Utilización eficiente de plataformas basadas en GPUs para acelerar modelos numéricos de gran porte

Franco Seveso Giordano

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

Montevideo – Uruguay
Diciembre de 2025

Utilización eficiente de plataformas basadas en GPUs para acelerar modelos numéricos de gran porte

Franco Seveso Giordano

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

Codirectores de tesis:
Dr. Ing. Pablo Ezzatti
Dr. Ing. Ernesto Dufrechou

Director académico:
Dr. Ing. Pablo Ezzatti

Montevideo – Uruguay
Diciembre de 2025

Seveso Giordano, Franco

Utilización eficiente de plataformas basadas en GPUs para acelerar modelos numéricos de gran porte / Franco Seveso Giordano. - Montevideo: Universidad de la República, Facultad de Ingeniería, 2025.

xxi, 102 p.: il.; 29, 7cm.

Codirectores de tesis:

Pablo Ezzatti

Ernesto Dufrechou

Director académico:

Pablo Ezzatti

Tesis de Maestría – Universidad de la República, Programa de Informática, 2025.

Referencias bibliográficas: p. 85 – 102.

1. Computación de alto rendimiento, 2. Dinámica de fluidos computacional, 3. Stencil, 4. Smoother, 5. DILU Multicolor, 6. AMG.

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE TESIS

Dra. Libertad Tansini

Dr. Deniel Caviedes Voullieme

Dr. Juan Diego Campo

Montevideo – Uruguay
Diciembre de 2025

RESUMEN

Las ecuaciones en derivadas parciales (EDPs) son una herramienta fundamental para modelar la realidad en diversas áreas de la ciencia y la ingeniería. Durante su resolución numérica, es habitual la aparición de sistemas de ecuaciones algebraicas lineales dispersos y de gran escala, cuya solución eficiente condiciona el resto del procedimiento. En este contexto, es común el uso de métodos específicos, como el *Strongly Implicit Procedure* (SIP) para discretizaciones de EDPs elípticas mediante diferencias finitas; o la utilización de métodos iterativos más generales, frecuentemente usados junto a precondicionadores que aceleran su convergencia, como los métodos multigrilla algebraicos (AMG, por su sigla en inglés).

En los últimos años, las GPU se han convertido en el dispositivo de cómputo intensivo por excelencia, siendo fundamentales tanto en el área de la inteligencia artificial como en la computación científica. Sin embargo, aprovechar su poder de cómputo para la resolución de EDPs plantea desafíos, por ejemplo, debido a las dependencias de datos entre los nodos de la malla, resultantes de los esquemas de discretización y de las moléculas de cálculo (*stencils*) definidas.

Esta tesis estudia mecanismos de sincronización eficientes que permitan aprovechar mejor la arquitectura de las GPU en la resolución de EDPs. Por un lado, se adopta un mecanismo de sincronización de hilos conocido como *synchronization-free* al cálculo de *stencils* en GPU. Esta estrategia, permite habilitar el cómputo tan pronto como una dependencia se resuelve, evitando esperas innecesarias y mejorando la utilización de los recursos del dispositivo. La aplicación de esta estrategia a la paralelización del método SIP en problemas representativos de mecánica de fluidos computacional (CFD) muestra mejoras significativas en el desempeño, que alcanzan hasta un 23 % respecto a las implementaciones basadas en el esquema clásico de sincronización por niveles (*level-sets*).

Por otro lado, se estudia la optimización del método AMG. En este contexto se presentan dos implementaciones, basadas en estrategias *synchronization-free*, del smoother DILU Multicolor para la biblioteca AmgX de Nvidia. Ambas versiones superan de manera consistente al smoother DILU Multicolor

original de AmgX, logrando una aceleración promedio de $3,7\times$. Esta optimización permite reducir el tiempo de cada iteración del método AMG hasta en un 80 % para los casos de CFD estudiados.

Palabras claves:
Computación de alto rendimiento, Dinámica de fluidos computacional, Stencil, Smoother, DILU Multicolor, AMG.

ABSTRACT

Partial differential equations (PDEs) are fundamental tools for modeling real-world phenomena across many areas of science and engineering. Their numerical solution typically yields large-scale, sparse linear systems, whose efficient solution determines the overall performance of the computation. In this context, it is common to rely on specialized methods such as the *Strongly Implicit Procedure* (SIP) for finite-difference discretizations of elliptic PDEs, or on more general iterative schemes frequently combined with preconditioners to accelerate convergence, including algebraic multigrid (AMG) methods.

In recent years, GPUs have become the dominant platform for high-performance computation, playing a central role in both artificial intelligence and scientific computing. However, exploiting their computational capabilities for PDE solvers introduces challenges, particularly due to data dependencies between mesh nodes arising from discretization schemes and the associated stencil operations.

This thesis investigates efficient synchronization mechanisms that better leverage GPU architectures in the numerical solution of PDEs. First, it applies a thread-synchronization strategy known as *synchronization-free* to stencil computations on GPUs. This approach enables computation to proceed as soon as a dependency is resolved, avoiding unnecessary stalls and improving device utilization. When applied to the parallelization of the SIP method for representative CFD problems, this strategy yields significant performance improvements, achieving speedups of up to 23 % compared with implementations based on the classical level-set synchronization scheme. Second, the thesis explores optimizations of the AMG method. In this context, it presents two sync-free implementations of the DILU Multicolor smoother for NVIDIA's AmgX library. Both versions consistently outperform the original DILU Multicolor smoother in AmgX, achieving an average acceleration of 3.7×. These optimizations reduce the per-iteration cost of the AMG cycle by up to 80 %

for the CFD cases analyzed.

Keywords:
High-performance computing, Computational Fluid Dynamics, Stencil, Smoother, DILU Multicolor, AMG.

Lista de figuras

2.1 Patrón *stencil* clásico de 7 puntos en malla 3D 10
2.2 AMG clásico, con ciclo V, extraído de [44] 16
2.3 Pseudocodigo de las etapas Setup y Solve de AMG, con ciclo V, extraído de [46] 17
2.4 Estructura utilizada por el formato de almacenamiento para matrices dispersas COO, imagen extraída de [52] 21
2.5 Estructura utilizada por el formato de almacenamiento para matrices dispersas CSR, imagen extraída de [52] 22
2.6 Flujo de ejecución de un *warp* en arquitecturas previas y posteriores a Volta, *independent thread scheduling*, imágenes extraídas de Volta Architecture Whitepaper [60]. 26

4.1 Dependencias del punto id. En verde se presentan dos niveles o planos. El punto id perteneciente al nivel L, mientras que $id_{W/S/B}$ pertenecen al nivel $L-1$ 48
4.2 *Speedups* de las versiones con sincronización basada en elementos (SIP_{ES}) frente a la basada en niveles (SIP_{LS}), variando N_i y N_j , N_k. 52
4.3 Media de cantidad de bloques (# of Blocks) por nivel (Level). . 53
4.4 *Speedups* de las versiones basadas en elementos (SIP_{ES}) frente a la basada en niveles (SIP_{LS}), variando la cantidad de niveles. 54
4.5 Tiempo de ejecución (en milisegundos) y *speedup* para mallas cúbicas de diferentes tamaños (el eje x muestra la raíz cúbica del tamaño del problema). 54
4.6 Campo de velocidad instantánea, normalizada por la velocidad uniforme de entrada, donde se aprecia la naturaleza turbulenta del fluído. 56

4.7 Rendimiento de las rutinas involucradas en el solver SIP para las versiones SIP_{LS} y SIP_{ES} sobre los problemas BFS. 57

5.1 Grilla de Kernel para $SFMILU_{2D}$. 60

5.2 Ejemplo de *warp* con desbalance de carga. 62

5.3 Características del conjunto grande de matrices. 65

5.4 (a) Dimensión y colores de las matrices, (b) *forward* y (c) *backward* *Speedup* de las implementaciones $SFMILU_{2D}$ y $SFMILU_{1D}$ respecto a la línea base, sobre el conjunto amplio de matrices ordenadas segun el criterio establecido. Los *speedups* por encima de $1,0$ refieren a mejoras obtenidas con las implementaciones propuestas. 66

5.5 Speedup de la sustitución hacia adelante de la propuesta $SFMILU_{1D}$ sobre la línea base ejecutado de forma aislada, y dimensiones para cada matriz, ordenadas por cantidad de colores. 67

5.6 *Speedup* Total de las implementaciones $SFMILU_{2D}$ y $SFMILU_{1D}$ sobre la línea base ejecutado de forma aislada, ordenadas por cantidad de colores. 68

5.7 *Speedup* del tiempo total del smoother de la implementación $SFMILU_{1D}$ sobre $SFMILU_{2D}$ ejecutada de forma aislada, ordenadas por cantidad de elementos no cero promedio por fila. . 69

5.8 *speedups* de la sustitución hacia adelante (*forward*) y hacia atrás (*backward*) de $SFMILU_{2D}$ sobre $MILU_{BASE}$ de AmgX, de forma ascendente según el tamaño de grilla de invocación del *kernel*. 74

5.9 *speedups* de la sustitución hacia adelante (*forward*) y hacia atrás (*backward*) de $SFMILU_{1D}$ sobre $MILU_{BASE}$ de AmgX, de forma ascendente según el tamaño de grilla de invocación del *kernel*. 75

5.10 *speedups* de la sustitución hacia adelante (*forward*) y hacia atrás (*backward*) de las versiones $SFMILU_{1D}$ y $SFMILU_{2D}$ sobre $MILU_{BASE}$ de AmgX, ordenado según los distintos niveles de cada matriz, por cantidad de colores del nivel mas fino. 77

5.11 *Speedup* de la sustitución hacia adelante (*forward*) y hacia atrás (*backward*) de $SFMILU_{1D}$ sobre $MILU_{BASE}$ de AmgX, en orden según la cantidad promedio de no ceros por fila. 78

5.12 *Speedup* de las versiones $SFMILU_{2D}$ y $SFMILU_{1D}$ sobre $MILU_{BASE}$ para toda la etapa solve de AmgX, según la cantidad de colores del nivel mas fino. 78
5.13 Tiempo de ejecución de AmgX para cada nivel en las versiones $SFMILU_{2D}$ y $SFMILU_{1D}$ sobre las matrices que poseen mas de un nivel. 79

Lista de tablas

4.1 Tiempo de ejecución de SIP_{LS} y SIP_{ES} en discretizaciones cubicas de diferentes tamaños. 55
4.2 Tiempo de ejecución total (en segundos) de los cuatro kernels para las diferentes discretizaciones del problema BFS. 56

5.1 Características de las matrices correspondientes a problemas de CFD. 72
5.2 Cantidad de iteraciones y tiempos (en milisegundos) de cada iteración dentro de la solve para $MILU_{BASE}$, $SFMILU_{2D}$ y $SFMILU_{1D}$. 77

Tabla de contenidos

Lista de figuras xiii

Lista de tablas xvii

1 Introducción 1

2 Conceptos preliminares 7

2.1 Ecuaciones en derivadas parciales . . . 7
2.2 Caffa3D.MBRi . . . 11
2.2.1 SIP (Strongly Implicit Procedure) . . . 13
2.3 Métodos de Multigrilla Algebraica . . . 15
2.3.1 AMG en GPU, AmgX . . . 18
2.3.2 DILU Multicolor . . . 19
2.4 Matrices dispersas . . . 19
2.5 GPU . . . 23

3 Trabajo relacionado 29

3.1 Stencils . . . 29
3.2 Manejo de dependencias . . . 31
3.2.1 Coloreo de grafos . . . 31
3.3 SIP . . . 34
3.3.1 SIP en CPU . . . 34
3.3.2 SIP en GPU . . . 35
3.3.3 SIP con planificación por niveles - Línea Base . . . 37
3.4 AMG . . . 38
3.4.1 Smoothers en GPU . . . 41
3.4.2 ILU . . . 42
3.4.3 DILU Multicolor AmgX - Línea Base . . . 43

4 Optimización del cómputo del método SIP en GPU **47**
4.1 Propuesta de planificación por elementos 48
4.1.1 Optimizaciones de memoria 49
4.2 Evaluaciones experimentales 50
4.2.1 Plataforma de pruebas 51
4.2.2 Resultados y Análisis 51
4.2.3 Aplicación a un problema de Dinámica de Fluidos 55

5 Optimización de DILU Multicolor, en la biblioteca AmgX de NVIDIA **59**
5.1 SFMILU 2D 60
5.2 SFMILU 1D 62
5.3 Evaluación Experimental 64
5.3.1 Plataforma de pruebas 64
5.3.2 Ambiente aislado 64
5.3.3 Integración con la biblioteca AmgX 69

6 Conclusiones y trabajo futuro **81**
6.1 Conclusiones 81
6.2 Trabajos publicados 83
6.3 Trabajo futuro 83

Referencias bibliográficas **85**

Capítulo 1

Introducción

Las ecuaciones diferenciales en derivadas parciales (EDP) son ampliamente utilizadas para el modelado de fenómenos físicos, en campos desde la dinámica de fluidos hasta electromagnetismo o transferencia de calor. En particular, la simulación de problemas de dinámica de fluidos computacional (CFD, por sus siglas en inglés) dependen de la resolución eficiente de sistemas lineales dispersos de gran tamaño. En este contexto, el rendimiento de los algoritmos empleados para resolver los sistemas lineales influye de manera determinante en la calidad del modelo numérico y en el desempeño computacional de la simulación. Por ello, la eficiencia, la robustez y la capacidad de escalado de los solvers resultan factores claves para el éxito de las simulaciones CFD y, en general, de cualquier aplicación que requiera la solución numérica de EDP.

La resolución numérica de EDP a menudo involucra discretizaciones del dominio, espaciales o temporales, con el fin de transformar un espacio continuo en discreto y así obtener una malla de puntos sobre la cual aplicar métodos iterativos para poder aproximar la solución.

El cómputo de los patrones *stencil* surge en este contexto, donde dentro de la malla obtenida de la discretización, el valor de cada punto debe ser actualizado por una combinación ponderada de sus vecinos.

En cambio, cuando las dependencias de un punto no están contenidas en un entorno cercano del mismo, ni siguen un patrón en especifico, la convergencia del método iterativo clásico, utilizado (como GMRES o Grandiente Conjugado) para la obtención de la solución aproximada, puede verse afectada, por lo que se busca acelerar la convergencia mediante el uso de un precondicionamiento. Entre otros, los métodos multigrilla algebraicas (AMG) son ampliamente

utilizados en estos contextos, por su robustez y escalabilidad [1, 2].

Los métodos iterativos tradicionales (como Jacobi) son muy eficientes a la hora de eliminar los errores de alta frecuencia, pero presentan carencias cuando se trata de errores de baja frecuencia, provocando que converjan lentamente después de unas cuantas iteraciones [3]. Los métodos multigrilla resuelven este problema, construyendo una jerarquía de problemas con diferentes niveles de resolución. Donde los errores suavizados son transferidos desde los niveles más finos hacia niveles más pequeños o gruesos (*coarser*), en donde se convierten en errores de alta frecuencia y pueden ser eliminados más eficazmente. En este contexto, el smoother sirve para reducir los errores de alta frecuencia en cada nivel, preparando el sistema para que la corrección en niveles más gruesos pueda actuar eficazmente sobre los componentes de error restantes.

La elección del smoother es un factor crítico, ya que influye directamente en la eficiencia y la convergencia del método iterativo. Entre los esquemas clásicos, Jacobi y Gauss-Seidel destacan por su bajo costo computacional y su simplicidad de implementación, aunque su capacidad para reducir errores de alta frecuencia es limitada. En contraste, los métodos basados en factorizaciones incompletas como ILU(0), ofrecen mayor robustez numérica al aproximar con mayor precisión la factorización LU de la matriz, pero esto implica un incremento significativo en el uso de memoria y en el número de operaciones. DILU se presenta como una alternativa intermedia entre Jacobi e ILU(0) [1], al combinar las ventajas de ambos enfoques. Esta simplificación reduce de manera notable los requisitos de memoria y disminuye el costo computacional, manteniendo al mismo tiempo una capacidad eficaz para atenuar errores de alta frecuencia.

La creciente disponibilidad de arquitecturas de cómputo masivamente paraleles, como las GPU, ha impulsado la migración de componentes esenciales de los solvers hacia implementaciones optimizadas para estos dispositivos [4–6]. Un ejemplo representativo es la biblioteca AmgX de NVIDIA. Esta herramienta integra una amplia variedad de métodos iterativos, incluidos PCG, GMRES y BiCGStab, junto con precondicionadores multigrilla algebraicas y geométricas (GMG). Dichos precondicionadores permiten incorporar distintos esquemas de smoothing, como Jacobi, Gauss-Seidel, ILU y DILU Multicolor. Un aspecto destacado de AmgX es su diseño modular y su implementación nativa en CUDA para todas las fases del ciclo multigrilla: desde la construcción de niveles *coarse* hasta la aplicación de operadores de restricción y prolongación,

incluyendo los smoothers. Esto posibilita incorporar modificaciones mediante la manipulación directa del código ejecutado en la GPU.

Sin embargo, lograr explotar eficientemente el paralelismo tanto en los métodos multigrillas, como en el cómputo de *stencils*, sobre plataformas masivamente paralelas como la GPU es un gran desafío, ya que presentan restricciones que limitan su aprovechamiento del paralelismo masivo.

En el caso de los patrones *stencils*, se debe a las inherentes dependencias que presentan los puntos entre sí. Un claro ejemplo de ello es *Strongly Implicit Procedure* (SIP) [7], que potencialmente debe resolver varios sistemas de ecuaciones lineales para cada malla de puntos, donde cada uno de estos presenta dependencias entre los puntos que fuerzan a sincronizar, limitando el paralelismo disponible. En este sentido, la planificación por niveles (*level-set scheduling* [8]) es una técnica que permite manejar las dependencias, asegurando que se proceda al cómputo únicamente cuando todos los datos de las dependencias estén resueltos.

Por otro lado, en el caso del precondicionamiento mediante métodos multigrillas, las operaciones de smoothing, que suelen exhibir patrones irregulares de acceso a memoria, sumado también, a las dependencias de datos que dificultan su escalabilidad eficiente en GPUs. Que, en problemas de CFD, esta situación se intensifica debido a las características estructurales de las matrices derivadas de discretizaciones espaciales de ecuaciones como las de Navier-Stokes [9]. Estas matrices suelen poseer grafos de adyacencia complejos y un elevado número de conjuntos de nodos independientes (frecuentemente denominados *colores* debido al uso de algoritmos de coloreo de grafos), lo que genera dependencias que serializan la ejecución. En consecuencia, la optimización de este componente impacta tanto en el tiempo de ejecución como en la escalabilidad del método al trabajar con mallas grandes e irregulares. Estas limitaciones, sumadas a la frecuencia con la que se invoca el smoother dentro de un ciclo AMG, generan un cuello de botella que compromete el rendimiento global del solver.

Ante estos desafíos, diversos trabajos han explorado el uso de técnicas de coloreo de grafos para mejorar el paralelismo y gestionar dependencias de datos en solvers iterativos y métodos multigrilla. Por ejemplo, en [10] se introduce el método Gauss–Seidel multicolor. Posteriormente, [11] aplicó el coloreo de grafos a la resolución de sistemas triangulares dispersos en GPU, demostrando su eficacia para exponer un paralelismo detallado y manejar dependencias en operaciones inherentemente secuenciales. Sobre estos principios, [12] implemen-

tó un Gauss–Seidel Multicolor en bloques dentro de un esquema multigrilla, mostrando que este enfoque permite equilibrar la carga computacional y mejorar la eficiencia. Trabajos más recientes [13, 14] extendieron este concepto a solvers multigrilla acelerados por GPU, proponiendo variantes del método Gauss–Seidel multicolor orientadas a eliminar conflictos entre hilos y preservar la independencia de subdominios, habilitando una ejecución altamente concurrente en múltiples niveles del ciclo multigrilla.

En esta tesis se abordan dos líneas de trabajo complementarias para las que se desarrollan propuestas orientadas a mejorar la eficiencia de métodos iterativos y multigrilla en GPU. En primer lugar se presenta e implementa un nuevo mecanismo de planificación para la ejecución paralela del SIP, sustituyendo así el método basado en niveles *level-set*. Esta propuesta es motivada por la observación de que varios puntos pertenecientes a un nivel pueden ser potencialmente procesados antes de que el nivel anterior sea resuelto por completo. El mecanismo de sincronización propuesto es basado en puntos o elementos (*element-based*), y resuelve agresivamente las dependencias entre los diferentes puntos de la malla en GPU. Con este objetivo, se busca contribuir con el estado del arte en el campo del cómputo de *stencils* acelerados por GPU, y por ende, contribuyendo a simulaciones numéricas más rápidas y eficientes de sistemas físicos complejos.

En segundo lugar se presentan implementaciones con optimizaciones específicas destinadas a acelerar funciones internas de AmgX. En particular, la propuesta incluye versiones optimizadas del smoother DILU Multicolor, con el objetivo de acelerar esta fase del ciclo AMG. Las rutinas desarrolladas emplean técnicas *synchronization-free* [15] para evitar barreras innecesarias que limiten la actividad de la GPU, logrando reducciones apreciables en el tiempo de ejecución. Dichas implementaciones fueron evaluadas experimentalmente en distintos escenarios. Por un lado, se efectuaron pruebas aisladas para comparar de forma directa el desempeño de cada componente interno del smoother respecto de la implementación incluida en la versión actual de AmgX. Por otro lado, las propuestas fueron evaluadas dentro del marco completo del solver AMG, utilizando matrices provenientes de problemas reales de dinámica de fluidos computacional, y analizando el rendimiento del solver multigrilla en cada nivel. Al estudiar el impacto de las optimizaciones en escenarios representativos de CFD, se confirma que se obtienen mejoras significativas en los tiempos de cómputo sin comprometer la estabilidad numérica ni la precisión

de la solución.

Esta tesis se organiza de la siguiente manera. El Capítulo 2 introduce los conceptos fundamentales necesarios para comprender los métodos y desarrollos presentados, de forma que el documento resulte autocontenido. En el Capítulo 3 se revisan los trabajos más relevantes de la literatura, conformando el estado del arte que sirve como base para las propuestas de esta tesis. El Capítulo 4 describe el nuevo mecanismo de planificación integrado en el solver SIP, junto con los experimentos realizados y el análisis correspondiente. En el Capítulo 5 se presentan las propuestas desarrolladas para DILU Multicolor, evaluadas tanto como solver independiente como en su función de smoother dentro del ciclo AMG en AmgX; además, se discuten los resultados y su análisis. Finalmente, el Capítulo 6 expone las conclusiones generales y señala posibles líneas de trabajo futuro.

Capítulo 2

Conceptos preliminares

Este capítulo reúne los conceptos preliminares necesarios para comprender el desarrollo de la tesis. Su objetivo es establecer las bases teóricas y técnicas que sustentan los métodos, algoritmos y modelos utilizados en los capítulos posteriores. Para ello, se introducen nociones fundamentales relacionadas con la formulación y discretización de ecuaciones diferenciales en derivadas parciales, así como principios esenciales de los métodos numéricos empleados para su resolución.

Asimismo, se describen los mecanismos computacionales y arquitecturas que intervienen en la implementación eficiente de los algoritmos estudiados, con énfasis en aquellos aspectos que influyen directamente en el desempeño y la paralelización en GPU. De este modo, el capítulo garantiza que el lector disponga de las herramientas conceptuales necesarias para seguir con claridad las contribuciones presentadas en la tesis, sin asumir conocimientos previos específicos más allá de los habituales en el área.

2.1. Ecuaciones en derivadas parciales

Una ecuación diferencial parcial (EDP) es una ecuación que relaciona una función desconocida de varias variables independientes (por ejemplo, espacio y tiempo) con sus derivadas parciales respecto a una de esas variables [16]. De forma general, una EDP expresa una ley física, geométrica o empírica mediante una relación entre los cambios locales (derivadas) de una magnitud y los valores que dicha magnitud toma en el espacio o el tiempo.

Por este motivo la resolución de EDPs es fundamental en diversas áreas,

como en finanzas económicas [17], biofísica [18] o mecánica de fluidos [19, 20].

Las EDPs se derivan a partir de principios fundamentales de la física, u otras áreas de las ciencias, que describen la conservación o el balance de magnitudes físicas. Su formulación proviene principalmente de tres tipos de leyes:

- Leyes de conservación
- Leyes constitutivas
- Relaciones geométricas o cinemáticas

Por ejemplo, las EDPs que describen el flujo de un fluido se derivan a partir de los principios de conservación (de masa, de la cantidad de movimiento, y de la energía) aplicados a una cantidad infinitesimal de fluido, llamados volúmenes de control diferenciales [21]. Estos principios conforman las ecuaciones de Navier-Stokes y describen el movimiento de fluidos viscosos, aunque en contextos simplificados estas ecuaciones también pueden encontrarse reducidas, en formas más simples, como la ecuación de Poisson para la presión o la ecuación de difusión del calor [22].

Las EDPs pueden clasificarse según la naturaleza del fenómeno físico que modelan (pudiendo combinarse entre ellas):

- Ecuaciones elípticas: describen estados estacionarios o en equilibrio, como la distribución de presión o temperatura en régimen permanente.
- Ecuaciones parabólicas: representan procesos de difusión o disipación, como la transferencia de calor transitoria.
- Ecuaciones hiperbólicas: modelan la propagación de ondas o el transporte de información a lo largo del flujo.

Estas pueden resolverse de forma analítica mediante técnicas clásicas como separación de variables, transformadas integrales (por ejemplo, de Fourier o Laplace), series de potencias y el uso de funciones de Green. Estos métodos permiten obtener soluciones exactas bajo condiciones muy restrictivas, generalmente aplicables únicamente a problemas de una o dos dimensiones con geometrías simples, condiciones de frontera homogéneas y coeficientes constantes. En este sentido, resolver las EDPs analíticamente es especialmente restrictivo y requiere mucha investigación y tiempo [23], y en la práctica, los problemas reales en ciencia e ingeniería suelen presentar dominios irregulares, condiciones de frontera complejas y no linealidades que hacen imposible o impracticable la obtención de soluciones analíticas. Incluso cuando una solución cerrada existe,

puede depender de series infinitas o integrales difíciles de evaluar numéricamente, lo que limita su utilidad computacional.

Esta dificultad fundamental motiva el uso de métodos numéricos para resolver EDPs, que permiten abordar problemas más complejos con dominios irregulares, cargas no uniformes, materiales heterogéneos y condiciones físicas realistas. Estas soluciones no solo permiten el modelado preciso de sistemas físicos, sino también la exploración de escenarios que son inaccesibles mediante técnicas analíticas, consolidando a la simulación numérica como una herramienta indispensable en aplicaciones realistas.

Para la resolución numérica de estas ecuaciones se requiere transformar el problema continuo en uno discreto, mediante un proceso de discretización. Estas técnicas reemplazan el dominio continuo de la variable por un conjunto discreto de puntos (malla) y aproximan las derivadas por expresiones que dependen de los valores de la función en esos puntos. Comúnmente para lograr este objetivo se definen tres etapas principales. Primero se divide el dominio del problema en una malla (estructurada o no estructurada, dependiendo de la geometría del problema) compuesta por elementos discretos. Luego se reemplazan las derivadas (espaciales y temporales) por aproximaciones numéricas basadas en los valores de las variables de puntos vecinos. Diferentes aproximaciones dan lugar a métodos como el de diferencias finitas (FDM) [24], volúmenes finitos (FVM) [25] o elementos finitos (FEM) [26]. [1]

Al momento de reemplazar las derivadas parciales en cada nodo de la malla por aproximaciones numéricas basadas en el valor de sus nodos vecinos, se genera un patrón conocido como *stencil*. Dicho patrón refleja la geometría de los puntos del dominio discreto que intervienen en la aproximación de una derivada o de un operador diferencial. Cada *stencil* define, por tanto, una relación local entre el punto donde se evalúa la ecuación y los valores de la variable dependiente en sus nodos vecinos. En una dimensión, por ejemplo, un *stencil* de tres puntos involucra el nodo central y sus dos vecinos inmediatos; mientras que en dos y tres dimensiones, los *stencils* de cinco y siete puntos constituyen las extensiones más comunes.

Por último, luego de discretizar las ecuaciones en cada punto, elemento o volumen, en los esquemas implícitos se obtiene un sistema de ecuaciones

[1] Existen otros métodos menos difundidos como elementos finitos discontinuos, elementos espectrales, elementos de contorno, entre otras [27–29].

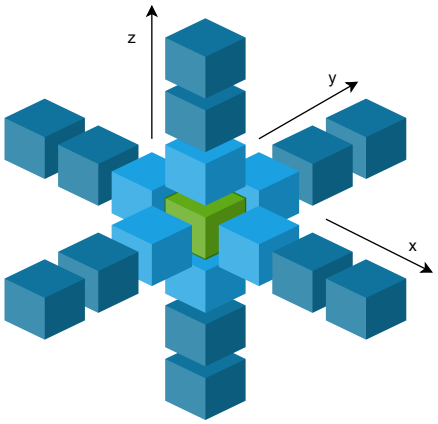


Figura 2.1: Patrón *stencil* clásico de 7 puntos en malla 3D

algebraicas que relaciona los valores discretos de las variables:

$$Ax = b \tag{2.1}$$

donde A es una matriz definida por los patrones del *stencil* utilizado, x el vector de incógnitas (presión, velocidades, temperatura, etc.) y b representa los términos fuente o condiciones de contorno [30].

