Universidad de la República
pedeciba informática

Aceleración de Métodos para la Resolución de Problemas de Reducción de Modelos mediante Procesadores Gráficos

Montevideo, Junio de 2011

Tesis Doctoral Presentada por: Pablo Ezzatti
Director Académico: Héctor Cancela
Director de Tesis: Enrique S. Quintana Ortí
Co-director de Tesis: Alfredo Remón

Universidad de la República
pedeciba informática

Aceleración de Métodos para la Resolución de Problemas de Reducción de Modelos mediante Procesadores Gráficos

Pablo Ezzatti

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Introducción

1.1.1. Motivación y objetivos

Diversos fenómenos físicos se pueden modelar mediante ecuaciones diferenciales. Estos modelos matemáticos se utilizan por ejemplo en el diseño de dispositivos electrónicos, donde es necesario especificar la posición y conexiones entre los circuitos. El alto costo temporal y económico hace inviable afrontar la fabricación y evaluación empírica de cada posible diseño de estos dispositivos. Una opción más realista consiste en especificar un modelo matemático que describa esta realidad (cómo afecta al funcionamiento y costo de producción del dispositivo la posición de cada uno de los circuitos y sus conexiones) y evaluar, utilizando el modelo matemático, todos los diseños posibles.

Generalmente, la simulación de los modelos matemáticos derivados de aplicar técnicas de control requiere un número elevado de cálculos. Este número depende de la dimensión del modelo, o lo que es lo mismo, del número de variables involucradas en el modelado. Además, la cantidad de operaciones necesarias para simular un modelo crece rápidamente con la dimensión del modelo, de forma que, aún en casos en los que se opera con modelos de dimensiones modestas, se require de la ayuda de computadoras para su resolución. El desarrollo vertiginoso de la computación en los últimos años, con la propuesta de nuevos algoritmos y técnicas de programación, y sobre todo, la aparición de nuevas plataformas hardware de gran desempeño, han posibilitado resolver cada vez problemas más complejos, de mayor dimensión y con mayor precisión. A pesar de ello, la necesidad de resolver problemas de mayor dimensión supera con creces las posibilidades de las computadoras actuales. Utilizando el caso antes mencionado de los dispositivos electrónicos, se puede tomar como ejemplo los teléfonos móviles, donde en un espacio físico cada día más reducido se integra una cantidad mayor de funcionalidades (teléfono propiamente dicho, cámara de foto, GPS, etc.) y cuyas componentes deben respetar diversas restricciones y cualidades deseables (consumo y trasmisión de energía, campo magnético, interferencias, etc.).

El interés creciente por contar con modelos matemáticos que permitan realizar simulaciones, evaluar posibles diseños, estudiar impactos, etc. en distintos campos de ingeniería, pero a su vez la necesidad de que estos modelos sean tratables en un tiempo aceptable, dan origen al campo de trabajo de la reducción de modelos. Estas técnicas buscan, dado un modelo matemático, encontrar otro cuya dimensión sea considerablemente menor pero que presente un comportamiento similar al del modelo original. De esta forma, es posible utilizar el modelo reducido en posteriores simulaciones o estudios, disminuyendo así las necesidades de cómputo y tiempo de ejecución.

De manera formal, y utilizando la formulación clásica de un sistema dinámico lineal (SDL) continuo e invariante en el tiempo, mediante el modelo de espacio de estados [185], un SDL queda definido por dos ecuaciones matriciales de la siguiente forma:

$$\begin{array}{rcll} \dot{x}(t) & = & Ax(t) + Bu(t), & t > 0, \quad x(0) = x_0, \\ y(t) & = & Cx(t) + Du(t), & t \geq 0, \end{array} \tag{1.1}$$

donde $x(t) \in \mathbb{R}^n$ es el vector de variables de estado, $x(0) = x_0$ es el estado inicial del sistema, $u(t) \in \mathbb{R}^m$ es el vector de entradas o controles, $y(t) \in \mathbb{R}^p$ es el vector de salidas, $A \in \mathbb{R}^{n \times n}$ se conoce como la matriz de estados, $B \in \mathbb{R}^{n \times m}$ es la matriz de entradas, $C \in \mathbb{R}^{p \times n}$ es la matriz de salidas, y $D \in \mathbb{R}^{p \times m}$. Además, n especifica el orden (o dimensión del espacio de estados) del sistema.

El modelo de representación en el espacio de estados ha permitido la aplicación de numerosos resultados provenientes del álgebra lineal numérica (ALN) al ámbito de la teoría de control [137, 185]. Así, los nuevos métodos desarrollados para este modelo permiten abordar, de manera numéricamente estable, problemas de grandes dimensiones que serían difícilmente tratables usando otro tipo de técnicas [150, 185].

