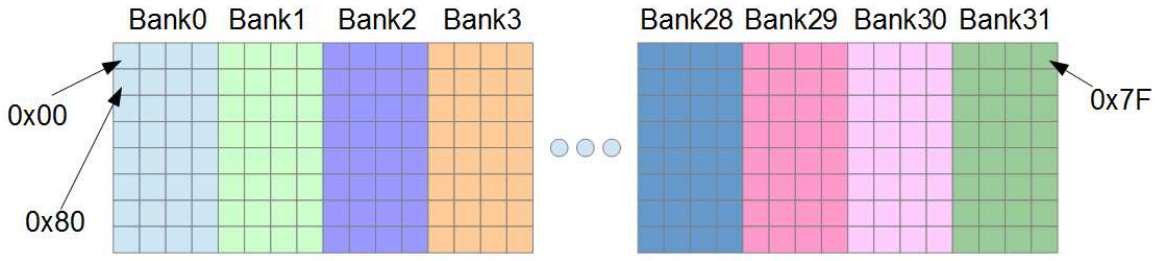


Figura 2.6: Ejemplo de la división de memoria compartida en bancos de tamaño 4 Bytes. Extraído de [3].

[7]La capacidad de cómputo o compute capabilites de un dispositivo está representada por un número de versión, también llamado a veces su "versión SM". Este número de versión identifica las funciones compatibles con el hardware de la GPU y las aplicaciones lo utilizan en tiempo de ejecución para determinar qué funciones y/o instrucciones de hardware están disponibles en la GPU actual. La capacidad de cómputo comprende un número de revisión mayor X y un número de revisión menor Y y se denota por X.Y.

Previo al surgimiento de las funciones de *shuffle* que fue posteriormente complementado por *cooperative groups* esta memoria era la única forma que tenían los hilos para trabajar en conjunto. Finalmente, con la introducción de los *Thread Block Clusters* se generó un nivel intermedio (lógico) en la jerarquía de memoria que es la memoria compartida del *cluster*. Esto se implementa permitiendo que los bloques puedan escribir y leer de la memoria compartida de otro bloque en el mismo *cluster* incluso aunque estén ejecutando en dos SM distintas; esto último es especialmente relevante ya que NVIDIA no brinda una forma de saber a priori qué bloques compartirán SM.

Como se mencionó anteriormente, los hilos pueden leer los valores de registros de otros hilos del mismo grupo utilizando primitivas de *shuffle*. En la Figura 2.7 se presenta un esquema con la jerarquía de memoria de las GPUs de NVIDIA.

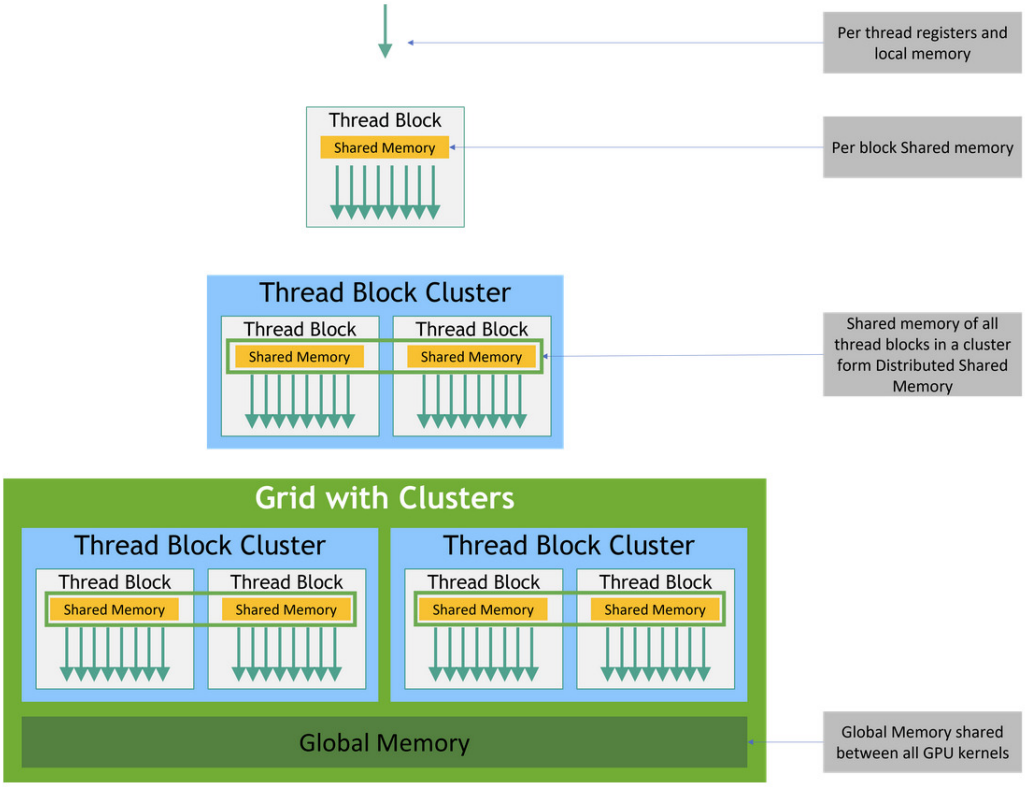


Figura 2.7: Esquema de la jerarquía de memoria en CUDA. Extraído de [4].

El impulso tomado por NVIDIA gracias a CUDA ha convertido a estas tarjetas gráficas en el estándar de facto en HPC, incluso cuando existen otros lenguajes y arquitecturas. En la actualidad, la principal área de aplicación de las GPU es el área de inteligencia artificial (IA) por su gran necesidad de

cómputo. Los métodos numéricos son otra de las áreas importantes de aplicación para estas tarjetas y, en el pasado, supieron tener un peso superlativo, aunque en el último tiempo hayan perdido terreno ante la IA.

2.2.2. Sincronización en GPU

En las arquitecturas iniciales de Nvidia (Compute Capabilities 6.x o anteriores) existían pocas alternativas para la sincronización entre hilos. En concreto, se brindaban dos herramientas. Por un lado existía la sincronización implícita a nivel de *warp* que estaba garantizada al contar con un único *program counter* (PC) por lo que era imposible despachar más de una instrucción simultánea. Cuando los hilos de un *warp* se dividían por *branches* lógicas (conocido en la jerga de CUDA como *thread divergences*) se requería serializar las dos alternativas y deshabilitar los hilos que no debían ejecutar en cada una utilizando una máscara. Es evidente que con esta solución se multiplica el tiempo requerido a las instrucciones por la cantidad de caminos distintos que siguen los hilos de un mismo *warp*. En este sentido los códigos en GPU priorizaban el no generar este tipo de bifurcaciones y, por lo tanto, no eran capaces de trabajar con granularidades más pequeñas.

La segunda alternativa que existía en las arquitecturas previas a Volta era la barrera `__syncthreads()`, que permite sincronizar a todos los hilos del bloque. Sin embargo, dada la relativamente gruesa granularidad de esta barrera, cuando eran necesarias sincronizaciones más finas debían ser implementadas por el programador utilizando operaciones atómicas.

Las arquitecturas modernas de Nvidia introdujeron un cambio importante al agregar PCs en todos los *CUDA cores*[8]. Este cambio generó que dejara de existir la sincronización implícita entre hilos de un mismo *warp*, por lo que cualquier tipo de sincronización debe hacerse de manera explícita. El objetivo de eliminar sincronizaciones implícitas y pasar a llamadas explícitas fue intentar que los códigos de CUDA sean más legibles, mejorar la compatibilidad hacia delante, reducir la fragilidad de los programas y las restricciones al compilador [47]. Estas modificaciones generaron, sin embargo, que muchos códigos desarrollados en versiones anteriores y que dependían de las sincronizaciones implícitas debieran ser modificados para evitar condiciones de carrera. Final-

[8]Un CUDA core es el análogo a un core (de CPU) en arquitecturas de GPUs de Nvidia, con la diferencia que fueron diseñados para realizar múltiples cálculos al mismo tiempo.

mente, para obtener el mismo comportamiento previo y sincronizar los hilos de un *warp* se brinda `__syncwarp()`. Esta funciona como barrera para todos los hilos del *warp* y, opcionalmente, se le puede brindar como parámetro una máscara para elegir cuáles hilos participarán de la sincronización.

Con este cambio en la arquitectura, NVIDIA presentó, en CUDA 9, la herramienta de los *cooperative groups*. Esta herramienta permitió flexibilizar las sincronizaciones entre hilos de un bloque y, en concreto, posibilitó la sincronización de manera nativa de un subconjunto de hilos aunque no compartan *warp*. En este contexto se puede generar un grupo de hilos de diversas maneras: particiendo un bloque equitativamente (`thread_block_tile`), entre los hilos que se encuentran *coalesced* en un momento dado (`coalesced_group`) o grupos implícitos (`thread_block` o `grid_group`). Los grupos, una vez definidos, pueden ser sincronizados utilizando la primitiva `sync()` que es una función pública de cualquiera de las clases mencionadas anteriormente. También cuentan con operaciones de *shuffle* (`shfl_up`, `shfl_down`, etc.) o de votación (`all`, `any`) entre otras. Los *cooperative groups* modificaron sustancialmente cómo se puede afrontar la sincronización en las GPUs de NVIDIA. Al serializar los subconjuntos del *warp* que divergen, es posible provocar un *deadlock* si el subgrupo que ejecuta después depende del que lo hace antes. Por el contrario, al usar *cooperative groups* esto se evita, ya que garantiza que todos los grupos ejecutarán en algún momento.

En CUDA 11 se agregó una nueva manera de agrupar hilos conocida como *clusters*. Este nuevo tipo de grupo puede estar compuesto por hilos de distintos bloques. Si bien en las iteraciones anteriores de *cooperative groups* contaban con algunos grupos que abarcaban más de un bloque (como `grid_group`) los *clusters* permiten mucho mayor flexibilidad al poder agrupar bloques de maneras relativamente arbitrarias. Al igual que otros grupos, los clusters pueden ser definidos en tiempo de compilación o de ejecución (utilizando `cudaLaunchKernelEx`). En el Algoritmo 3 se presenta un ejemplo con el mismo código que el Algoritmo 2 pero utilizando *clusters*. Notar que el cálculo del índice i se hace de la misma forma que antes, esto se debe a que incluso utilizando *clusters* los índices de bloque siguen siendo relativos a la grilla y no al cluster definido (esto permite compatibilidad hacia atrás). Una ventaja importante de utilizar clusters es que estos permiten a hilos de distintos bloques leer y escribir en la memoria compartida de los otros. Esto se debe a que CUDA despacha los bloques de un mismo *cluster* en un mismo *GPU Processing*

Cluster (GPC) [4].

Algoritmo 3 Declaración del mismo kernel del Algoritmo 2 utilizando cluster en tiempo de compilación

```
1  // Definicion del kernel
2  // Tamaño del clúster en tiempo de compilación 2 en la dimensión X y
       1 en las dimensiones Y y Z
3  __global__ void __cluster_dims__(2, 1, 1)
       VecAdd(float * A_d, float * B_d, float * C_d){
4          int i = blockIdx.x * blockDim.x + threadIdx.x;
5          C[i] = A[i] + B[i];
6  }
7
8  int main(){
9      float *input, *output;
10     // Ejemplo de invocación con definición de clusters en tiempo de
           compilación
11     dim3 threadsPerBlock(16, 16);
12     dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
13
14     // La dimensión de la grilla no se ve afectada por el
           lanzamiento del clúster y aún se enumera utilizando el número
           de bloques
15     // La dimensión de la grilla debe ser múltiplo de la del cluster
16         dim3 numThreads(256,1,1)
17         dim3 numBlocks(N,1,1) //N debe ser múltiplo de 2
18         VecAdd<<<numBlocks, numThreads>>>(A_d, B_d, C_d);
19 }
```

2.3. Operaciones de Álgebra Dispersa

En esta sección se presentan las principales operaciones del álgebra lineal dispersa. En el Apartado 2.3.1 se presenta la multiplicación matriz dispersa vector, probablemente la operación más popular del álgebra lineal dispersa. Luego, en el Apartado 2.3.2 se describe la resolución de sistemas triangulares dispersos. En el Apartado 2.3.3 se presenta la multiplicación entre dos matrices dispersas, una operación que ha tomado relevancia en los últimos años debido al auge del cómputo relacionado a los grafos y redes sociales. Finalmente, en el Apartado 2.3.4 se explica la factorización LU incompleta, operación relevante en el precondicionamiento de *solvers* de sistemas lineales.

2.3.1. Multiplicación Matriz Dispersa Vector (SpMV)

La multiplicación de una matriz dispersa con un vector ($SpMV$ por sus siglas en inglés -Sparse Matrix Vector Multiplication-) es, probablemente, la operación más utilizada del álgebra dispersa. Esto se debe a que es el cuello de botella de muchos métodos iterativos para la resolución de sistemas lineales o la obtención de valores y vectores propios para matrices de gran tamaño. Esta operación se define de la siguiente forma:

$$y = Ax \tag{2.2}$$

donde A es una matriz dispersa y los vectores x e y son los vectores de entrada y salida respectivamente, ambos densos.

La importancia del $SpMV$ ha generado que se dediquen grandes esfuerzos en la comunidad científica a optimizar la rutina. La optimización del $SpMV$ es un problema abierto. En esta línea, se han propuesto una gran cantidad de implementaciones y formatos de almacenamiento diseñados específicamente para acelerar $SpMV$. La eficiencia de las implementaciones (y formatos) de $SpMV$ es especialmente dependiente del patrón de dispersión de la matriz y de la plataforma de hardware subyacente.

A lo largo del tiempo, la comunidad ha dedicado un intenso trabajo al desarrollo de implementaciones eficientes en formatos generlistas como CSR y, en particular, en adaptarlas a las distintas plataformas de hardware. En esta línea, uno de los trabajos pioneros en GPU es el de Bell y Garland [38, 48] que marcó un punto de inflexión en el desarrollo de la $SpMV$ en esta plataforma. Allí, los autores propusieron una serie de implementaciones para los principales formatos (COO, CSR, DIA, ELL, HYB, etc.) e hicieron un estudio en profundidad de su rendimiento, en el que hallaron que la implementación en el formato HYB supera en una gran cantidad de casos a las restantes opciones. También hallaron que la implementación en CSR denominada CSR-*vector*) era competitiva. Esta implementación asigna un *warp* para procesar cada una de las filas de manera independiente.

La diversidad de implementaciones y el hecho de que ninguna sea mejor en todos los casos hacen que la elección de qué rutina usar sea un problema en sí mismo. En [49] los autores analizaron implementaciones de $SpMV$ en CSR y se propusieron elegir el mejor método en cada caso. En estudios preliminares hallaron que, aunque la implementación *vector* comentada anteriormente es

la que tiene mejores resultados en más casos, otras implementaciones como *merge* [50] la superaba en el ahorro de tiempo total. Esta última soluciona el problema los malos resultados de *vector* en los escenarios en los que la matriz tiene muchas filas con pocos no-ceros en cada una. Dado que la versión *vector* asigna un número fijo de hilos (originalmente un *warp*) en estos casos se tienen muchos hilos ociosos. Por el contrario, *merge* intenta equilibrar la carga de trabajo de los hilos con el objetivo que el desempeño de $SpMV$ escale con nnz.

2.3.2. Resolución de Sistemas Triangulares Dispersos ($SpTRSV$)

Otra operación común en el álgebra lineal dispersa es la resolución de sistemas triangulares dispersos ($SpTRSV$ por su sigla en inglés). Esta operación consiste en resolver un sistema de ecuaciones lineales triangular y disperso, es decir un sistema $Ax = b$ donde A es una matriz dispersa triangular (superior o inferior) y b es un vector denso. A efectos de este trabajo la operación se expresa de la siguiente manera:

$$Lx = b \tag{2.3}$$

donde L es una matriz triangular inferior, x el vector de incógnitas y b el de términos independientes. Podría hacerse una formulación análoga con una matriz U triangular superior.

Esta operación es uno de los desafíos fundamentales del álgebra lineal dispersa y cuenta con varias aplicaciones muy relevantes. Dos de los ejemplos más comunes son la etapa final de los *solvers* directos de sistemas lineales dispersos (donde se resuelven los dos sistemas surgidos de la descomposición LU) y la aplicación de precondicionadores de tipo ILU en cada paso de métodos iterativos para resolver sistemas lineales como el *Preconditioned Conjugate Gradients method* (PCG) o el *Generalized minimal residual method* (GMRES)

En el Algoritmo 4 se presenta el pseudocódigo de la resolución de sistemas triangulares dispersos con la matriz en formato CSR. La forma más natural de implementar la $SpTRSV$ consiste en la sustitución hacia delante. Las entradas de este algoritmo son la matriz (como sus tres vectores del formato CSR), el vector independiente (b) y la cantidad de filas. El algoritmo itera por las filas

de arriba hacia abajo, eso corresponde a la línea 7. Luego, en cada fila (líneas 8-10) se sustituyen las incógnitas correspondientes a las filas anteriores y las acumula en `left_sum`. Por último, en la línea 11, se calcula la incógnita y se guarda en el vector de salida.

Algoritmo 4 *SpTRSV* (Secuencial) para $Lx = b$ en formato CSR

```
1  // In: n > 0, row_ptr[], val[], col_idx[], b[]
2  // Out: x[]
3
4  x[0] = b[0] / val[0]
5
6  for i = 1  to  n-1 :
7    left_sum = 0;  // Inicializacion left_sum = 0
8    for j = row_ptr[i]  to  row_ptr[i+1] -1
9      left_sum = left_sum + val[j] * x[col_idx[j]]
10   endfor
11   x[i] = ( b[i] - left_sum ) / val[i]
12 endfor
```

La paralelización de la *SpTRSV* no es trivial debido a la cantidad de dependencias entre las filas, que implica la necesidad de sincronizaciones que no son necesarias en otras operaciones como la *SpMV*, complejizando su implementación en dispositivos masivamente paralelos como la GPU. Estas dependencias surgen del patrón de dispersión de la matriz dispersa, donde si una fila j tiene un no-cero en la columna i la fila j depende de i, debido a que la ecuación j del sistema involucra a la variable x_i. En esta línea, es muy importante la distribución de los datos entre las distintas unidades de cómputo a efectos de explotar el paralelismo, decisión que debe tomarse para cada matriz.

La forma más intuitiva de paralelizar el *SpTRSV* consiste en procesar las filas en paralelo y, dentro de éstas, paralelizar la sustitución. La tarea de cada hilo es, entonces, obtener la incógnita correspondiente, multiplicarla por el coeficiente y sumarla a un acumulador local. Luego, dependiendo de la cantidad de hilos asignados a la fila y su número de coeficientes, podrá (o no) repetir el proceso para otro coeficiente. Posteriormente, los valores de todos los hilos serán reducidos y se grabará en memoria el resultado. El Algoritmo 5 presenta el enfoque descrito.

Algoritmo 5 *SpTRSV* (paralelo) para $Lx = b$ en formato CSR

```
//In:  n > 0, row_ptr[], val[], col_idx[], b[]
//Out: x[]

for i = 0  to  n - 1 in parallel do
    writelock(x[i])
endfor
for i = 0  to  n - 1 in parallel do
    sync()         //Sincroniza todos los hilos
    left_sum = 0   //Una variable left_sum por hilo
    for j = row_ptr[i]  to  row_ptr[i+1]-2 in parallel do
        readlock(x[j]) //Espera si x[j] se encuentra bloqueado por
            escritura
        local_sum = x[j]*val[j]
        readunlock(x[j])
    endfor
    left_sum = reduce(local_sum) //Requiere que todos los j hayan
        terminado
    x[i] = (b[i] - left_sum) / val[i]
    writeunlock(x[i])
endfor
```

2.3.3. Multiplicación de Matriz Dispersa por Matriz Dispersa (SpGEMM)

La multiplicación general de matrices dispersas (SpGEMM) es otra rutina de creciente importancia en el álgebra dispersa. Su definición es la siguiente:

$$C = \alpha A \times B + \beta C \tag{2.4}$$

donde A, B, C y D son matrices dispersas y las tres primeras son los parámetros de entrada. Esta operación es importante en varias aplicaciones del álgebra lineal, como algoritmos de búsqueda de camino más corto [51] o los algoritmos de coloreado de grafos [52, 53].

Esta operación tiene varias restricciones si se la compara con su equivalente denso. En primer lugar, tiene una muy baja densidad computacional, es decir, la cantidad de bytes de datos que se requiere leer de memoria por operación de punto flotante. Sin embargo, el principal desafío de esta operación, a diferencia del resto de las operaciones presentadas en esta sección, es que esta no solo depende del patrón de dispersión de una matriz, sino de cómo se relacionan los

patrones de dispersión de las dos matrices que se multiplican. Esto último hace que sea especialmente difícil determinar cuántos coeficientes no-cero tendrá la matriz resultado y su patrón de dispersión aún sabiendo los de las matrices de entrada.

Además del cálculo en sí mismo, SpGEMM tiene tres etapas importantes a considerar. Primero, como se dijo anteriormente, se debe saber cuánta memoria será necesario reservar para la matriz resultado. Para esta etapa hay tres grandes estrategias: hacer el cálculo simbólico para saber el resultado exacto de la matriz resultado [54], estrategias probabilísticas basadas en métodos montecarlo y reservar memoria considerando la cota superior [55]. Adicionalmente, se puede hacer un enfoque "bajo demanda" en el que se va pidiendo más memoria cuando es necesaria [56]. Las ventajas y desventajas dependen mucho de la aplicación. Mientras el método exacto implica un gran sobrecosto de procesamiento (que podría ser reutilizado), el método de la cota implica en general un sobrecosto de memoria. Finalmente, el enfoque probabilístico será un punto medio entre ambos dependiendo de qué tan buena sea la estimación.

Otra parte importante de un algoritmo de SpGEMM en GPUs es la estrategia de equilibrio de carga entre los distintos hilos y *warps*. Al igual que en otras operaciones de álgebra dispersa, la irregularidad del cómputo implica que hacer una asignación inteligente y pareja de trabajo es muy importante para la eficiencia de la rutina. Finalmente, dado que SpGEMM genera muchos resultados intermedios que deben ser acumulados, existen distintas estrategias para guardar los resultados parciales. Las estrategias de acumulación se dividen en tres grandes grupos: acumulación densa [57], basada en tablas de hash [58] y basada en listas u ordenamientos [59].

2.3.4. Factorización LU incompleta

La última operación que se presenta en esta sección es la factorización LU incompleta (ILU por sus siglas en inglés -Incomplete LU factorization-). Esta operación consiste en hallar, para una matriz de entrada A, dos matrices L y U (ambas dispersas) tales que:

$$LU \approx A \tag{2.5}$$

donde L es una matriz triangular inferior cuya diagonal está compuesta por unos y U es una matriz triangular superior.

Cuando se trabaja con matrices dispersas, la factorización LU estándar puede volverse impracticable ya que esta factorización no garantiza que las matrices resultantes sean dispersas. Este fenómeno, de que aparezcan coeficientes no-ceros en las posiciones que tenían cero en la matriz original, se conoce como problema de *zero fill-in*. En matrices de dimensiones muy grandes, esto puede generar que el consumo de memoria aumente a niveles insostenibles. En esta línea, la ILU da una factorización aproximada pero que garantiza que las matrices resultantes sean dispersas [60].

Dependiendo del grado de dispersión que se le exige a las matrices resultado se genera un abanico de $ILU - x$ donde x es el nivel de *fill-in* permitido. A mayor *fill-in* más parecido es el resultado de multiplicar $L \times U$ a la matriz original, pero a cambio las matrices L y U ocupan más memoria al ser menos dispersas. A efectos de este trabajo, siempre que se haga referencia a ILU sin aclaración del *fill-in* se asumirá ILU-0, es decir que el patrón de dispersión de la matriz A es el mismo que el de $L + U$.

Las aplicaciones de la ILU también están ligadas a los métodos iterativos para resolver sistemas de ecuaciones lineales. En particular, métodos como el gradiente conjugado (CG) o el método de residuo mínimo generalizado (GMRES) [61] utilizan la ILU para acelerar su convergencia. Cuando la matriz A está mal condicionada (cuando el ratio entre el valor propio de mayor y menor magnitud es grande) estos métodos pueden ser lentos para converger, haciendo necesario el precondicionamiento del sistema. Un precondicionador es una matriz M que multiplica al sistema convirtiéndolo en uno equivalente pero con menor número de condición. Por ejemplo, un sistema precondicionado por izquierda es de la siguiente forma:

$$M^{-1}Ax = M^{-1}b. \tag{2.6}$$

La ILU es uno de los precondicionadores más utilizados en el álgebra numérica, en especial cuando no se tiene información adicional del problema [60]. En el Algoritmo 6 se presenta el pseudocódigo serial de la factorización ILU *in-place*, es decir que la matriz A de entrada es sustituida por L en el triángulo estrictamente inferior y U en el triángulo superior. Se procesan las filas en orden. Para cada fila *current* (i) se procesan los no-ceros de izquierda a derecha obteniendo la fila *reference* (k) que es la que se utiliza para hacer 0 el elemento A_{ik}. En la línea 8 se calcula el pivot haciendo la división entre el valor A_{ik}

y el valor de la diagonal de *reference* y, al ser *in-place*, se lo guarda en A_{ik}. Finalmente, en las líneas 11-13, se actualizan todos los valores de la fila *current* que tengan un correspondiente en *reference*.

Algoritmo 6 Variante de ILU-0 serial in-place

```
1  //In:  A
2  //Out: L, U
3  //L es el triángulo estrictamente inferior y U es el triángulo
       superior de A al finalizar
4
5  for i = 2  to  n do
6      for k= 1  to  i-1 do
7          if (A_ik ≠ 0)
8              A_ik = A_ik / A_kk //Calcula el pivot y se guarda en A_ik
9          endif
10         for j= k+1  to  n do
11             if(A_ij ≠ 0)
12                 A_ij =A_ij−A_ik×A_kj //Impacta la fila superior en los
                       coeficientes no-cero de la fila inferior
13             endif
14         endfor
15     endfor
16 endfor
```

Las factorizaciones ILU pueden ser muy costosas en cuanto a tiempo de cómputo, especialmente para matrices de gran tamaño. Esto se debe, en parte, a la cantidad de dependencias entre las filas que surge del carácter secuencial de la eliminación gaussiana.

Esta operación tiene dos tipos de dependencia, si se mira a nivel de fila las dependencias son idénticas al $SpTRSV$. Si la fila i tiene un no-cero en la columna j entonces i depende de esta última. Esto es porque para calcular el pivote es necesario tener el valor final del elemento diagonal de j. Adicionalmente, si se mira a nivel de no-ceros en la fila, el coeficiente de la columna a depende de uno anterior (es decir, una columna b con $b < a$), si en la fila *ref* hay no-ceros tanto en a como b. Esto se debe a que luego de calcular el pivote de b se debe impactar la fila *ref* y esto debe hacerse antes de procesar el elemento de la columna a.

Como la factorización ILU es una operación muy relevante en la computación científica, se han dedicado varios esfuerzos por parte de la comunidad para obtener implementaciones eficientes en tarjetas gráficas. Estos aborda-

jes incluyen enfoques basados en paradigmas *level-set* [62, 63], estrategias de coloración de grafos [64] y métodos iterativos [65, 66].

2.4. Paradigma Self Scheduled

La aplicación de las estrategias *self scheduled* o *synchronization free* (o simplemente *sync-free*) viene de larga data. En general, la estrategia *self scheduled* consiste en dividir la ejecución en una lista de sub-tareas que pueda ejecutarse de manera paralela por un número arbitrario de procesadores con un cierto nivel de dependencias. Esta estrategia fue planteada, entre otros [9] en el contexto de multiprocesadores para la resolución de sistemas de ecuaciones dispersos y factorización de Cholesky [67, 68]. Una idea similar es utilizada por Park et al. [69] para acelerar el $SpTRSV$.

La idea general consiste en despachar un conjunto de sub-tareas a las que se identifica con un valor identificador. La única propiedad que debe cumplir este conjunto es que, si una de estas tareas tiene un identificador i y otra j con $i < j$, la primera tarea no depende de la segunda. Dicho de otra manera, el orden de los identificadores genera un orden bien definido de las dependencias. En la Figura 2.8 se presenta una matriz dispersa de ejemplo y en la Figura 2.9 cómo serían las dependencias generadas por dicha matriz.

El Algoritmo 7 presenta una especificación genérica del paradigma *self scheduled* sobre un conjunto de tareas que luego serán tomadas por los procesadores, de manera ordenada, a medida que estén libres. A diferencia de otras estrategias, la asignación de tareas a procesadores no se encuentra definida a priori, sino que se va descubriendo en la ejecución del programa. La gran ventaja de este enfoque es su flexibilidad, ya que funciona con tareas no necesariamente uniformes, diversos números de procesadores e incluso cuando estos tienen capacidades muy distintas. Por el contrario, el enfoque dinámico introduce un *overhead* producto de la comunicación y/o sincronización entre los distintos procesadores, pero se espera que sea despreciable en proporción al tiempo de cómputo. Por este último motivo, en los primeros trabajos se hablaba de que esta estrategia es más adecuada para paralelismo *large grained* o *medium grained*, es decir $t >> p$ donde t es la cantidad de tareas y p de procesadores.

[9] Este tipo de estrategias son muy comunes en contextos de Redes o Sistemas Operativos

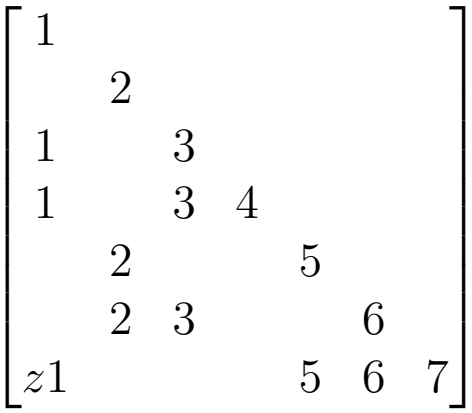


Figura 2.8: Matriz dispersa de ejemplo. Los valores no mostrados son ceros.

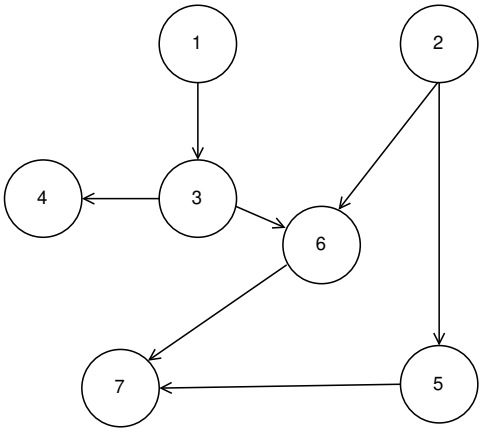


Figura 2.9: Dependencias para la estrategia *self-scheduled* correspondientes a la matriz de la Figura 2.8. Dependencias redundantes fueron removidas.

Algoritmo 7 Especificación genérica de la estrategia *self scheduled*.

```
1 for i = 1  to  n do
2     schedule(Task[i])
3 endfor
```

En [67] se aborda la resolución de sistemas generales dispersos $Ax = b$ donde A es dispersa y definida positiva. En todo el trabajo se asumen matrices almacenadas por columna (es decir, elementos contiguos en una columna se encuentran contiguos en memoria). Concretamente, la resolución de estos sistemas requiere aplicar reordenamientos a la matriz A, factorizar la matriz reordenada y resolver dos sistemas triangulares dispersos dependientes ($Lv = b$ y $L^T x = v$ donde L es triangular inferior). Para esto, se proponen sendas rutinas que resuelven los sistemas triangulares paralelamente haciendo sustitución hacia adelante y hacia atrás, respectivamente. Para estas resoluciones se utilizó un enfoque *self-scheduled*. Primero, en el Algoritmo 8 se presenta una versión ligeramente simplificada de la rutina que se propuso para la resolución del primer sistema. Posteriormente, en el Algoritmo 9 se hace lo mismo para la rutina de resolución del sistema $L^T x = v$. Es importante notar que esta rutina accede a la matriz L (almacenada por columnas) ya que L^T no se mantiene en memoria.