La resolución de estos sistemas de ecuaciones algebraicos, que por la configuración de la matriz A son sistemas dispersos, es la etapa clave de la mayoría de los simuladores de sistemas dinámicos [31, 32]. Dependiendo del tipo de ecuaciones y del esquema temporal utilizado, pueden emplearse diferentes estrategias. Los métodos directos, como la factorización LU, son adecuados únicamente para sistemas pequeños, debido a su elevado costo computacional y al fenómeno de *fill-in*, que introduce elementos no nulos durante la factorización. Para matrices grandes y dispersas, resultan más eficientes los métodos iterativos, entre ellos, Jacobi, Gauss–Seidel, GMRES y BiCGStab. Adicionalmente, con el fin de acelerar su convergencia, se emplean técnicas multigrilla, ya sean geométricas (GMG) o algebraicas (AMG), que resuelven el problema de forma jerárquica en múltiples niveles.

Entre muchos otros ejemplos, una disciplina que utiliza fuertemente estas técnicas es la Dinámica de Fluidos Computacional (CFD, según sus siglas en inglés). La CFD es una rama de la mecánica de los fluidos que utiliza métodos numéricos y algoritmos computacionales para analizar y resolver problemas

que involucran desde movimientos de fluidos o transferencia de calor hasta electromagnetismo [33].

2.2. Caffa3D.MBRi

Caffa [34] es un método de volúmenes finitos para el cálculo de problemas de flujo utilizando mallas de dos dimensiones estructuradas en bloques. En los sistemas de ecuaciones lineales resultantes, todos los bloques están implícitamente acoplados, por lo que no se tratan condiciones de frontera entre ellos, y el método es totalmente convergente, *"dado que las interfaces entre bloques se tratan de manera implícita y en forma totalmente conservativa, las propiedades de convergencia son casi las mismas que en el caso de una malla estructurada"* [34]. La discretización es tanto en el espacio como en el tiempo, y el algoritmo de solución se basa en el método SIMPLE (*Semi-Implicit Method for Pressure-Linked Equations* [35]).

Caffa3D.MBRi [36] es una extensión que añade, entre otras cosas, la posibilidad de aplicar este método a mallas 3D estructuradas por bloques, el solver AMG y un esquema de interpolación lineal a la herramienta original Caffa. El resultado es un solver de flujo incompresible de propósito general desarrollado en Uruguay. Diseñado para simular problemas del mundo real que requieren tanto flexibilidad geométrica como capacidad de cálculo en grandes escalas (decenas o cientos de millones de celdas).

El problema de flujo se plantea con las ecuaciones de Navier–Stokes para un fluido incompresible (conservación de masa y momento), además de posibles ecuaciones auxiliares (temperatura, turbulencia y transporte). El uso de mallas estructuradas por bloques (*block-structured grids*) permite aislar la geometría del problema mediante técnicas *body-fitted* [37] o incorporar geometrías complejas mediante la estrategia *immersion boundary* [38], representando las fronteras del objeto dentro del dominio como una malla estructurada sin tener que generar una malla completamente irregular y logrando que cada bloque de la malla sea internamente estructurado. Este enfoque tiene importantes ventajas respecto a la alternativa tradicional, de discretizar y modelar toda la geometría con una malla irregular, como la regularidad en los accesos a memoria y *stencils* simples, mientras conserva la flexibilidad geométrica del problema.

Para la discretización el dominio continuo (espacio tridimensional) es divi-

dido en bloques, cada uno formado por una malla estructurada, donde, cada celda tiene un índice ordenado (i, j, k), y caras que conectan con sus vecinas en direcciones según los ejes x, y y z.

El método de volúmenes finitos (FVM) que emplea Caffa3D.MBRi, integra cada EDP sobre un volumen de control (celda), transformando las derivadas espaciales en flujos a través de las caras de la celda, aplicando el teorema de la divergencia [33, 39]. Se calculan los flujos de masa, momento y energía a través de sus caras, e implícitamente se "conectan" con las celdas vecinas mediante esos flujos. Los flujos que cruzan las caras de la celda (por ejemplo, flujo convectivo, difusivo) deben aproximarse en función de los valores de los nodos de la variable (velocidad, presión, temperatura) y su gradiente. Para estimar el valor de las variables en las caras, a partir de los valores en los centros de las celdas vecinas se utiliza interpolación (en grillas no ortogonales la interpolación puede ser corregida o "mejorada" a partir de desarrollos en series de Taylor o distintas correcciones). Se emplean esquemas discretos como CDS (según sus siglas en inglés *Central Difference Scheme*) que promedian los valores de celdas adyacentes o UDS (según sus siglas en inglés *Upwind Differencing Scheme*) que toma el valor desde la celda de donde proviene el flujo. Para los términos advectivos y para el término difusivo, se relaciona el gradiente con diferencias entre nodos vecinos, ponderado según la geometría de la celda y la orientación de las caras.

Una vez evaluados los flujos de cada cara para la celda, se reescribe el balance integral como una ecuación algebraica que involucra el valor de la variable en la celda central y los valores en las celdas vecinas que comparten caras con ella, según la Ecuación 2.2.

$$a_P\phi_P = a_E\phi_E + a_W\phi_W + a_N\phi_N + a_S\phi_S + a_T\phi_T + a_B\phi_B + b_P \tag{2.2}$$

donde P es el punto central a calcular y E,W,N,S,T,B representan las celdas vecinas dependiendo de los puntos cardinales, este ($East$), oeste ($West$), norte ($North$), sur ($South$), arriba (Top) y abajo ($Bottom$), respectivamente y b_P incluye términos fuente (como fuerzas, gradientes de presión o condiciones de frontera). Los coeficientes a_* dependen de la geometría, propiedades del fluido y los esquemas discretos usados.

Al repetir este proceso para cada celda del dominio, se obtiene un sistema

lineal disperso de la forma:

$$A\phi = b \tag{2.3}$$

donde A es una matriz dispersa con estructura *"penta- o hepta-diagonal"* según sea un *5-point stencil* en grilla 2-D o *7-point stencil* en grilla 3-D, cuyos elementos provienen de los elementos $a_P, a_E, a_W ...$, ϕ es el vector de incógnitas (presión, velocidad, temperatura, etc.), y b contiene los términos fuente y condiciones de frontera.

Caffa3D.MBRi resuelve estos sistemas con diferentes métodos lineales en muchas de sus rutinas, pero cuando el sistema es *"de banda"* con dependencia local limitada, el solver preferido es *Strongly Implicit Procedure* (SIP) [8]. Este método actúa como un solver iterativo interno, precondicionado mediante una factorización incompleta, aunque el encargado de iterar externamente es el método SIMPLE asegurando la convergencia global del sistema acoplado.

2.2.1. SIP (Strongly Implicit Procedure)

El solver SIP [7] es un método iterativo para resolver sistemas de ecuaciones lineales de banda, dispersos y de gran escala, como los resultantes de las discretizaciones de EDPs elípticas en mallas regulares. Es una variante del método de relajación incompleta, también conocido como ILU (según sus siglas en inglés, *Incomplete LU factorization*), aplicada dentro de un esquema iterativo. Este procedimiento produce una precondición implícita que reduce el acoplamiento entre las ecuaciones y mejora la estabilidad y la velocidad de convergencia del método.

Un pseudocodigo del mismo se presenta en el Algoritmo 1. La idea central es construir una factorización incompleta de la matriz, buscando aproximar la matriz A mediante el producto de matrices triangulares inferiores y superiores, de modo que:

$$A \approx (L + D)(D^{-1})(U + D) \tag{2.4}$$

donde L representa la parte estrictamente inferior de A, U la estrictamente superior y D es la diagonal principal.

A diferencia de una factorización LU completa, en la factorización incompleta se descartan ciertos términos fuera de la estructura de conectividad original de la matriz, lo que reduce significativamente el costo computacional y el almacenamiento. Esta aproximación preserva la estructura de los coeficien-

tes del sistema original, lo cual es fundamental para las matrices derivadas de mallas estructuradas tridimensionales (como las que utiliza caffa3D.MBRi).

Una vez obtenida la factorización incompleta, cada iteración del método SIP realiza los siguientes pasos:

- Cálculo del residuo (donde $\phi^{(0)}$ es un valor inicial dado):

$$r^{(k)} = b - A\phi^{(k)} \tag{2.5}$$

- Resolución del sistema precondicionado: Se resuelve aproximadamente el sistema

$$(L + D)D^{-1}(U + D)\Delta\phi = r^{(k)} \tag{2.6}$$

  mediante dos pasos fundamentales:

  - Sustitución hacia adelante (*Forward Substitution*), encargado de resolver el sistema triangular inferior $(L + D)x = r^{(k)}$
  - Sustitución hacia atrás (*Backward Substitution*), encargado de resolver el sistema triangular superior $(U + D)\Delta\phi = Dx$

  Si bien estas dos sustituciones no son iterativas en sí mismas, constituyen el paso interno de cada iteración global del SIP.
- Actualización de la solución:

$$\phi^{(k+1)} = \phi^{(k)} + \alpha\Delta\phi \tag{2.7}$$

  donde α es un parámetro de relajación (usualmente cercano a 1) que controla la estabilidad y la velocidad de convergencia del método.
- Criterio de convergencia: El proceso iterativo se repite hasta que la norma del residuo relativo

$$\frac{\|r^{(k)}\|}{\|b\|} \tag{2.8}$$

  alcanza una tolerancia establecida (por ejemplo, 10^6).

Este método combina la simplicidad estructural de los métodos "de punto fijo", con la eficiencia y robustez de los métodos basados en precondicionamiento ILU, por lo que resulta una herramienta muy adecuada para resolver ecuaciones elípticas que surgen por ejemplo en el tratamiento de la presión en los algoritmos de acoplamiento presión–velocidad (como SIMPLE).

Algoritmo 1 Strongly Implicit Procedure (SIP)

Input: A, b, x_0
Output: x

1: Do incomplete LU decomposition $\hat{L}\hat{U} \approx A$
2: Calculate initial residual: $r_0 = b - Ax_0$
3: **while** residual is not small enough **do**
4: Calculate vector R_n (Forward substitution): $R_n = \hat{L}^{-1} r_n$
5: Calculate δx (Backward substitution): $\hat{U}\delta x = R_n$
6: Update solution: $x_{n+1} = x_n + \delta x$
7: Update residual: $r_{n+1} = b - Ax_{n+1}$
8: **end while**

2.3. Métodos de Multigrilla Algebraica

Los métodos multigrilla (MG) surgen de la observación de que los métodos iterativos clásicos, como Jacobi o Gauss–Seidel, son eficientes para eliminar los errores de alta frecuencia (variaciones locales), pero ineficaces para reducir los errores de baja frecuencia (variaciones suaves o globales) y son ampliamente utilizados para resolver sistemas lineales de gran escala, teniendo asociadas aplicaciones en diversos campos, entre los cuales están incluidos los problemas de dinámica de fluidos computacional [40–42].

La idea del enfoque multigrilla es combinar procesos de suavizado (*smoothing*) con la transferencia de información entre mallas de diferente resolución, de modo que los errores de baja frecuencia en una malla fina se transformen en errores de alta frecuencia en una malla más gruesa, donde pueden eliminarse de manera eficiente.

A diferencia del enfoque multigrilla geométrico, que requiere una jerarquía explícita de mallas, los métodos de multigrilla algebraica (AMG) construyen dicha jerarquía, únicamente a partir de la matriz $\mathbf{A}$ de un sistema $Ax = b$, sin utilizar información geométrica del dominio físico. En términos generales, AMG opera sobre una secuencia de matrices $\mathbf{A_0}, \mathbf{A_1}, ..., \mathbf{A_L}$ donde $\mathbf{A_0} = \mathbf{A}$ representa el sistema original y los niveles siguientes corresponden a aproximaciones más gruesas (*coarse*) del problema.

Típicamente el método AMG mediante el enfoque de Galerkin [43] es presentado como:

$$\mathbf{A}_{l+1} = \mathbf{R}_l \mathbf{A}_l \mathbf{P}_l \tag{2.9}$$

donde $\mathbf{R}_l$ es el operador de restricción que transfiere información del nivel fino

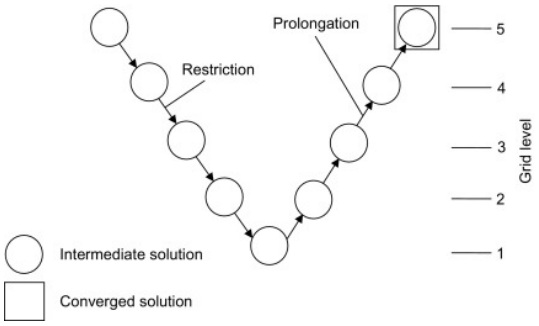


Figura 2.2: AMG clásico, con ciclo V, extraído de [44] .

al grueso (*coarse*) y $\mathbf{P}_l$ es el operador de interpolación desde el nivel grueso (*coarse*), cumpliéndose

$$\mathbf{R}_l = \mathbf{P}_l^{\mathbf{T}} \tag{2.10}$$

para cada nivel l. Ademas, en los diferentes niveles se realizan las siguientes operaciones clave:

- Restricción del residuo:

$$\mathbf{b}_{l+1} = \mathbf{R}_l(\mathbf{b}_l - \mathbf{A}_l\mathbf{x}_l) \tag{2.11}$$

- Resolución aproximada del sistema en el nivel grueso (*coarse*):

$$\mathbf{A}_{l+1}\mathbf{x}_{l+1} = \mathbf{b}_{l+1} \tag{2.12}$$

- Prolongación de la corrección:

$$\mathbf{x}_l \leftarrow \mathbf{x}_l + \mathbf{P}_l\mathbf{x}_{l+1} \tag{2.13}$$

El ciclo en un algoritmo multigrilla algebraica es lo que determina el orden y la frecuencia con que el método recorre los distintos niveles de la jerarquía para reducir errores de diferente frecuencia. Los tipos de ciclos más utilizados son V, W o F, la forma clásica de AMG, como se muestra en la Figura 2.2 es con ciclo V, el cual es el más simple y estándar por defecto, el algoritmo desciende una vez por cada nivel hasta el más grueso y luego asciende una vez por cada nivel.

La calidad, y eficiencia, del método dependen fuertemente del esquema de "coarsening" y del tipo de smoother utilizado en cada nivel. El rol del smoother es reducir los componentes de error de alta frecuencia antes y después de cada

Input: A_0: fine-grid operator
`max_size`: threshold for max size of coarsest problem
`nongalerkin`: (optional) non-Galerkin method
$\gamma_1, \gamma_2, \ldots$ (optional) drop tolerances for each level

Output: $A_1, \ldots, A_L$,
$P_0, \ldots, P_{L-1}$

while `size`$(A_\ell) >$ `max_size`
  $S_\ell =$ `strength`(A_ℓ) {Strength-of-connection of edges}
  $P_\ell =$ `interpolation`(A_ℓ, S_ℓ) {Construct interpolation and injection}
  $A_{\ell+1} = P_\ell^T A_\ell P_\ell$ {Galerkin product}

  if `nongalerkin` {(optional) described in section 2.1}
    $\hat{A}_{\ell+1} =$ `sparsify`$(A_{\ell+1}, A_\ell, P_\ell, S_\ell, \gamma_\ell)$ {Remove nonzeros in $A_{\ell+1}$}
    $A_{\ell+1} = \hat{A}_{\ell+1}$

Input: x_0: fine-level initial guess
b_0: fine-level right-hand side
$A_0, \ldots, A_L$
$P_0, \ldots, P_{L-1}$
Output: x_0, fine-level approximation

for $\ell = 0, \ldots, L-1$ **do**
  `relax`$(A_\ell, x_\ell, b_\ell, \nu_1)$ {Presmooth ν_1 times}
  $b_{\ell+1} = P_\ell^T (b_\ell - A_\ell x_\ell)$ {Restrict residual}
$x_L =$ `solve`(A_L, b_L) {Coarsest-level direct solve}
for $\ell = L-1, \ldots, 0$ **do**
  $x_\ell = x_\ell + P_\ell x_{\ell+1}$ {Interpolate and correct}
  `relax`$(A_\ell, x_\ell, b_\ell, \nu_2)$ {Postsmooth ν_2 times}

Figura 2.3: Pseudocodigo de las etapas Setup y Solve de AMG, con ciclo V, extraído de [46] .

paso *coarse*. Esta actividad típicamente se realiza mediante la solución de un sistema de ecuaciones usando Jacobi, Gauss–Seidel, ILU o variantes coloreadas como DILU Multicolor [45].

En términos de implementación, el algoritmo AMG se organiza en dos fases principales: Setup y Solve, como se puede ver en los algoritmos de la Figura 2.3.

La fase de Setup está orientada a la construcción de la estructura interna necesaria para el posterior proceso del solver. Esto implica la selección de los nodos *coarse* en cada nivel, la definición de los patrones de dependencia entre variables y la generación de las estructuras de datos que representarán a los operadores de restricción, prolongación y las matrices *coarse*.

En esta etapa también se interpreta y aplica la configuración especificada —por ejemplo, a través de un archivo de parámetros—, estableciendo aspectos como el esquema de *coarsening*, el tipo y número de aplicaciones de smoother, la forma del ciclo multigrilla (V, W o F), solver a utilizar en el nivel *coarse*, criterios de parada y límites de tamaño de niveles. El Setup puede implicar un costo computacional considerable, pero se realiza solo una vez para cada sistema y se reutiliza mientras la matriz no cambie de forma significativa.

La fase de Solve, en cambio, utiliza la jerarquía, estructura y operadores generados durante el Setup para realizar las iteraciones del solver multigrilla. Aquí se aplica el patrón de *smoothing* y corrección previamente definido, siguiendo el tipo de ciclo determinado en la fase de Setup, repitiendo el proceso hasta alcanzar la tolerancia deseada o un número máximo de iteraciones. Dado que esta fase puede ejecutar repetidamente cada uno de los pasos, la eficiencia de estos es clave para el desempeño global del solver.

2.3.1. AMG en GPU, AmgX

La biblioteca AmgX, desarrollada por Nvidia e implementada en CUDA, ofrece una infraestructura de propósito general para la resolución eficiente de sistemas lineales dispersos en GPU, basada en métodos iterativos y multigrillas algebraicas. Su arquitectura modular permite configurar los solvers y precondicionadores mediante archivos en formato JSON, lo que facilita su adaptación a distintos tipos de problemas sin modificar el código fuente. La posibilidad de ajustar parámetros como la tolerancia de convergencia, el número máximo de iteraciones, el tipo de ciclo multigrilla o la estrategia de *coarsening* permite adaptar el comportamiento del solver a las características físicas y numéricas del problema. Además, debido a su código abierto y sus ejemplos de prueba, brinda extensibilidad para personalizar componentes internos del solver.

AmgX ofrece una amplia gama de solvers iterativos, como *Conjugate Gradient* (CG), BiCGStab, GMRES y FGMRES, que pueden emplearse de forma independiente o combinados con precondicionadores multigrilla. Estos precondicionadores pueden ser de tipo *classical* o *aggregation-based* y permiten configurar jerarquías multigrillas con diferentes tipos de ciclos (V, W o F), así como seleccionar distintos métodos de suavizado en cada nivel. Entre los smoothers disponibles se incluyen Jacobi, Gauss–Seidel, Chebyshev, y variantes basadas en factorizaciones incompletas como ILU y DILU.

2.3.2. DILU Multicolor

El smoother DILU Multicolor combina la factorización incompleta diagonal (*Diagonal Incomplete LU*, DILU) con un esquema de coloreo del grafo de la matriz con el objetivo de explotar el paralelismo. El algoritmo clásico de factorización DILU [47] realiza una simplificación eficiente de la factorización incompleta LU sin *fill-in* (ILU(0)), en la cual las actualizaciones se limitan únicamente a los elementos pertenecientes a la diagonal de la matriz. Mas precisamente, el método DILU descompone la matriz como $A = L_A + D_A + U_A$ y construye el precondicionador $M = (L_A + D)D^{-1}(D + U_A)$ donde D contiene los pivotes generados durante la factorización LU incompleta. Mientras que L_A y U_A son matrices triangulares y representan la parte triangular inferior y superior de las entradas de A. Como únicamente D necesita calcularse y almacenarse explícitamente para aplicar M durante las iteraciones del solver, se reduce significativamente el costo de almacenamiento y cómputo en comparación con ILU(0). En cambio, el método DILU resulta menos robusto que ILU(0), aunque en algunos casos, por ejemplo, en matrices tridiagonales, ambos métodos son equivalentes.

El coloreo de la matriz es una técnica que permite identificar conjuntos independientes de filas que pueden procesarse en paralelo. De forma general, a partir del grafo de adyacencia de una matriz donde cada nodo representa una fila (o variable), se asigna un color distinto a cada grupo de nodos que no comparten aristas. Dos nodos comparten una arista si sus correspondientes filas tienen elementos no nulos en las mismas columnas, lo que implica dependencia de datos. Por lo tanto, el coloreo de una matriz asigna a cada fila de la misma un color, de modo que aquellas que comparten el mismo color no presenten dependencias entre sí [45]. Esto permite resolver algunas ecuaciones en paralelo durante las sustituciones hacia adelante y hacia atrás propias de la factorización incompleta, reduciendo la naturaleza secuencial de este procedimiento y habilitando un mayor aprovechamiento del paralelismo masivo en GPU.

2.4. Matrices dispersas

Una matriz se denomina dispersa cuando la cantidad de elementos nulos que contiene es significativamente mayor que la de elementos distintos de cero, lo

cual contrasta con las matrices densas. Por esta razón, las matrices dispersas ocupan un lugar esencial debido a su presencia en una amplia variedad de problemas científicos.

Las matrices dispersas se emplean extensamente en el análisis de grafos, donde representan de forma eficiente estructuras como matrices de adyacencia y laplacianos. Esta representación permite resolver problemas de particionado, clasificación y análisis espectral en redes de gran escala [48]. También son esenciales en optimización numérica de gran dimensión, particularmente en programación lineal, cuadrática y métodos de punto interior. La estructura dispersa de jacobianos y hessianos reduce el costo computacional y posibilita algoritmos escalables [49, 50]. Además, la esparsidad es clave en modelos de recomendación y análisis de datos, donde las matrices usuario–ítem son inherentemente dispersas. Esto permite aplicar factorización matricial y métodos iterativos eficientes [51].

En métodos numéricos para la solución de ecuaciones en derivadas parciales, cuando se aplica sistemáticamente a través de todos los nodos (o volúmenes de control) de una malla, se genera naturalmente un sistema de ecuaciones algebraicas cuya matriz de coeficientes presenta una estructura inherentemente dispersa. Esta dispersión de los elementos surge directamente de la naturaleza local de las aproximaciones numéricas: cada ecuación discretizada involucra únicamente un pequeño número de puntos vecinos (determinado por el *stencil*), mientras que permanece desconectada del resto de los puntos.

La manipulación eficiente de matrices dispersas requiere estrategias específicas diferentes a las empleadas en matrices densas, para evitar almacenar y operar con sus elementos nulos. Tratar una matriz dispersa como si fuera densa conduce a un consumo de memoria excesivo y a la ejecución de operaciones innecesarias sobre los ceros implícitos, lo cual resulta ineficiente e incluso prohibitivo en problemas de gran escala. En consecuencia, se han desarrollado diferentes estrategias de almacenamiento que permiten representar únicamente los valores no nulos junto con su localización en la matriz.

Una forma natural y sencilla de representar matrices dispersas es mediante el formato de coordenadas (COO). En esta estrategia, la matriz se almacena utilizando tres arreglos como muestra la Figura 2.4: uno que contiene los valores no nulos, y otros dos que almacenan los índices de fila y columna correspondientes a cada valor almacenado. De esta manera, cada elemento distinto de cero se describe completamente mediante una terna (fila, columna, valor),

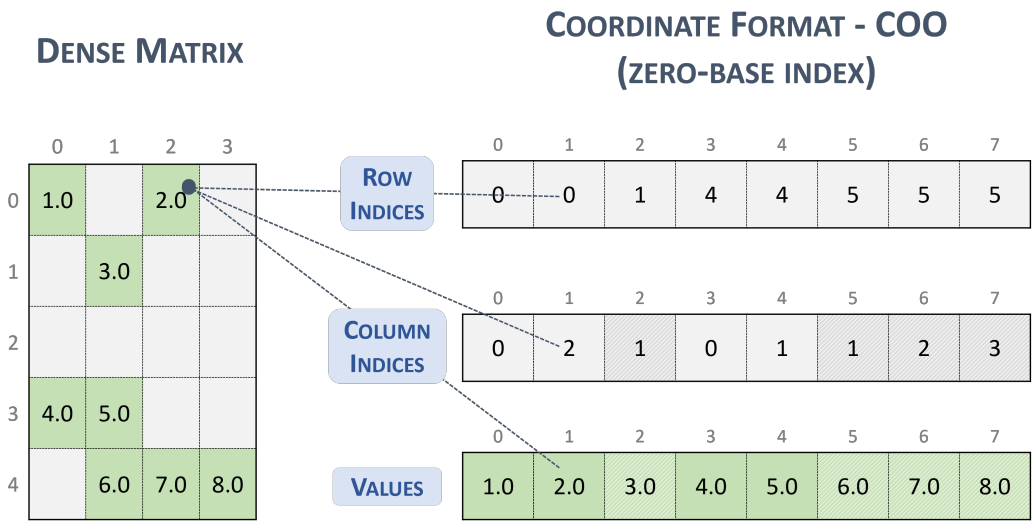


Figura 2.4: Estructura utilizada por el formato de almacenamiento para matrices dispersas COO, imagen extraída de [52]

lo que permite una representación directa y fácilmente interpretable. Esta estructura resulta especialmente útil cuando se requiere modificar la matriz, ya que permite insertar o eliminar elementos de forma sencilla, siendo eficiente tanto en uso de memoria como en tiempo de inserción [53]. Desde el punto de vista del almacenamiento, si bien ofrece una reducción sustancial del consumo de memoria respecto a las matrices densas, su estructura no compacta implica redundancia de memoria y por lo tanto un mayor costo computacional [54]. Esta ineficiencia se manifiesta particularmente en operaciones que requieren un acceso ordenado a los elementos, como la multiplicación matriz-vector, donde el formato COO resulta significativamente menos eficiente debido a la falta de un agrupamiento eficiente por filas o columnas [53], limitando su rendimiento en comparación con otros formatos especializados para operaciones específicas [54]. En [55–57] se abordan diferentes estrategias para mejorar dichas ineficiencias del formato sobre la operación matriz-vector en arquitecturas paralelas.

Por otro lado, uno de los formatos más utilizados para almacenar matrices dispersas es el comprimido por filas (CSR, según sus siglas en inglés *Compressed Sparse Row* [58]). A diferencia del formato COO, CSR organiza los elementos de manera ordenada por filas y más eficiente mediante tres arreglos, como se muestra en la Figura 2.5: el arreglo *values* contiene los valores no nulos ordenados por filas, el arreglo *col_index* almacena los índices de columna

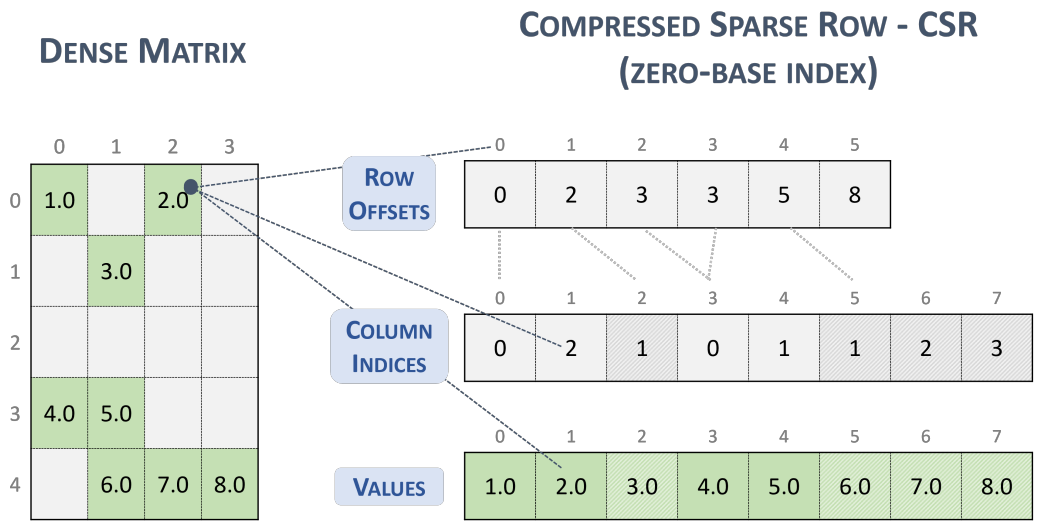


Figura 2.5: Estructura utilizada por el formato de almacenamiento para matrices dispersas CSR, imagen extraída de [52]

correspondientes, y el arreglo *row_ptr* indica la posición de inicio de cada fila en los arreglos anteriores.