Considerando el SDL de orden n presentado en (1.1) los métodos de reducción de modelos buscan obtener un segundo SDL,

$$\begin{array}{rcll} \dot{x}_r(t) & = & A_r x_r(t) + B_r u(t), & t > 0, \quad x_r(0) = \bar{x}_0, \\ y_r(t) & = & C_r x_r(t) + D_r u(t), & t \geq 0, \end{array} \tag{1.2}$$

de orden r mucho menor que n, que mantenga la información fundamental sobre la dinámica del sistema original. Bajo estas premisas el nuevo SDL en (1.2) es una aproximación del SDL original en (1.1), lo que permite reemplazarlo en sucesivos análisis o simulaciones.

Existen dos grandes familias de métodos para la resolución del problema de reducción de modelos. Los métodos basados en la descomposición en valores singulares (SVD, del inglés *singular value decomposition*) y los basados en subespacios de Krylov. Ambas familias de métodos de reducción de modelos pueden ser vistas como una función, que va del conjunto de modelos originales al de los modelos reducidos. Ello implica que estos métodos deben operar sobre el sistema original, y dada su dimensión, en muchos casos su costo computacional es excesivo para ser tratados con computadores personales. Dicha situación ha motivado la aplicación de técnicas de computación de alto desempeño (HPC, del inglés *high performance computing*) para poder abordar problemas de grandes dimensiones y acelerar las técnicas. En este trabajo, el foco se centra en la familia de métodos basados en la SVD, que destacan por presentar diversas cualidades matemáticas deseables (entre otras estabilidad y pasividad) y porque permiten en forma directa la aplicación de técnicas de HPC. De hecho, existen bibliotecas de rutinas para abordar la reducción de modelos que utilizan este enfoque numérico y que incluyen técnicas de HPC sobre multiprocesadores de memoria distribuida o clusters de computadores, como es el caso de PLiC [52].

El principal inconveniente que presentan las técnicas de HPC son los altos costos económicos asociados. A la complejidad de desarrollar los programas, se le añaden los elevados costos de la compra y administración de los equipos informáticos de altas prestaciones, los de acondicionamiento edilicio, consumo de energía, mantenimiento, etc.

En los últimos años en el campo de HPC han surgido nuevas alternativas hardware de bajo costo. Por un lado, las computadoras actuales integran varias unidades computacionales (4–8) en un sólo equipo en forma de procesadores multinúcleo. Por otro lado, los aceleradores hardware pueden funcionar como un dispositivo secundario y especializado hacia los que el procesador principal descargue núcleos computacionales intensivos. En particular, este trabajo se centra en el uso

de computadores multinúcleo combinados con procesadores gráficos (GPUs, del inglés *graphics processing unit*) como aceleradores hardware. Las principales características de las GPUs son su bajo costo económico y el alto ratio que vincula prestaciones con consumo de energía. Además, la constante presión de la industria de los videojuegos motiva a los fabricantes de tarjetas gráficas (principalmente ATI y NVIDIA) a liberar tarjetas cada vez más potentes y económicas. Esta situación propicia una evolución del ratio mencionado más beneficiosa si se compara con la ofrecida por las plataformas de HPC tradicionales.

En respuesta a las posibilidades que brindan las nuevas tecnologías de hardware de HPC, y teniendo en cuenta la importancia del área de trabajo en reducción de modelos para abordar la resolución de diversos problemas de la ingeniería, el objetivo principal de esta tesis es el diseño, desarrollo y evaluación de una biblioteca de rutinas para acelerar la resolución del problema de reducción de modelos, y en particular de la familia de métodos basados en la SVD, que aproveche la capacidad de cálculo ofrecida por arquitecturas de altas prestaciones compuestas por procesadores de propósito general multinúcleo y GPUs.

El camino hacia la concreción del objetivo principal de esta tesis plantea los siguientes objetivos específicos:

- Dada la importancia del cómputo de núcleos de ALN para la implementación de los métodos de reducción de modelos, relevar y evaluar la eficiencia de los métodos y bibliotecas disponibles para el cómputo de operaciones de ALN sobre GPUs.

- En base a las carencias detectadas en este relevamiento, diseñar, desarrollar y validar núcleos de ALN que se ejecuten eficientemente sobre plataformas híbridas compuestas por procesadores multinúcleo conectadas a GPUs, con particular interés en aquellas operaciones que sean de utilidad para la aceleración de métodos de reducción de modelos basados en la técnicas de la SVD.

