Universidad de la República

Facultad de Ingeniería

Aplicación de métodos avanzados de sincronización en GPUs para la resolución de sistemas triangulares dispersos

Informe de Proyecto de Grado presentado por

Juan Ferrand, Manuel Freire y Franco Seveso

como requisito de graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Pablo Ezzatti
Ernesto Dufrechou

Montevideo, 24 de febrero de 2023

Resumen

La resolución de sistemas de ecuaciones dispersas lineales triangulares ($SpTrSv$) es, en muchos casos, el cuello de botella de distintos métodos numéricos. En GPUs mayoritariamente se utilizan tres enfoques. Las estrategias Level-set utilizan un costoso preprocesamiento para determinar dependencias entre filas de la matriz y generar un cronograma estático de ejecución en paralelo para la etapa de resolución. Por otro lado, los métodos synchronization-free van generando el cronograma de manera dinámica a medida que las dependencias se van completando y evitando la etapa de análisis. Finalmente, se puede utilizar un enfoque híbrido que utilice la etapa de análisis para acelerar una etapa de resolución que utilice una estrategia synchronization-free. En este trabajo presentamos una rutina eficiente en GPU para la etapa de análisis y luego aplicamos algunas de estas ideas para la etapa de resolución. La comparación con otras rutinas disponibles de manera pública muestran una mejora en tiempo de ejecución para la resolución de sistemas tringulares generados por muchas matrices pertenecientes a la biblioteca SuiteSparse.

Palabras clave: $SpTrSv$, Synchronization-free, GPU

Índice general

Capítulo 1

Introducción

En el área de las ciencias aplicadas o la ingeniería es común utilizar como modelo sistemas de ecuaciones diferenciales parciales (o PDEs por su sigla en inglés) para aproximar realidades físicas. En la actualidad es común que estos modelos contemplen una gran cantidad de parámetros físicos generando que dichos sistemas de PDEs sean cada vez más grandes y complejos. Para resolver numéricamente estos sistemas se aplican métodos de discretización (diferencias finitas, volúmenes finitos, elementos finitos) que generan sistemas lineales de gran tamaño del tipo $Ax = b$ donde la matriz A tiene una baja proporción de coeficientes distintos de 0, x es el vector de las incógnitas y b el de los valores independientes. La resolución de estos sistemas es habitualmente la parte más costosa en cuanto a tiempo de cómputo y/o memoria y en función del tipo de problema (la matriz de coeficientes) se aplican distintos métodos. Los métodos de resolución pueden clasificarse en tres tipos: que hallan la solución exacta a menos de errores numéricos de redondeo (directos), que parten de una solución inicial y la mejoran hasta cumplir ciertas condiciones (iterativos) o que combinan ambas estrategias (híbridos).

Hallar la matriz A^{-1} es complejo y, en caso de que esta no esté bien condicionada, puede introducir importantes errores numéricos. Por esta razón los métodos más extendidos no calculan la matriz inversa. En el caso de los métodos directos tienden a factorizar la matriz en dos componentes triangulares en los que poder aplicar sustitución. El método más común es la eliminación Gaussiana que factoriza $A = LU$ y luego resuelve los sistemas $Ly = b$ y $Ux = y$ aplicando sustitución (hacia delante y hacia atrás, respectivamente). Sin embargo los métodos directos tienen algunos problemas, el principal es el conocido como *fill-in* (o llenado) que se genera al factorizar y consiste en que las matrices L o U tienden a tener valores no-cero en entradas en las que A tenía ceros. Esto genera que los costos de memoria aumenten y puedan llegar a ser prohibitivos. Para un abordaje en profundidad de los métodos directos puede verse [13].

Los métodos iterativos, por el contrario, no tienen las limitaciones antes mencionadas tendiendo a ser más rápidos en general y utilizar menos memoria. La estrategia general de estos métodos consiste en partir de una solución inicial

y mejorarla hasta que cumpla cierta condición con respecto al error o cantidad de iteraciones.

Existen dos tipos de métodos iterativos para sistemas de ecuaciones lineales, los estacionarios, que "definen" matrices M tales que sean relativamente parecidas a A pero más fáciles de invertir (por ejemplo Gauss-Seidel, Jacobi), o los métodos de Krylov, que forman una base ortogonal de la secuencia de potencias de la matriz por el residuo inicial (por ejemplo conjugate gradient). La desventaja de los primeros sobre los segundos es que no garantizan la convergencia salvo para tipos específicos de matrices. Para una revisión extensiva de esta clase de métodos ver [37].

Debido a que los métodos iterativos de Krylov generan una base, es posible probar que convergen en, como mucho n iteraciones donde n es la dimensión de la matriz A. Sin embargo, n puede ser muy grande y, además, cuando se introducen errores de redondeo la garantía de convergencia en esta cantidad de iteraciones ya no se cumple. La convergencia depende sustancialmente de la distribución de los valores propios y en menor medida del número de condición.

Una estrategia para acelerar la convergencia consiste en utilizar precondicionadores. Dado que las características de A son dadas, la idea general del precondicionamiento consiste en hallar una matriz M tal que la matriz $M^{-1}A$ tenga mejores características (es decir, que tenga un menor número de condicionamiento) y resolver el sistema $M^{-1}Ax = M^{-1}b$. La matriz M debe cumplir ser fácilmente invertible y tiene algunas otras características deseables. Si M puede ser descompuesta como la multiplicación de dos matrices L y U la multiplicación es equivalente a resolver primero la multiplicación $Ax = w$ y luego resolver el sistema $Mw = b$ para el que se aplica la misma idea que en los métodos directos. En este sentido, e intentando evitar el *fill-in*, se utilizan estrategias como la factorizacion *incomplete-LU* (ILU) que brindan matrices L y U tales que $LU = M \approx A$ con lo que $M^{-1}A \approx I$. Estas estrategias son las más comunes para el precondicionamiento en la actualidad como se plantea en [33] *"precondicionar como se conoce en la actualidad refiere en su mayoría a factorizaciones incompletas o aproximadas de la matriz de coeficientes"*.

En todos estos algoritmos la resolución de sistemas triangulares dispersos, operacion conocida como ($SpTrSv$) cumple un rol fundamental siendo en muchos casos el principal cuello de botella en cuanto a costo computacional[1]. Es por esto que en los últimos años se han dedicado muchos esfuerzos a la implementación de rutinas que resuelvan esta operación de manera eficiente.

El procedimiento para resolver la operación $SpTrSv$ es la sustitución. En estos casos cada incógnita resuelta es modificada en el resto de las ecuaciones lo que genera que siempre haya una incógnita lista para ser utilizadas en las otras ecuaciones. Una forma intuitiva de paralelizar la sustitución es resolver cada ecuación (o fila) de forma paralela con un procesador distinto una vez que se cumplen sus dependencias. Dado que la matriz de coeficientes es dispersa, cada ecuación depende únicamente de un puñado de incógnitas previas.

Las GPUs son plataformas muy utilizadas en el álgebra lineal dispersa debido

[1] Junto con la multiplicacion dispersa matriz y vector SPMV.

a su gran poder de cómputo y, sobre todo, al gran ancho de banda de memoria que es la principal restricción al trabajar con matrices dispersas. Existen varios esquemas para organizar el procesamiento, los más comunes en GPU son *level-set* y *synchronization-free*. El primero divide la operación en una primera etapa de preprocesamiento (llamada análisis) y una segunda etapa de resolución. El análisis calcula una estructura que organiza las filas de la matriz en conjuntos llamados niveles, tal que las filas de cada nivel pueden ser procesadas en paralelo y solo dependen de filas de niveles de índice inferior. Esto genera una planificación de ejecución estática que evita *deadlocks* [2] y permite ejecutar en paralelo. Si bien estos preprocesamientos son costosos pueden ser reutilizados por sistemas que compartan la misma matriz de coeficientes (o matrices con el mismo patrón de dispersión) con lo que si se ejecutan suficientes resoluciones el costo base del análisis es compensado por la mejora en la segunda etapa.

El otro paradigma de computación, también conocido como *self-scheduled*, consiste en que cada hilo consulte constantemente el estado de las incógnitas de las que depende la fila siendo procesada utilizando *busy waiting* [3]. Además de evitar el costo del análisis, este tipo de estrategias toman su nombre (*synchronization-free*) de que evitan la sincronización que es necesaria en las *level-set* al terminar cada nivel. Por otro lado, al utilizar *busy waiting* se incurre en un costo asociado al cómputo extra generado por la consulta sistemática. Sin embargo, la GPU puede reducir el costo ya que para enmascarar la latencia del acceso a memoria cuando un *warp*[4] hace un acceso es inmediatamente retirado de ejecución hasta que este se resuelva. Se generan bloqueos implícitos cada vez que se hacen estas consultas con lo que en la práctica no se incurre en este sobre costo.

Por último es posible hacer enfoques híbridos que utilicen la etapa de análisis para acelerar una etapa de resolución *sync-free* [16]. En estos casos es importante balancear el costo fijo que implica el análisis con la reducción obtenida en la ejecución de la etapa de resolución. Para esto es fundamental considerar la cantidad de ejecuciones que se realizara para el sistema.

En este trabajo se continua con los esfuerzos comenzados en [15, 16] y se avanza en la optimización de ambas etapas. En concreto, se presentan dos enfoques distintos para la optimización de la rutina de análisis, utilizando nuevas estrategias de sincronización y avanzando en la mejora de las rutinas de resolución. Además, se genera una biblioteca pública con las distintas alternativas de resolución y análisis modificadas con las optimizaciones propuestas en este documento.

El resto del trabajo se estructura de la siguiente manera. En el Capítulo 2, se introducen conceptos básicos tanto en lo relativo a operaciones de álgebra

[2] Situación dentro de un sistema informático en la que dos o más tareas por ejemplo: hilos o procesos estan esperándose el uno al otro para terminar o continuar su trabajo. Esta situación detiene las tareas y les impide continuar.

[3] Conocido en español como espera activa o espera ocupada es una técnica donde un proceso repetidamente verifica una condición, tal como esperar una entrada de teclado o si el ingreso a una sección crítica está habilitado.

[4] Un warp es un conjunto de threads (32) que ejecutan simultáneamente en un mismo Streaming Multiprocesor (SM) de la GPU.

dispersa como de hardware. En el Capítulo 3, se aborda la optimización del análisis utilizando dos enfoques distintos. En el Capítulo 4, utilizando nuevas herramientas de sincronización, se presentan optimizaciones a una de las versiones de la etapa de resolución. En el Capítulo 5, se aborda una exhaustiva evaluación experimental comparando con otras implementaciones disponibles de manera pública. Por último, en el capítulo 6, se presentan las conclusiones de este trabajo y las posibles líneas de trabajo futuro.

Capítulo 2

Conceptos preliminares

En este capítulo se presentan conceptos importantes para la comprensión del resto del trabajo. En concreto, se introducen conceptos básicos sobre matrices dispersas, como por ejemplo una revisión sobre sus formatos de almacenamiento. Luego, se presentan fundamentos sobre el uso del álgebra lineal dispersa. En específico se abordan dos operaciones muy utilizadas como son la multiplicación matriz dispersa vector y la resolución de sistema triangular disperso. Por último se brinda una introducción al hardware de alto desempeño (HPC, por su sigla en inglés) con énfasis en el uso de GPUs y ciertas funcionalidades de esta tecnología que son utilizadas en el trabajo.

2.1. Matrices Dispersas

Una matriz dispersa es una matriz en la que una gran proporción de sus coeficientes tienen valor 0. Estas matrices tienen un amplio rango de aplicación y particularmente en el área de la física como se mencionó en la sección anterior. Al ser una definición relativamente genérica existe una gran variedad de matrices dispersas que pueden clasificarse según diferentes características, como la dimensión o la cantidad de no-ceros. Adicionalmente, en el álgebra dispersa es importante tener en cuenta el patrón de dispersión de las matrices (cómo se distribuyen los no-ceros en la matriz), además de las otras características mencionadas anteriormente.

En general para almacenar matrices densas se utiliza un arreglo de $n \times m$ elementos donde n es la cantidad de filas y m la cantidad de columnas. Dependiendo de si se almacenan las filas o las columnas de forma contigua, se tiene un ordenamiento *row major* o *column major*. La principal ventaja de esta representación es la facilidad en la que se puede realizar acceso directo a los elementos (sabiendo los índices se puede acceder a cualquier coeficiente en $O(1)$) lo que permite implementaciones sencillas y eficientes para operaciones como la suma, multiplicación escalar o trasposición. Otra ventaja importante es que este tipo de almacenamiento en arreglos permite explotar la localidad

espacial de los datos cuando se implementan operaciones.

Las matrices dispersas, sin embargo, agregan complicaciones adicionales. Utilizar una estrategia similar a la de las matrices densas implicaría un desperdicio de memoria importante debido a que se almacenaría una gran cantidad de entradas con valor cero. Problemas similares sucederían al trasladar directamente operaciones del álgebra densa puesto que implicaría una gran cantidad de cálculos inútiles al operar con las entradas nulas.

Cómo almacenar las matrices dispersas de la mejor manera en cada situación es un problema abierto. A lo largo del tiempo han surgido diversos formatos de almacenamiento que buscan explotar la gran cantidad de ceros que existen en las matrices dispersas para utilizar menos memoria y a la vez conseguir implementaciones eficientes para las operaciones algebraicas. En otras palabras, la alta dependencia entre el ahorro de almacenamiento, el desempeño en las operaciones y el patrón de dispersión de cada matriz implica que no existe un formato de almacenamiento que sea el mejor para todas las matrices (y todas las plataformas de *hardware*).

Quizá la estrategia más intuitiva para almacenar matrices dispersas es el formato de coordenadas (COO), que solamente representa los elementos distintos a cero ordenados por fila en un arreglo unidimensional. Debido a que los índices de fila y columna de un elemento no pueden ser recuperados a partir de su posición en memoria (a diferencia del formato para matrices densas) es necesario almacenarlos explícitamente. En resumen este formato representa a los elementos distintos de cero como una terna de valores (val, row_idx, col_idx) que representa su valor, índice de fila y columna, respectivamente. Un esquema con una matriz de ejemplo puede verse en la Figura 2.1. Este formato tiene dos problemas importantes para su uso general. El más evidente es la irregularidad de los datos. Al usar este formato se pierde el acceso directo que existía en las matrices densas y encontrar un elemento pasa a ser $O(nnz)$ o en el mejor de los casos $O(log(nnz))$ si está ordenada. El segundo problema puede verse en la Figura 2.1 donde el vector de filas repite los índices 0, 1 y 2. Esto es algo general que cuando se ordenan los elementos por fila o columna se incurre en redundancia de información en el índice correspondiente. Si bien su costo de memoria significa una gran reducción comparando contra la estrategia densa sigue siendo elevado requiriendo 16 bdyte por cada no-cero de los cuales la mitad se utiliza para almacenar los índices. Estos problemas han sido foco de diversos esfuerzos, los trabajos de Dang et al. [22][23] y Yan et al. [34] proponen mejoras para afrontarlos.

Uno de los formatos más utilizados es el Compresed Sparse Row (CSR) [31]. Este formato ordena los elementos por fila y elimina la redundancia explicada anteriormente "comprimiendo" el índice de fila. Además es lo suficientemente genérico para funcionar bien independientemente de la estructura o el patrón de dispersión de la matriz a almacenar. CSR (al igual que como su formato equivalente CSC) utiliza tres vectores para representar una matriz. El primer vector *val* mantiene todos los valores distintos a cero, el segundo vector *col_idx* mantiene los índices asociados a cada valor. La diferencia con el formato COO radica únicamente en el vector *row_ptr* que, en lugar de mantener los índices

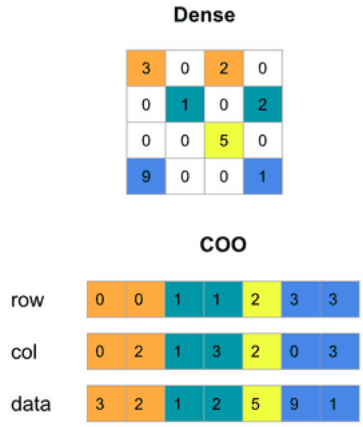

Figura 2.1: Esquema del formato COO para una matriz de ejemplo. Extraído de [1].

de fila de cada coeficiente mantiene el índice, en el vector de valores, del primer elemento de la fila. Un ejemplo con la misma matriz puede verse en la Figura 2.2. En general esta estrategia requiere una mayor cantidad de accesos para obtener los índices de un elemento dado debido a que es necesario buscar el índice de fila, esto es especialmente problemático cuando se quiere acceder por columna. Sin embargo, el ahorro significativo que se produce en la cantidad de memoria (en lugar de requerir $16 \times nnz$ bytes se requieren únicamente $(12 \times nnz + 4 \times dim)$ bytes) compensa cuando se procesan suficientes elementos. El ahorro de memoria es más notorio cuanto más elementos hayan en promedio en las filas (o columnas en el formato CSC).

Reducir el sobre-costo de almacenar los índices es uno de los principales objetivos cuando se desarrollan nuevos formatos de almacenamiento. En este sentido, una estrategia utilizada para optimizar formatos es por medio de bloques. La idea general de los formatos por bloque es dividir la matriz en sub-bloques regulares. Una vez definidos los bloques, pueden ser tratados como una unidad que se almacenará entera en memoria, permitiendo aplicar cualquier formato disperso a la matriz de bloques. Estos formatos tienen buenos resultados cuando los no-ceros están agrupados en *clusters* dado que ahorran en almacenamiento de índices al almacenar solo un par de índices por cada bloque en lugar de uno por no-cero. Por otro lado, si los coeficientes no-ceros se encuentran más repartidos, la reducción en costo de índices será negativamente compensada por el costo de almacenar gran cantidad de ceros de manera explícita.

En la literatura se encuentan diversas propuestas de formatos para bloques, Yan et al. propusieron el Block-based Compressed Common Coordinate (BCCOO) [34]. Este formato aplica la idea de COO a bloques y además incorpora una bandera para determinar si el bloque es el último de la fila, permitiendo comprimir este índice. Otro formato es el Blocked CSR (BCSR) propuesto por

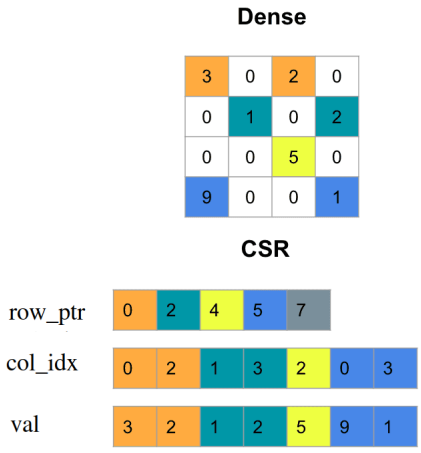

Figura 2.2: Esquema del formato CSR para una matriz de ejemplo. Extraído de [1].

Choi et al. [24]. Como su nombre indica este formato consiste en aplicar CSR a una matriz dividida en bloques densos.

Por último es interesante comentar el formato ELLPACK-ITPAK (ELL). Este formato almacena la matriz en dos arreglos distintos de tamaño $n \times c$ donde:

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

$nnz(j)$ es el número de elementos distintos a cero en la fila j.

El primer arreglo, *val*, mantiene los valores no-cero correspondiente a cada fila. Cuando una fila tiene menos de c valores distintos de cero, es completada con ceros al final de la fila. En este formato los índices de fila están implícitos, dado que pueden ser inferidos a partir de la posición en memoria. Por el contrario, los índices de columna deben ser mantenidos de forma explícita en el segundo arreglo, *col_idx*, donde el elemento col_idx_i es el índice de la columna del elemento val_i en el primer arreglo. Un ejemplo de aplicación de este formato se muestra en la Figura 2.3.

Este formato fue diseñado específicamente para acelerar el cálculo de la operación *SpMV* en GPUs *Nvidia*. En la implementación original el arreglo es guardado de forma *column-major* permitiendo acceso *coalesced* a memoria dado que en esta operación, en general, cada hilo procesa una fila secuencialmente.

Es fácil ver que ELL tendrá buenos resultados en los casos en los que la cantidad de no-ceros sea similar entre todas las filas, pero en matrices con algunas filas con cantidad de elementos mucho mayor que el promedio, el *zero padding* (almacenamiento de ceros explícitos al final para alinear las filas) se convierte en prohibitivo y el formato deja de ser conveniente.

Figura 2.3: Esquema del formato ELL para una matriz de ejemplo.

Afrontar la falta de adaptabilidad de ELL es un campo importante de trabajo. En esta línea de trabajo se propuso el formato Sliced-Ell (SELL-C) [7] que parte las filas de la matriz en *slices*, cada una con un c_i local en lugar de un único c global. El número de filas por *slice* es un parámetro proporcionado por el usuario que depende del problema. Debido a que este valor afecta directamente el rendimiento es importante tener buenas heurísticas para elegirlo (que elijan un buen número pero no agreguen mucho costo extra). ELL puede ser visto como un caso particular de SELL-C cuando $c = n$.

Otra forma de atacar este problema se afronta combinando ELL con otro formato. Un ejemplo de esto es el trabajo de Bell y Garland. Estos autores propusieron el formato HYB [10] que combina ELL y COO. Se elije un valor k tal que $2/3$ de las filas sean más cortas que k y se toma $c = k$. Para las filas más largas que k los últimos elementos son guardados en formato COO.

La revisión de formatos de este trabajo no pretende ser exhaustiva sino que solamente aborda algunas de las propuestas más destacadas. Existe una gran variedad de formatos de almacenamiento que explotan distintas características (más o menos particulares) de un conjunto de las matrices dispersas (por ejemplo DIA, JDS). Se han propuesto también formatos focalizados en optimizar operaciones de álgebra dispersa particulares. Para una revisión en mayor profundidad puede verse [18].

2.2. Operaciones de álgebra dispersa

Las operaciones *SpMV* (*sparse matrix vector multiplication*) y *SpTrSv* (*sparse triangular solver*) son dos de las operaciones predominantes en el álgebra lineal dispersa. La importancia de éstas radica tanto en su uso muy difundido como en sus contextos de aplicación. Debido a su importancia existe una gran cantidad de implementaciones para cada una de ellas en distintos formatos de almacenamiento, así como diversas plataformas de hardware. En este apartado se presentan estas operaciones y se mencionan algunas características.

2.2.1. Multiplicación Matriz dispersa-Vector ($SpMV$)

El producto de una matriz dispersa y un vector ($SpMV$ sus siglas en inglés) es de la forma:

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

donde la matriz de entrada A es dispersa y el vector x de entrada e y de salida son vectores densos.

Las multiplicaciones matriz dispersa vector son, probablemente, las operaciones más utilizadas del álgebra dispersa. En especial porque esta operación representa la mayor parte del costo de muchos métodos iterativos para resolver sistemas lineales o realizar el cálculo de valores propios para matrices de gran tamaño. Esto ha motivado que existan una gran diversidad de implementaciones de la operación para todos los formatos de almacenamiento. La optimización de la implementación de la $SpMV$ es un problema abierto al que se dedica una gran cantidad de esfuerzo por parte de la comunidad científica.

La eficiencia de las implementaciones de esta operación depende de la estructura de la matriz y el formato en el que se encuentra almacenada la misma. Si bien existe una gran cantidad de implementaciones para formatos genéricos como CSR, en la mayoria de las plataformas se han desarrollado formatos de almacenamiento específicos para mejorar el rendimiento de esta operación para distintos contextos.