Este algoritmo define una tarea por incógnita y, antes de comenzar, pre-

Algoritmo 8 Resolución con sustitución hacia adelante del sistema $Lv = b$.

```
1  for j= 1  to  n do
2      nmod[j] = nnz(L_j*)
3  endfor
4  for j= 1  to  n do
5      schedule(Task(j))
6  endfor
7
8  Task(j){
9      wait until nmod[j] = 0 //bloqueo tipo semáforo
10     vdiv(j); //division del valor independiente b_j por el valor de la
           diagonal L_jj
11     for each nocero_no_diagonal L_kj en la columna j do
12         vmod(k, j) //modifica v_k restando b_j × L_jk
13         nmod[k] := nmod[k]- 1 //resta 1 a la cantidad de dependencias
               faltantes de la incognita k
14     endfor
15 }
```

calcula para cada fila su cantidad de no-ceros ($nnz(L_{j^*})$). Esto se hace a efectos de evitar acceder a los elementos por fila, algo que sería necesario para calcular la incógnita correspondiente a dicha fila. En este enfoque, la tarea $Task(j)$ solamente hace la división final del elemento de v_j por el coeficiente de la diagonal y asume que las tareas previas ya impactaron el valor de sus respectivas incógnitas en el término independiente. Posteriormente, en las líneas 11-14, se recorren todos los elementos de la columna j y se modifican los v_k correspondientes. Finalmente, se le resta 1 al valor de $nmod[k]$ simbolizando que una dependencia más ha finalizado y ha sido impactada.

La rutina especificada en el Algoritmo 9 mantiene la misma estrategia asignando una tarea por incógnita. Sin embargo, para procesar la incógnita debe acceder a una columna (fila del sistema correspondiente a L^T) en lugar de una fila. Esto facilita sustancialmente el procesamiento ya que no se requieren pre-cálculos, notar que los elementos de la columna se encuentran contiguos en memoria y pueden ser procesados serialmente.

En la resolución descrita anteriormente, se recorre la columna chequeando si las dependencias han sido cumplidas y, una vez que esto es así, se impacta en el valor del elemento del vector independiente (líneas 9 a 12). Posteriormente, una vez que se han impactado las dependencias, se divide el término independiente por el valor diagonal (línea 13) y se marca la incógnita como calculada y lista

Algoritmo 9 Resolución con sustitución hacia atrás del sistema $L^T x = v$.

```
for j= 1  to  n do
    ready[j] = 0
endfor
for j= n down  to  1 do
    schedule(Task(j))
endfor

Task(j){
    for each nocero_no_diagonal L_kj en la columna j do
        wait until ready[k] = 1 //bloqueo tipo semáforo
        wmod(j, k) //modifica v_j restando x_k × L_kj
    endfor
    wdiv(j) //division del valor independiente v_j por el valor de la
        diagonal L_jj
    ready[j] = 1 //Explicita que la incógnita j está lista para
        usarse
}
```

para usarse (14).

2.4.1. Paradigma *level-set*

Previo a la adopción del paradigma *self-scheduled*, el estado del arte de resolución de sistemas triangulares dispersos en GPU estaba dado por la implementación conocida como *level-set* [15].

Esta estrategia consiste en organizar las ecuaciones (filas en la matriz dispersa) en una secuencia de conjuntos (o niveles). Las ecuaciones en el primer nivel no dependen de ninguna otra y pueden ser resueltas inmediatamente. Una vez que el primer nivel es procesado, todas las filas del segundo pueden ser procesadas en paralelo y así sucesivamente.

Los niveles son construidos en una primera etapa de cálculo simbólico conocida como etapa de análisis. La implementación original de esta etapa hace una recorrida similar a BFS por el grafo correspondiente a interpretar la matriz como una matriz de adyacencia [16]. A diferencia de BFS, cada nodo es incorporado al nivel siguiente la última vez que se lo visita (es decir, cuando las dependencias están completas) en lugar de la primera vez. Explicaciones en profundidad de esta implementación y otras posteriores se presentan en el Apartado 3.3. En la Figura 2.10 se muestra la estructura de niveles resultante

de aplicar este paradigma a la matriz de la Figura 2.8.

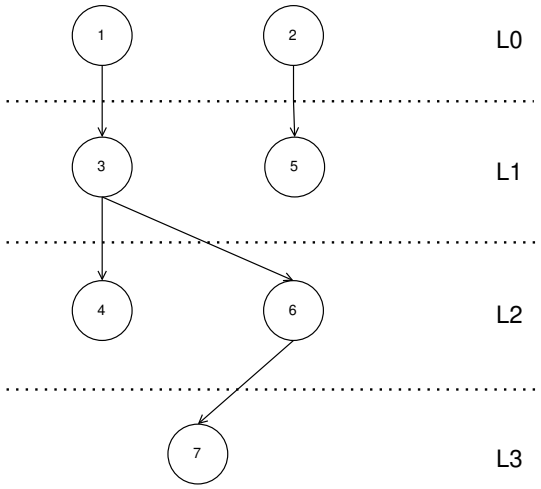


Figura 2.10: Estructura de niveles generada a partir de la matriz de ejemplo utilizando la estrategia level-set.

La etapa de resolución es directa. Primero, se procesan todas las filas del primer nivel (que no dependen de ninguna otra) en paralelo sustituyendo sus incógnitas por los valores respectivos. Una vez que todas estas han terminado, se repite el proceso con el siguiente nivel, con las dependencias del nivel anterior eliminadas.

Dado que las GPU de NVIDIA no solían dar soporte a formas de sincronización global durante la ejecución de un *kernel*, la estrategia clásica para implementarlas consiste en despachar un *kernel* por nivel.

Estas implementaciones tienen dos grandes problemas. En primer lugar, la estrategia de utilizar un *kernel* (relativamente liviano en cuanto tiempo de ejecución) por nivel provoca que el costo de lanzamiento de los *kernels* sea significativo en el total de la rutina. Por otro lado, se genera una barrera global que bloquea la ejecución de todas las filas de un nivel hasta que el previo ha sido completado. Esto provoca que filas que dependen solo de una o pocas filas del nivel anterior deban esperar a que todo el nivel finalice, limitando el paralelismo. Por ejemplo, en la Figura 2.10, la fila 4 tiene que esperar por las filas 2 y 5 porque están en el nivel anterior aunque no depende de ellas.

2.4.2. Self-scheduled en GPU (o Sync-Free)

El paradigma *self-scheduled* fue adaptado a GPU como *sync-free* por Liu et al. [25] en 2016 para acelerar la $SpTRSV$. El uso de esta estrategia superó

a la propuesta *level-set*. Los autores se enfocaron en el algoritmo de resolución hacia adelante propuesto por George y propusieron cambios algorítmicos que permitieran adaptarlo a esta nueva arquitectura. Un gran problema para adaptar este paradigma a la GPU es la sincronización para la que, en las arquitecturas en las que se trabajó esta propuesta, no se contaba con demasiadas posibilidades. En concreto, las alternativas para comunicar entre hilos de distintos bloques requerían utilizar la memoria global y sumas atómicas. Debido al alto costo de las operaciones atómicas en GPU, esto puede convertirse en el cuello de botella de una implementación en esta plataforma.

Como se dijo anteriormente, el enfoque *self scheduled* es eficiente cuando la cantidad de tareas es mucho mayor que la de procesadores y el tamaño de estas últimas hace que el tiempo requerido para procesarlas sea lo suficientemente grande para que los costos relacionados con la comunicación y/o sincronización sean despreciables. Sin embargo, esto no es posible en GPU ya que para explotar el paralelismo es necesario generar una gran cantidad de tareas muy pequeñas. En este sentido, la cantidad de procesadores (por ejemplo, la NVIDIA RTX 3090 TI tiene 10752 cores) hace que no se cumpla que $t >> p$ y las tareas sean extremadamente livianas generando que el sobrecoste de sincronización y comunicación tenga un peso relativo mayor.

La principal observación que posibilitó la adaptación a GPU de la estrategia de *self-scheduling* para el $SpTRSV$ consiste en cómo está hecho el acceso a memoria de las tarjetas gráficas. En concreto, el hecho de que cuando un *warp* accede a memoria las GPUs NVIDIA lo remueven de la ejecución y lo mantienen inactivo hasta que los datos estén listos permite reducir tanto el costo de comunicación (cuando cada *warp* debe obtener de un lugar centralizado qué fila procesar) como el de sincronización (el bloqueo hasta que las dependencias estén listas) que se puede sustituir por una espera activa. Si bien puede parecer que esto introduce un gran sobrecosto de cómputo, dado que en cada iteración de la espera se accede a memoria, hasta que la dependencia se finaliza el *warp* está la gran mayoría del tiempo bloqueado esperando por los datos. Una versión simplificada de la rutina propuesta puede verse en el Algoritmo 10.

Puede verse que el Algoritmo 10 tiene varias similitudes con el Algoritmo 8. La principal diferencia es que, al contar con una mucha mayor cantidad de procesadores, cada *warp* procesa una incógnita y cada hilo procesa uno u unos pocos no-ceros. Concretamente, cada *warp* es encargado de, una vez que

Algoritmo 10 Algoritmo *sync-free* para la resolución de sistemas triangulares en GPU con matrices almacenadas en CSC. Simplificado de [25]

```
1  for j= 1  to  n do
2      d_in_degree_row[i] = n(L_j*)
3      d_left_sum[i] = 0
4  endfor
5  for j= 1  to  n in parallel do  //Un warp por incógnita
6      while(d_in_degree[i] != 0)
7          //busy wait
8      endwhile
9      x[i] = (b[i] -d left sum[i])/val[col_ptr[i]]
10     for j = col_ptr[i] + 1  to  col_ptr[i+1] - 1 in parallel do //un
           hilo por no-cero hasta llenar el warp
11         rid = row_idx[j]
12         ATOMIC_ADD(d_left_sum[rid], val[j] × x[j])
13         ATOMIC_SUB(d_in_degree[rid], 1)
14     endfor
15 endfor
```

se ha cumplido la etapa de espera activa y el valor de la incógnita es despejado, impactar el valor de su incógnita en todas las ecuaciones que dependen de este. Esto genera que distintos hilos (de distintos *warps*) deban acumular para las distintas incógnitas por lo que se utiliza una variable global por ecuación (`d_left_sum`, accedida de manera atómica) para almacenar resultados intermedios. Una vez terminadas todas las dependencias este será el valor que se restará a b.

El segundo cambio importante sobre la rutina propuesta por George et al. es el *loop* de las líneas 6-8. Como se explicó anteriormente, esta parte consiste en una espera activa que termina en que el hilo está constantemente bloqueado por acceder a memoria al consultar por la variable `d_in_degree`, que guarda la cantidad de dependencias que faltan por resolverse para cada ecuación.

La principal diferencia entre el Algoritmo 10 y el propuesto en [25] consiste en que, a efectos de facilitar su comprensión, se han eliminado las optimizaciones hechas en la propuesta original consistentes en utilizar la memoria compartida para comunicarse entre *warps* del mismo bloque.

Otra adaptación de este paradigma fue propuesta por Dufrechou y Ezzatti en [70]. Los autores se enfocaron en adaptar esta estrategia para el popular formato CSR. Una de las principales fortalezas de la implementación propuesta es la eliminación de las operaciones atómicas que, según evaluaciones

experimentales, son el mayor cuello de botella del algoritmo propuesto por Liu et al. cuando el tamaño de las matrices es medio o grande. En este sentido, la implementación propuesta por Dufrechou y Ezzatti no emplea operaciones atómicas. En el Algoritmo 11 se presenta una versión simplificada de la rutina. Al igual que en el Algoritmo 10 por claridad no se presentan optimizaciones muy específicas de la plataforma como el uso de memoria compartida para la comunicación entre *warps* del mismo bloque.

El Algoritmo 11 tiene dos grandes etapas. Primero, en el *loop* principal (líneas 13-23) se hace espera activa por las dependencias (vector `is_solved`) y una vez que estas filas están terminadas, se acumula el resultado de multiplicar el coeficiente por la incógnita. Luego, avanza 32 lugares hasta el siguiente no-cero que aún no ha sido procesado. Posteriormente, una vez que todos los no-ceros han sido procesados, se reduce el valor, generando que, al terminar, en el primer hilo del *warp* se tenga el resultado de $b[fila] - acumulado[fila]$. Finalmente, este hilo actualiza el valor de la incógnita y la marca como resuelta.

Esta implementación superó notablemente a la implementación *sync-free* en CSC en casi todas las matrices del conjunto de evaluación. En particular, los resultados son especialmente buenos en punto flotante de doble precisión. Gran parte de la ganancia se explica en la eliminación de las operaciones atómicas tanto en la etapa de preprocesamiento como en la propia etapa de resolución. Por último, llevar este paradigma a CSR es un aporte en sí mismo ya que permite expandir su utilidad. Posteriormente, otros autores experimentaron con la implementación de este paradigma en GPU tanto en CSR como CSC y las compararon con sendas implementaciones del paradigma *level-set* [71].

Algoritmo 11 Algoritmo *sync-free* para la resolución de sistemas triangulares en GPU con matrices almacenadas en CSR. Simplificado de [70].

```
for i = 1  to  n do
    is_solved[i] = 0
endfor
for wrp = 1  to  n in parallel do  //Un warp por fila
    lne = threadIdx.x % 32
    elem = row_ptr[wrp] + lne
    if lne == 0
        left_sum = b[wrp]
    else
        left_sum = 0
    endif
    //Espera activa por las dependencias
    while(elem < row_ptr[wrp+1])
        colidx = col_idx[elem]
        my_val = val[elem]
        ready = is_solved[colidx]
        //Cuando la dependencia se finaliza el hilo la acumula
            localmente
        if ready
            xx = x[colidx]
            left_sum -= xx × my_val
            elem += 32
        endif
    endwhile
    //Etapa de reducción del warp
    for i = [16, 8 ... 2, 1] do
        left_sum += __shfl_down(left_sum, i);
    endfor
    //El primer hilo del warp escribe a memoria
    if lne == 0
        x[wrp] = left_sum × piv //Se escribe el resultado
        is_solved[wrp] = 1 //Se marca la fila como resuelta
    endif
endfor
```

Capítulo 3

Trabajo relacionado

En el último lustro, los grandes avances en el desarrollo de *kernels* para resolver sistemas triangulares dispersos sobre plataformas de hardware masivamente paralelas han sido en torno al paradigma *sync-free*. En este marco, el presente capítulo compendia el estado del arte de los métodos *sync-free* aplicados al ALN dispersa.

En la Sección 3.1 se presenta una descripción general de los trabajos que aplican el paradigma *sync-free* al ALN dispersa. En la Sección 3.2 se explican los trabajos relacionados a la etapa de resolución del $SpTRSV$ bajo dicho paradigma. Posteriormente, en la Sección 3.3 se explican los aportes relativos a la etapa de análisis (o preprocesamiento del kernel $SpTRSV$). Además, la aplicación del paradigma *sync-free* no se limita al kernel $SpTRSV$, en esta línea la Sección 3.4 compendia los aportes de esta técnica en otras aplicaciones del ALN dispersa. Finalmente, en la Sección 3.5 se presentan las conclusiones del capítulo.

3.1. Uso de técnicas *sync-free* en ALN dispera

Los aportes hechos sobre el uso de técnicas *sync-free* para optimizar kernels de ALN dispersa en dispositivos masivamente paralelos pueden separarse en tres grandes categorías que se presentan a continuación.

El primer grupo consiste en trabajos que buscan mejorar la resolución de sistemas triangulares aplicando estrategias *sync-free* en la etapa de resolución. Estos trabajos se presentan en la Sección 3.2. En esta categoría se ubican trabajos como [72], donde se adapta el *solver* para manejar múltiples vecto-

res del lado derecho (operación conocida como SpTRSM). Otro enfoque, que proviene del álgebra densa, consiste en descomponer la resolución de sistemas triangulares en multiplicaciones de matriz-vector y sistemas triangulares más pequeños [5, 6]. Esto se debe a que es significativamente más fácil explotar el paralelismo en la multiplicación matriz-vector que en el *solver* triangular y por esto se cuenta con implementaciones paralelas muy eficientes. Finalmente, algunos trabajos se enfocan en mejorar el aprovechamiento de poder de cómputo mejorando la estrategia de asignación de tareas a los *warps* [7–9]. En las implementaciones originales del método *sync-free*, cada *warp* procesa una incógnita por lo que, cuando muchas filas (o columnas en formato CSC) tienen menos de 32 elementos (la cantidad de hilos de un *warp* de NVIDIA) algunos procesadores quedan ociosos limitando fuertemente el desempeño.

El segundo conjunto de trabajos está compuesto por esfuerzos que buscan aplicar el paradigma *sync-free* a la etapa de preprocesamiento (o análisis) de la operación *SpTRSV*. En la Sección 3.3 se presentan estos trabajos. En el paradigma *level-set* se requiere una primera etapa de análisis que busca descubrir el paralelismo posible en forma de niveles. Posteriormente, en una segunda etapa, todas las filas de un mismo nivel pueden ser procesadas en paralelo. En esta línea, en [73] los autores propusieron utilizar el paradigma *sync-free* para calcular la estructura de niveles en lugar del enfoque que se seguía en [16]. En [9] los autores buscaron utilizar la información obtenida en la etapa de análisis *level-set* para acelerar la resolución *sync-free*, generando un mejor orden de ejecución para las filas. En este trabajo, además de la información de la estructura *level-set*, se implementa otra estrategia de análisis que tiene en cuenta los tamaños de las filas y permite resolver más de una fila en paralelo con en el mismo *warp*. Posteriormente, en [26] los autores paralelizaron dicha etapa de análisis.

Finalmente, el tercer grupo de trabajos consiste en aplicar el paradigma a otras rutinas que no son la resolución de sistemas triangulares, esto se presenta en la Sección 3.4. En esta línea existen trabajos aplicando el paradigma *sync-free* a la factorización LU [74], al kernel del *Strongly Implicit Procedure* (SIP) [75, 76] o métodos iterativos como el Método Generalizado de Residuo Mínimo (GMRES por sus siglas en inglés -Generalized Minimal Residual Method-) [77], o el gradiente bi-conjugado (BiCG por sus siglas en inglés -Bi-Conjugate Gradient method-).

3.2. Etapa de resolución del kernel SpTRSV

Una primera extensión de [25] consistió en ampliar el trabajo original para resolver sistemas triangulares con múltiples vectores del lado derecho (operación conocida como SpTRSM) [72]. Los autores optaron por un enfoque adaptativo que, en función de la relación entre el tamaño de la columna a procesar y la cantidad de vectores del lado derecho ("ancho de la matriz"), decide dónde explotar el paralelismo. Si la cantidad de elementos en la columna es mayor que la cantidad de vectores independientes, se paralelizará la columna (similar a las líneas 10-14 del Algoritmo 10 pero repitiendo para todos los b). Por el contrario, si el "ancho" de la matriz resultado es mayor, se paralelizarán los distintos b (y las líneas 10-14 del Algoritmo 10 serán serializadas). Otra extensión de este algoritmo fue su adaptación a multi-GPU [78].

A diferencia del caso disperso, en la resolución de sistemas triangulares densos no es posible procesar varias filas en paralelo. Esto se debe a que cada fila depende de todas las anteriores al tener coeficientes distintos de cero en todas las entradas por debajo de la diagonal. En este sentido, la descomposición en multiplicaciones matriz-vector y sistemas triangulares más pequeños permite explotar el paralelismo de la primera operación. Esta estrategia se adaptó al caso disperso en distintos trabajos, que descomponen la operación *SpTRSV* en multiplicaciones de matriz dispersa-vector (SpMV) e instancias de *SpTRSV* para realizar actualizaciones más pequeñas.

Un enfoque en este sentido es el propuesto por Lu et al. [5]. En este trabajo se proponen tres variantes que siguen este principio llamadas *column block*, *row block* y *recursive block* (recblock) siendo este último el que obtiene mejores resultados preliminares. La idea general de los algoritmos se muestra en la Figura 3.1. En el caso de recblock, el ejemplo muestra dos particiones recursivas. Finalmente, dependiendo de características de la matriz y de la cantidad de niveles de la rutina recblock, se define qué implementaciones de *SpTRSV* (*sync-free*, *level-set* o incluso CUSPARSE [79, 80]) y *SpMV* (`scalar` y `vector` [38, 48] implementados tanto en CSR como el formato desarrollado por los propios autores para mejorar redblock llamado DCSR).

Otra estrategia en esta línea es `TileSPTRSV` [6]. Esta implementación extiende la biblioteca `Tile` [81–83] que explota la estructura 2D de las matrices para optimizar las rutinas BLAS. Este formato divide las matrices en bloques de 16x16 y a cada uno de ellos se lo almacena en un formato distinto. En la

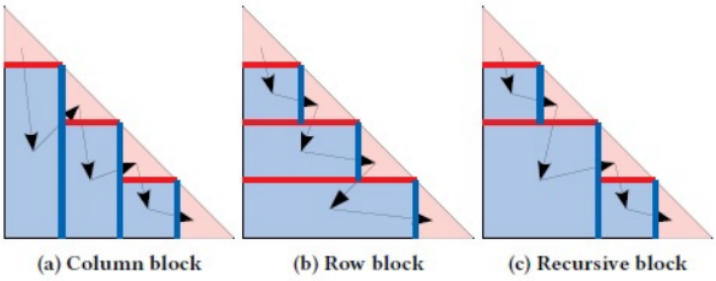


Figura 3.1: Esquemas de partición de matrices para los tres algoritmos a bloques. Tomado de [5].

Figura 3.2 se muestra el formato de almacenamiento aplicado a una matriz de ejemplo utilizando bloques de 4x4 por simplicidad. Los bloques cuadrados son procesados con las mismas estrategias que la implementación de SpMV de esta biblioteca. Las resoluciones diagonales para los *tiles* que solamente tienen elementos en la diagonal (y son almacenados en DIA) son procesados por un kernel que calcula las 16 incógnitas en paralelo. Finalmente, los bloques son tomados como elementos y organizados para procesarse tanto con el paradigma *level-set* y *sync-free*. Para los bloques diagonales que cuentan con más elementos por debajo de la diagonal (y, por ende, tienen dependencias) se utilizan distintas implementaciones. En el caso del paradigma *sync-free* se asigna a un hilo para resolver serialmente el *tile* (`thread_level_sptrsv`). Por otro lado, en la implementación *level-set* se utiliza un enfoque adaptativo que, en función del nivel de paralelismo de la matriz, resuelve el bloque triangular utilizando `thread_level_sptrsv` o `warp_level_sptrsv` una implementación *sync-free* similar a [25] pero en la que cada *warp* procesa un bloque diagonal.

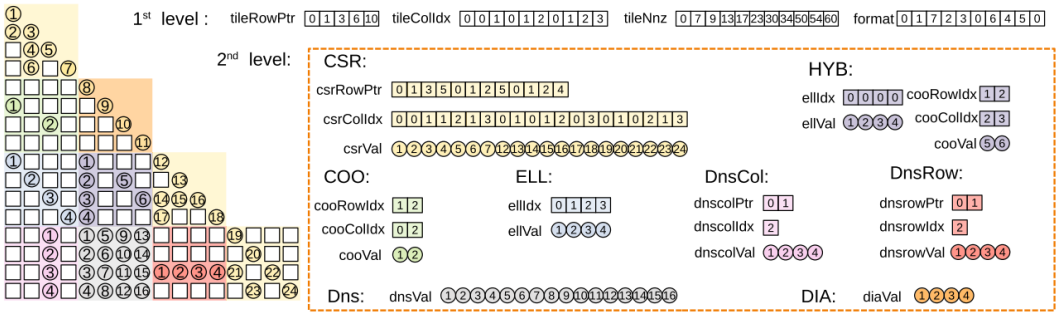


Figura 3.2: Formato `Tile` aplicado a una matriz de ejemplo. Tomado de [6].

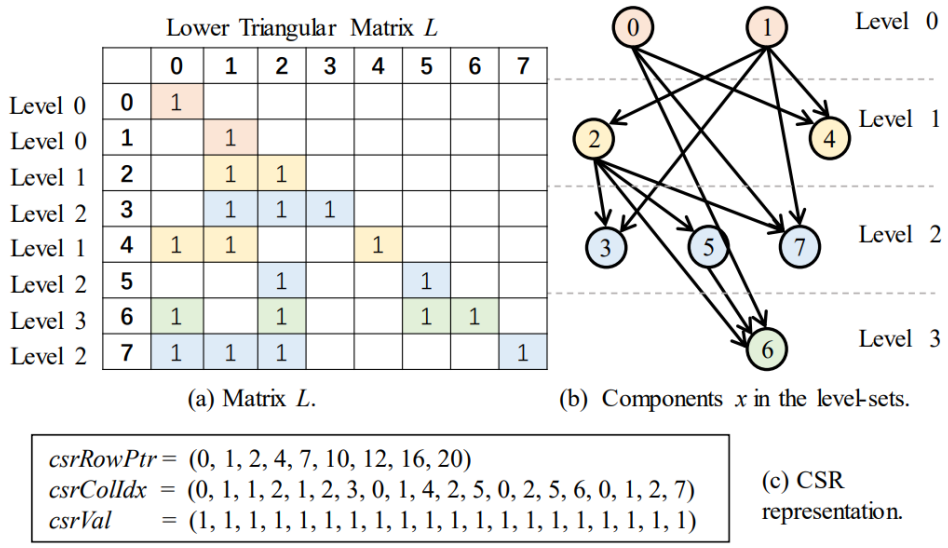
Como se mencionó anteriormente, uno de los principales problemas de las

implementaciones *sync-free* originales es la asignación de incógnitas a *warps*. En concreto, se tiene un problema cuando las filas (o columnas) tienen tan pocos elementos que generan que algunos de los procesadores asignados al *warp* queden ociosos. En esta línea, existen varios esfuerzos para mejorar la asignación de tareas a los *warps* e incrementar así el aprovechamiento de poder de cómputo. Como se mencionó anteriormente, el formato CSR es la estrategia de almacenamiento de matrices dispersas más popular. Todos los trabajos que se enumeran a continuación utilizan este formato.

Una propuesta enfocada en reducir los procesadores que quedan ociosos es la rutina CAPELLINI [7] que extendió la propuesta *sync-free* original en [25]. Esta rutina utiliza paralelismo a nivel de hilos en lugar de *warps*, procesando cada fila por un único hilo. En la Figura 3.4 se muestran tres esquemas de ejecución con la matriz de la Figura 3.3: un esquema *level-set* a nivel de hilo (en lugar de *warp* como [16]), el esquema tradicional *sync-free* a nivel de *warp* y el nuevo esquema *sync-free* a nivel de hilos, respectivamente.

Si bien, por simplicidad se muestra la versión *level-set* a nivel de hilos, se ve el problema de cómo el diseño en base a niveles limita el paralelismo. En un segundo lugar, se ve que la estrategia de *scheduling sync-free* genera una importante mejora tanto en tiempo como en uso de la GPU al aprovechar el segundo *warp*. Sin embargo, también se ve cómo quedan muchos hilos sin utilizarse. Finalmente, en la parte c, se ve cómo el esquema de ejecución es el que toma menos tiempo de ejecución pero genera dependencias entre hilos del mismo *warp* (por ejemplo hilos 2 y 3). A cada *warp* se le asignan $warp\_size$ (32) filas y, para evitar los posibles *deadlocks* si las filas dependen entre ellas, se divide la ejecución de los *warps* en dos partes. La primera parte procesa los elementos que no dependen entre sí y luego un *for* procesa los elementos que tienen dependencia dentro del mismo *warp*.

Al utilizar el formato de almacenamiento CSR (a diferencia de [25]) ya no es necesario el arreglo que mantiene la cantidad de filas de las que depende cada incógnita, sino que es suficiente con un arreglo booleano (get_value) que representa si la variable está o no resuelta. Con estos cambios, los hilos pueden trabajar en los elementos de la fila sin necesidad de esperar a que todas las incógnitas sean resueltas. Cada hilo verifica el estado de la variable y, si esta está en un estado no resuelto, realiza *busy waiting* hasta que el estado cambie, permitiéndole procesar el elemento. Finalmente, se acumulan los resultados parciales obtenidos tras calcular todos los elementos no nulos de la fila, lo que



(a) Matrix L. (b) Components x in the level-sets. (c) CSR representation.

Figura 3.3: Matriz de ejemplo y esquema de niveles resultante. Tomada de [7].

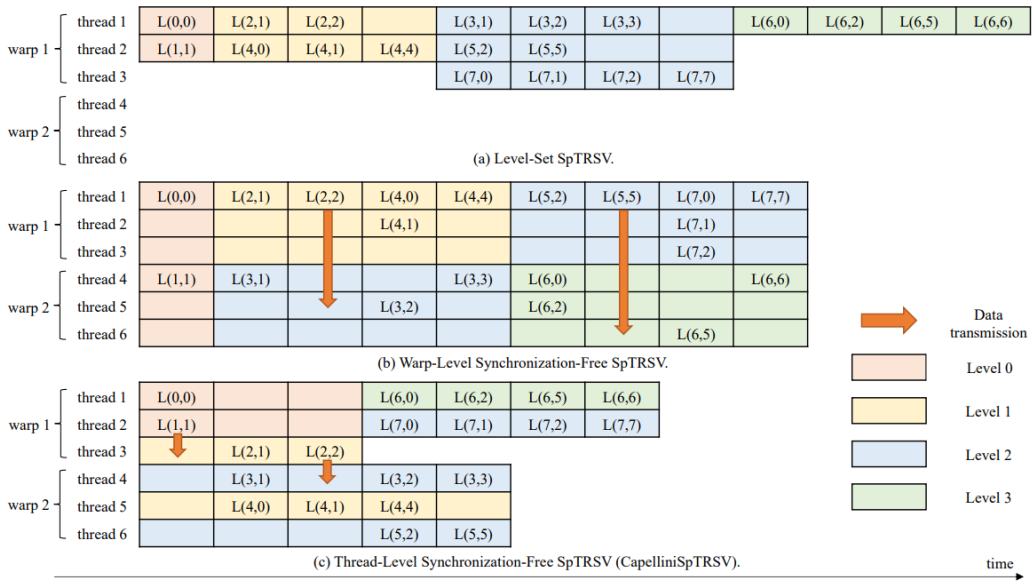


Figura 3.4: Esquema de ejecución *level-set* a nivel de hilos (a), esquema *sync-free* a nivel de *warp* (b) y esquema de ejecución *sync-free* a nivel de hilos (c). Tomada de [7]

determina el valor de la incógnita correspondiente. Luego, el valor de get_value de la fila se establece en *true*.

La rutina CAPELLINI fue extendida por YUENYEUNG [8]. Esta nueva propuesta buscó balancear el paralelismo a nivel de hilos de CAPELLINI con el paralelismo a nivel de *warp* (usado en la mayoría de las implementaciones

previas). Para definir qué filas son procesadas con qué enfoque se utiliza un preprocesamiento.

La estrategia para definir con qué esquema de paralelismo se procesa la fila intenta que las que tienen pocos no-ceros sean procesadas a nivel de hilo (un hilo por fila) mientras que las que tienen un número más elevado sean a nivel de *warp* (con *warp size* hilos por fila, 32 en el caso de NVIDIA). Además, debe tener en cuenta que las filas mantengan el orden de ejecución original (para evitar *deadlocks*) y evitar generar problemas de partición al combinar las dos estrategias (por ejemplo, si las filas 1 y 2 son procesadas a nivel de hilo pero la 3 a nivel de *warp*). En este sentido, los autores se decidieron por una estrategia bastante sencilla en la que dividen la matriz horizontalmente en bloques de *warp size* filas. Posteriormente, cada bloque es procesado con un mismo enfoque en función de si el promedio de no-ceros por fila es menor o mayor a cierto umbral. Para elegir el umbral los autores hacen un estudio sobre un conjunto de matrices relativamente grande y lo definen como 10.