- Estudiar, diseñar e implementar métodos para la resolución del problema de reducción de modelos basados en la técnica de la SVD que aprovechen los núcleos de ALN (tanto los disponibles originalmente como aquellos desarrollados en el marco de este trabajo), y que exploten las cualidades de las arquitecturas antes descritas, construidas a partir de procesadores de propósito general tipo multinúcleo conectados a GPUs.

- Validar las propuestas realizadas aplicando los métodos desarrollados a la resolución de problemas reales de reducción de modelos.

En línea con el objetivo principal de la tesis, y motivado por la necesidad de una respuesta eficiente en el marco de las futuras plataformas de HPC basadas en procesadores multinúcleo y GPUs, se plantea una lista complementaria de objetivos específicos para la tesis:

- Evaluar la posible extensión de los métodos desarrollados para resolver problemas de ALN y ecuaciones matriciales para abordar otros problemas de computación científica.

- Estudiar métodos que permitan aprovechar plataformas que utilicen múltiples GPUs al mismo tiempo.

1.1.2. Estructura de la Tesis

En lo que resta de este capítulo se revisa el estado actual de las arquitecturas y técnicas de programación de altas prestaciones; seguidamente se resume el estado del arte en el desarrollo de las bibliotecas de ALN que incluyen estrategias de HPC; en tercer lugar, se repasan brevemente los esfuerzos tendentes a la introducción de técnicas de HPC en los métodos de reducción de modelos. Posteriormente se releva el uso de las GPUs para la resolución de problemas de propósito general, con particular interés en la aceleración de operaciones de ALN. Por último, se describen los entornos de experimentación utilizados para evaluar las diferentes propuestas realizadas durante el resto del trabajo.

En el Capítulo 2 se estudia el problema de reducción de modelos. Primero se introducen unos pocos conceptos básicos de teoría de control y se presenta el problema de reducción de modelos. En especial, se estudian los métodos basados en la SVD, que tienen como etapa más costosa la resolución de ecuaciones de Lyapunov. Seguidamente, se describe la técnica para la resolución de una ecuación de Lyapunov mediante la función signo, y se profundiza en técnicas de aceleración y disminución del espacio de almacenamiento para dicha función. Posteriormente, se aborda la reducción de modelos en el caso de sistemas generalizados. Por último, se introducen los métodos de error relativo, en particular, el método de truncamiento estocástico balanceado. Esta técnica implica la resolución de una ecuación de Riccati, para lo cual se estudian los métodos basados en la función signo y el método de Newton.

En el Capítulo 3 se proponen diferentes métodos para acelerar la inversión de matrices utilizando GPUs. En primer lugar, se describen los métodos desarrollados para la inversión de matrices generales, en particular estudiando las estrategias tradicionales basadas en la factorización LU y en el método de Gauss-Jordan (GJE). Se presentan diversas variantes para cada una de estas dos clases de algoritmos y, posteriormente, se evalúan y validan de forma experimental sobre distintas plataformas de hardware. En segunda instancia, se estudian los métodos para la inversión de matrices simétricas y definidas positivas (SDP), estudiando tanto algoritmos basados en la factorización de Cholesky como en el método de GJE. Por último, se incluyen algunos trabajos introductorios sobre el uso de múltiples GPUs para acelerar el cómputo de la matriz inversa, con propuestas basadas en una planificación de ejecución estática.

En el Capítulo 4 se aborda la aceleración de los métodos de reducción de modelos utilizando plataformas que incluyen procesadores multinúcleo y GPUs. Primero se describen los casos de prueba utilizados para evaluar y validar las propuestas desarrolladas. A continuación, se presenta una propuesta para la resolución del problema estándar de reducción de modelos mediante el método de truncamiento balaceado (BT). Una característica destacable de esta propuesta es la inclusión de estrategias de precisión mixta, que posibilitan explotar la capacidad de cómputo de la GPU en simple precisión y, al mismo tiempo, conseguir resultados similares, en cuanto a calidad numérica, a los obtenidos al trabajar con números en doble precisión. Posteriormente, se extienden las ideas antes presentadas al caso de un sistema generalizado, es decir, un método basado en la resolución de dos ecuaciones generalizadas de Lyapunov. También se aborda la aceleración de la resolución de la ecuación matricial de Riccati en el contexto de técnicas de reducción de modelos con el método BST. Por último, se resumen algunas líneas de trabajo abordadas de forma preliminar.

Finalmente, en el Capítulo 5 se ofrecen las conclusiones generales del trabajo desarrollado en esta tesis, se relacionan los principales resultados obtenidos en forma de publicaciones, y se apuntan las líneas abiertas de investigación.