El trabajo de Bell y Garland [9][10] marcó un punto de inflexión en el desarrollo de $SpMV$ en GPU. En este trabajo los autores propusieron varias implementaciones de $SpMV$ en los principales formatos (COO, CSR, DIA, ELL, HYB, etc.) y presentaron un estudio en profundidad con respecto al rendimiento de los formatos donde hallaron que la implementación en formato HYB es significativamente superior a las otras, con una de las implementaciones en CSR (denominada *vector*) siendo competitiva.

En [17] los autores estudiaron diferentes implementaciones de $SpMV$ exclusivamente en formato CSR con el objetivo de elegir el mejor método para cada caso. En los estudios preliminares hallaron que la implementación *vector* comentada anteriormente era la que obtenía los mejores resultados en más casos pero que la nueva implementación *merge* [12] la superaba en otros. Esta implementación soluciona el problema de *vector* que obtiene malos resultados cuando existen muchas filas con pocos no-ceros debido a que asigna un número de hilos fijo (originalmente un *warp*) por fila. La idea de *merge* consiste en intentar balancear la carga de trabajo de los hilos independiente de las filas con el objetivo de que el desempeño de este $SpMV$ escale con nnz.

2.2.2. Resolución de Sistema Triangular Disperso ($SpTrSv$)

La resolución de sistemas triangulares dispersos ($SpTrSv$ por su sigla en inglés) es otra operación común en el álgebra lineal. Como su nombre lo indica esta operación consiste en resolver un sistema de ecuaciones lineales triangular disperso[1]. A efectos de este trabajo vemos a los sistemas representados por la

[1] Un sistema $Ax = b$ donde la matriz A es triangular y dispersa.

ecuación:

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

donde L es una matriz triangular inferior[2], x es el vector de incógnitas y b el vector de valores independientes.

La $SpTrSv$ es una de las piedras fundamentales del álgebra lineal dispersa siendo una de las partes más importantes en diversas aplicaciones. Dos ejemplos muy comunes son en la última etapa de los *solvers* directos de sistemas lineales dispersos (donde se resuelven los dos sistemas surgidos de la descomposición LU) y como parte de los métodos iterativos que utilizan precondicionamiento (donde los precondicionadores son generalmente factorizaciones incompletas triangulares).

En el Algoritmo 1 se presenta el pseudocódigo para resolver un sistema triangular disperso de manera secuencial para una matriz en formato CSR. La implementación más natural en dicho formato consiste en la implementación de una sustitución hacia delante. Esta implementación recibe como parámetro los tres vectores que representan a la matriz de coeficientes así como la cantidad de filas. Este procesamiento itera para todas las filas en orden. Para cada fila, en la línea 10, se sustituyen todas las incógnitas correspondientes a filas anteriores y se acumulan en `left_sum`. Finalmente, en la línea 12, se despeja el valor de la incógnita restando el acumulado al valor correspondiente del vector b y dividiendo por el coeficiente de la diagonal.

Algoritmo 1 $SpTrSv$ (Secuencial) para $Lx = b$ en formato CSR

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

2.2.3. $SpTrSv$ paralelo

A diferencia de otras operaciones como la $SpMV$, la paralelización de la $SpTrSv$ no resulta intuitiva. Esto se debe a que existen dependencias entre las

[2]Se podría plantear un problema análogo con U, una matriz triangular superior.

filas por lo tanto procesar distintas filas concurrentemente implica sincronizaciones entre las unidades de cómputo que no existen en la $SpMV$. En concreto estas dependencias surgen del patrón de dispersión de la matriz dispersa y por tanto es necesario distribuir los datos entre las unidades de procesamiento (hilos) que permita ejecutar la mayor cantidad de trabajo en paralelo.

Existe una enorme variabilidad dado que cualquier fila i puede (o no) depender de alguna otra fila j con $j < i$ para poder computarse. La forma más sencilla de paralelizar el algoritmo $SpTrSv$ es, al momento de realizar el cálculo de una fila, asignar un hilo para cada elemento de la fila. La tarea de cada hilo es calcular el valor de la variable que le corresponde, realizar el producto del coeficiente con la incógnita, y acumular el valor en una variable para que sea utilizado al momento de resolver la ecuación correspondiente a la fila. En el Algoritmo 2 se muestra un ejemplo del método descrito.

Algoritmo 2 $SpTrSv$ (data-parallel $SpTrSv$) para $Lx = b$ en formato CSR

```
1  In:   n > 0, row_ptr[], val[], col_idx[], b[]
2  Out:  x[]
3
4  for i = 0 to n - 1 in parallel do
5      loc(x[i])
6      sync()         //Sincroniza todos los hilos
7      left_sum = 0   //Una variable left_sum por hilo
8
9      for j = row_ptr[i] to row_ptr[i+1]-2 in parallel do
10         local_sum = x[j]*val[j] //Espera si x[j] se encuentra bloqueado
11     end for
12     left_sum = reduce(local_sum) //Requiere que todos los j hayan
           terminado
13
14     x[i] = (b[i] - left_sum) / val[i]
15     unlock(x[i])
16 end for
```

Como se puede observar el Algoritmo 2 sigue una idea similar al Algoritmo 1. Debido a que las filas no dependen necesariamente de todas las anteriores se paraleliza también la resolución de las filas pero al comienzo se garantiza que la incógnita correspondiente a la fila i estará bloqueada hasta que la fila i termine su ejecución. Además de esto el otro cambio es que cada fila ejecuta la sustitución en paralelo y por ende debe hacer una reducción antes de resolver su incógnita.

2.3. Graphics Processing Units (GPUs)

Las unidades de procesamiento gráfico o en inglés *Graphics Processing Units* conocidas popularmente como GPU son una de las arquitecturas de hardware más utilizadas en el campo de HPC. Estas plataformas cuentan con grandes

cantidades de procesadores "livianos" (son básicamente una ALU, no cuentan con registros o caché propios), que permiten hacer un elevado número de operaciones en paralelo. A diferencia de las CPU que cuentan con muchos transistores dedicados a otras tareas (predicción de *branches*, cache de datos, etc.), pueden dedicar una mayor cantidad al cómputo permitiendo que sea una plataforma con gran poder de cálculo.

Si bien originalmente las GPUs fueron diseñadas para enfocarse en el trabajo relacionado con la representación gráfica, en las últimas dos décadas fueron adoptadas para cómputo de propósito general. Esto se acentuó con el desarrollo, por parte de la empresa *Nvidia*, de *CUDA* (Arquitectura Unificada de Dispositivos de Cómputo por su sigla en inglés). *CUDA* es una plataforma de cómputo paralelo y API que permite desarrollar cómputo general en GPUs de *Nvidia* utilizando una variación del lenguaje C (*CUDA* C).

Los programas en *CUDA* ejecutan parte de su código en la CPU (a la que se llama *host*) y parte en la o las GPU (*device*). Su naturaleza queda representada en el propio código para que permita generalizar un programa en C como un programa en *CUDA* que no tiene código en GPU. Un programa *CUDA* no puede ser compilado por un compilador de C, para esto *Nvidia* brinda su compilador propio, NVCC, que compila las partes de código que ejecutarán en GPU (identificadas por la palabra reservada kernel) y delega en el compilador estándar de C el código del *host*.

Los *kernels* en *CUDA* especifican un código que será ejecutado por todos los hilos invocados en paralelo, la variedad en el comportamiento entre los hilos depende únicamente del uso de sus identificadores para calcular o acceder a los datos. Por lo anterior, *CUDA* es una instancia del estilo de programación para sistemas de computación masivamente paralelo conocido como SPMD (*Single Program Multiple Data*) [8].

La ejecución comienza en el *host* idénticamente a un programa en C con la excepción de los *kernels*. Cuando el *host* lanza la ejecución de un *kernel* se genera una grilla de hilos con una organización de dos niveles. La grilla está organizada como un arreglo de bloques de hilos, estos bloques tienen todos la misma cantidad fija de hilos. Tanto la cantidad de hilos por bloque como el número de bloques son definidos al momento de invocar el *kernel*. Los bloques pueden contener, como máximo, 1024 hilos cada uno que se agrupan de a 32 en grupos llamados *warps*. Además cuentan con una memoria propia y rápida que comparten todos los hilos del bloque llamada *memoria compartida*.

Cada hilo puede ser identificado en la grilla con dos valores, el identificador de su bloque y su identificador de hilo dentro del bloque. Estos identificadores están disponibles en las variables `blockIdx` y `threadIdx` respectivamente. Además la cantidad de hilos en cada bloque puede obtenerse utilizando la variable blockDim. Tanto la grilla como los bloques son, conceptualmente, tridimensionales por lo que todas estas variables tienen valores en las distintas direcciones identificadas como `x`, `y` y `z`. La Figura 2.4 muestra un esquema del modelo de ejecución donde tanto la grilla como los bloques son unidimensionales y la dimensión en el eje x es 256 [26].

A nivel de *hardware* las GPU agrupan los procesadores en *Streaming Mul-*

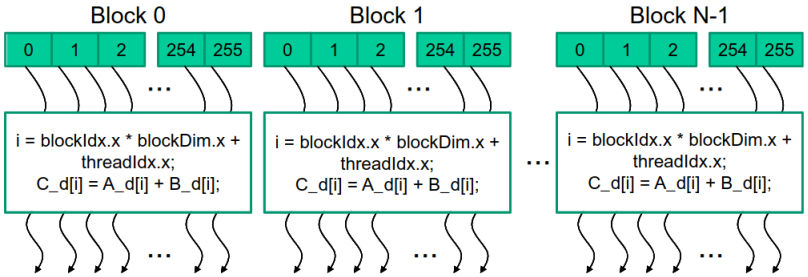

Figura 2.4: Modelo de ejecución de dos niveles de *CUDA*. Todos los hilos ejecutan el mismo código. Tomado de [26]

tiprocessors (SMs) que comparten registros y distintas memorias (cache L1, memoria compartida, cache constante y cache de texturas). Dependiendo de la arquitectura cuentan con uno o maś *warp scheduler*[3] y por cada uno de estos un Program Counter (PC). Estos SMs son a su vez agrupados en clusters (TPCs). La cantidad de procesadores por SM y SMs por TPC varía según la arquitectura específica. Finalmente, las tarjetas cuentan con una memoria RAM llamada memoria global a la que todos los SM pueden acceder. La latencia de esta memoria es significativa y, dado que las *caches* de GPU no son lo suficientemente grandes, para ocultar el costo de acceso a memoria se remueve de la ejecución al *warp* en cuestión hasta que esté disponible el resultado. Un diagrama de ejemplo de la arquitectura G80 se presenta en la Figura 2.5. Puede verse que se cuenta con 8 TPCs compuestos, cada uno, por dos SMs agrupados.

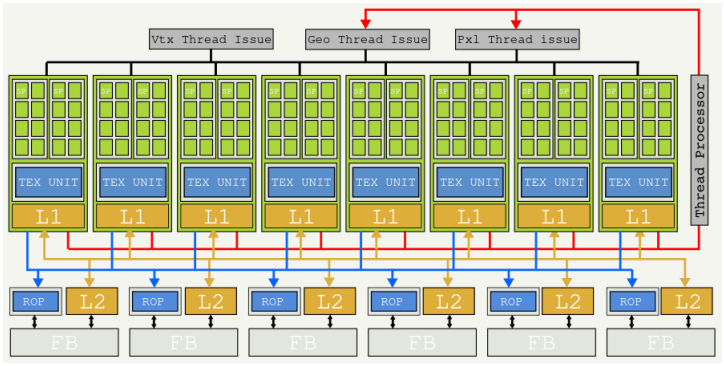

Figura 2.5: Diagrama de la arquitectura G80 de *CUDA* extraído de la documentación de *Nvidia*.

Como se mencionó anteriormente la cantidad de procesadores genera un gran

[3]Unidad que gestiona la ejecución de un *warp* como una unidad aunque ejecute en más de un ciclo de reloj.

poder de cálculo si es posible mantener los procesadores siempre ocupados. En este sentido el acceso a memoria funciona como cuello de botella. Las GPUs cuentan con varias memorias rápidas y pequeñas (por ejemplo memoria de texturas) y una memoria RAM principal a la que se llama memoria global. Debido a que, comparativamente con la CPU, no se cuenta con tanto cache para reducir la latencia al acceso a memoria este es costoso.

Para ocultar la latencia la estrategia principal es que cuando un conjunto de hilos hace acceso a memoria se suspende su ejecución hasta que el resultado está disponible dejando los procesadores libres para otros hilos que estén listos para ejecutar. Esto genera que virtualmente se cuente con más hilos y bloques de los que ejecutan en un momento dado en la GPU, estos últimos son conocidos como bloques activos. Debido a que la planificación está hecha por *hardware* esto permite que los cambios de contexto sean prácticamente sin costo y transparentes para el programador.

A nivel de programación también se puede mejorar notablemente el costo de acceso a memoria si se explota el ancho de banda del bus de memoria de la tarjeta. Esto se logra si hilos de índices contiguos acceden a datos contiguos en memoria ya que la arquitectura es capaz de agrupar dichos accesos en una menor cantidad de transacciones de memoria, lo cual en $CUDA$ se llama *acceso coalesced*.

Si bien existen otros lenguajes y arquitecturas en tarjetas gráficas el impulso tomado por $Nvidia$ gracias a $CUDA$ lo ha convertido en un estándar de facto en HPC. Las tarjetas gráficas son especialmente utilizadas en el área de los métodos numéricos debido a la gran cantidad de cálculo requerido. La estrategia general para utilizar de forma eficiente la GPU consiste en dividir el problema de forma relativamente independiente. Esto se debe a que la única forma de compartir datos globalmente es el acceso a la memoria principal el cual es costoso y, a diferencia de la CPU, no está optimizado.

2.3.1. Sincronización en GPU

Las GPUs de $Nvidia$ trabajan, en general, bajo el paradigma SIMT (*Single Instruction Multiple Thread*. Esto modelo de ejecución combina el paradigma SIMD (*Single Instruction Multiple Data*) con *multithreading*[4] Es diferente de SPMD ya que, en este último, todas las instrucciones en todos los "hilos" se ejecutan a la vez.

En las arquitecturas anteriores a Volta (compute capabilites[5] 6.x o inferio-

[4]Es una característica que permite que dos o más hilos de instrucciones ejecuten de forma independiente mientras comparten los mismos recursos de proceso. Un subproceso es una secuencia autónoma de instrucciones que se puede ejecutar en paralelo con otros subprocesos que forman parte del mismo proceso raíz.

[5]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.

res [2]) la sincronización nativa era relativamente acotada. A nivel de *warp* esta era brindada de manera implícita dado que se contaba con un único Program Counter (PC) por *warp scheduler* lo que forzaba a que por ciclo de reloj se despachara una única instrucción a cada *warp*. Esto generaba que los hilos de un *warp* siempre estuvieran ejecutando la misma instrucción incluso cuando se dividían por una *branch* lógica. Estas bifurcaciones (llamadas *thread divergences*) se resolvían serializando la ejecución de las *branches* y deshabilitando a los hilos que no debían ejecutar cada una.

Esta solución es costosa puesto que por cada bifurcación lógica se debe serializar la ejecución. En este sentido los códigos en GPU buscan siempre evitar bifurcaciones a la interna de un *warp* y por tanto no son capaces de hacer uso de paralelismo fino intra-*warp*.

A nivel de bloque, sin embargo, los hilos no se encontraban coordinados en todo momento y por tanto *CUDA* brindaba una instrucción para sincronizarlos explícitamente. La función `__syncthreads()` era el único mecanismo de sincronización nativo. Dado que esta barrera es de una granularidad relativamente gruesa muchas veces son necesarias sincronizaciones más finas que el programador debía implementar.

Un cambio importante en las arquitecturas modernas de *Nvidia* fue la incorporación de PCs para todos los *cuda cores* [6]. Como se explicó anteriormente la existencia de un PC por *warp scheduler* era la que permitía la sincronización implícita a nivel de *warp*. Estos cambios permiten a cada hilo de un *warp* ejecutar de manera independiente. Por otro lado esto generó la necesidad de cambios en la sincronización incorporando nuevas herramientas nativas. En las nuevas arquitecturas existen diversas herramientas de sincronización.

Además de la sincronización a nivel de bloque mencionada anteriormente, en las nuevas versiones de *CUDA* se incorporó la herramienta `__syncwarp` para sincronizar a nivel de *warp*, una operación que sustituye a la sincronización implícita. Además se incorpora la posibilidad de brindar como argumento una maścara que permite elegir qué hilos serán detenidos en dicha sincronización.

El principal cambio de paradigma con respecto a la sincronización fue generado con la incorporación de los *cooperative groups*. Estos fueron incorporados en *CUDA* 9 y posteriormente ampliados en *CUDA* 11. Esta API permite agrupar hilos en grupos de tamaño arbitrario y sin restricciones sobre el *warp* o bloque al que pertenecen. La motivación detrás de los cambios es, según Harris y Perelygin [21], ayudar a que el código sea menos frágil, reducir las restricciones a la optimización del compilador y mejorar la compatibilidad con versiones posteriores. Además, facilitan la modularización. Sin embargo, esto implica que códigos desarrollados en arquitecturas anteriores deben adaptarse y añadir sincronizaciones para evitar condiciones de carrera[7] entre hilos de un mismo *warp*.

Además de poder agrupar hilos de distintos bloques o *warps*, los *cooperative*

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

[7] 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.

groups también permiten una mayor granularidad a la hora de sincronizar. Al no existir la sincronización implícita y contar con *independent thread scheduling* es posible partir el *warp* y sincronizarlo de manera independiente. A diferencia de la serialización que se hacía en el pasado cuando se subdivide a un *warp* en grupos que trabajan por separado y se sincronizan de manera independiente, está garantizado que todas las partes ejecutarán en algún momento. Esto último permite hacer optimizaciones que en el pasado no eran posibles sin arriesgarse a *deadlocks* o condiciones de carrera.

Además se sustituyeron las operaciones conocidas como *warp intrinsics* tales como *shuffle* (permite intercambiar valores de variables entre hilos de un *warp*) o funciones de votación (permiten evaluar una condición para todos los hilos del *warp* y tomar una decisión conjunta). Estas operaciones funcionan en la actualidad a nivel de grupo aunque las versiones a nivel de *warp* siguen funcionando sustituídas por nuevas versiones que utilizan el sufijo sync luego del nombre de la operación(e.g. `shuffle_down_sync`).

2.4. $SpTrSv$ masivamente paralelo

La operación $SpTrSv$ es compleja de paralelizar debido a las dependencias de datos que presenta, sin embargo, como la matriz A del sistema es dispersa pueden existir, a diferencia de la operación $TrSv$ densa, muchas filas independientes entre sí. Si bien esto está asociado al patrón de dispersión de cada matriz particular puede ser explotado para sacar partido de la GPU a la hora de paralelizar la $SpTrSv$.

En muchos casos las aplicaciones de la $SpTrSv$ implican la resolución de varios sistemas triangulares dispersos en los que se repite la matriz A (o al menos el patrón de dispersión) y varía únicamente el vector del lado derecho. Teniendo esto en cuenta se puede reutilizar la parte "simbólica" del trabajo (es decir lo relacionado al cálculo de dependencias y/o definición de un orden de procesamiento) y solamente repetir lo relativo al cálculo de los valores. En este sentido, se puede dividir la resolución en dos partes, una primera etapa de análisis simbólico que depende únicamente del patrón de dispersión (los índices) de la matriz A y una segunda etapa de resolución que depende del vector b. El trabajo de O. Wing y J. W. Huang [36] dio los primeros pasos en esta línea.

Una primera estrategia para paralelizar la $SpTrSv$ consiste en calcular en la etapa de análisis, para cada fila, todas las incógnitas de las que depende. En la etapa de resolución se pueden procesar en paralelo todas las filas que no dependan de ninguna otra y, una vez terminadas, se pueden sustituir las incógnitas y eliminar las dependencias ya resueltas antes de repetir.

Una mejora de esta idea es interpretar las dependencias como aristas en un grafo dirigido sin ciclos (DAG, por su sigla en inglés) con A su matriz de adyacencia. La cual se basa en una etapa de preprocesamiento para agregar las incógnitas del sistema triangular en conjuntos de niveles. Esto determina un orden de ejecución para la solución del sistema, donde los conjuntos de niveles deben procesarse secuencialmente, mientras que las incógnitas que pertenecen a

un mismo conjunto de niveles pueden resolverse en paralelo. Uno de los trabajos más relevantes en esta línea fue el artículo de Saad y Schultz [32]. Si bien el trabajo se centra en la mejora del gradiente conjugado precondicionado (PCG, por su sigla en inglés) parte de la etapa de implementación implicó la paralelización de sistemas triangulares dispersos. En particular, contribuyeron en las líneas de determinar la cantidad de unidades de procesamiento necesarias y mejorar la paralelización de este sistema.

Este paradigma fue utilizado por Naumov [28] en su implementación para GPUs que fue posteriormente integrada a la biblioteca CUSPARSE. En la implementación de Naumov la etapa de análisis es la encargada de crear la estructura de niveles. Para esto computa un orden topológico para el grafo representado por A utilizando una variación del algoritmo de Kahn [25]. La segunda etapa es la encargada de calcular la solución. El principal problema de esta implementación es que la etapa de resolución requiere sincronizar cada vez que termina un nivel para evitar problemas de concurrencia. Esto implica no solo un costo de sincronización al lanzar un nuevo *kernel* cada vez sino que fuerza a esperar a las filas de un nivel por todas las del nivel anterior aunque solo dependa de unas pocas.

Otro paradigma diferente para la resolución de sistemas triangulares es el uso de algoritmos $Sync - Free$. El paradigma $Sync - Free$ toma un enfoque distinto al anterior y no utiliza un preprocesamiento o un orden estático. La idea general de esta estrategia consiste en que cada vez que un hilo esté libre vaya a buscar tareas de un conjunto, evitando el costo de sincronización. En este caso las tareas equivalen, por ejemplo, a la resolución de una ecuación. El primer trabajo en esta área fue en los 80s por George et al. [20], más recientemente Liu et al. [27] presentaron una implementación para GPU utilizando el formato CSC. El pseudocódigo de el algoritmo propuesto en este trabajo puede verse en el Algoritmo 3.

Este algoritmo hace un preprocesamiento (líneas 3 a 7) para calcular de cuántas entradas depende cada fila y lo almacena en una variable, `in_degree`, de tamaño dim. Una vez con este dato, es sencillo que el hilo que procesa la fila i haga una espera activa preguntando si se han solucionado todas las dependencias. Para esto cada hilo al resolver la incógnita que le corresponde actualiza, utilizando suma atómica, la cantidad de dependencias que le faltan (líneas 19 o 22 dependiendo si puede hacerse a través de memoria compartida o no).

Una forma distinta a la propuesta por Naumov para obtener una estructura de niveles se basa en la profundidad de los nodos del DAG asociado. Wing y Huang [36] definieron la profundidad (*depth*) como la distancia máxima a la raíz de este grafo.

Dufrechou y Ezzatti [15] utilizaron esta definición para construir la estructura de niveles de una matriz dispersa. Esta estructura es utilizada para generar un ordenamiento en el que serán procesadas las filas pero la etapa de cálculo sigue un paradigma $Sync - Free$. Posteriormente, los mismos autores combinaron las estrategias de resolución $Sync - Free$ con el análisis basado en la profundidad para definir diferentes métodos para resolver la operación $SpTrSv$.