Para almacenar el resultado del preprocesamiento se utiliza el vector `execute_row_id` que mantiene en qué índice de fila empieza a procesar cada *warp*. En la Figura 3.5 se muestra el esquema de ejecución de YUENYEUNG con m *warps* de n filas. El primer *warp* sabrá que debe procesar a nivel de hilo ya que la diferencia entre su entrada correspondiente en el vector `execute_row_id` y la siguiente es n. Por el contrario, el *warp* 2 procesa a nivel de *warp* ya que la diferencia es 1.

Otro enfoque para solucionar las restricciones al aprovechamiento de la capacidad de cómputo fue la rutina de resolución multi-filas ($SpTRSV_{mr}$) presentada en [9]. La rutina $SpTRSV_{mr}$ utiliza un preprocesamiento que tiene en cuenta la estructura de niveles y el tamaño de las filas de la matriz para reordenar las filas de la matriz. Además, a efectos de aprovechar todos los procesadores, las filas "chicas" (de menos de 16 no-ceros) son agrupadas con otras de igual o similar tamaño y que no dependan entre ellas. La etapa de resolución para filas chicas de la rutina subdivide a los *warps* que procesan filas chicas en vectores de 2^n hilos (con $n = 0, 1, ..., 5$), como las filas asignadas a vectores del mismo *warp* no dependen entre sí no existe la posibilidad de generar *deadlock*. Una explicación en detalle de esta rutina se brinda en la Sección 4.1.

En el mismo trabajo los autores propusieron otras implementaciones que mejoraban la propuesta original en [70]. Específicamente, propusieron una rutina que eliminó la necesidad del vector *is_solved* (equivalente al *get_value*

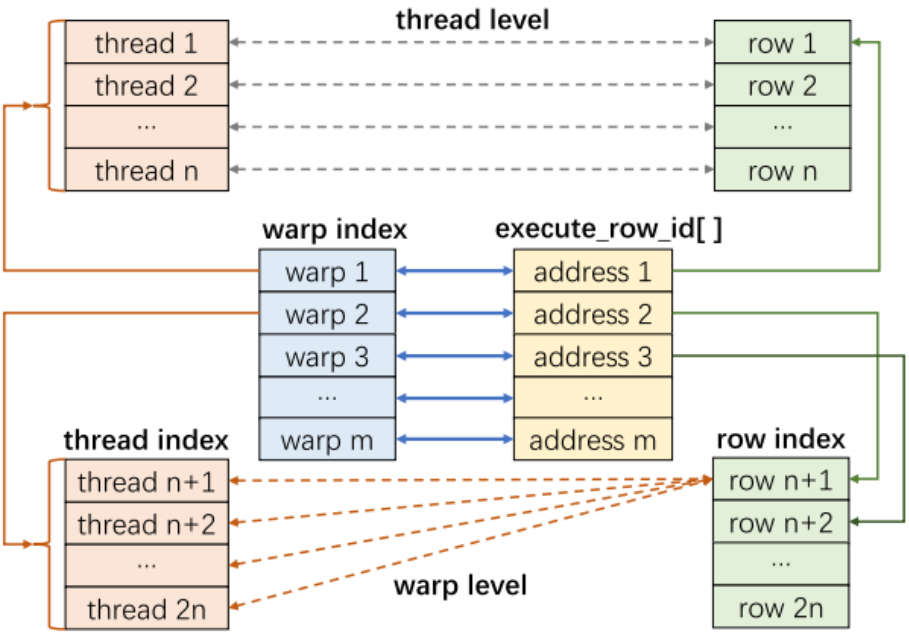


Figura 3.5: Esquema de ejecución de YUENYEUNG. Tomada de [8]

de CAPELLINI) y otra que reordena la ejecución de las filas en función de su nivel. El reordenamiento busca una ejecución similar a *level-set* en el sentido de que las filas del nivel i siempre ejecutan antes que las del nivel $i+1$ pero sigue un enfoque *sync-free*. En este enfoque no se espera a que todo el nivel termine para despachar el siguiente sino que se va haciendo parcialmente a medida que las dependencias se van cumpliendo.

3.2.1. Comparación de las distintas implementaciones

En la Tabla 3.1 se presenta un resumen de los trabajos mencionados en esta sección. No se incluye [78] al no ser resultados comparables ([78] es multi-GPU mientras que los otros no). En todos los casos, cuando se dice que una propuesta supera a otra se refiere a que lo hace en la gran mayoría de los casos de prueba.

En [72] se compara *sync-free*$_{liu2}$ contra la rutina original (*sync-free*$_{liu}$) obteniendo, en general, mejores resultados. Posteriormente, en [8] se compara YUENYEUNG contra *sync-free*$_{liu2}$ y CAPELLINI superando a ambas rutinas. Similarmente, en [6] se muestra que esta rutina supera a *sync-free*$_{liu2}$, CUSPARSE y recBlock. Por otro lado, en [9] los autores comparan la rutina $SpTRSV_{mr}$ contra *sync-free*$_{duf}$,*sync-free*$_{liu}$ y CUSPARSE superando a todas estas.

En la Tabla 3.2 se resumen estas comparaciones. En concreto, se presenta

Tabla 3.1: Resumen de los trabajos evaluados para la resolución de sistemas triangulares.

Id	Nombre	Año	Trabajo
1	cuSPARSE	2010	[80]
2	*sync-free*$_{liu}$	2016	[25]
3	*sync-free*$_{liu2}$	2017	[72]
4	*sync-free*$_{duf}$	2018	[70]
5	$SpTRSV_{mr}$	2020	[9]
6	CAPELLINI	2020	[7]
7	recBlock	2020	[5]
8	YUENYEUNG	2021	[8]
9	TileSpTRSV	2023	[6]

cada rutina especificando a qué rutinas supera y si el conjunto de matrices de evaluación es o no mayor a 40 matrices. Adicionalmente, a efectos de clarificar las comparaciones en la última columna se explicita qué rutinas la superan.

Tabla 3.2: Resumen de cuáles trabajos superan a otros y si el conjunto de evaluación es pequeño (< 40 matrices) o grande (≥ 40).

Id	Nombre	Supera a	Tamaño (P/G)	Superada por
1	cuSPARSE	-	-	[2-9]
2	*sync-free* $_{liu}$	[1]	P	[3-9]
3	*sync-free* $_{liu2}$	[1]	P	[6-9]
4	*sync-free* $_{duf}$	[1,2]	P	[5]
5	$SpTRSV_{mr}$	[1,2,4]	P	-
6	CAPELLINI	[1,3]	G	[8]
7	recBlock	[1,3]	G	[9]
8	YUENYEUNG	[1,3,6]	G	-
9	TileSpTRSV	[1,3,7]	P	-

Puede verse que tres rutinas ($SpTRSV_{mr}$, YUENYEUNG y TileSpTRSV) no son superadas por ninguna de las otras no habiendo trabajos que las comparen. Es importante notar que algunas comparaciones cuentan con conjuntos bastante acotados de matrices para su evaluación (por ejemplo, las comparaciones presentadas para TileSpTRSV en [6] son sobre un conjunto de 16 matrices).

3.3. Etapa de análisis del kernel SpTRSV

Previo a la incorporación del paradigma *sync-free* en GPU, el paradigma *level-set* definía el estado del arte para implementaciones del kernel de

SpTRSV siendo utilizado, por ejemplo, por la biblioteca CUSPARSE. La estructura de niveles también es utilizada en otras operaciones como la factorización ILU, por lo que, si bien el paradigma *level-set* ha perdido popularidad con la incorporación de *sync-free* para la resolución de sistemas triangulares aún cuenta con escenarios de aplicación.

Para generar la estructura de niveles que permita explotar el paralelismo en estas operaciones la implementación estándar consistía en hacer una recorrida similar a BFS [16]. En el Algoritmo 12 se presenta el pseudocódigo de esta estrategia que interpreta a la matriz como la matriz de adyacencia de un grafo y lo recorre asignando los nodos (filas) a niveles. Puede verse que el algoritmo hace una recorrida muy ineficiente ya que en el peor caso es $O(n^2)$ cuando cada nivel está compuesto por un solo elemento. Esto se debe a que cada nodo es visitado en todos los niveles hasta que es asignado a uno.

Algoritmo 12 Pseudocódigo de la rutina para calcular la estructura *level-set* adaptado de BFS. Simplificado de [16].

```
// In: M: Matriz
// Out: levs[]: lista de listas
l = 0
while hay nodos no visitados do:
    for i=1:n do:
        if no_tiene_dependencias(i)
            add(i,levs[l])
        endif
    endfor
    for nodo in levs[l]
        marcarProcesado(nodo)     //Marcar para no ser considerado en
            el primer for
        removerDependencias(nodo) //Todos los nodos que dependen de
            nodo eliminan dicha dependencia
    endfor
    l = l+1
endwhile
```

El paradigma *sync-free* puede ser utilizado para el cálculo de esta estructura. En esta línea en [73] los autores propusieron una implementación siguiendo una idea similar a [70] en cuanto al proceso de ejecución y la asignación de filas a *warps*. Sin embargo, en lugar de proceder a calcular la incógnita haciendo operaciones de punto flotante, calcula el nivel de la fila como el máximo de los niveles de las filas de las que depende más 1. En el Algoritmo 13 se muestra

una versión simplificada de esta rutina.

Algoritmo 13 Pseudocódigo de la rutina para calcular la estructura *level-set*. Simplificado de [73].

```
// In: n > 0, row_ptr[], val[], col_idx[]
// Out: levs[]
for wrp = 1  to  n-1 in parallel do: //Un warp por fila
    lne = threadIdx.x % 32 , elem = row_ptr[wrp] + lne , lev = 0
    while elem < row_ptr[wrp+1] do:
        colidx = col_idx[elem] , my_val = val[elem]
        ready = is_solved[colidx] //Acumulación local
        if ready
            lev = max(lev,levs[colidx])
        endif
        if all(ready) //Todos los hilos del warp terminaron
            elem += 32
        endif
    endwhile
    for i = [16, 8 ... 2, 1] do //Reducción de valores parciales
        lev =max(lev, __shfl_down(lev, i))
    endfor
     if lne == 0
        levs[wrp] = lev //Se escribe el resultado
        is_solved[wrp] = 1 //Se marca la fila como resuelta
    endif
endfor
```

La estructura *level-set* permite generar conjuntos de filas que no dependen entre ellas. Esto a su vez puede ser utilizado para permitir que varias filas ejecuten dentro de un mismo *warp* sin peligro de *deadlock* por depender una de la otra. En [9] se propone una rutina de análisis que busca hacer esto mismo para mitigar la capacidad de cómputo ociosa. Para esto, se divide las filas en grupos según dos criterios: el nivel en la estructura *level-set* y el tamaño de la fila. Respecto al tamaño, se definen 7 clases de equivalencia: $[0], [1], [2], [3-4], [5-8], [9-16], [17+]$. Dichas clases se definen en función de cuántas de ellas podrían ser procesadas en un mismo *warp*, es decir, un *warp* puede procesar 32 filas de la primera y segunda clase, 16 de la tercera, 8 de la cuarta y así sucesivamente. Finalmente, se genera un vector para hacer un reordenamiento simbólico de las filas que genere que las que comparten nivel y clase de equivalencia de tamaño se encuentren juntas. Luego, se agrupan las filas de las primeras seis clases de tamaño para explotar mejor los *warps* y no

dejar los procesadores ociosos.

Posteriormente, en un proyecto final de carrera de la Facultad de Ingeniería - UDELAR (Uruguay) [84] se hicieron implementaciones paralelas de la etapa de análisis. En concreto, se propusieron dos implementaciones paralelas, una de las cuales se ejecuta completamente en GPU. Estos trabajos fueron publicados en [26].

3.4. Otras aplicaciones

La resolución de *SpTRSV* es uno de los cuellos de botella más importantes para algunos métodos numéricos. Uno de los ejemplos más comunes en este sentido es la resolución de sistemas de ecuaciones lineales dispersos.

En [85] se presentó una variante del método BiCG de ILUPACK [86] en servidores con dos GPUs. Posteriormente, en [87] los autores extienden [85] con foco en explotar el paralelismo en servidores con una única tarjeta. La apuesta consiste en utilizar distintos *streams* para procesar en paralelo los dos conjuntos de operaciones relativamente independientes del método BiCG.

Los autores hacen dos propuestas iniciales. La primera divide los dos conjuntos de instrucciones y los asigna a dos *streams* distintos. Sin embargo, no logran que los *kernels* ejecuten a la vez en la GPU. La segunda alternativa busca mover a CPU los cálculos con matrices traspuestas en los que se ve que el rendimiento de las versiones en GPU no es bueno y se mantienen los dos *streams* para el resto de las operaciones de los dos conjuntos. Si bien esto permite paralelizar ciertas operaciones, los autores observaron que gran parte del tiempo de ejecución de la CPU se desperdicia en "overhead de lanzamiento de *kernels* de CUDA", es decir que la CPU gasta mucho tiempo en llamadas a la API. El problema es que para la resolución de sistemas triangulares se utilizaba `cusparseDcsrsv_solve` de la librería CUSPARSE que utiliza un enfoque *level-set* y despacha un *kernel* por nivel. Al sustituir esta llamada por la implementación *sync-free* de [70] logran reducir significativamente este sobrecosto y obtener *speedups* significativos.

Otro esfuerzo en esta línea se presenta en [77] donde se busca acelerar el método GMRES. En concreto, este trabajo continuó esfuerzos previos de acelerar la implementación de ILUPACK migrando parte de las rutinas a GPU [88, 89]. Los autores aplican la implementación *sync-free* de *SpTRSV* y los resultados obtenidos demostraron mejoras significativas en cuanto al tiempo de cómputo.

En otra línea, también se han empleado las técnicas *sync-free* directamente para la resolución de otros kernels de ALN dispersa. En especial kernels relacionados con factorizaciones LU. Un ejemplo relevante es el método *Strongly Implicit Procedure* (SIP), un método iterativo para la resolución de sistemas lineales provenientes de ecuaciones en derivadas parciales (PDEs por sus siglas en inglés -Partial Differential Equations-) en grillas regulares. Este método es utilizado en problemas de dinámica de fluidos y transferencia de calor, donde frecuentemente se resuelven sistemas de la forma $Ax = b$ donde A es una matriz penta o hepta diagonal si la grilla es 2D o 3D respectivamente, dado que los puntos de la grilla dependen únicamente de los inmediatamente adyacentes (izquierda, derecha, arriba y abajo en 2D). En el Algoritmo 14 se presenta el pseudocódigo serial del SIP. El método puede verse como una factorización LU incompleta y una etapa de refinamiento iterativo donde se resuelven dos sistemas triangulares en cada paso. Dada la geometría de la grilla, tanto la factorización LU incompleta como los sistemas triangulares se resuelven con ecuaciones explícitas que provienen de la grilla, es decir sin formar explícitamente la matriz dispersa.

Para paralelizar este algoritmo se utiliza lo que se llama "estrategia de hiperplanos". Esta estrategia se basa en que, debido a las dependencias de la grilla mencionadas anteriormente, los elementos de esta en los que la suma de sus índices es la misma son independientes y pueden ser procesados en paralelo. Por ejemplo, en una grilla 3D los elementos (1,3,2) y (2,2,2) pertenecen al mismo hiperplano (6) y son independientes entre ellos.

El enfoque tradicional para implementarlo en GPU consistía en procesar los hiperplanos en orden y utilizando un *kernel* por cada uno de estos [90–92]. El problema de esta estrategia es que cada *kernel* es relativamente liviano en cómputo y esto genera que el costo de lanzamiento de cada una de las rutinas sea significativo. El formato de ejecución recuerda al paradigma *level-set* en el que se fuerza a que todos los elementos de un nivel sean procesados antes de pasar al siguiente aunque cada elemento del segundo dependa solamente de una pequeña fracción de elementos del primero. En esta línea, se dedicaron esfuerzos a adaptarlo a un paradigma *sync-free* que evite el sobrecosto de lanzar múltiples *kernels*. En [75] los autores proponen un enfoque en el que cada hiperplano es partido para ser procesado por varios bloques. Cuando un bloque termina de procesar su parte del hiperplano actualiza una variable global y espera a que todos los otros bloques que estaban trabajando en ese

Algoritmo 14 Pseudocódigo del algoritmo SIP. Adaptado de [76].

```
// Entrada: A, b
// Salida: x

// Realizar la descomposición LU incompleta L̂Û ≈ A
L,U = ILU_decomposition(A)

// Calcular el residuo inicial: r_0 = b − A × x_0
r = b − A × x

while(r no es lo suficientemente pequeño) do:
    // Calcular el vec to r R_n = L̂^{-1} × r_n (sustitución hacia adelante)
    R_n = forward_substitution(L, r)

    // Calcular δx = Û × δx = R_n (sustitución hacia atrás)
    delta_x = backward_substitution(U, R_n)

    // Actualizar la solución: x_{n+1} = x_n + δx
    x = x + delta_x

    // Actualizar el residuo: r_{n+1} = b − A × x_{n+1}
    r = b − A × x
endwhile
```

plano terminen para pasar al siguiente. Este enfoque evita la necesidad de ejecutar un *kernel* por hiperplano. Sin embargo, dado que deben despacharse suficientes bloques para procesar el mayor hiperplano, en los primeros y los últimos se tienen hilos (y bloques) ociosos. En esta línea, en [76] se propone un enfoque en el que cada hilo procesa un elemento (e hilos contiguos procesan a elementos del mismo hiperplano). Este esquema de ejecución es similar a la propuesta *sync-free* de $SpTRSV$ original, en la que cada hilo hace espera activa hasa que sus dependencias han sido satisfechas.

Otro ejemplo de la aplicación del paradigma *sync-free* fue empleado también para la resolución de la factorización LU. En [74] los autores proponen una nueva implementación de la factorización LU conocida como *Sync Free LU* (SFLU) que sigue este paradigma.

La Rutina SFLU realiza la factorización LU de matrices dispersas dividiéndose en dos etapas: simbólica y numérica. En la etapa simbólica, se determina la estructura no nula de las matrices L y U sin calcular aún sus valores. Cada columna es procesada en paralelo por un bloque de hilos, donde se verifica el

arreglo `degree` para rastrear las dependencias entre columnas. Este arreglo almacena cuántos elementos sobre la diagonal aún deben ser eliminados antes de que una columna pueda procesarse. A medida que las columnas se completan, los valores de `degree` se actualizan de forma atómica, liberando el procesamiento de otras columnas dependientes. El objetivo de esta etapa es obtener el patrón de no-ceros de la matriz (considerando el *fill-in*). En las estructuras `indexL` e `indexU` se almacenan estos datos que luego serán utilizados por la etapa numérica

En la etapa numérica se calculan los valores de L y U utilizando la estructura generada en la etapa simbólica. Los bloques de hilos procesan en paralelo los elementos de cada columna, primero resolviendo los valores de U (incluyendo el elemento diagonal) y luego calculando los valores de L dividiendo por el elemento diagonal correspondiente. La sincronización entre niveles se evita utilizando el mismo mecanismo de dependencias basado en `degree`. En el Algoritmo 15 se presenta la etapa de resolución (numérica), donde puede verse que el algoritmo tiene similitudes con el Algoritmo 10 en cuanto a su estrategia por columnas y el uso de operaciones atómicas para mantener la consistencia.

3.5. Conclusiones

En esta sección se presenta un repaso de los distintos campos de aplicación del paradigma *sync-free* en el contexto del ALN dispersa. Entre los kernels abordados se destaca especialmente la resolución de sistemas triangulares dispersos donde existe una considerable cantidad de trabajos. Para la $SpTRSV$, si bien no existe una implementación que supere a las otras en todos los casos, las implementaciones que tienen los mejores resultados siguen este paradigma. En concreto, tres implementaciones ($SpTRSV_{mr}$ [9], YUENYEUNG [8] y TileSPTRSV [6]) no son superadas sistemáticamente por ninguna otra propuesta.

Si bien el $SpTRSV$ es la aplicación de referencia para el paradigma *sync-free*, el campo de aplicación de este paradigma excede esta rutina. En concreto, este paradigma puede ser aplicado a diversos kernels de álgebra lineal como la factorización LU y el SIP, o a métodos iterativos como GMRES y BiCG.

Algoritmo 15 Algoritmo *sync-free* para la factorización numérica en GPU con matrices dispersas. Adaptado de [74].

```
for k = 1  to  n in parallel do // Un bloque por columna
    while (degree[k] != 0)
        // busy wait
    endwhile

    // Calcular los valores de U (incluyendo la diagonal)
    for i = 1  to  k-1 where A[i, k] != 0 do
        for j = i+1  to  n where A[j, k] != 0 in parallel do // Un
            hilo por no-cero
            A[j, k] = A[j, k] - A[j, i] * A[i, k]
        endfor
        ATOMIC_SUB(degree[k], 1)
    endfor

    // Calcular los valores de L
    for i = k+1  to  n where A[i, k] != 0 in parallel do
        A[i, k] = A[i, k] / A[k, k]
    endfor

    // Actualizar el grado de la columna
    ATOMIC_SUB(degree[k], 1)
endfor
```

Capítulo 4

Propuesta para la mejora de la resolución de Sistemas Triangulares Dispersos

En este capítulo se presentan los aportes principales de la tesis aplicados a la resolución de sistemas triangulares dispersos. En la Sección 4.1 se explica en profundidad la línea base que se utiliza como punto de partida para la comparación de las nuevas propuestas desarrolladas en este trabajo. En la Sección 4.2 se describen en detalle las dos extensiones al kernel de $SpTRSV$ propuestas en el marco de la maestría. Finalmente, en la Sección 4.3 se resume la evaluación experimental llevada adelante y, en la Sección 4.4 se resumen los resultados obtenidos y se brindan las conclusiones finales relativas a las propuestas.

4.1. Línea base ($SpTRSV_{mr}$)

Este apartado explica los principales detalles de la implementación de $SpTRSV_{mr}$ que es utilizada como línea base para este trabajo. El código está disponible de forma pública en GitHub[1] y fue publicado en [9].

La rutina $SpTRSV_{mr}$ es una rutina híbrida que utiliza ideas de ambas estrategias de sincronización (*sync-free* y *level-set*) y cuenta con dos etapas. Primero, se hace un preprocesamiento (que es común para todas las resoluciones hechas para la misma matriz) y luego una etapa de resolución.

[1] https://github.com/HCL-Fing/SPTRSV/tree/main

La etapa de preprocesamiento se describe en profundidad en [26] por lo que a continuación se explica la rutina a grandes rasgos. Primero, se asigna un número de nivel a cada fila siguiendo el paradigma *level-set*. Posteriormente, las filas del mismo nivel son clasificadas en varios grupos dependiendo de la cantidad de coeficientes no-cero fuera de la diagonal ($max_p/nnz \leq 2^{p-1}$ con $p = 1, 2, 3, 4, 5$ más los casos de $nnz \geq 16$ y $nnz = 0$). Por ejemplo, filas con dos no-ceros (sin contar el no-cero en la diagonal) son asignadas a la categoría $p = 2$ pero filas con tres (o cuatro) no-ceros son asignadas a la categoría $p = 3$. Finalmente, se asignan filas a *warps*. Para evitar *deadlocks* las filas procesadas por cada *warp* deben ser independientes, es decir, pertenecer al mismo nivel. Para esto, se asigna iterativamente filas del mismo nivel y grupo a un *warp* hasta llenarlo.

La etapa de análisis puede ser implementada en paralelo y computada en la GPU. La implementación paralela de esta rutina fue parte importante de un proyecto final de carrera de la Facultad de Ingeniería - UDELAR (Uruguay) [84]. La salida de esta rutina es un vector de permutaciones (`row_order`, que da el orden de las filas), un vector que delimita dónde comienzan las filas que procesa cada *warp* (`base_index`) y un vector que contiene el grupo relativo al tamaño de los vectores (p) de dicho *warp* (`part_size`).

La etapa de resolución divide los *warps* en vectores de igual tamaño, que es dado por el tercer vector de salida del análisis. El tamaño de los vectores está relacionado con las filas que procesan. En concreto, si los vectores de un *warp* específico son de tamaño 2^p esto significa que procesarán filas de tamaño relativamente similar (entre $2^{p-1} + 1$ y 2^p). Por ejemplo, si el *warp* tiene asignadas filas del grupo $p = 2$ será dividido en 16 vectores de dos hilos cada uno y procesará, por ende, 16 filas. Por el contrario, una fila con 20 no-ceros será procesada por un vector del grupo $p = 6$, que estará compuesto por todos los hilos de su *warp*. En la Figura 4.1 se presenta un esquema que muestra el ciclo de vida de un *warp* que procesa filas del grupo $p = 3$ (es decir, 4 filas de entre 5 y 8 no-ceros cada una).

Cada *warp* recibe como parámetro (en memoria global) los tres vectores que se obtienen como salida de la etapa de análisis. Utilizando estos vectores, y en función de su posición relativa dentro del *warp*, cada hilo puede calcular la fila y el no-cero asignados para él. Una vez que el hilo tiene esta información, comienza la etapa de *busy waiting* en la que constantemente pregunta por el valor de la incógnita (que es inicializada en `NaN`) y si obtiene un valor válido

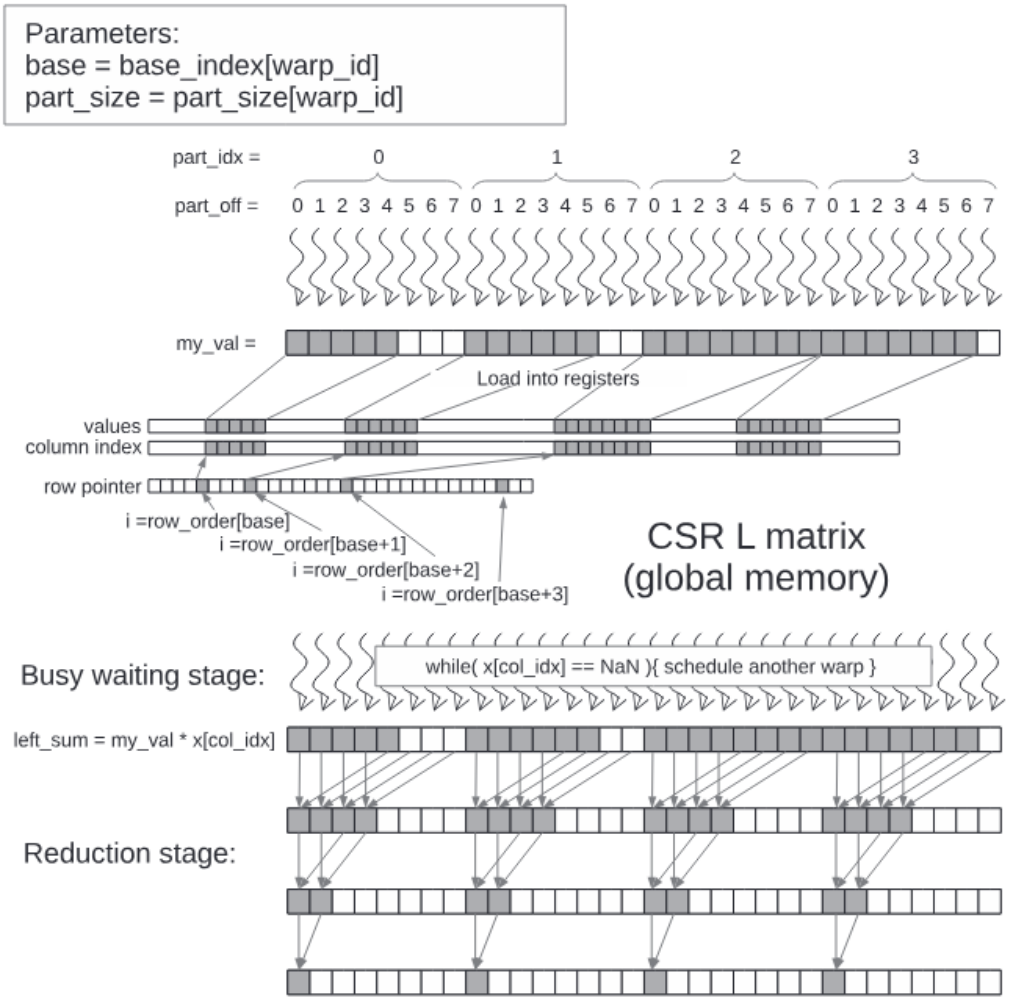


Figura 4.1: Ciclo de vida de un *warp* de $SpTRSV_{mr}$. El *warp* recibe el tamaño de la partición (8 en este caso) y el índice, en el vector `row_order`, de la primera fila a procesar. Tomada de [9].

puede continuar con la ejecución.

Una vez que todos los hilos del *warp* obtienen los valores de las incógnitas correspondientes, el *warp* pasa a la etapa de resolución en la que los hilos multiplican la incógnita por el coeficiente obtenido. Finalmente, los valores obtenidos son reducidos como muestra la Figura 4.1 haciendo las reducciones de las distintas particiones en paralelo utilizando las operaciones de *shuffle*. Si la fila se encuentra en el último grupo (una sola fila para todo el *warp*) es posible que los hilos deban iterar antes de pasar a la etapa de reducción (en el caso de que la fila tenga más de 32 no-ceros).

4.2. Propuestas

En esta sección se presentan las grandes líneas de trabajo propuestas en lo relativo a la optimización de la rutina $SpTRSV_{mr}$. En primer lugar, en el Apartado 4.2.1 se presenta el nuevo formato de almacenamiento para matrices dispersas específicamente diseñado para optimizar la eficiencia de la resolución de sistemas triangulares dispersos. Este formato busca optimizar el acceso a memoria y el aprovechamiento de caché. Posteriormente, en el Apartado 4.2.2, se presenta la propuesta del cambio en la asignación de niveles a filas.

4.2.1. Nuevo formato de almacenamiento ($HYB_{syncfree}$)

En [9], los autores se enfocaron en evitar la espera innecesaria al reordenar y procesar las filas en orden de sus niveles, y evitar desperdiciar cómputo ejecutando varias filas en un mismo *warp* cuando estas son lo suficientemente pequeñas. Para avanzar con estos dos desafíos se usan estructuras auxiliares que explicitan el orden de ejecución y la asignación de filas a *warps*. En la práctica dicha propuesta hace un reordenamiento simbólico de la matriz ya que en memoria se sigue manteniendo la original en formato CSR. Si bien, en general, el análisis experimental muestra que estos reordenamientos tienen un impacto positivo, existen casos en los que empeoran el tiempo de ejecución.

Como muchas rutinas de álgebra dispersa, la $SpTRSV_{mr}$ es una rutina *memory bound* y sufre de accesos irregulares a memoria. Esto empeora al aplicar los reordenamientos salidos de la etapa de análisis. Como se dijo anteriormente, la salida del análisis son las estructuras para un reordenamiento simbólico pero en ningún punto se hace un reordenamiento físico. Por lo tanto, no hay ninguna garantía de que los distintos bloques procesarán filas contiguas lo que genera dificultades para explotar la memoria compartida o las caches. Adicionalmente, agrupar las filas pequeñas en un solo *warp* genera que éste tendrá que operar con valores de, potencialmente, varias filas que se encuentran separadas en memoria.

El desarrollo de formatos de almacenamiento para matrices es una estrategia muy común para mejorar el desempeño de las rutinas de álgebra dispersa como la $SpMV$ o $SpTRSV$. En general estos formatos buscan mejorar el acceso a memoria, a veces a cambio de aumentar el costo de almacenamiento. Además de considerar el factor del tamaño en memoria, al desarrollar formatos nuevos se debe considerar el *overhead* que surge de transformar la matriz desde

un formato más general (como CSR) para ejecutar la operación. Sin embargo, este sobrecosto se puede amortizar, en particular si la rutina es ejecutada muchas veces con la misma matriz.