Esta estructura elimina la redundancia presente en COO, donde cada elemento no nulo requiere almacenar explícitamente su índice de fila. En CSR, el arreglo *row_ptr* tiene longitud igual al número de filas más uno, en lugar de depender de la cantidad total de elementos no nulos. Como resultado, CSR reduce significativamente el consumo de memoria y proporciona un acceso secuencial optimizado por filas, lo que lo hace particularmente eficiente para operaciones como la multiplicación matriz-vector

Sin embargo, el formato CSR presenta limitaciones, por ejemplo, cuando se requiere recorrer la matriz por columnas (en cuyo caso podría utilizarse el formato CSC, *Compressed Sparse Column*), como también en contextos donde a menudo se quiere modificar la matriz, ya que, al contrario de COO, insertar elementos implica un gran esfuerzo al reorganizar los tres vectores.

Estas diferencias reflejan que no existe un formato único óptimo, sino que la elección depende tanto del contexto donde se desee utilizar como del patrón de dispersión que presente la matriz.

En cuanto a la manipulación computacional, los algoritmos para matrices dispersas también deben adaptarse a estas características. Mientras que en matrices densas las operaciones tradicionales (como el producto matriz-matriz, matriz-vector o la factorización LU) se implementan sobre estructuras

compactas con acceso secuencial a memoria, en el caso disperso las operaciones requieren un manejo cuidadoso de índices, estructuras dinámicas y técnicas de compresión que reduzcan accesos innecesarios a memoria. Por lo que la gama de técnicas o estrategias a seguir para sus implementaciones es muy amplia y permite optimizaciones importantes dependiendo del contexto y la aplicación.

Las matrices dispersas representan un desafío computacional significativo, debido a su estructura irregular y acceso no contiguo a memoria dificultan su procesamiento eficiente en arquitecturas tradicionales, a pesar de contar con un alto grado de paralelismo inherente. Esta situación introduce la necesidad de explorar plataformas de hardware que puedan aprovechar mejor este paralelismo y superar las limitaciones actuales.

2.5. GPU

Las unidades de procesamiento gráfico (GPU, por sus siglas en inglés Graphics Processing Units) constituyen una de las arquitecturas de hardware más relevantes en el ámbito del cómputo de alto desempeño. Aunque las GPU surgieron con el propósito de acelerar la renderización y el procesamiento gráfico, su campo de aplicación se expandió considerablemente durante las últimas dos décadas. Este cambio de paradigma se consolidó gracias al desarrollo de CUDA (por sus siglas en inglés *Compute Unified Device Architecture*), una plataforma y API de programación paralela propuesta por NVIDIA. CUDA permite aprovechar la capacidad de las GPU para realizar cómputo de propósito general mediante una extensión del lenguaje C++, conocida como CUDA C++, facilitando así la implementación a bajo nivel de algoritmos sobre hardware específico.

Además, la plataforma incluye un conjunto de bibliotecas aceleradas en GPU (como cuBLAS, cuSPARSE, cuSOLVER, Thrust, entre otras) y herramientas de *profiling* y depuración (como *nvprof*, *Nsight Compute* y *Nsight Systems*) que simplifican el desarrollo, la optimización y la validación del rendimiento de las aplicaciones. Gracias a esta infraestructura, CUDA se ha convertido en una herramienta esencial en el desarrollo de aplicaciones masivas en GPU, siendo ampliamente utilizada en diversas áreas científicas.

La arquitectura CUDA se caracteriza por una organización jerárquica del hardware, optimizada para el procesamiento masivo de datos mediante la ejecución simultánea de hilos. En términos estructurales, una GPU se compone

de múltiples *Streaming Multiprocessors* (SMs), que constituyen las unidades básicas de cómputo. Cada SM contiene un conjunto de núcleos CUDA (*CUDA cores*), los cuales son las unidades aritmético-lógicas (ALUs) responsables de ejecutar las operaciones elementales. Además, cada SM integra otros componentes especializados, como unidades de control, registros, una memoria de tipo *scratchpad* llamada *"memoria compartida"* (*Shared Memory*), y una memoria cache (*Cache L1*).

La jerarquía de memoria de una GPU se completa por una cache L2 accesible a todos los SMs y una memoria global fuera del chip, abundante pero de alta latencia. Los registros, son relativamente escasos, y de acceso exclusivo para cada hilo. Constituyen la memoria más rápida del dispositivo, utilizada para almacenar variables locales y resultados intermedios.

La memoria compartida representa un espacio intermedio, accesible por todos los hilos de un mismo bloque, permitiéndoles cooperar y reutilizar datos, siendo especialmente útil en aplicaciones con alta localidad espacial[2] o temporal[3]. En arquitecturas recientes, este espacio ocupa la misma memoria física con la caché L1, que almacena datos de uso reciente y reduce los accesos a niveles más lentos.

La memoria caché L2 tiene la función principal de reducir el tráfico hacia la memoria global, que constituye el espacio principal de almacenamiento del dispositivo. Esta última ofrece gran capacidad y ancho de banda, pero también alta latencia, por lo que los accesos deben realizarse de forma ordenada (*coalesced*) para evitar penalizaciones en el rendimiento.

Además, las GPU incluyen memorias especializadas que optimizan ciertos patrones de acceso. La memoria constante y la memoria caché de solo lectura son empleadas para almacenar datos inmutables o parámetros globales, ofreciendo un acceso eficiente cuando los hilos leen direcciones idénticas. Por su

[2]El bloqueo espacial (o *Spatial blocking*) es una técnica que busca mejorar la localidad espacial de los datos. Consiste en dividir el dominio computacional en bloques (o *tiles*) que se procesan de forma independiente, reutilizando los datos de los puntos vecinos almacenados temporalmente en memoria compartida o caché. De esta manera, se minimizan los accesos redundantes a la memoria global.

[3]El bloqueo temporal (o *Temploral blocking*) se utiliza para explotar la localidad temporal de los datos, es decir, la reutilización de valores en diferentes pasos de tiempo del cálculo. En lugar de calcular un único paso temporal para toda la grilla, el dominio se divide en regiones (espaciales o *tiles*) donde se realizan varios pasos de tiempo consecutivos antes de avanzar al siguiente bloque. Esto reduce la frecuencia de lecturas y escrituras a memoria global, pero introduce mayores requerimientos de sincronización y gestión de dependencias, ya que las actualizaciones deben respetar el orden causal entre los puntos.

parte, la *texture memory* dispone de una caché orientada a patrones espaciales de acceso, resultando útil en operaciones sobre dominios multidimensionales.

En conjunto, esta jerarquía permite equilibrar el costo de acceso a memoria con la necesidad de paralelismo masivo. La correcta utilización de cada nivel resulta sumamente importante para lograr un buen rendimiento, especialmente en aplicaciones con alta intensidad de cómputo y patrones de acceso regulares.

CUDA proporciona una interfaz de programación para gestionar la ejecución de código paralelo en la GPU. Su modelo de ejecución se basa en la noción de *kernels*, definidos por el usuario que se ejecutan en paralelo por un gran número de hilos (*threads*) en el dispositivo, siguiendo en alto nivel un enfoque SPMD (según sus siglas en inglés, *Single Program Multiple Data*) [59], por lo que el comportamiento entre hilos varia únicamente dependiendo de sus identificadores para calcular o acceder a los datos. Cuando un *kernel* es invocado desde el *host*, la GPU ejecuta una grilla (*grid*) de hilos organizada jerárquicamente en dos niveles. En el nivel superior, la grilla se compone de un conjunto de bloques de hilos, mientras que cada bloque, a su vez, agrupa una cantidad fija de hilos. Tanto el número de bloques como la cantidad de hilos por bloque se especifican en el momento de la invocación del *kernel*. Cada bloque puede incluir hasta un máximo de 1024 hilos, los cuales se ejecutan en grupos de 32 hilos denominados *warps*.

El modelo de ejecución a bajo nivel, conocido como SIMT (según sus siglas en inglés *Single Instruction Multiple Threads*), es característico de todas las arquitecturas CUDA de NVIDIA. En él, los hilos de un *warp* ejecutan de manera simultánea la misma instrucción, aunque sobre diferentes datos. Si bien el tamaño del *warp* (32 hilos) se ha mantenido constante desde las primeras arquitecturas CUDA (como Tesla y Fermi) hasta las más recientes (Ampere, Ada Lovelace y Hopper), la implementación del modelo SIMT ha evolucionado. En arquitecturas antiguas, como Kepler o Maxwell, los hilos dentro de un *warp* estaban fuertemente sincronizados a nivel de instrucción, lo que implicaba una ejecución estrictamente bloqueante ante divergencias de flujo (*branch divergence*). A partir de arquitecturas Volta en adelante, NVIDIA introdujo el concepto de *Independent Thread Scheduling* [60], que permite a los hilos de un mismo *warp* divergir y reconverger de forma más flexible, reduciendo la penalización asociada a la ejecución de caminos condicionales y mejorando la eficiencia del hardware.

Este cambio es apreciable también desde el punto de vista del hardware,

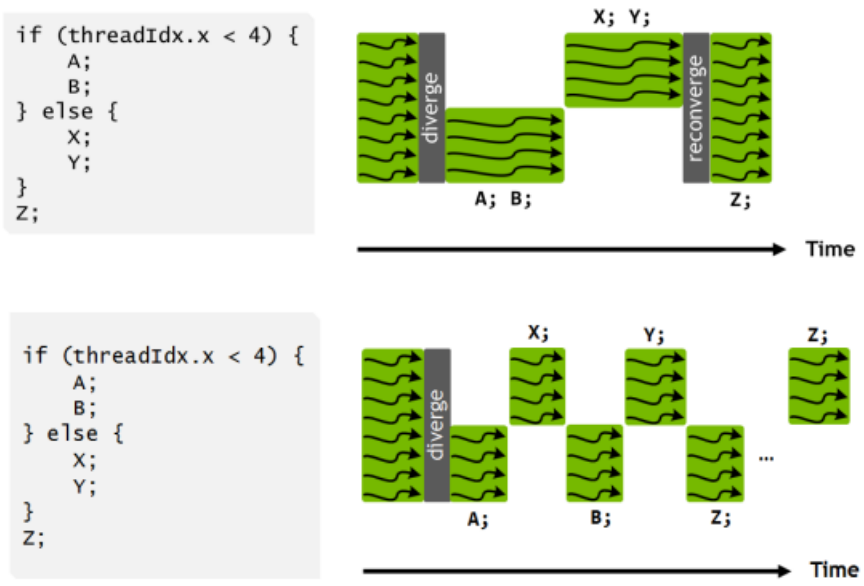


Figura 2.6: Flujo de ejecución de un *warp* en arquitecturas previas y posteriores a Volta, *independent thread scheduling*, imágenes extraídas de Volta Architecture Whitepaper [60].

debido a que en arquitecturas previas a Volta cada GPU contaba con un PC (*Program Counter*) por *warp*; en cambio, en la arquitectura Volta y posteriores, cada CUDA Core cuenta con un PC y *Stack* de ejecución por lo que, aunque el modelo SIMT y la estructura de *warps* se mantienen conceptualmente invariables en todas las generaciones de GPUs con soporte CUDA, las arquitecturas recientes ofrecen una gestión más avanzada y eficiente de la ejecución paralela a nivel de hilo. Adicionalmente, si bien cada hilo puede tener un estado de ejecución diferente, debido a que tiene su propio PC, el planificador a nivel de *warp* (*warp scheduler*) administra los recursos de forma que en cada ciclo de reloj se ejecute una única instrucción por *warp*. Como muestra la Figura 2.6, potencialmente se puede mantener hilos inactivos, de forma de ejecutar unicamente la introducción en algunos hilos, e intercambiar asignaciones de recursos entre instrucciones aunque no hayan terminado su flujo, en diferentes ciclos de reloj, lo que puede ayudar a evitar deadlocks ante la presencia de bifurcaciones en el flujo de ejecución a nivel de *warp*.

Estos cambios provocan que los hilos que conforman un mismo *warp* ya no se encuentran necesariamente sincronizados en todo momento, sino que pueden avanzar de forma independiente en el flujo de ejecución. Por este motivo, cuando en una sección del código se requiere que todos los hilos de un *warp*

alcancen un punto común y ejecuten de manera coordinada una misma instrucción, es necesario realizar una sincronización explícita, *__syncwarp()*, la cual garantiza la sincronización de los hilos pertenecientes a un *warp*. Por defecto, esta función sincroniza los 32 hilos del *warp*, aunque también puede recibir una máscara de bits que especifica un subconjunto del mismo que deben esperar hasta alcanzar dicho punto de sincronización. No obstante, CUDA dispone de otras primitivas que permiten sincronizar hilos con distintos alcances jerárquicos. Mecanismos de sincronización a niveles superiores, como *__syncthreads()*, que coordina todos los hilos de un bloque, o *__cudaDeviceSynchronize()*, que fuerza la sincronización a nivel global de la GPU, ya existían en arquitecturas previas a Volta. Sumado a que, a partir de CUDA 9 se incorporaron los *Cooperative Groups*, que ofrecen una mayor flexibilidad para la organización y sincronización de hilos.

Capítulo 3

Trabajo relacionado

En este capítulo se aborda una revisión del estado del arte en técnicas y enfoques orientados a la optimización de algoritmos en entornos de cómputo paralelo, con especial foco a los desarrollos aplicados a la resolución de sistemas lineales dispersos y al tratamiento de patrones de cómputo de tipo *stencil* en GPU. Su propósito es contextualizar los avances más relevantes en el área, identificar tendencias comunes y ofrecer un marco de referencia para el trabajo que se desarrolla en los capítulos posteriores.

3.1. Stencils

Este apartado presenta una revisión de algunos de los avances más recientes en el cómputo de *stencils* sobre GPUs, con énfasis en aquellos trabajos que proponen nuevos mecanismos de sincronización entre hilos o que optimizan la planificación de la carga de trabajo.

Uno de los primeros estudios exhaustivos que aborda el cálculo de patrones *stencil* en GPUs modernas es el presentado por Maruyama et al. [61], donde un *7-point stencil kernel*, originalmente desarrollado para la arquitectura Fermi, fue adaptado a la arquitectura Kepler [62]. La principal diferencia señalada por los autores entre las arquitecturas Fermi y Kepler de GPU radica en el acceso a la memoria global. A diferencia de la generación Fermi, las GPUs Kepler almacenan en caché los accesos a memoria global únicamente en el nivel L2 de manera predeterminada, reservando la caché L1 exclusivamente para accesos de solo lectura (aunque en algunos modelos Kepler el desarrollador puede habilitar manualmente un comportamiento de almacenamiento en caché similar al de

Fermi).

Según los autores, esta característica afecta negativamente el rendimiento de los códigos *stencil* que dependen fuertemente de la caché L1 para explotar la localidad de los datos, haciendo necesario el uso de memoria compartida. En este contexto, se analizan técnicas de bloqueo espacial y temporal mediante solapamiento de *tiles* (bloque o subregión de la grilla que se procesa de manera conjunta para evitar repetir instrucciones), utilizando memoria compartida y *warp specialization* para evitar divergencias entre hilos. La sincronización se realiza a nivel de bloque y es identificada por los autores como uno de los principales cuellos de botella de rendimiento, especialmente en la variante de bloqueo temporal.

En [63], Rawat et al. presentan un *framework* denominado ARTEMIS para la generación de códigos *stencil*, el cual incorpora diversas optimizaciones provenientes de trabajos previos, tales como *tiling* espacial y temporal, *streaming*, desenrollado de bucles (*loop-unrolling*) y *prefetching*. Al igual que en investigaciones anteriores, un paso temporal en un dominio tridimensional se procesa de manera que los datos se comparten en dos dimensiones y se transmiten en la tercera, la cual se procesa de forma secuencial. En cuanto a la sincronización, también se lleva a cabo a nivel de bloque de hilos. Antes de ARTEMIS, Rawat et al. habían trabajado en otros *stencil frameworks* basados en lenguajes DSL [64, 65]. Otro ejemplo es AN5D, presentado por Matsumura et al. [66]. Por otro lado, en [67] se introduce un esqueleto de programación paralela para cómputos iterativos de *stencil* en entornos distribuidos con múltiples GPUs. Esta propuesta está implementada en C99 y soporta cualquier tipo de *stencil* geométrico, y se caracteriza por su especificación abstracta del patrón *stencil*.

Otros trabajos recientes, como el de Sai et al. [68], muestran los beneficios de utilizar el algoritmo *semi-stencil* para *stencils* de alto orden, y aplicar optimizaciones específicas de hardware, tales como copias asincrónicas de memoria compartida, con el fin de incrementar el rendimiento en la GPU A100.

Por su parte, Fine Licht et al. [69] presentan un *stencil* de código abierto con representación mediante DAGs como un caso general, particularmente motivado por la predicción numérica del clima y del tiempo atmosférico, así como por la aplicación del *Consortium for Small-scale Modeling* (COSMO). Un aspecto a destacar de este trabajo es su ejecución sobre FPGAs.

La mayoría de los trabajos revisados se centran en la reutilización de datos y en optimizaciones de bajo nivel para mejorar el rendimiento en GPU.

Asimismo, abordan *stencils* iterativos, en los cuales el alto rendimiento se obtiene a partir de la fusión de múltiples pasos temporales (*temporal blocking*). En 2020, Zhang et al. evaluaron distintos métodos y niveles de sincronización incorporados en CUDA mediante microbenchmarking [70]. El trabajo abarca mecanismos de sincronización tales como *cooperative groups*, sincronización a nivel de *warp*, sincronización a nivel de bloque (*__syncthreads*), sincronización a nivel de grilla y sincronización entre multiples grillas. Los autores también analizan el tiempo requerido (ejecución y sobrecarga) de un *kernel* en función de su método de invocación. Adicionalmente, se considera la aplicación de *persistent kernel grids* (*kernels* en los que todos los bloques de hilos lanzados permanecen activos durante toda la ejecución) y la sincronización a nivel de grilla para coordinar diferentes pasos temporales de *kernels* iterativos con patrón *stencil* , donde habitualmente esto se realiza mediante la invocación de múltiples *kernels*. Posteriormente, los autores integran estas nuevas características de sincronización en el *framework* EBISU [71] (*Epoch (temporal) Blocking for Iterative Stencils, with Ultracompact Parallelism*).

3.2. Manejo de dependencias

En el álgebra lineal numérica con matrices dispersas, el manejo de dependencias es de suma importancia para el aprovechamiento eficiente de las plataformas altamente paralelas. En esta sección se presentan algunos de los métodos más utilizados para el manejo de dependencias durante el cómputo de *kernels* de álgebra lineal en GPU.

Los primeros trabajos en este contexto se centraron en la estrategia de *level-set*, o planificación por niveles, presentada en el Capítulo 1.

Por ejemplo, en [72–75] se utiliza para la resolución paralela de sistemas triangulares en GPU. En este caso se realiza un preprocesamiento de la matriz del sistema para determinar a qué nivel corresponde cada fila. Luego, se ejecuta un *kernel* de CUDA por cada nivel, en el que las ecuaciones de cada nivel se resuelven en paralelo por varios hilos.

3.2.1. Coloreo de grafos

Otra técnica empleada para identificar conjuntos de operaciones independientes dentro de un grafo de dependencias es el coloreo de grafos. En el contex-

to del álgebra lineal numérica, cada fila de la matriz dispersa puede representarse como un nodo de un grafo, y las dependencias entre filas —por ejemplo, cuando el cálculo de una fila depende de los valores previamente computados en otra— se representan mediante aristas. El objetivo del coloreo consiste en asignar un "color" a cada nodo de modo que dos nodos adyacentes no compartan el mismo color, lo que garantiza que las operaciones asociadas a nodos del mismo color puedan ejecutarse en paralelo sin conflictos de datos.

En [45] se compara la planificación por niveles (*level-scheduling*) y la planificación basada en colores para la factorización incompleta LU y el solver triangular en GPU. Mostrando de forma experimental que el algoritmo basado en el coloreo del grafo obtiene un mejor rendimiento debido a lograr un mayor paralelismo. Aunque utilizar coloreo de grafos puede afectar la convergencia de métodos iterativos, los autores opinan que el paralelismo logrado con esta estrategia puede compensar el tiempo insumido en las iteraciones extras.

Dado que encontrar la mínima cantidad de colores para colorear un grafo ha sido probado que pertenece a la clase de problemas NP-hard [76], el diseño de algoritmos que proponen coloreos es un área sumamente interesante, ya que, al no poder obtener el resultado óptimo con certeza a un costo computacional acotado, hay un gran compromiso entre qué tan bueno o refinado es el coloreo y cuánto es el esfuerzo que implica obtenerlo.

Dado que el paralelismo depende de la cantidad de colores utilizados para colorear el grafo de dependencias, y a su vez, los distintos algoritmos de coloreo pueden implicar distintas cantidades de colores, el paralelismo máximo a alcanzar depende del algoritmo en cuestión. Las estrategias abordadas por los algoritmos de coloreo pueden ser muchas. Entre ellas, la más simple es la estrategia *greedy*, que consta, en su forma secuencial clásica, de recorrer cada nodo y asignar el color más pequeño disponible, es decir, el que no esté ya asignado a uno de sus vecinos. Esta estrategia es muy sencilla de implementar pero puede producir un muy buen coloreo, logrando incluso un coloreo muy cercano al óptimo [77]. Versiones paralelas de la misma son presentadas en [78], basándose en ejecuciones especulativas que cuentan con dos etapas principales, en una se asigna un posible color a cada nodo (*tentative coloring*) y en la otra se corrigen los defectos (o conflictos entre nodos) que presente el coloreo (*conflict detection*). Posteriormente en [79, 80], se reduce el número de conflictos y se mejora la forma de resolverlos, por ende realizando menos iteraciones y disminuyendo la sincronización entre hilos por cada iteración, lo que provoca

que el algoritmo obtenga una mayor velocidad y escalabilidad.

Otra linea a seguir en algoritmos de coloreo es mediante la construcción de conjuntos maximales de nodos independientes (*maximal independent set*, MIS), debido a que no puede ser extendido y no hay dependencias entre ningún par del mismo, todos los pertenecientes a un MIS pueden tener el mismo color. Una estrategia sencilla y paralela para hallar un MIS se presentó en [81], basada en la trasformación del algoritmo de Monte Carlo para el problema de MIS que buscar alcanzar un algoritmo cuasi-determinista simple y práctico con el mismo tiempo de ejecución en paralelo, demostrando la pertenencia del mismo a NC^2 [82]. Como esta última, diferentes heurísticas pueden ser utilizadas para el coloreo de grafos [83–85], en particular, [86] presenta una versión asincróica para arquitecturas paralelas con memoria distribuida. Posteriormente [87] propone dos algoritmos de alto rendimiento para computadoras con multi-procesadores, donde uno apunta a una mayor eficiencia y el otro hacia el equilibrio entre los distintos colores, es decir, busca balancear la cantidad de nodos que pertenecen a cada color con el objetivo de, potencialmente, lograr un paralelismo más equitativo.

Si bien hace ya varios lustros que fueron presentadas la mayoría de las heurísticas propuestas, el campo de trabajo continúa abierto y al día de hoy se siguen proponiendo algoritmos que mejoren su velocidad y rendimiento. En [88] se presenta GPA, un algoritmo paralelo para coloreo de grafos de gran tamaño, basado en Graphx [89] un *framework* de procesamiento de grafos integrado, construido sobre Apache Spark, un sistema de flujo de datos distribuido ampliamente utilizado.

Implementaciones paralelas para algoritmos de coloreo en GPU son presentadas en [90], utilizando CUDA e invocando *kernels* por conjunto de puntos independientes para lograr la sincronización de forma global, con el objetivo de realizar la cantidad mínima de sincronizaciones posibles. En otra línea, [91] realiza comparaciones de diferentes algoritmos de coloreo utilizando SYCL [92], herramienta que permite abstraer la implementación de la arquitectura utilizada para ejecutar, probando tanto en plataformas NVIDIA como AMD. La utilidad de la propuesta radica en la portabilidad y no en el rendimiento obtenido, concluyendo que la forma más eficiente de obtener rendimiento es realizando un manejo fino de los datos e implementando cada instrucción adaptada al hardware específico.

La gran mayoría de heurísticas presentadas por la comunidad son cons-

truidas empíricamente sin pruebas teóricas explicitas sin embargo en [93] se aportan bases teóricas sólidas para algoritmos de colores, definiendo los tres aspectos principales de un algoritmo de coloreo de grafos paralelos: *Work*, que refiere a la cantidad de trabajo realizada por el algoritmo para lograr el coloreo, apuntando a asegurar que sea casi lineal en el tamaño del grafo, es decir, cercano al óptimo secuencial; *Depth*, que refiere a cuán paralela es la solución, dependiendo de qué tan largo sea el camino crítico del grafo; y *Quality*, que refiere la cantidad de colores utilizados y la consistencia del coloreo.

Estas técnicas de coloreo pueden ser utilizadas en diferentes aplicaciones, por ejemplo, [94] emplea en su pre-procesamiento un algoritmo paralelo para obtener un coloreo de la matriz y posteriormente usarlo en la implementación de un solver que emplea métodos de volúmenes finitos no estructurados, Multi-Colored Gauss-Seidel (MCGS), para simulaciones de flujo compresible en estado estacionario, ejecutado en un entorno con varias GPU.

3.3. SIP

El método SIP fue originalmente concebido para ejecutarse en CPU, presentando una estructura inherentemente dependiente de datos que limita su explotación del paralelismo. Sin embargo, la evolución del hardware computacional ha impulsado la exploración de diversas estrategias de implementación en diferentes plataformas. Las arquitecturas *multi-core* con memoria compartida, aprovechando paradigmas como OpenMP, ofrecen una primera aproximación al paralelismo mediante el procesamiento por diagonales. Por otro lado, las arquitecturas *many-core*, particularmente las GPUs, proporcionan una alternativa prometedora mediante la explotación masiva del paralelismo a nivel de datos. Esta diversidad de arquitecturas computacionales motiva el análisis comparativo de las implementaciones del SIP, evaluando su eficiencia en términos de escalabilidad, orden de procesamiento de los nodos, localidad de datos y rendimiento computacional.

3.3.1. SIP en CPU

En [95] se presentan diversas implementaciones del método SIP tanto para arquitecturas de CPU como de GPU. Siguiendo las ideas de Deserno et al. [96] se desarrollaron dos variantes para CPU.

La primera implementación, denominada *Sequential SIP* (sSIP-CPU), hace uso de un solo núcleo y procesa los coeficientes secuencialmente según el número de nodo, aprovechando eficientemente la localidad de datos.

La segunda implementación, *Parallel SIP* (pSIP-CPU), hace uso de varios núcleos y emplea el paradigma de memoria compartida mediante OpenMP para procesar los nodos de la malla por sus diagonales, este enfoque busca explotar el paralelismo inherente del algoritmo.

Contrariamente a lo esperado, la versión paralela no superó el rendimiento de la implementación secuencial en ninguna de las configuraciones evaluadas. Este comportamiento se atribuye fundamentalmente a que el nivel de paralelismo obtenido al procesar nodos a lo largo de las diagonales no compensa la pérdida de localidad de datos que introduce dicha estrategia. Los resultados experimentales demuestran que, incluso al utilizar múltiples hilos en plataformas multi-core, la versión secuencial mantiene una ventaja significativa en términos de eficiencia computacional. Esta limitación motivó la exploración de implementaciones alternativas en arquitecturas GPU para mejorar el rendimiento del método SIP.

3.3.2. SIP en GPU

En [95], se presentan dos variantes del SIP que ofrecen cómputo paralelo en GPU, desarrolladas con el fin de levantar las restricciones de cómputo paralelo en arquitecturas tipo CPU.

La primera versión, denominada SIP-GPU, sigue un esquema de transferencia de datos convencional entre CPU y GPU y las operaciones se ejecutan sobre memoria global. En este enfoque, múltiples *kernels* se lanzan de forma independiente para procesar cada diagonal en paralelo, generando implícitamente una cola de *kernels* que procesan secuencialmente en un *stream* de la GPU. Esta implementación fue optimizada utilizando transferencias de memoria *page-locked* para reducir los tiempos de comunicación entre dispositivos.

La segunda versión, SIP-GPUsm, introduce el uso intensivo de memoria compartida, con el objetivo de disminuir la latencia de acceso, que según los autores, conforman un factor crítico en etapas como la factorización incompleta LU y las sustituciones hacia adelante y hacia atrás.

Los resultados experimentales demuestran que ambas implementaciones en GPU superan significativamente el rendimiento de las versiones para CPU, al-

canzando aceleraciones de hasta cuatro veces, para matrices de gran dimensión. Sin embargo, la implementación con memoria compartida no mostró mejoras sustanciales respecto a la versión estándar en GPU, posiblemente debido al bajo número de operaciones computacionales realizadas por cada coeficiente transferido a memoria compartida. El análisis detallado revela que las etapas de refinamiento iterativo representan aproximadamente el $75-80\,\%$ del tiempo total de ejecución, y aunque los tiempos de transferencia entre dispositivos son significativos, su impacto se mantiene acotado (máximo $12\,\%$ del tiempo total) gracias al uso de memoria *page-locked*. Estos resultados confirman que las arquitecturas GPU constituyen una alternativa eficiente y de bajo costo para la resolución de sistemas lineales pentadiagonales de gran escala mediante el método SIP.

Para una grilla en tres dimensiones de tamaño $N_i \times N_j \times N_k$, donde se utiliza la misma notación que anteriormente para sus nodos vecinos (E,W,S,N,T,B), sea l la posición de un punto en un arreglo lineal donde los puntos (i, j, k), se ordenan primero por la coordenada i, luego por la j y finalmente por k, es decir, $l = (k-1)N_iN_j + (j-1)N_i + i$.