Algoritmo 3 *SpTrSv Sync − Free* para $Lx = b$ para formato CSC propuesta por Liu et al.

```
MALLOC(*d_left_sum, *s_left_sum, *d_in_degree, *s_in_degree, n)
MEMSET(*d_left_sum, *s_left_sum, *d_in_degree, *s_in_degree, 0)
function PREPROCESSING-STAGE()
    for i = 0 to nnz - 1 in parallel do
        ATOMIC-ADD(&d_in_degree[row_idx[i]], 1)
    end for
end function
function SOLVING-STAGE()
    th = SET()
    for i = 0 to n - 1 in parallel do
        while s_in_degree[i] + 1 <> d_in_degree[i] do
            // busy wait
        end while
        x[i] = (b[i] - d_left_sum[i] - s_left_sum[i]) / val[col_ptr[i]]
        for j = col_ptr[i] + 1 to col_ptr[i+1] - 1 in parallel do
            rid = row_idx[j]
            if ( rid < i + th - i%th ) then
                ATOMIC-ADD(&s_left_sum[rid], val[j] * x[i])
                ATOMIC-ADD(&s_in_degree[rid], 1)
            else
                ATOMIC-ADD(&d_left_sum[rid], val[j] * x[i])
                ATOMIC-ADD(&d_in_degree[rid], 1)
            end if
        end for
    end for
end function
FREE(*d_left_sum, *s_left_sum, *d_in_degree, *s_in_degree)
```

Capítulo 3

Etapa de Análisis

En este capítulo se aborda la etapa de análisis simbólico del solver de sistemas lineales triangulares dispersos, la operación $SpTrSv$.

En la Sección 3.1, se presentan las plataformas de ejecución utilizadas en el resto del documento y las matrices para las experimentaciones preliminares. En la Sección 3.2, se analiza en profundidad la propuesta de Dufrechou y Ezzatti [15][16] para resolver sistemas triangulares dispersos en GPUs. En la Sección 3.3, se describen las nuevas rutinas propuestas en el marco del actual proyecto. Se presentan dos enfoques para el análisis, uno híbrido (ejecuta en CPU-GPU) y otro que ejecuta enteramente en la GPU. A diferencia del primer enfoque la rutina en GPU no paraleliza una etapa concreta sino que aborda el análisis en su conjunto. Finalmente, en la Sección 3.4, se evalúan experimentalmente las propuestas presentadas. Inicialmente se brinda una evaluación a cada una de las modificaciones de manera incremental sobre un conjunto acotado de matrices de prueba y finalmente se comparan las tres rutinas sobre un conjunto mayor de matrices.

3.1. Contexto de experimentación

Para las evaluaciones preliminares se eligió un conjunto relativamente pequeño de 22 matrices representativo de la colección Suitesparse [14] (anteriormente conocida como la colección de la Universidad de Florida). Estas matrices tienen una gran variedad en sus características tanto en dimensión (Dim), cantidad de no-ceros (NNZ), densidad (NNZ/Dim), así como cantidad de niveles (Lev) [1]. Las principales características de las matrices utilizadas se presentan en la Tabla 3.1.

En cuanto a las plataformas de hardware utilizadas todas las experimentaciones fueron ejecutadas en un equipo que incluye una CPU `Intel(R) Core(TM) i7-6700 CPU @ 3,40GHz` con $64kB$ de caché L1, $256kB$ de caché L2 y 8MB

[1] Lev representa la cantidad de niveles que posee la matriz, es decir, la cantidad de nodos del camino mas largo dentro del grafo dependencias.

	Id	Dim	Lev	NNZ
chipcool0	1	20082	534	150616
hollywood-2009	2	1139905	82735	57515616
nlpkkt160	3	8345600	2	118931856
road_central	4	14081816	59	31015229
road_usa	5	23947347	77	52801659
ship_003	6	121728	4367	4103881
webbase-1M	7	1000005	512	2348442
wiki-Talk	8	2394385	515	3072221
crankseg_1	9	52804	4056	5333507
cant	10	62451	2397	2034917
cvxbqp1	11	50000	25	199984
ncvxqp3	12	75000	25	299982
luxembourg_osm	13	114599	427	234265
rajat29	14	643994	44	3146703
bayer01	15	57735	6	195628
circuit5M_dc	16	3523317	275	10631719
exdata_1	17	6001	1501	1137751
CO	18	221119	216817	3943588
2D_54019_highK	19	54019	1368	526117
cond-mat	20	16726	92	64320
TSOPF_RS_b300_c2	21	28338	112	2914479
LeGresley_87936	22	87936	57	337449

Tabla 3.1: Grupo de matrices utilizado para las pruebas preliminares con sus principales características

de caché L3. En cuanto a las ejecuciones en GPU estas fueron hechas sobre dos tarjetas gráficas, RTX 2080 Ti (compute capabilites 7.5) y RTX 3090 Ti (compute capabilites 8.). La primera es de arquitectura Turing y cuenta con 4352 CUDA cores y 11GB de memoria GDDR6. La segunda, por su parte, es de la arquitectura Ampere y cuenta con 10752 CUDA cores y 24GB de memoria GDDR6X.

3.2. Rutina original para el análisis

El enfoque propuesto en [16] incluye tres estrategias para la resolución de sistemas lineales dispersos triangulares ($solver_{smpl}$, $solver_{ord}$ y $solver_{mr}$). Mientras que la rutina $solver_{smpl}$ no requiere etapa de análisis, las rutinas $solver_{ord}$ y $solver_{mr}$ se emplean dos enfoques de análisis diferente: Anl_{ord} y Anl_{mr} respectivamente. Ambas rutinas se encuentran implementadas casi completamente en CPU. En cuanto al diseño, ambas rutinas pueden ser expresadas como una combinación de 8 tareas básicas que se presentan a continuación de las cuales únicamente T2 ejecuta en GPU.

- **T1-Init:** Se reserva memoria y calculan los datos necesarios para la eje-

cución del kernel.

- **T2-Kernel:** Calcula el nivel de cada fila. Es la única etapa que se ejecuta en paralelo. La implementación, siguiendo la propuesta [15], es muy similar al método del solver, solo que en lugar de calcular el valor correspondiente a la incógnita de cada fila, calcula su nivel como el máximo entre las filas de las que depende más 1.
- **T3-Copy to CPU:** Transfiere a CPU la estructura que mantiene el nivel de cada fila dado que el resto de la rutina se ejecuta íntegramente en CPU.
- **T4-Max:** Calcula el nivel máximo de la matriz.
- **T5-Order and size calc:** Ordena las filas en función del nivel (si la fila i tiene menor nivel que j entonces será ordenada después) y tamaño (filas con cantidades similares de elementos se asignan juntas, 0, 1, 2, 4, 8, 16 o 32 sin contar la diagonal).
- **T6-Warp info:** Asigna filas a *warps* en orden, filas de menos de 16 noceros (sin contar la diagonal) son agrupadas.
- **T7-Copy to device:** Transfiere los vectores calculados a la GPU para ser utilizados por el solver.
- **T8-Free** Libera la memoria que se reservó en CPU y las estructuras auxiliares en GPU.

En particular, la rutina Anl_{ord} ejecuta las tareas T1 a T5 mientras que la rutina Anl_{mr} ejecuta las 8 tareas. Por tanto, en lo que resta de este capítulo, se analiza esta última variante que permite entender ambas. El Algoritmo 4 presenta dicha rutina.

El análisis es costoso en cuanto a tiempo de ejecución puesto que, en definitiva, incluye una resolución simbólica del sistema triangular. Es importante recordar que, en la medida que se efectúen varias resoluciones con la misma matriz, se puede reutilizar el resultado de un único análisis. En otras palabras, su peso relativo aumenta si se ejecutan pocas instancias de resolución. En definitiva es un costo base que acota el mínimo tiempo de ejecución que requerirá el algoritmo de resolución, por lo tanto una reducción en esta etapa impacta directamente en el tiempo global.

En la Tabla 3.2 se resumen los tiempos de ejecución correspondientes a la rutina de análisis original (también llamada Anl_{og}) para las diferentes matrices estudiadas. Los detalles del entorno de evaluación (plataforma y matrices) se presentan en la Sección 3.1.

En los resultados presentados en la Tabla 3.2 puede verse que, en la mayoría de los casos, las tres etapas con mayor peso en el costo total son: T2 (kernel), T5 (cálculo del orden) y T6 (asignación de filas a warps). En particular, en todos los casos presentados en la tabla, la suma de estas tres etapas supera el 80 % del costo total. La etapa de ordenamiento es, indudablemente, la más

Algoritmo 4 Pesudocódigo del análisis original

```
In:   n > 0, row_ptr[], val[], col_idx[], b[]
Out:  iorder[], ibase_row[], warp_vect_size[]

INIT();
Kernel_analysis(csr_matrix, aux_vect, num_rows, out: lev_vect)
if( lne == 0 )
 wrp = ATOMIC-ADD(row_ctr, 1)
end if
for row = 1 to n do
 lev = vect_niveles[row]
 nnz = nnz_row(csr_matrix, row) - 1
 vect_size = calc_vs(nnz_row)
 ivects[7*lev + vect_size]++; ilevels[lev]++
end for
exclusive_scan(ivects, 7 * nLevs_dfr)
for row = 1 to n do
 lev = vect_niveles[row];
 nnz = nnz_row(csr_matrix, row) - 1
 vect_size = calc_vs(nnz_row)
 iorder[ivects[7*lev + vect_size]] = row
 ivect_size[ivects[7 * lev + vect_size]] = if (vect_size == 6) 0 else
     2^vect_size
 ivects[7 * lev + vect_size]
end for
n_warp=1; warp_rows=1
for row = 2 to n do
 if( is_different(row, row-1) || warp_full(n_warp))
 warp_rows = 1
 n_warp++
 else
 warp_rows++
 end if
end for
MALLOC(ibase_row, ivect_size_warp, iwarp_lev)
warp_rows = 1; warp = 1
for row = 2 to n do
 if(is_different(row, row-1) || warp_full(n_warp))
    ibase_row[warp] = row
        ivect_size_warp[warp] = ivect_size[row]
        iwarp_lev[warp] = dfr_analysis_info[iorder[row]] - 1
        warp_rows = 1
        warp++
    else
     warp_rows++
    end if
end for
ibase_row[warp] = n
COPY_TO_DEVICE()
FREE()
```

Matriz	T1	T2	T3	T4	T5	T6	T7	T8
Turing								
1	0,05	0,68	0,03	0,01	1,11	0,09	0,13	0,01
2	0,26	105,68	1,55	0,33	52,32	10,32	2,22	0,44
3	0,38	9,52	5,44	2,02	238,02	28,74	11,65	1,98
4	0,57	21,59	9,07	3,41	219,84	73,75	12,06	2,83
5	0,76	52,76	14,32	5,72	364,65	132,33	20,15	3,02
6	0,06	3,79	0,10	0,03	6,75	0,78	0,36	0,01
7	0,25	2,88	1,41	0,29	15,08	4,26	1,16	0,33
8	0,29	2,76	2,03	0,60	15,41	8,72	2,34	0,67
9	0,05	5,06	0,05	0,02	2,97	0,36	0,16	0,01
10	0,05	3,82	0,06	0,02	3,53	0,40	0,18	0,01
11	0,05	0,11	0,06	0,02	1,24	0,17	0,09	0,01
12	0,15	0,12	0,07	0,02	1,55	0,26	0,12	0,01
13	0,14	1,44	0,09	0,03	1,79	0,45	0,15	0,01
14	0,24	1,94	0,87	0,16	17,84	2,42	0,97	0,33
15	0,05	0,08	0,08	0,03	1,04	0,20	0,10	0,01
16	0,32	9,57	2,83	0,99	38,71	16,49	3,54	0,91
17	0,06	1,61	0,02	0,00	0,14	0,02	0,04	0,01
18	0,16	160,57	0,19	0,06	13,86	1,12	0,85	0,02
19	0,05	1,57	0,05	0,02	3,02	0,28	0,11	0,01
20	0,04	0,10	0,03	0,00	0,65	0,07	0,05	0,01
21	0,05	0,50	0,05	0,01	1,03	0,10	0,09	0,01
22	0,14	0,35	0,10	0,04	2,61	0,33	0,13	0,01
Ampere								
1	0,03	0,58	0,03	0,01	1,09	0,09	0,16	0,01
2	0,27	67,42	1,35	0,28	51,83	9,73	3,46	0,91
3	0,37	5,85	6,06	2,01	238,29	28,39	19,10	2,41
4	0,49	14,08	9,71	3,37	220,20	73,72	19,69	2,83
5	0,67	32,77	16,33	6,73	371,42	132,59	32,95	3,26
6	0,05	2,52	0,13	0,03	6,95	0,79	0,51	0,01
7	0,26	1,53	1,18	0,28	14,89	3,67	1,78	0,69
8	0,29	1,99	2,03	0,58	15,03	8,50	3,73	1,02
9	0,04	3,12	0,07	0,02	2,99	0,36	0,24	0,01
10	0,04	2,72	0,08	0,02	3,50	0,40	0,27	0,01
11	0,03	0,09	0,08	0,02	1,14	0,17	0,13	0,01
12	0,14	0,10	0,11	0,02	1,50	0,26	0,17	0,01
13	0,13	0,67	0,13	0,03	1,75	0,43	0,22	0,01
14	0,25	1,30	0,92	0,19	17,40	2,49	1,45	0,33
15	0,04	0,07	0,10	0,03	0,94	0,19	0,14	0,01
16	0,32	6,16	2,99	0,86	38,47	16,33	5,60	1,25
17	0,05	1,06	0,02	0,00	0,14	0,02	0,05	0,01
18	0,14	113,34	0,25	0,06	14,12	1,19	1,21	0,11
19	0,05	1,00	0,07	0,02	3,02	0,28	0,17	0,01
20	0,04	0,09	0,04	0,01	0,55	0,07	0,06	0,01
21	0,05	0,37	0,04	0,01	0,97	0,10	0,12	0,01
22	0,15	0,19	0,11	0,03	2,56	0,32	0,20	0,01

Tabla 3.2: Tiempo de ejecución (en ms) de la rutina original de análisis (Anl_{og}) separado por etapas en ambas plataformas de cómputo.

costosa con un promedio del 54 %. Luego, la segunda etapa en costo relativo, es la ejecución del kernel (promedio 26 %) y, por último, la asignación de filas a *warps* (aproximadamente 11 %). La Figura 3.1 presenta un esquema que muestra las salidas relevantes de cada una de estas etapas.

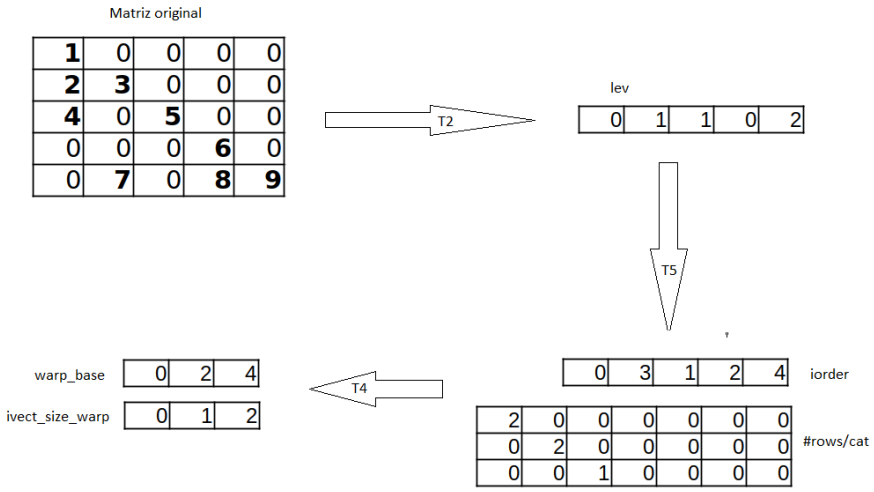

Figura 3.1: Esquema de salidas relevantes de las etapas T2, T5 y T6 para una matriz de ejemplo.

3.3. Propuestas

En esta sección se presentan dos enfoques para la optimización a la rutina de análisis. En primer lugar, en el apartado 3.3.1 se plantea una estrategia de paralelización centrada en la división de la rutina en etapas auto-contenidas y la paralelización de los principales cuellos de botella. Este enfoque da como resultado una rutina híbrida que ejecuta parte de su cómputo en GPU y parte en CPU. Dicho enfoque se encuentra limitado debido a que la rutina original está planteada de forma secuencial. Además el enfoque híbrido implica un costo debido a la transferencia de datos en etapas intermedias entre CPU y GPU. Por otro lado en el apartado 3.3.2 se plantea un paradigma distinto en el que se aborda el problema general y se implementa una rutina que realiza el análisis enteramente en la GPU.

3.3.1. Enfoque híbrido

En este apartado se presentan las principales optimizaciones a la rutina de análisis original. En particular se presentan mejoras a las tres etapas más costo-

sas en tiempo de ejecución aplicando el uso de paralelismo y nuevas herramientas brindadas por arquitecturas modernas de las GPUs de *Nvidia*. De las tres etapas abordadas T2 y T5 son parte de ambas rutinas de análisis: Anl_{mr} y Anl_{ord}, mientras que la última, T6, solo afecta a Anl_{mr}.

Mejora de T5, Anl_{base}

Como se explicó previamente, la etapa de ordenamiento tiene como objetivo generar un vector de permutaciones. Este vector reordena las matrices en un orden creciente según la estructura de niveles, asegurando que, dentro de cada nivel, las filas con cantidades similares de elementos se encuentran contiguas. Se definen 7 clases de tamaño para las filas en función de la cantidad de no-ceros según la Ecuación (3.1). Se consideran estos tamaños para las clases debido a que con estas son definidas con el propósito de agrupar diferentes filas pertenecientes a la misma clase, ejecutando todas ellas en un mismo warp, por lo que es conveniente que estas sean potencias de 2 (expresado como logaritmo en la ecuación). Es por esto que, además de generar el vector de permutaciones, T5 clasifica las filas en tantas categorías como el número de niveles por siete (el número de clases de largo de fila). Por ende, esta etapa recibe como entrada el vector de niveles proveniente de T2 y el número de niveles calculado por T4.

$$\begin{aligned} clase\_tam(fila) &= 0 && si \quad nnz(fila) - 1 = 0 \\ &= 6 && si \quad nnz(fila) - 1 > 32 \\ &= \lceil \log_2(nnz(fila) - 1) \rceil && si \quad 0 < nnz(fila) - 1 \leq 32 \end{aligned} \tag{3.1}$$

donde el -1 surge de que el elemento de la diagonal no afecta el *loop* principal y por ende no es relevante para el agrupamiento.

La etapa de ordenamiento consiste, a grandes rasgos, en dos iteraciones sobre todas las filas, una primera en la que se hace un cálculo simbólico para contar cuántas filas hay en cada categoría y una segunda en la que se construyen los vectores de salida. El vector en el que se almacena la cantidad de filas por categoría se genera ordenado primero por nivel y luego por la clase de tamaño de la fila. Posteriormente se ejecuta una suma prefija (*exclusive scan*) para generar un vector que contenga los índices de cada clase en el vector resultado. Una vez que se generó este vector de desplazamientos, la segunda iteración llena el vector resultado con los índices de las filas de cada categoría. El valor del número de niveles generado en T4 es utilizado al momento de reservar memoria para las distintas estructuras.

Al igual que todas las etapas a excepción de T2, T5 ejecuta de forma secuencial en la rutina original. Sin embargo, dado que tanto el cálculo de tamaño como la clasificación de cada fila es independiente del resto, cada fila puede procesarse en paralelo. Finalmente el conteo puede plantearse como una reducción utilizando como *key* la categoría y por tanto también puede ser hecho en paralelo.

En este sentido se implementó una nueva rutina que mueve las etapas T4

y T5 a GPU para sacar partido de su paralelismo. La nueva implementación depende fuertemente de la biblioteca Thrust [6] que brinda implementaciones eficientes en GPU de rutinas básicas como *sort*, *scan* o *reduce*. El motivo de mover T4 no está relacionado con su peso en el tiempo global sino debido a la facilidad de hacerlo con THRUST y que permite evitar transferencias, en particular en la etapa T3 que ahora solo debe transferir el vector de niveles requerido por T6.

A diferencia de T4 que puede ser mapeada a una única primitiva de THRUST, `thrust::max`, la implementación de T5 es significativamente más compleja. En primer lugar, se computa el tamaño de cada fila utilizando `thrust::transform` con una función que toma como parámetro el vector de punteros a fila del formato CSR.

Posteriormente, se cuenta la cantidad de filas que van en cada categoría utilizando una combinación de primitivas. Primero se utiliza `thrust::transform` utilizando como parámetro los vectores de tamaño y nivel para obtener la categoría de cada fila (calculada como $7 \times lev(fila) + clase\_tam(fila)$. Luego se ordena el resultado y es utilizado como el *keys vector* de una reducción de un vector de unos, hecha con `thrust::reduce_by_key`. Este vector ($\#rows/cat$ en la Figura 3.1) no es estrictamente necesario como salida de T5 (es decir, si se ejecuta Anl_{ord}) pero es necesario como entrada para T6. El resultado de esta operación es un vector del conteo de la cantidad de filas por categorías. Por último, como el resultado de T5 es un vector de permutación, se repite la idea anterior pero en lugar de usarlo como clave para reducir, se lo utiliza como clave para aplicar `thrust::sort_by_key` y ordenar un vector de números naturales.

Mejora de T2, Anl_{nk}

La etapa T2 es, en promedio, la segunda etapa de mayor tiempo de ejecución en las matrices estudiadas. En determinadas matrices esta etapa es incluso la más costosa de la rutina, llegando a acumular en alguna matriz el 90 % del tiempo total de ejecución. Dado que esta etapa está implementada como un kernel de GPU en la rutina original, su optimización no resulta tan directa como las etapas anteriormente abordadas. Sin embargo, las nuevas versiones de CUDA brindan oportunidades para mejorar su implementación. En particular, para esto se utilizan dos nuevas herramientas ofrecidas en arquitecturas modernas: los *cooperative groups* [29] y la reducción por hardware [30].

La principal optimización que se realiza en esta etapa es enfocada en las matrices que cuentan con, en promedio, pocos no-ceros por fila. En la versión original del *kernel* cada *warp* procesa una única fila debido a que en caso de ejecutar dos (o más) filas en un mismo *warp* era posible que se produjera un *deadlock* de haber una dependencia entre filas que estuvieran asignadas al mismo *warp*. Esto se debe a que la escritura se hace una vez que termina el *loop* principal como se muestra en el Algoritmo 4.

Asignar una única fila por *warp* implica un desperdicio de recursos de cómputo en filas con menos de 32 elementos, puesto que varios procesadores no ejecutan. Por ejemplo una fila con dos elementos implicará que hayan 30 procesadores

ociosos durante toda la ejecución.

La introducción de la planificación independiente por hilos y los *cooperative Groups* en la generación Volta de las GPUs *Nvidia* permiten subdividir los *warps* en grupos independientes. Por tanto es posible procesar más de una fila en el mismo *warp* si se asigna cada una a un grupo distinto sin que sea posible generar un *deadlock*. Para definir la cantidad de filas por *warp* se utiliza como parámetro el promedio de elementos por fila para intentar balancear la carga de trabajo de los distintos grupos y hacer un uso mas eficiente de esta herramienta. Dependiendo de este promedio la rutina llama un *kernel* distinto que utiliza grupos de distinto tamaño siguiendo la Ecuación 3.2.

$$r_{wrp} = \text{máx}\left(1, \left\lfloor \frac{32}{nnz_{avg}} \right\rceil\right) \tag{3.2}$$

donde nnz_{avg} es el promedio de no-ceros por fila y $\lfloor\rceil$ es el operador de redondeo. Si bien finalmente se optó por utilizar el redondeo entero para determinar la categoría también se experimentó con las estrategias de redondeo "piso" y "techo" pero los resultados preliminares fueron peores.