En este marco se desarrolló un nuevo formato de almacenamiento disperso, $HYB_{syncfree}$, que mejora el acceso a memoria de la etapa de resolución. El diseño de este formato toma ideas de CSR y ELL. Una matriz en este formato es representada por dos arreglos, `vals` y `cols` de tamaño $n_{wrp} \times 32$ donde n_{wrp} es el número de *warps* calculado por la etapa de análisis. Adicionalmente, $HYB_{syncfree}$ cuenta con una estructura similar al formato CSR para almacenar las filas grandes (es decir, con más de 32 elementos). En las filas correspondientes a los *warps* que procesan hasta 32 no-ceros (tanto en una como muchas filas), la posición i de los arreglos `vals` y `cols` almacena el coeficiente e índice de columna que procesará el i-ésimo hilo del *warp*. Para estos *warps* el formato es similar a ELL. La principal diferencia es que, dado que en $SpTRSV_{mr}$ los *warps* son subdivididos en particiones del mismo tamaño que procesan distintas filas pequeñas, el *zero padding* se aplica dentro de cada partición. Esto genera que hilos contiguos procesen elementos contiguos en memoria permitiendo un mejor aprovechamiento de la cache de cada bloque, algo que es especialmente importante en los *warps* que procesan varias filas.

Por el contrario, cuando el *warp* procesa una única fila de más de 32 elementos, el primer valor de la fila en cada uno de los arreglos se utiliza como puntero a la parte CSR de la estructura que guarda la fila. Mientras en `cols` se guarda el elemento del vector `row_ptr` que corresponde al inicio de la fila, en `vals` se apunta al comienzo de la siguiente. Esto permite no almacenar el vector `row_ptr` en la parte CSR del formato y reduce ligeramente los tiempos de acceso a memoria de la etapa de resolución. Finalmente, para enmascarar la latencia de la indirección provocada por CSR se utiliza el resto de la fila para almacenar los primeros 31 valores. De esta manera los hilos $1-31$ pueden iniciar su ejecución mientras el hilo 0 decodifica la dirección del resto de la fila.

En la Figura 4.2 se presenta un diagrama del formato aplicado a una matriz de ejemplo. La primera fila que se muestra en detalle corresponde a un *warp* que procesa 8 filas de (hasta) 4 elementos cada una. Puede verse que la fila de la matriz que se encuentra en la primera partición es de tamaño 3 y por tanto se completa con un 0 en el cuarto lugar en ambos arreglos. Finalmente, las otras dos filas que se muestran en detalle corresponden a *warps* que procesan

una única fila de más de 32 elementos. En la primera, se puede ver que en el arreglo de índices de columna se almacena el 28, correspondiente al elemento de la fila en el vector `row_ptr` de la matriz, y en la matriz de valores se almacena el 71, correspondiente al elemento de la fila siguiente en el vector `row_ptr`. Por último, puede verse que en el resto de la fila se repiten los elementos de la matriz CSR para poder procesarlos en paralelo con la decodificación de la posición que hace el hilo 0.

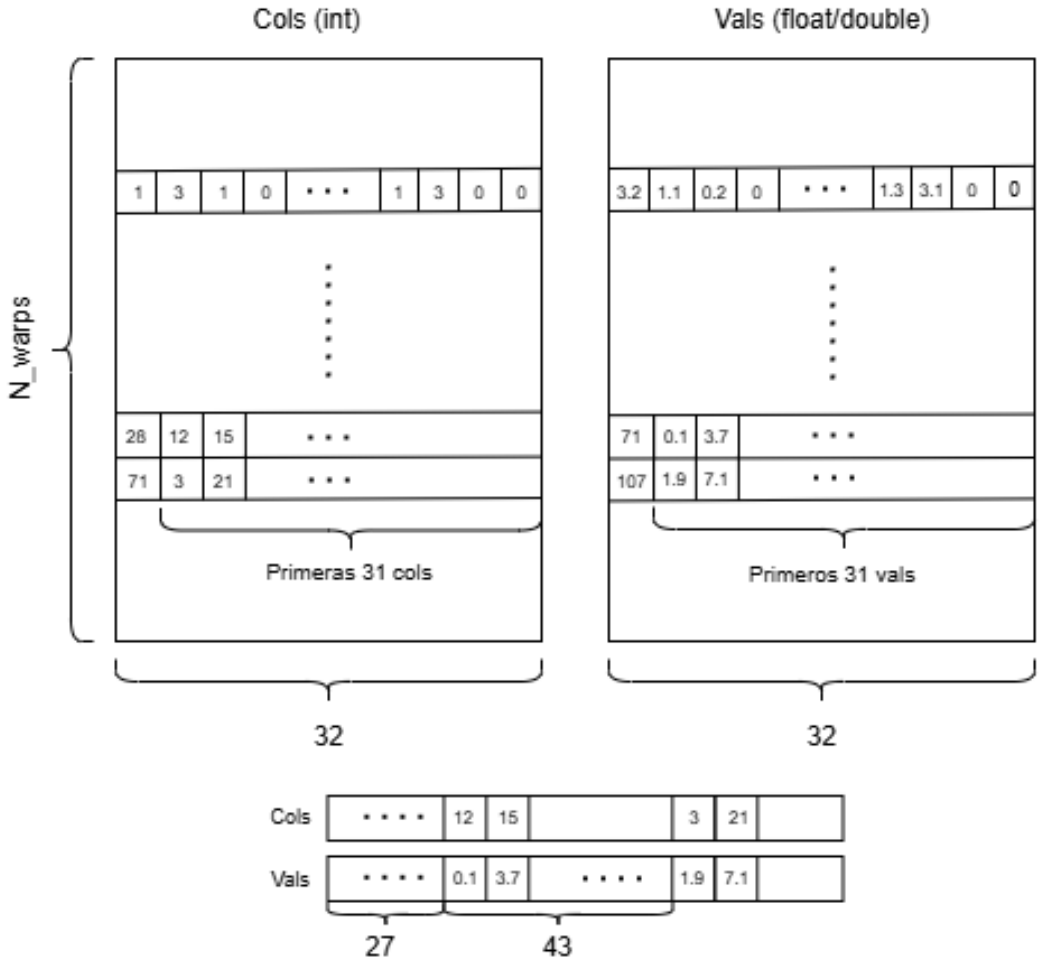


Figura 4.2: Formato de almacenamiento de la matriz aplicado a una matriz de ejemplo. La primera fila que se muestra en detalle corresponde a un *warp* particionado de a 4 mientras que las otras dos filas a *warps* que procesan una única fila de más de 32 elementos.

Es relativamente fácil ver que el nuevo formato tendrá un acceso a memoria mucho más regular dado que los hilos contiguos acceden a direcciones de memoria contiguas incluso en los *warps* que procesan varias filas pequeñas

y separadas en la matriz original. Para filas grandes, se diluye la indirección generada por el uso de CSR ya que, si bien se debe pasar por el vector `row_ptr` solamente el primer hilo debe postergar su ejecución para leer el índice mientras los otros 31 ya pueden procesar sus elementos en el primer instante. Finalmente, el acceso a memoria es alineado, lo que no está garantizado en CSR.

Como se mencionó anteriormente, las estrategias de diseñar formatos específicos suelen implicar el aumento de uso de memoria para mejorar el tiempo de ejecución. En esta línea, es interesante evaluar el costo de memoria del formato y compararlo con el costo de CSR. El costo de CSR puede ser calculado directamente en función de la cantidad de filas y el número de no-ceros obteniendo la Ecuación 4.1

$$M = nnz \times (4 + 8) + n + 1 Bytes \tag{4.1}$$

donde n es el número de filas y nnz el de no-ceros.

Por el contrario, el costo del nuevo formato depende también de otras variables como el largo de las filas y la estructura de las dependencias (que impactan en la cantidad de *warps* requeridos). La cantidad de memoria utilizada por los dos arreglos de la parte del formato similar a ELL está dado por la Ecuación 4.2.

$$M = 32 \times (4 + 8) \times n_{wrp} \tag{4.2}$$

En cambio, calcular el consumo de memoria de la parte estilo CSR no es trivial, ya que requeriría contabilizar cuántos no-ceros quedan por fuera de la estructura principal. Por estos motivos es complejo comparar analíticamente el costo de memoria de ambos formatos, aunque es evidente que la nueva propuesta requerirá más memoria, ya que utilizará la misma cantidad de memoria por no-cero (12 bytes si se asume punto flotante) pero se agrega, al menos, los costos de *zero padding*.

Si la matriz será utilizada posteriormente (por ejemplo si la resolución del sistema triangular es parte de un método iterativo) se puede sustituir la parte CSR por la propia matriz original accediendo a las partes correspondientes. Esta filosofía es la que sigue la implementación desarrollada entendiendo que, en la mayoría de las aplicaciones, será necesario tener la matriz en formato CSR, y el cambio hacia y desde el formato propuesto será un *overhead* para

aplicar esta rutina en lugar de otra implementación de $SpTRSV$. Si bien la Ecuación 4.2 da una expresión analítica de cuánta memoria adicional se utilizará en el nuevo formato, la comparación con CSR es difícil ya que los formatos dependen de distintas variables.

Considerando que la descripción analítica no ofrece conclusiones sólidas, en el apartado de evaluación experimental se calcula el uso de memoria para diversas matrices.

4.2.2. Cambio en la estructura de niveles

En este apartado, se presenta en detalle la nueva estrategia propuesta para la asignación de niveles a filas. Para comprender la idea detrás de la nueva estructura de niveles propuesta es útil analizar cuál es el significado del nivel en el paradigma *level-set*. Supongamos entonces un escenario ideal con una cantidad infinita de procesadores, y en el que se divide el tiempo en intervalos discretos tales que procesar cada fila (computar multiplicaciones y acumulaciones una vez que las dependencias fueron calculadas) toma una unidad de tiempo. En el escenario descrito, asignar a la fila a el nivel i es equivalente a decir que el primer momento en el que esta puede procesarse es T_i. Es decir, que T_i es el primer momento en el que todas las dependencias de la fila a ya están computadas. Este paradigma se conoce como *As Soon As Possible* (ASAP) [93] ya que cada fila es procesada en el primer instante en el que es posible hacerlo.

Sin embargo, no es estrictamente necesario procesar la fila a en el instante T_i. Concretamente, si la ecuación b es la primera que tiene un no-cero en la columna a (es decir, es la primera fila que depende de la fila a), a puede ser procesada en el rango de tiempos entre i y $nivel(b) - 1$ sin afectar el tiempo de finalización de la rutina. En particular, si ninguna ecuación tiene el término x_a, la ecuación a puede ser resuelta en cualquier tiempo posterior a i. Si bien esto es óptimo a efectos de resolver la fila, este planteamiento *greedy* puede no ser óptimo para el sistema en su conjunto. Otra forma de ver esta situación es que la fila no está necesariamente en el camino crítico de ejecución.

En el escenario ideal planteado anteriormente, en el que se puede procesar cualquier número de ecuaciones en paralelo, no tendría sentido en ningún caso atrasar la resolución de una ecuación. Sin embargo, en un escenario con una cantidad de recursos (procesadores) limitado, posponer una ecuación que

no forma parte del camino crítico puede impactar positivamente, permitiendo que alguna otra ecuación que sí forma parte de este ejecute antes. En la Figura 4.3 se muestra un ejemplo de cómo la resolución puede acelerarse al posponer ecuaciones que no están en el camino crítico en un escenario con dos procesadores. En el ejemplo original la rutina toma 4 tiempos para finalizarse ya que, al haber tres filas en el nivel 2 la fila 5 puede ejecutar recién en T_3 atrasando la resolución de la fila 6 que depende de ésta. Por el contrario, con el nuevo modelo de niveles se posterga la fila 4 que no forma parte del camino crítico y se procesa en T_3 en conjunto con la fila 6 que ahora sí puede ejecutar. Este modelo se conoce como *As Late As Possible* (ALAP) [93] ya que se espera para procesar la fila al instante anterior al que es necesaria.

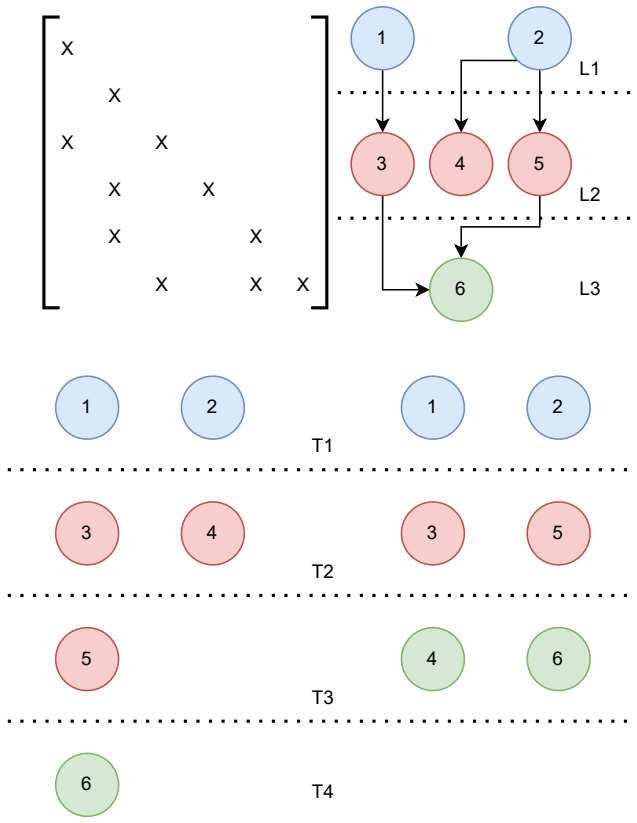


Figura 4.3: Ejemplo de la resolución de un sistema lineal en una plataforma que puede resolver hasta dos ecuaciones en paralelo. En la parte superior se muestra la matriz dispersa y la estructura de niveles original. En la parte inferior izquierda la ecuación 6 no puede ser resuelta en T_3 porque depende de la ecuación 5. En la parte inferior derecha, la ecuación 4 es pospuesta y enviada al siguiente nivel al no pertenecer al camino crítico.

Como se explica en la Sección 4.1, la rutina $SpTRSV_{mr}$ usa una estrategia

de tipo *level-set* para generar el orden en el que las filas serán procesadas en la etapa de resolución *sync-free*. En particular, asigna filas a *warps* de acuerdo al nivel y la cantidad de no-ceros, garantizando que todas las filas de un nivel serán asignadas a *warps* antes de comenzar a asignar filas del nivel siguiente. Dado que las GPUs pueden procesar solamente una cantidad finita de filas en paralelo, esta forma de asignación puede generar que filas que no están en el camino crítico ejecuten antes y dejen a las que si lo están postergadas esperando por procesadores libres.

Para atacar este problema en la asignación se modificó cómo se asignan los niveles a las filas a efectos de generar que las filas que no son parte del camino crítico sean asignadas a niveles posteriores. La propuesta modifica el significado de los niveles, que pasan a representar el último tiempo en el que una fila puede ejecutar sin demorar la ejecución de ninguna de las filas que dependen de ella. En otras palabras el nivel asignado a la fila i es el máximo número que cumple que:

- Es mayor o igual que el nivel que tenía asignado la fila en la estructura original de niveles
- Es menor que el mínimmo de todas las filas que dependen de i

Se implementó una versión del análisis siguiendo esta nueva idea para la estructura de niveles. El cálculo de niveles se divide en dos kernels: iteraciones hacia adelante y hacia atrás. La iteración hacia adelante funciona de manera similar al kernel del cálculo de niveles de la estrategia anterior, comenzando por la primera fila y calculando el nivel de cada fila como el máximo nivel de todas sus dependencias más uno. La iteración hacia atrás, por el contrario, sigue una lógica completamente distinta, comenzando desde las hojas del árbol de dependencias. Las hojas mantienen el nivel de la estructura anterior, mientras que para una fila i que no es hoja, el nivel se calcula como el mínimo nivel de las filas que dependen de i menos uno. Los Algoritmos 16 y 17 muestran el pseudocódigo de las dos iteraciones. Ambos *kernels* siguen un paradigma *sync-free*.

Es importante notar que la iteración hacia atrás requiere, como *input*, el número de filas que dependen de cada fila (es decir el número de no-ceros en cada columna). Dado que las matrices estan guardadas en CSR esta información no es fácil de computar para una fila cualquiera, por lo que se añadió una

Algoritmo 16 Pseudocódigo de la iteración hacia delante

```
for fila = 1  to  n in parallel do
  lvl = 0;
  // esperar a que  to das las dependencias estén listas
  for col in fila
    lvl = max(lvl,lvl_vect[col])
  endfor
  lvl_vect[fila] = lvl+1
endfor
```

etapa de preprocesamiento que cuenta, para cada fila, el número de no-ceros. Esto se guarda en el vector llamado `depending`.

Algoritmo 17 Pseudocódigo de la iteración hacia atrás

```
//lvl_vect es inicializado en MAXINT previo a ejecutar esta iteración
for fila in parallel do
  while(depending[fila] > 1){} // esperar a que las filas que
      dependen de ésta hayan calculado sus valores
  if(lvl_vect[fila] == MAXINT) //verdadero si es una hoja
    lvl = max_level        //max_level es el máximo número de nivel
       hallado en la primera iteración
    lvl_vect[fila] = lvl
  else
    lvl =lvl_vect[fila]
  endif
  for col in fila
    lvl_vect[col] = a to micMin(lvl_vect[col], lvl-1)
    depending[col] = a to micSub(depending[col],1)
  endfor
endfor
```

En la iteración hacia atrás, cada fila hace *busy waiting* sobre un contador que representa la cantidad de filas dependientes de ésta que aún no se han procesado. Una vez que este contador es cero hay dos escenarios: o bien en `lvl_vect` se encuentra el valor `MAXINT` (lo cual significa que no tiene filas dependientes ya que ninguna ha modificado esta entrada del vector) o bien contiene un índice de nivel correspondiente al mínimo de todas las filas dependientes. Si contiene `MAXINT`, quiere decir que la fila en cuestión es una hoja. Por tanto el número de nivel debe ser el máximo obtenido en la primera iteración a efectos de enviar a la fila al final de la ejecución, ya que no hay filas

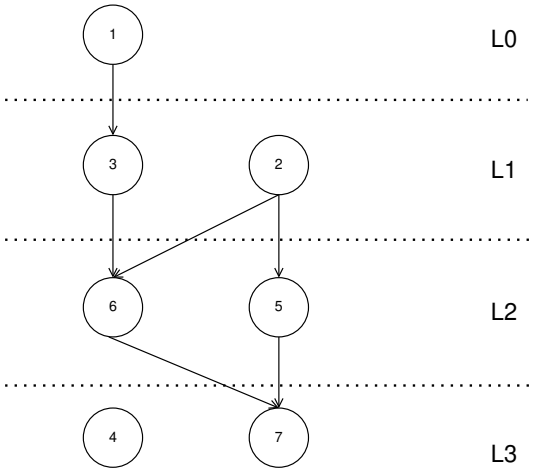


Figura 4.4: Estructura de niveles resultante de la iteración hacia atrás

que dependan de ella. En caso contrario se debe asignar como nivel el valor encontrado en el vector menos uno para garantizar que se la hará ejecutar lo más tarde posible pero antes que todas las filas dependientes. Finalmente, una vez que el nivel es calculado, la fila es la encargada de notificar su valor a todas las filas que esperan por ella. Esto es equivalente a actualizar la entrada correspondiente en `level_vect` para todas las filas de las que i depende (índices de columna de sus no-ceros) y posteriormente restarle uno a la entrada en el vector `depending` correspondiente a estas (líneas 10 a 13).

Por ejemplo, la estructura de niveles para la matriz presentada en la Figura 2.8 se muestra en la Figura 4.4. El camino crítico (ecuaciones 1, 3, 6 y 7) permanece incambiado pero la fila 4 pasa de nivel 2 a nivel 3 ya que ninguna fila depende de ella. La arista de 3 a 4 desaparece ya que pasa a no tener efecto. Por el contrario, la fila 5 no pasa a nivel 3 como la Figura 2.10 sugeriría, ya que la dependencia con la fila 7, que pasa a tener importancia, significa que tiene que ser, como mucho, nivel 2.

4.3. Evaluación experimental

Esta sección describe los experimentos desarrollados para validar las nuevas rutinas y sus comparaciones con las rutinas disponibles de manera pública.

La medida general que se utiliza para comparar las distintas rutinas es el *speedup*. El *speedup* de la rutina X con respecto a la rutina Y se calcula como la división de sus tiempos de ejecución (t_Y/t_X). Por lo tanto, valores

mayores a 1 significan que la rutina X es más rápida. La evaluación está organizada de la siguiente manera. Primero, en los Apartados 4.3.1 y 4.3.2 se presenta la comparación contra la resolución de $SpTRSV_{mr}$ de la etapa de resolución que usa el nuevo formato ($SpTRSV_{Fmt}$) y la nueva estructura de niveles ($SpTRSV_{Lev}$) respectivamente. Finalmente, las rutinas son comparadas con implementaciones de estado del arte disponibles en bibliotecas públicas.

Todas las rutinas se encuentran publicadas como código abierto en GitHub[2]. Las evaluaciones se hacen sobre un conjunto de matrices proveniente de la colección SuiteSparse [94] que tienen más de $10k$ filas y caben en la memoria de la GPU.

El servidor utilizado en la evaluación tiene un procesador Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz con 64GB de RAM, 64kB de cache L1, 256kB de cache L2, y 8MB de cache L3. La GPU utilizada es una NVIDIA RTX 3090 Ti con 10.496 cores y 24 GB de RAM GDDR6X. La versión utilizada del CUDA Toolkit es 11.4. Todas las experimentaciones fueron hechas utilizando aritmética de punto flotante de doble precisión.

4.3.1. Evaluación de $SpTRSV_{Fmt}$

En este apartado se presenta la comparación de la implementación propuesta de $SpTRSV$ que utiliza el nuevo formato de almacenamiento con la línea base. Se inicia la evaluación comparando el tiempo de ejecución de la resolución de $SpTRSV_{Fmt}$ con el de $SpTRSV_{mr}$. Los resultados son presentados en la Figura 4.5. Para facilitar la visualización se presentan los resultados ordenados por el tiempo de ejecución de $SpTRSV_{mr}$ y en tres grupos separados: las matrices que tienen un tiempo de ejecución entre 1 y 10 ms, las que tienen entre 10 y 100 ms y las que tienen tiempos de ejecución mayores a 100 ms. Se deja fuera de esta imagen las matrices que toman menos de 1 ms en el entendido de que los cambios no son significativos. Este estudio permite valorar los *speedup* obtenidos contextualizados por su impacto en el tiempo absoluto de ejecución.

Los resultados de tiempo de ejecución muestran que la nueva rutina tiene, en general, menores tiempos de ejecución que la línea base. Estos tiempos incluyen el tiempo requerido para convertir la matriz al formato propuesto. Las

[2] https://github.com/HCL-Fing/SPTRSV/tree/A-new-sparse-storage-format-and-level-set-analysis-for-the-GPU-synchronization-free-SPTRSVFormat-Levels

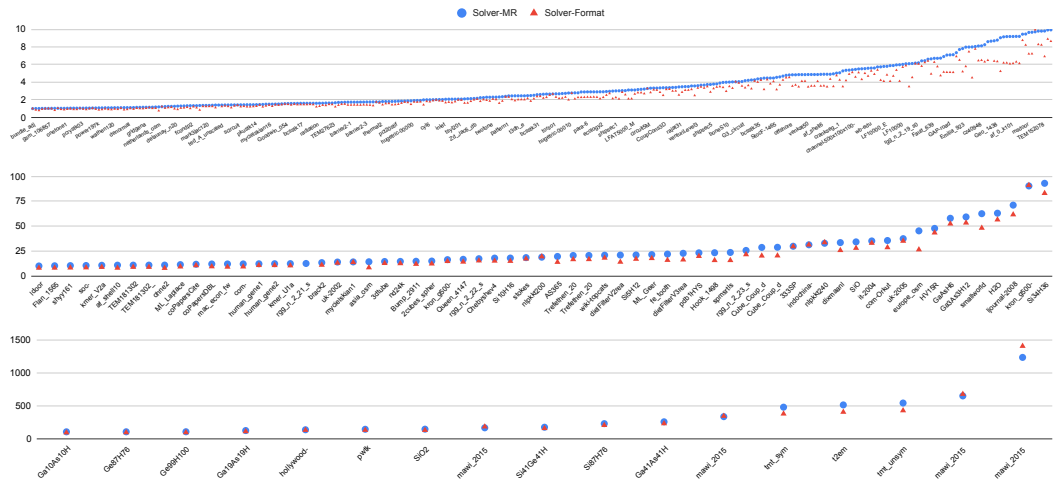


Figura 4.5: Tiempo de ejecución de $SpTRSV_{Fmt}$ y $SpTRSV_{mr}$ (en ms). Los resultados están ordenados por el tiempo de ejecución de $SpTRSV_{mr}$ y divididos en tres grupos de matrices: las que tienen un tiempo de ejecución entre 1 y 10 ms (arriba), las que están entre 10 y 100 (medio), y las que toman más de 100 ms (abajo). Matrices que toman menos de 1 ms son dejadas afuera.

mayores disminuciones en tiempo de ejecución son obtenidas en el tramo inicial (entre 1 y 10 ms). Por el contrario, en el último tramo salvo en 4 matrices, las diferencias no son significativas. Los tiempos de ejecución de esta variante no presentan una relación obvia con una métrica específica. En la Figura 4.6 se presenta el *speedup* de $SpTRSV_{Fmt}$ contra la línea base en función del número de no-ceros de la matriz.

La tendencia general es que cuando las matrices son más grandes, los resultados mejoran llegando a obtener un *speedup* de 1.77×. Es importante notar que existe un conjunto de matrices de entre 64.000 y 85.000 no-ceros en el que el *speedup* varía entre $0,5$ y 1. Un análisis detallado muestra que este conjunto está compuesto en su mayoría por un grupo de matrices llamadas `as_caida_-G_...` donde el nuevo formato obtiene resultados peores que la línea base.

Otro estudio importante de la propuesta refiere al *overhead* de memoria, en este sentido la Figura 4.7 muestra el uso de memoria de CSR y el nuevo formato. Las áreas representan el logaritmo de la memoria usada en Bytes de CSR y la estructura secundaria de manera independiente (es decir, el área roja no incluye el área azul). En general, la estructura secundaria usa un tamaño similar al de la representación original en formato CSR lo que implica que el costo de memoria de la nueva rutina es 2× la línea base.

Es importante notar que no es estrictamente necesario utilizar toda la ma-

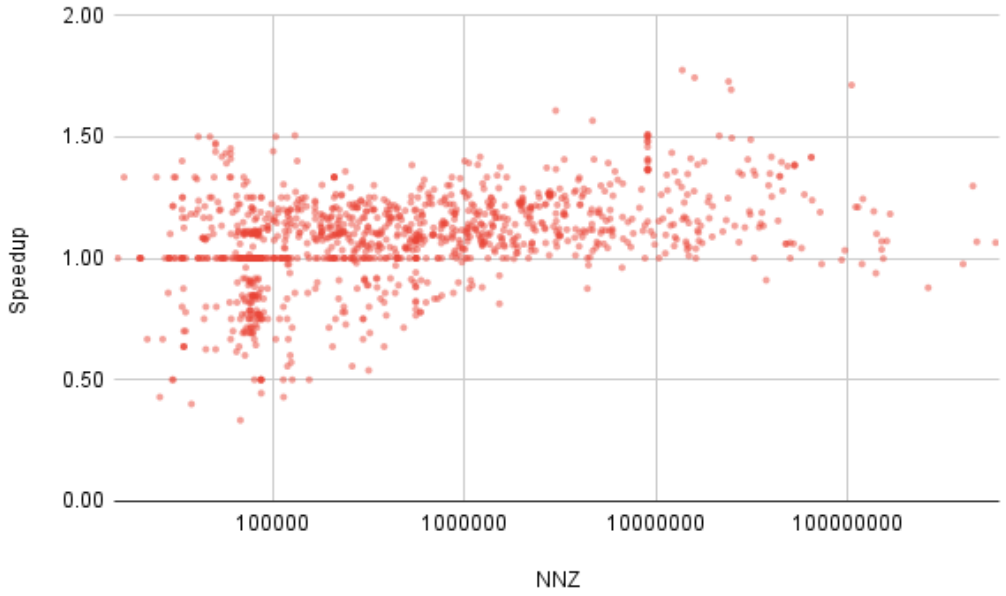


Figura 4.6: Valor de *speedup* de $SpTRSV_{Fmt}$ contra $SpTRSV_{mr}$ en función del tamaño de la matriz (tomado como número de no-ceros).

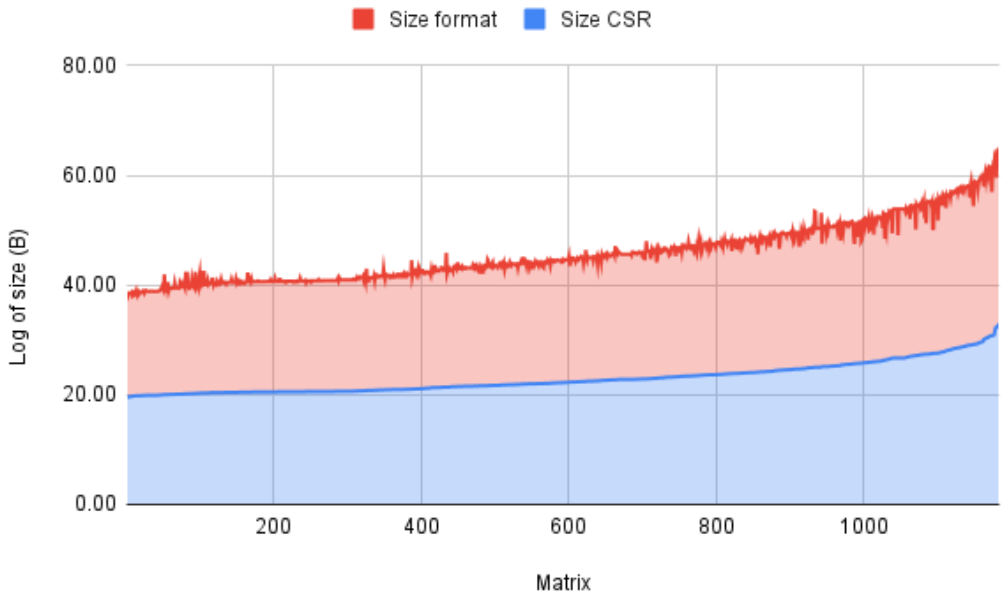


Figura 4.7: Logaritmo del costo de memoria (en Bytes) de la estructura CSR y la estructura secundaria.

triz CSR en el formato propuesto. En concreto, como se explicitó en la Sección 4.2.1, no es necesario almacenar el vector `row_ptr` ni los coeficientes o

índices de columna de los valores que se almacenan en la estructura secundaria.

Para terminar este análisis, se explora el *trade-off* entre aumentar el tiempo de la etapa de análisis y la mejora de la etapa de resolución. Esto es importante en el sentido de que varias aplicaciones requieren muchas resoluciones de sistemas triangulares definidos por la misma matriz dispersa con distintos coeficientes o distinto término independiente (es decir, matrices con el mismo patrón de dispersión). Dado que el análisis solamente depende del patrón de la matriz este puede ser reutilizado en todas las iteraciones de resolución y por lo tanto las ganancias en la segunda etapa pueden compensar el aumento del costo en la primera. La Figura 4.8 muestra la cantidad de iteraciones de la etapa de resolución requeridas para compensar el sobrecosto del análisis de $SpTRSV_{Fmt}$ contra $SpTRSV_{mr}$. No se incluyen puntos por las matrices en las que la línea base es mejor que la nueva rutina. Esto pasa en un total de 209 matrices de un total de 1182.