Los coeficientes de la matriz L asociados a un punto l pueden calcularse mediante las siguientes reglas, donde α es un parámetro del algoritmo:

$$\begin{aligned} \hat{L}_B^l &= A_B^l/(1+\alpha(\hat{U}_N^{l-N_iN_j} + \hat{U}_E^{l-N_iN_j})) \\ \hat{L}_W^l &= A_W^l/(1+\alpha(\hat{U}_N^{l-1}\hat{U}_T^{l-1})) \\ \hat{L}_S^l &= A_S^l/(1+\alpha(\hat{U}_E^{l-N_i}\hat{U}_T^{l-N_i})) \end{aligned} \tag{3.1}$$

Una vez calculados los coeficientes de L, deben actualizarse de forma coherente los coeficientes correspondientes de U.

Posteriormente, se lleva a cabo la etapa de sustitución hacia adelante mediante la expresión:

$$R^l = (r^l - \hat{L}_S^l R^{l-1} - \hat{L}_W^l R^{l-N_j})/\hat{L}_P^l. \tag{3.2}$$

De manera análoga, la sustitución hacia atrás se calcula mediante:

$$\delta x^l = R^l - \hat{U}_N^l \delta x^{l+1} - \hat{U}_E^l \delta x^{l+N_j}. \tag{3.3}$$

A partir de las reglas anteriores utilizadas para calcular los coeficientes de L y U, se observa que el procesamiento del punto l requiere haber calculado

previamente los puntos $l-1$, $l-N_i$, y $l-N_iN_j$. Dependencias de datos similares aparecen en el cálculo de R^l y δx^l.

Al traducir el índice l a las coordenadas de la malla, se obtiene que el punto (i, j, k) depende del cálculo previo de los puntos $(i, j-1, k)$, $(i-1, j, k)$ y $(i, j, k-1)$. Estas mismas dependencias se presentan tanto en las etapas de sustitución hacia adelante como en las de sustitución hacia atrás.

Esto implica que varios puntos de la malla pueden ser procesados de forma independiente en un instante dado. En particular, los puntos que cumplen con igual $i + j + k$, y que forman un plano en el espacio tridimensional, son independientes entre sí. De manera más general, otras mallas, tanto regulares como irregulares, pueden presentar patrones de dependencia similares.

Las dependencias entre los puntos pueden representarse mediante un grafo acíclico dirigido (DAG, segun sus siglas en inglés *Directed Acyclic Graph*), en el cual cada nodo del grafo corresponde a un punto de la malla y existe una arista dirigida desde el nodo i hacia el nodo j si i debe ser calculado antes que j.

Los nodos del DAG pueden organizarse en conjuntos de niveles (*level-sets*), donde los nodos dentro de un mismo nivel son independientes entre sí y dependen, al menos, de un nodo del nivel anterior. En consecuencia, el procesamiento de una malla de puntos puede realizarse nivel por nivel, ejecutando en paralelo todos los elementos pertenecientes a cada nivel. Dada la geometría regular del problema, existe un paralelismo directo entre niveles e hiperplanos, todos los puntos que pertenecen a un mismo nivel pertenecen a un mismo hiperplano.

3.3.3. SIP con planificación por niveles - Línea Base

En [8] se utilizó la herramienta de *profiling* proporcionada por NVIDIA (*nvprof*) para identificar las partes del SIP solver en $Caffa3D.MBRi$ que insumían más tiempo e implementar propuestas para mejorarlas. El análisis reveló que los aspectos más costosos principalmente eran dos. Por un lado, las cuatro subrutinas principales del solver (*ComputeResInnerCellsSIP_GPUk, SolveLUforFiBackwardSIP_GPUk, SolveLUforFiForwardSIP_GPUk y ComputeLuCoefficientsSIP_GPUk*) que realizan un proceso basado por hiperplano. Y por otro lado, el *overhead* de invocar una gran cantidad de *kernels*.

En términos generales, el tiempo de lanzamiento de un *kernel* es despreciable, pero en este caso puede llegar a ser comparable con el tiempo de procesa-

miento de un hiperplano. Sin embargo, dado que la versión original del solver se apoya en un *kernel* para procesar cada hiperplano en la GPU, lo cual resuelve la necesaria sincronización entre el procesamiento de hiperplanos consecutivos, se genera un considerable costo acumulado originado por la invocación de un gran número de *kernels*.

La estrategia de [8], tomada como línea base en este trabajo consiste en consolidar todos los *kernels* lanzados para cada hiperplano en un único *kernel*, reduciendo así el *overhead* asociado al lanzamiento. Este *kernel* de mayor escala procesa secuencialmente los hiperplanos, de manera ordenada, gestionando la sincronización entre ellos y manteniendo la consistencia de los datos durante la ejecución mediante operaciones atómicas.

El *kernel* se lanza con tantos bloques como sean necesarios para cubrir el hiperplano de mayor tamaño, donde cada punto de la malla es procesado por un hilo. Cada bloque procesa una parte de un hiperplano y luego avanza al siguiente. Antes de pasar al siguiente hiperplano, los bloques que terminan de procesar sus puntos o que quedan fuera de los límites del hiperplano y no tienen puntos que procesar deben esperar hasta que el resto de los bloques finalicen el procesamiento del hiperplano actual. Una vez que todos los bloques terminen el hiperplano en curso, avanzan al siguiente. Este proceso continúa hasta que se procesan todos los niveles necesarios de la malla.

A esta implementación del SIP en GPU se le llamará SIP_{LS} de ahora en adelante.

3.4. AMG

Los algoritmos multigrilla constituyeron uno de los primeros campos en los que se exploró sistemáticamente el uso de la GPU para acelerar la resolución de sistemas lineales de gran escala [97, 98]. Debido a que sus operaciones fundamentales presentan una estructura altamente regular y masivamente paralelizable, el método multigrilla se adaptó tempranamente a arquitecturas *many-core*. En el contexto de la física computacional, donde la necesidad de resolver ecuaciones elípticas y sistemas provenientes de discretizaciones de EDPs es crítica, las GPUs ofrecieron beneficios inmediatos, permitiendo explotar un paralelismo fino difícil de alcanzar en arquitecturas tradicionales. Esta combinación posicionó a los métodos multigrilla como pioneros en la adopción de aceleradores gráficos para computación científica de alto rendimiento.

En este contexto, diversas bibliotecas especializadas han sido desarrolladas con el propósito de implementar esta familia de métodos de forma eficiente y paralela, proporcionando una infraestructura numérica robusta que sirva como base para aplicaciones científicas y de ingeniería de gran escala. Un ejemplo paradigmático es AmgX [99], biblioteca propuesta por NVIDIA que proporciona implementaciones en C++ y CUDA orientadas al alto rendimiento de distintos solvers y precondicionadores, aunque también cuenta con extensiones en Python [100] o interfaz en PETSc [101]. En particular, ofrece alternativas para las distintas etapas de AMG, aprovechando el ancho de banda y la capacidad de paralelismo brindado por las GPU. Cuenta con una estructura modular y configurable según el usuario, con un gran nivel de detalle, que permite tener una amplia gama de posibilidades a la hora de abordar un problema. Además, es de código abierto, lo que proporciona transparencia e información extra de las implementaciones utilizadas de los distintos solvers y smoothers, permitiendo la posibilidad de optimizar operaciones de la biblioteca (como SpGEMM y SpMV [102]) así como también poder tener un detalle más fino y un control más específico a la hora de implementar distintas estrategias. En el ámbito de las simulaciones de CFD es ampliamente utilizada, por ejemplo, para acelerar herramientas de simulación como OpenFOAM [103–106].

Hypre [107] es una biblioteca que, análogamente a AmgX, presenta diferentes algoritmos de solvers lineales y precondicionadores. Fue diseñada originalmente para CPU, pero con el paso del tiempo se han integrado versiones paralelas utilizando GPU. Incluye un solver AMG bajo el nombre de *BoomerAMG* [108], implementado en C++ y CUDA, que al igual que AmgX ofrece una versión paralela y puede ser modificado y/o configurado según el contexto de aplicación. Además, varios resultados de trabajos relacionados son incorporados y probados en esta biblioteca. En [109] se lleva a cabo un estudio de diferentes smoothers paralelos para BoomerAMG, contemplando tanto Gauss-Seidel y Jacobi clásicos, como distintas modificaciones de los mismos, analizando la efectividad de cada smoother dependiendo del contexto de ejecución de AMG y de los recursos de hardware disponibles. Recientemente, con las tarjetas gráficas de última generación y la introducción de hardware específico como los *Tensor Cores*, se ha implementado e incorporado a Hypre, AmgT [110], brindando versiones de las distintas etapas del AMG, haciendo uso de éstos junto al aprovechamiento de distintas precisiones para lograr mayor eficiencia y paralelismo.

Otra propuesta de AMG paralelo es *AMG4PSBLAS*, presentado por [111] e implementada sobre *PSBLAS* [112]. También existen trabajos relacionados en los cuales se comparan diferentes smoothers, como por ejemplo, en [113] que se proponen smoothers polinomiales basados en polinomios de Chevyshev, implementados de forma paralela en CUDA, logrando convertirlos en una alternativa posible frente a los smoothers tradicionales debido a su buen rendimiento y escalabilidad en GPU.

Por otro lado, existen interfaces o bibliotecas, como AMGCL, PETSc, CUSP, Trilinos ML que varían según su completitud y lo que permiten al usuario configurar del solver a utilizar; sin embargo, todas ellas incluyen la implementación de AMG como precondicionador y permiten hacer uso de las diferentes implementaciones como una "caja negra". Por ejemplo, AMGCL [114] es una biblioteca en C++ *header-only* que está diseñada para facilitar la integración de AMG en proyectos existentes y soporta implementaciones en OpenMP, CUDA u OpenCL, permitiendo que el usuario defina datos, implementaciones y parámetros de forma flexible. Su filosofía es explícitamente "plug-and-play" y "extensible", lo que la hace muy adecuada para integración en entornos donde se desea adaptar AMG a arquitecturas modernas. PETSc [115], por su parte, proporciona un marco genérico de precondicionadores en el que AMG se inserta como clase de objeto y donde el usuario puede elegir variantes del mismo, por ejemplo, para establecer el tipo de agregación en su AMG. Particularmente PETSc se sitúa en un nivel de abstracción aún más general, ya que permite seleccionar de que biblioteca utilizar las implementaciones, es decir, mediante *-pc_type hypre* o *-pc_type amgx* se puede escoger entre Hypre o AmgX, lo que la convierte en una interfaz muy cómoda y práctica para distintas aplicaciones, por ejemplo, para simulaciones de elementos finitos [116]. De igual modo, Trilinos ML ofrece una implementación de *smoothed-aggregation* AMG y otros esquemas, que se exponen mediante una interfaz configurable dentro del marco general de Trilinos [117]. Por último, CUSP [118], constituye una de las primeras bibliotecas de propósito general para cómputo en GPU basada en CUDA, y ofrece implementaciones de precondicionadores (como *Smoothed Aggregation AMG*), así como una interfaz de alto nivel para solvers iterativos y operaciones sobre matrices dispersas. Su diseño se centra en la portabilidad y simplicidad de uso, permitiendo experimentar con técnicas multigrilla directamente sobre GPU. Aunque su desarrollo activo se ha reducido, CUSP ha sido pionera en trasladar el paradigma AMG a GPUs y sirvió como base

conceptual para bibliotecas posteriores como AmgX.

Aunque todas estas bibliotecas actuarán como un "solver" o precondicionador eficaz en muchos casos, la diferencia reside en cuán estructurada es la interfaz de configuración (qué parámetros de coarsening, interpolación, smoother, ciclo multigrilla se exponen al usuario), para qué tipo de hardware se ofrece (por ejemplo GPU, CPU multinúcleo, distribuidos) y qué tan modular o extensible es la biblioteca (por ejemplo permitir la adición nuevas estrategias de coarsening, exportar jerarquías, reutilizar datos).

3.4.1. Smoothers en GPU

En el marco de un algoritmo de multigrilla geométrico acelerado en GPU [13], se propone, entre otros aportes, un esquema *multi-color simetric lower-upper Gauss-Seidel* (por sus siglas en inglés, multi-color LU-SGS) como smoother, diseñado para abordar de manera eficiente los problemas de dependencias inherentes a la ejecución paralela en GPU. Otra variante de Gauss-Seidel como smoother para GMG, llamada X-shaped Multi-Color Gauss-Seidel (X-MCGS) es presentada en [14]. Esta variante particiona la grilla en 4 colores para 2D o en 8 para 3D, con ordenamiento en "X" que elimina ramificaciones condicionales (*branching*) y permite accesos de memoria regulares, mejorando el paralelismo de hilos en GPU.

Por otro lado, en AMG también existen variantes de Gauss-Seidel como smoother que utilizan coloreo, Block Multi-Color Gauss-Seidel [119] aunque en CPU, propone su propio coloreo apuntando a lograr el mayor balance de carga y así poder asegurar la eficiencia del método.

Todos los smoothers analizados se evalúan en distintos tipos de problemas, pero ninguno resulta óptimo en todos los escenarios. Por ello, la elección del smoother adecuado suele depender tanto del problema específico como de las características de los datos de entrada. En [120] se presenta un *framework* basado en *Convolutional Neural Networks (CNNs)*, entrenado a partir de problemas de EDPs de pequeña escala y guiado por teorías de convergencia multigrilla, con el fin de garantizar propiedades específicas de suavizado. El enfoque emplea una función de pérdida orientada a minimizar el error de la aproximación y demuestra que el modelo resultante puede generalizarse a problemas de gran escala pertenecientes a la misma clase de EDPs, con el objetivo de suplantar o complementar el rol de los smoothers en los métodos multigrilla.

3.4.2. ILU

Entre los smoothers utilizados para algoritmos multigrillas, se encuentra la factorización incompleta LU (ILU) y sus distintas variantes.

En [121] se propone para los contextos donde la solución puede ser aproximada, una estrategia iterativa para resolver los sistemas triangulares generados por la ILU (*forward and backward substitution*) cuando se aplica como precondicionador. Con el objetivo de aumentar la utilización de la GPU y explotar el paralelismo, reemplaza estrategias exactas para resolver las dependencias inherentes del sistema, como la planificacion por niveles (que, según los autores limita el paralelismo), con iteraciones de Jacobi clásicas y asíncronas por bloques, permitiendo un paralelismo mucho más fino.

De la misma forma, en [122] se presenta una implementación paralela bastante básica (basada en *level-scheduling*) de ILU-PCG con el fin de resolver problemas de CFD, en particular, la ecuación de Poisson a partir de discretizaciones espaciales para dominios irregulares.

En [123] se propone una nueva forma de paralelizar la obtención de la descomposición matricial de LU, mediante la reformulación matemática y la resolución de un conjunto de ecuaciones bilineales, que pueden resolverse de forma asíncrona y paralela (utilizando por ejemplo algoritmos iterativos de punto fijo). Posteriormente los mismos autores presentan ParILUT [124], un ILU basado en umbrales (*Threashold-based ILU*), que se basa fuertemente en el patrón de dispersión de la descomposición LU.

En [125] se presenta un algoritmo paralelo escalable para construir precondicionadores de factorización incompleta, en particular ILU(k) e ILUT, orientado a resolver sistemas lineales de gran escala en arquitecturas distribuidas. La propuesta se basa en particionar el grafo asociado a la matriz de dependencias en subdominios equilibrados, reordenar los nodos de cada subdominio para maximizar el paralelismo y aplicar un coloreo sobre el grafo de subdominios para organizar el procesamiento de las fronteras sin introducir dependencias innecesarias. Este enfoque permite calcular la factorización incompleta de manera concurrente, manteniendo bajo el coste de comunicación y preservando la calidad del precondicionador.

Sangback y Saad en [126] avanzan en versiones de ILU(0) de forma distribuida que aprovechan diferentes enfoques de manejo de las dependencias, tanto *multicoloring* como *level-scheduling*, obteniendo resultados que favorecen

a la estrategia de coloreo. Recientemente en [127] propone una estrategia para la descomposición de dominio basada la factorización incompleta (ILU) en paralelo de dos niveles para GPU. Lo relevante es que, esta misma estrategia, también es probada como smoother para el nivel más fino de AMG, en CPU. Este trabajo en GPU utiliza implementaciones de cuSPARSE (por ejemplo, ILU(0)), basadas en planificación por niveles para resolver las dependencias y aprovechar el paralelismo. Este enfoque fue ampliamente optimizado en trabajos recientes, por ejemplo en [128] mediante la realización de un análisis previo y la explotación de técnicas de *synchronization-free*.

En [129] se presenta StructILU, un esquema ILU paralelo sobre mallas estructuradas, que toma ventaja de la geometría del problema y preserva las dependencias de los datos del método secuencial mientras explota el paralelismo jerárquico. En esta línea, presenta tres jerarquías de dependencias: malla, intra-fila e intra-elemento. Cada jerarquía es resuelta con diferentes estrategias, el paralelismo de malla se explota mediante una asignación de tareas basada en dependencias acíclicas, el paralelismo intra-fila se optimiza mediante el correcto uso de las memorias y sus latencias de acceso asociadas, y el paralelismo intra-elemento se mejora mediante operaciones matriciales especializadas que aprovechan los *Tensor Cores* de las GPU modernas.

3.4.3. DILU Multicolor AmgX - Línea Base

La biblioteca AmgX de NVIDIA cuenta con una implementación de Diagonal Incomplete LU (*multicolor_dilu_solver*), encargada de aplicar el smoother a la grilla del nivel correspondiente. La rutina toma una matriz y su definición de colores y realiza dos sustituciones, una hacia adelante (*forward*), procesando las filas correspondientes a cada color de forma ascendente por colores, y otra hacia atrás (*backward*) sobre la matriz. Más detalladamente, estos dos pasos están implementados con un core secuencial, que itera recorriendo los colores y aplicando, para cada uno de ellos, un *kernel* de GPU encargado del cómputo necesario según los coeficientes no ceros que tenga cada fila perteneciente al color.

Dado un color, una sustitución (tanto para adelante como para atrás) consiste en lanzar tantos hilos como sea necesario para cubrir las filas de ese color (o 4096 si la cantidad es más grande), siendo que cada fila es procesada por 8 hilos. Como se ve en el Algoritmo 2 cada hilo es asignado a una única fila

según su posición en la grilla y es encargado de iterar entre los coeficientes no nulos de la misma. Para cada coeficiente distinto de cero que contenga una fila se verifica si la columna a la cual pertenece es de un color ya computado (un color menor en *forward* o mayor en *backward* al color en cuestión) y en ese caso se procesa, acumulando el producto entre el coeficiente no cero y el valor de la solución hasta el momento de su columna. Una vez computados todos los no ceros de una fila, los acumulados son reducidos a un único valor y se guarda el resultado en memoria.

Luego de terminar el *kernel* para dicho color, se avanza al siguiente color, ya sea para adelante o para atrás, hasta procesar la totalidad de colores en ambas sustituciones, resultando en un barrido bidireccional que atenúa las componentes de error de alta frecuencia en la solución aproximada, cumpliendo el rol de suavizado dentro del ciclo multigrilla.

Algoritmo 2 Pseudocódigo del *kernel* original de la sustitución hacia adelante para un color, propuesta por la biblioteca AmgX [99]

Entrada: Matriz A en CSR (A_vals, A_cols, A_rows), Vectores b, $delta$, $Einv$ y x

Calcular $warp\_id$ del hilo
Calcular $lane\_id$ del hilo
Calcular número de fila al que pertenece (it_{row})
for cada fila perteneciente al color **do**
  $id_{row} = sorted\_rows\_by\_color[it_{row}]$
  El primero de la fila carga $my\_b = b[id_{row}]$
  Se obtiene col_{it} y col_{end} limites de la fila id_{row}
  while Algún hilo del $warp$ le falte procesar **do**
    $col_{id} = A\_cols[col_{it}]$
    Cargar $my\_x = x[col_{id}]$
    if La fila del color de la columna es anterior: $row\_colors[col_{id}] < current\_color$ **then**
      $my\_x+ = delta[col_{id}]$
    end if
    Cargar $my\_val = A\_vals[col_{it}]$
    Actualizar $my\_b = my\_val * my\_x$
    Incrementar col_{it} según la cantidad de hilos por fila (8)
  end while
  Reducir todos los my_b de la misma fila a uno solo
  El primer hilo guarda en memoria $delta[id_{row}] = Einv[id_{row}] * my\_b$
end for

Para el resto de la tesis, la rutina de AmgX en la versión 2.5.0 anteriormente presentada, a la cual se hará referencia como $MILU_{BASE}$, se adopta como línea base y es establecida como punto de partida para las propuestas presentadas en el Capítulo 5.

Capítulo 4

Optimización del cómputo del método SIP en GPU

En este capítulo se resume el trabajo realizado para mejorar el desempeño computacional del método SIP sobre GPUs. Como se menciona anteriormente, el cómputo del método SIP es una de las etapas principales de la herramienta $Caffa3D.MBRi$ desde el punto de vista de costo computacional.

Sobre la base de los esfuerzos presentados en el Capítulo 3, se realizo un nuevo *profiling* con la herramienta *Nsight Systems* de NVIDIA, del cual se obtuvo como resultado que las mismas cuatro rutinas identificadas en trabajos previos, continúan siendo las que más tiempo consumen en relación a la cantidad de veces que se invocan, por lo que reducir su tiempo de ejecución sigue siendo de interés si se quiere acelerar el solver.

A partir de este análisis es que, en las secciones siguientes, se propone un nuevo mecanismo de planificación del solver SIP con el objetivo de mejorar el enfoque basado en niveles. La propuesta está motivada en el hecho que muchos puntos agrupados en un mismo nivel podrían procesarse antes de que se complete el nivel previo. El mecanismo de sincronización propuesto se basa en planificación por nodos o elementos y evalúa de forma intensiva las dependencias entre nodos de la malla en arquitecturas GPU, resolviéndolas y permitiendo computar inmediatamente después que éstas estén resueltas. Además se realiza un análisis comparativo con la sincronización clásica basada en niveles dentro del marco del SIP y se aplica a un problema del mundo real de dinámica de fluidos.

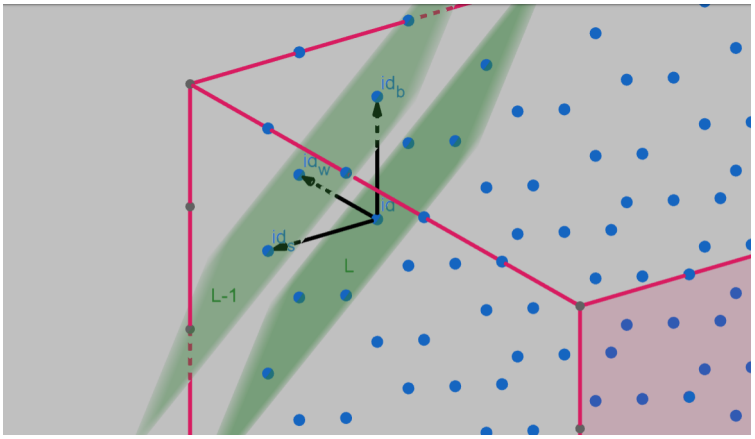


Figura 4.1: Dependencias del punto *id*. En verde se presentan dos niveles o planos. El punto *id* perteneciente al nivel L, mientras que $id_{W/S/B}$ pertenecen al nivel $L-1$

4.1. Propuesta de planificación por elementos

En este apartado se presenta el nuevo mecanismo de planificación propuesto para acelerar el solver SIP, al que llamamos SIP_{ES} (por su sigla en inglés, *Element Synchronization*). Las mismas ideas pueden ser también aplicadas a otros problemas de *stencil* basados en niveles. La propuesta se presenta utilizando una de las cuatro rutinas principales del SIP. Esta rutina realiza la sustitución hacia adelante resolviendo el sistema lineal correspondiente a la línea 4 del Algoritmo 1. Además, la mecánica de cómputo puede ser aplicada directamente a cualquiera de las otras tres rutinas del SIP de alto costo computacional.

Como se dijo en la Sección 3.3.3, cada punto de la malla puede depender de hasta otros 3 puntos, que corresponden al nivel o hiperplano anterior. En otras palabras, un punto *id* que pertenece al nivel L depende de id_W (Oeste), id_S (Sur) e id_B (Inferior), los cuales pertenecen al nivel $L-1$, como se muestra en la Figura 4.1.

En la nueva estrategia, en lugar de iterar sobre un conjunto de niveles, cada hilo procesa únicamente un punto de la malla. Esto implica que el kernel debe lanzar suficientes hilos para cubrir toda la malla en lugar de solo el nivel de mayor tamaño. Aunque el nuevo enfoque incrementa significativamente la cantidad de hilos a utilizar, esto no se traduce necesariamente en una sobrecarga mayor en cuanto al manejo de hilos en la GPU. Además, las GPU modernas han incrementado notablemente su número de núcleos y otros recursos en los últimos años, por lo que un mayor número de hilos puede incluso contribuir a

una utilización más eficiente del dispositivo.

Al inicio del *kernel*, cada hilo debe esperar a que los hilos previos calculen los valores de sus tres dependencias. Esto se logra haciendo *polling* sobre las tres posiciones correspondientes de un vector, consultando su valor, el cual contiene un 0 si el valor no está listo y un 1 en caso contrario. Dado que esta consulta implica acceder a la memoria global de la GPU, el *warp* que realiza la operación de lectura se suspende, lo que provoca que el planificador seleccione otro *warp* para ejecutar. De esta forma, no existe el riesgo de que un *warp* bloquee la GPU, siempre que los hilos dentro de un *warp* pertenezcan al mismo nivel. De lo contrario, un hilo podría depender de otro del mismo *warp*, lo que obligaría al *warp* a esperar a que este mismo termine la ejecución, generando un bloqueo (*deadlock*). Este problema se evita configurando una grilla bidimensional de hilos, donde la dimensión y corresponde a los planos y la dimensión x tiene suficientes hilos para cubrir el plano de mayor tamaño. De este modo, se garantiza que los hilos de cada *warp* (32 hilos consecutivos en la dimensión x) procesan el mismo plano, al costo de tener un número considerable de *warps* inactivos (porque quedan por fuera de los limites del plano correspondiente). Se estima que este costo es despreciable en comparación con las ganancias derivadas del nuevo esquema de planificación.

4.1.1. Optimizaciones de memoria

Como es sabido, organizar el acceso a la memoria de manera eficiente es de vital importancia, especialmente en aplicaciones, como en este caso, donde el acceso memoria es una de las principales limitantes. Por lo tanto, además del nuevo esquema de planificación de hilos, para la propuesta se realizan optimizaciones específicas de memoria para mejorar el rendimiento del *kernel*. Aunque estas optimizaciones son más específicas, también pueden aplicarse a los cuatro *kernels* que componen el solver SIP de Caffa3D.MBRi.

Para llevar a cabo esta propuesta, se realizó un análisis detallado de las dependencias involucradas en el cómputo de cada hilo. Como resultado, se obtuvo que para el cálculo de cada punto de la malla, es necesario cargar información desde la memoria global. Algunos de estos valores presentan la particularidad de no depender de los puntos vecinos (por ejemplo, id_w, id_s e i_b en el caso de id), y es sobre estos que se busca trabajar.

La optimización principal consiste en realizar un *prefetching* de todos aque-

llos datos que no son modificados por el *kernel* antes de detenerse a esperar por sus dependencias, aprovechando el tiempo que cada hilo desperdicia esperando sobre el vector de dependencias. Esto se basa en que los *warps* solo se bloquean por dependencias de datos, de modo que realizar *prefetching* de estos valores permite tener en curso esas transacciones de memoria mientras se espera a que otros *warps* procesen los valores requeridos. Por ejemplo, en los pasos de sustitución hacia adelante y hacia atrás, los coeficientes de las matrices L y U en las ecuaciones (3.2) y (3.3) pueden cargarse tan pronto como inicia la ejecución. Por el contrario, los elementos de R^l y δx^l no pueden cargarse hasta que otros *warps* del bloque los modifiquen. En la implementación actual del SIP en $Caffa3D.MBRi$, también pueden precargarse otros valores, como los índices de acceso a la memoria de los hiperplanos, los cuales se almacenan explícitamente para que puedan ser reutilizados por cada *kernel*.

De esta forma, para cada hilo, una vez que sus vecinos terminen de computarse, este tendrá a su disposición los valores necesarios cargados, prontos para realizar el cómputo sin necesidad de acceder a la memoria global del dispositivo y así poder obtener un mejor rendimiento.

Con el fin de prevenir un uso desmedido de los registros del *kernel*, lo que podría afectar negativamente la ocupación de la GPU, se realiza tal *prefetching* almacenando los valores de una de dos formas; o en registros locales del hilo, memoria que posee el menor tiempo de acceso; o en memoria compartida, memoria que tiene un tiempo de acceso menos performante que los registros pero menor a la global, con la ventaja de que es compartida entre hilos del mismo bloque.

Esta optimización se aplica a la mayoría de las nuevas rutinas sincronizadas a nivel de elementos. La rutina de sustitución hacia adelante resultante, que incluye todas las optimizaciones, se resume en el Algoritmo 3, donde *Res* es el vector solución, *Bp* contiene los coeficientes de L para cada punto, *Sync* es el vector de sincronización, y *InSIP*, *LvlsInf* contienen información de indexado.