El otro cambio propuesto es en la etapa de reducción del algoritmo. Para todas las filas que cuentan con más de un elemento distinto de 0 (sin contar la diagonal) sus elementos son procesados por threads distintos (o sea el elemento i es procesado por el thread $imodulo32$). Esto genera que, una vez que todos los hilos terminaron de procesar, se deba reducir el resultado utilizando la función máximo para calcular el nivel de la fila. En versiones anteriores dicha reducción se hacía, como es común en CUDA, a través de un *loop* y funciones de *shuffle*. Como se dijo anteriormente, a partir de CUDA 11.7 la reducción de enteros a nivel de *warp* se brinda como primitiva. Además, en GPUs con *Compute Capabilites* de 8,0 en adelante esta reducción es soportada por el *hardware*.

Se implementó una nueva variante del kernel que ejecuta la etapa T2 siguiendo las ideas antes descritas. Esta nueva rutina de análisis calcula la cantidad promedio de elementos no-cero por fila la cual se obtiene en $O(1)$ mediante $(row\_ptr[dim] - row\_ptr[0])/dim$, y en función de esta métrica determina cuántas filas serán asignadas a un *warp* y por ende el tamaño de sus grupos. Además esta rutina saca partido de la reducción por *hardware* en tarjetas que la soportan.

Mejora de T6, Anl_{omp}

La etapa T6 es, en promedio, la tercera etapa de mayor tiempo de ejecución de la rutina original luego de las etapas T2 y T5. En la rutina paralela, con la reducción general de T5, el peso específico de esta etapa aumenta y, con este, el interés por optimizarla.

Como se explicó anteriormente T6 es la etapa encargada de agrupar las filas para su procesamiento por $solver_{mr}$ y producir la información requerida para que cada *warp* pueda trabajar sobre las filas que le corresponden. En este *solver* los *warps* procesan una cantidad variable de filas garantizando que todas tienen el mismo nivel y la misma clase de tamaño (es decir son de la misma categoría) generando que los distintos *warps* queden divididos en particiones según la clase

de tamaño correspondiente. La información requerida por cada *warp* es, además de lo requerido por $solver_{ord}$, el índice inicial de las filas procesadas por éste en el vector de permutaciones y el número de filas a procesar lo que permite calcular el tamaño de sus particiones. La Figura 3.2 muestra la asignación de hilos a elementos de $solver_{mr}$.

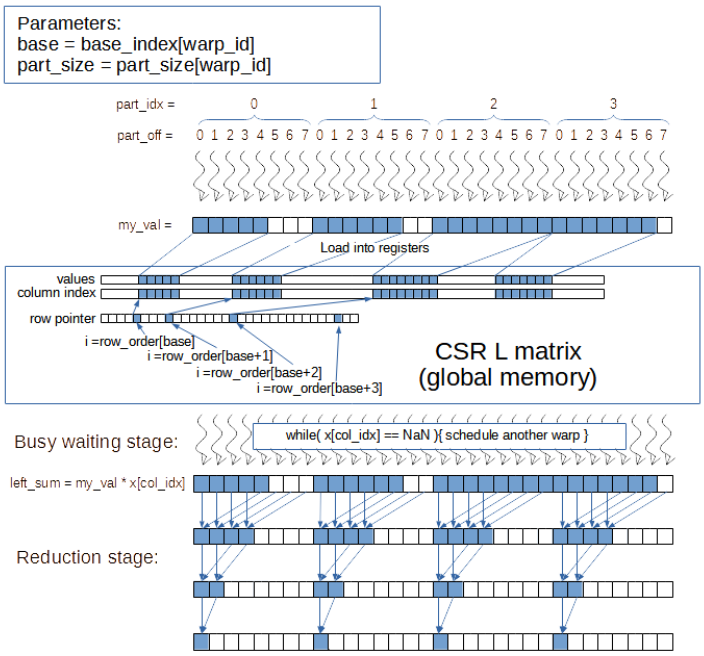

Figura 3.2: Esquema de asignación de hilos de $solver_{mr}$ tomado de [16].

Al igual que en T5, la versión secuencial de T6 está compuesta por dos iteraciones muy similares entre sí. La primera hace un cálculo simbólico para obtener la cantidad de *warps* requeridos y así poder reservar la memoria para las estructuras de datos necesarias. La segunda iteración se encarga de asignar filas (y tamaño de particiones) a *warps*. La estrategia para asignar filas consiste en agregar filas a *warps* hasta que este no tenga particiones libres.

La rutina original asigna filas de manera secuencial siguiendo el procedimiento explicado previamente. Sin embargo las filas que tienen distinta categoría no pueden ser asignadas al mismo *warp* por lo que se pueden procesar las distintas categorías de manera independiente.

En esta línea, como primera idea, se implementó una nueva versión paralela de la etapa T6 utilizando la biblioteca OpenMP [11]. Esta implementación de la etapa T6 explota el paralelismo utilizando múltiples hilos en CPU. Las modificaciones principales en esta etapa fueron la transformación de cada uno de los

loops en dos *loops* anidados. Mientras que el externo itera sobre las categorías, el interno lo hace sobre las filas de cada una. Posteriormente los *loops* exteriores pueden ser paralelizados utilizando la directiva `parallel for` de OpenMP. Esta biblioteca también permite definir cuántos hilos se generan como máximo entre otras funcionalidades de manejo de paralelismo en CPU.

En [19] presentamos resultados preliminares de la paralelización de esta etapa.

3.3.2. Rutina en GPU, Anl_{GPU}

En la sección anterior el enfoque consistió en dividir la rutina en etapas autocontenidas y atacar los principales cuellos de botella hallados, explotando el paralelismo cuando es posible. En este apartado se presenta un nuevo enfoque que consiste en trasladar la totalidad del análisis a la GPU.

De las tareas del análisis, la única que ejecuta en CPU en la propuesta del apartado 3.3.1 es la asignación de filas a *warps* (T6) mientras que tanto el cálculo de niveles como el ordenamiento y cálculo de tamaño ya ejecutan en GPU. Si bien no es esperable que la implementación en GPU de la etapa T6 genere una mejora significativa en el global de la rutina, al comparar con Anl_{omp} el hecho de ejecutarla en GPU implica evitar una serie de transferencias entre memorias. En concreto, debido a que T6 ejecuta en CPU es necesario transferir estructuras para su ejecución, como son el vector de niveles, el vector de permutación y el de tamaños, entre otros, además de requerir reservas y liberaciones de memoria adicionales al duplicar estructuras. La Tabla 3.3 muestra el tiempo de ejecución empleado en el *overhead* de memoria generado por la ejecución en las dos plataformas (suma de transferencias y reservas o liberaciones de memoria en CPU).

Los resultados muestran un *overhead* nada despreciable provocado por el uso de la CPU, en general siendo mayor en la plataforma con la GPU Ampere. En promedio este *overhead* ronda el 20 % y en matrices como `nlpkkt160`, `webbase-1M`, `wiki-Talk` o `rajat29` llega a superar el 30 %. Por lo anterior, se optó por evitar estas transferencias en base a implementar una nueva rutina que ejecuta íntegramente en GPU.

Aunque luego de las optimizaciones anteriores la rutina ejecuta prácticamente solo en GPU hasta la etapa T5 es necesario hacer algunos cambios a estas implementaciones. Primero es necesario modificar la etapa T5 para eliminar las operaciones relacionadas con el manejo de memoria en CPU. El otro cambio es crear las estructuras de memoria requeridas para la ejecución del *kernel* correspondiente a la etapa T6 en GPU, las cuales no son equivalentes a las que existían en CPU. Específicamente este *kernel* requiere de dos vectores con información sobre las categorías. El primer vector almacena el índice de inicio de cada categoría en el vector de filas ordenadas y el segundo almacena el primer *warp* asignado a cada categoría. En el Algoritmo 5 se presenta el *kernel* propuesto.

En el *kernel* de asignación cada *warp* procesa una única categoría y su función consiste en almacenar en el índice correspondiente de dos vectores: el ta-

	Turing		Ampere	
Id	**Tiempo**	**%**	**Tiempo**	**%**
1	0,13	6,56	0,16	8,59
2	5,28	4,81	5,62	7,38
3	22,44	36,43	26,53	44,37
4	29,83	24,40	31,60	26,44
5	48,91	24,50	53,52	27,98
6	0,89	16,97	1,10	25,47
7	3,95	31,07	4,03	34,37
8	6,48	30,17	6,54	30,96
9	0,47	7,08	0,54	11,67
10	0,53	10,58	0,63	15,38
11	0,25	17,87	0,29	19,48
12	0,37	21,81	0,44	24,02
13	0,51	19,13	0,68	24,68
14	2,94	34,47	3,31	37,12
15	0,31	21,63	0,38	24,26
16	8,97	26,68	10,13	32,64
17	0,08	3,53	0,10	5,41
18	1,63	0,77	2,03	1,55
19	0,30	8,78	0,38	12,95
20	0,12	10,03	0,12	10,29
21	0,19	12,39	0,23	16,27
22	0,42	18,89	0,53	24,47

Tabla 3.3: Tiempo (en ms) y peso relativo de los *overheads* de tiempo por el manejo de memoria en las plataformas experimentales detalladas en el Aparatado 3.1.

maño de categoría y el *offset* correspondiente a las filas que va a procesar el *warp* de $solver_{mr}$. Esta rutina recibe además como parámetro el análisis que se requiere. Este parámetro puede valer Anl_{mr} o Anl_{ord} en función de cuál rutina de resolución se utilizará posteriormente. Si bien Anl_{ord} podría llegar a utilizar un ordenamiento distinto dado que no requiere agrupar por tamaño pruebas preliminares mostraron que el costo de calcular por tamaño es insignificante por lo que se mantuvo el mismo ordenamiento para ambos. El resultado de la rutina de análisis es una estructura que, además de guardar la información requerida, mantiene qué estrategia de análisis se aplicó, así la rutina de resolución puede despachar el *kernel* correspondiente al análisis utilizado.

En las líneas 4 a 6 el algoritmo calcula cuántas (y cuáles) filas va a procesar el hilo. Posteriormente las filas 13 y 14 obtienen el tamaño de las filas a procesar, como cada bloque procesa una categoría todos los hilos usarán el mismo tamaño. Finalmente el *loop* de las filas 21 a 26 hace la asignación a *warps* de las filas.

Algoritmo 5 kernel que procesa la asignación de filas a *warps*

```
__global__ void kernel(int* buckets_off, int* rows_off, int* warp_base,
    int* warp_size){

 //Calcular num de filas a procesar por el bloque
   int off = buckets_off[blockIdx.x], lst=buckets_off[blockIdx.x+1];
   int cant = lst -off;
   int size;
   if(blockIdx.x== gridDim.x-1 && threadIdx.x==0)
       warp_base[lst] = rows_off[gridDim.x];

   if(cant == 0) return;
   if(threadIdx.x > cant) return;
   //Calcular size de las filas del bloque
   size = ((blockIdx.x%7) == 6) ? 0 : pow(2,(blockIdx.x%7));
   int rows = rows_off[blockIdx.x];

   __syncthreads();

   int x=threadIdx.x;
   int sz = size; if(sz == 0) sz =1;
   //Threads procesan las filas de un bucket en orden
   while(x+off<lst){
       warp_size[off + x] = size;
       warp_base[off + x] = rows + x*(32/sz);

       x += blockDim.x ;
       __syncwarp(__activemask());
   }
}
```

3.4. Resultados experimentales

En esta sección se brinda una evaluación de las propuestas presentadas anteriormente. En los apartados 3.4.1, 3.4.2 y 3.4.3 se evalúan, de manera preliminar e independientemente, las mejoras propuestas en el enfoque híbrido. Finalmente en 3.4.4 se presenta una evaluación completa del análisis y se expande el conjunto de matrices utilizado.

3.4.1. Evaluación de T5

Los cambios en esta etapa también implican la modificación de T4 que pasa a ejecutar en GPU. En la Tabla 3.4 se presentan los resultados de tiempo de ejecución en ambas plataformas de las dos etapas y se los compara contra la suma de las mismas etapas en Anl_{og}. Las matrices se corresponden con las presentadas en la Tabla 3.1.

	Turing				Ampere			
Id	**T4**	**T5**	Anl_{og}	**Speedup**	**T4**	**T5**	Anl_{og}	**Speedup**
1	0,08	0,53	1,12	1,85	0,08	0,46	1,10	2,04
2	0,09	5,82	52,65	8,90	0,08	5,48	52,11	9,37
3	0,14	19,78	240,04	12,05	0,12	17,99	240,30	13,27
4	0,18	34,20	223,25	6,49	0,15	30,46	223,57	7,30
5	0,25	54,22	370,37	6,80	0,20	48,26	378,14	7,80
6	0,08	1,14	6,78	5,56	0,08	1,11	6,98	5,88
7	0,09	4,48	15,37	3,36	0,08	4,05	15,16	3,67
8	0,10	6,95	16,00	2,27	0,09	6,88	15,61	2,24
9	0,08	1,00	2,98	2,76	0,08	1,02	3,01	2,74
10	0,08	0,78	3,55	4,17	0,07	0,71	3,52	4,49
11	0,08	0,70	1,26	1,62	0,09	0,74	1,16	1,40
12	0,08	0,80	1,57	1,79	0,08	0,82	1,52	1,68
13	0,08	1,12	1,82	1,52	0,11	1,09	1,78	1,49
14	0,08	3,38	18,01	5,20	0,09	3,64	17,58	4,71
15	0,08	0,73	1,06	1,32	0,10	0,72	0,97	1,19
16	0,11	10,62	39,70	3,70	0,10	9,48	39,34	4,11
17	0,08	0,50	0,14	0,25	0,07	0,43	0,15	0,29
18	0,08	3,80	13,92	3,59	0,08	3,48	14,18	3,99
19	0,08	0,99	3,03	2,84	0,08	1,00	3,03	2,83
20	0,07	0,53	0,65	1,09	0,10	0,53	0,55	0,88
21	0,08	0,55	1,04	1,66	0,08	0,55	0,98	1,55
22	0,08	0,82	2,65	2,95	0,12	0,83	2,58	2,73

Tabla 3.4: Tiempo de ejecución (en ms) de las etapas T4 y T5 de la propuesta (Anl_{opt}) comparado contra la rutina original (Anl_{og}) en ambas plataformas.

Los resultados de la nueva rutina propuesta mejoran notablemente los tiempos de ejecución de Anl_{og} en casi todas las matrices con *speedup* promedio de $3,8\times$. Las dos únicas matrices que no lo hacen son `cond-mat` (en Ampere) y `ex_data1`. Si bien el *speedup* mínimo es muy bajo ($\approx 0,27$), este se produce en la matriz más pequeña cuyo tiempo de ejecución total es menor que $2ms$. Por otro lado los mejores *speedups* (siendo el máximo $13,3\times$ en la Ampere) son alcanzados en las matrices que requieren mayor tiempo de ejecución como `hollywood-2009`, `nlpkkt160`, `road_central` o `road_usa` que toman más de $100ms$ (en algún caso incluso más de $600ms$).

En general los *speedups* son explicados básicamente por la mejora de T5 que, a excepción de `ex_data1`, obtiene resultados positivos en todas las matrices. Por el contrario T4 tiene resultados más diversos, logrando grandes *speedups* en matrices como `road_usa` ($37\times$ en Ampere) pero también algunos resultados muy negativos ($0,07\times$ en `cond-mat`).

Las modificaciones a T5 (y T4) tienen impacto en otras etapas. Por ejemplo para ejecutar T4 ya no es necesario esperar que sea transferido el vector de

niveles pero, por el contrario, es necesario transferir los resultados de T5 a CPU. Con el objetivo de analizar el impacto de estos cambios en el tiempo global de la rutina en la Tabla 3.5 se presentan y comparan los tiempos totales de Anl_{og} y Anl_{opt}.

	Turing			Ampere		
Matriz	Anl_{og}	Anl_{opt}	**Speedup**	Anl_{og}	Anl_{opt}	**Speedup**
1	2,11	1,70	1,24	2,01	1,68	1,19
2	173,12	126,51	1,37	135,25	88,83	1,52
3	297,76	78,00	3,82	302,49	80,49	3,76
4	343,10	154,57	2,22	344,09	151,61	2,27
5	593,70	278,43	2,13	596,72	267,14	2,23
6	11,88	6,37	1,87	10,99	5,24	2,10
7	25,65	14,92	1,72	24,28	13,36	1,82
8	32,81	24,09	1,36	33,17	24,96	1,33
9	8,68	6,81	1,27	6,84	4,97	1,38
10	8,07	5,41	1,49	7,03	4,34	1,62
11	1,74	1,29	1,35	1,68	1,41	1,18
12	2,31	1,64	1,41	2,31	1,75	1,32
13	4,11	3,52	1,17	3,39	2,91	1,16
14	24,77	10,28	2,41	24,32	10,67	2,28
15	1,58	1,33	1,19	1,53	1,43	1,07
16	73,36	44,57	1,65	71,99	42,18	1,71
17	1,90	2,37	0,80	1,36	1,75	0,78
18	176,83	166,85	1,06	130,41	119,85	1,09
19	5,11	3,18	1,60	4,60	2,68	1,72
20	0,97	0,94	1,03	0,87	1,00	0,87
21	1,84	1,44	1,27	1,67	1,36	1,23
22	3,72	1,98	1,88	3,57	1,96	1,82

Tabla 3.5: Tiempo de ejecución total (en ms) de las rutinas Anl_{opt} y Anl_{og} en ambas plataformas.

Nuevamente los resultados se condicen con el comportamiento de la etapa T5 obteniendo los mejores *speedups* en la matriz `nlpkkt160` y el peor en `ex_data1`. En general, los bloques de matrices con mejores y peores rendimientos se mantienen similares, aunque dentro de estos bloques hay algunos cambios de orden. Además, los extremos se ven moderados dejando un rango de *speedup* de entre $0,8\times$ a $3,8\times$.

Finalmente, a falta de una experimentación en mayor profundidad, parece haber una correlación entre la dimensión de la matriz y los resultados de esta etapa. Esta teoría es razonable dado que las implementaciones de tanto T4 como T5 en la rutina original tienen un costo en tiempo de ejecución proporcional a la cantidad de filas de la matriz. En el caso de Anl_{opt}, T4 consume un tiempo de ejecución $O(\log_2 dim)$, mientras que en la tapa T5 todas las filas pueden

procesarse en paralelo. Esto genera que al aumentar la cantidad de filas los resultados de la propuesta mejoren relativamente a Anl_{og} hasta el punto en que se sature la GPU.

3.4.2. Evaluación de T2

A diferencia del apartado anterior, dado que esta etapa ya ejecutaba en GPU en la rutina original, los cambios introducidos a T2 no tienen impacto en el resto de las etapas. Por lo tanto, en este apartado se compara únicamente la implementación de T2 propuesta con la de Anl_{og}. La Tabla 3.6 presenta dicha comparación en ambas plataformas de ejecución. Al igual que en las tablas anteriores el factor de *speedup* corresponde a la división t_{og}/t_{opt} donde t_{opt} es el tiempo de ejecución de la nueva implementación propuesta y t_{orig} el de Anl_{og}.

	Turing			Ampere		
Matriz	**T2$_{og}$**	**T2$_{opt}$**	**Speedup**	**T2$_{og}$**	**T2$_{opt}$**	**Speedup**
1	0,68	0,93	0,73	0,58	0,73	0,80
2	105,68	94,36	1,12	67,42	60,46	1,12
3	9,52	7,31	1,30	5,85	4,61	1,27
4	21,59	7,55	2,86	14,08	4,86	2,89
5	52,76	14,74	3,58	32,77	9,73	3,37
6	3,79	2,96	1,28	2,52	1,96	1,29
7	2,88	1,09	2,65	1,53	0,92	1,67
8	2,76	3,34	0,83	1,99	2,81	0,71
9	5,06	4,73	1,07	3,12	2,67	1,17
10	3,82	3,39	1,13	2,72	2,42	1,13
11	0,11	0,09	1,27	0,09	0,13	0,67
12	0,12	0,09	1,29	0,10	0,14	0,74
13	1,44	0,53	2,73	0,67	0,40	1,66
14	1,94	0,78	2,50	1,30	0,64	2,03
15	0,08	0,06	1,26	0,07	0,10	0,66
16	9,57	3,23	2,97	6,16	2,00	3,07
17	1,61	1,37	1,17	1,06	1,00	1,07
18	160,57	202,91	0,79	113,34	123,31	0,92
19	1,57	1,74	0,90	1,00	1,15	0,87
20	0,10	0,13	0,78	0,09	0,15	0,59
21	0,50	0,46	1,10	0,37	0,37	1,01
22	0,35	0,54	0,64	0,19	0,35	0,55

Tabla 3.6: Comparación del tiempo total de ejecución (en ms de la etapa T2 en Anl_{og} y Anl_{nk}.

En la gran mayoría de las matrices el nuevo *kernel* mejora la implementación anterior. Los mayores valores de *speedup* parecen correlacionarse con dos variables: la dimensión de la matriz y la densidad medida como el promedio de ele-

mentos no nulos por fila nnz_{avg}. En general los mejores resultados se dan en matrices que son de dimensiones medias ($dim \geq 100000$) o altas ($dim \geq 1000000$) y caen en las primeras tres categorías ($nnz_{avg} \leq 5$)

Por otro lado los peores resultados son obtenidos por matrices pequeñas ($dim \leq 100000$). Este comportamiento es esperable dado que, en matrices de menor dimensión, al agrupar filas (y por ende lanzar menor cantidad de *warps*) no se logrará saturar la GPU.

Finalmente, hay dos matrices que requieren comentario adicional: `wiki-Talk` y `CO`. La primera es una matriz grande, en la que se esperarían buenos resultados, pero sin embargo sus resultados empeoran considerablemente respecto a la línea base. Este caso parece explicarse por la densidad de la matriz ($\approx 1,28$), lo cual podría significar que hay muchas filas que tienen 1 y 2 elementos (clases de tamaño 0 y 1 respectivamente) puesto que todas las filas tienen, al menos, el elemento en la diagonal. Agrupar filas que solo cuentan con el elemento en la diagonal con otras que deben ejecutar el *loop* principal no generaría una gran mejora (dado que las filas de clase 0 terminarían generalmente mucho antes al no depender de otras) y generaría que no se compense el *overhead* generado por el agrupamiento. De todas formas es necesaria una mayor experimentación para confirmar esta hipótesis.

Con respecto a `CO` la explicación parece más directa ya que esta matriz cuenta con, en promedio, $1,02$ filas por cada nivel. Esto genera que la mayoría de los niveles cuente con una sola fila, con lo cuál es muy probable que, si se agrupan filas, éstas dependan (directa o indirectamente) entre sí provocando serialización dentro del *warp*. Nuevamente no habrían grandes ganancias para contrarestar el *overhead* del agrupamiento.

3.4.3. Evaluación de T6

La Tabla 3.4.3 presenta el tiempo requerido por la etapa T6 obtenida luego de ejecutar el programa con diferente cantidad de hilos (dos, cuatro y ocho) comparando con la línea base.

Se puede observar que las dos versiones que resultan más rápidas en general, siendo superiores en 20 matrices del total de 22, son la línea base y la versión de Anl_{omp} que utiliza 4 hilos. Sin embargo, la versión de Anl_{omp} que utiliza 2 hilos, aunque generalmente pierda ante ellas, tiene un desempeño similar. En general, las matrices en las cuales la línea base obtiene los mejores resultados son de menor dimensión que en las que el mejor resultado se da en la versión con 4 hilos.

Las tres versiones que usan OpenMP tienen comportamientos similares en el sentido de que cuando una supera la línea base, las demás también lo hacen. Existen algunos casos excepcionales en los que la versión de 4 hilos o la de 2 son la única versión en tener un mejor desempeño que la línea base.