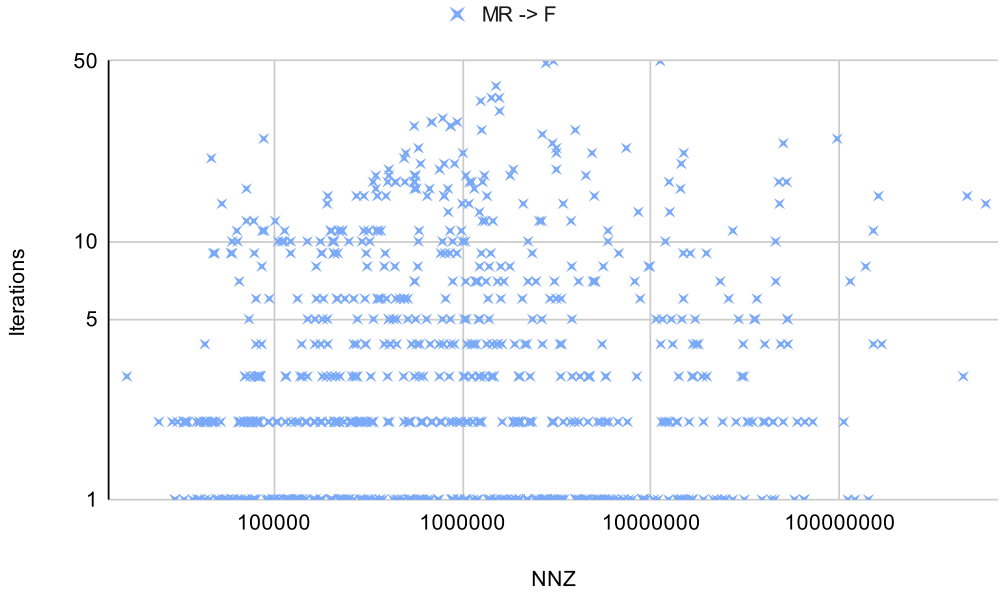


Figura 4.8: Número de iteraciones de la etapa de resolución requeridas para compensar el costo de ejecutar la etapa de análisis de $SpTRSV_{Fmt}$ (F) si se compara con $SpTRSV_{mr}$ (MR).

Los resultados muestran que, en la mayoría de las matrices, la etapa de resolución de $SpTRSV_{Fmt}$ requiere 10 iteraciones o menos para compensar el análisis. Además, hay un gran conjunto de matrices que requiere tres o menos

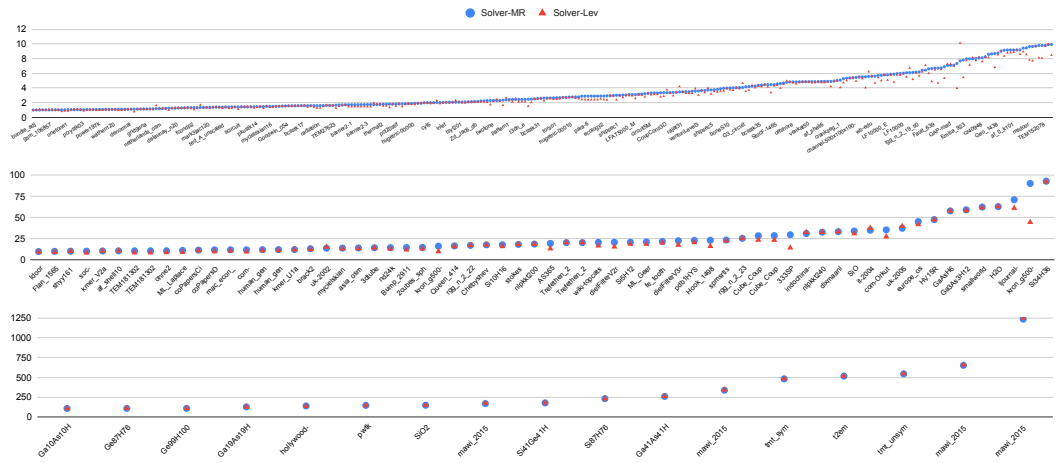


Figura 4.9: Tiempo de ejecución de $SpTRSV_{Lev}$ y $SpTRSV_{mr}$ (en ms). Los resultados están ordenados por el tiempo de ejecución de $SpTRSV_{mr}$ y divididos en tres grupos de matrices: las que tienen un tiempo de ejecución entre 1 y 10 ms (arriba), las que están entre 10 y 100 (medio), y las que toman más de 100 ms (abajo). Matrices que toman menos de 1 ms son dejadas afuera.

iteraciones (y, en particular, muchas una única iteración).

4.3.2. Evaluación de $SpTRSV_{Lev}$

En este apartado se presenta la comparación de la implementación propuesta de $SpTRSV$ que utiliza la nueva estrategia de asignación de niveles contra la línea base. Al igual que en el apartado anterior se inicia el análisis de la rutina presentando el tiempo de ejecución de la etapa de resolución de $SpTRSV_{Lev}$ comparado con la de $SpTRSV_{mr}$. Los resultados son presentados en la Figura 4.9 y muestran un patrón similar a la variante anterior.

La primera observación es que la diferencia entre las dos variantes es, en general, pequeña. Por otro lado, en la mayoría de las matrices ($\approx 60\,\%$) el *speedup* es mayor a 1 y en la mayoría de los casos en los que esto no sucede, el empeoramiento es acotado, con *speedup* mayor a 0.8 ($30\,\%$). Esto significa que en más del $90\,\%$ de las matrices la nueva etapa de resolución es ligeramente mejor o comparable con la de $SpTRSV_{mr}$. Sin embargo, existen algunas matrices, en especial en la figura del medio y la superior, donde la nueva estrategia obtiene una mejora de rendimiento considerable.

Para entender las diferencias de rendimiento se puede analizar los resultados de *speedup* en función del número de niveles, dado que $SpTRSV_{Lev}$ modifica cómo se asigna las filas a niveles. En la Figura 4.10 se muestra el *speedup* de

$SpTRSV_{Lev}$ contra $SpTRSV_{mr}$ en función de esta métrica.

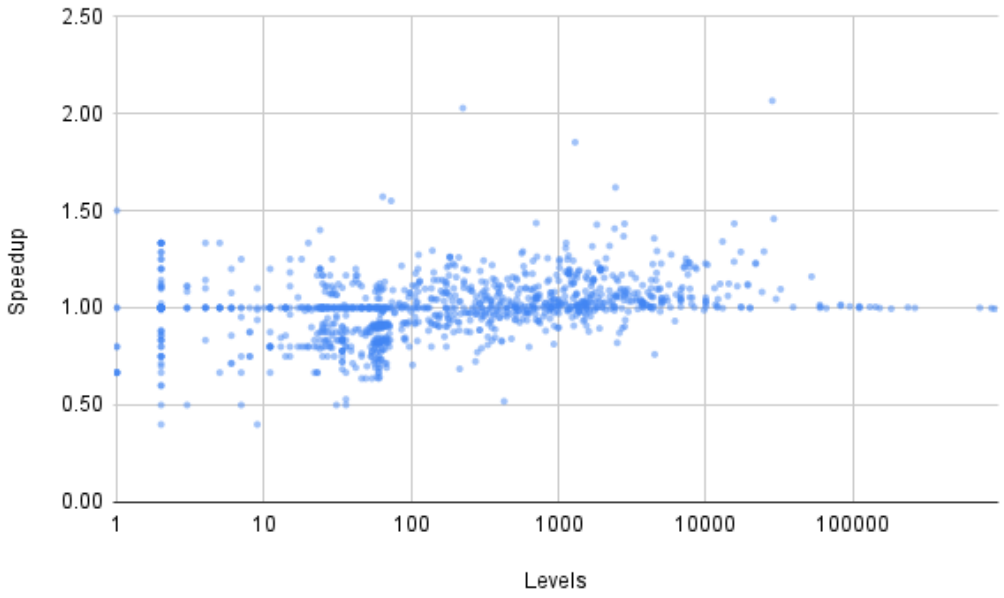


Figura 4.10: Valor de *speedup* de la etapa de resolución de $SpTRSV_{Lev}$ contra $SpTRSV_{mr}$ en función del número de niveles.

Como regla general, puede verse que cuando el número de niveles crece la aparición de valores de *speedup* menores a 1 es cada vez menos frecuente. Cuando una matriz tiene pocos niveles, hay menos restricciones para el paralelismo y por lo tanto es más probable que las filas puedan ejecutar en un orden similar al original. Mantener el orden original (o uno similar) ayuda a que el acceso a memoria sea mejor y se explote la localidad de datos ya que aumenta la probabilidad de que las filas contiguas sean procesadas por el mismo *warp* o *warps* cercanos (en el mismo bloque). Por el contrario, cambiar el nivel de las filas puede generar que estas sean enviadas a *warps* distintos y, por tanto, empeorar el uso de memoria y caché de la rutina. Por lo anterior, es razonable esperar que la línea base no sea demasiado peor que la nueva propuesta o incluso la supere.

Por el contrario, cuando aumenta el número de niveles, hay más espacio para hacer cambios con mayor impacto (pueden haber más filas que están fuera del camino crítico y pueden ser movidas a niveles superiores). Además, al haber muchos niveles es probable que la gran mayoría de las filas hayan sido reordenadas. En este caso, el reordenamiento más agresivo que hace $SpTRSV_{Lev}$ no es penalizado, ya que el acceso a memoria ya está distorsionado en $SpTRSV_{mr}$.

Sin embargo, en matrices con muchos niveles (y en particular en las que hay pocas filas por nivel como las tridiagonales) es probable que no existan muchos cambios para hacer y por tanto el tiempo de ejecución sea muy similar para ambas variantes.

En esta línea, se probaron distintas métricas que permiten cuantificar la diferencia entre la estructura de niveles tradicional y la propuesta. Estas métricas fueron el máximo cambio de nivel para una fila, el número de filas en las que no hay cambio (es decir, el número de filas en las que el cambio es 0) y el promedio de cambios de nivel en sus filas. Estudios preliminares muestran que las dos primeras métricas están muy relacionadas con el número de niveles. En este sentido, la Figura 4.11 muestra los resultados de *speedup* ordenados en función del promedio de cambios de nivel para sus filas.

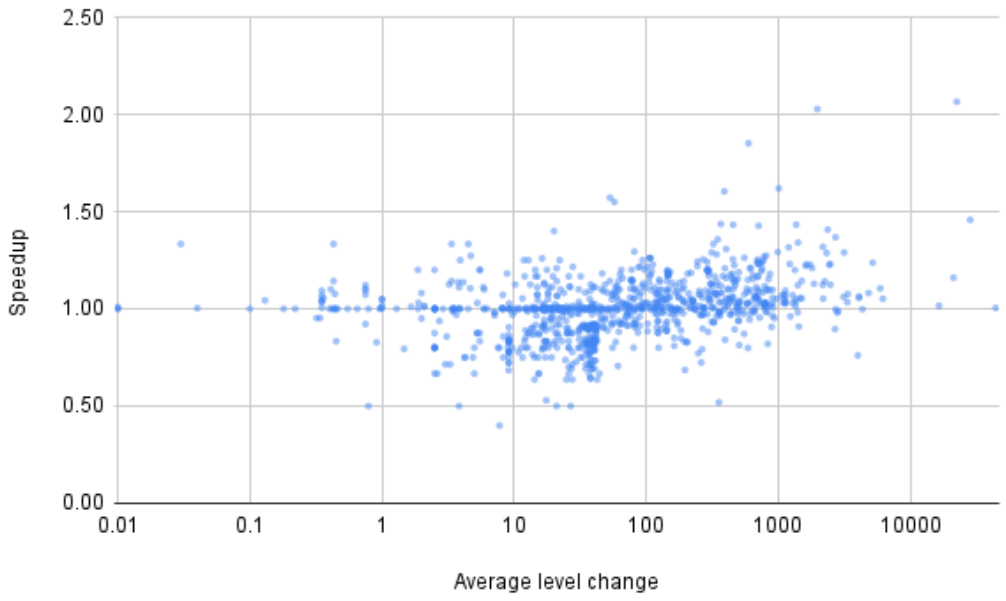


Figura 4.11: Resultados de *speedup* de $SpTRSV_{Lev}$ contra $SpTRSV_{mr}$ como función del promedio de cambios de nivel en las filas de la matriz.

Hay varias observaciones para hacer sobre este gráfico. Es claro que el promedio es un buen estimador de cuánto ha cambiado la estructura de niveles entre las dos estrategias. En este sentido, es razonable que, cuando el promedio de cambios es bajo, los resultados sean similares a los obtenidos por la línea base. En un segundo lugar, existe un intervalo intermedio cuando el promedio varía entre 1 y 100 donde los resultados son mezclados en los que ninguna rutina tiene más de un 40 % de ventaja sobre la otra. Finalmente, cuando el

promedio de cambios es mayor que 100, los valores de *speedup* se ubican, en general, por arriba de 1.

Una conclusión razonable es que cuando el cambio es suficiente, la nueva estrategia de asignación de filas tiene un impacto positivo que supera a las pérdidas generadas por un peor acceso a memoria. Esto se genera debido a que se priorizan las filas que forman parte del camino crítico en desmedro de las otras que son postergadas. Por el contrario, cuando el cambio es moderado, las mejoras obtenidas por un esquema de ejecución más inteligente pueden (o no) ser compensadas por una peor localidad de datos que impacta directamente en el uso de la caché.

Considerando lo anterior, se puede incorporar esta optimización en la rutina y esperar que los resultados positivos pesen más que los negativos. Sin embargo, dado que existen casos en los que los resultados son bastante negativos, es interesante explorar enfoques adaptativos que permitan elegir dinámicamente entre las dos estrategias. Dado que la etapa de análisis de $SpTRSV_{Lev}$ incluye a la de $SpTRSV_{mr}$ es posible decidir, una vez finalizada esta última, si continuar con la segunda parte. Esto permitiría, por ejemplo, usar datos como el número de niveles y otras métricas calculadas en esta etapa para decidir si continuar o no. Una estrategia de estas características debería priorizar tomar la decisión correcta en las matrices en las que la diferencia entre ambas estrategias es mayor. Esta línea de trabajo se deja como trabajo futuro.

Para finalizar el apartado, es interesante repetir el análisis sobre la cantidad de iteraciones de la etapa de resolución necesarias para compensar el sobrecosto de análisis. En este caso, se presentan las comparaciones tanto contra la línea base como contra $SpTRSV_{Fmt}$. A la variante que utiliza tanto la nueva estructura de niveles como el formato de almacenamiento se la llama $SpTRSV_{FL}$. Al igual que en el análisis anterior, se dejan afuera de la figura las matrices en las que la etapa de resolución de $SpTRSV_{Lev}$ es más lenta que aquella con la que se la compara. Esto sucede en 663 matrices al comparar $SpTRSV_{Lev}$ contra la línea base y en 475 si se la compara contra $SpTRSV_{Fmt}$ (de un total de 1182 matrices).

La primera conclusión es que el nuevo formato complementa positivamente a la estructura de niveles ya que hay 180 matrices en las que aplicar esta estrategia de niveles en $HYB_{syncfree}$ mejora la etapa de resolución mientras que hacerlo en CSR empeora. Esto tiene sentido porque, como se explicó anteriormente, la nueva estrategia de niveles puede perjudicar la localidad en ciertos

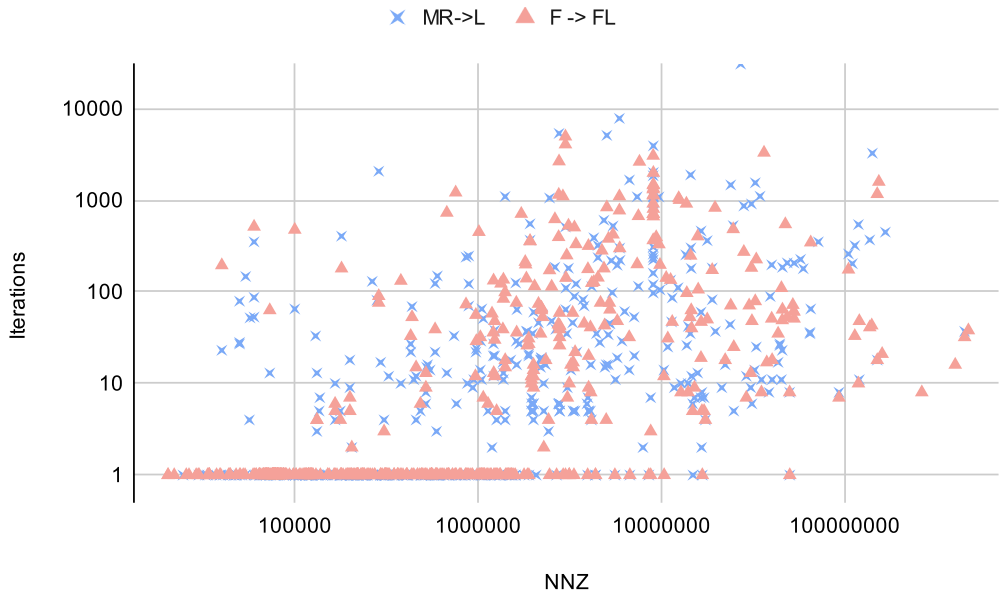


Figura 4.12: Número de iteraciones requeridos para justificar la ejecución del análisis que genera la nueva estructura de niveles si se compara con $SpTRSV_{mr}$ (MR->L) y con $SpTRSV_{Fmt}$ (F->FL). Esta gráfica muestra todas las matrices en las que la nueva estrategia de resolución supera a $SpTRSV_{mr}$ o $SpTRSV_{Fmt}$.

casos, pero al utilizar $HYB_{syncfree}$ esto ya no sucede. En segundo lugar, incluso en los casos en los que al comparar con ambas rutinas se obtienen ganancias, generalmente se requieren más iteraciones para compensar el sobrecosto del análisis cuando se parte de $SpTRSV_{mr}$ que de $SpTRSV_{Fmt}$. En concreto, existe un conjunto nada despreciable de matrices en el que se necesitan más de 100 iteraciones, aunque en promedio se requieren menos de 6 iteraciones para $SpTRSV_{Fmt}$ y tan solo 1 para $SpTRSV_{mr}$. Es decir que si es posible caracterizar los casos en los que $SpTRSV_{Lev}$ no empeora los resultados, en general serán necesarias pocas iteraciones para justificarlo.

4.3.3. Comparación con el estado del arte

Para finalizar la evaluación experimental de las propuestas, se compara $SpTRSV_{FL}$ contra otras implementaciones de $SpTRSV$ en GPU publicadas recientemente. En particular, se eligen tres implementaciones conocidas: cusparse [80], Capellini [7][3] y Yuenyeung [8][4]. Del análisis del estado del

[3]https://github.com/JiyaSu/CapelliniSpTRSV
[4]https://github.com/JiyaSu/YuenyeungSpTRSV

arte se tiene que YUENYEUNG, $SpTRSV_{mr}$ y TileSPTRSV [6] son las rutinas candidatas a tener mejores desempeños, desgraciadamente en [6] no se brinda la implementación de manera pública. Por otro lado, si bien CAPELLINI es superada por YUENYEUNG según el análisis presentado en [8], análisis preliminares muestran que CAPELLINI supera en un conjunto no despreciable de matrices a YUENYEUNG y, en general, es competitiva con esta última. Finalmente, aunque CUSPARSE no parece ser la mejor implementación de $SpTRSV$, es una biblioteca madura que ha sido mantenida y revisada durante muchos años, y es utilizada generalmente como punto de comparación para las distintas propuestas. En el caso de CUSPARSE se utiliza la rutina de dos etapas `cusparse_csrsv2` (con la bandera `CUSPARSE_SOLVE_POLICY_USE_LEVEL`), que es relativamente comparable con las implementaciones propuestas y se presume que utiliza una implementación *sync-free* [95].

Las nuevas rutinas requieren una relativamente costosa etapa de análisis y están diseñadas para *solvers* iterativos en los que se ejecutan muchas etapas de resolución por cada una de análisis. Aunque el nuevo análisis no está completamente optimizado, se incluye su costo de ejecución para hacer una evaluación justa con CAPELLINI y YUENYEUNG que tienen etapas de análisis extremadamente livianas.

Se presentan tres escenarios distintos para la comparación. Primero, un escenario como el mencionado anteriormente, con una gran cantidad de iteraciones sobre la misma matriz. En la Figura 4.13, se presenta el tiempo de ejecución de la etapa de análisis más 100 iteraciones de la etapa de resolución para las cuatro rutinas mencionadas anteriormente. Luego, en la Figura 4.14, se presenta un escenario con menor cantidad de iteraciones (10 resoluciones más análisis). Finalmente, en la Figura 4.15, se presentan comparaciones de únicamente el tiempo de ejecución de la rutina de resolución.

La Figura 4.13 muestra que CUSPARSE es la peor rutina en casi todas las matrices y, en general, por un amplio margen. Luego, los resultados de CAPELLINI y YUENYEUNG son relativamente similares con una pequeña diferencia a favor del segundo. Finalmente, la nueva propuesta tiene los mejores resultados de las cuatro rutinas evaluadas. En concreto, $SpTRSV_{FL}$ obtiene los mejores resultados en 922 de las 1156 matrices para las que se evaluaron los *kernels*.

Dado que la Figura 4.14 representa un contexto con solamente unas pocas iteraciones por matriz se incluyen los resultados de $SpTRSV_{Fmt}$. Esto se debe a que el análisis de esta última requiere bastante menos tiempo y, por los estudios

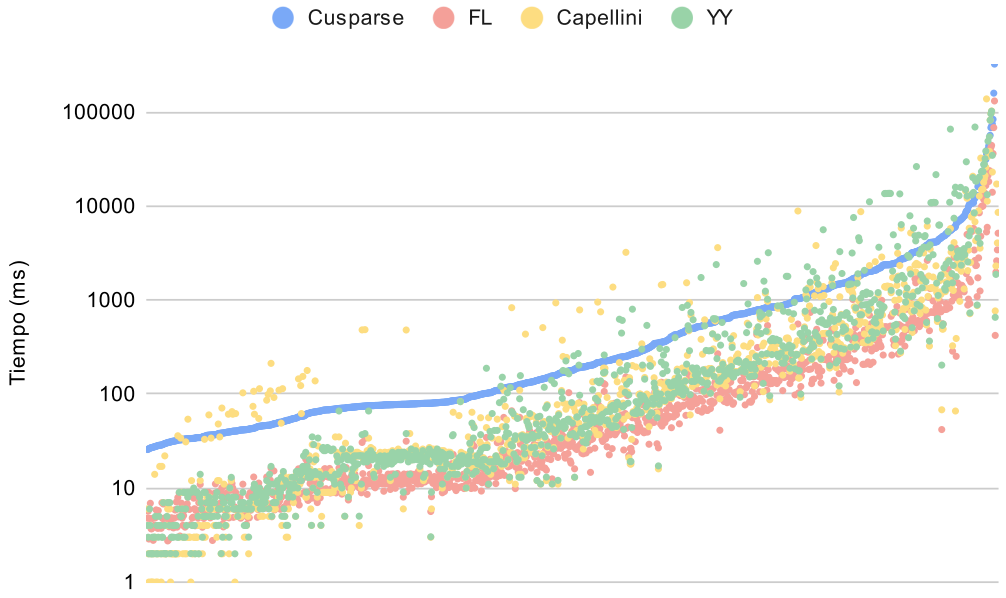


Figura 4.13: Comparación del tiempo de ejecución de 100 iteraciones de la etapa de resolución más el análisis para $SpTRSV_{FL}$ (FL) y otras rutinas disponibles públicamente: CUSPARSE, CAPELLINI y YUENYEUNG (YY). Cada punto es una matriz. Las matrices se encuentran ordenadas en función del tiempo de ejecución de CUSPARSE.

presentados en la Figura 4.12, se sabe que existe un conjunto de matrices en las que $SpTRSV_{FL}$ requiere una gran cantidad de iteraciones para justificar su análisis.

A diferencia del estudio anterior, en este los resultados no son masivamente favorables a las rutinas propuestas. De las 1156 en las que las cinco rutinas completan la ejecución de manera satisfactoria, en 541 la mejor es una de las dos nuevas propuestas y en 615 es CAPELLINI o YUENYEUNG. Sin embargo, las nuevas propuestas obtienen mejores resultados en las matrices que requieren mayor tiempo de ejecución. Adicionalmente, solamente hay dos casos en las que alguna de las dos rutinas propuestas ($SpTRSV_{Fmt}$ y $SpTRSV_{FL}$) es superada por CUSPARSE, algo que sucede para CAPELLINI y YUENYEUNG en un mayor conjunto de matrices.

Finalmente, se repite el análisis con foco en la etapa de resolución. Los resultados presentados en la Figura 4.15 son similares a los de la Figura 4.13, pero el número de matrices en las que la rutina $SpTRSV_{FL}$ es la mejor aumenta a 1000. Adicionalmente, también aumenta la cantidad de matrices en las que las rutinas CAPELLINI y YUENYEUNG son superadas por CUSPARSE.

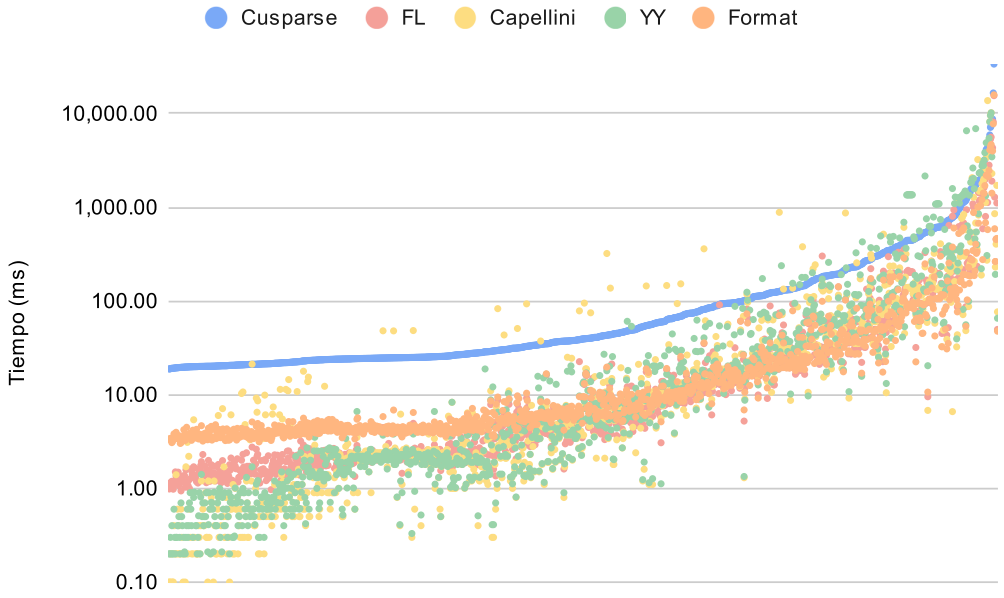


Figura 4.14: Comparación del tiempo de ejecución de 10 iteraciones de la etapa de resolución más el análisis para $SpTRSV_{FL}$ (FL), de la versión $SpTRSV_{Fmt}$ y otras rutinas disponibles públicamente: CUSPARSE, CAPELLINI y YUENYEUNG (YY). Cada punto es una matriz. Las matrices se encuentran ordenadas en función del tiempo de ejecución de CUSPARSE.

Para mostrar los resultados de manera más clara, en la Tabla 4.1 se presenta un resumen de los tres escenarios. Esta tabla presenta la media, rango intercuartil (IQR por sus siglas en inglés), máximo y mínimo *speedup* contra CUSPARSE para 100, 10 y 1 iteración. Los valores en gris oscuro y claro representan la mejor y segunda mejor rutina respectivamente. Tanto en el escenario de gran cantidad de iteraciones (100 más análisis) como cuando se considera solamente la etapa de resolución, las dos rutinas superan a YUENYEUNG y CAPELLINI. Por el contrario, si se considera el escenario con pocas iteraciones (10 más análisis), se obtienen resultados mixtos con $SpTRSV_{FL}$ y $SpTRSV_{Fmt}$. En concreto, las rutinas propuestas, mejoran a YUENYEUNG y CAPELLINI en los peores casos (cuando se obtienen peores *speedup*s contra CUSPARSE) pero obtienen peores resultados máximos. Además, las rutinas propuestas tienen una menor desviación estándar en los valores de *speedup* lo cual sugiere que se obtienen resultados más estables. Este último escenario es expandido presentando el diagrama de caja (boxplot) en la Figura 4.16.

Los resultados presentados en la Figura 4.16 muestran una imagen similar a los de la Tabla 4.1, en la que las nuevas rutinas se concentran en un rango

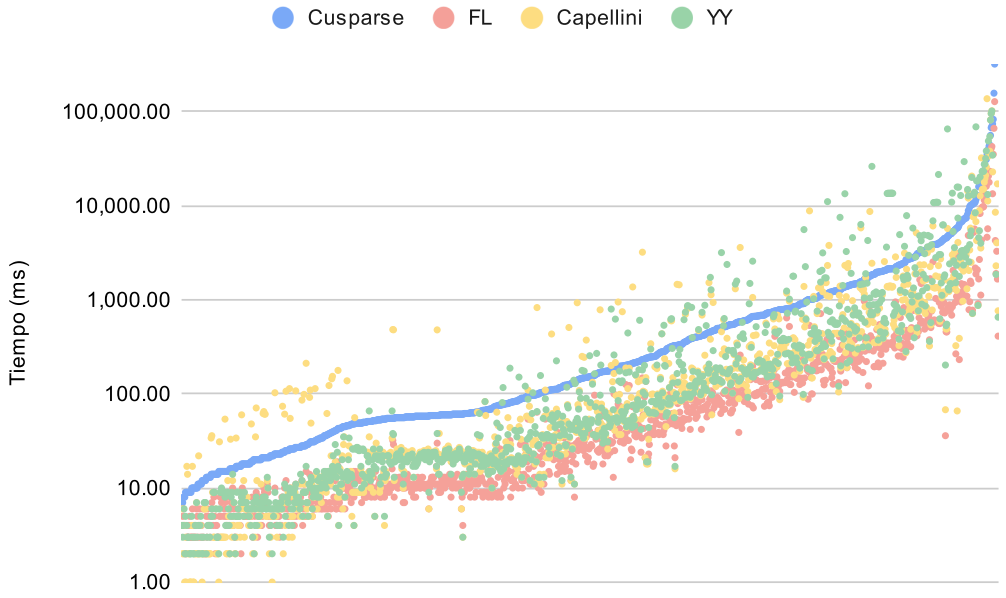


Figura 4.15: Comparación del tiempo de ejecución de la etapa de resolución de $SpTRSV_{FL}$ (FL) y otras rutinas disponibles públicamente: CUSPARSE, CAPELLINI y YUENYEUNG (YY). Cada punto es una matriz. Las matrices se encuentran ordenadas en función del tiempo de ejecución de CUSPARSE.

		YY	Cap	FL	Fmt
1	**Min**	0.07	0.07	1.25	1.14
	Median$_{IQR}$	$2.88_{1.76}$	$3.03_{1.99}$	$5.33_{2.29}$	$5.37_{2.26}$
	Max	22.41	88.5	126.36	110.95
10	**Min**	0.19	0.13	0.6	1.33
	Median$_{IQR}$	$8.25_{16.75}$	$8.07_{16.01}$	$8.31_{11.63}$	$5.75_{6.37}$
	Max	102.8	205.6	57.95	51.65
100	**Min**	0.08	0.08	1.25	1.26
	Median$_{IQR}$	$3.67_{2.65}$	$3.64_{2.52}$	$6.06_{2.09}$	$5.54_{1.87}$
	Max	26.36	91.36	110.84	97.65

Tabla 4.1: Resumen de los valores de *speedup* contra CUSPARSE de las 4 rutinas.

más pequeño. Por el contrario, las otras dos rutinas tienen mayor varianza. Además, tanto en CAPELLINI como en YUENYEUNG se encuentran más valores por fuera del extremo derecho (es decir, más de $Q3 \times 1.5 \times IQR$) y con los máximos valores de *speedup* superando las tres figuras. Sin embargo, es importante notar que los valores de *speedup* son agnósticos del tiempo total de ejecución y, como se explicó anteriormente, las rutinas propuestas superan a YUENYEUNG y CAPELLINI en las matrices que requieren más tiempo de ejecu-

ción. En este sentido, se complementa este análisis para considerar el tiempo de ejecución.