4.2. Evaluaciones experimentales

En esta sección se presentan las evaluaciones experimentales realizadas para validar el nuevo esquema de planificación. En primer lugar, se evalúa la rutina *SolveLUforFiForwardSIP_GPUk* de forma independiente para diferentes tamaños de malla y cantidades de hiperplanos. En segundo lugar, se mide

Algoritmo 3 Pseudocodigo del *kernel* de la sustitución hacia adelante LU_-Forward_2F.

Entrada: *Res*, *Bp*, *InSIP*, *LvlsInf*, *Sync*
Salida: *Res*

1: Inicializar variables como el desplazamiento dentro de la grilla (idx) y del bloque (tid)
2: **if** El hilo esta dentro de los limites de la malla *stencil* **then**
3: Obtener el índice del punto a procesar: $idp = LvlsInf[idx]$
4: Obtener los índices del los vecinos: $ids, idw, idb = InSIP[idp]$
5: Guardar en registros desde memoria global: $Bp_R[0..3] = Bp[idp][0..3]$
6: Guardar en memoria compartida desde memoria global: $Res_S = Res[idp]$
7: **while** hay dependencias sin resolver: $(Sync[ids, idw, idb] \neq 1)$
8: Procesar el punto idp:
$Res[idp] = Res_S - Bp_R[0] * (Bp_R[1] * Res[idb] - Bp_R[2] * Res[idw] - Bp_R[3] * Res[ids])$
9: **end if**
10: Marcar el punto como resuelto: $Sync[idp] = 1$

el rendimiento de las nuevas rutinas sobre un problema real de dinámica de fluidos computacional (CFD).

4.2.1. Plataforma de pruebas

La GPU sobre la que se realizaron todas las ejecuciones presentadas en este capítulo es una NVIDIA GeForce GTX 1080 Ti con 11 GB de memoria, 11Gbps de ancho de banda y una interfaz de 352-bits. Además, esta GPU cuenta con arquitectura Pascal y tiene 3584 CUDA cores. La versión de CUDA utilizada fue la 11.4.

El procesador utilizado es un Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz con 8 cores y 64 GB de RAM, con sistema operativo Linux.

4.2.2. Resultados y Análisis

Para obtener los resultados que se presentan en esta sección, las dimensiones que definen el tamaño del problema (N_i, N_j, y N_k) han sido variadas desde 32 a 512, con un paso fijo de 16 para el caso de N_i y de 32 en los demás.

La Figura 4.2 presenta los *speedups* obtenidos para el *kernel* de sustitución hacia adelante respecto a la versión basada en niveles, escalando las dimensiones de la grilla de modo que la cara frontal de la malla de puntos (N_j y N_k)

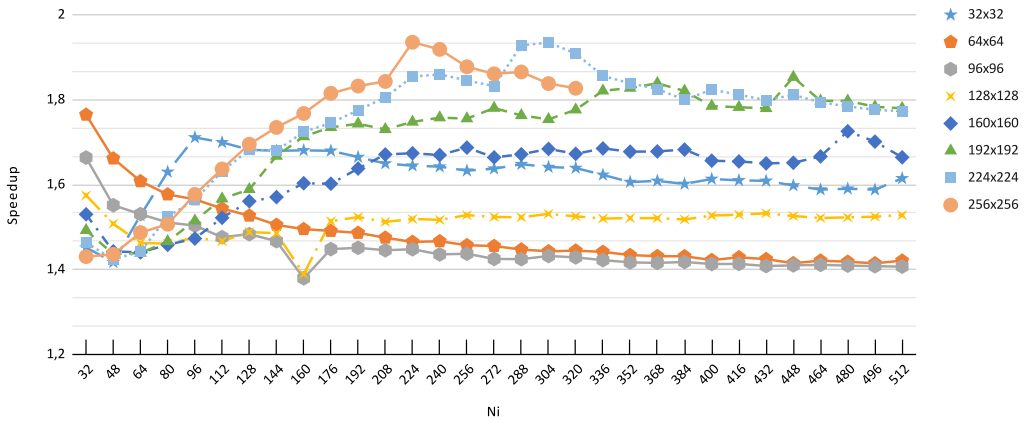


Figura 4.2: *Speedups* de las versiones con sincronización basada en elementos (SIP_{ES}) frente a la basada en niveles (SIP_{LS}), variando N_i y N_j , N_k.

sea cuadrada. La primera observación es que el desempeño de la sincronización basada en elementos (SIP_{ES}) supera claramente al de la implementación basada en niveles (SIP_{LS}), con aceleraciones entre $1,38\times$ y $1,94\times$.

A primera vista, puede notarse que tamaños mayores de N_j y N_k alcanzan *speedups* superiores. Ordenando por tamaño y dividiendo los resultados en dos partes iguales, la primera mitad acumula un promedio de $1,52\times$ y la segunda mitad $1,70\times$. La causa de esta diferencia es la ocupación de la GPU. La tarjeta gráfica NvidiaGTX 1080 Ti puede procesar hasta 8 bloques simultáneamente por SM, y su capacidad máxima se alcanza con aproximadamente 28672 bloques. Por ejemplo, si se observa en la figura la gráfica correspondiente a N_j y N_k iguales a 160, esta comienza a superar el promedio de la primera mitad ($1,5\times$) una vez que N_i sobrepasa 128. Esto se debe a que, para resolver una malla de tamaño $128 \times 160 \times 160$, es necesario lanzar el *kernel* con una grilla de ejecución de tamaño 65×446, lo que implica un total de 28990 bloques, suficientes para aprovechar al máximo las capacidades de la GPU.

Por otro lado, para N_j y N_k iguales a 256, el procedimiento se interrumpe luego de N_i igual a 320, aunque no puede afirmarse con certeza, este comportamiento sugiere una posible insuficiencia de memoria del dispositivo. Sería necesario un análisis más exhaustivo para confirmar esta hipótesis y determinar una solución adecuada.

Exceptuando los tres casos más pequeños, los beneficios obtenidos por SIP_{ES} tienden a aumentar con N_i. Esta dimensión está directamente relacionada con la cantidad de planos y la cantidad promedio de puntos por plano.

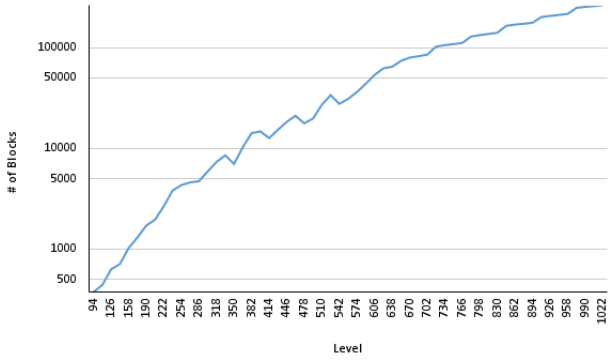


Figura 4.3: Media de cantidad de bloques (# of Blocks) por nivel (Level).

Debido a la construcción de la malla, para $N_j = N_k = m$, la cantidad de puntos por nivel aumenta con N_i hasta que $N_i = m$. Los niveles siguientes mantienen la misma cantidad de puntos hasta los últimos m niveles, donde la cantidad de puntos comienza a disminuir. El aumento y decremento de la cantidad de puntos es análoga en ambos pasos de creación, y responde según la sucesión de números triangulares.

Para seguir con la evaluación, es relevante estudiar el comportamiento de ambas versiones según la cantidad de niveles. Aumentar esta cantidad no siempre afecta proporcionalmente la cantidad de bloques a lanzar. Por cómo es la construcción de los niveles, añadir uno provoca que se añadan los bloques necesarios para cubrirlo, y a su vez, los niveles anteriores se verán modificados (aumentando su tamaño). En otras palabras, observando la Figura 4.3, se puede observar que la gráfica no presenta un crecimiento lineal. Además, podemos ver que si buscamos obtener una cantidad de bloques cercana a 28600 o superior, debemos tener en el entorno de 414 niveles o más.

La Figura 4.4 muestra los *speedups* obtenidos en función de la cantidad de niveles. Puede observarse que, como se mencionó anteriormente, a partir de 414 niveles en adelante, el beneficio de la implementación propuesta comienza a incrementarse. Dado que la versión propuesta adelanta en el tiempo algunas operaciones de cada nivel, resulta razonable observar una mayor aceleración en escenarios con muchos niveles. Cuando hay pocos niveles, el rendimiento de ambos enfoques debería ser relativamente similar. Sin embargo, los resultados muestran primero un incremento y posteriormente una caída en las aceleraciones de los primeros casos de prueba. En la gráfica también se observa el efecto del tamaño de la malla, ya que los casos con menor cantidad de niveles son

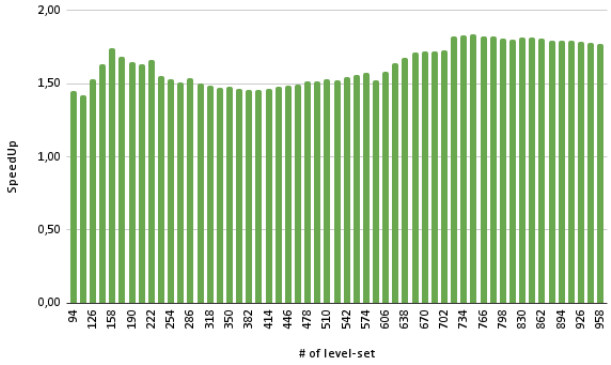


Figura 4.4: *Speedups* de las versiones basadas en elementos (SIP_{ES}) frente a la basada en niveles (SIP_{LS}), variando la cantidad de niveles.

los más pequeños en la dimensión N_i, aunque se requieren pruebas adicionales para poder caracterizar este comportamiento. También es importante ver la relación del desempeño con la dimensión de estos niveles. Por la forma de construcción de la malla, la cantidad de puntos por nivel entre los primeros m y los últimos m niveles (en caso que N_i sea más grande que las otras dos dimensiones) cuentan todos con la misma cantidad de puntos que el nivel m, la máxima. Para analizar cómo impacta esta cantidad en el rendimiento de la propuesta se evalúa los rendimientos en problemas obtenidos de discretizaciones cuadradas de distintos tamaños. Estas últimas cuentan todas con una cantidad de niveles diferente, pero tienen siempre un único nivel de tamaño máximo.

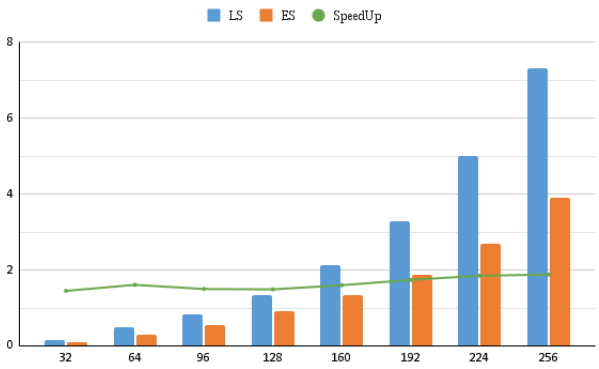


Figura 4.5: Tiempo de ejecución (en milisegundos) y *speedup* para mallas cúbicas de diferentes tamaños (el eje x muestra la raíz cúbica del tamaño del problema).

En la Figura 4.5 se muestra el efecto de variar las tres dimensiones a la vez. Este análisis permite obtener una idea de la escalabilidad débil de la solución

Dimensión	# de niveles	T. SIP_{LS} (ms)	T. SIP_{ES} (ms)	Speedup
32^3	94	0,151	0,104	1,45
64^3	190	0,484	0,301	1,61
96^3	286	0,842	0,560	1,50
128^3	382	1,344	0,903	1,49
160^3	478	2,134	1,331	1,60
192^3	574	3,274	1,878	1,74
224^3	670	5,020	2,707	1,85
256^3	766	7,317	3,897	1,88

Tabla 4.1: Tiempo de ejecución de SIP_{LS} y SIP_{ES} en discretizaciones cubicas de diferentes tamaños.

propuesta. Como puede observarse, el tiempo de ejecución crece a un ritmo considerablemente menor que el tamaño del problema, y la mejora relativa respecto al enfoque basado en niveles se mantiene. A su vez, la Tabla 4.1 añade a las mejoras y los tiempos exactos de ejecución obtenidos, la cantidad de niveles de cada problema. Y junto con los mismos se puede corroborar que la aceleración de la implementación sobrepasa *speedups* de $1,50\times$ y comienza a ser más estable luego de alcanzar la ocupación máxima de la tarjeta gráfica, es decir, luego de sobrepasar los 414 niveles, al igual que en problemas no cúbicos.

4.2.3. Aplicación a un problema de Dinámica de Fluidos

Luego de haber estudiado el rendimiento de la propuesta (utilizando el kernel de sustitución hacia adelante) para una amplia gama de tamaños de malla, se evaluaron las cuatro rutinas con planificación por elementos en un escenario real. Se eligió un problema de dinámica de fluidos que consiste en un canal con escalón orientado hacia atrás (*3D turbulent backward-facing step BFS*) con Re=$5,0\times10^4$, relativo a la altura del canal (como se muestra en la Figura 4.6). Este *benchmark* se utiliza frecuentemente en el ámbito de CFD y corresponde al mismo problema abordado en [8]. Una descripción más detallada de este problema puede encontrarse en [130].

La Tabla 4.2 resume el rendimiento de la nueva versión del solver SIP dentro de $Caffa3D.MBRi$. Donde LU_fw, LU_bw, Res_Inn y LU_coef representan a las rutinas del solver SIP sobre las que se trabajó: *SolveLUforFiForwardSIP_GPUk*, *SolveLUforFiBackwardSIP_GPUk*, *ComputeResInnerCellsSIP_GPUk* y *ComputeLuCoefficientsSIP_GPUk* respectivamente. Aunque la mejora global en el caso BFS se encuentra por debajo del máximo obtenido

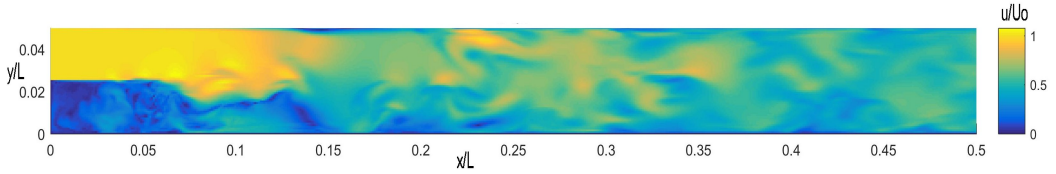


Figura 4.6: Campo de velocidad instantánea, normalizada por la velocidad uniforme de entrada, donde se aprecia la naturaleza turbulenta del fluído.

Nombre	N_i	N_j	N_k	Versión	*LU_fw*	*LU_bw*	*Res_Inn*	*LU_coef*	Total	Accel.
bfsw0	192	192	192	SIP_{LS}	55,8	65,3	84,7	16,0	441,9	$1,21\times$
				SIP_{ES}	36,3	51,8	41,5	17,0	365,6	
bfsw1	1024	64	64	SIP_{LS}	46,7	57,4	67,5	24,4	334,0	$1,15\times$
				SIP_{ES}	43,7	56,8	21,3	27,1	289,7	
bfsw2	512	128	128	SIP_{LS}	57,3	66,1	85,8	19,9	490,8	$1,12\times$
				SIP_{ES}	44,5	59,5	44,7	21,4	437,9	
bfsw3	256	192	192	SIP_{LS}	74,1	83,5	117,3	19,3	588,7	$1,23\times$
				SIP_{ES}	43,8	62,3	56,6	20,3	478,0	
bfsw4	512	64	64	SIP_{LS}	25,2	31,1	36,5	13,4	175,7	$1,17\times$
				SIP_{ES}	23,4	31,0	10,9	14,8	150,7	

Tabla 4.2: Tiempo de ejecución total (en segundos) de los cuatro kernels para las diferentes discretizaciones del problema BFS.

para el *kernel* de sustitución hacia adelante en la sección anterior, se puede observar en la Figura 4.7 que las optimizaciones producen mejoras de rendimiento en todos los casos excepto en *ComputeLuCoefficientsSIP*, que para todos los escenarios implica tiempos de cómputo apenas mayores que la versión original. Sin embargo, dado que este *kernel* se ejecuta solo una vez por cada paso temporal, su impacto en el tiempo total es menor que el del resto. La mayor ganancia de rendimiento se obtuvo en *ComputeResInnerCellsSIP*, obteniendo aceleraciones de entre $1,92\times$ y $3,35\times$ dependiendo del caso.

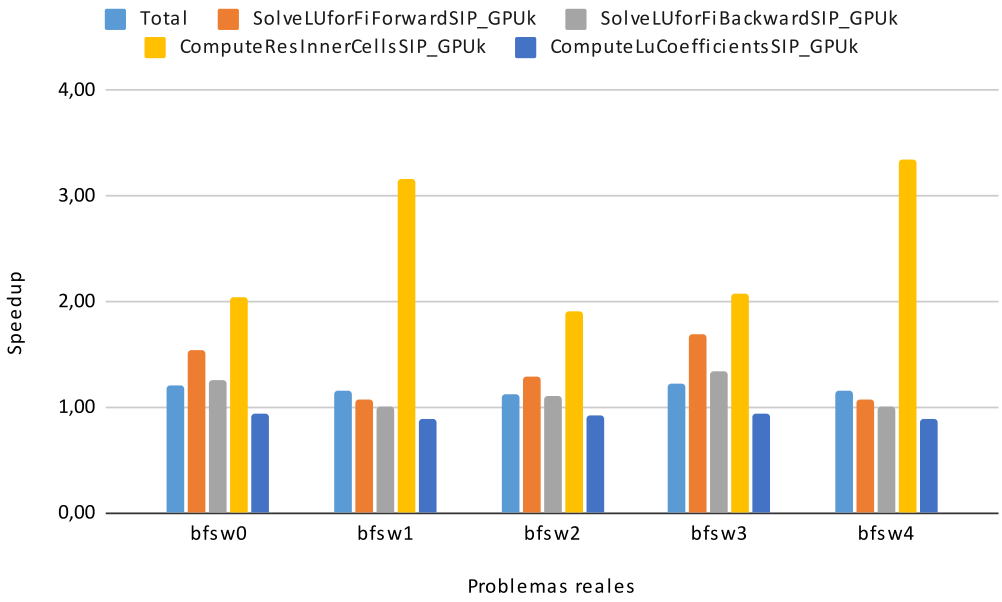


Figura 4.7: Rendimiento de las rutinas involucradas en el solver SIP para las versiones SIP_{LS} y SIP_{ES} sobre los problemas BFS.

Capítulo 5

Optimización de DILU Multicolor, en la biblioteca AmgX de NVIDIA

Al utilizar AMG como precondicionador para mejorar la convergencia de un método iterativo, el papel del smoother elegido es de suma importancia, ya que representa uno de los componentes de mayor costo computacional, por lo que su optimización resulta fundamental para mejorar el rendimiento global del método. La implementación $MILU_{BASE}$, presentada en el Capítulo 3, puede requerir el lanzamiento secuencial de un número considerable de *kernels*, cuya magnitud depende directamente del algoritmo de coloreo empleado. Este esquema introduce limitaciones en el aprovechamiento del paralelismo disponible en plataformas como las GPU y puede generar sobrecargas asociadas al lanzamiento de *kernels* y al acceso a memoria global.

Considerando los aspectos mencionados en el párrafo anterior, en esta sección se presentan las implementaciones propuestas del método DILU Multicolor orientadas a GPUs NVIDIA de arquitectura reciente. El principal objetivo de la propuesta es abatir las limitaciones observadas en la línea base y explotar de manera más eficiente el paralelismo inherente al método.

En esta línea, se presentan las variantes $SFMILU_{2D}$ y $SFMILU_{1D}$, que difieren en el patrón de acceso a memoria de los diferentes *kernels* y las sincronizaciones necesarias para respetar las dependencias de datos inherentes a las sustituciones triangulares.

5.1. SFMILU 2D

La principal diferencia de diseño de la línea base con las propuestas en este trabajo, es que las últimas están basadas en la invocación de un único *kernel* de GPU que procesa todos los colores mientras que en la versión original se invoca un *kernel* de CUDA por cada color. La configuración de la grilla para este *kernel* debe, entonces, contar con la suficiente cantidad de hilos para cubrir todas las filas de la matriz, sin importar su color.

Específicamente, el *kernel* propuesto utiliza 8 hilos por fila, al igual que la línea base, pero utiliza una grilla rectangular de dos dimensiones (ver la Figura 5.1). En el eje vertical se ubican los colores, asignando cada color a una fila de la grilla. En el eje horizontal se dispone un número de hilos suficiente para cubrir al color de mayor tamaño, es decir, aquel que contiene la mayor cantidad de filas de la matriz. Con un análisis previo, liviano en cómputo, es posible determinar la cantidad de hilos que requiere el procesamiento de cada color. El color más grande define así una cota superior de la cantidad de hilos a utilizar para cada uno de ellos. Además, queda directamente determinada una grilla de hilos rectangular (cantidad de colores × cota de hilos por color) lo que simplifica la identificación y asignación entre los hilos.

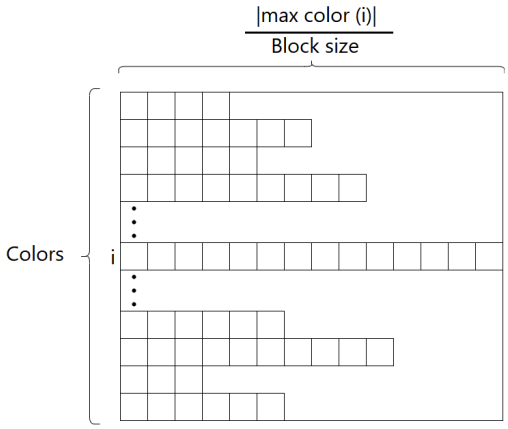


Figura 5.1: Grilla de Kernel para $SFMILU_{2D}$.

La estrategia seguida permite mapear cada fila de cada color a 8 hilos, pero también habilita potencialmente procesar concurrentemente todos los colores, evitando esperas innecesarias. La ventaja de este enfoque se basa en que las filas de un color no necesariamente tienen dependencias con todas las filas de los colores anteriores (sea *forward* o *backward*), por lo que podrían comenzar

a procesarse antes de que se complete el color previo, de haber recursos disponibles en la GPU. Por lo tanto, la estrategia propuesta permite computar cada fila cuando todas sus dependencias ya están resueltas, a diferencia de la versión original, que necesita completar el cómputo de todas las filas de los colores anteriores. Dicho de otra forma, si la fila f_i del color i depende solamente de la fila f_j del color j, para procesar f_i solamente se debe esperar a que f_j sea computada, a diferencia de la línea base, en la que no solo se tiene que esperar por todas las filas del color j sino que también por todos las filas de los colores anteriores a i.

La implementación de la estrategia emplea dos arreglos de enteros (uno para cada sustitución) de igual dimensión que la solución. Cada vez que una fila es procesada se cambia a 1 el valor de la celda del arreglo correspondiente según su índice mientras que las posiciones de filas no resueltas permanecen en 0. En paralelo, aquellas filas que tienen dependencias no resueltas consultan periódicamente las posiciones correspondientes hasta obtener un 1, lo que indica que la dependencia fue resuelta.

Como se menciona anteriormente, la propuesta ofrece más paralelismo que la implementación base, especialmente en los casos donde la matriz contiene muchos colores y las filas contienen un número constante de no ceros cercano a 8. Sin embargo, dependiendo del contexto o la matriz en cuestión, la estrategia puede ser ineficiente desde el punto de vista del trabajo de cada hilo o el balance de carga entre ellos.

Dejando de lado que los bloques sin filas asociadas terminan inmediatamente su ejecución, con esta estrategia de lanzamiento pueden quedar muchos hilos ociosos o con muy poca cargar si la fila a la que pertenecen tiene pocos elementos. Por el contrario, si una fila tiene muchos elementos, los hilos asignados a la misma resultan muy cargados. Es por esto que una asignación fija de hilos por fila, como se propone, genera esfuerzos muy desbalanceados entre los diferentes hilos, provocando que algunos *warps* tengan más carga que otros e incluso diferencias de trabajo entre hilos del mismo *warp*, desperdiciando potencial de cómputo. Un ejemplo puede verse en la Figura 5.2, donde el *warp* j encargado de procesar las filas desde row hasta $row + 3$, distribuye sus hilos de la siguiente manera: los primeros 8 hilos se asignan a la fila row, los 8 siguientes a $row + 1$, los 8 siguientes a $row + 2$ y los últimos 8 a $row + 3$. Dentro del primer grupo, 4 hilos procesan tres elementos cada uno y los 4 restantes procesan dos elementos. En la fila $row + 1$, unicamente los primeros

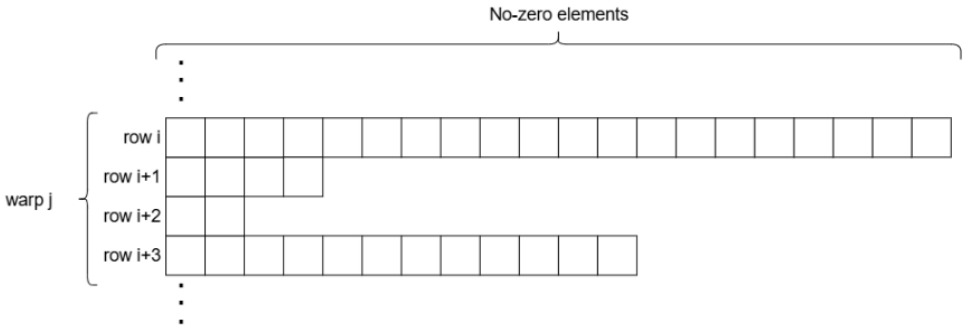


Figura 5.2: Ejemplo de *warp* con desbalance de carga.

4 hilos procesan elementos; en $row + 2$, solo los primeros 2 hilos participan; finalmente, para $row + 3$, los primeros 4 hilos procesan dos elementos cada uno y los últimos 4 hilos procesan un único elemento cada uno. Para mitigar esta problemática implementamos la versión $SFMILU_{1D}$, que se describe a continuación.

5.2. SFMILU 1D

Con el objetivo de alcanzar una estrategia que cuente con las fortalezas de la versión anterior, pero que agregue una distribución equitativa de trabajo entre hilos, se desarrolló la variante llamada $SFMILU_{1D}$.

El desbalance en el cómputo antes referido tiene dos grandes causas:

- La primera se relaciona con la cantidad de hilos totales empleados. El único contexto en el que todos los hilos son utilizados es cuando todos los colores necesitan la misma cantidad de hilos para ser procesados debido a que cada uno tiene asociada la misma cantidad de filas. En cualquier otro caso inevitablemente quedarán hilos ociosos, i.e., dependiendo de cómo sean las dependencias de la matriz, utilizar una cota superior de la cantidad de hilos puede conducir a una asignación de recursos poco eficiente.
- La segunda refiere a diferencias de esfuerzo para procesar las distintas filas. Dado que la versión anterior asigna una cantidad fija de hilos para procesar cada fila, cuando las filas tienen importantes diferencias en la cantidad de coeficientes no ceros (como puede verse en la Figura 5.2), quedan hilos ociosos o sobrecargados.

Para poder abordar los puntos mencionados es necesario acceder a cierta información que permita una correcta asignación de los recursos según la demanda de cada fila. Esto se puede alcanzar mediante una etapa de análisis sobre la matriz, que la recorra y clasifique cada fila según su cantidad de elementos distintos de cero, para que posteriormente se asigne a cada *warp* una cantidad de filas a resolver dependiendo de la cantidad de no ceros de tales filas.

En esta línea, la versión $SFMILU_{1D}$ no utiliza una cantidad fija de 8 hilos por fila, sino que la cantidad de hilos por fila es particular para cada *warp* y puede valer 32, 16, 8, 4, 2 o 1, dependiendo de la cantidad de no ceros de las filas agrupadas en el *warp*. Además, las filas asignadas a un mismo *warp* son de tamaño similar, para que todos los hilos pertenecientes al *warp* deban procesar una cantidad equitativa de elementos. Esta asignación asegura que no se generen desbalances *intra-warp* y provoca que se procese, por *warp*, una cantidad de filas igual a una potencia de dos (1, 2, 4, 8, 16 o hasta 32) en simultáneo, lo que facilita la posterior reducción de los valores de una misma fila a un único valor con la intrínseca *__shfl_xor()*. Por otro lado, las filas que comparten *warp* deben pertenecer al mismo color para evitar *deadlocks*.

En este contexto el análisis utilizado para esta propuesta se centra en reunir, por cada color, filas de tamaño similar y agruparlas, de modo que cada *warp* pueda encargarse de procesar cada uno de los grupos formados.

La etapa de cálculo de la propuesta, posterior al análisis, procede igual que $SFMILU_{2D}$, en el sentido de que se invoca un único *kernel* para procesar todos los colores. A partir de la información detallada de cada fila y de su agrupación dentro de *warps*, es posible determinar con exactitud la cantidad de *warps* requeridos para procesar todos los colores de la matriz. En esta versión, dicha información se utiliza para configurar el lanzamiento del *kernel* con el número mínimo y suficiente de bloques que permitan contener los *warps* necesarios. En consecuencia, el *kernel* se invoca empleando una grilla unidimensional de hilos, optimizando así la asignación de recursos y evitando la sobreasignación innecesaria de bloques.