Finalmente, es interesante evaluar los resultados teniendo en cuenta la importancia de la etapa T6 en cada caso. Por ejemplo, en las matrices `road_usa` y `road_central` dicha etapa utiliza 55 % y 58 % del tiempo total luego de aplicados los cambios de T5 y T2. En general, las matrices en las que la línea base

Matrix	**T6**$_{omp8}$	**T6**$_{omp4}$	**T6**$_{omp2}$	**T6**$_{base}$
1	0,34	0,22	0,11	0,09
2	4,23	3,96	6,04	10,32
3	15,86	12,89	14,02	28,74
4	58,13	49,11	55,97	73,75
5	96,14	83,03	94,11	132,33
6	0,62	0,48	0,60	0,78
7	4,07	3,29	3,43	4,26
8	6,06	4,78	5,18	8,72
9	0,44	0,34	0,39	0,36
10	0,52	0,41	0,37	0,40
11	0,33	0,28	0,21	0,17
12	0,41	0,29	0,28	0,26
13	0,63	0,44	0,48	0,45
14	1,99	1,46	1,98	2,42
15	0,41	0,27	0,22	0,20
16	14,16	10,10	11,38	16,49
17	0,28	0,15	0,11	0,02
18	2,38	1,87	2,50	1,12
19	0,38	0,30	0,26	0,28
20	0,35	0,21	0,13	0,07
21	0,28	0,20	0,13	0,10
22	0,53	0,36	0,38	0,33

Tabla 3.7: Comparación del tiempo de ejecución (en ms) de la etapa T6 para la línea base y Anl_{omp} con $n_{threads} = 2, 4, 8$

es más rápida son aquellas en las cuales esta etapa requiere menos tiempo en comparación al tiempo total. Específicamente, el máximo peso que tiene esta etapa en las matrices en las que la línea base es la de mejor desempeño es 16 % y en promedio el peso es de 9 %. Por otro lado, el promedio en las matrices en las cuales gana T6$_{omp4}$ es de 30 %, incluso con picos de hasta 58 %.

Debido a lo mencionado anteriormente se optó por seleccionar la rutina de Anl_{omp4} como la implementación final, considerando que se obtienen los "mejores resultados" en la mayoría de las matrices donde esta etapa represente una parte significativa del tiempo de ejecución ($\geq$ 15 %). En las matrices para las cuales Anl_{omp4} obtiene un peor resultado, la diferencia entre estos es bastante pequeña. Sin embargo, este comportamiento no se espera necesariamente para todas las matrices. Se deja como trabajo futuro explorar la idea de modificar la rutina para que tenga un comportamiento adaptativo en función de las características de las matrices.

Para el resto de la evaluación experimental se adopta la rutina que integra las optimizaciones para cada etapa como la línea base, y se compara esta va-

riante con la rutina del análisis que corre enteramente sobre GPU. La Tabla 3.5 presenta los tiempos de ejecución para cada etapa de la rutina.

3.4.4. Evaluación general

En esta sección se presenta una evaluación general de las tres rutinas de análisis: Anl_{og}, Anl_{opt} y Anl_{GPU}. En primer lugar, se compara el costo de todo lo correspondiente a la asignación de filas a *warps* (etapas T6-T8 de Anl_{og}). Posteriormente, se comparan los tiempos totales de las tres rutinas. Por último se expande el conjunto de casos de prueba a un conjunto más amplio de matrices.

Debido a que el foco de los cambios de Anl_{GPU} está concentrado en la asignación de filas a *warps*, primero se concentra la evaluación experimental allí. En este sentido, en la Tabla 3.8 se presentan los tiempos de ejecución de las tres rutinas Anl_{og}, Anl_{opt} y Anl_{GPU} para las etapas correspondientes a la asignación. Además se presentan los *speedups* alcanzados por la rutina Anl_{GPU} con respecto a cada una de las otras rutinas.

Primero que nada, es relevante destacar en la gran mayoría de las matrices los *speedups* son mayores en la plataforma AMPERE. Al comparar la rutina Anl_{GPU} con Anl_{opt} se encuentra que en prácticamente todos los casos la nueva rutina mejora a la anterior. En particular, la única excepción es la matriz `CO` en la tarjeta TURING. Esta matriz es excepcionalmente mala para nuestra estrategia de análisis al tener, en promedio, $1,02$ filas por nivel, generando que en la etapa de asignación la gran mayoría de los *warps* despachados solamente realicen una asignación (dejando 31 hilos ociosos).

La comparación de Anl_{GPU} contra Anl_{og} arroja resultados similares y, al igual que en la comparación anterior, la mayoría de las matrices tienen *speedups* positivos. Además del caso de `CO`, también `exdata_1` tiene un *speedup* menor que 1. Con respecto a `CO`, dado que el *kernel* correspondiente a la etapa T2 es el factor preponderante, es posible que la diferencia de comportamiento entre tarjetas se explique con la reducción por hardware. El caso de `exdata_1` se explica por el hecho de que la etapa T5 es la que acumula mayor costo en la matriz y, como se mostró anteriormente, las modificaciones aumentan el tiempo de ejecución. Dejando de lado esto último en general las comparaciones arrojan los mismos datos siendo (como es esperable) los speedups más grandes.

Si bien a primera vista podría parecer extraño que las matrices con mayor *speedup* sean `road_usa`, `road_central` y `circuit5M_dc` en lugar de las que lograron mejorar más sus resultados al paralelizar T6 con OpenMP, es importante recordar que gran parte de las ganancias proviene de las transferencias de memoria evitadas. Sin embargo, es interesante analizar las mejoras en la asignación desacopladas de lo ganado por transferencias de memoria. Para esto se define la métrica *speedup esperado* que consiste en calcular el *speedup* de remover el *overhead* de memoria ($\frac{t_{base}}{t_{base}-t_{overhead}}$). Si bien la métrica no es exacta puesto que es necesario crear nuevas estructuras para la ejecución de la nueva implementación de T6 permite dar una idea general de qué tanto mejora la asignación. La gran mayoría de las matrices mejoran el *speedup esperado*, las únicas que no lo hacen son `CO`, `nlpkkt160`, `ship_003`, `hollywood-2009` (TURING) y

Matriz	Anl_{GPU}	Anl_{og}	**Speedup**	Anl_{opt}	**Speedup**
TURING					
1	0,14	0,24	1,76	0,39	2,86
2	5,86	12,98	2,21	9,30	1,59
3	23,59	42,37	1,80	33,84	1,43
4	6,70	88,63	13,23	80,05	11,95
5	7,25	155,49	21,46	130,29	17,98
6	0,25	1,15	4,60	0,99	3,96
7	0,97	5,75	5,94	6,89	7,11
8	2,16	11,73	5,43	10,83	5,01
9	0,19	0,53	2,80	0,69	3,67
10	0,17	0,59	3,48	0,73	4,32
11	0,11	0,27	2,45	0,50	4,52
12	0,11	0,39	3,48	0,61	5,43
13	0,13	0,61	4,60	0,80	6,04
14	0,99	3,71	3,76	4,12	4,16
15	0,11	0,31	2,68	0,52	4,52
16	2,72	20,95	7,71	19,35	7,13
17	0,17	0,07	0,43	0,28	1,59
18	4,71	1,99	0,42	3,67	0,78
19	0,24	0,41	1,73	0,60	2,51
20	0,11	0,13	1,19	0,36	3,22
21	0,11	0,20	1,75	0,39	3,48
22	0,11	0,48	4,26	0,62	5,59
AMPERE					
1	0,13	0,27	2,11	0,31	2,45
2	3,96	14,10	3,56	9,69	2,45
3	17,01	49,90	2,93	36,48	2,14
4	6,46	96,24	14,89	83,17	12,87
5	6,97	168,80	24,21	133,08	19,09
6	0,20	1,31	6,70	1,12	5,72
7	0,96	6,15	6,43	6,52	6,82
8	2,02	13,25	6,57	10,74	5,33
9	0,16	0,61	3,88	0,74	4,72
10	0,14	0,68	4,77	0,82	5,69
11	0,11	0,31	2,94	0,45	4,18
12	0,11	0,44	4,08	0,58	5,33
13	0,12	0,67	5,41	0,94	7,59
14	0,97	4,27	4,40	4,30	4,43
15	0,11	0,35	3,13	0,57	5,15
16	2,56	23,18	9,05	19,33	7,55
17	0,16	0,08	0,53	0,35	2,25
18	3,41	2,50	0,73	3,89	1,14
19	0,22	0,46	2,10	0,66	3,01
20	0,11	0,14	1,35	0,29	2,77
21	0,11	0,23	2,15	0,33	3,06
22	0,11	0,53	4,97	0,69	6,38

Tabla 3.8: Tiempo de ejecución (en ms) de la parte relacionada a asignación de filas a *warps* en las tres rutinas Anl_{GPU}, Anl_{og} y Anl_{opt} en ambas plataformas. Comparación de Anl_{GPU} contra las dos rutinas previas.

`TSOPF_RS_b300_c2` (AMPERE). Por otro lado hay matrices que destacan positivamente como `road_usa` (supera el speedup esperado por $\approx 16,9$), `road_central` ($\approx 10,75$) o `circuit5M_dc` ($\approx 5,6$).

El costo de la asignación de filas a *warps* es lineal con la cantidad de filas, además las nuevas implementaciones paralelizan la asignación de las categorías. Por lo anterior es razonable esperar que matrices con gran cantidad de filas y categorías sean las que obtengan mejores resultados. Sin embargo la matriz `CO` es la que tiene mayor cantidad de niveles (y por tanto mayor cantidad de categorías) pero tiene muy malos resultados en la asignación. La explicación de este resultado es que esta matriz tiene un bajo nivel de filas por categoría, lo que genera que, dado que cada categoría es procesada por un bloque, no haya trabajo suficiente para todos los hilos de un bloque. En el otro extremo se ubican matrices con muchas filas pero pocas categorías (por ejemplo `nlpkkt160`), estas matrices, si bien obtienen buenos resultados no logran superar el *speedup* esperado. Es interesante como trabajo futuro analizar una paralelización más fina (por ejemplo, a nivel de un *warp* por categoría) en casos como este.

Las tres matrices que tienen mejores resultados relativos al *speedup* esperado son las que logran un balance entre una cantidad de categorías suficiente para saturar la GPU y suficientes filas por categoría como para que las 7 categorías de cada nivel funcionen sin procesadores ociosos. Por ejemplo, la matriz con mejores resultados, `road_usa` tiene $77 * 7$ categorías y más de 300000 filas por nivel.

Una vez hecho el análisis de la asignación, es interesante ver cómo impacta esta parte en el total de la rutina. Para esto en la Tabla 3.9 se presentan los tiempos totales de ejecución de Anl_{opt} y Anl_{GPU} para las dos plataformas de prueba.

Por último, es interesante evaluar las propuestas en un conjunto más amplio de matrices. En este sentido se generó un conjunto de matrices de la colección SuiteSparse. Este conjunto de matrices se generó seleccionando las matrices cuya dimensión sea mayor a 10000 y tales que su componente diagonal inferior tenga al menos un no-cero por fila (el elemento de la diagonal).

Se intentaron ejecutaron las tres implementaciones de la rutina de análisis (Anl_{og}, Anl_{opt} y Anl_{GPU}) en todo el conjunto de matrices mencionado anteriormente. Debido a limitaciones de las plataformas no fue posible completar la ejecución en todo el conjunto de matrices. El conjunto final (de matrices en que se puso hacer la ejecución completa) está compuesto por 1119 matrices de distintas dimensiones y patrones de dispersión.

Como primer punto se comparan las mejoras obtenidas para la versión híbrida (CPU-GPU) y la versión en GPU de análisis optimizadas frente a la original (i.e. Anl_{opt} y Anl_{GPU} sobre Anl_{og}) en las matrices correspondientes. La Figura 3.3 muestra los resultados en las dos plataformas (con las tarjetas TURING y AMPERE).

De los resultados puede verse que los *speedups* crecen claramente al aumentar la dimensión con líneas de tendencia logarítmicas. Los resultados son similares en ambas plataformas en términos generales aunque en la tarjeta AMPERE los *speedups* tienden a ser mayores.

	Turing			Ampere		
Matriz	Anl_{GPU}	Anl_{opt}	**Speedup**	Anl_{GPU}	Anl_{opt}	**Speedup**
1	1,65	2,04	1,23	1,35	1,83	1,36
2	103,64	109,87	1,06	66,57	76,11	1,14
3	38,64	61,61	1,59	26,92	59,79	2,22
4	28,11	122,28	4,35	20,46	119,52	5,84
5	44,48	199,61	4,49	31,25	191,31	6,12
6	3,96	5,24	1,32	2,75	4,33	1,58
7	4,33	12,71	2,94	3,54	11,73	3,31
8	8,19	21,48	2,62	6,72	21,13	3,14
9	5,57	6,58	1,18	3,32	4,59	1,38
10	4,19	5,05	1,20	3,04	4,08	1,34
11	0,88	1,42	1,61	0,72	1,50	2,07
12	0,99	1,70	1,72	0,81	1,82	2,26
13	1,54	2,66	1,72	1,25	2,75	2,21
14	3,59	8,54	2,38	3,08	8,92	2,90
15	0,88	1,41	1,61	0,69	1,56	2,27
16	10,84	33,62	3,10	7,92	31,02	3,91
17	2,09	2,30	1,10	1,52	1,92	1,27
18	209,26	210,61	1,01	127,60	130,93	1,03
19	2,78	3,47	1,25	2,07	2,94	1,42
20	0,80	1,15	1,44	0,63	1,17	1,85
21	1,12	1,53	1,37	0,87	1,40	1,61
22	1,39	2,20	1,58	1,04	2,15	2,07

Tabla 3.9: Tiempo de ejecución (en ms) de las rutinas Anl_{GPU} y Anl_{opt} en ambas plataformas.

Estos resultados van en la línea de los presentados en el Apartado 3.4.1 donde se consiguen *speedups* positivos (mayores a 1) en casi todas las matrices a excepción de las que cuentan con muy pocas filas. Es razonable que, a grandes rasgos, los resultados se comporten de manera similar a T5 dado que es, con diferencia, la etapa más costosa en el algoritmo original. Notar que, dado que T5 obtiene mayores *speedups* cuando hay mayor cantidad de filas puesto que las filas se procesan en paralelo en la nueva implementación los *speedups* crecen linealmente con la dimensión hasta $dim \approx 350000$ (presumiblemente hasta saturar la GPU) y luego moderan su crecimiento. También puede verse que, si bien las dos versiones optimizadas Anl_{opt} y Anl_{GPU} tienen un comportamiento similar, en general la versión que ejecuta completamente en GPU obtiene significativamente mejores resultados.

Dado que las mejoras de T5 son significativamente más importantes estas pueden ocultar los resultados de las otras optimizaciones. A efectos de evaluarlas se presenta un análisis que centra las comparaciones en dichas etapas. A continuación se analizan los resultados de la versión original de T2 con la

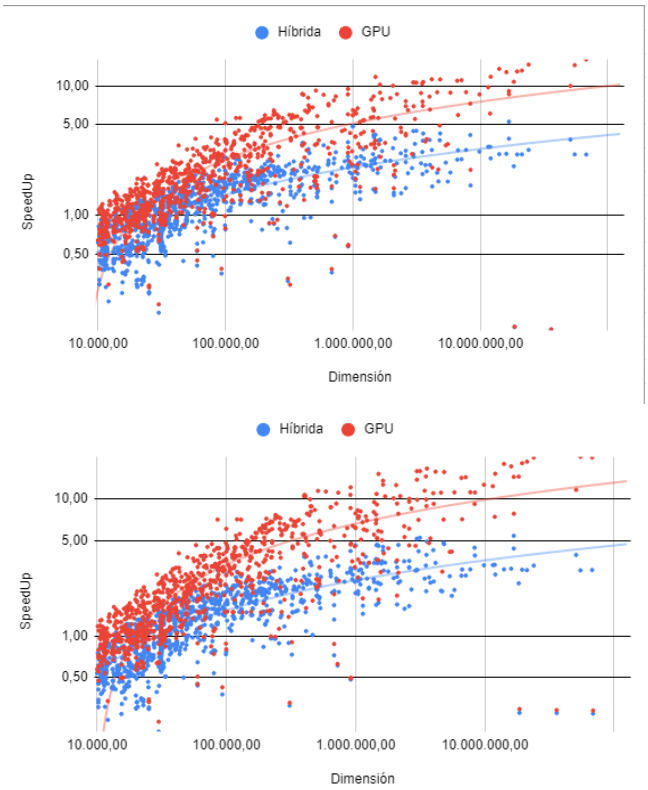

Figura 3.3: *speedups* del tiempo de ejecución de Anl_{opt} y Anl_{GPU} sobre Anl_{og} en la GPU en la plataforma Turing (arriba) y Ampere (abajo).

optimización propuesta en el Apartado 3.3.1.

Como se explicó al presentarla, la optimización consiste en agrupar filas que sean procesados por un mismo *warp* cuando hay una cantidad baja de no ceros por fila promedio. Esto se debe a que permitirá utilizar procesadores que de otra forma estarían ociosos. Para esto dependiendo del promedio de no-ceros por fila se despacha un *kernel* distinto, en concreto existen versiones que agrupan de a 2, 4, 8, o 16 filas además de la original. Por lo tanto para analizar el impacto de esta mejora se debe tomar en cuenta las matrices que tienen el promedio de no-ceros por fila en el rango de las variantes que agrupan, es decir no tiene sentido analizar matrices con grandes cantidades de no-ceros por fila ya que utilizan la variante original. La Tabla 3.10 presenta los resultados discriminados según cuál de las versiones será despachada. El subconjunto de las matrices que tienen, en promedio, hasta 24 no-ceros por fila consiste de 940 matrices.

Los resultados muestran que, como era esperable, las versiones con una menor cantidad de no-ceros por fila promedio (1-3, 3-6) son las que obtienen mayores resultados mientras que en las versiones con mayor cantidad (6-12, 12-24)

	Turing			Ampere		
NNZ$_{prom}$	Anl_{og}	Anl_{opt}	**Speedup**	Anl_{og}	Anl_{opt}	**Speedup**
1-3	2.213,25	1.673,79	1,32	1.299,18	1.070,86	1,21
3-6	2.354,30	1.550,20	1,52	1.268,12	874,47	1,45
6-12	1.021,55	895,61	1,14	621,59	553,85	1,12
12-24	619,78	568,97	1,09	410,65	357,58	1,15

Tabla 3.10: Comparación de la suma de los tiempos de ejecución de T2 (en ms) de Anl_{og} y Anl_{opt} para las matrices con menos de 24 no-ceros por fila en promedio en Turing y Ampere.

las mejoras son más acotadas. Esto se debe a que cuanto menor sea la cantidad de no-ceros promedio mayor es la proporción de procesadores ociosos en la implementación original por lo que existe un mayor margen de mejora.

Sin embargo, si se observan únicamente las dos primeras filas se ve una tendencia inversa a la esperada. Esto podría deberse a las matrices con promedios de no-ceros cercanos a uno. Estas matrices, dependiendo del patrón de dispersión, pueden generar muchos agrupamientos en los que haya filas de nivel 0 y 1. Este tipo de agrupamientos genera que haya una parte del *warp* que ejecute una gran cantidad lecturas a memoria global (*busy waiting* en el *loop* principal) mientras que la otra parte solo deba leer al iniciar. Sería necesaria una mayor investigación para probar esta hipótesis lo que está fuera del alcance de este trabajo.

Finalmente es interesante analizar los dos enfoques para la mejora de la asignación de hilos a *warps*. Por un lado se presentó la estrategia de paralelización en CPU que utiliza la biblioteca openMP para el manejo de hilos mientras que luego se presentó una alternativa en GPU. Debido a que una de las ventajas principales de la segunda estrategia es evitar las transferencias y el manejo de memoria en CPU (etapas T7 y T8) se toma en cuenta estas etapas para el análisis de los resultados.

La Figura 3.4 presenta los *speedups* del tiempo de ejecución de las etapas comprendidas entre T6 y T8 de las versiones Anl_{opt} (híbrida) y Anl_{GPU} sobre Anl_{og} para las dos plataformas. Para la versión híbrida se utilizó la versión de T6 que utiliza 4 hilos.

Puede verse que, similar a lo que ocurre con T5, existe una clara correlación entre la dimensión de la matriz y los *speedups* obtenidos. Esto es coherente con los análisis previos dado que la asignación de filas a *warps* es $O(dim)$. Si se comparan los resultados entre ambas versiones se ve que las tendencias son similares (un crecimiento acentuado hasta $dim \approx 100000$ y posteriormente un estancamiento/crecimiento moderado) pero hay gran diferencia entre los valores de *speedup* obtenidos. Esto genera que mientras que Anl_{GPU} tiene *speedups* positivos (mayores a 1) en casi todas las matrices, en Anl_{opt} los resultados positivos comienzan a ser mayoría con matrices de dimensión mayor a 100000.

A partir de este análisis resulta evidente que en enfoque de Anl_{GPU} es netamente superior a la versión híbrida, probablemente debido en parte a la reduc-

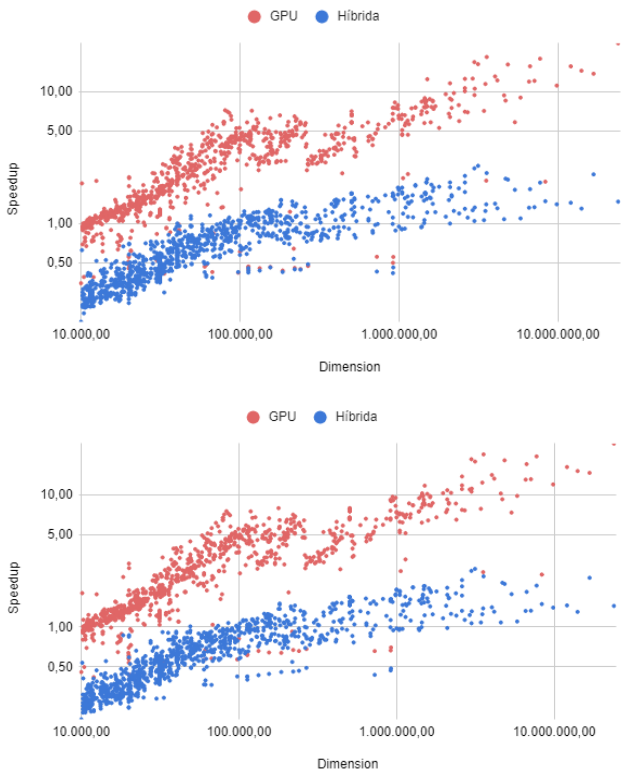

Figura 3.4: *speedups* del tiempo de ejecución de T6-T8 para Anl_{opt} y Anl_{GPU} sobre Anl_{og} en la plataforma TURING (arriba) y AMPERE (abajo).

ción de tiempos de transferencia de memoria.

Independientemente de lo anterior es interesante analizar los efectos en el tiempo global del conjunto de matrices. En la Figura 3.5 se presenta el tiempo de ejecución de todo el conjunto de 1119 matrices discriminado por las etapas T1 a T5 y T6 a T8.

Puede verse que, aunque los resultados de Anl_{GPU} son notoriamente mejores, los resultados de Anl_{opt} muestran un *speedup* del entorno de 1,25 para el conjunto de estas etapas. Sin embargo los resultados de Anl_{GPU} la superan ampliamente con valores de *speedup* de entre $5,5$ y $6,0$.