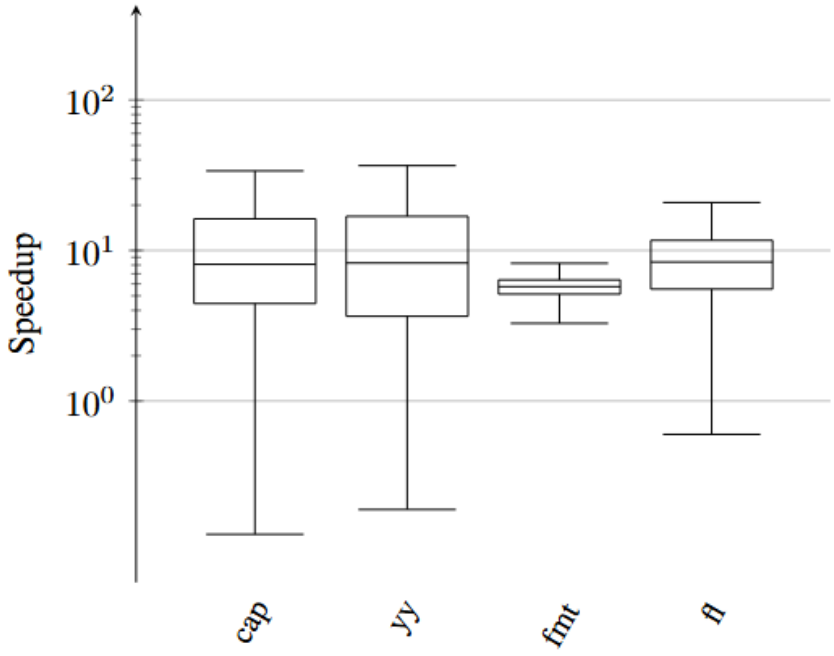


Figura 4.16: Boxplots de las distribuciones del valor de *speedup* en comparación con CUSPARSE para las rutinas CAPELLINI, YUENYEUNG, $SpTRSV_{Fmt}$ y $SpTRSV_{FL}$.

	Total time (ms)	**Speedup**
Cusp	296,328.80	1.00
Fmt	94,367.60	3.14
FL	111,899.20	2.65
Cap	122,792.00	2.41
YY	172,731.61	1.72

Tabla 4.2: Tiempo de ejecución agregado de cada rutina para todas las matrices y *speedup* contra CUSPARSE de este tiempo en el escenario de 10 iteraciones más análisis.

En la Tabla 4.2 se compara el tiempo global de las rutinas para el escenario de 10 iteraciones y análisis (y *speedup* contra CUSPARSE) sobre todo el conjunto de matrices. Los resultados muestran que $SpTRSV_{Fmt}$ pasa a ser la mejor rutina seguida por $SpTRSV_{FL}$. Esto es razonable, ya que en un escenario con tan pocas iteraciones es esperable que en muchos casos no sea posible compensar el sobrecosto del análisis. La misma relación (aunque con peores valores) se da entre CAPELLINI y YUENYEUNG en las que la ventaja de la primera puede ser explicada por no tener etapa de análisis.

4.4. Comentarios finales

En este capítulo se presentaron dos grandes modificaciones a la rutina de resolución de sistemas triangulares *sync-free*. En concreto, se propuso una rutina que utiliza un nuevo formato de almacenamiento para la matriz ($HYB_{syncfree}$) que mejora el acceso a memoria con un bajo sobrecosto de cómputo para generarlo a partir de CSR. Adicionalmente, se propuso un nuevo esquema de niveles basado en el modelo de planificación ALAP. Ambas estrategias son combinadas en un kernel de resolución llamado $SpTRSV_{FL}$.

La evaluación experimental de $HYB_{syncfree}$ muestra que, en general, el formato de almacenamiento propuesto tiene un consumo de memoria alrededor de 2×. Por otro lado, los análisis de tiempo de ejecución permiten afirmar que, con matrices suficientemente grandes, el mejor acceso a memoria logra *speedup*s positivos de hasta 1.77×. Adicionalmente, se observa que en general son necesarias pocas iteraciones (menos de 5) para compensar el sobrecosto de pasar de CSR a $HYB_{syncfree}$.

La evaluación experimental de la nueva estrategia de niveles arroja resultados similares. En concreto, en el 60 % de las matrices se obtienen valores de *speedup* positivos y en un 30 % el *speedup* se ubica entre 0.8 y 1 (es decir, el empeoramiento es acotado). Finalmente, se repitieron los estudios relativos a la cantidad de iteraciones de la etapa de resolución requeridas para compensar el sobrecosto del análisis. En general, se observa que se requieren más iteraciones que en el caso de $HYB_{syncfree}$ si se compara con la línea base. Por otro lado, se ve que aplicar $HYB_{syncfree}$ conjuntamente con la nueva estrategia de niveles obtiene mejores resultados que las dos estrategias por separado requiriendo, en promedio, 6 iteraciones para compensar el sobrecosto del análisis.

Finalmente, se comparó una versión de la $SpTRSV$ que utiliza estas dos variantes ($SpTRSV_{FL}$) con bibliotecas con otras disponibles públicamente. Los resultados muestran que la nueva versión mejora la etapa de resolución (*speedup* contra CUSPARSE de hasta 126.4× y en promedio 5.3×) e incluso cuando se toma en consideración la etapa de análisis (*speedup* contra CUSPARSE de hasta 110.8× y en promedio de 6.1×). Estos resultados superan a las implementaciones YUENYEUNG y CAPELLINI. Sin embargo, las nuevas rutinas no superan claramente a estas dos implementaciones en contextos en los que solamente se ejecutan pocas iteraciones de la etapa de resolución (10 resoluciones por ejecución del análisis).

Capítulo 5

Propuesta para la mejora de la factorización LU incompleta

En este capítulo se explican los principales aportes de esta tesis relativos a la factorización LU incompleta. A diferencia de la resolución de sistemas triangulares dispersos, en el caso de la ILU (o Cholesky en caso de matrices simétricas y definidas positivamente) la evaluación del estado del arte no arroja una gran cantidad de trabajos previos. En concreto, no se encontraron trabajos que exploren la aplicación del paradigma *sync-free* a dicha operación.

Como se dijo anteriormente en la Sección 2.3.4, la factorización LU incompleta tiene una estructura de dependencias similar a la resolución de sistemas triangulares dispersos. Específicamente, si se toma las filas como unidades el esquema de dependencias es el mismo. En la Figura 5.1 se presenta una matriz que comparte su componente triangular inferior y diagonal con la matriz de la Figura 2.8 pero que, a diferencia de esta, tiene elementos por arriba de la diagonal. Por lo mencionado anteriormente, el esquema de dependencias entre filas de esta matriz es el mismo que el presentado anteriormente en la Figura 2.9. De todas formas, se agrega nuevamente en la Figura 5.2 para facilitar la lectura.

El resto del capítulo se estructura de la siguiente forma. En la Sección 5.1 se presentan las distintas rutinas propuestas. Posteriormente, en la Sección 5.2 se evalúan estas rutinas comparando con bibliotecas establecidas como CUSPARSE e INTEL MKL. Finalmente, en la Sección 5.3 se resumen los resultados obtenidos por las distintas implementaciones propuestas y se presentan comentarios finales relativos a los esfuerzos presentados en este capítulo.

$$\begin{bmatrix} 1 & & v & & & & z \\ & 2 & & w & x & & \\ 1 & & 3 & w & & y & \\ 1 & & 3 & 4 & x & & \\ & 2 & & & 5 & y & \\ & 2 & 3 & & & 6 & \\ 1 & & & & 5 & 6 & 7 \end{bmatrix}$$

Figura 5.1: Matriz de ejemplo. El triángulo inferior y la diagonal es idéntico a la Figura 2.8, pero esta ha sido expandida para agregar elementos arriba de la diagonal. Los elementos no mostrados son ceros.

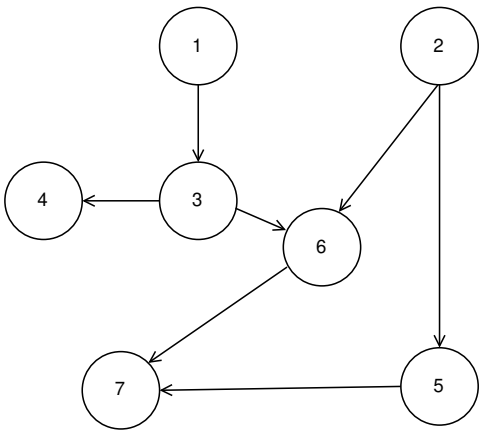


Figura 5.2: Dependencias para la estrategia *sync-free* correspondientes a la matriz de la Figura 5.1. Dependencias redundantes fueron removidas.

5.1. Propuestas

Los algoritmos que se proponen en este capítulo siguen el paradigma *sync-free* que, como se explicita en los capítulos anteriores, es el estado del arte en la resolución de sistemas triangulares. La idea general consiste en dividir los hilos en grupos (o vectores), de forma que cada uno de estos procese un conjunto de elementos (en general una fila). Cada conjunto de elementos es procesado apenas todas sus dependencias han sido computadas. Los elementos procesados por un mismo vector son, en todos los casos, pertenecientes a la misma fila (que se le llama *current* siguiendo la nomenclatura usada por Naumov en [63]). Todas las implementaciones son desarrolladas para matrices en formato CSR.

Cada vector itera sobre los elementos que tiene asignados de izquierda a derecha haciendo espera activa hasta que la fila correspondiente a ese elemento (a la que se llama *ref*) haya sido procesada completamente por otro vector. Una vez que la dependencia ha sido computada, el elemento que está siendo procesado es dividido por la entrada diagonal de la fila *ref* para obtener el nuevo coeficiente de L y todos los elementos de la fila *current* que se encuentran a la derecha en la matriz son impactados por los correspondientes de *ref*, es decir, se le restan a *current* los elementos de U en *ref* multiplicados por el nuevo coeficiente de L. Como no se admite *fill-in*, esto solo ocurre para las columnas donde hay no-ceros tanto en *current* como en *ref*.

El proceso de espera activa consiste en consultar repetitivamente (*polling*) las entradas correspondientes del vector *is_solved* que, cuando tiene un valor válido, almacena el índice de la entrada diagonal (pivot) de la fila en la estructura CSR. Este valor es útil para el procesamiento posterior a la etapa de *busy-waiting* dado que los elementos relevantes para la etapa de actualización son los que se encuentran a la derecha de la diagonal. Además, permite acceder al elemento diagonal directamente, lo que es útil al hacer la división para computar las entradas de L. Es importante notar que la complejidad de encontrar el pivot en la fila es, en el caso general, $O(n)$ siendo n el número de no-ceros de la fila. Si se asumen índices de columna ordenados (algo que sucede en CSR) la complejidad es $O(log(n))$.

La principal diferencia con respecto a la $SpTRSV$ es la parte de actualización de los valores posterior al cumplimiento de la dependencia que es, además, el principal cuello de botella del algoritmo. Por cada no-cero a la izquierda de la diagonal, esta etapa compara los patrones de no-ceros de las filas *current* y *ref* verificando los índices de columna de cada elemento.

En esta tesis se proponen un conjunto de algoritmos para paralelizar la factorización ILU. Todas las variantes tienen una estructura similar que se resume en el Algoritmo 20. Dado que el cuello de botella principal es la fase de actualización, los hilos del vector asignado a esa parte de *current* iteran juntos por los elementos, esperan por las mismas dependencias y calculan la actualización en paralelo una vez que esta se cumple.

Las primeras dos variantes difieren en la estrategia utilizada para hacer la actualización de los valores a la derecha. En la Sección 5.1.1 se utiliza una estrategia para comparar los índices de columna recorriéndolos iterativamente a efectos de optimizar los accesos a memoria global, mientras que en la Sección 5.1.2 se propone una búsqueda por bipartición. Las siguientes dos variantes exploran distintas estrategias para asignar vectores a conjuntos de elementos. En la Sección 5.1.3 se presenta un enfoque con un tamaño de vector variable en función de características de la matriz. Por otro lado, en la Sección 5.1.4 varios vectores de 32 elementos trabajan, colaborativamente, en la misma fila. Dado que gran parte de la etapa de actualización consiste en hallar qué elementos serán afectados, en la Sección 5.1.5 se explora la posibilidad de hacer estos cálculos previamente mientras se está en la etapa de *busy waiting*. Finalmente, en la Sección 5.1.6 se propone una rutina que utiliza un preprocesamiento para acelerar la resolución.

Algoritmo 18 Estructura general de las distintas variantes de ILU propuestas

```
// IN/OUT: row_ptr[], col_idx[], val[], is_solved

init()
row = getRowNumber(vect)
piv = calculatePiv(row,row_ptr[])
li = row_ptr[row]
while li < piv do:
    ref_row=col_idx[li];
    ready = is_solved[ref_row]
    if(ready != 0) //Si no es 0 tiene el índice de piv[ref_row] +1
        if(vect.rank()==0)
            val[li]/=val[ready-1]
        endif
        vect.sync()
    endif
    update_row(curr_row, ref_row, li, ready-1, row_ptr[], col_idx[],
        val[]) //Se calcula la intersección del patrón de no-ceros y
        se acumulan los valores que corresponden
    li++
    ready = 0
endwhile
if vect.rank()==0
    is_solved[curr_row] = piv+1
endif
```

5.1.1. Actualización vía intersección de los vectores en memoria compartida

La primera variante, a la que se llama ILU_{shared}, usa vectores que ocupan un *warp* completo y procesan una sola fila de manera completa. La etapa de actualización se hace en memoria compartida computando la intersección entre los índices de columna de los no-ceros entre la fila *current* y *ref* adaptando las ideas propuestas en [96]. Los hilos del vector recorren la fila *current* en tramos de tamaño igual a la cantidad de hilos. Para cada tramo, cada hilo guarda un elemento de *current* en un registro. Luego los hilos del vector cargan en memoria compartida los índices de columna (y valores) de un segmento de la fila *ref*. Por último, cada hilo compara el segmento de *ref* en memoria compartida con el valor de *current* almacenado en registro. Luego, los elementos que lo requieren son actualizados y, posteriormente, se mueve el vector que corresponde según los índices de columna de las últimas entradas de cada uno.

El Algoritmo 19 presenta una versión simplificada de esta estrategia para la etapa de actualización.

Algoritmo 19 Intersección de dos vectores ordenados que representan los patrones de no-ceros de las dos filas utilizando memoria compartida. El Vector a es almacenado en registros mientras que el b es almacenado en memoria compartida.

```
//In: i_a, i_b, l_a, l_b -> Inicio y fin de vectores a intersectar
//In: piv, col_idx[], val[] -> Pivote y vectores de la matriz csr
//In idx_sh -> buffer en memoria compartida

lne threadIdx.x % 32
while i_a < l_a and i_b < l_b do:

  // 1. Cargar los vectores (32)
  my_idx    =(i_a+lne < l_a)?col_idx[i_a+lne]:-1;
  idx_sh[lne]=(i_b+lne < l_b)?col_idx[i_b+lne]:-1;
  __syncwarp();

  // 2. Comparación entre todos
  for i=1:WARPSIZE do:
    if ( my_idx == idx_sh[j] )
      // hacer update y cortar iteración
    endif
  endfor

  // 3. Se actualiza el vector a la siguiente iteración
  my_idx = __shfl_sync(my_idx, 31) //Trasmite el último índice de a
  if my_idx < idx_sh[WARP_SIZE] or vect_b_termino
    i_a += WARP_SIZE
  else if vec to r_b_no_termino
    i_b += WARP_SIZE
  endif

endwhile
```

5.1.2. Actualización vía búsqueda binaria

Dado que ILU_{shared} compara, en el peor caso, cada no-cero de la fila *current* con cada elemento de la fila *ref*, la complejidad computacional de esta rutina es $O(nnz_{curr} \times nnz_{ref})$. Esto puede tener un costo prohibitivo en filas con elevado número de no-ceros. En efecto, esta estrategia tiene malos resultados

en matrices que tienen muchas filas con estas características. En esta línea, se propone una nueva estrategia de actualización enfocada en este tipo de filas.

Esta nueva estrategia (ILU_{vect}) mantiene de la anterior tanto el tamaño de los vectores (`WARP_SIZE`) como la política de asignación a filas (cada fila es procesada por un solo vector). Sin embargo, en lugar de trabajar en conjunto para computar la intersección entre los patrones de dispersión de las dos filas, los hilos trabajan de manera independiente. Cada hilo toma un elemento de la fila *current* y busca su índice de columna en la fila *ref* haciendo una búsqueda por bipartición. Aunque esta estrategia reduce la cantidad de elementos evaluados, también empeora la localidad de datos ya que los hilos divergen y las lecturas no son alineadas. Esta estrategia solamente tiene sentido cuando el tamaño de la fila (y por ende la cantidad de accesos) compensa este fenómeno. Por lo anterior se decidió mantener la estrategia de acumulación de ILU_{shared} para las filas pequeñas. El límite entre una y otra estrategia se definió en 128 no-ceros en función de los resultados de pruebas primarias.

5.1.3. Tamaño de vectores adaptativo

Las versiones anteriores utilizan vectores del mismo tamaño que un *warp*. Esto genera que, cuando las filas tienen menos de 32 no-ceros algunos hilos estén ociosos y se pierda poder de cómputo. Esta problemática es especialmente preocupante en matrices que poseen muchas filas con muy pocos no-ceros (significativamente menos que 32). Utilizar un tamaño de vector más pequeño solucionaría (o reduciría) este problema pero como contrapartida no se podría explorar el paralelismo en matrices con filas grandes.

En esta línea se propone una variante (ILU_{adapt}) que busca balancear estas dos necesidades contrapuestas utilizando un enfoque adaptativo que define el tamaño de los vectores dinámicamente para cada matriz. A estos efectos se utiliza como heurística la cantidad promedio de no-ceros por fila.

Es evidente que esta heurística tendrá malos resultados en matrices donde la diferencia entre los largos de las filas sea muy grande (es decir si la desviación estándar es grande). El caso paradigmático sería una matriz en la que la mayoría de las filas tienen muy pocos elementos y unas pocas filas concentran la mayoría de los no-ceros, generando que el promedio crezca muy por arriba de la mediana. Sin embargo, esta característica de la matriz tiene un costo despreciable de computar ya que puede ser obtenida en $O(1)$ mientras se lee

Algoritmo 20 Intersección de dos vectores ordenados que representan los patrones de no-ceros de las dos filas utilizando una búsqueda binaria.

```
1  //In: i_a, int i_b, int l_a, int l_b -> Inicio y fin de vectores a
       intersectar
2  int * col_idx, volatile VALUE_TYPE * val // CSR sparse matrix
3  int lne // thread lane
4
5  for (int k = i_a+lne; k < l_a; k+=WARP_SIZE){
6
7    int curr_col_idx=col_idx[k];
8    beg=i_b+1; end=l_b-1;
9
10   while(beg<=end)
11
12     int med=(end+beg)>>1;
13     int col_idx_med=col_idx[med];
14
15     if(col_idx_med==curr_col_idx)
16       // perform the update and exit
17     else if(curr_col_idx > col_idx_med)
18       beg = med+1;
19     else
20       end = med-1;
21     end if
22
23   end while
```

la matriz, lo que la vuelve atractiva.

Para evitar que los vectores involucren a hilos de *warps* distintos se definen los tamaños de vector 2^p con $1 < p \leq 5$. La sincronización y comunicación entre hilos del mismo vector se hace definiendo *cooperative groups* [97] para cada uno de estos.

5.1.4. Procesamiento paralelo de la fila

Las estrategias anteriores fuerzan a que todo el vector procese a la vez el mismo elemento de L a efectos de paralelizar la etapa de actualización. Esta estrategia ($ILU_{parallel}$) busca paralelizar el procesamiento de los elementos de L dentro de la fila.

A diferencia de $SpTRSV$, los elementos de una misma fila no son todos independientes, sino que, como se explicó anteriormente, existen dependencias

en función de las filas *ref* respectivas. En concreto, para $k \leq j \leq i$ un elemento A_{ij} depende de otro A_{ik} si A_{kj} es distinto a 0. Por ejemplo, si se toma como referencia la fila 7 de la matriz de la Figura 5.1 puede verse que el elemento de la columna 7 depende del de la columna 1 porque en A_{17} se tiene valor z. Lo mismo sucede entre los elementos de las columnas 5 y 6. La matriz simplificada, manteniendo únicamente lo relevante para las dependencias, se muestra en la Figura 5.3.

$$\begin{bmatrix} 1 & \cdots & \cdots & \cdots & \cdots & \cdots & z \\ \vdots & & & & & & \vdots \\ \vdots & & & & & & \vdots \\ \vdots & & & & & & \vdots \\ \vdots & & & & 5 & \cdots y & \vdots \\ \vdots & & & & \vdots & \vdots & \vdots \\ 1 & & & & 5 & 6 & 7 \end{bmatrix}$$

Figura 5.3: Dependencias entre los elementos de la fila 7 de la matriz de la Figura 5.1. Se mantienen solamente los elementos de otras filas que son relevantes para las dependencias

Utilizando este esquema de dependencias se puede paralelizar la resolución de los elementos de la fila. En este sentido, la implementación de $ILU_{parallel}$ asigna un bloque a procesar cada fila. Este bloque es subdividido en vectores que procesan independientemente los elementos. Similarmente a los casos anteriores, si la cantidad de vectores es menor a la cantidad de elementos, cada vector procesa más de uno. Dado que no se tiene información a priori sobre las dependencias entre los elementos, los vectores van procesando alternativamente los mismos y, en caso de depender de algún elemento anterior procesado por otro vector, hacen espera activa por este. El tamaño de los vectores es configurable, dando lugar a distintas configuraciones con más o menos vectores.

5.1.5. Estrategia de *prefetch*

Las estrategias presentadas anteriormente tienen como objetivo acelerar el cómputo de la etapa de actualización ya que es la parte más costosa de la rutina. Un estudio en mayor profundidad de esta etapa muestra que el principal costo proviene del análisis de qué elementos de la fila *current* tienen

correspondencia en *ref* y no del cómputo mismo de las operaciones en punto flotante.

El cálculo de qué elementos de *current* serán afectados en cada iteración depende únicamente del patrón de no-ceros de las dos filas y no de sus valores numéricos. Por lo tanto, aunque un vector no tenga los datos para calcular el valor numérico final de la fila (las dependencias no están listas) sí podrá calcular cuáles de los elementos de la fila serán alterados. En esta línea, el tiempo dedicado a responder, para cada elemento, si será afectado o no, puede ser ocultado parcialmente al hacerlo en el tiempo en que se espera por la dependencia. Esta estrategia se conoce como *prefetch*.

Esta estrategia es, en general, independiente de cuál de las variantes es utilizada. Por lo anterior, no se desarrolló una variante independiente, sino que se probó aplicar la estrategia en las rutinas más prometedoras.

5.1.6. Estrategia de preprocesamiento

Las estrategias anteriores se basan en dos grandes líneas: mejorar la etapa de actualización y modificar la asignación de hilos a filas. Esta estrategia utiliza un enfoque distinto y busca modificar el orden en el que las filas son procesadas siendo completamente agnóstico de qué estrategia será usada para computar las filas.

La observación que motiva esta variante (que llamamos ILU_{levs}) es que, en matrices suficientemente grandes, despachar las filas en el orden numérico puede no ser óptimo. Al igual que sucede en el caso del $SpTRSV$, hay matrices que por sus patrones de no-ceros generan que existan filas con índices altos que no dependen de ninguna otra mientras que las primeras filas deben ejecutarse cuasi serialmente. En este sentido, el orden de ejecución tradicional generaría que mientras algunas filas que están en condiciones de ejecutar estén asignadas a bloques inactivos otras, con índices más bajos y asignadas a bloques activos, harían espera activa durante mucho tiempo.

Para ejecutar las filas de manera más eficiente se genera una estructura de niveles tipo *level-set* y posteriormente se ordenan las filas de menor a mayor nivel manteniendo el ordenamiento parcial entre filas del mismo nivel. Es importante recordar que las filas de nivel i solamente dependen de filas de niveles j con $j < i$. Este patrón de cómputo garantiza que las filas ejecutan luego de todas sus dependencias pero "no demasiado después" (como mucho al

final del nivel siguiente) mientras que las filas con muchas dependencias son postergadas.

Para calcular los niveles de cada fila es necesario hacer un preprocesamiento. Al igual que en [73] se utiliza un enfoque *sync-free* para generar la estructura de niveles, en la que a cada fila se le asigna como nivel el máximo nivel de todas sus dependencias más uno.

La etapa de preprocesamiento es costosa y, a diferencia de lo que sucede con la $SpTRSV$, generalmente no se cuenta con sucesivas ejecuciones para compensar el costo. En esta línea, para justificar el preprocesamiento, la reducción de tiempo de ejecución en la ejecución de la ILU debería ser mayor al costo de este. Por lo anterior, este enfoque solamente tendrá sentido en matrices muy grandes y que, al comienzo de la ejecución, tendrán muchos bloques inactivos y en las que, además, el orden de ejecución estándar esté suficientemente lejos del óptimo.

Por otro lado, si bien a nivel de ILU solamente se hace una ejecución de la etapa de resolución y el costo del preprocesamiento de niveles tendría que ser compensado por esta, los contextos de aplicación de esta rutina implican, posteriormente, la resolución de los sistemas triangulares generados por L y U. Dado que la ILU no modifica el patrón de no-ceros de la matriz, la estructura de dependencias de L es idéntica a la de la matriz original permitiendo reutilizar el análisis. En este sentido, es importante matizar lo comentado en el párrafo anterior, ya que el costo del preprocesamiento se podría compensar globalmente en la ejecución de un método numérico que utilice a la ILU como precondicionador.

5.2. Evaluación experimental

Esta sección presenta los principales resultados de la evaluación experimental de las distintas alternativas propuestas en la Sección 5.1.

5.2.1. Rutinas básicas

En este apartado se presentan los resultados de las tres rutinas que procesan las filas con un único vector: ILU_{shared}, ILU_{vect} e ILU_{adapt}. En esta línea, en la Figura 5.4 se presentan los *speedup* de estas rutinas relativos a la peor de estas. El *speedup* se presenta como logarítmico a efectos de permitir el análisis

de los valores más pequeños. Es decir, un valor de 0 implica un *speedup* de 1 y, por lo tanto, la rutina en cuestión es la peor.

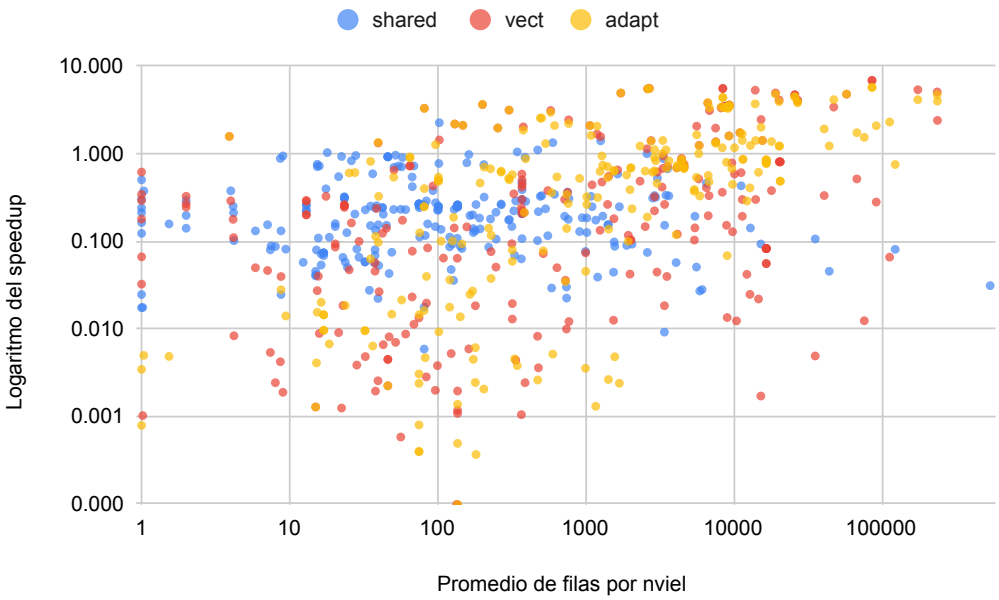


Figura 5.4: Valor de *speedup* de ILU_{shared}, ILU_{vect} e ILU_{adapt} con respecto a la peor de ellas en función del promedio de la cantidad de filas por nivel.

Los resultados muestran una clara relación entre los comportamientos de las distintas rutinas y la cantidad de filas promedio por nivel. En concreto, ILU_{shared} tiene resultados positivos cuando no hay demasiado paralelismo disponible (pocas filas por nivel, lo que equivale a pocas filas siendo procesadas a la vez). Por otro lado, ILU_{vect} e ILU_{adapt} son superiores cuando aumenta el promedio de filas que pueden ejecutarse a la vez. Esto es particularmente notable si se observa ILU_{adapt} cuyos resultados son especialmente malos cuando el promedio de filas por nivel se encuentra entre 1 y 10. Esto se debe a que la ventaja de ILU_{adapt} sobre las otras rutinas es que, en matrices determinadas por filas con pocos no-ceros, emplea menos hilos por fila, lo que le permite ejecutar más filas en paralelo. En casos en los que hay pocas filas por nivel no es posible ejecutar muchas filas en paralelo. Por esta razón, el empeoramiento del rendimiento individual en cada fila al asignar menos hilos no es compensado por un mayor paralelismo entre las filas. Las rutinas ILU_{vect} e ILU_{adapt} tienen comportamientos similares y, en general, las diferencias son menos notables que al compararlas con ILU_{shared}. Sin embargo, la segunda rutina es generalmente mejor, teniendo un promedio de *speedup* en el entorno de 1.16×.

5.2.2. Procesamiento paralelo de las filas

En este apartado se evalúa en profundidad la rutina $ILU_{parallel}$. A estos efectos, se ejecutaron sobre el conjunto de matrices de evaluación 6 variantes que difieren en el tamaño de vector utilizado. En la Figura 5.5 se presenta el *speedup* de las distintas versiones contra la peor de ellas.

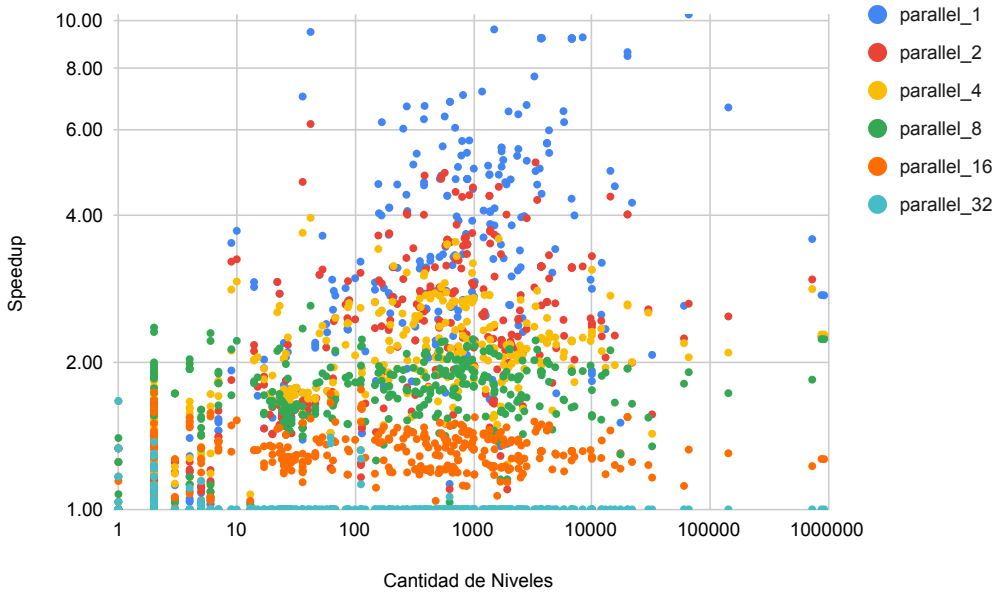


Figura 5.5: Valor de *speedup* de las distintas variantes de $ILU_{parallel}$ comparadas contra la que peores resultados obtiene en función de la cantidad de niveles de la matriz.