Tanto en $SFMILU_{2D}$ como en $SFMILU_{1D}$, el tamaño de la grilla se define de modo que siempre haya hilos suficientes para cubrir todos los colores de la matriz. Esto asegura que cada hilo procese exclusivamente los elementos de una única fila, evitando iteraciones adicionales dentro del *kernel*.

5.3. Evaluación Experimental

En esta sección se resume la evaluación experimental de las implementaciones propuestas y el análisis de los resultados obtenidos. La evaluación se realizó en dos etapas diferenciadas. Por un lado, se evalúan las rutinas de manera aislada, es decir, se estudian los rendimientos computacionales de las rutinas específicas y se los compara con sus contrapartes en la línea base para un conjunto amplio de matrices dispersas. Por otro lado, se evalúan los desempeños de los solvers completos (incluyendo las rutinas propuestas o las de la línea base) como parte de la biblioteca AmgX sobre un conjunto más reducido de problemas de CFD.

5.3.1. Plataforma de pruebas

Las evaluaciones experimentales presentadas en este trabajo se realizaron en la plataforma de hardware *labgpu*03, equipada con un procesador Intel(R) Core(TM) i7-6700 @ 3.40GHz, 64 GB de memoria RAM, 64 kB de caché L1, 256 kB de caché L2 y 8 MB de caché L3. El sistema cuenta además con una unidad de procesamiento gráfico (GPU) NVIDIA RTX 3090 Ti perteneciente a la arquitectura Ampere. Respecto a las bibliotecas se utilizó CUDA en su versión 11.4 y AmgX en su versión 2.5.0

5.3.2. Ambiente aislado

Para el estudio aislado se utiliza un conjunto de 1232 matrices dispersas de la colección de *SuiteSparse* [131]. Este conjunto de matrices tiene suficiente variedad en los valores de sus características, como la dimensión (N), cantidad de elementos distintos de cero (nnz) y densidad (nnz/N). La Figura 5.3 resume algunas características de las matrices. Todos los tiempos presentados en esta sección son el promedio de repetir 100 veces cada ejecución.

Para simplificar, a la hora de colorear cada matriz del conjunto se utilizó la primitiva *cusparseDcsrcolor* de la biblioteca CUSPARSE, ofrecida por NVIDIA [132]. El mismo coloreo fue el utilizado para todas las implementaciones, es decir: línea base ($MILU_{BASE}$), $SFMILU_{2D}$ y $SFMILU_{1D}$. Sin embargo, existen bibliotecas que brindan otros algoritmos de coloreo basados en diferentes heurísticas, las cuales provocan variaciones de eficiencia y calidad, en cuanto al tiempo y a la cantidad promedio de colores utilizados [45, 133, 134].

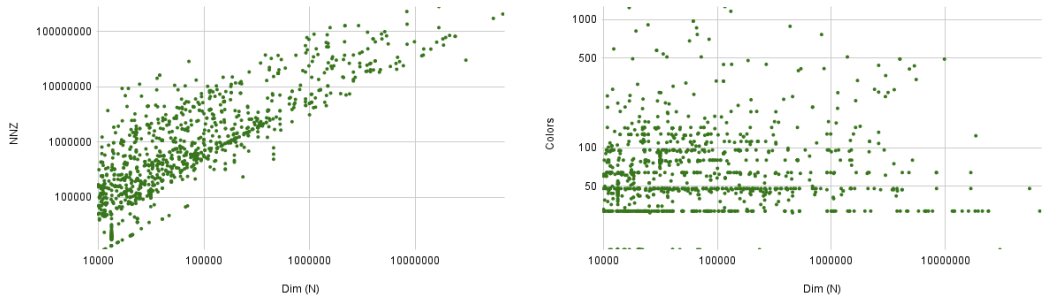


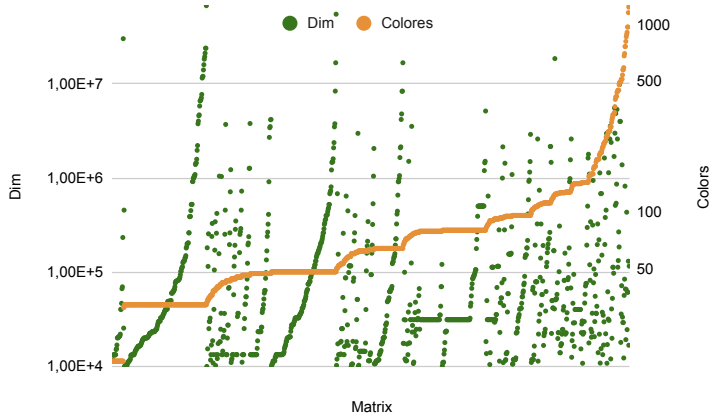
Figura 5.3: Características del conjunto grande de matrices.

El conjunto de matrices considerado en esta sección se ordena según la cantidad de colores obtenidos tras la aplicación del algoritmo de coloreo. De esta forma, las matrices con menor número de colores aparecen en primer lugar, lo que facilita analizar la relación entre el paralelismo disponible y el desempeño de las implementaciones propuestas. En los casos en que las matrices presentan la misma cantidad de colores, el criterio de desempate se establece según la dimensión de la matriz, ordenándolas de menor a mayor tamaño. El orden resultante puede verse en la Figura 5.4a.

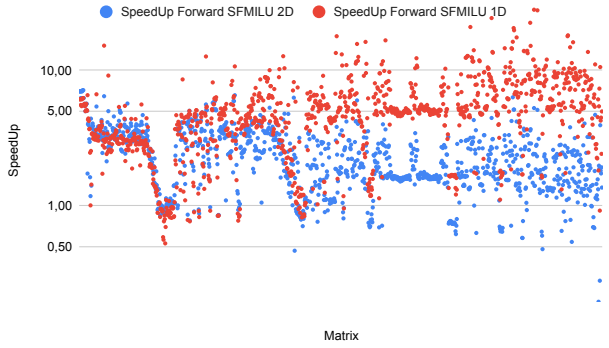
El primer experimento es aplicar las rutinas $MILU_{BASE}$, $SFMILU_{2D}$ y $SFMILU_{1D}$ sobre las matrices del conjunto antes mencionadas. En la Figura 5.4 se muestra en azul y en rojo el valor de *speedup* obtenido por las implementaciones de $SFMILU_{2D}$ y $SFMILU_{1D}$ para cada matriz del conjunto sobre la línea base.

En la gráfica referida a la sustitución hacia adelante (Figura 5.4b) se puede observar que la mayoría de los puntos están por arriba de $1,0$. En el caso de la sustitución hacia atrás (Figura 5.4c), los resultados son similares aunque los *speedups* de esta etapa aparecen desplazados levemente hacia abajo. En otras palabras, si bien se logran reducir los tiempos de cómputo, estas mejoras son menores que en la sustitución hacia adelante. Esta situación se puede explicar a que generalmente los tiempos de ejecución de este paso en la línea base son menores. Lo que significa que algunos costos fijos de la ejecución se mantienen provocando que las optimizaciones trabajen sobre una parte más reducida del tiempo de ejecución.

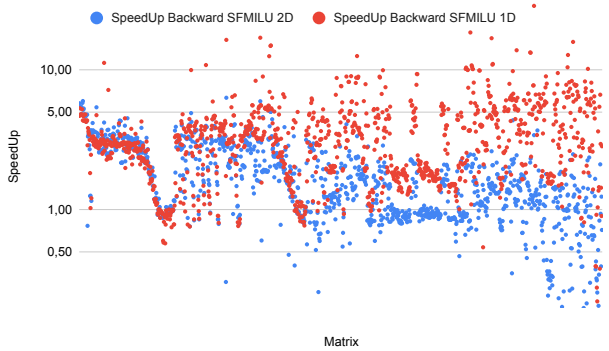
De igual forma, las mejoras son significativas, presentando medias de $2,09\times$ y $4,81\times$ para la sustitución hacia adelante y medias de $1,5\times$ y $3,0\times$ para la sustitución hacia atrás para las versiones $SFMILU_{2D}$ y $SFMILU_{1D}$ respec-



(a) Dimensión y colores de las matrices según el orden establecido.



(b) *forward Speedup* de las implementaciones $SFMILU_{2D}$ y $SFMILU_{1D}$ sobre la línea base.



(c) *backward Speedup* de las implementaciones $SFMILU_{2D}$ y $SFMILU_{1D}$ sobre la línea base.

Figura 5.4: (a) Dimensión y colores de las matrices, (b) *forward* y (c) *backward Speedup* de las implementaciones $SFMILU_{2D}$ y $SFMILU_{1D}$ respecto a la línea base, sobre el conjunto amplio de matrices ordenadas segun el criterio establecido. Los *speedups* por encima de $1,0$ refieren a mejoras obtenidas con las implementaciones propuestas.

tivamente.

Dado el orden en el que se muestran las matrices en la gráfica, se puede observar que el rendimiento de las versiones propuestas tiende a igualarse a la línea base cuando la cantidad de filas promedio por color es muy grande. Esto puede apreciarse mejor en la Figura 5.5, observando que cuando los puntos verdes que están sobre la izquierda de la gráfica toman valores de Dim grandes, los puntos rojos comienzan a tener valores más pequeños, acercándose a $1,0$. Un promedio de filas por color alto implica que la matriz se puede colorear con una cantidad muy baja de colores o que la matriz tiene una gran cantidad de filas en comparación a la cantidad de colores. En ambos casos, es razonable que el rendimiento de las propuestas se parezca al de $MILU_{BASE}$, ya que, proporcionalmente, la cantidad de sincronizaciones no es importante en comparación con el cómputo implicado.

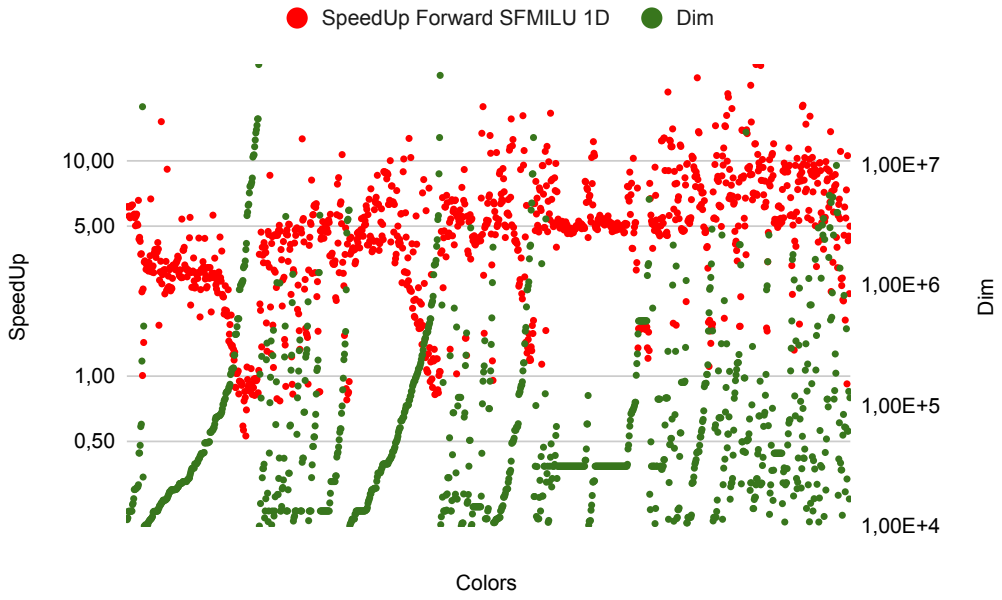


Figura 5.5: Speedup de la sustitución hacia adelante de la propuesta $SFMILU_{1D}$ sobre la línea base ejecutado de forma aislada, y dimensiones para cada matriz, ordenadas por cantidad de colores.

Ambos pasos unidos conforman una aplicación del smoother, por lo que es interesante evaluar su tiempo de cómputo en cada implementación propuesta con respecto a $MILU_{BASE}$. En esta línea, el siguiente experimento estudia los *speedup* de las versiones propuestas frente a la línea base sobre el tiempo total de ambos pasos ejecutados consecutivamente de forma aislada para el conjunto

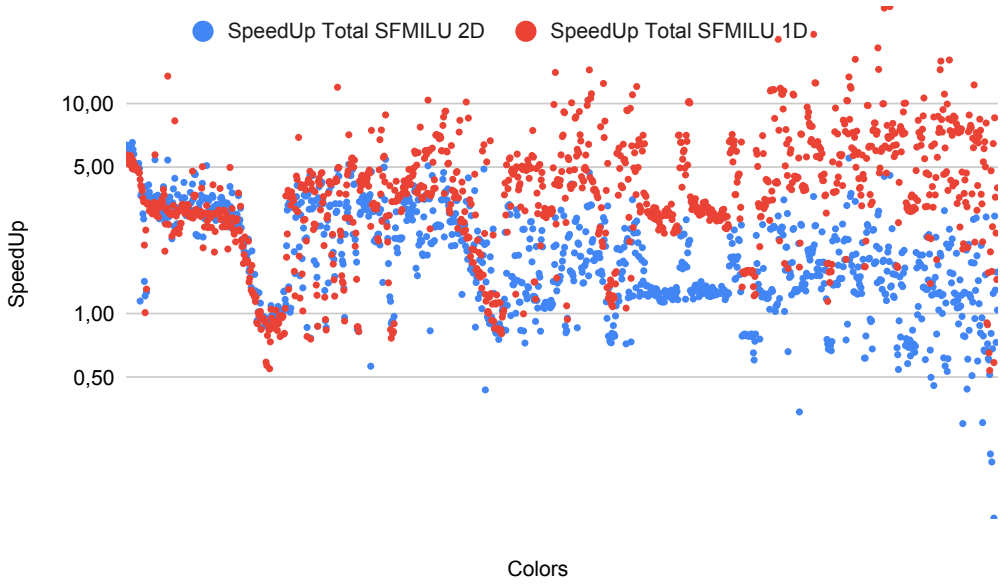


Figura 5.6: *Speedup* Total de las implementaciones $SFMILU_{2D}$ y $SFMILU_{1D}$ sobre la línea base ejecutado de forma aislada, ordenadas por cantidad de colores.

de matrices.

En la Figura 5.6 se resumen los tiempos totales de aplicar el smoother. Los resultados obtenidos siguen la misma tendencia que ambos pasos por separado, tomando un valor intermedio de mejora para cada matriz. Específicamente, se alcanza una aceleración media de $1,8\times$ y $3,7\times$ para la implementación de $SFMILU_{2D}$ y $SFMILU_{1D}$ respectivamente.

$SFMILU_{2D}$ es más estable a lo largo del conjunto de matrices, manteniéndose en valores de *speedups* entre $1,0\times$ y $2,0\times$, mientras que $SFMILU_{1D}$ obtiene *speedups* desde $1,0\times$ hasta $10,0\times$. Estos picos del gráfico se deben a la estructura de cada matriz. En particular, matrices en las que las filas tienen muchos no ceros pueden explotar más recursos computacionales en la variante $SFMILU_{1D}$, ya que en la versión $SFMILU_{2D}$ la cantidad de hilos por fila está limitada en 8. Como puede verse en la Figura 5.7 para la gran mayoría de las matrices que tienen en promedio 8 o más elementos por fila, la versión $SFMILU_{1D}$ obtiene mayores beneficios que $SFMILU_{2D}$. Por otro lado, las matrices que tienen un muy bajo promedio de no ceros por fila, también alcanzan importantes mejoras dado que pueden beneficiarse fuertemente de procesar varias filas en paralelo con un *warp*.

Adicionalmente, es notorio que tanto en las dos etapas de sustitución, co-

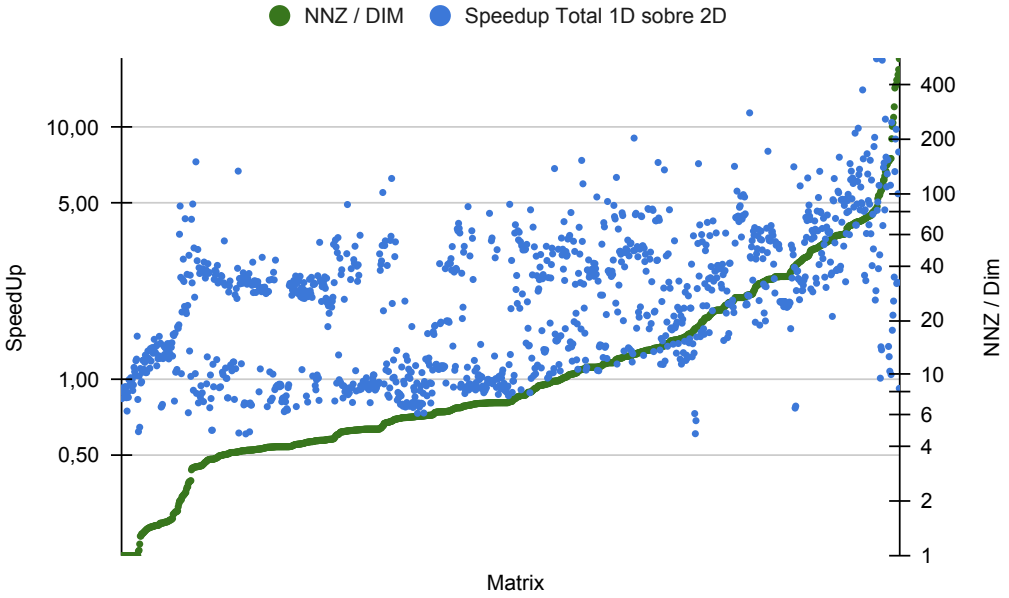


Figura 5.7: *Speedup* del tiempo total del smoother de la implementación $SFMILU_{1D}$ sobre $SFMILU_{2D}$ ejecutada de forma aislada, ordenadas por cantidad de elementos no cero promedio por fila.

mo en el tiempo total del smoother, para la gran mayoría de las matrices la versión $SFMILU_{1D}$ es superior a $SFMILU_{2D}$. Es decir, los puntos rojos de las gráficas en general están por encima de los azules, corroborando así que una distribución de carga más uniforme entre hilos mejora significativamente la eficiencia de estas rutinas. Adecuándose al contexto de la matriz, organizando los *warps* en función a la cantidad de elementos no nulos que contiene cada fila y optimizando el trabajo de cada hilo es que el análisis permite a $SFMILU_{1D}$ tener *speedups* mayores, siendo en promedio $2,4$ veces más rápido que $SFMILU_{2D}$ en ambas sustituciones.

La única situación en la que este comportamiento se invierte levemente ocurre cuando la matriz puede ser coloreada con un número muy reducido de colores. En tales casos, $SFMILU_{2D}$ alcanza un mejor rendimiento que $SFMILU_{1D}$, al evitar la lógica adicional que este último introduce y que resulta innecesaria en escenarios de baja complejidad.

5.3.3. Integración con la biblioteca AmgX

A continuación se presentan los resultados recabados luego de utilizar la biblioteca AmgX. La misma tiene la posibilidad de manipular la interfaz de

configuración mediante un archivo JSON, que proporciona flexibilidad para definir solvers, parámetros de convergencia y estrategias de precondicionamiento sin necesidad de modificar el código fuente.

Para este trabajo la configuración utilizada tiene el método de Gradiente Conjugado Precondicionado (PCG) como solver principal, ampliamente utilizado en problemas CFD, particularmente los derivados de las ecuaciones de *Navier-Stokes* o simplificaciones, siguiendo las líneas de *PCG_ CLASSICAL_ - V* incluido como plantilla en la biblioteca. Como precondicionador se emplea un esquema de multigrilla algebraica (AMG), con *coarsening* agresivo en los dos primeros niveles, ciclo tipo V, un interpolador de tipo D2, el *coarse* solver por defecto (*Dense LU Solver*), y el criterio de convergencia utilizado está basado en la norma del residuo relativo al inicial, con una tolerancia de 1×10^{-06}. Como smoother, en cada nivel se utiliza DILU Multicolor, que permite una resolución paralela por colores, como se presentó anteriormente. El smoother se aplica una vez antes de restringir el residuo (*pre-smoothing*) al siguiente nivel *coarse* y una vez después de prolongar la corrección (*post-smoothing*) desde el nivel *coarse*.

Por otro lado, para las pruebas en este apartado se utiliza un conjunto más reducido que incluye seis matrices, todas ellas del área de *"Computational Fluid Dynamics"* de la colección de *SuiteSparse*. Como se puede ver en la Tabla 5.1 las matrices elegidas son representativas en cuanto a su dimensión, cantidad de colores necesarios para colorearlas, cantidad de no ceros, densidad y desviación estándar (respecto a la cantidad de elementos distintos de cero por fila). Asimismo, si bien es un conjunto pequeño de matrices, en este apartado se evalúan también las diferentes versiones de las matrices en cada nivel de AMG para cada una de ellas. Recordar que las matrices intermedias construidas a partir de productos de Galerkin y utilizadas en el *V-Cycle* tienen características diferentes a la del nivel más fino.

Todos los resultados presentados a continuación son el promedio de 50 ejecuciones.

El Algoritmo 4 muestra de forma simplificada el procedimiento que se utilizó para evaluar el impacto de incorporar las nuevas rutinas a la biblioteca AmgX. El flujo de ejecución comienza cargando la matriz en formato CSR (en la línea 1) y la configuración a utilizar desde un archivo JSON en la línea 2, definido más adelante. Con esta configuración, en la línea 3 se crea e inicializa el objeto encargado de manejar los recursos de GPU. Las estructuras internas

Algoritmo 4 Invocación de ejecución

Input: File with A in CSR, $config$ in json file

1: Load matrix from file
// Create cfg from $config$ file:
2: AMGX_config_create_from_file(cfg, $config$)
// Create resources object from config:
3: AMGX_resources_create_simple(rsrc, cfg)
// Create matrix and vectors, A, b and x:
4: AMGX_matrix_create(A, rsrc, AMGX_mode_dDDI)
5: AMGX_vector_create(x, rsrc, AMGX_mode_dDDI)
6: AMGX_vector_create(b, rsrc, AMGX_mode_dDDI)
// Bind vectors b and x with the matrix A:
7: AMGX_vector_bind (b, A)
8: AMGX_vector_bind (x, A)
// Load the matrix to AmgX from CSR format:
9: AMGX_matrix_upload_all(A, dimension, nnz, csrRow, csrCol, csrVal, 0)
// Initialize vectors in AmgX (x=0 and b=1):
10: AMGX_vector_set_zero(x, n, 1)
11: AMGX_vector_upload(b, n, 1, b_local) { b_local is a Ones vector}
// Create solver:
12: AMGX_solver_create(solver, rsrc, AMGX_mode_dDDI, cfg)
// Invoke solver setup with A:
13: AMGX_solver_setup(solver, A)
// Invoke solver solve with b and x,:
14: AMGX_solver_solve(solver, b, x) { Solve iteration and smoother applications }
// Get status of the solver:
15: AMGX_solver_get_status(solver, status)
// Destroy A, b, x and resources:
16: AMGX_matrix_destroy(A)
17: AMGX_vector_destroy(b)
18: AMGX_vector_destroy(x)
19: AMGX_resources_destroy(rsrc)

Matrices	Nivel	Dim	NNZ	Promedio de NNZ	Desviación Estándar por fila	Colores
parabolic_fem	1	525825	3674625	6,99	0,15	9
parabolic_fem	2	51540	418740	8,12	1,18	9
parabolic_fem	3	5826	48990	8,41	1,70	11
parabolic_fem	4	2134	24636	11,54	3,01	13
parabolic_fem	5	737	7911	10,73	2,87	13
poisson3Da	1	13514	352762	26,10	13,76	37
Goodwin_071	1	56021	1813056	32,36	15,90	41
Goodwin_071	2	1696	22550	13,30	2,43	17
ifiss_mat	1	96307	3610797	37,49	13,35	45
ifiss_mat	2	5093	100723	19,78	4,12	23
ifiss_mat	3	456	5706	12,51	3,45	15
raefsky3	1	21200	1488768	70,22	6,33	83
raefsky3	2	1615	65673	40,66	12,34	49
ns3Da	1	20414	1680777	82,33	43,96	141

Tabla 5.1: Características de las matrices correspondientes a problemas de CFD.

necesarias son creadas en las líneas 4, 5 y 6, la matriz A en formato CSR y los vectores x y b respectivamente, que quedan asociados (en las líneas 7 y 8) entre sí. Luego, en las líneas 9, 10 y 11 se inicializan dichas estructuras, y posteriormente, en las líneas 12 y 13, se construye el solver, instancia en la cual AmgX prepara la jerarquía multigrilla generando los niveles *coarse*, los operadores de transferencia y los smoothers. La resolución del sistema se realiza a partir de la instrucción 14, llevando a cabo la ejecución de PCG precondicionado con AMG, donde en cada cambio (ascenso o descenso) de nivel se aplica el smoother seleccionado. Para finalizar, antes de liberar la memoria utilizada en las líneas 16 a 19, en la línea 15 se consulta el estado del solver para verificar la convergencia.

Para estudiar el costo computacional de AmgX se utilizó la herramienta *nsys profile* de NVIDIA. Se analizaron las rutinas implicadas en la resolución mediante el método clásico de AMG para el problema *parabolic_fem*.

La evaluación experimental muestra que las dos rutinas en GPU que implican mayor tiempo de cómputo son:

- $DILU\_forward\_1 \times 1\_kernel$
- $DILU\_backward\_1 \times 1\_kernel$

acaparando entre ambos procedimientos un $57,5\,\%$ del tiempo total de la rutina

AMGX_solver_solve.

Estas dos rutinas son utilizadas durante el proceso del smoother elegido para aplicar al precondicionador. La importancia, en costo computacional, de estas rutinas se debe a que el smoother es invocado en un número elevado de ocasiones durante el proceso. En el esquema utilizado, basado en PCG con precondicionador AMG y smoother DILU Multicolor, las operaciones de sustitución hacia adelante y hacia atrás se ejecutan cada vez que el smoother es aplicado dentro del ciclo multigrilla. Con la configuración establecida (ciclo tipo V y una única iteración de smoother cada vez), el smoother se invoca dos veces por nivel: una antes de la restricción (*pre-smoothing*), para atenuar los componentes de error de alta frecuencia presentes en el nivel actual, y otra después de la prolongación (*post-smoothing*), para eliminar los errores de alta frecuencia introducidos por la interpolación desde el nivel *coarse*. El smoother DILU Multicolor resuelve un sistema triangular inferior y uno superior de forma secuencial por colores como se mencionó anteriormente, lo que se traduce dentro de AmgX en la ejecución de los *kernels DILU_forward_1×1_kernel* y *DILU_backward_1×1_kernel*. Dado que el ciclo multigrilla se ejecuta una vez por iteración de PCG, y que cada aplicación del smoother implica ambas sustituciones, el número total de ejecuciones de estos *kernels* crece rápidamente con el número de niveles y de iteraciones, acumulando así un porcentaje elevado del tiempo total de cómputo. Específicamente, cada *kernel* es invocado $(v_1 + v_2) \times N_{lvl} \times N_{ip} \times N_{is}$ veces, siendo v_1 y v_2 la cantidad de iteraciones del smoother para *pre-smoothing* y *post-smoothing* respectivamente, N_{lvl} la cantidad de cambios de niveles (cantidad de niveles -1); N_{ip} la cantidad de iteraciones establecidas para el precondicionador; y N_{is} la cantidad de iteraciones necesarias para que el solver principal converja. Por ejemplo, en el caso de *parabolic_fem*, con una jerarquía de seis niveles y cinco iteraciones de PCG, el ciclo V aplica un total de $(1+1)\times(6-1)\times1\times5 = 50$ veces cada sustitución. Lo que implica 100 ejecuciones de sustituciones y se traduce en 50 ejecuciones de *DILU_forward_1×1_kernel* y 50 de *DILU_backward_1×1_kernel*. La elevada frecuencia de estas llamadas, combinada con las limitaciones inherentes de paralelismo y acceso a memoria de las sustituciones, terminan contribuyendo de forma significativa al tiempo global de cómputo del solver.