Al analizar el tiempo de ejecución tomado por Anl_{GPU} y compararlo con Anl_{og} puede verse que, por ejemplo en la tarjeta TURING, el tiempo total de ejecución en el conjunto se reduce un 57 % (de $24,5s$ a $10,6s$ aproximadamente). Sin embargo, al desglosar ese resultado en los dos bloques se ve una notoria diferencia, entre T1 y T5 la reducción es ligeramente menor (53 %) mientras que en T6-T8 aumenta notoriamente (la reducción es de (83 %). Puede verse que Anl_{opt} da resultados intermedios en ambos bloques (47 y 22 % respectivamente).

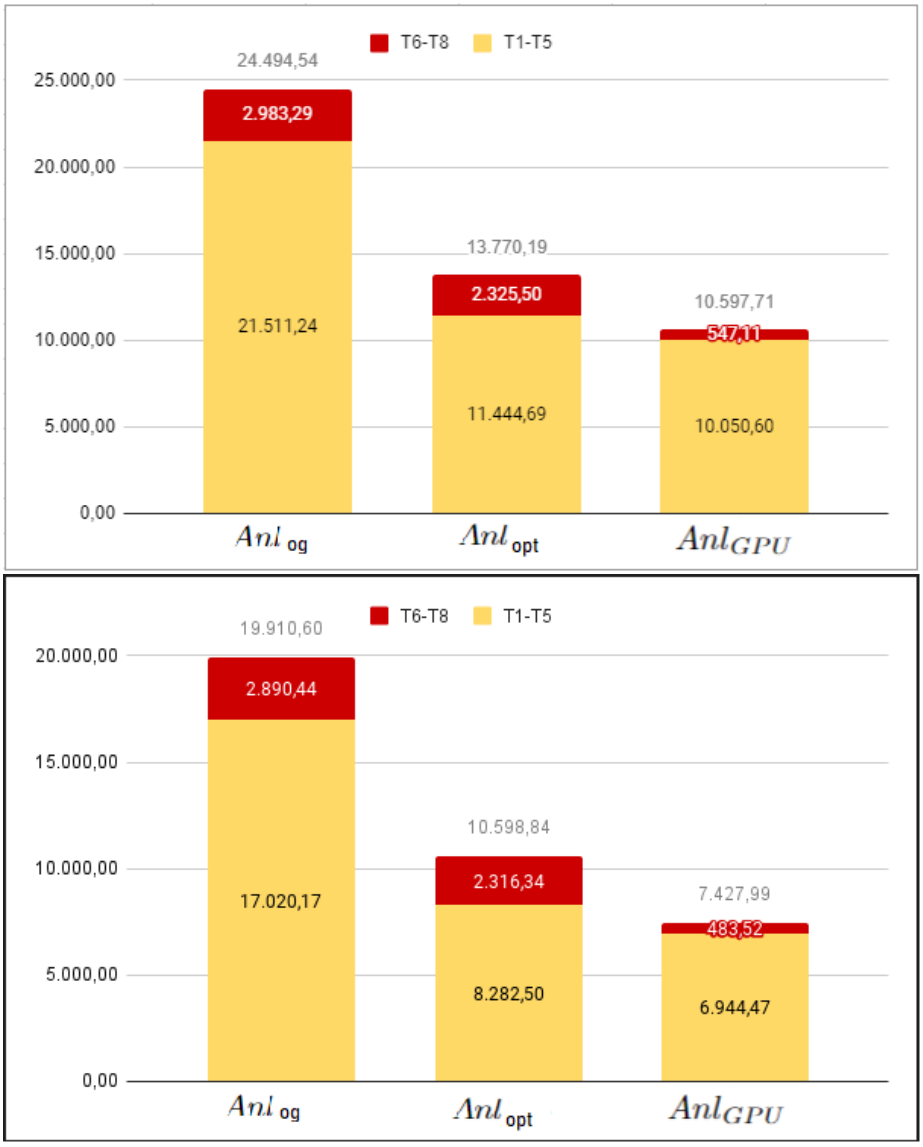

Figura 3.5: Suma de los tiempos de ejecución para Anl_{og}, Anl_{opt} y Anl_{GPU} separado por T1-T5 y T6-T8 para el conjunto grande de matrices en las plataformas Turing (arriba) y Ampere (abajo).

La conclusión final que puede tomarse a partir de estas figuras consiste en que la optimización del segundo bloque parece ser un problema resuelto y futuros esfuerzos de optimización deberían concentrarse en las etapas de T1 a T5. En

concreto en la plataforma que utiliza la tarjeta AMPERE la importancia de esta etapa se reduce de $21,85\,\%$ a $6,5\,\%$ si se compara con Anl_{opt}. Esto sería mayor si se compara con una versión que aplique las optimizaciones a T2 y T5 pero mantenga la versión original de T6.

Capítulo 4

Etapa de resolución

En este capítulo se aborda la etapa de resolución de los *solvers* triangulares de dos etapas propuestos por Dufrechou y Ezzatti en [16]. De las tres variantes presentadas, la primera no requiere información adicional por lo que no ejecuta etapa de análisis. Por el contrario las otras dos utilizan los análisis abordados en la sección anterior.

Es importante destacar que ninguna de las tres estrategias supera a las otras en todo el conjunto de matrices estudiado. Si bien en general $solver_{mr}$ es la que cuenta con la etapa de resolución más rápida, existen matrices que obtienen mejores resultados en los otros *solvers*. Además, si se tiene en cuenta la etapa de análisis, también se debe sopesar la cantidad de resoluciones a computar. En otras palabras, dependiendo de las características y estructuras de las matrices, así como de la cantidad de ejecuciones a realizar, se determina cuál de los *solvers* debería utilizarse. Por lo anterior es interesante cualquier optimización que se pueda hacer a alguno de éstos.

Los tres *solvers* utilizan los mismos principios y consisten en un *loop* principal que ejecuta en modo de espera activa (*busy waiting*) sobre los elementos de los que depende la o las filas que se encuentra procesando. Una vez que se termina el *loop* principal hay una etapa de reducción en la que los distintos hilos que trabajan para la misma fila suman sus resultados parciales. Por último, el primer hilo de cada *warp* o grupo debe escribir en memoria global el resultado y marcar la fila como resuelta.

Como norma general el uso de *busy waiting* implica que se consuman ciclos de procesadores con cómputo inútil. Sin embargo, se puede explotar la arquitectura de la GPU para evitar esto. Como se describe en la Sección 2.3 las tarjetas gráficas no cuentan con memorias *cache* lo suficientemente grandes para ocultar el costo de acceso a memoria, por lo tanto se utiliza otra estrategia para lograr dicho objetivo. Cuando un *warp* se encuentra ocioso esperando por el resultado de un acceso a memoria o la disponibilidad de cierta funcionalidad este es removido de la ejecución por uno de los *warp schedulers* del SM y es intercambiado por un *warp* que se encuentre listo para ejecutar (llamado *eligible warp*) [4]. Esto genera que, en la práctica, $CUDA$ bloquee a los *warps* que acceden a

memoria para consultar si cierta fila de la que dependen está lista y por tanto estos no desperdician tiempo de cómputo ni impiden que otros *warps* ejecuten su próxima instrucción.

Las rutinas de resolución $solver_{ord}$ y $solver_{smpl}$ procesan cada fila en un *warp* distinto para evitar *deadlocks* de manera similar a la etapa T2 del análisis. Matrices que tengan, en general, filas con pocos no-ceros desperdician cómputo en hilos de un *warp* que no hacen cálculos. Utilizando *cooperative groups* se puede agrupar filas en estos casos garantizando que no habrán *deadlocks*, en la Sección 4.2 se explora esta optimización para $solver_{smpl}$.

4.1. Rutina original de $solver_{smpl}$

En esta sección se presenta la rutina original $solver_{smpl}$, que resuelve el sistema triangular sin información previa. Esta rutina asigna las filas a *warps* en orden de ejecución, es decir, tan pronto como se activan, y estos iteran sobre las filas haciendo operaciones de "suma y multiplicación" cada vez que avanzan. En caso de que una incógnita necesaria esté todavía sin calcular se quedan haciendo espera activa como se mencionó anteriormente.

Como se explicó en el Capítulo 2, los cambios a versiones más nuevas generaron que dejara de producirse la sincronización implícita entre hilos de un mismo *warp*. Dado que esta rutina (tanto como los otros solvers y el kernel de análisis) dependía de esta sincronización, en primer lugar fue necesario actualizarla para a las nuevas arquitecturas agregando sincronizaciones explícitas entre los *warps* en algunos puntos.

Otro cambio menor fue modificar la asignación de *warps* a filas. En el pasado esto se hacía de manera estática utilizando los índices de *warp* y bloque, lo cual fue modificado para asignar dinámicamente mediante una variable global y sumas atómicas. El mecanismo anterior asumía que los bloques se irían activando en orden, algo que si bien era cierto en la práctica no está garantizado por *CUDA*. Técnicamente es posible que una arquitectura o tarjeta específica active los bloques en distinto orden, dejando que hayan bloques activos que dependan de inactivos lo que podría generar un *deadlock*. La asignación utilizando `atomicAdd` genera que cada vez que hay un bloque activo se le asigne un conjunto de filas tales que todas sus dependencias se encuentran en bloques activos o ya fueron procesadas.

El Algoritmo 6 presenta la rutina de $solver_{smpl}$ incorporando las adaptaciones mencionadas anteriormente para su funcionamiento en nuevas tarjetas. Los datos de la matriz dispersa a procesar se encuentran guardados en formato CSR. Se define una constante $warp\_size$ con el valor 32, esta constante es la cantidad de hilos que constituyen un *warp*.

Para cada hilo se calcula un valor $local\_warp\_id$ que identifica a cada *warp* de manera única; algo similar se hace a nivel de hilo dentro del *warp* asignando un valor relativo que se identifica en la variable lne. Ambos valores pueden ser calculados utilizando módulo y división entera con respecto a $warp\_size$. Una vez que se cuenta con estos resultados se puede procesar la matriz de forma

organizada.

Luego de calcular estos valores se obtiene, para cada bloque, un identificador que le permitirá definir a cada *warp* qué fila procesará. Este contador es actualizado utilizando suma atómica para evitar condiciones de carrera. Los *warps* de un mismo bloque terminan procesando filas contiguas. Las filas son repartidas entre los *warps* en función de la variable *local_warp_id* mencionada anteriormente.

Posteriormente se itera sobre cada uno de los elementos no ceros que tiene la fila, cada thread se encarga de obtener el valor de la variable y multiplicarlo por la constante, acumulando en *left_sum*. Una vez calculado este valor, se pasa al siguiente elemento en la fila con un offset de *warp_size* y vuelve a realizar el cálculo, si ya no existen más elementos para procesar, se termina de iterar y se espera a que todos los thread hayan terminado.

Lo último es juntar los valores parciales que calculó cada hilo, guardarlos en el vector solución de la variable y marcar la fila como "resuelta" en el arreglo *is_solver*.

Esta rutina fue utilizada como línea base para la evaluación.

4.2. Optimizaciones a $solver_{smpl}$

En esta sección se presentan las optimizaciones hechas a las línea base. Como puede verse, el *kernel* que implementa la rutina de resolución es similar a la etapa T2 del análisis. En concreto la principal diferencia es que se utiliza punto flotante en lugar de enteros, y la reducción se implementa utilizando la suma en lugar del máximo. Por lo anterior es natural intentar aplicar las mismas estrategias de optimización que se plantearon para abordar dicha etapa anteriormente. En este sentido se evaluaron todas las propuestas de mejora hechas en el Apartado 3.3.1.

Con respeto al agrupamiento de filas es fácil ver que es aplicable de manera directa. Sin embargo, utilizar la reducción por hardware no es posible dado que en las versiones actuales de $CUDA$ ésta es brindada únicamente para enteros. Dado que las matrices, en general, están en punto flotante, se debe mantener la reducción por software. El Algoritmo7 muestra la nueva implementación de la rutina. Esta rutina depende del tamaño de partición definido por `tile_size`. La diferencia con la versión anterior consiste en que, utilizando *cooperative groups*, se agrupa a los hilos de cada *warp* según la fila que procesan permitiendo que la sincronización posterior al *loop* principal, la reducción y la escritura a memoria global se hagan independientemente para cada grupo.

4.2.1. Evaluación experimental

Debido a la similitud entre el solver y el kernel de análisis es esperable que se obtengan mejoras similares. Se implementó una nueva variante ($solver_{smpl-coop}$) siguiendo el Algoritmo 7. Los resultados de tiempos de ejecución comparados con $solver_{smpl}$ se muestran en la Tabla 4.1.

Algoritmo 6 $solver_{smpl}$

```
local_warp_id = threadix.x / warpsize
lne = threadIdx.x MOD warpsize

if(threadIdx.x == 0)
    s_row = atomicAdd(group_coutner, 1) * WARP_PER_BLOCK
End if

__syncthreads()
wrp = s_row + local_warp_id
row = row_ptr[wrp]
nxt_row = row_ptr[wrp + 1]
off = row + myTile.thread_rank()

while(off < nxt_row - 1)
    my_val = val[off]
    colidx = col_idx[off]
    if(is_solved[cold_idx])
        left_sum = left_sum - my_val * x[col_idx]
        off = off + warp_SIZE
    End if
 End while

  __syncthreads()

 for (int i = 16; i >= 1; i /= 2)
    left_sum = left_sum + __shfl_down_sync(left_sum, i)
 endfor

 if(lne == 0)
    x[groupId] = left_sum * piv
    is_solved[wrp] = 1
 End if
```

De los resultados hallados en la Tabla 4.1 podemos dividir las matrices en tres grandes grupos. El primero está compuesto por las matrices que no tienen cambios significativos (variación menor a 10 %), este grupo está compuesto por 6 matrices: `hollywood-2009`, `ship_003`, `crankseg_1`, `cant`, `exdata_1` y `TSOPF_RS_b300_c2`. Estos resultados son esperables dado que todas estas matrices tienen un promedio de no-ceros por fila mayor o igual a 32 con lo cual no son afectados por estos cambios.

Dentro de las matrices que sí son afectadas por estos cambios, solamente el 25 % es afectado negativamente mientras que en el otro 75 % tiene resultados positivos. Como era esperable, los resultados en esta etapa tienen una similitud con los que se encontraron en el Apartado 3.3.1, a excepción de la matriz

Algoritmo 7 Nueva rutina $solver_{smpl}$ aplicando el agrupamiento de filas ($solver_{smpl-coop}$)

```
1  thread_block_tile<tile_size> myTile
2  local_group_id = threadix.x / tile_size
3  groupsPerwarp = warp_SIZE / tile_size
4
5  if (threadIdx.x == 0)
6      s_row = atomicAdd(group_coutner, 1) * warp_PER_BLOCK
7      * groupsPerwarp
8  End if
9
10 this_thread_block().sync()
11 groupId = s_row + local_group_id
12 row = row_ptr[groupId]
13 nxt_row = row_ptr[groupId + 1]
14 off = row + myTile.thread_rank()
15
16 while (off <= nxt_row - 1)
17     my_val = val[off]
18     colidx = col_idx[off]
19     if{is_solved[cold_idx]}
20         left_sum = left_sum - my_val * x[col_idx]
21         off = off + tile_size
22    End if
23 End while
24
25 myTile.sync()
26 for (int i = myTile.size(); i >= 1; i /= 2)
27   left_sum = left_sum + myTile.shfl_down(left_sum, i)
28 endfor
29
30 if (myTile.thread_rank() == 0)
31     x[groupId] = left_sum * piv
32     is_solved[wrp] = 1
33 End if
```

`wiki-Talk` que pasa a tener resultados positivos en lugar de empeorar.

La motivación principal de estos cambios proviene del comportamiento de $solver_{smpl}$ con matrices de baja densidad. En estos casos las matrices cuentan con muchas filas con menos de 32 elementos, lo que genera que se desperdicie una gran cantidad de cómputo al no trabajar todo el *warp*. En este sentido, aunque de manera primitiva, el asignar grupos de filas a *warps* en función de la densidad de la matriz busca aliviar este problema.

Como se explicó anteriormente es común que las aplicaciones de la $SpTrSv$ requieran resolver muchas veces el mismo sistema con un distinto valor de b. En estos casos se puede reutilizar el análisis y solo ejecutar la etapa de resolución.

Matriz	$solver_{smpl}$	$solver_{smpl-coop}$	**Speedup**
1	0,79	1,29	0,61
2	114,81	114,31	1,00
3	20,49	13,76	1,49
4	36,59	9,85	3,71
5	74,16	17,23	4,30
6	5,59	5,56	1,01
7	5,63	1,38	4,09
8	8,31	3,06	2,72
9	8,22	8,25	1,00
10	4,53	4,60	0,98
11	0,22	0,17	1,33
12	0,29	0,16	1,86
13	2,64	0,74	3,56
14	3,48	1,16	3,00
15	0,23	0,13	1,77
16	16,94	5,18	3,27
17	2,29	2,30	0,99
18	178,28	280,16	0,64
19	2,44	2,83	0,86
20	0,14	0,25	0,57
21	0,78	0,79	0,99
22	0,61	0,30	2,04

Tabla 4.1: Comparación del tiempo total de ejecución (en ms) de los solvers, la línea base ($solver_{smpl}$) y la nueva propuesta ($solver_{smpl-coop}$) en la plataforma con arquitectura Turing, *Nvidia* 2080.

En este sentido, a la hora de evaluar las distintas rutinas es razonable hacer evaluaciones que contemplen varias iteraciones de la etapa de resolución y una única ejecución de la etapa de análisis. En estos casos es esperable que $SpTrSv_{mr}$ obtenga mejores resultados pues compensa el costo de la ejecución del análisis con un menor costo de la etapa de resolución al hacer una partición de las filas de manera más inteligente.

En la Figura 4.1 se presentan los resultados de *speedup* de la linea base $solver_{smpl}$ y $solver_{smpl-coop}$ tomando como referencia $solver_{mr}$ para 10 iteraciones ($t_{mr}/t_{smpl/smpl\_coop}$). La Figura 4.2 muestra los mismos resultados pero tomando en cuenta el tiempo de análisis. En ambas gráficas las matrices se encuentran ordenadas de menor a mayor densidad. Se utilizó el criterio de usar el resultado de ejecutar 10 iteraciones de resolución porque permitían una mejor visualización de las gráficas, los resultados con 100 resoluciones siguen la misma tendencia.

Es importante destacar que los resultados, en ambas tarjetas, tienden a ubi-

carse por debajo de 1 o ligeramente por arriba. Esto es esperable cuando no se toma en cuenta el análisis o se hacen suficientes iteraciones. Estos cambios no tienen por objetivo desarrollar una rutina de resolución más rápida que $solver_{mr}$ sino reducir la brecha (especialmente en matrices con poca densidad) lo que permita que en casos en los que se requieren pocas resoluciones sea una alternativa competitiva.

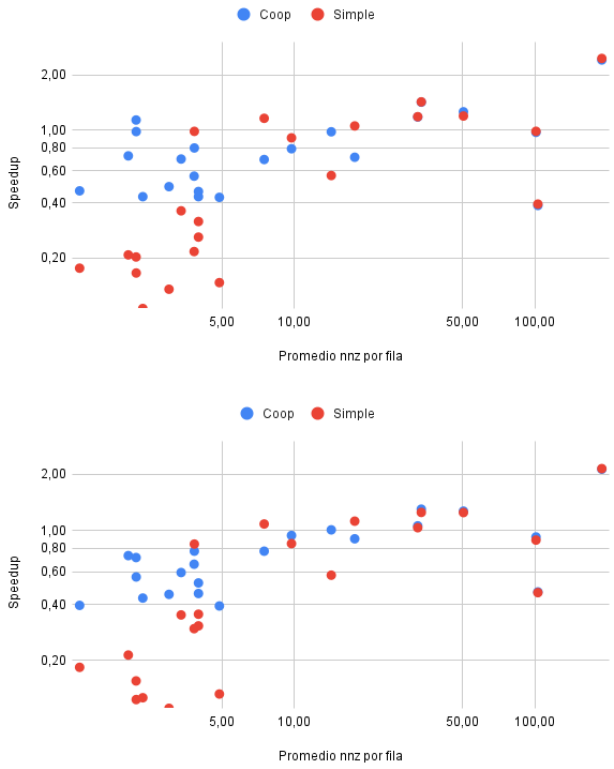

Figura 4.1: Factor de *speedup* relativo (sin análisis) entre las rutinas $solver_{smpl}$ (línea base) y $solver_{smpl-coop}$ con $solver_{mr}$ en las plataformas Turing (arriba) y Ampere (abajo).

Como puede verse en la Gráfica 4.1 los *speedups* de $solver_{smpl}$ contra $solver_{mr}$ tienden a estar por debajo de 0,4 para las densidades bajas ($d \leq 5$) con un mínimo en $0,15$. Esto se debe al poco aprovechamiento de la GPU. Por el contrario $solver_{smpl-coop}$, en ese mismo cuadrante, tiene *speedups* mayores a $0,4$ en general y alcanzando en algún caso valores cercanos a 1. Al aumentar la densidad los resultados comienzan a ser más mezclados y, como era de esperarse, con densidades mayores a 32 los resultados son prácticamente idénticos. Cuando se agrega el análisis, como muestran la Figura 4.2, se mantiene el comportamiento solo que en este caso se obtienen más valores que superan el 1 especialmente en

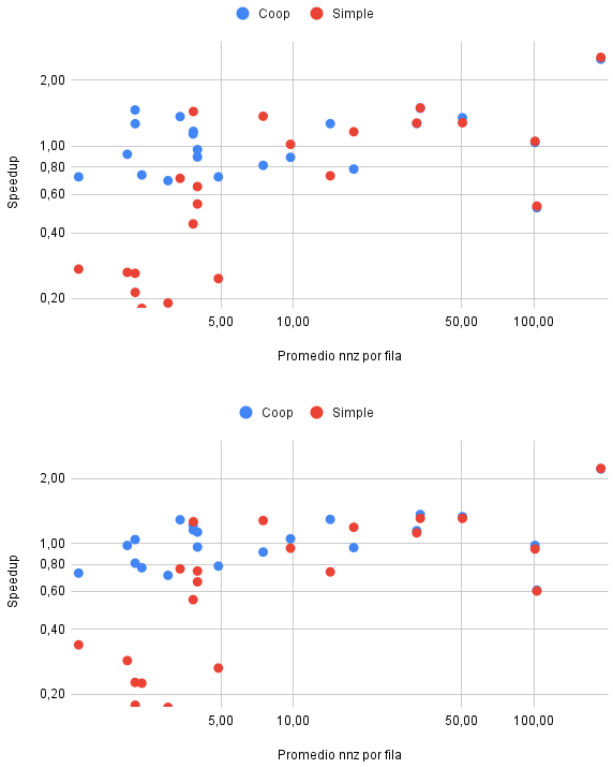

Figura 4.2: Factor de *speedup* relativo (con análisis) entre las rutinas $solver_{smpl}$ (línea base) y $solver_{smpl-coop}$ con $solver_{mr}$ en las plataformas TURING (arriba) y AMPERE (abajo).

densidades mayores.

Capítulo 5

Comparación con otros solvers

En capítulos anteriores se presentaron propuestas de mejora tanto a la etapa de análisis como a la etapa de resolución de la operación $SpTrSv$ y, para cada uno de estos cambios, se mostraron resultados experimentales que los justifican. En este capítulo se presenta una comparación experimental de las rutinas trabajadas junto a otras implementaciones disponibles de manera pública. En la Sección 5.1 se comentan las distintas propuestas a comparar y en la Sección 5.2 se muestran los resultados.

5.1. Solvers externos

En esta sección se presentan brevemente los distintos algoritmos para resolver sistemas triangulares dispersos contra los que se compara nuestra implementación.