Puede verse que, en general, las versiones que utilizan vectores con menor cantidad de hilos (1, 2, 4) superan a las que emplean vectores de tamaños mayores (8, 16, 32). En concreto, las dos versiones con más cantidad de hilos no obtienen los mejores resultados en casi ninguna matriz evaluada. El caso de 32 hilos es, en la mayoría de las matrices, la peor rutina. Por el contrario, si se hace foco en las tres primeras rutinas se puede ver que obtienen mejores resultados a medida que aumenta la cantidad de niveles. Es importante destacar que un mayor *speedup* solamente marca una diferencia mayor entre una rutina y la peor en dicha matriz, no un mejor resultado absoluto. En esta línea, es razonable que las rutinas con menos hilos por elemento obtengan mejores resultados relativos al reducirse el paralelismo disponible (aumentar cantidad de niveles) ya que se reduce la cantidad de elementos que pueden procesarse a la vez en una fila.

Para profundizar en la comparación, en la Tabla 5.1 se muestran la cantidad

de matrices en que cada rutina es la mejor, el promedio de *speedup* y la mediana de éste.

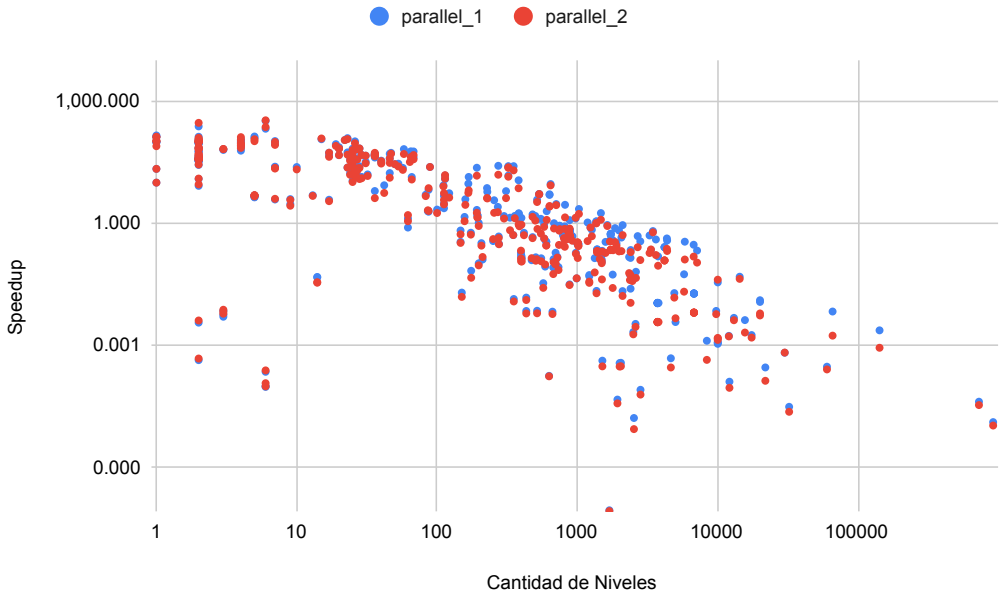
	Tamaño de vectores en $ILU_{parallel}$					
	1	**2**	**4**	**8**	**16**	**32**
Cantidad	248	37	53	69	6	3
Promedio	3.15	2.23	2.02	1.75	1.32	1.02
Mediana	2.58	2.13	1.98	1.76	1.32	1.00

Tabla 5.1: Cantidad de matrices, promedio y mediana de *speedup* de las distintas variables de $ILU_{parallel}$.

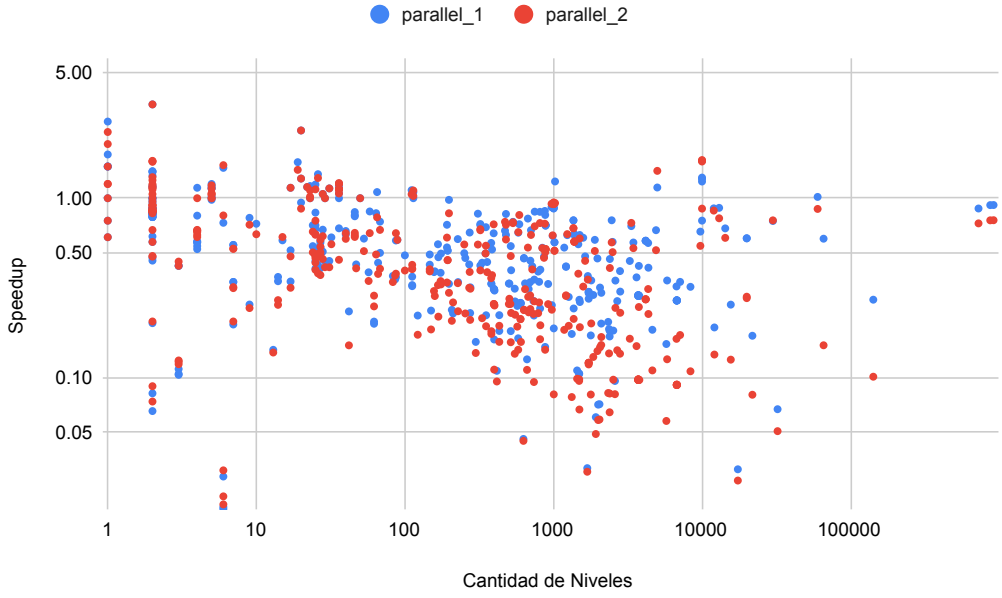
Los resultados presentados en la Tabla 5.1 muestran que la rutina que utiliza un único hilo supera claramente al resto. En un segundo conjunto se encuentran las tres siguientes versiones (2, 4 y 8) para las que, progresivamente, cada una obtiene los mejores resultados en más matrices, mientras que sus valores de *speedup* son peores. En esta línea, es razonable suponer que estas últimas logran ser las mejores en un conjunto de casos en que todas las versiones son similares. Por lo anterior, para el resto del análisis se toman las rutinas con vectores de 1 y 2 hilos. En la Figura 5.6 se comparan los resultados de $ILU_{parallel}$ contra ILU_{adapt} e ILU_{shared}.

Los resultados mostrados en la Figura 5.6 muestran que las variantes $ILU_{parallel}$ tienden a obtener peores resultados que ILU_{shared} e ILU_{adapt} cuando el número de niveles es suficientemente grande. Si bien el comportamiento es similar al comparar contra ILU_{shared} e ILU_{adapt}, es más pronunciado en el primer caso, donde el rango entre mejores y peores valores de *speedup* es más amplio. Es importante destacar que los mayores valores de *speedup* obtenidos en la Figura 5.6(a) (que llegan a un máximo de 333 en la versión con 2 hilos) son justamente en los casos en que ILU_{shared} es peor en general.

Por el contrario, los resultados de *speedup* contra ILU_{adapt} tienen un rango de variación más moderado ya que el comportamiento de esta última es más similar a $ILU_{parallel}$. De todas formas, ILU_{adapt} supera en general a $ILU_{parallel}$ salvo en casos especiales en los que el paralelismo es casi completo (uno o muy pocos niveles). Estudios en profundidad muestran que el cuello de botella de estas rutinas está dado por la sincronización entre los vectores que procesan distintos elementos dependientes. En esta línea, sería necesario desarrollar mejores estrategias para esta etapa que permitan evaluar si los cambios en dedicar más hilos a la etapa de actualización mejoran los resultados.



(a) Comparación contra ILU_{shared}.



(b) Comparación contra ILU_{adapt}.

Figura 5.6: Valor de *speedup* de las las variantes de $ILU_{parallel}$ contra ILU_{shared} (a) ILU_{adapt} (b) en función de la cantidad de niveles de la matriz.

Hay otra observación relevante sobre $ILU_{parallel}$ no relativa a su comparación con las otras rutinas. Puede verse que mientras hay pocos niveles, la

versión con dos hilos es similar o incluso mejor que la de un hilo, mientras que cuando se reduce el paralelismo, la segunda es claramente superior. Estos resultados ayudan a explicar los obtenidos en la Figura 5.5. Es evidente que los mayores valores de *speedup* relativo que obtenía la versión con 1 hilo sobre el resto se dan en matrices donde las implementaciones obtienen peores resultados. Es decir, la implementación de $ILU_{parallel}$ con 1 hilo logra limitar el deterioro de los resultados pero en general no supera a ILU_{adapt} en más matrices que la alternativa con dos hilos (47 y 49 matrices respectivamente).

5.2.3. Prefetch

En este apartado se presentan los resultados de aplicar una estrategia de *prefetch*. Esta estrategia tiene sentido en contextos en los que se desperdicia mucho tiempo en esperar por las dependencias. En este sentido, se aplicó a las rutinas ILU_{adapt} e ILU_{vect} que son las más afectadas en este tipo de contextos. Por otro lado, por cómo se implementa la sincronización entre vectores de un mismo *warp*, no es posible aplicar el *prefetch* sin modificar sustancialmente la implementación de $ILU_{parallel}$. En la Figura 5.7 se presentan los resultados de *speedup* de la estrategia de *prefetch* en las dos variantes mencionadas.

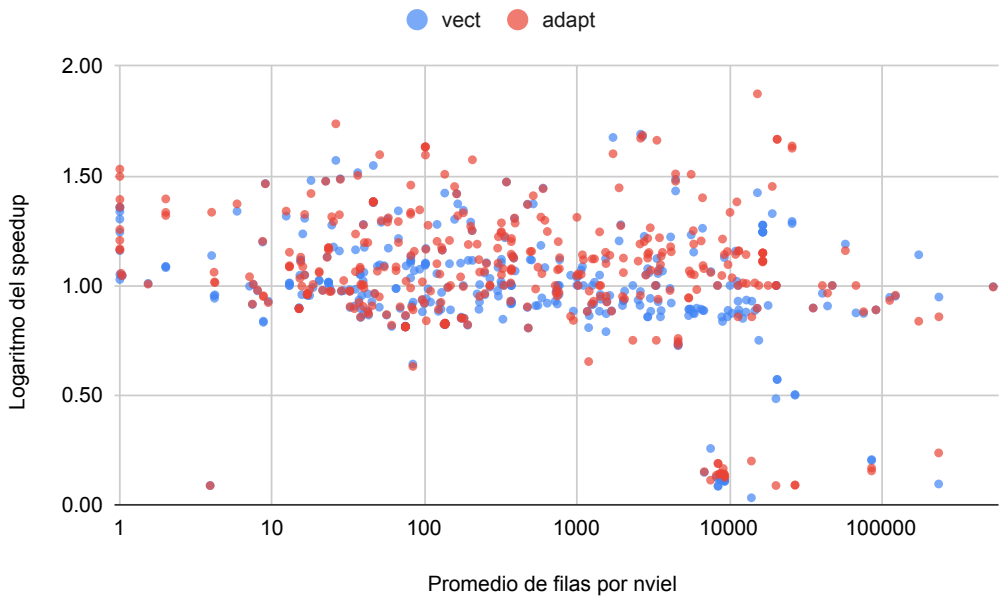


Figura 5.7: Valor de *speedup* de la aplicación de la estrategia de *prefetch* a las variantes ILU_{vect} e ILU_{adapt}.

En la Figura 5.7 puede verse que para los casos en que hay pocas filas por nivel (menos de 10), la alternativa del *prefetch* mejora sustancialmente las rutinas y prácticamente no hay matrices en las que perjudique los tiempos de ejecución. A medida que hay más paralelismo disponible, el uso del *prefetch* es menos eficiente, llegando a casos como la matriz `Chebyshev4` en las que la versión original es 11 veces mejor que cuando se aplica esta estrategia. En general, en los casos en los que hay suficiente paralelismo disponible para explotar razonablemente el poder de cómputo de la GPU es esperable que los resultados sean peores. Finalmente, mientras que en el caso de ILU_{vect} aproximadamente un 40 % de las matrices obtienen resultados positivos al utilizar *prefetch*, en el caso de ILU_{adapt} este porcentaje es un 60 %, y si se extiende el conjunto para permitir hasta un 10 % de degradación, los porcentajes suben a 70 % y 80 % respectivamente.

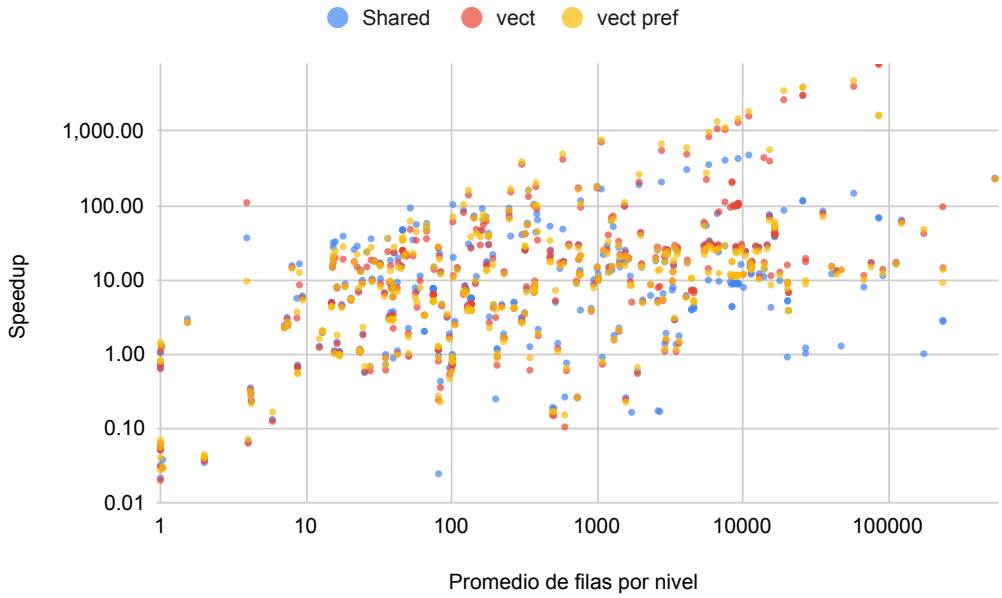
5.2.4. Comparación contra otras bibliotecas

En este apartado se presenta una comparación de las rutinas propuestas (ILU_{shared}, ILU_{vect}, ILU_{adapt} e $ILU_{parallel}$) con dos bibliotecas muy establecidas: Intel MKL y cusparse. La primera es una biblioteca que ejecuta en CPU mientras que la segunda lo hace en GPU.

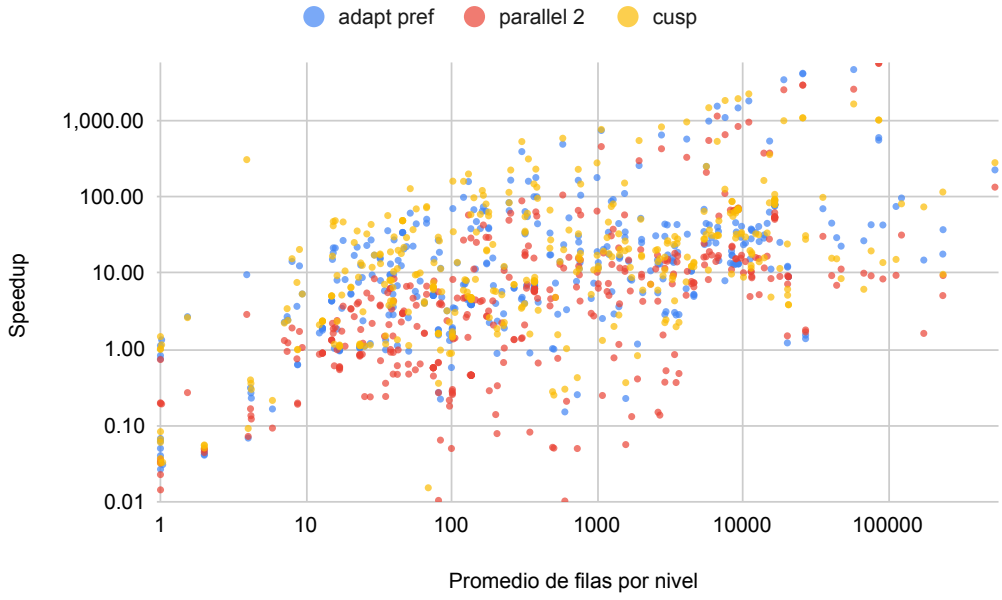
De las rutinas propuestas en secciones anteriores se tomaron las siguientes versiones. Para el caso de ILU_{vect} se utilizan las alternativas con y sin *prefetch* ya que los resultados no son concluyentes hacia uno u otro lado. En el caso de ILU_{adapt} se utiliza únicamente la versión con *prefetch*. Finalmente, en el caso de $ILU_{parallel}$ se presentan los resultados de la versión con dos hilos. Esto se hace porque si bien la versión de un único hilo es mejor que esta en muchas matrices, la versión con 2 hilos la supera en las matrices con pocos niveles que son las únicas en las que estas versiones superan a ILU_{shared} e ILU_{adapt}.

Como Intel MKL presenta resultados significativamente peores que las otras rutinas en la gran mayoría de las matrices, se la utiliza como punto de referencia. En esta línea, en la Figura 5.8 se presentan los resultados de *speedup* de las 6 rutinas mencionadas contra Intel MKL. Debido a la cantidad de rutinas, se dividen en dos grupos para permitir su visualización.

En general, todas las rutinas superan a Intel MKL a excepción de un conjunto pequeño de matrices en las que, en su mayoría, el promedio de filas por nivel es menor a 2. La única excepción a este comportamiento es la rutina



(a) Rutinas ILU_{shared} e ILU_{vect} con y sin *prefetch*.



(b) Rutinas ILU_{adapt}, $ILU_{parallel}$ con dos hilos y CUSPARSE.

Figura 5.8: Resultados de *speedup* contra INTEL MKL de las rutinas ILU_{shared}, ILU_{vect} (con y sin *prefetch*) (a) y las rutinas ILU_{adapt}, $ILU_{parallel}$ con dos hilos y CUSPARSE (b). Todos los valores son presentados en función del promedio de filas por nivel.

$ILU_{parallel}$ en la que, en casi 30 % de las matrices (110), se obtienen valores de *speedup* menores a 1. En el resto de las rutinas los *speedups* menores a 1 no superan las 50 matrices. Adicionalmente, en la Figura 5.8(b) puede verse que no existen prácticamente puntos azules que sean superados por puntos rojos, por lo que puede considerarse que la versión de ILU_{adapt} que utiliza *prefetch* supera a $ILU_{parallel}$. Por lo anterior, en el resto del análisis no se incluye $ILU_{parallel}$.

En la Figura 5.9 se presentan los valores de *speedup* contra CUSPARSE de las otras cuatro rutinas. Los resultados muestran que, en general, la rutina ILU_{adapt} supera al resto de las rutinas propuestas en una gran cantidad de matrices. Sin embargo, la mayoría de los valores de *speedup* (alrededor de 2/3) se encuentran por debajo de 1. Por otro lado, cuando el promedio de filas por nivel es lo suficientemente grande, todas las rutinas propuestas a excepción de ILU_{shared} superan a CUSPARSE. Estos resultados sugieren que, con suficiente paralelismo, las implementaciones *sync-free* pueden superar a CUSPARSE.

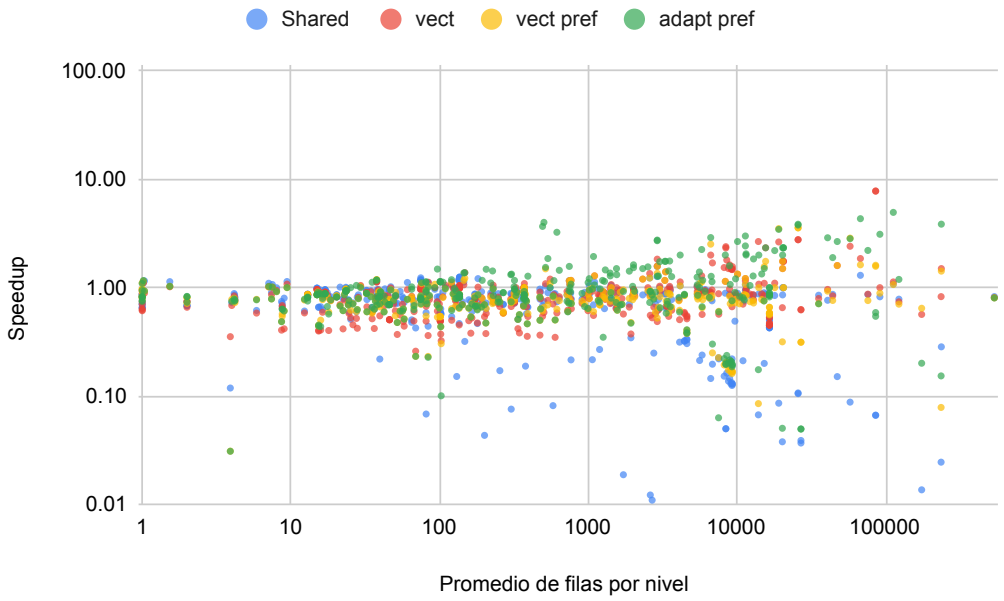


Figura 5.9: Valor de *speedup* de ILU_{shared}, ILU_{vect} (con y sin *prefetch*) e ILU_{adapt} contra CUSPARSE en función del promedio de filas por nivel.

5.2.5. Rutinas con Preprocesamiento

En este apartado se presentan los resultados de la rutina ILU_{levs} y se la compara con CUSPARSE utilizando la bandera `CUSPARSE_SOLVE_POLICY_-`

`USE_LEVEL`. Al igual que en la sección anterior, se compara a estas rutinas con Intel MKL.

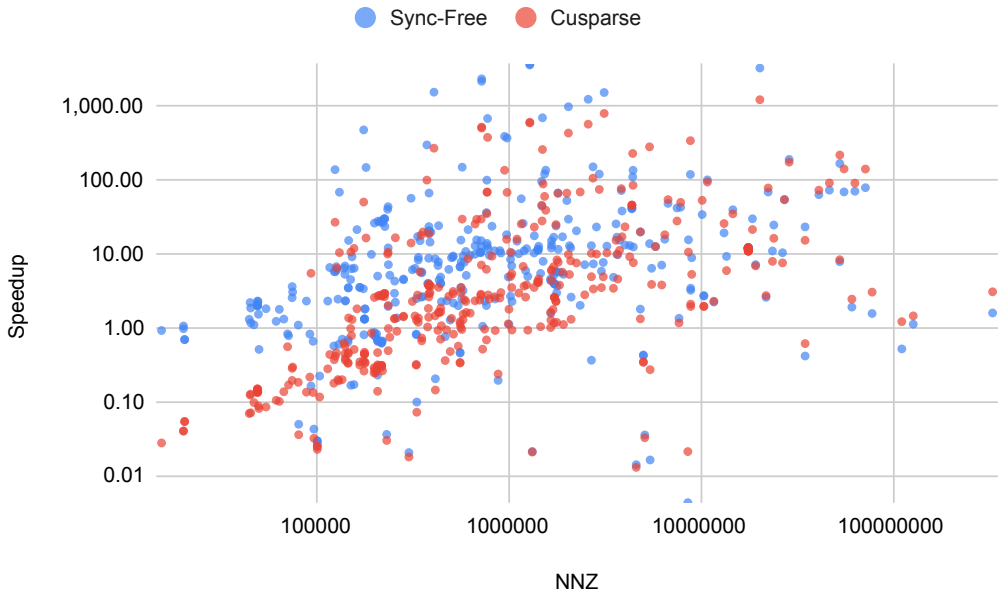
Si bien, como se mencionó en el Apartado 5.1.6 el costo del preprocesamiento podría compensarse al hacer resoluciones triangulares sobre la matriz L resultante, este análisis se centra únicamente en la operación ILU. Por lo anterior, los tiempos de cómputo de estas rutinas se calculan como la suma de las dos etapas. En esta línea, la Figura 5.10 presenta los resultados de *speedup* de las rutinas cusparse e ILU_{levs} contra Intel MKL. Los resultados son presentados en función de dos variables: la cantidad de no-ceros de la matriz y la cantidad de actualizaciones en punto flotante que deben hacerse (a lo que en este trabajo se referencia como colisiones).

Los resultados muestran que las dos implementaciones en GPU superan a Intel MKL en la mayoría de las matrices, en especial en aquellas con el número suficiente de no-ceros. En ambas figuras se ve que ILU_{levs} supera a cusparse, notando que la nube de puntos azules tiene una forma similar a la de puntos rojos pero desplazada hacia arriba. En este sentido, es razonable suponer que las dos implementaciones tienen un comportamiento similar relativo a Intel MKL pero ILU_{levs} tiene una implementación mejor que permite explotar el paralelismo disponible.

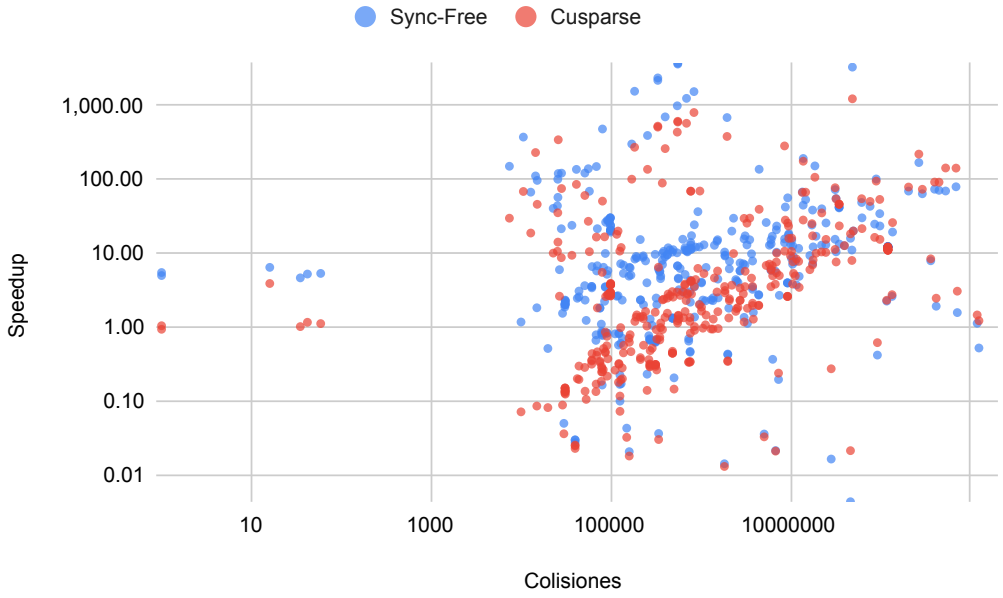
Si se hace foco en la Figura 5.10(a) puede verse que el hecho de que ILU_{levs} supere a cusparse es especialmente significativo en las matrices relativamente pequeñas (menos de 10^6 no-ceros), ya que en estas cusparse es generalmente peor que Intel MKL. Concretamente, en el rango entre 10^5 y 10^6 no-ceros, ILU_{levs} es generalmente superior a Intel MKL mientras que cusparse empeora los resultados al compararse con la versión en CPU. Si se observa la Figura 5.10(b) puede verse que ambas versiones paralelas aumentan su *speedup* relativo a Intel MKL exponencialmente con el número de colisiones. Esto es algo razonable ya que a mayor cómputo, mayor oportunidad de paralelismo. Por otro lado, esta figura muestra cómo, en general, ILU_{levs} supera a cusparse. Para ver más claramente la comparación entre las dos rutinas paralelas, en la Figura 5.11 se presenta el *speedup* de ILU_{levs} contra cusparse.

Los resultados muestran que las mayores mejoras de ILU_{levs} en relación a cusparse se obtienen en los dos primeros cuadrantes (menos de 10^6 no-ceros), se obtienen *speedups* razonables en el tercero (entre 10^6 y 10^7 no-ceros) y se estanca con matrices más grandes, obteniendo *speedups* entre $0.5\times$ y $2\times$.

Aunque la comparación con Intel MKL debe hacerse agrupando los costos



(a) Valor de *speedup* en función de no-ceros (NNZ).



(b) Valor de *speedup* en función de colisiones.

Figura 5.10: Resultados de *speedup* contra INTEL MKL de ILU_{levs} y CUSPARSE expresados en función de no-ceros (a) y colisiones (b).

de las dos etapas, es interesante comparar ILU_{levs} y CUSPARSE separando las dos etapas para evaluar el costo del análisis y cómo mejora la etapa de

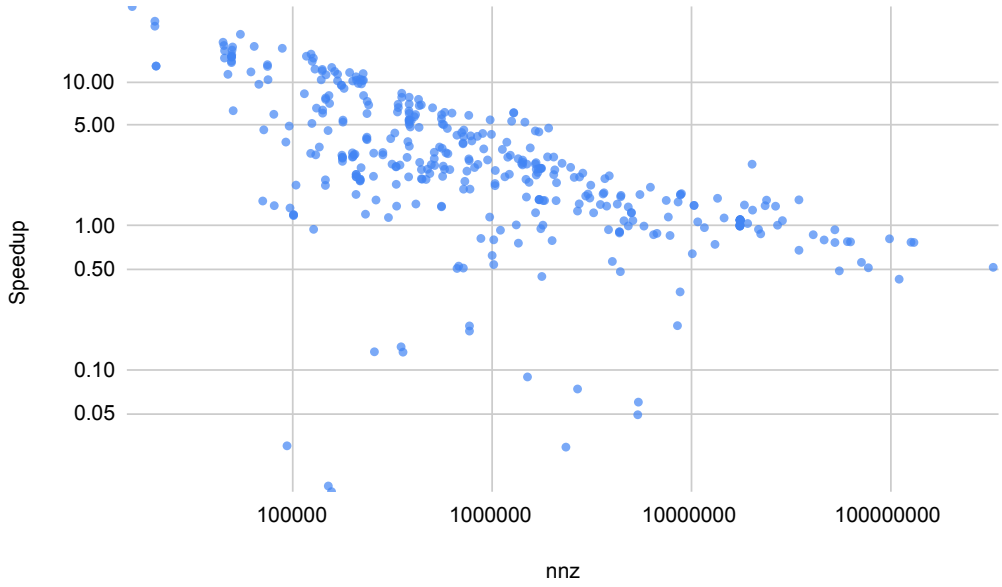


Figura 5.11: Valor de *speedup* de ILU_{levs} contra CUSPARSE utilizando `CUSPARSE_SOLVE_POLICY_USE_LEVEL` en función del número de no-ceros (NNZ).

resolución. Aunque no es público cómo están implementadas las rutinas de CUSPARSE, se supone que genera alguna estructura de niveles que capture las dependencias [64, 95]. Las comparaciones se presentan en la Figura 5.12.

Los resultados muestran que la etapa de análisis de ILU_{levs} supera a la de CUSPARSE en prácticamente todas las matrices. Por otro lado, la etapa de resolución obtiene resultados mixtos. En concreto, a partir de 10^6 no-ceros la implementación de CUSPARSE obtiene mejores resultados en general. En este sentido, se entiende que la etapa de análisis es una contribución en sí misma ya que podría ser complementada (o adaptada para) la resolución de CUSPARSE u otra implementación de la etapa de resolución.

Por último, es interesante estudiar en qué casos es razonable utilizar preprocesamiento y en cuáles no. En la Figura 5.13 se presentan las comparaciones de cada una de las implementaciones contra la versión de CUSPARSE que no requiere preprocesamiento. Los resultados muestran que, aunque para la implementación de CUSPARSE solo es justificable utilizar el preprocesamiento en matrices muy grandes (en algunos casos a partir de 5M de no-ceros pero la mayoría a partir de 20M), ILU_{levs} es competitiva en el rango 10^5-10^7 y supera a CUSPARSE sin preprocesamiento en matrices más grandes.