El primer estudio de esta etapa pone foco en comparar las sustituciones hacia adelante y hacia atrás de las diferentes versiones con la línea base sobre el conjunto de matrices seleccionado y sus diferentes grillas intermedias

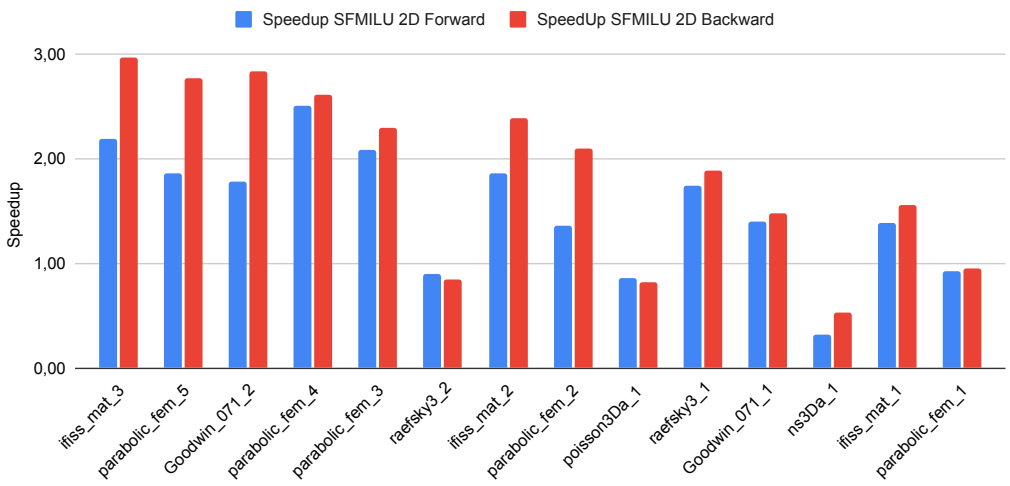


Figura 5.8: *speedups* de la sustitución hacia adelante (*forward*) y hacia atrás (*backward*) de $SFMILU_{2D}$ sobre $MILU_{BASE}$ de AmgX, de forma ascendente según el tamaño de grilla de invocación del *kernel*.

correspondientes a cada nivel. A estas grillas intermedias las notaremos como $name\_L$ donde $name$ es el nombre de la matriz y L es el número del nivel al que pertenece, donde 1 corresponde al nivel más fino.

La Figura 5.8 presenta los *speedups* obtenidos utilizando la versión $SFMILU_{2D}$ en comparación a la línea base. Los mismos alcanzan máximos de aproximadamente $2,5\times$ y $3,0\times$ para la sustitución hacia adelante y hacia atrás respectivamente. Aunque, en la gran mayoría de casos, los resultados están por encima de $1,0$, es notorio que cuanto más grande es la grilla de invocación del *kernel*, menor mejora sobre la línea base alcanza la versión $SFMILU_{2D}$ (en ambas sustituciones). Dado que el tamaño de dicha grilla es la cantidad de colores que posee la matriz por la cantidad de filas del color más grande por 8 hilos, las matrices sobre la derecha de la gráfica presentan un mayor desbalance de carga, ya sea por la varianza en cantidad de filas por color o por la cantidad de hilos asignados por fila. Esto indica que en este tipo de contextos es donde la versión $SFMILU_{1D}$ puede ofrecer mayores ventajas.

En la Figura 5.9 se muestran los tiempos de ejecución insumidos por la versión $SFMILU_{1D}$ sobre la línea base. En este caso, en todas las matrices excepto una (*parabolic_fem_1*) se obtienen mejoras, con medias de $3,1\times$ y $3,5\times$ para *forward* y *backward* respectivamente, aumentando sin excepciones las ganancias de la versión $SFMILU_{2D}$ frente al original, alcanzando picos mayores a $5,0\times$ para *forward* y $6,0\times$ para *backward*. Además, es interesante

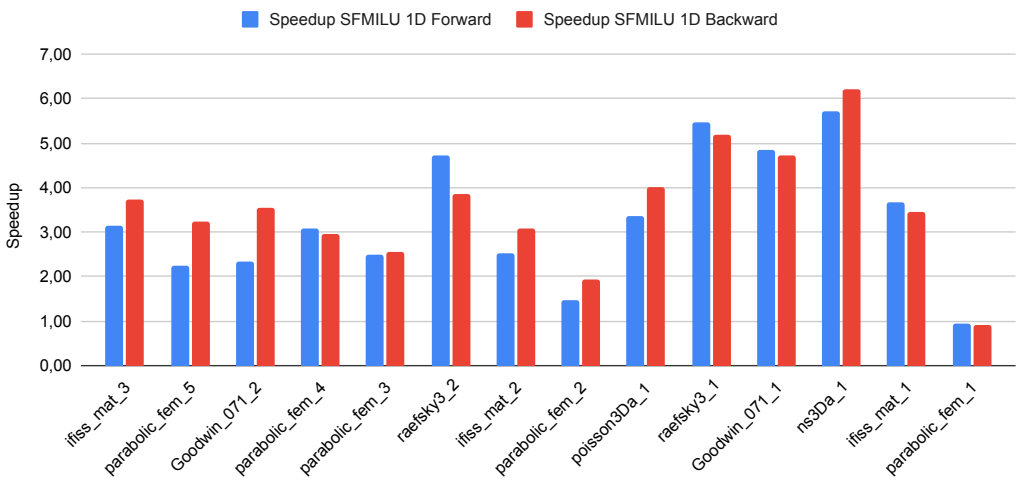


Figura 5.9: *speedups* de la sustitución hacia adelante (*forward*) y hacia atrás (*backward*) de $SFMILU_{1D}$ sobre $MILU_{BASE}$ de AmgX, de forma ascendente según el tamaño de grilla de invocación del *kernel*.

ver que los picos de desempeño ocurren justamente en las matrices en las cuales $SFMILU_{2D}$ no obtiene mejoras respecto a la línea base. Por ejemplo, para *ns3Da_1*, que es una matriz con una alta cantidad de colores (141) y cuyo color más grande tiene muchas filas (lanza 416796 hilos), la versión $SFMILU_{2D}$ obtiene *speedups* desfavorables para *forward* y *backward* ($0,3\times$ y $0,5\times$ respectivamente) mientras que en $SFMILU_{1D}$ los *speedups* correspondientes son $5,7\times$ y $6,2\times$.

Por otro lado, el caso en el que $SFMILU_{1D}$ no presenta un mejor tiempo de ejecución que $SFMILU_{2D}$ frente a la versión original se da en la matriz *parabolic_fem_1*, con *speedups* muy parecidos en ambas propuestas (los cuales rondan entre $0,9\times$ y $1.0\times$) tanto para *forward* como para *backward*. En otras palabras, en este contexto el balance de cargas no impacta en el desempeño. Esto se debe a características muy particulares de la matriz que se describen a continuación, las cuales provocan que la versión original esté por debajo de las versiones propuestas en cuanto a tiempo de ejecución.

La matriz *parabolic_fem_1* tiene la grilla de invocación de *kernel* más grande, sin embargo, al contar con muy pocos colores, y cantidad de filas por colores muy estable y elevada, causa que las implementaciones propuestas no obtengan una mejora significativa en comparación con la versión original. Con estas características la recorrida de colores e invocación de los *kernels* correspondientes a cada color de la línea base no abarcan un cómputo significativo en

relación con procesamiento de las filas. Además, al contar con colores de gran tamaño, cada color tiene la cantidad de filas necesarias para poder saturar la GPU, por lo que no se obtiene un beneficio significativo de ejecutar más de un color a la vez. Por otro lado, esta matriz tiene un promedio muy estable de no ceros por fila (de 7 elementos) provocando que el análisis previo y el balanceo de carga de la implementación $SFMILU_{1D}$ no tenga efecto ninguno, dado que la asignación de recursos para esta matriz es la misma en cualquiera de las implementaciones. En otras palabras, incluso en la versión original, el reparto equitativo de recursos entre las distintas filas es correcto, evitando tener que lidiar con el manejo de sincronización entre hilos.

Si bien este comportamiento se observa en el nivel más fino de la matriz *parabolic_fem* (*parabolic_fem_1*), al descender en la jerarquía multigrilla las matrices intermedias modifican progresivamente sus características estructurales, perdiendo las propiedades favorables presentes en el nivel inicial. Esto significa que la distribución de recursos equitativa realizada por la línea base ya no es óptima, afectando negativamente su tiempo de ejecución y provocando la ventaja de las implementaciones $SFMILU_{2D}$ y $SFMILU_{1D}$ que se puede ver en la Figura 5.10.

A medida que el nivel de AMG aumenta, la dimensión y cantidad de coeficientes no ceros de la matriz disminuye, mientras que la densidad, es decir, la cantidad de elementos diferentes de cero por fila tiende a aumentar [135]. En la Figura 5.11 se observan los resultados de $SFMILU_{1D}$ en función del promedio de elementos distintos de cero por fila. Allí se puede ver que, a mayor densidad, mayor es el beneficio obtenido, por lo que la versión $SFMILU_{1D}$ teóricamente debería tender a obtener mejores tiempos a medida que los niveles aumentan. Experimentalmente se puede apreciar que lo anterior depende mucho del contexto. Si bien es una tendencia, hay matrices para las cuales el aumento del nivel provoca diferentes efectos, tanto en las características sobre las grillas (notar que en la Figura 5.11 los niveles de una misma matriz están desordenados) como en el rendimiento de las versiones sobre estas, habiendo matrices en las que el *speedup* aumenta, disminuye u oscilan a lo largo de los diferentes niveles. Por esto último, y con el objetivo de obtener una visión general de la ejecución, es interesante ver como evoluciona el tiempo total de la resolución incluyendo, para cada matriz, los diferentes niveles en cada iteración del AMG.

En la Figura 5.12 se puede ver cómo impactan $SFMILU_{2D}$ y $SFMILU_{1D}$ sobre el tiempo total de la ejecución de la solve de AmgX (rutina $AMGX\_$-

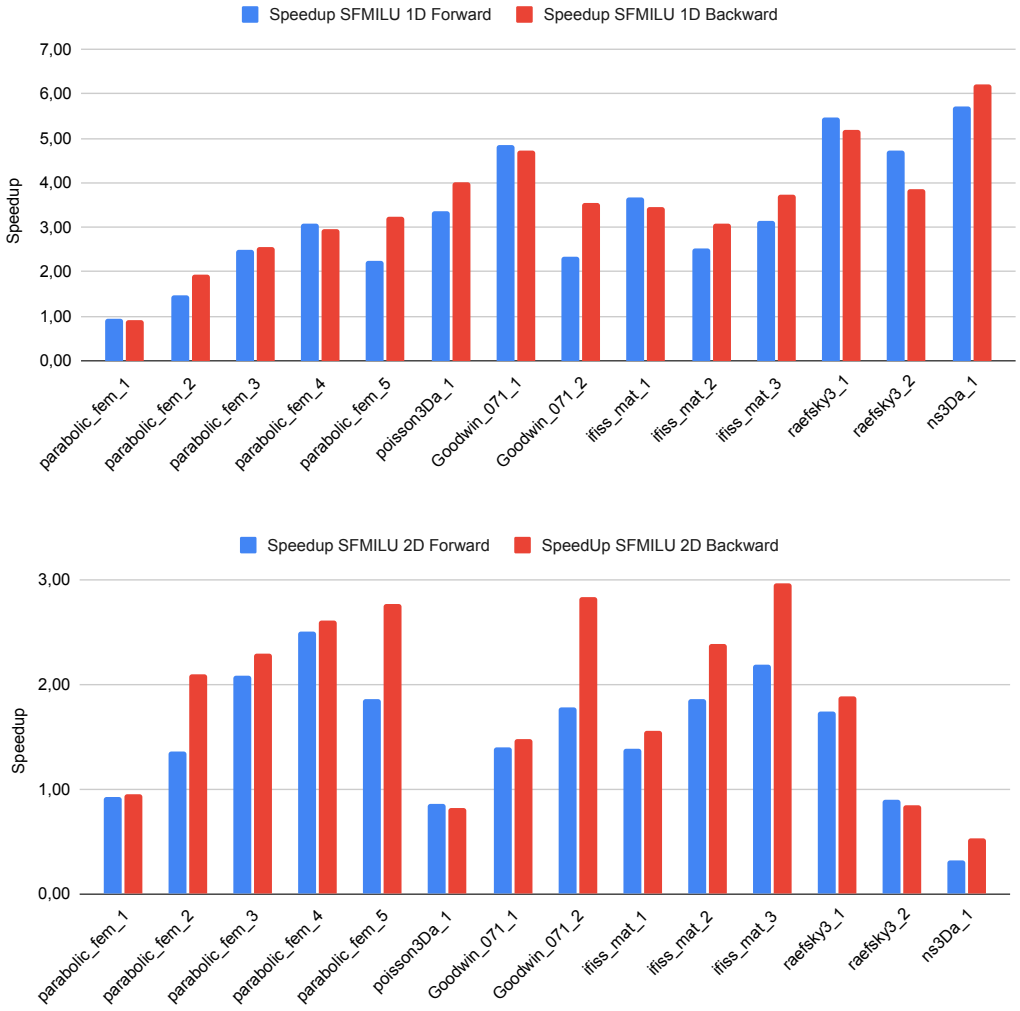


Figura 5.10: *speedups* de la sustitución hacia adelante (*forward*) y hacia atrás (*backward*) de las versiones $SFMILU_{1D}$ y $SFMILU_{2D}$ sobre $MILU_{BASE}$ de AmgX, ordenado según los distintos niveles de cada matriz, por cantidad de colores del nivel mas fino.

Matrices	Cant de Iteraciones	$MILU_{BASE}$	$SFMILU_{2D}$	$SFMILU_{1D}$
parabolic_fem	5	4,18	3,47	3,49
poisson3Da	5	2,75	3,18	1,18
Goodwin_071	5	3,68	2,64	1,47
ifiss_mat	3	5,17	3,56	2,34
raefsky3	2	8,48	6,65	2,63
ns3Da	4	20,28	51,46	4,09

Tabla 5.2: Cantidad de iteraciones y tiempos (en milisegundos) de cada iteración dentro de la solve para $MILU_{BASE}$, $SFMILU_{2D}$ y $SFMILU_{1D}$.

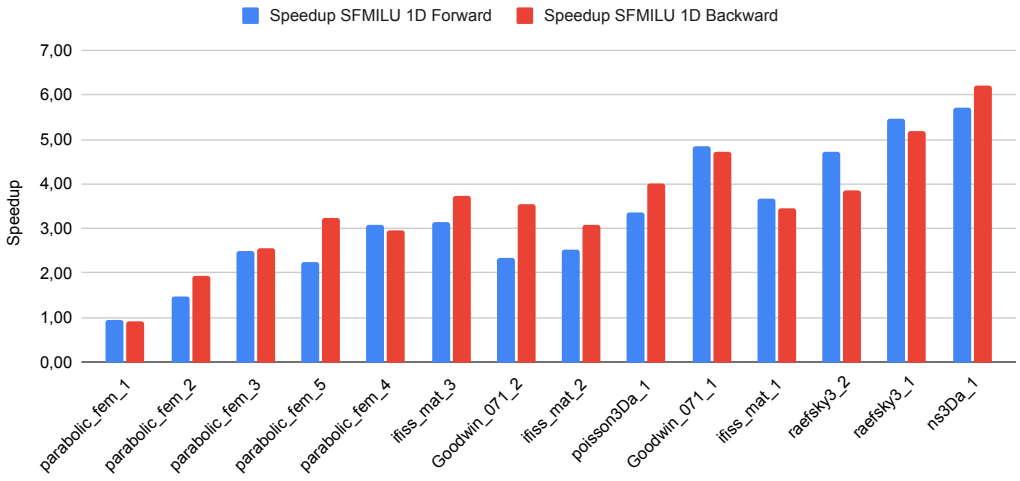


Figura 5.11: *Speedup* de la sustitución hacia adelante (*forward*) y hacia atrás (*backward*) de $SFMILU_{1D}$ sobre $MILU_{BASE}$ de AmgX, en orden según la cantidad promedio de no ceros por fila.

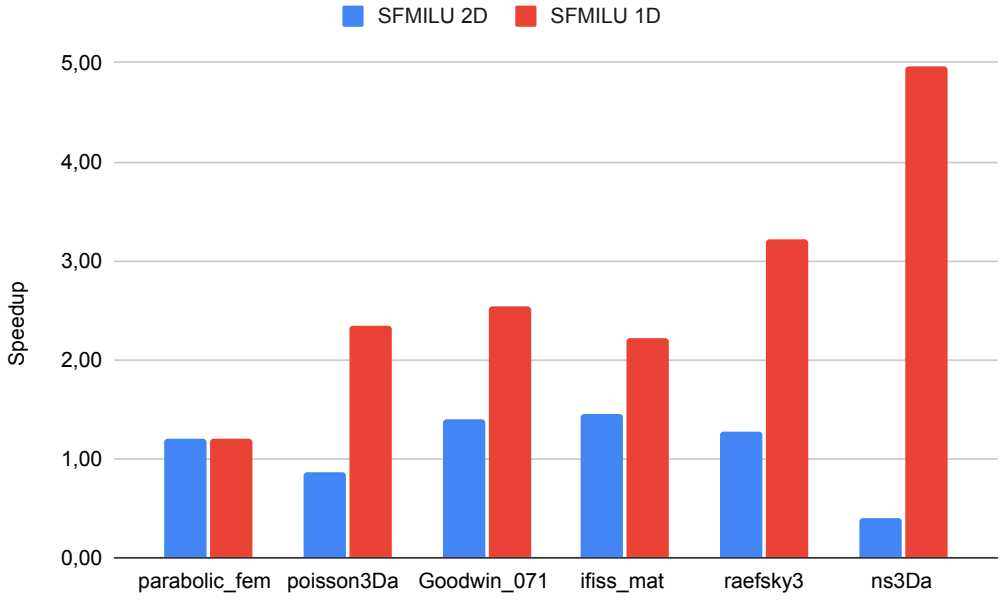


Figura 5.12: *Speedup* de las versiones $SFMILU_{2D}$ y $SFMILU_{1D}$ sobre $MILU_{BASE}$ para toda la etapa solve de AmgX, según la cantidad de colores del nivel mas fino.

solver_solve), ordenando los resultados por la cantidad de colores de la matriz inicial. En azul y rojo se resume la comparación de las versiones $SFMILU_{2D}$ y $SFMILU_{1D}$ respectivamente frente a la original.

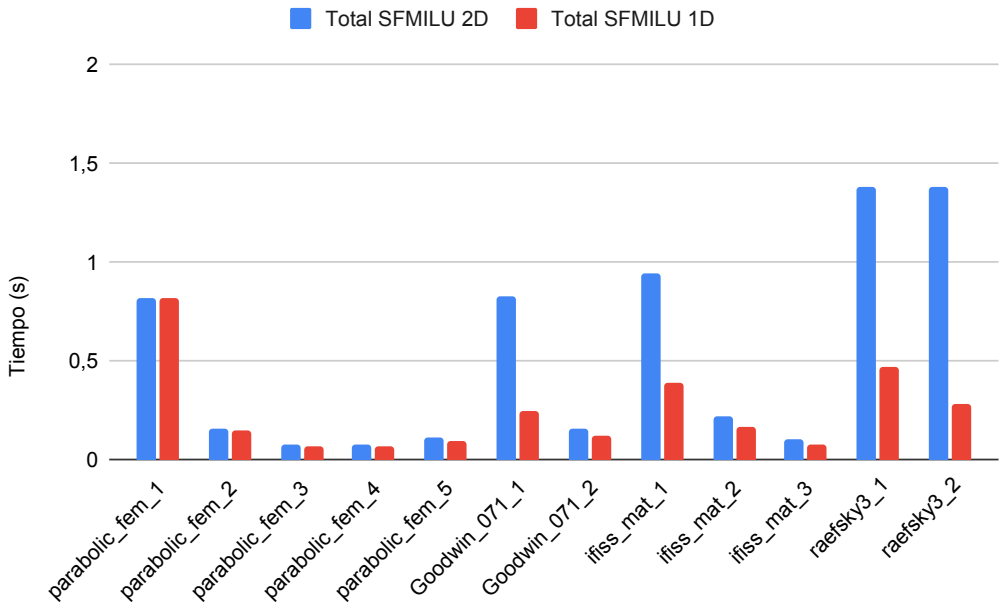


Figura 5.13: Tiempo de ejecución de AmgX para cada nivel en las versiones $SFMILU_{2D}$ y $SFMILU_{1D}$ sobre las matrices que poseen mas de un nivel.

En la Figura 5.13 se pueden ver los tiempos de ejecución totales de las versiones propuestas para las matrices que cuentan con más de un nivel, discriminado por su profundidad. A pesar de que los primeros niveles tienden a ser los que tienen más influencia sobre el tiempo global del solver, se puede apreciar que ambas implementaciones obtienen mejoras en el tiempo global para matrices en que no alcanzan mejoras en su nivel más fino (por ejemplo *parabolic_fem*).

En rasgos generales, en los experimentos realizados se observa que las ganancias o pérdidas obtenidas en las sustituciones de cada nivel se correlacionan casi directamente con el tiempo global de la ejecución. Sin embargo, esta no es la única característica que lo condiciona. Ambas versiones propuestas añaden el tiempo de la creación e inicialización de estructuras de datos en memoria para su implementación. Para cada nivel se crean los vectores para el manejo de dependencias correspondientes a la cantidad de filas de la malla, además del análisis en el caso de $SFMILU_{1D}$. La creación de estructuras se realiza una única vez, lo que impacta de manera leve en el tiempo de resolución. En

cambio, la inicialización de algunas estructuras se debe realizar para cada iteración en cada nivel, por lo que antes de aplicar las sustituciones se consume un tiempo extra en la inicialización de los vectores utilizados para la resolución de dependencias. Como se puede ver en la Tabla 5.2, pese a este tiempo extra agregado en cada iteración, la versión $SFMILU_{1D}$ únicamente es superada por la original en matrices donde la variante $SFMILU_{2D}$ no obtenía mejoras, mientras que para el resto se sigue manteniendo la mejora, por lo que la inicialización no afecta drásticamente al tiempo de cada iteración ni al global de la resolución.

En la versión original, los tiempos de ejecución de una sustitución, tanto hacia atrás como hacia adelante, tienen media de $0,3$ milisegundos para la gran mayoría de matrices del conjunto. Este tiempo es relativamente pequeño comparado con la media del tiempo global $17,69$ milisegundos que requiere la ejecución del solver completo. Sin embargo, la frecuencia con la que se ejecutan las sustituciones dentro de un ciclo AMG hace que el tiempo asociado a las sustituciones represente una proporción significativa del costo total de ejecución. Gracias a esto, la versión $SFMILU_{1D}$ consigue mejorar el tiempo de cada iteración y por tanto el tiempo de resolución global en todas las matrices del conjunto, logrando *speedups* desde $1,2\times$ hasta $5,0\times$.

Capítulo 6

Conclusiones y trabajo futuro

Este capítulo final sintetiza el trabajo realizado en esta tesis, enfatizando los resultados obtenidos y los principales aportes al estado del arte. Se enumeran los artículos científicos derivados de este trabajo, validando su rigor y alcance académico. Finalmente, se proponen líneas de investigación futura, orientadas a abordar limitaciones identificadas y a explorar aplicaciones o extensiones de los resultados obtenidos.

6.1. Conclusiones

La solución computacional de muchos problemas, como los EDPs que surgen en diversas áreas (en particular en CFD), presenta desafíos al momento de aprovechar el poder de cómputo de una GPU.

Uno de los principales es manejar las dependencias entre etapas del cómputo, de forma que maximice el paralelismo y minimice los tiempos ociosos de la GPU. En esta tesis se proponen optimizaciones a operaciones mediante hardware paralelo, basándose fuertemente en estrategias *synchronization-free* para resolver dependencias, en niveles jerárquicos de memoria para realizar de forma eficiente los accesos a memoria y en una correcta administración de recursos de hardware para abordar los problemas.

Por un lado, se estudiaron diferentes estrategias de planificación basadas en niveles para el contexto de un solver de volúmenes finitos. Este estudio desembocó en la implementación de una nueva propuesta de planificación basada en elementos. La evaluación experimental de la propuesta en mallas de tres dimensiones de tamaño variable demuestra que el nuevo mecanismo de plani-

ficación puede provocar aceleraciones significativas. Aunque la ventaja varía con el tamaño de la malla —lo cual, a su vez, se relaciona con el número de niveles—, el solver SIP basado en elementos alcanza hasta un 23 % más de rendimiento que la implementación basada en niveles en casos reales, obteniendo aceleraciones de hasta 2× en *kernels* individuales.

Por otro lado, se aborda el método de multigrillas algebraicas, tomando como caso de estudio la optimización de las rutinas ofrecidas por la biblioteca AmgX de NVIDIA, con particular foco en el smoother DILU Multicolor en GPU. Con el objetivo de reducir las esperas innecesarias asociadas al manejo de dependencias entre filas y aprovechar al máximo el paralelismo disponible en las plataformas de cómputo, se desarrollaron y evaluaron dos propuestas basadas en la técnica *synchronization-free*.

La primera de las propuestas, denominada $SFMILU_{2D}$, se apoya en un análisis previo liviano para definir la grilla de ejecución, buscando maximizar el uso de los recursos brindados por las plataformas masivamente paralelas. La segunda $SFMILU_{1D}$, se basa fuertemente en una etapa de análisis simbólico más detallado sobre las características de la matriz, que permite una distribución más granular y equilibrada de la carga de trabajo entre las unidades de procesamiento para el posterior cómputo.

Las implementaciones fueron ejecutadas y comparadas tanto de forma aislada como en el contexto de la biblioteca AmgX.

En primera instancia, sobre un conjunto extenso de aproximadamente 1300 matrices dispersas, los resultados muestran mejoras de media de $1,8\times$ para $SFMILU_{2D}$ y $3,7\times$ para $SFMILU_{1D}$, evidenciando la eficiencia de ambas versiones por sobre la original de NVIDIA en un escenario general.

En segunda instancia, se considera la integración de las implementaciones propuestas en los solvers ofrecidos por la biblioteca, donde se obtienen mejoras consistentes en las dos etapas principales del smoother, mostrando para $SFMILU_{2D}$ una media de $1,6\times$ y $2,0\times$ para la sustitución hacia adelante y hacia atrás respectivamente, mientras que para $SFMILU_{1D}$ $3,1\times$ y $3,5\times$ sobre problemas provenientes de dinámica de fluidos. Estos resultados reducen significativamente el tiempo de ejecución del smoother, impactando directamente en el desempeño de cada iteración de AMG y, por tanto, en el tiempo global de resolución, alcanzando una aceleración media sobre $MILU_{BASE}$ de 20 % para $SFMILU_{2D}$ y 60 % en $SFMILU_{1D}$, llegando, en este último caso, a obtener mejoras de hasta un 80 %.

Estos resultados permiten concluir que un manejo adecuado de las dependencias y una distribución equilibrada de la carga de trabajo entre los recursos de la plataforma de ejecución resultan factores clave para alcanzar mejoras significativas en eficiencia.

6.2. Trabajos publicados

A continuación se presentan las publicaciones realizadas en el marco de esta tesis, donde Franco Seveso es el autor principal.

- Franco Seveso, Ernesto Dufrechou, Pablo Ezzatti, and Gabriel Usera. *Element scheduling for GPU-accelerated finite-volumes computations*. In Caino-Lores., et al. Euro-Par 2024: Parallel Processing Workshops, pages 438–449, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-90200-0.
- Franco Seveso, Ernesto Dufrechou, Pablo Ezzatti.
  "GPU Synchronization-free multicolor DILU for NVIDIA AmgX library"
  en elaboración, se prevé su envío para evaluación en breve.

6.3. Trabajo futuro

Partiendo de los avances alcanzados, se proponen las siguientes líneas de trabajo:

- Explorar la aplicación de técnicas de planificación basadas en elementos a otros tipos de códigos iterativos tipo *stencil* en GPU, gestionando las dependencias entre pasos temporales.
- Adaptar las nuevas implementaciones a arquitecturas GPU más recientes, explorando especialmente las capacidades de sincronización introducidas en la generación Ampere, como los grafos de *kernels*, la sincronización global de grillas, los grupos cooperativos y las nuevas barreras de sincronización.
- Ahondar en una implementación optimizada del análisis para $SFMILU_{1D}$, por ejemplo, tomando como referencia las estrategias implementadas en [136], permitiendo alcanzar mayores reducciones en el tiempo de las dos etapas de AMG y por lo tanto el tiempo total de ejecución.

- Analizar el comportamiento de las implementaciones en diferentes contextos, con diversas configuraciones de AMG, en arquitecturas GPUs de última generación o en entornos multi-GPU, con el objetivo de evaluar la escalabilidad y robustez de las técnicas presentadas.
- Aplicar las estrategias propuestas en otros *smoothers* utilizados en AMG, tales como ILU o Gauss–Seidel, con el fin de maximizar la utilidad y extender la aplicabilidad de estas estrategias sobre diferentes áreas de problemas computacionales.

Estas direcciones no solo ayudarán a consolidar las contribuciones actuales, sino que también podrían abrir caminos para aplicar estas ideas en nuevos desafíos.

Referencias bibliográficas

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

[2] U. Trottenberg, C.W. Oosterlee, and A. Schuller. *Multigrid*. Academic Press, 2000. ISBN 9780080479569. URL https://books.google.com.uy/books?id=9ysyNPZoR24C.

[3] William L Briggs, Van Emden Henson, and Steve F McCormick. *A multigrid tutorial*. SIAM, 2000.

[4] José Ignacio Aliaga, Ernesto Dufrechou, Pablo Ezzatti, and Enrique S. Quintana-Ortí. Accelerating the task/data-parallel version of ilupack's bicg in multi-cpu/gpu configurations. *Parallel Comput.*, 85:79–87, 2019. doi: 10.1016/j.parco.2019.02.005. URL https://doi.org/10.1016/j.parco.2019.02.005.

[5] José Ignacio Aliaga, Ernesto Dufrechou, Pablo Ezzatti, and Enrique S. Quintana-Ortí. An efficient GPU version of the preconditioned GMRES method. *J. Supercomput.*, 75(3):1455–1469, 2019. doi: 10.1007/s11227-018-2658-1. URL https://doi.org/10.1007/s11227-018-2658-1.

[6] 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.