De la revisión efectuada parece evidente que la estrategia $Sync - Free$ es el estado del arte en la actualidad para la resolución de estos sistemas en GPU. Entre las rutinas evaluadas se agrega CUSPARSE a efectos de comparación. Entonces, los algoritmos evaluados son los siguientes:

- CUSPARSE SPTRSV V2
- Sync-Free (Liu et al.) [27]
- CapelliniSPTRSV (Su et al.) [35]
- YuenyeungSpTRSV (Zhang et al.) [38]

En los siguientes apartados se presentan brevemente las distintas bibliotecas y sus estrategias para la resolución de sistemas triangulares. Las explicaciones son poco profundas y, de ser necesaria una mayor comprensión de dichas propuesta, se sugiere al lector consultar las publicaciones originales respectivas.

5.1.1. Sync-Free

El algoritmo asigna un *warp* para cada columna de la matriz dispersa. Siguiendo la misma lógica que en el algoritmo paralelo una vez que una incógnita está procesada puede impactarse en todos los elementos del vector resultado que dependan de esta incógnita. Cada thread del *warp* es el encargado de aplicar una "multiplicación y suma" en el resultado de su respectiva fila. Una explicación en detalle se presenta en el Algoritmo 3 del Capítulo 2.

Dado que esta implementación utiliza el formato CSC, no es fácil obtener, para cada entrada del resultado, de cuántas incógnitas depende. Para esto se utiliza la etapa de preprocesamiento, que genera una suerte de histograma de dichas dependencias. Cada *warp* queda en *busy waiting* hasta que se han procesado todas las incógnitas de las que depende que no fueron asignadas al mismo bloque.

Finalmente cuando se hace el cálculo de la columna el hilo correspondiente a la diagonal escribe en memoria y notifica a todos quienes dependen de su incógnita que ésta ha sido resuelta.

5.1.2. Capellini $SpTrSv$

Este algoritmo es una continuación del mencionado anteriormente, utilizando el formato CSR. A cada *warp* se le asignan $warp\_size$ componentes (o filas). Debido a que este algoritmo procesa más de una fila en el mismo *warp* existe la posibilidad de que se produzca un *deadlock* si las filas dependen entre ellas debido al modelo de ejecución de $CUDA$. Para solucionar este problema se divide la ejecución de los *warps* en dos partes, una primera que procesa los elementos que no dependen entre sí y luego un *for* que procesa los elementos que tienen dependencia dentro del mismo *warp*.

A diferencia del algoritmo anterior, se remplazó el arreglo que mantiene la relación de dependencia entre *warps* y filas. Este se substituye por un arreglo get_{value} booleano de largo igual a cantidad de filas de la matriz. Cada posición en el arreglo representa el estado de la incógnita correspondiente a la fila, un valor en $false$ representa que la variable no fue resuelta y un valor $true$ representa que si fue resuelta. Con este cambio, se puede liberar los threads para que trabajen sobre los elementos de la fila sin tener que esperar a que todas las incógnitas sean resueltas. Cada thread verifica el estado de la variable, si ésta se encuentra en el estado de no resuelta, el thread realiza *busy waiting* hasta que cambie de estado y pueda procesar el elemento.

Por último se acumulan los resultados parciales obtenidos luego de calcular todos los elementos no ceros de la fila, esto da como resultado el valor de la incógnita correspondiente a la fila. Luego se coloca el valor de get_{value} de la fila en $true$.

5.1.3. Yuenyeung $SpTrSv$

Nuevamente, este método es una extensión del trabajo realizado en el algoritmo *Capellini*. Se mantiene el arreglo get_{value} para sincronizar los threads a medida que necesitan acceder al valor de una variable. Este nuevo algoritmo agrega dos etapas al momento de hallar la solución.

El algoritmo Yuenyeung utiliza un preprocesamiento para determinar qué componentes serán procesados con un paralelismo a nivel de *warp* y cuáles a nivel de hilo (apartado anterior) en función de una métrica definida por los autores.

Para evitar *deadlocks* se utiliza una estrategia similar de dos etapas a la presentada para Capellini.En la primera etapa se identifican los elementos que no tienen dependencia entre *warps*, es decir elementos que no dependen de una fila asignada a su propio *warp*. Esto permite garantizar que un *warp* podrá resolver toda una fila sin esperar por una variable que aún está siendo procesada. Este dato se obtiene mediante un análisis liviano que se realiza previamente a la ejecución del solver. Como estas filas no presentan problemas de *deadlock* y no dependen entre si, son las primeras en resolverse, actualizando el valor correspondiente a la variable en el arreglo get_{value}.

En la segunda etapa del algoritmo se trabaja sobre los elementos que tienen dependencia entre *warps*. Cada *warp* resuelve $warp\_size$ filas de la matriz dispersa, donde cada thread del *warp* trabaja sobre una fila. Al igual que en la implementación anterior, se consulta en el arreglo *get_value* el estado de la variable. Si la variable aún no fue resuelta, el thread queda en *busy wating* hasta que el estado cambie a resuelto y el thread pueda avanzar con el próximo elemento de la fila. Cuando termina de calcular el resultado, actualiza el valor correspondiente a la variable en *get_value* para que sea utilizada.

5.2. Evaluación Experimental

En esta sección se presentan los resultados experimentales para las tres rutinas externas presentados en la sección anterior y las tres rutinas propias abordadas en este trabajo. Además se toman los resultados de las rutinas de la biblioteca CUSPARSE como referencia.

Como se explicó anteriormente es relevante la cantidad de iteraciones del solver por cada vez que se utiliza la rutina de análisis. En este sentido se optó por hacer tres comparaciones distintas que implicaron 1, 10 y 100 iteraciones de la rutina de resolución. La experimentación se realizó en las dos tarjetas mencionadas en capítulos anteriores y sobre el conjunto de matrices presentado en el Apartado 3.1. Estas matrices fueron ordenadas en tamaño creciente según la cantidad de no-ceros.

Debido a la gran cantidad de resultados se optó por presentarlos en forma de gráfica. A efectos de permitir una mejor visualización, y debido a lo dispar de los tiempos de ejecución, se utilizó como medida el factor de *speedup* contra CUSPARSE como métrica de comparación. La Figura 5.1 presenta los resultados

para una única iteración del *solver*, la Figura 5.2 para diez iteraciones y por último la Figura 5.3 para cien iteraciones de la rutina de resolución.

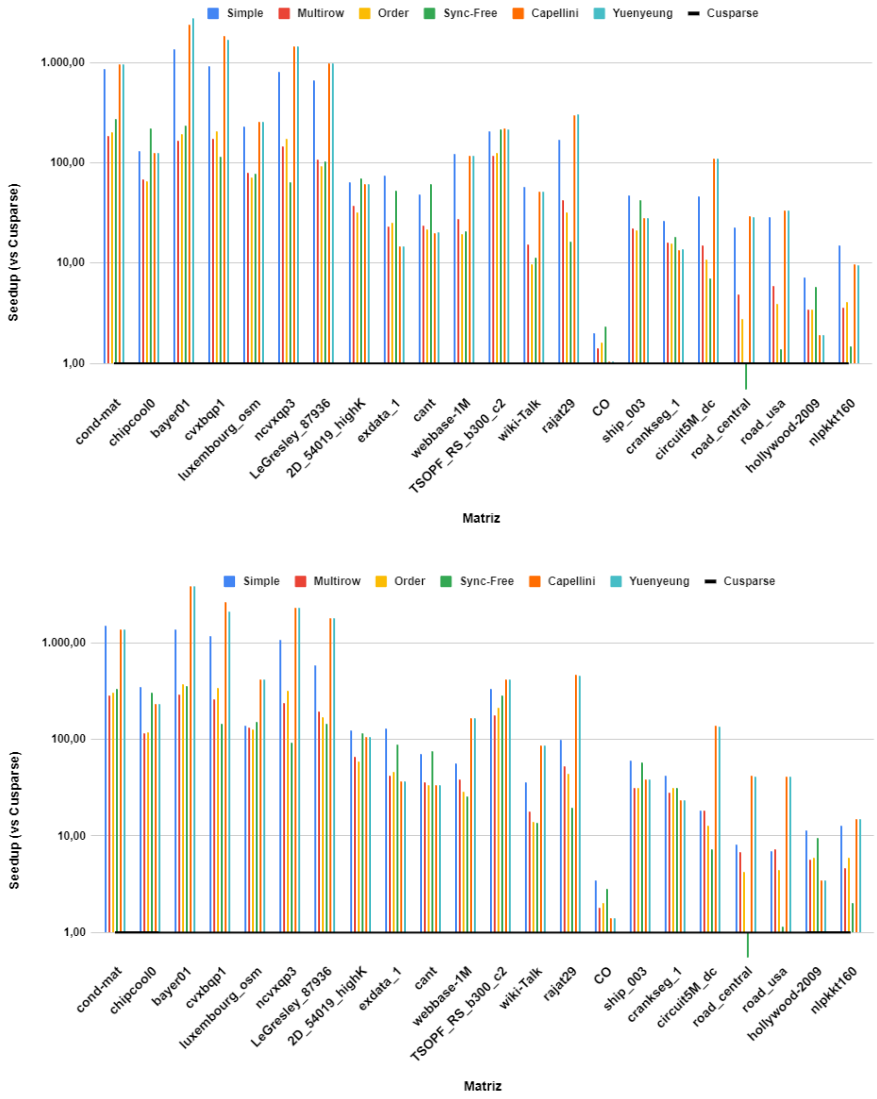

Figura 5.1: Comparación del *speedup* relativo a CUSPARSE de las distintas implementaciones para 1 iteración del *solver* en la GPU TURING (arriba) y AMPERE (abajo).

Los resultados de *speedup* en el caso de una única iteración del *solver* son

en general grandes debido, en gran medida, a que el análisis de CUSPARSE es órdenes de magnitud más costoso que las otras etapas de análisis. Esto genera que en este caso, donde ya de por sí aplicar el análisis por lo general no es eficiente, se obtengan *speedups* de hasta 3000× en la plataforma TURING. Es importante notar que, en casos en que se resuelve una única vez, estrategias con una rutina de análisis como la de CUSPARSE no tienen sentido y se optaría por alternativas que no utilicen análisis ($solver_{smpl}$) o cuyo análisis sea casi insignificante (Yuenyeung y Capellini). Estos resultados se brindan a efectos de completitud y para poder comparar estas alternativas.

En esta línea los resultados, en ambas tarjetas, siguen la regla general de que los mejores resultados son obtenidos por quienes requieren menor tiempo de análisis. De los *solvers* trabajados, el que obtiene mejores resultados en todos los casos es $solver_{smpl}$ que además se mantiene competitivo con Yuenyeung y Capellini. En concreto lo supera en varias matrices pequeñas y medianas, y mantiene resultados relativamente competitivos en algunas matrices grandes.

Finalmente es interesante mencionar que la rutina $Sync-Free$ de Liu et al. es la única que en alguna matriz es superada por CUSPARSE, y en general en las matrices grandes tiene resultados relativos malos.

En la Figura 5.2 puede verse cómo al aumentar la cantidad de iteraciones $solver_{mr}$ mejora significativamente sus resultados, siendo competitivo con la nueva versión de $solver_{smpl}$. En particular $solver_{smpl}$ es mejor en las matrices pequeñas y $solver_{mr}$ obtiene mejores resultados en las matrices más grandes. Sin embargo, en prácticamente todas las matrices, Yuenyeung y Capellini continúan siendo las que obtienen mejores resultados.

Nuevamente la propuesta de Liu et al. es superada por CUSPARSE pero en este caso también Yuenyeung y Capellini son superadas por la biblioteca en la matriz `CO`.

Finalmente, como puede verse en la Figura 5.3, al aumentar la cantidad de iteraciones $solver_{mr}$ se convierte en la mejor rutina en la mayoría de las matrices.

En general la propuesta $Sync-Free$ de Liu et al. solo tiene buenos resultados en tres matrices y, a excepción de alguna matriz pequeña, no es competitiva. Con matrices de mayor tamaño esta rutina es peor que CUSPARSE, probablemente debiéndose a la gran cantidad de sumas atómicas que se emplean, que en matrices de mayor tamaño restringe demasiado el paralelismo.

Las rutinas $solver_{smpl}$ y $solver_{ord}$ también logran tener buenos resultados en dos y una matriz respectivamente, pero en el resto de los casos generalmente no son competitivas. A diferencia de la rutina anterior éstas no tienen un claro empeoramiento relativo al aumentar el tamaño y no es tan claro qué factor afecta a sus resultados.

Las rutinas Yuenyeung y Capellini mantienen resultados similares prácticamente en todos los casos. Esto es esperable dado que las variaciones presentadas en [38] únicamente afectan casos puntuales (cuando el promedio de filas por nivel es alto y el promedio de no-ceros por fila es bajo). Por estos motivos, generalmente Yuenyeung es mejor o igual que Capellini.

Por último, Yuenyeung/Capellini supera a $solver_{mr}$ en cuatro matrices. En

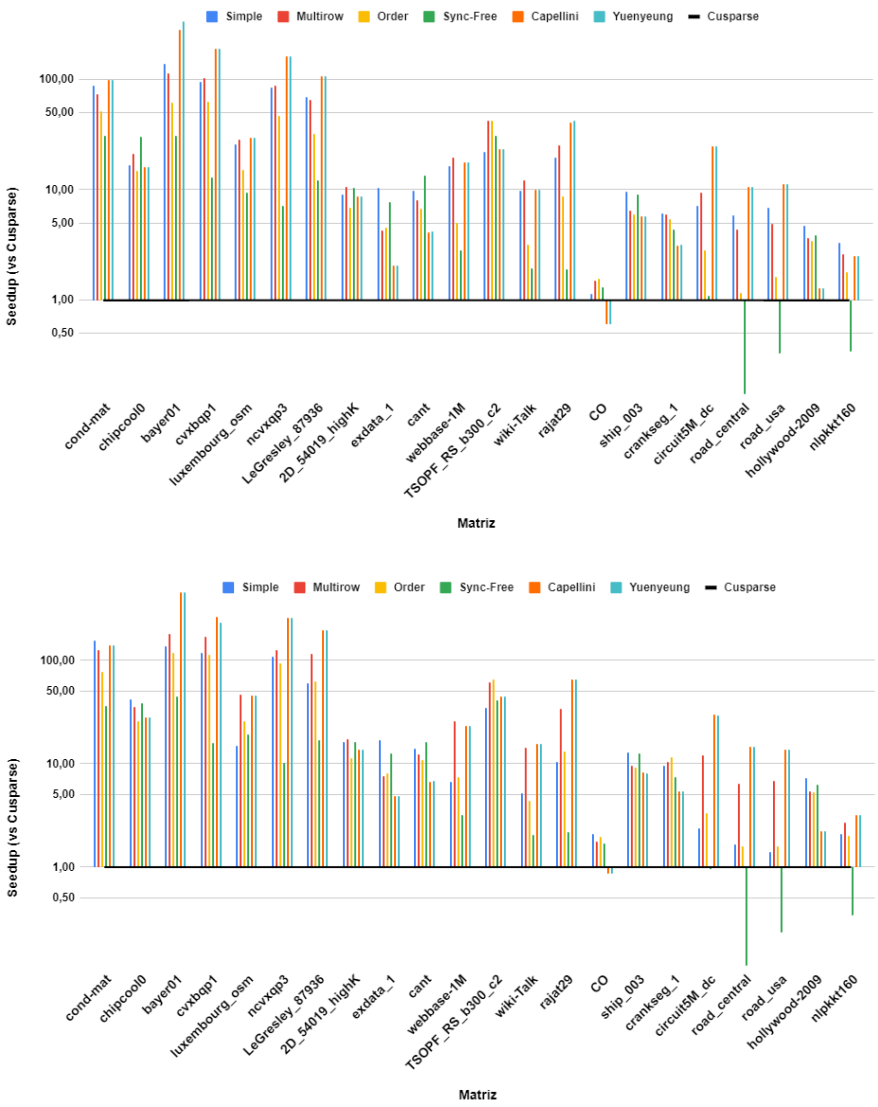

Figura 5.2: Comparación del *speedup* relativo a CUSPARSE de las distintas implementaciones para 10 iteraciones del *solver* en la GPU TURING (arriba) y AMPERE (abajo).

todas ellas, alguna de estas rutinas es la que obtiene los mejores resultados. La primera de estas, `bayer01`, es un caso excepcional debido a que la distancia es pequeña y la ventaja sobre nuestra rutina se basa exclusivamente en el tiempo

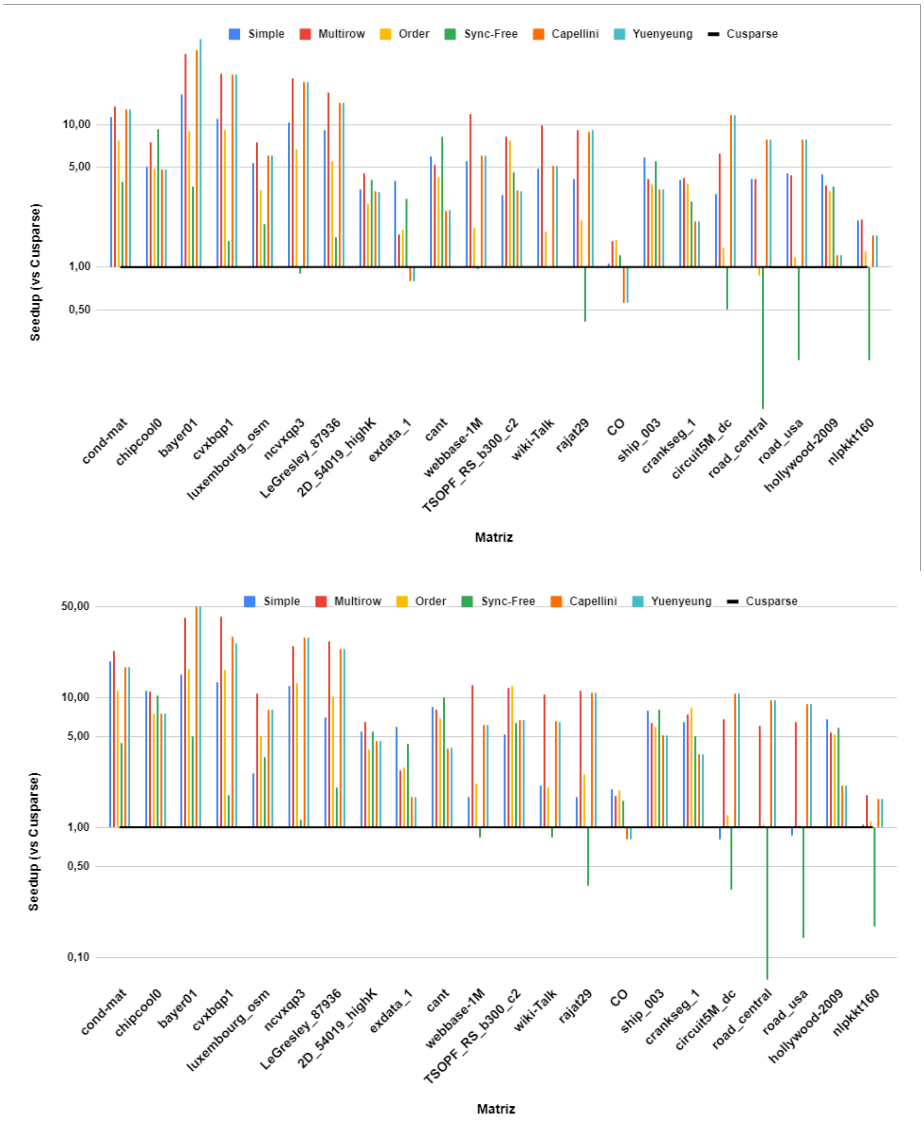

Figura 5.3: Comparación del *speedup* relativo a CUSPARSE de las distintas implementaciones para 100 iteraciones del *solver* en la GPU TURING (arriba) y AMPERE (abajo).

de análisis (0,96 contra 0,01) y si se aumentan las iteraciones dicha ventaja desaparecerá. Con respecto a las otras tres matrices es interesante notar que están entre las cinco con mayor tamaño evaluadas. Si bien a priori parece indicar

que estas rutinas muestran mejora relativa respecto a $solver_{mr}$ al agrandar las matrices, esto no es coherente con otros resultados hallados, tanto en este trabajo como en otros previos.

En este sentido se decidió ampliar el análisis tomando en cuenta la densidad de las matrices (nnz/dim) y la desviación estándar de la cantidad de no-ceros por fila. A efectos de concentrar el análisis se optó por comparar las rutinas $solver_{mr}$ y $solver_{smpl}$ contra Yuenyeung dado que son las que generalmente obtienen los mejores resultados. En la Figura 5.4 se muestra el *speedup* relativo de las rutinas de resolución trabajadas contra Yuenyeung en función de la desviación estándar y la densidad, respectivamente.

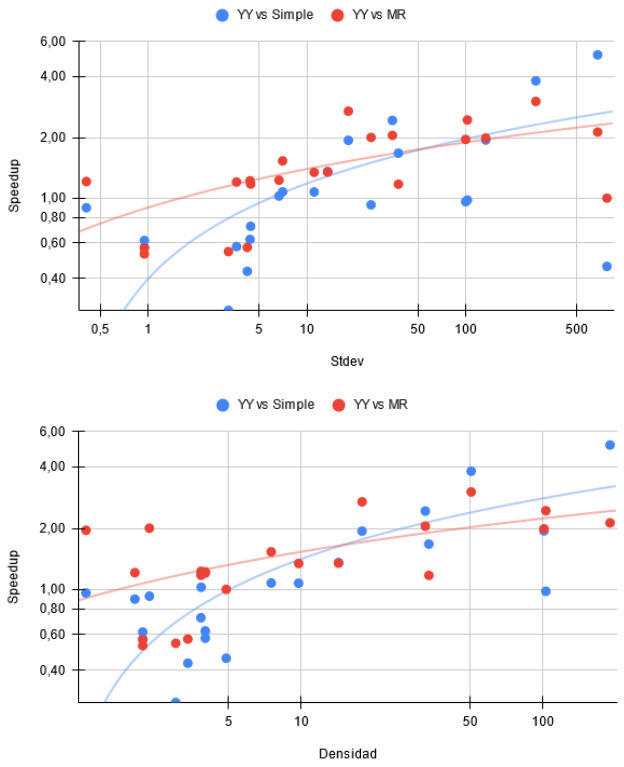

Figura 5.4: Factor de *speedup* relativo entre las rutinas $solver_{mr}$ y $solver_{smpl}$ contra Yuenyeung en función de la densidad de las matrices en las plataformas Turing (arriba) y Ampere (abajo).

Como puede verse, en general al aumentar tanto la densidad como la desviación estándar los resultados relativos de nuestras rutinas mejoran con respecto a la rutina Yuenyeung. Es interesante ampliar este estudio para una mayor cantidad de matrices a efectos de confirmar esta tendencia. Otra observación relevante consiste en que si bien las líneas de tendencia parecerían indicar que

al crecer tanto desviación estándar como densidad la rutina $solver_{smpl}$ mejora a $solver_{mr}$, esto se debe en gran medida a la matriz `exdata_1` que en todo el trabajo se ha comportado como anomalía estadística. Su alta desviación estándar y densidad (676 y 189 aproximadamente) junto a su pequeñísimo tamaño (6001 filas) hacen que su comportamiento sea una excepción en prácticamente todas las evaluaciones. De retirar esta matriz, en ambos casos las líneas de tendencia de $solver_{mr}$ superan a su contraparte.