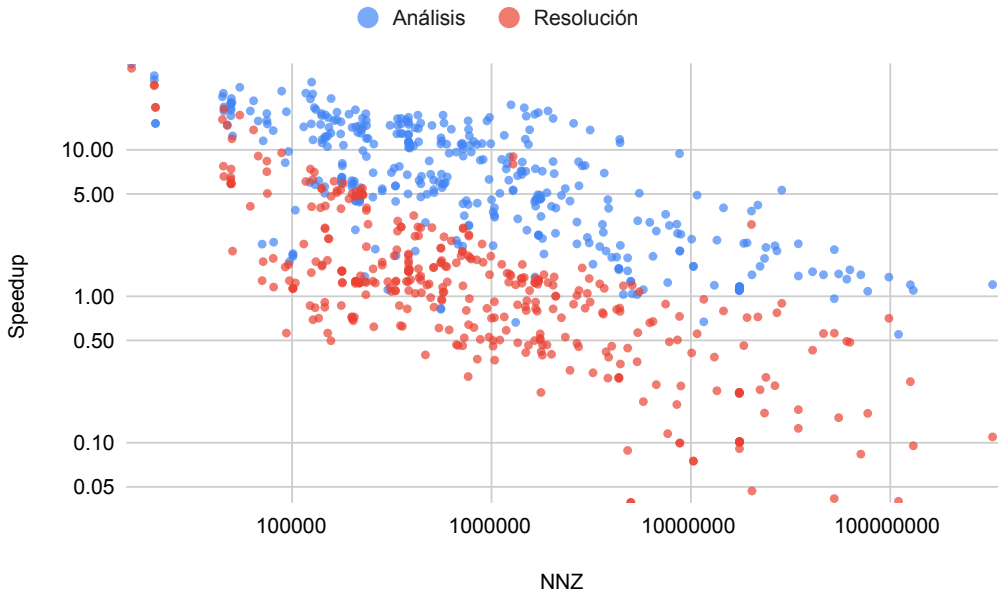


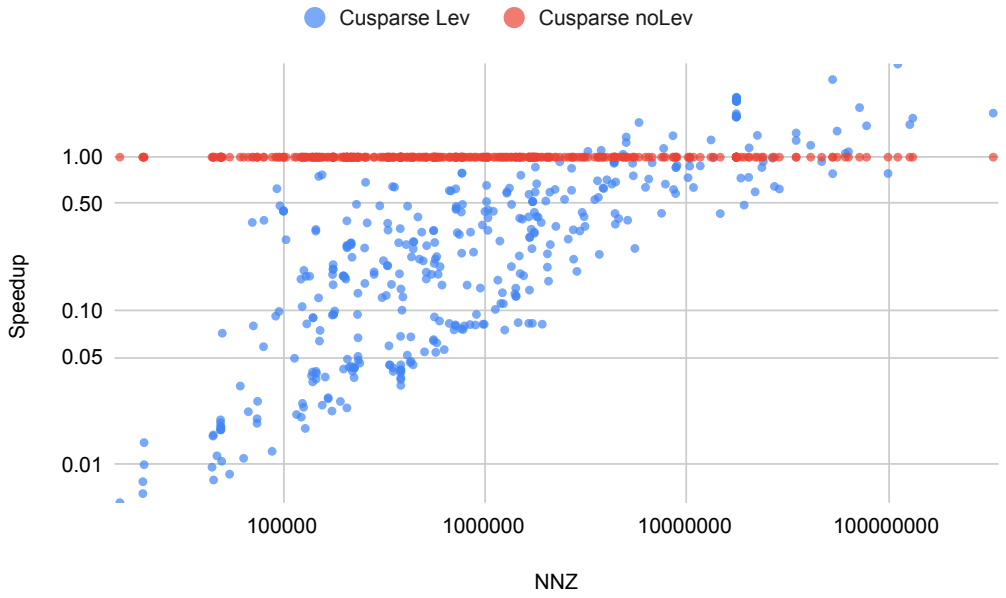
Figura 5.12: Valor de *speedup* de ILU_{levs} contra CUSPARSE utilizando `CUSPARSE_SOLVE_POLICY_USE_LEVEL` dividido en las etapas de análisis (azul) y resolución (rojo) en función del número de no-ceros (NNZ).

5.3. Comentarios finales

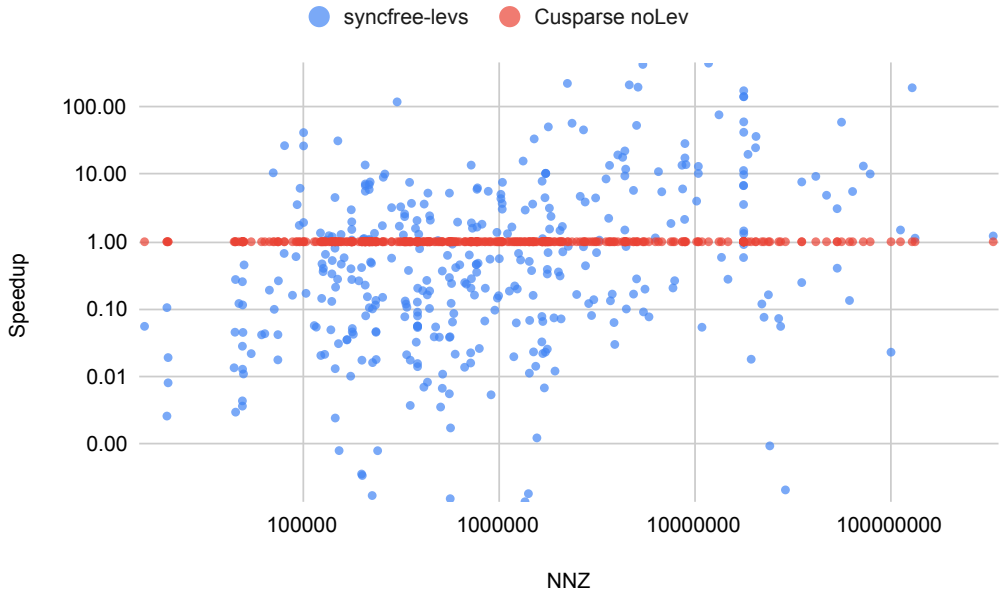
En este capítulo se propusieron distintas alternativas para la implementación de la factorización LU incompleta utilizando el paradigma de sincronización *sync-free*. Análisis sobre implementaciones preliminares mostraron que el principal cuello de botella de las implementaciones *sync-free* de la ILU es la etapa de impactar los resultados de filas precedentes. En esta línea, se propusieron un conjunto de rutinas que varían en su forma de impactar los resultados de las filas precedentes y la asignación de hilos a filas.

Las implementaciones propuestas se compararon con dos bibliotecas establecidas: INTEL MKL y CUSPARSE. Las comparaciones con la primera mostraron que casi todas las implementaciones, con excepción de $ILU_{parallel}$, la superan en más del 80 % de los casos. La comparación de las rutinas propuestas que no usan preprocesamiento muestra que CUSPARSE es la mejor rutina en casi 2/3 de las matrices, siendo ILU_{adapt} la mejor de las restantes. Sin embargo, cuando hay suficientes filas por nivel tanto ILU_{vect} (tanto la alternativa con como sin *prefetch*) superan a CUSPARSE.

Finalmente, también se propuso una rutina que utiliza una estrategia de



(a) CUSPARSE.



(b) *sync-free*.

Figura 5.13: Resultados de *speedup* contra CUSPARSE sin procesamiento de CUSPARSE utilizando `CUSPARSE_SOLVE_POLICY_USE_LEVEL` (arriba) e ILU_{levs} (abajo).

preprocesamiento. Los resultados experimentales muestran que la nueva propuesta supera claramente a CUSPARSE con valores de *speedup* de hasta 33.4×.

Los mejores valores de *speedup* tienden a darse con las matrices más pequeñas y se relacionan, en gran parte, con el *speedup* obtenido en la etapa de análisis (hasta 33.8×, obtenido en la misma matriz). Por último, estos valores de *speedup* obtenidos generan que la estrategia de uso de preprocesamiento pueda ser utilizada en un mayor campo de matrices. En concreto, la versión de CUSPARSE solamente compensaba su ejecución consistentemente en matrices de más de $20M$ no-ceros, mientras que la nueva propuesta supera a la versión sin niveles en matrices con más de $10M$ no-ceros.

Capítulo 6

Conclusiones y Trabajo Futuro

Este trabajo se buscó alcanzar el estado del arte en la aplicación de estrategias *sync-free* en el área de álgebra numérica. Luego de relevar el estado del arte, se decidió poner foco en las operaciones $SpTRSV$ e ILU en el entendido de que las estructuras de dependencias de estas rutinas pueden beneficiarse de estas estrategias. Adicionalmente, dado que estas dos operaciones demandan una gran parte del tiempo de cómputo de métodos iterativos como GMRES o CG, las mejoras a estas rutinas impactan directamente en el rendimiento global de estos.

6.1. Conclusiones

En esta tesis se trabajó en la aplicación del paradigma *sync-free* a rutinas de álgebra lineal dispersa. En concreto, se desarrollaron implementaciones de las rutinas $SpTRSV$ e ILU. En el caso de $SpTRSV$ se trabajó sobre la implementación propuesta en [9]. Del análisis de dicha implementación surgieron dos líneas de trabajo: la mejora del acceso a memoria y la búsqueda de optimizar el orden de ejecución de las filas. Debido a que no existían implementaciones *sync-free* de la ILU de código abierto, se puso foco en desarrollar distintas rutinas de ILU-0 competitivas con el estado del arte en el tema.

Para mejorar el acceso a memoria de la $SpTRSV$ se propuso un nuevo formato de almacenamiento ($HYB_{syncfree}$) diseñado para que los hilos de un *warp* accedan a direcciones contiguas de memoria de manera simultánea. Por otro lado, se propuso un ordenamiento siguiendo el paradigma ALAP en el que se posterga la ejecución de las filas que no son parte del camino crítico. Se

compararon las nuevas rutinas optimizadas contra la línea base y otras rutinas disponibles públicamente. Los resultados obtenidos muestran que los nuevos *solvers* propuestos superan al estado del arte. Adicionalmente, se observa que la ganancia obtenida por los *solvers* compensa claramente el sobrecosto del análisis en los contextos en los que se ejecutan muchas iteraciones del mismo y es posible reusar el análisis.

Se propusieron varias estrategias distintas para implementar la ILU. Las distintas implementaciones se diferenciaron en cómo se hace la etapa de actualización, el tamaño de los vectores y cómo se asignan las filas a dichos vectores. Aunque los resultados experimentales se encuentran ligeramente por debajo de los de CUSPARSE, las rutinas propuestas son competitivas y muestran fortalezas en algunos conjuntos de matrices. Adicionalmente, las implementaciones propuestas se encuentran disponibles de manera pública, por lo que sirven como punto de partida para otros esfuerzos de la comunidad en el área.

Finalmente, la implementación propuesta que hace una etapa de análisis para generar una estructura *level-set* que acelere el cómputo posterior, supera a su contraparte de CUSPARSE. Gran parte de la mejora de la nueva propuesta proviene de la etapa de análisis. Dado que, en general, en la ILU no se hacen múltiples ejecuciones por etapa de preprocesamiento, las mejoras en el análisis aportadas impactan significativamente en la viabilidad de esta estrategia. Por otro lado, esta rutina puede ser especialmente útil cuando se usa la ILU como un precondicionador de un método iterativo, ya que todo el costo del preprocesamiento puede ser reutilizado para la etapa de preprocesamiento de la $SpTRSV$. En concreto, sería posible integrar las implementaciones de $SpTRSV$ e ILU propuestas en esta tesis reusando todos los datos obtenidos en la etapa de análisis de ILU lo que reduciría significativamente el tiempo de ejecución del análisis de $SpTRSV$.

6.2. Publicaciones

A continuación se listan las publicaciones que se hicieron en el marco de esta tesis, correspondientes a dos conferencias internacionales. En ambas, Manuel Freire es el autor principal.

- M. Freire, E. Dufrechou, P. Ezzatti "A new level-set analysis and sparse storage format for the SPTRSV in GPUs", *2024 IEEE 36th International*

Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Hawaii, USA, pp 56-69, Noviembre 2024. IEEE. Trabajo premiado como *best paper* de la conferencia.

- M. Freire, E. Dufrechou, P. Ezzatti "A synchronization-free incomplete LU factorization for GPUs with level-set analysis", *33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2025)*, Turin, Italia, Marzo 2025. Trabajo aceptado para publicación.

6.3. Trabajo Futuro

Los esfuerzos hechos en esta tesis dejan varias líneas prometedoras de trabajo futuro relacionadas a la aplicación del paradigma *sync-free* a rutinas de álgebra lineal. En concreto, se identificaron las siguientes líneas:

- Desarrollar implementaciones del preprocesamiento que no requieran el cálculo de la estructura de niveles. De los análisis hechos en [26] surge que esta etapa es el cuello de botella del algoritmo, por lo que buscar alternativas a esta etapa que permitan evitar el *deadlock* al reordenar las filas debería generar un impacto significativo en el total del tiempo de cómputo.
- Aplicar estrategias de inteligencia computacional para elegir cuál de las rutinas ejecutar en un contexto dado. Esta línea tiene una importancia superlativa dado que en ninguna de las dos operaciones ($SpTRSV$ e ILU) existe una implementación que supere al resto en todos los casos. Particularmente en el caso de $SpTRSV$ también debe considerarse la variación entre los tiempos de preprocesamiento de las distintas rutinas.
- Integrar las implementaciones *sync-free* de $SpTRSV$ e ILU en implementaciones de métodos numéricos iterativos como por ejemplo el Gradiente Conjugado precondicionado. En esta línea, es interesante explorar la integración de las dos rutinas, en especial en lo que respecta al preprocesamiento donde, debido a que el patrón de no-ceros de las filas no cambia durante la ILU, existen margen para generar implementaciones que solapen el cómputo o reutilicen los datos del análisis en ambas operaciones.
- Relacionado con el punto anterior, si bien en general la ILU no puede

compensar el tiempo de cómputo del preprocesamiento con sucesivas ejecuciones de la rutina, sí podría hacerlo en contextos en los que los datos calculados sean utilizados por una posterior ejecución del $SpTRSV$. En estos contextos tendría sentido contar con una rutina que utilice todos estos datos aún cuando la ganancia en el tiempo de cómputo sea moderada ya que solo adelantaría el cómputo que posteriormente haría el análisis de la $SpTRSV$.

- Finalmente, esta tesis se centra exclusivamente en estrategias *sync-free* que utilizan únicamente los *cooperative groups* como herramienta de sincronización. Sin embargo, las nuevas arquitecturas de CUDA brindan otras alternativas para atacar el problema del costo de sincronización (*cuda graphs*, paralelismo dinámico, *static grids*). A priori, parece posible utilizar estas herramientas para expandir la *suite* de implementaciones de ambas operaciones.

Referencias bibliográficas

[1] Converting dataframe to csr matrix. https://hippocampus-garden.com/pandas_sparse/. Último acceso: 2024-27-12.

[2] David Blair Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufmann, 2010. ISBN 978-0-12-381472-2. URL http://www.elsevier.com/wps/find/bookdescription.cws_home/722320/description#description.

[3] Cap 6: Shared memory. syncfusion.com/succinctly-free-ebooks/cuda/shared-memory. Último acceso: 2025-20-1.

[4] *CUDA C++ Programming Guide*, . URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[5] Zhengyang Lu, Yuyao Niu, and Weifeng Liu. Efficient block algorithms for parallel sparse triangular solve. In *Proceedings of the 49th International Conference on Parallel Processing*, pages 1–11, 2020.

[6] Zhengyang Lu and Weifeng Liu. TileSpTRSV: a tiled algorithm for parallel sparse triangular solve on GPUs. *CCF Transactions on High Performance Computing*, 5(2):129–143, 2023.

[7] Jiya Su, Feng Zhang, Weifeng Liu, Bingsheng He, Ruofan Wu, Xiaoyong Du, and Rujia Wang. CapelliniSpTRSV: A Thread-Level Synchronization-Free Sparse Triangular Solve on GPUs. *Proceedings of the 49th International Conference on Parallel Processing*, 2020.

[8] Feng Zhang, Jiya Su, Weifeng Liu, Bingsheng He, Ruofan Wu, Xiaoyong Du, and Rujia Wang. Yuenyeungsptrsv: A thread-level and warp-level fusion synchronization-free sparse triangular solve. *IEEE Trans. Parallel Distributed Syst*, pages 2321–2337, 2021.

[9] Ernesto Dufrechou and Pablo Ezzatti. Using analysis information in the synchronization-free GPU solution of sparse triangular systems. *Concurr. Comput. Pract. Exp.*, 32(10), 2020.

[10] Martin Stoll. A literature survey of matrix methods for data science. *GAMM-Mitteilungen*, 43(3):e202000013, 2020.

[11] Biswa Nath Datta. *Numerical linear algebra and applications*. SIAM, 2010.

[12] Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU press, 2013.

[13] Blas. https://www.netlib.org/blas/. Último acceso: 2023-31-12.

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

[15] Edward Anderson and Youcef Saad. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing*, 1(01):73–95, 1989.

[16] Maxim Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011*, 1, 2011.

[17] Kevin Wainwright et al. *Fundamental methods of mathematical economics*. McGraw-Hill, 2005.

[18] Henry William Wyld and Gary Powell. *Mathematical methods for physics*. CRC Press, 2020.

[19] Timothy A Davis. *Direct methods for sparse linear systems*. SIAM, 2006.

[20] Jörg Liesen and Zdenek Strakos. *Krylov subspace methods: principles and analysis*. Numerical Mathematics and Scie, 2013.

[21] Gilbert W Stewart. *Matrix algorithms: volume 1: basic decompositions*. SIAM, 1998.

[22] Gilbert W Stewart. *Matrix Algorithms: Volume II: Eigensystems*. SIAM, 2001.

[23] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[24] John Hennessy and David Patterson. A new golden age for computer architecture: domain-specific hardware/software co-design, enhanced. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[25] Weifeng Liu, Ang Li, Iain S. Duff Jonathan D. Hogg, and Brian Vinter. A synchronization-free algorithm for parallel sparse triangular solves. *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings, Lecture Notes in Computer Science*, 9833:617–630, 2016.

[26] Manuel Freire, Juan Ferrand, Franco Seveso, Ernesto Dufechou, and Pablo Ezzatti. A GPU method for the analysis stage of the SPTRSV kernel. *J. Supercomput.*, 2023.

[27] Yousef Saad. *Numerical methods for large eigenvalue problems: revised edition*. SIAM, 2011.

[28] Hoang-Vu Dang and Bertil Schmidt. The sliced COO format for sparse matrix-vector multiplication on CUDA-enabled GPUs. In *Proceedings of the International Conference on Computational Science, ICCS 2012, Omaha, Nebraska, USA, 4-6 June, 2012*, volume 9 of *Procedia Computer Science*, pages 57–66. Elsevier, 2012.

[29] Hoang-Vu Dang and Bertil Schmidt. CUDA-enabled sparse matrix-vector multiplication on GPUs using atomic operations. *Parallel Comput*, 39(11): 737–750, 2013.

[30] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. yaSpMV: yet another SpMV framework on GPUs. *ACM SIGPLAN Notices*, 49(8): 107–118, August 2014.

[31] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale Sparse Matrix Package (YSMP) – I : The symmetric codes. Technical Report 112, Dept. of Computer Science, Yale Univ., 1977.

[32] Xiaowen Feng, Hai Jin, Ran Zheng, Kan Hu, Jingxiang Zeng, and Zhiyuan Shao. Optimization of sparse matrix-vector multiplication with variant CSR on GPUs. In *ICPADS*, pages 165–172. IEEE Computer Society, 2011.

[33] Zbigniew Koza, Maciej Matyka, Sebastian Szkoda, and Lukasz Miroslaw. Compressed multirow storage format for sparse matrices on graphics processing units. *SIAM J. Sci. Comput*, 36(2), 2014.

[34] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *High Performance Embedded Architectures and Compilers, 5th International Conference. Proceedings*, volume 5952, pages 111–125, 2010.

[35] Davide Barbieri, Valeria Cardellini, Alessandro Fanfarillo, and Salvatore Filippone. Three storage formats for sparse matrices on GPGPUs. Technical report, Università di Roma Tor Vergata, 2015.

[36] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing*, 36(5):401–423, 2014.

[37] Hartwig Anzt, Stanimire Tomov, Piotr Luszczek, William Sawyer, and Jack Dongarra. Acceleration of GPU-based krylov solvers via data transfer reduction. *The International Journal of High Performance Computing Applications*, 29(3):366–383, 2015.

[38] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC*. ACM, 2009. ISBN 978-1-60558-744-8. URL http://www.computer.org/csdl/proceedings/sc/2009/8744/00/index.html.

[39] Daniel Weber, Jan Bender, Markus Schnoes, André Stork, and Dieter W. Fellner. Efficient GPU data structures and methods to solve sparse linear systems in dynamics applications. *Computer Graphics Forum*, 32(1):16–26, 2013.

[40] 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 (15th PPOPP'10), pages 115–125, Bangalore, India, January 2010. ACM SIGPLAN.

[41] Ruipeng Li and Yousef Saad. GPU-accelerated preconditioned iterative linear solvers. *J. Supercomput*, 63(2):443–466, 2013.

[42] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. Sparse matrix-vector multiplication on GPGPUs. *ACM Trans. Math. Softw*, 43(4):30:1–30:49, 2017.

[43] 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. ISBN 158113374X. doi: 10.1145/383259.383274. URL https://doi.org/10.1145/383259.383274.

[44] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966. doi: 10.1109/PROC.1966.5273.

[45] Mikhail J. Atallah, editor. *Algorithms and theory of computation handbook*. CRC Press, pub-CRC:adr, 1999. ISBN 0-8493-2649-4.

[46] CUDA GPUs Compute Capabilites. https://developer.nvidia.com/cuda-gpus, . Último acceso: 2024-25-11.

[47] Mark Harris and Kyrylo Perelygin. Cooperative Groups: Flexible CUDA Thread Programming. https://developer.nvidia.com/blog/cooperative-groups/. Último acceso: 2022-21-11.

[48] Nathan Bell and Michael Garl. Efficient sparse matrix-vector multiplication on CUDA. Technical report, 2014. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.463.7511;http://sbel.wisc.edu/Courses/ME964/Literature/techReportGarlandBell.pdf.

[49] Ernesto Dufrechou, Pablo Ezzatti, and Enrique S. Quintana-Ortí. Selecting optimal SpMV realizations for GPUs via machine learning. *Int. J. High Perform. Comput. Appl*, 35(3), 2021.

[50] Steven Dalton, Sean Baxter, Duane Merrill, Luke N. Olson, and Michael Garland. Optimizing sparse matrix operations on GPUs using merge path. In *IPDPS*. IEEE Computer Society, 2015.

[51] Timothy M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. STOC '07, page 590–598, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936318. doi: 10.1145/1250790.1250877. URL https://doi.org/10.1145/1250790.1250877.

[52] 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. ISBN 1595933409. doi: 10.1145/1137856.1137866. URL https://doi.org/10.1145/1137856.1137866.

[53] 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. doi: 10.1109/IPDPS.2016.54.

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

[55] 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.

[56] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[57] The MathWorks Inc., 2024. URL https://www.mathworks.com/products/matlab.html.

[58] 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. ISBN 9781450343619. doi: 10.1145/2925426.2926273. URL https://doi.org/10.1145/2925426.2926273.

[59] 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. doi: 10.1137/110838844. URL https://doi.org/10.1137/110838844.

[60] Tony F. Chan and Henk A. Van der Vorst. *Approximate and Incomplete Factorizations*, pages 167–202. Springer Netherlands, Dordrecht, 1997. ISBN 978-94-011-5412-3. doi: 10.1007/978-94-011-5412-3_6.

[61] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second edition, 2003. doi: 10.1137/1.9780898718003. URL https://epubs.siam.org/doi/abs/10.1137/1.9780898718003.

[62] David Hysom and Alex Pothen. A scalable parallel algorithm for incomplete factor preconditioning. volume 22, pages 2194–2215, 2001. doi: 10.1137/S1064827500376193. URL https://doi.org/10.1137/S1064827500376193.

[63] Maxim Naumov. Parallel incomplete-LU and cholesky factorization in the preconditioned iterative methods on the GPU. *Nvidia Technical Report NVR-2012-003*, 2012.

[64] Maxim Naumov, Patrice Castonguay, and Jonathan Cohen. Parallel graph coloring with applications to the incomplete-LU factorization on the GPU. *NVIDIA Technical Report NVR-2015-001*, 2015.

[65] Edmond Chow and Aftab Patel. Fine-grained parallel incomplete lu factorization. *SIAM Journal on Scientific Computing*, 37(2):C169–C193, 2015. doi: 10.1137/140968896. URL https://doi.org/10.1137/140968896.

[66] Edmond Chow, Hartwig Anzt, and Jack Dongarra. Asynchronous iterative algorithm for computing incomplete factorizations on GPUs. In Julian M. Kunkel and Thomas Ludwig, editors, *High Performance Computing*, pages 1–16, Cham, 2015. Springer International Publishing. ISBN 978-3-319-20119-1.

[67] Alan George, Michael T. Heath, Joseph Liu, and Esmond Ng. Solution of sparse positive definite systems on a shared-memory multiprocessor. *International Journal of Parallel Programming*, 15(4):309–325, 1986. ISSN 1573-7640. doi: 10.1007/BF01407878.

[68] Alan George, Michael T. Heath, and Joseph Liu. Parallel cholesky factorization on a shared-memory multiprocessor. *Linear Algebra and its Applications*, 77:165–187, 1986. ISSN 0024-3795. doi: https://doi.org/10.1016/0024-3795(86)90167-9. URL https://www.sciencedirect.com/science/article/pii/0024379586901679.

[69] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In *Supercomputing: 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings 29*, pages 124–140. Springer, 2014.

[70] Ernesto Dufrechou and Pablo Ezzatti. Solving sparse triangular linear systems in modern gpus: A synchronization-free algorithm. In Ivan Merelli, Pietro Liò, and Igor V. Kotenko, editors, *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2018, Cambridge, United Kingdom, March 21-23, 2018*, pages 196–203. IEEE Computer Society, 2018.

[71] Ruipeng Li and Chaoyu Zhang. Efficient parallel implementations of sparse triangular solves for GPU architectures. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*, pages 106–117. SIAM, 2020.

[72] Weifeng Liu, Ang Li, Jonathan D. Hogg, Iain S. Duff, and Brian Vinter. Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides. *Concurr. Comput. Pract. Exp*, 29(21), 2017.

[73] Ernesto Dufrechou and Pablo Ezzatti. A new GPU algorithm to compute a level set-based analysis for the parallel solution of sparse triangular systems. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS*, pages 920–929, 2018.

[74] Jianqi Zhao, Yao Wen, Yuchen Luo, Zhou Jin, Weifeng Liu, and Zhenya Zhou. SFLU: Synchronization-free sparse lu factorization for fast circuit simulation on GPUs. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 37–42, 2021.

[75] Herbert L. Stone. Iterative solution of implicit approximations of multidimensional partial differential equations. *SIAM Journal on Numerical Analysis*, 5(3):530–558, 1968.

[76] Franco Seveso, Ernesto Dufrechou, Pablo Ezzatti, and Gabriel Usera. Element scheduling for GPU-accelerated finite-volumes computations. In *Proceedings of the 22 International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar 2024)*, 2024.

[77] José Ignacio Aliaga, Ernesto Dufrechou, Pablo Ezzatti, and Enrique S. Quintana-Ortí. An efficient GPU version of the preconditioned GMRES method. *J. Supercomput.*, 75(3):1455–1469, 2019.

[78] Chenhao Xie, Jieyang Chen, Jesun Firoz, Jiajia Li, Shuaiwen Leon Song, Kevin Barker, Mark Raugas, and Ang Li. Fast and scalable sparse triangular solver for multi-GPU based HPC architectures. In *Proceedings of the 50th International Conference on Parallel Processing*, pages 1–11, 2021.

[79] cuSPARSE documentation. https://docs.nvidia.com/cuda/cusparse/index.html, . Último acceso: 2024-31-12.

[80] cuSPARSE library. https://cusplibrary.github.io/, . Último acceso: 2024-31-12.

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

[82] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. Tilespgemm: A tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 90–106, 2022.

[83] Haonan Ji, Huimin Song, Shibo Lu, Zhou Jin, Guangming Tan, and Weifeng Liu. Tilespmspv: A tiled algorithm for sparse matrix-sparse vector multiplication on GPUs. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–11, 2022.

[84] Juan Ferrand, Manuel Freire, and Franco Seveso. *Aplicación de métodos avanzados de sincronización en GPUs para la resolución de sistemas triangulares dispersos*. Proyecto de grado, Udelar, FIng, Febrero 2023.

[85] José I Aliaga, Matthias Bollhöfer, Ernesto Dufrechou, Pablo Ezzatti, and Enrique S Quintana-Ortí. Extending ILUPACK with a task-parallel version of bicg for dual-GPU servers. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 71–78, 2018.

[86] Matthias Bollhöfer, José Ignacio Aliaga, Alberto F Martín, and Enrique S Quintana-Ortí. Ilupack. http://ilupack.tu-bs.de/, 2011. Último acceso: 2025-15-02.

[87] José I Aliaga, Ernesto Dufrechou, Pablo Ezzatti, and Enrique S Quintana-Ortí. Accelerating the task/data-parallel version of ILUPACK's BiCG in multi-CPU/GPU configurations. *Parallel Computing*, 85:79–87, 2019.

[88] José I. Aliaga, Matthias Bollhöfer, Ernesto Dufrechou, Pablo Ezzatti, and Enrique S. Quintana-Ortí. A data-parallel ILUPACK for sparse general and symmetric indefinite linear systems. In *Euro-Par 2016: Parallel Processing Workshops*. Springer International Publishing, 2017.

[89] José I. Aliaga, Rosa M. Badia, Maria Barreda, Matthias Bollhöfer, Ernesto Dufrechou, Pablo Ezzatti, and Enrique S. Quintana-Ortí. Exploiting task and data parallelism in ILUPACK's preconditioned CG solver on NUMA architectures and many-core accelerators. *Parallel Computing*, 54:97–107, 2016. ISSN 0167-8191. 26th International Symposium on Computer Architecture and High Performance Computing.

[90] Pablo Igounet, Pablo Alfaro, Gabriel Usera, and Pablo Ezzatti. GPU acceleration of the caffa3d.mb model. In *Computational Science and Its Applications – ICCSA 2012*, pages 530–542, Berlin, Heidelberg, 2012.

[91] Pablo Igounet, Pablo Alfaro, Gabriel Usera, and Pablo Ezzatti. Towards a finite volume model on a many-core platform. *Int. J. High Perform. Syst. Archit.*, 4(2):78–88, December 2012.

[92] Pablo Igounet, Pablo Alfaro, Martín Pedemonte, and Pablo Ezzatti. A GPU implementation of the SIP method. In *2011 30th International Conference of the Chilean Computer Science Society*, pages 195–201, 2011.

[93] Zoltan Baruch. Scheduling algorithms for high-level synthesis. *ACAM Scientific Journal*, 5(1-2):48–57, 1996.

[94] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, November 2011. ISSN 0098-3500 (print), 1557-7295 (electronic).

[95] Lung-Sheng Chien. How to avoid global synchronization by domino scheme, 2014.

[96] Benjamin Schlegel, Thomas Willhalm, and Wolfgang Lehner. Fast sorted-set intersection using SIMD instructions. *ADMS@ VLDB*, 1(8), 2011.

[97] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda c++ programming guide: 11.7.0, 2022. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-reduce-functionsl.