[7] Herbert L. Stone. Iterative solution of implicit approximations of multi-dimensional partial differential equations. *SIAM Journal on Numerical Analysis*, 5(3):530–558, 1968. ISSN 00361429. URL http://www.jstor.org/stable/2949703.

[8] Ernesto Dufrechou, Pablo Ezzatti, and Gabriel Usera. Avoiding synchronization to accelerate a cfd solver in gpu. In *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 204–211, 2019. doi: 10.1109/SBAC-PAD.2019.00041.

[9] D.W. Zingg, S. De Rango, M. Nemec, and T.H. Pulliam. Comparison of several spatial discretizations for the navier–stokes equations. *Journal of Computational Physics*, 160(2):683–704, 2000. ISSN 0021-9991. doi: https://doi.org/10.1006/jcph.2000.6482. URL https://www.sciencedirect.com/science/article/pii/S0021999100964829.

[10] Youcef Saad. Krylov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1232, 1989. doi: 10.1137/0910073. URL https://doi.org/10.1137/0910073.

[11] B. Suchoski, C. Severn, M. Shantharam, and P. Raghavan. Adapting sparse triangular solution to gpus. In *2012 41st International Conference on Parallel Processing Workshops*, pages 140–148, Sept 2012. doi: 10.1109/ICPPW.2012.23.

[12] Mark Adams, Marian Brezina, Jonathan Hu, and Ray Tuminaro. Parallel multigrid smoothing: polynomial versus gauss–seidel. *Journal of Computational Physics*, 188(2):593–610, 2003. ISSN 0021-9991. doi: https://doi.org/10.1016/S0021-9991(03)00194-3. URL https://www.sciencedirect.com/science/article/pii/S0021999103001943.

[13] Hongyu Liu, Xing Ji, Yuan Fu, and Kun Xu. A geometric multigrid-accelerated compact gas-kinetic scheme for fast convergence in high-speed flows on gpus, 2025. URL https://arxiv.org/abs/2509.06347.

[14] Jiale Meng, Shuqi Tang, Steven M. Wise, and Zhenlin Guo. A gpu-accelerated matrix-free fas multigrid solver for navier-stokes equations with memory-efficient implementations, 2025. URL https://arxiv.org/abs/2510.11152.

[15] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S Duff, and Brian Vinter. A synchronization-free algorithm for parallel sparse triangular solves. In *European Conference on Parallel Processing*, pages 617–630. Springer, 2016.

[16] Walter A Strauss. *Partial differential equations: An introduction*. John Wiley & Sons, 2007.

[17] Xiaolong Zeng, Li Zheng, and Yixuan Wu. Numerical solutions of PDEs for corporate bond pricing: A computational finance approach. *Applied and Computational Engineering*, 82(1):112–117, November 2024.

[18] Alessandro Contri, André Massing, and Padmini Rangamani. A finite element framework for bulk-surface coupled pdes to solve moving boundary problems in biophysics, 2025. URL https://arxiv.org/abs/2510.23459.

[19] SN Antontsev, JI Diaz, S Shmarev, and AJ Kassab. Energy methods for free boundary problems: applications to nonlinear pdes and fluid mechanics. progress in nonlinear differential equations and their applications, vol 48, 2002.

[20] Filipe De Avila Belbute-Peres, Thomas Economon, and Zico Kolter. Combining differentiable PDE solvers and graph neural networks for fluid flow prediction. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 2402–2411. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/de-avila-belbute-peres20a.html.

[21] Juan Luis Vázquez. La ecuación de navier-stokes. un reto fisicomatemático para el siglo xxi. *Departamento de Matemáticas. Univ. Autónoma de Madrid, Real Academia de Ciencias de Zaragoza*, 26:31–56, 2004.

[22] Roger Temam. *Navier–Stokes equations: theory and numerical analysis*, volume 343. American Mathematical Society, 2024.

[23] A R El-metwaly and M A Kamal. A brief review of numerical methods for solving the boundary value problems of pde. *Journal of Physics: Conference Series*, 2847(1):012001, sep 2024. doi: 10.1088/1742-6596/2847/1/012001. URL https://doi.org/10.1088/1742-6596/2847/1/012001.

[24] Steven C Chapra, Raymond P Canale, et al. *Numerical methods for engineers*, volume 1221. Mcgraw-hill New York, 2011.

[25] Robert Eymard, Thierry Gallouët, and Raphaèle Herbin. Finite volume methods. *Handbook of numerical analysis*, 7:713–1018, 2000.

[26] Claes Johnson. *Numerical solution of partial differential equations by the finite element method*. Courier Corporation, 2009.

[27] David A. Kopriva, Stephen L. Woodruff, and M. Y. Hussaini. Discontinuous spectral element approximation of maxwell's equations. In Bernardo Cockburn, George E. Karniadakis, and Chi-Wang Shu, editors, *Discontinuous Galerkin Methods*, pages 355–361, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-642-59721-3.

[28] Alfio Quarteroni and Alberto Valli. *Numerical approximation of partial differential equations*. Springer, 1994.

[29] Alexander H-D Cheng and Daisy T Cheng. Heritage and early history of the boundary element method. *Engineering analysis with boundary elements*, 29(3):268–302, 2005.

[30] Randall J LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. SIAM, 2007.

[31] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 2nd edition, 2003. ISBN 0898715342.

[32] Xiaodong Yang and Huiyang Ma. Cubic eddy-viscosity turbulence models for strongly swirling confined flows with variable density. *International Journal for Numerical Methods in Fluids*, 45(9):985–1008, 2004. doi: https://doi.org/10.1002/fld.735. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.735.

[33] Joel H Ferziger, Milovan Perić, and Robert L Street. *Computational methods for fluid dynamics*. springer, 2019.

[34] Željko Lilek, Samir Muzaferija, Milovan Perić, and Volker Seidl. An implicit finite-volume method using nonmatching blocks of structured

grid. *Numerical Heat Transfer, Part B: Fundamentals*, 32(4):385–401, 1997. doi: 10.1080/10407799708915015. URL https://doi.org/10.1080/10407799708915015.

[35] S.V Patankar and D.B Spalding. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *International Journal of Heat and Mass Transfer*, 15(10):1787–1806, 1972. ISSN 0017-9310. doi: https://doi.org/10.1016/0017-9310(72)90054-3. URL https://www.sciencedirect.com/science/article/pii/0017931072900543.

[36] Mariana Mendina, Martin Draper, Ana Paula Kelm Soares, Gabriel Narancio, and Gabriel Usera. A general purpose parallel block structured open source incompressible flow solver. *Cluster Computing*, 17(2):231–241, Jun 2014. ISSN 1573-7543. doi: 10.1007/s10586-013-0323-2. URL https://doi.org/10.1007/s10586-013-0323-2.

[37] SB Beale. A finite volume method for numerical grid generation. *International journal for numerical methods in fluids*, 30(5):523–540, 1999.

[38] Charles S Peskin. The immersed boundary method. *Acta numerica*, 11: 479–517, 2002.

[39] Joel H Ferziger and MILOVAN PERIĆ. Further discussion of numerical errors in cfd. *International journal for numerical methods in fluids*, 23 (12):1263–1274, 1996.

[40] Lee Chen, Don Chen, Chang Li, Bing Pan, Lixuan Zhang, and Zheng Xiang. Scalability of algebraic multigrid in computer science. *American-Eurasian Journal of Scientific Research*, 11(2023):2998–3005, 2023.

[41] Klaus Stüben. A review of algebraic multigrid. *Numerical Analysis: Historical Developments in the 20th Century*, pages 331–359, 2001.

[42] Shenren Xu, Jiazi Zhao, Hangkong Wu, Sen Zhang, Jens-Dominik Müller, Huang Huang, Mohammad Rahmati, and Dingxi Wang. A review of solution stabilization techniques for rans cfd solvers. *Aerospace*, 10(3), 2023. ISSN 2226-4310. doi: 10.3390/aerospace10030230. URL https://www.mdpi.com/2226-4310/10/3/230.

[43] Stefan Reitzinger. *Algebraic multigrid methods for large scale finite element equations*. Trauner, 2001.

[44] Jiyuan Tu, Guan-Heng Yeoh, and Chaoqun Liu. Chapter 5 - cfd techniques: The basics. In Jiyuan Tu, Guan-Heng Yeoh, and Chaoqun Liu, editors, *Computational Fluid Dynamics (Third Edition)*, pages 155–210. Butterworth-Heinemann, third edition edition, 2018. ISBN 978-0-08-101127-0. doi: https://doi.org/10.1016/B978-0-08-101127-0.00005-2. URL https://www.sciencedirect.com/science/article/pii/B9780081011270000052.

[45] Maxim Naumov, Patrice Castonguay, and Jonathan M. Cohen. Parallel graph coloring with applications to the incomplete-lu factorization on the gpu. 2015. URL https://api.semanticscholar.org/CorpusID:15964368.

[46] Amanda Bienz, Robert D. Falgout, William Gropp, Luke N. Olson, and Jacob B. Schroder. Reducing parallel communication in algebraic multigrid through sparsification. *SIAM Journal on Scientific Computing*, 38(5):S332–S357, 2016. doi: 10.1137/15M1026341. URL https://doi.org/10.1137/15M1026341.

[47] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, 1994. doi: 10.1137/1.9781611971538. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611971538.

[48] Fan RK Chung. *Spectral graph theory*, volume 92. American Mathematical Soc., 1997.

[49] Stephen J Wright. *Primal-dual interior-point methods*. SIAM, 1997.

[50] Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 2006.

[51] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009. doi: 10.1109/MC.2009.263.

[52] NVPL Storage Formats. Sparse matrix formats. https://docs.nvidia.com/nvpl/latest/sparse/storage_format/sparse_matrix.html. Access date: 2025-07-10.

[53] James King, Thomas Gilray, Robert M Kirby, and Matthew Might. Dynamic sparse-matrix allocation on gpus. In *International Conference on High Performance Computing*, pages 61–80. Springer, 2016.

[54] Basel Bani-Ismail and Ghassan Kanaan. Comparing different sparse matrix storage structures as index structure for arabic text collection. *Int. J. Inf. Retr. Res.*, 2(2):52–67, April 2012. ISSN 2155-6377. doi: 10.4018/ijirr.2012040105. URL https://doi.org/10.4018/ijirr.2012040105.

[55] 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.

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

[57] 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.

[58] 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.

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

[60] Volta architecture whitepaper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Access date: 2025-07-10.

[61] Naoya Maruyama and Takayuki Aoki. Optimizing stencil computations for nvidia kepler gpus. 2014. URL https://api.semanticscholar.org/CorpusID:17605545.

[62] NVIDIA. Kepler tm gk110 the fastest, most efficient hpc architecture ever built, 2012. [Online; accessed 10-June-2017].

[63] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan. On optimizing complex stencils on gpus. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 641–652, 2019. doi: 10.1109/IPDPS.2019.00073.

[64] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. Register optimizations for stencils on gpus. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 168–182, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450349826. doi: 10.1145/3178487.3178500. URL https://doi.org/10.1145/3178487.3178500.

[65] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Mahesh Ravishankar, Vinod Grover, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan. Domain-specific optimization and generation of high-performance gpu code for stencil computations. *Proceedings of the IEEE*, 106(11):1902–1920, 2018. doi: 10.1109/JPROC.2018.2862896.

[66] Kazuaki Matsumura, Hamid Reza Zohouri, Mohamed Wahib, Toshio Endo, and Satoshi Matsuoka. An5d: automated stencil framework for high-degree temporal blocking on gpus. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 199–211, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370479. doi: 10.1145/3368826.3377904. URL https://doi.org/10.1145/3368826.3377904.

[67] Manuel de Castro, Inmaculada Santamaria-Valenzuela, Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. Epsilod: efficient parallel skeleton for generic iterative stencil computations in distributed gpus. *J. Supercomput.*, 79(9):9409–9442, jan 2023. ISSN 0920-8542. doi:

10.1007/s11227-022-05040-y. URL https://doi.org/10.1007/s11227-022-05040-y.

[68] Ryuichi Sai, John Mellor-Crummey, Xiaozhu Meng, Mauricio Araya-Polo, and Jie Meng. Using the semi-stencil algorithm to accelerate high-order stencils on gpus. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 63–68, 2021. doi: 10.1109/PMBS54543.2021.00012.

[69] Johannes de Fine Licht, Andreas Kuster, Tiziano De Matteis, Tal Ben-Nun, Dominic Hofer, and Torsten Hoefler. Stencilflow: Mapping large stencil programs to distributed spatial computing systems. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 315–326, 2021. doi: 10.1109/CGO51591.2021.9370315.

[70] Lingqi Zhang, Mohamed Wahib, Haoyu Zhang, and Satoshi Matsuoka. A study of single and multi-device synchronization methods in nvidia gpus. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 483–493, 2020. doi: 10.1109/IPDPS47924.2020.00057.

[71] Lingqi Zhang, Mohamed Wahib, Peng Chen, Jintao Meng, Xiao Wang, Toshio Endo, and Satoshi Matsuoka. Revisiting temporal blocking stencil optimizations. In *Proceedings of the 37th International Conference on Supercomputing*, ICS '23, page 251–263, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700569. doi: 10.1145/3577193.3593716. URL https://doi.org/10.1145/3577193.3593716.

[72] EDWARD ANDERSON and YOUCEF SAAD. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing*, 01(01):73–95, 1989. doi: 10.1142/S0129053389000056. URL https://doi.org/10.1142/S0129053389000056.

[73] Ernesto Dufrechu and Pablo Ezzatti. A new gpu algorithm to compute a level set-based analysis for the parallel solution of sparse triangular systems. *2018 IEEE International Parallel and Distributed Processing*

Symposium (IPDPS), pages 920–929, 2018. URL https://api.semanticscholar.org/CorpusID:51923669.

[74] Zhengyang Lu and Weifeng Liu. Tilesptrsv: a tiled algorithm for parallel sparse triangular solve on gpus. *CCF Transactions on High Performance Computing*, 5:129 – 143, 2023. URL https://api.semanticscholar.org/CorpusID:259246949.

[75] Manuel Freire, Ernesto Dufrechou, and Pablo Ezzatti. A new level-set analysis and sparse storage format for the sptrsv in gpus. In *2024 IEEE 36th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 59–69. IEEE, 2024.

[76] Juris Hartmanis. Computers and intractability: A guide to the theory of np-completeness (michael r. garey and david s. johnson). *SIAM Review*, 24(1):90–91, 1982. doi: 10.1137/1024022. URL https://doi.org/10.1137/1024022.

[77] Thomas F. Coleman and Jorge J. Moré. Estimation of sparse jacobian matrices and graph coloring blems. *SIAM Journal on Numerical Analysis*, 20(1):187–209, 1983. doi: 10.1137/0720013. URL https://doi.org/10.1137/0720013.

[78] Assefaw Hadish Gebremedhin and Fredrik Manne. Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience*, 12 (12):1131–1146, 2000. doi: https://doi.org/10.1002/1096-9128(200010)12:12<1131::AID-CPE528>3.0.CO;2-2. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/1096-9128%28200010%2912%3A12%3C1131%3A%3AAID-CPE528%3E3.0.CO%3B2-2.

[79] Ümit V. Çatalyürek, John Feo, Assefaw H. Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. Graph coloring algorithms for multicore and massively multithreaded architectures. *Parallel Computing*, 38 (10):576–594, 2012. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2012.07.001. URL https://www.sciencedirect.com/science/article/pii/S0167819112000592.

[80] Georgios Rokos, Gerard Gorman, and Paul H.J. Kelly. A fast and scalable graph coloring algorithm for multi-core and many-core architectures. In

Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015: Parallel Processing*, pages 414–425, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-48096-0.

[81] M Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, page 1–10, New York, NY, USA, 1985. Association for Computing Machinery. ISBN 0897911512. doi: 10.1145/22145.22146. URL https://doi.org/10.1145/22145.22146.

[82] Stephen A Cook. A taxonomy of problems with fast parallel algorithms. *Information and control*, 64(1-3):2–22, 1985.

[83] Thomas F Coleman and Jorge J Moré. Estimation of sparse jacobian matrices and graph coloring blems. *SIAM journal on Numerical Analysis*, 20(1):187–209, 1983.

[84] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. Ordering heuristics for parallel graph coloring. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, page 166–177, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328210. doi: 10.1145/2612669.2612697. URL https://doi.org/10.1145/2612669.2612697.

[85] Dániel Marx. Graph colouring problems and their applications in scheduling. *Periodica Polytechnica Electrical Engineering (Archives)*, 48(1-2): 11–16, 2004.

[86] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, 1993. doi: 10.1137/0914041. URL https://doi.org/10.1137/0914041.

[87] Christina Giannoula, Athanasios Peppas, Georgios I. Goumas, and Nectarios Koziris. High-performance and balanced parallel graph coloring on multicore platforms. *The Journal of Supercomputing*, 79:6373–6421, 2022. URL https://api.semanticscholar.org/CorpusID:253416226.

[88] Assia Brighen and Hachem Slimani. Parallel algorithm for coloring large-scale graphs using pregel api of graphx. In Noureddine Seddari and

Mohammed Redjimi, editors, *Modeling, Simulation and Computer Technology*, pages 465–479, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-032-01922-6.

[89] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 599–613, USA, 2014. USENIX Association. ISBN 9781931971164.

[90] Jonathan Cohen and Patrice Castonguay. Efficient graph matching and coloring on the gpu. In *GPU Technology Conference*, pages 1–10, 2012.

[91] Zheming Jin and Jeffrey S. Vetter. Experience deploying graph applications on gpus with sycl. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops*, ICPP Workshops '23, page 30–39, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400708428. doi: 10.1145/3605731.3605744. URL https://doi.org/10.1145/3605731.3605744.

[92] Ronan Keryell, Ruyman Reyes, and Lee Howes. Khronos sycl for opencl: a tutorial. In *Proceedings of the 3rd International Workshop on OpenCL*, pages 1–1, 2015.

[93] Maciej Besta, Armon Carigiet, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, and Torsten Hoefler. High-performance parallel graph coloring with strong guarantees on work, depth, and quality. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17, 2020. doi: 10.1109/SC41405.2020.00103.

[94] Liu Yang and Jian Yang. A multi-colored gauss-seidel solver for aerodynamic simulations of a transport aircraft model on graphics processing units. *Advances in Aerodynamics*, 7, 06 2025. doi: 10.1186/s42774-024-00200-5.

[95] Pablo Igounet, Pablo Alfaro, Martín Pedemonte, and Pablo Ezzatti. A gpu implementation of the sip method. In *2011 30th International Conference of the Chilean Computer Science Society*, pages 195–201, 2011. doi: 10.1109/SCCC.2011.26.

[96] Frank Deserno, G Hager, F Brechtefeld, and G Wellein. Basic optimization strategies for cfd-codes. *Regionales Rechenzentrum Erlangen, Technical report*, 2002.

[97] Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, page 193–es, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 9781450378338. doi: 10.1145/1198555.1198784. URL https://doi.org/10.1145/1198555.1198784.

[98] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007. doi: https://doi.org/10.1111/j.1467-8659.2007.01012.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2007.01012.x.

[99] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka. Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 37(5):S602–S626, 2015. doi: 10.1137/140980260. URL https://doi.org/10.1137/140980260.

[100] Ashwin Srinath. pyamgx: Python interface to nvidia's amgx library, 2018. URL https://github.com/shwina/pyamgx. [Online; accessed 11-Nov-2025].

[101] Pi-Yueh Chuang and Lorena A. Barba. AmgXWrapper: An interface between PETSc and the NVIDIA AmgX library. *The Journal of Open Source Software*, 2(16), aug 2017. doi: 10.21105/joss.00280. URL https://doi.org/10.21105/joss.00280.

[102] Yizhuo Wang, Fangli Chang, Bingxin Wei, Jianhua Gao, and Weixing Ji. Optimization of sparse matrix computation for algebraic multigrid on gpus. *ACM Trans. Archit. Code Optim.*, 21(3), September 2024. ISSN 1544-3566. doi: 10.1145/3664924. URL https://doi.org/10.1145/3664924.

[103] Stefano Oliani, Ettore Fadiga, Ivan Spisso, Luigi Capone, and Federico Piscaglia. Gpu-accelerated linear algebra for coupled solvers in industrial cfd applications with openfoam, 2024. URL https://arxiv.org/abs/2403.07882.

[104] Matt Martineau, Stan Posey, and Filippo Spiga. Amgx gpu solver developments for openfoam. In *Proceedings of the 8th OpenFOAM Conference, Virtual*, pages 13–15, 2020.

[105] FLORENTIN TRISTAN EMILIEN AYRAULT. Evaluation of a novel fast matrix conversion algorithm for accelerated cfd simulations using nvidia amgx in openfoam. 2023.

[106] Yue Zhu, Jin Gan, Yongshui Lin, and Weiguo Wu. Graphics processing unit-accelerated propeller computational fluid dynamics using amgx: Performance analysis across mesh types and hardware configurations. *Journal of Marine Science and Engineering*, 12(12), 2024. ISSN 2077-1312. doi: 10.3390/jmse12122134. URL https://www.mdpi.com/2077-1312/12/12/2134.

[107] Robert D. Falgout and Ulrike Meier Yang. hypre: A library of high performance preconditioners. In Peter M. A. Sloot, Alfons G. Hoekstra, C. J. Kenneth Tan, and Jack J. Dongarra, editors, *Computational Science — ICCS 2002*, pages 632–641, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-47789-1.

[108] Van Emden Henson and Ulrike Meier Yang. Boomeramg: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155–177, 2002. ISSN 0168-9274. doi: https://doi.org/10.1016/S0168-9274(01)00115-5. URL https://www.sciencedirect.com/science/article/pii/S0168927401001155. Developments and Trends in Iterative Methods for Large Systems of Equations - in memorium Rudiger Weiss.

[109] Allison H. Baker, Robert D. Falgout, Tzanio V. Kolev, and Ulrike Meier Yang. Multigrid smoothers for ultraparallel computing. *SIAM Journal on Scientific Computing*, 33(5):2864–2887, 2011. doi: 10.1137/100798806. URL https://doi.org/10.1137/100798806.

[110] Yuechen Lu, Lijie Zeng, Tengcheng Wang, Xu Fu, Wenxuan Li, Helin Cheng, Dechuang Yang, Zhou Jin, Marc Casas, and Weifeng Liu. Amgt: Algebraic multigrid solver on tensor cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24. IEEE Press, 2024. ISBN 9798350352917. doi: 10.1109/SC41406.2024.00058. URL https://doi.org/10.1109/SC41406.2024.00058.

[111] Pasqua D'Ambra, Fabio Durastante, and Salvatore Filippone. Amg preconditioners for linear solvers towards extreme scale. *SIAM Journal on Scientific Computing*, 43(5):S679–S703, 2021. doi: 10.1137/20M134914X. URL https://doi.org/10.1137/20M134914X.

[112] Salvatore Filippone and Michele Colajanni. Psblas: a library for parallel linear algebra computation on sparse matrices. *ACM Trans. Math. Softw.*, 26(4):527–550, December 2000. ISSN 0098-3500. doi: 10.1145/365723.365732. URL https://doi.org/10.1145/365723.365732.

[113] Pasqua D'Ambra, Fabio Durastante, Salvatore Filippone, Stefano Massei, and Stephen Thomas. Optimal polynomial smoothers for parallel amg. *Numerical Algorithms*, pages 1–30, 06 2025. doi: 10.1007/s11075-025-02117-6.

[114] D. Demidov. Amgcl: An efficient, flexible, and extensible algebraic multigrid implementation. *Lobachevskii Journal of Mathematics*, 40(5):535–546, May 2019. ISSN 1818-9962. doi: 10.1134/S1995080219050056. URL https://doi.org/10.1134/S1995080219050056.

[115] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[116] Dinesh Parthasarathy, Tommaso Bevilacqua, Martin Lanser, Axel Klawonn, and Harald Köstler. Towards automated algebraic multigrid preconditioner design using genetic programming for large-scale laser beam welding simulations. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '25, page 1–11, New York, NY, USA,

2025. Association for Computing Machinery. ISBN 9798400718861. doi: 10.1145/3732775.3733589. URL https://doi.org/10.1145/3732775.3733589.

[117] M.W. Gee, C.M. Siefert, J.J. Hu, R.S. Tuminaro, and M.G. Sala. Ml 5.0 smoothed aggregation user's guide. Technical Report SAND2006-2649, Sandia National Laboratories, 2006.

[118] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014. URL http://cusplibrary.github.io/. Version 0.5.0.

[119] Masatoshi Kawai, Akihiro Ida, Hiroya Matsuba, Kengo Nakajima, and Matthias Bolten. Multiplicative schwartz-type block multi-color gauss-seidel smoother for algebraic multigrid methods. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia '20, page 217–226, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450372367. doi: 10.1145/3368474.3368481. URL https://doi.org/10.1145/3368474.3368481.

[120] Ru Huang, Ruipeng Li, and Yuanzhe Xi. Learning optimal multigrid smoothers via neural networks. *SIAM Journal on Scientific Computing*, 45(3):S199–S225, 2023. doi: 10.1137/21M1430030. URL https://doi.org/10.1137/21M1430030.

[121] Hartwig Anzt, Edmond Chow, and Jack Dongarra. Iterative sparse triangular solves for preconditioning. In Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015: Parallel Processing*, pages 650–661, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-48096-0.

[122] Frédéric Gibou and Chohong Min. On the performance of a simple parallel implementation of the ilu-pcg for the poisson equation on irregular domains. *Journal of Computational Physics*, 231(14):4531–4536, 2012. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2012.02.023. URL https://www.sciencedirect.com/science/article/pii/S0021999112001325.

[123] Edmond Chow and Aftab Patel. Fine-grained parallel incomplete lu factorization. *SIAM Journal on Scientific Computing*, 37(2):C169–C193,

2015. doi: 10.1137/140968896. URL https://doi.org/10.1137/140968896.

[124] Hartwig Anzt, Edmond Chow, and Jack Dongarra. Parilut—a new parallel threshold ilu factorization. *SIAM Journal on Scientific Computing*, 40(4):C503–C519, 2018. doi: 10.1137/16M1079506. URL https://doi.org/10.1137/16M1079506.

[125] David Hysom and Alex Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM Journal on Scientific Computing*, 22(6):2194–2215, 2001. doi: 10.1137/S1064827500376193. URL https://doi.org/10.1137/S1064827500376193.

[126] Sangback Ma and Y Saad. *Distributed ILU (0) and SOR preconditioners for unstructured sparse linear systems*. Citeseer, 1994.

[127] Tianshi Xu, Rui Peng Li, and Daniel Osei-Kuffuor. A two-level gpu-accelerated incomplete lu preconditioner for general sparse linear systems. *Int. J. High Perform. Comput. Appl.*, 39(3):424–442, April 2025. ISSN 1094-3420. doi: 10.1177/10943420251319334. URL https://doi.org/10.1177/10943420251319334.

[128] Manuel Freire, Ernesto Dufrechou, and Pablo Ezzatti. A synchronization-free incomplete lu factorization for gpus with level-set analysis. In *2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 217–225, 2025. doi: 10.1109/PDP66500.2025.00037.

[129] Hao Luo, Qianchao Zhu, Xiaochen Hao, Chunxi Lei, Chengdi Ma, Chenchen Zhang, Yun Liang, and Chao Yang. Structilu: Dependency-preserving incomplete lu with hierarchical parallelism for structured grid pdes on gpus. In *Proceedings of the 39th ACM International Conference on Supercomputing*, ICS '25, page 119–134, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715372. doi: 10.1145/3721145.3725745. URL https://doi.org/10.1145/3721145.3725745.

[130] G. Usera, A. Vernet, and J.A. Ferré. A parallel block-structured finite volume method for flows in complex geometry with sliding interfaces.

Flow, Turbulence and Combustion, 81(3):471–495, 2008. doi: 10.1007/s10494-008-9153-3.

[131] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011. ISSN 0098-3500. doi: 10.1145/2049662.2049663. URL https://doi.org/10.1145/2049662.2049663.

[132] NVIDIA Corporation. NVIDIA cuSPARSE Library. https://developer.nvidia.com/cusparse, 2024.

[133] J. Allwright, Rajesh Bordawekar, P. Coddington, Kıvanç Dinçer, and C. Martin. A comparison of parallel graph coloring algorithms. 03 1995.

[134] Xuhao Chen, Pingfan Li, Jianbin Fang, Tao Tang, Zhiying Wang, and Canqun Yang. Efficient and high-quality sparse graph coloring on gpus. *Concurrency and Computation: Practice and Experience*, 29(10):e4064, 2017. doi: https://doi.org/10.1002/cpe.4064. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4064. e4064 cpe.4064.

[135] Amanda Bienz, Robert D. Falgout, William Gropp, Luke N. Olson, and Jacob B. Schroder. Reducing parallel communication in algebraic multigrid through sparsification. *SIAM Journal on Scientific Computing*, 38(5):S332–S357, 2016. doi: 10.1137/15M1026341. URL https://doi.org/10.1137/15M1026341.

[136] Manuel Freire, Juan Ferrand, Franco Seveso, Ernesto Dufrechou, and Pablo Ezzatti. A gpu method for the analysis stage of the sptrsv kernel. *J. Supercomput.*, 79(13):15051–15078, April 2023. ISSN 0920-8542. doi: 10.1007/s11227-023-05238-8. URL https://doi.org/10.1007/s11227-023-05238-8.