Los resultados anteriores parecen indicar que las rutinas desarrolladas superan a Yuenyeung/Capellini al aumentar la densidad y/o la desviación estándar. Esto resulta intuitivo dado que las últimas emplean un hilo por fila salvo en algunos casos de borde. Por el contrario la rutina $solver_{mr}$ evita desperdiciar tiempo de cómputo, agrupando las filas pequeñas y de igual tamaño, mientras que la versión de la rutina $solver_{smpl}$ presentada en el Capítulo 4 también presenta cierta adaptabilidad, definiendo la cantidad de hilos por fila en función de la densidad de la matriz.

Para explorar esta hipótesis se decidió ampliar el conjunto de matrices a evaluar. En este sentido se tomaron 100 matrices de la colección SuiteSparse en función de sus características. En concreto, se calcularon los cuartiles de la desviación estándar y cantidad de no-ceros. Se eligieron 25 matrices de manera aleatoria dentro de cada cuartil generado por la desviación estándar garantizando que las matrices de cada cuartil estuvieran equitativamente distribuidas entre los cuartiles generados por el número de no-ceros. En la Tabla 5.1, se presentan los límites superiores de los cuartiles para desviación estándar y no-ceros.

Cuartil	**Stdev**	**NNZ**
1	4,91	86625
2	23,05	351931
3	36,45	1839990,5
4	17514,2	165773835

Tabla 5.1: Extremos superiores de los cuartiles definidos.

Para esta evaluación se removieron los solvers $Sync - Free$ y $solver_{ord}$ porque en general eran superados por otros, y Capellini, por ser prácticamente indistinguible de Yuenyeung. Los resultados en tiempo de ejecución de esta nueva prueba se muestran en la Figura 5.5. El *speedup* mostrado en estas figuras considera el tiempo de ejecución del análisis más 100 iteraciones de la rutina de resolución comparando contra CUSPARSE.

Como puede verse en la Figura 5.5, en general $solver_{mr}$ es el que obtiene mejores resultados. En ambas tarjetas los resultados son mezclados cuando la desviación estándar está ubicada en el primer cuartil o en la mitad inferior del segundo pero al aumentarla rápidamente los puntos rojos comienzan a ubicarse en la parte superior de la gráfica.

Por otro lado $solver_{smpl}$ se mantiene competitivo contra Yuenyeung a lo largo de la gran mayoría de los valores de desviación estándar. Únicamente

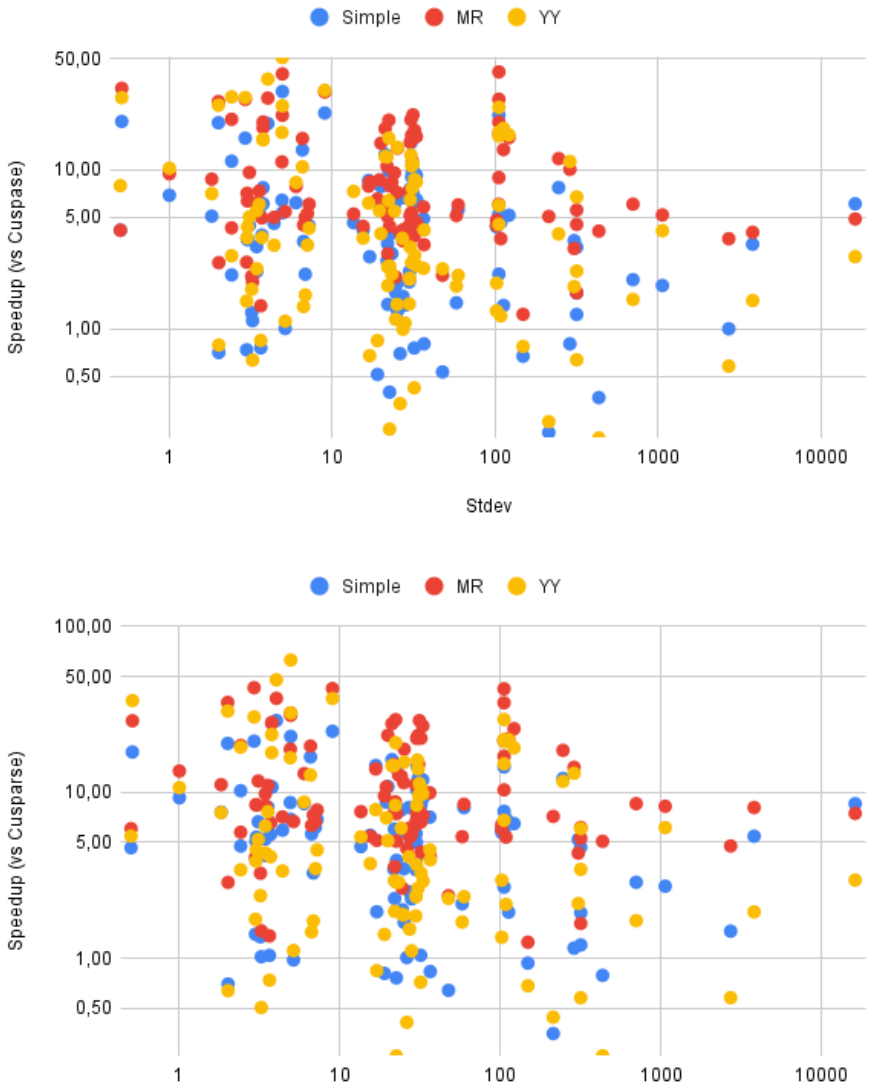

Figura 5.5: Comparación del *speedup* relativo a CUSPARSE de las distintas implementaciones para 100 iteraciones del *solver* en las plataformas TURING (arriba) y AMPERE (abajo).

cuando la desviación estándar supera el millar parece haber una tendencia a separarse, pero debido a la poca cantidad de muestras en ese entorno no pueden sacarse conclusiones. Sería interesante ampliar el estudio con matrices de alta

desviación estándar para considerar la dimensión y cantidad de no-ceros, pero no se cuenta en la colección con una cantidad suficiente de matrices para hacer evaluaciones concluyentes.

Finalmente, a efectos de mostrar visualmente una evaluación clara entre $solver_{mr}$ y Yuenyeung en la Figura 5.6 se presenta el *speedup* de $solver_{mr}$ sobre su contraparte (t_{YY}/t_{MR}). Nuevamente estos resultados muestran los tiempos para 100 iteraciones del solver y una ejecución de análisis en ambas plataformas.

Ambas imágenes muestran cómo, a medida que aumenta la desviación estándar, los puntos tienden a acumularse en valores más altos y, en particular, los valores menores a 1 dejan de aparecer. Esto permite concluir con relativa seguridad que la hipótesis planteada anteriormente sobre la relación entre el *speedup* y la desviación estándar es correcta. Por otro lado pueden mostrarse ejemplos de matrices de dimensiones (y cantidad de no-ceros) relativamente grandes en las que Yuenyeung es superado por $solver_{mr}$ mostrando que al aumentar la dimensión no necesariamente empeoran los *speedups*. Dos casos interesantes son las matrices `wb-edu` y `asia_osm` que con dimensiones relativamente grandes (casi 10 y 12 millones respectivamente) obtienen *speedups* superiores a 3×.

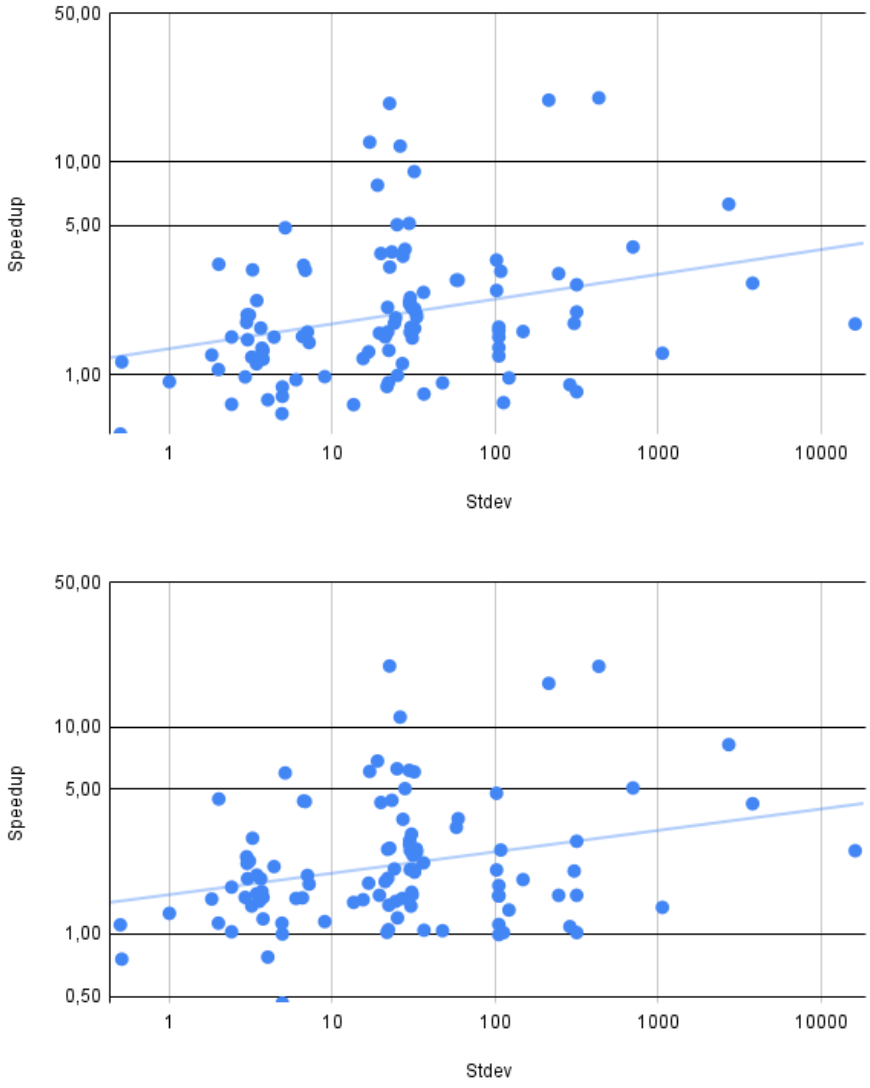

Figura 5.6: Speedup del Solver multirow vs Yuenyeung para 100 iteraciones del *solver* en las plataformas Turing (arriba) y Ampere (abajo).

Capítulo 6

Comentarios finales

En este capítulo se presentan observaciones finales de este trabajo. Primero en la Sección 6.1 se enumeran las principales conclusiones del trabajo. En la sección 6.2 se comentan los principales trabajos de difusión hechos en el marco de este proyecto de grado. Finalmente, en la Sección 6.3, se proponen las principales líneas de trabajo futuro

6.1. Conclusiones

En este trabajo se abordó el problema de la resolución de sistemas triangulares dispersos ($SpTrSv$) en GPU. Debido a que, en las aplicaciones en las que se utilizan este tipo de resoluciones, muchas veces es necesario resolver varios sistemas dispersos representados por la misma matriz cuya diferencia radica en el vector b, es oportuno buscar diferentes soluciones al sistema que aprovechen esta característica. En este sentido, Dufrechou y Ezzatti propusieron una implementación en GPU en dos etapas: una primera de análisis, que trabaja sobre las dependencias generadas por el patrón de no ceros de la matriz, y por tanto común para los sistemas, y una segunda encargada de la resolución numérica, que utiliza información generada por la primera etapa para organizar la ejecución de los hilos.

En general, la etapa de análisis tiene un peso importante en el tiempo de ejecución total. En este sentido, una línea de trabajo para la optimización de la $SpTrSv$ consiste en reducir el costo fijo producto de la etapa de análisis. En este trabajo se abordaron sucesivas mejoras a esta rutina centradas, en primera instancia, en la paralelización de etapas para su ejecución en GPU y optimizaciones utilizando nuevas arquitecturas. Posteriormente se presentó un cambio algorítmico a la rutina de análisis para que esta ejecute completamente en GPU. Estos cambios permitieron mejoras de hasta $4\times$ con respecto a la versión que ejecutaba solo algunas etapas en GPU, entre otros motivos, debido a evitar las transferencias de resultados intermedios entre la CPU y la GPU.

En cuanto a la etapa de resolución, se propusieron implementaciones que

permitieron mejorar los resultados de ejecución en matrices con una baja cantidad de no-ceros por fila buscando minimizar el desperdicio de los recursos de cómputo de la GPU. Esta idea también se aplicó para optimizar una etapa del análisis.

Finalmente se evaluaron las rutinas propuestas comparando con otras bibliotecas disponibles públicamente. En general los resultados mostraron que el paradigma sync-free es el estado del arte. En particular, que la rutina $SpTrSv_{mr}$ tiene los mejores resultados en la etapa de resolución y, si se alcanza determinado número de resoluciones por cada etapa de análisis, supera a las otras alternativas a pesar de requerir un análisis más costoso. En concreto, nuestra propuesta mejora relativamente a las alternativas más prometedoras ($Yuenyeung/Capellini$) cuando la desviación estándar de no-ceros por fila sube, es decir, se adapta mejor que éstas a las matrices más irregulares.

6.2. Difusión

Durante el desarrollo de este trabajo se hizo una presentación en el congreso internacional *22th International Conference Computational and Mathematical Methods in Science and Engineering*, que resulto en la invitación a publicar una versión extendida del trabajo en un *special issue* de la revista *Journal of Supercomputing*. Dicho trabajo se encuentra en evaluación.

En el trabajo de conferencia se presentaron ideas y resultados preliminares para paralelización del análisis en CPU. En particular, se exploró el uso de la biblioteca OpenMP para paralelización de una etapa del análisis y se mostraron estudios sobre la importancia de las distintas etapas.

Finalmente, en el trabajo enviado a la revista *Journal of Supercomputing* se presenta las dos alternativas para el análisis presentadas en este trabajo, así como las optimizaciones propuestas para la etapa de resolución. Se presentaron resultados experimentales comparando las rutinas contra CUSPARSE.

El código de este trabajo se encuentra publicado en GitHub[1]. En el Anexo 1 se describe la estructura y contenido de nuestra biblioteca.

6.3. Trabajo futuro

Del abordaje de las estrategias de sincronización en GPUs para el caso concreto de la operación $SpTrSv$ surgen varias líneas interesantes de trabajo futuro.

A partir de la arquitectura Volta las GPUs de *Nvidia* extendieron sus mecanismos de sincronización, y arquitecturas más nuevas como Ampere agregaron aceleración por *hardware* a mecanismos avanzados de sincronización. Debido a que son relativamente recientes y su aplicación a problemas de álgebra lineal es compleja, hay pocos trabajos de aplicación de estos mecanismos de sincronización a otros problemas distintos al $SpTrSv$. Una línea prometedora de trabajo

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

futuro es intentar aplicar estrategias avanzadas de sincronización a este y otros problemas del álgebra lineal dispersa.

Tanto este trabajo como otros relacionados están enfocados concretamente en las GPUs *Nvidia* y el lenguaje *CUDA*. Otra línea interesante de trabajo consiste en evaluar estas ideas en otras plataformas de cómputo paralelo como son las tarjetas gráficas AMD [3] o Intel [5]. Dado que las implementaciones propuestas utilizan particularidades del hardware será necesario hacer adaptaciones para poder replicar las estrategias.

Una línea de trabajo futuro es evaluar otras formas de organizar el procesamiento de las filas que permita reducir el tiempo de cómputo evitando *deadlocks*. Las filas son procesadas en distinto orden según la rutina. Por ejemplo, tanto en el *kernel* de análisis como $solver_{smpl}$ se utiliza el orden dado por el índice, mientras que en $solver_{ord}$ se tienen en cuenta los niveles, y en $solver_{mr}$ se agrega el tamaño.

Por último se plantea la posibilidad de explorar el efecto de modificar el tipo de dato tanto para los valores como para los índices. Hay trabajos que exploran distintos tipos de compresiones que permiten utilizar tipos de datos con menor cantidad de bits (por ejemplo punto flotantes en 16 bits) y es interesante evaluar cómo se complementan.

Bibliografía

[1] Converting dataframe to csr matrix. https://hippocampus-garden.com/pandas_sparse/. Access date: 2022-28-12.

[2] CUDA GPUs Compute Capabilites. https://developer.nvidia.com/cuda-gpus. Access date: 2022-21-11.

[3] Fundamentals of hip programming. https://developer.amd.com/resources/rocm-learning-center/fundamentals-of-hip-programming/. Access date: 2022-27-11.

[4] Issue efficiency. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/issueefficiency.htm. Access date: 2022-27-11.

[5] Latest intel graphics developer's guide. https://www.intel.com/content/www/us/en/developer/articles/guide/hd-graphics-developers-guides.html. Access date: 2022-27-11.

[6] Thrust: Algorithms. http://thrust.github.io/doc/group__algorithms.html. Access date: 2022-21-07.

[7] 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.

[8] Mikhail J. Atallah, editor. *Algorithms and theory of computation handbook*. CRC Press, pub-CRC:adr, 1999.

[9] Nathan Bell and Michael Garl. Efficient sparse matrix-vector multiplication on CUDA. Technical report, December 04 2014.

[10] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC*. ACM, 2009.

[11] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[12] Steven Dalton, Sean Baxter, Duane Merrill, Luke N. Olson, and Michael Garland. Optimizing sparse matrix operations on GPUs using merge path. In *IPDPS*, pages 407–416. IEEE Computer Society, 2015.

[13] Timothy A. Davis. *Direct methods for sparse linear systems*, volume 2 of *Fundamentals of algorithms*. SIAM, 2006.

[14] 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.

[15] 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 2018, Vancouver, BC, Canada, May 21-25, 2018*, pages 920–929. IEEE Computer Society, 2018.

[16] 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.

[17] 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.

[18] 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.

[19] Manuel Freire, Franco Seveso, Juan Ferrand, Ernesto Dufrechou, and Pablo Ezzatti. Accelerating the level-set analysis stage of a SpTrSV algorithm for GPUs. In *22th International Conference Computational and Mathematical Methods in Science and Engineering (CMMSE 2022), Cadiz, Spain, 2022*, 2022.

[20] 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.

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

[22] 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.

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

[24] 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.

[25] Arthur B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, 1962.

[26] David Blair Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors - A Hands-on Approach.* Morgan Kaufmann, 2010.

[27] 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.

[28] 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.

[29] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 9.2.89, 2018.

[30] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda c++ programming guide: 11.7.0, 2022.

[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] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986.

[33] Y. Saad and H. A. van der Vorst. Iterative solution of linear systems in the 20th century, February 27 2001.

[34] 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.

[35] 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. 2020.

[36] Wing, O. and Huang, J. W. A computation model of parallel solution of linear equations. *IEEE Trans. Comput.*, C-29:632–638, 1980.

[37] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS publishing, 1996.

[38] 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.

Anexo

La biblioteca brinda las implementaciones de las rutinas de análisis y resolución presentadas en este trabajo. A efectos de facilitar su uso se brinda una ejecución sencilla y se especifican una serie de pasos que son encapsulados en rutinas auxiliares.

El archivo `main.cu` cuenta con una implementación para generar una matriz en formato CSR a partir de un archivo *mtx*. Esta implementación no está optimizada y puede ser sustituida. Luego de generar la matriz este archivo invoca a la rutina de testeo que presenta una ejecución de ejemplo que sigue la secuencia de pasos establecidos. Estos son reservar la memoria en GPU, transferir la matriz, ejecutar el análisis y luego la resolución. Al finalizar se libera dicha memoria.

Se implementaron cuatro rutinas auxiliares para facilitar el uso de la biblioteca: dos para la gestión de la memoria, la rutina que se encarga del análisis y la equivalente para la resolución. En este sentido se implementaron rutinas que encapsulan cada una de estas etapas. En concreto se brindan funciones para reservar la memoria y cargar la matriz, ejecutar el análisis y ejecutar la resolución. Tanto las rutinas de análisis como de resolución engloban las distintas estrategias mencionadas anteriormente. En la Figura 1 se presenta el flujo de ejecución de la biblioteca.

La función `allocate_memory` recibe los tres vectores que representan a la matriz en CPU (`row_ptr`, `col_idx` y `val`) y devuelve una estructura del tipo `sp_mat_t`. Esta estructura se encuentra ubicada en memoria de la GPU y contiene la información de la matriz y datos requeridos para las etapas de análisis y resolución.

La rutina `run_analysis` recibe el modo de ejecución y ejecuta el análisis correspondiente (en el caso de simple no hace nada). La rutina crea y inicializa una estructura `dfr_analysis_info` con los campos correspondientes al análisis utilizado e información sobre el análisis que la generó.

La rutina `run_solver` por su parte es la encargada de despachar la implementación de la etapa de resolución correspondiente según el análisis que se ha utilizado. Para esto recibe como parámetro la referencia a la estructura `dfr_analysis_info` generada por el análisis y a partir de esta define qué rutina de resolución utilizar. Además recibe los parámetros tipico de una resolución: la matriz, el vector b y el vector de incógnitas x. Este vector debe estar inicializado en el valor NaN (*not a number*) ya que es utilizado en la etapa de *bussy waiting*.

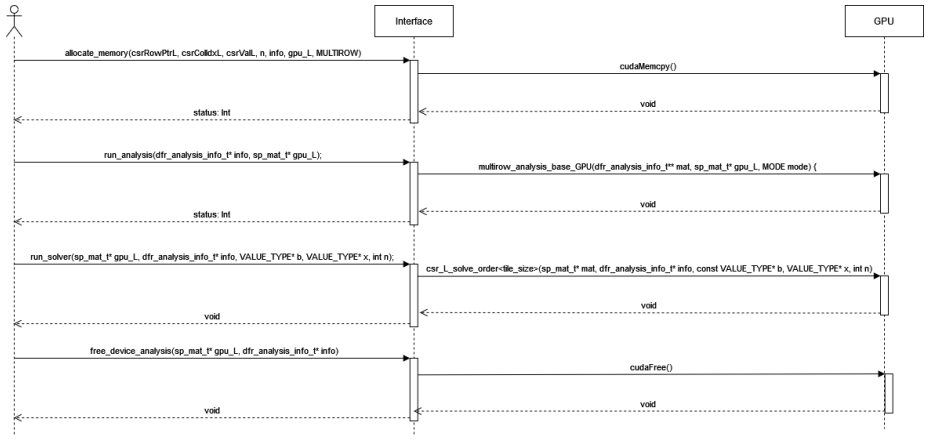

Figura 1: Flujo de uso de las rutinas expuesta por la biblioteca.

Dependiendo del modo seleccionado, esta función puede ser llamada una o varias veces. En el caso del modo SIMPLE, se invoca una única vez dando como resultado una única solución. Para los modos ORDER y MULTIROW, se puede invocar la función run_solver() con distinto vector b la cantidad de veces que se desee sin tener que volver a solicitar un nuevo análisis.

A continuación se muestra un ejemplo de una invocación a las funciones expuesta por la biblioteca en una corrida para el método Multirow. se asume que $csrRowPtrL$, $csrColIdxL$ y $csrValL$ son los datos de la matriz que fueron cargado previamente con el algoritmo mencionado anteriormente o alguna otra implementación. La variable b es el vector solución de la ecuación $Ax = b$ que se utiliza para resolver el sistema.

Algoritmo 8 $solver_{smpl}$

```
1     dfr_analysis_info_t* info;
2     sp_mat_t* gpu_L;
3     VALUE_TYPE* x;
4
5     allocate_memory(csrRowPtrL, csrColIdxL, csrValL, n, info, gpu_L,
          MULTIROW);
6     run_solver(gpu_L, info, b, x, n);
7     free_device_analysis(gpu_L, info, x);
8
9
10    free(b);
11    free(info);
12    free(gpu_L);
```