1.2. Computación de alto desempeño

A medida que la tecnología ha avanzado, la intención de resolver problemas cada vez mayores o con mayor precisión ha generado un ciclo en el cual la necesidad de cómputo crece de forma exponencial. Si bien los diseñadores de computadoras buscan lograr diseños cada vez más potentes, mejorando los diversos componentes y la capacidad de integración, existen varias limitaciones a la mejora del hardware. En particular, las principales limitaciones son económicas, ya que en muchos casos, para mejorar el desempeño de los computadores, es necesario incorporar componentes que implican un costo demasiado elevado. También existen limitaciones de carácter físico, relacionadas principalmente con la velocidad de las señales y la disipación del calor generado por los circuitos, cada vez más compactos y con más componentes. Ante la problemática expuesta, ha crecido la importancia de las técnicas de HPC capaces de extraer la mayor eficiencia de la arquitectura subyacente.

Uno de los problemas de la aplicación de técnicas de HPC son los altos costos económicos del hardware y diversos costos asociados (por ejemplo administración, mantenimiento y energía consumida por las máquinas). Sin embargo, en los últimos años, han irrumpido nuevas tecnologías de HPC de bajo costo, como es el caso del uso de las GPUs para computación de propósito general. Una de las grandes ventajas de estas arquitecturas es que permiten equiparar la capacidad de cómputo con equipos basados en CPUs tradicionales que costarían decenas o centenas de veces más. Además, la arquitectura intrínsecamente paralela y la exigencia ejercida por la industria de los videojuegos provocan una constante evolución de la capacidad de cómputo de las GPUs a la vez que mantienen los costos moderados. Una prueba fehaciente de lo expresado anteriormente es la evolución de equipos basados en GPUs en la lista Top500 [16], dedicada a enumerar los 500 equipos con más poder de cómputo en el mundo. Hasta 2008, no había ningún equipo que utilizara GPUs en el Top500; en la lista de noviembre de 2008 apareció un equipo en el puesto 29 (TSUBAME del *Tokyo Institute of Technology* de Japón [13]); en la lista publicada en noviembre de 2010 se pueden apreciar varios equipos que utilizan GPUs y, en particular, el primero, tercero y cuarto de la lista son plataformas basadas en GPUs (Tianhe-1A y Nebulae del *National Supercomputing Center* de China [11], y TSUBAME2, una evolución del equipo antes presentado de Japón).

En esta sección se reseñan las distintas estrategias utilizadas para mejorar las prestaciones utilizando técnicas de computación de alto desempeño. Inicialmente, se presenta una breve introducción a las arquitecturas paralelas y se describen algunas de las técnicas disponibles de programación paralela. En segundo lugar, se describe la aplicación de técnicas de HPC al ALN, y en particular, la aplicación de HPC a la resolución del problema de la reducción de modelos. Finalmente, se estudia y presenta el uso de GPUs para acelerar el cómputo de problemas de propósito general. Realizar un relevamiento exhaustivo y genérico sobre estrategias de HPC mediante el uso de GPUs, es una tarea que supera al alcance de este trabajo, por lo que se optó por realizar un resumen de los conceptos y trabajos más importantes o directamente relacionados con el ámbito de la tesis, el uso de GPUs para acelerar el cómputo de núcleos de ALN vinculados con la resolución de problemas de reducción de modelos.

1.2.1. Técnicas de programación paralela/distribuida

Las técnicas de programación paralela están estrechamente relacionadas con las arquitecturas de hardware, por esta razón, en este apartado se describen las diferentes familias de arquitecturas utilizadas para HPC.

Arquitecturas de computadoras paralelas

En la década de 1960, con la computadora CDC 6600, con 9 MFLOPS (millones de operaciones aritméticas en coma flotante por segundo) de poder de cómputo, se comenzó a utilizar el término supercomputadora. Desde entonces, el avance y las mejoras en el hardware de HPC han sido constantes y notables. Las opciones de arquitecturas paralelas disponibles son diferentes, tanto desde el punto de vista del poder de cómputo de los equipos, como de las conexiones entre los elementos de procesamiento, el uso de memoria, etc., teniendo cada una de las plataformas de HPC características, limitaciones y bondades específicas. La gran variedad de plataformas de hardware de HPC ha propiciado la existencia de múltiples clasificaciones basadas en diferentes criterios. A continuación se presentan algunas de las taxonomías más difundidas.

Flynn [85] utiliza como criterio para clasificar las computadoras el manejo que realizan de las instrucciones y de los datos. Se distingue así entre instrucciones unitarias (SI, del inglés *single instruction*) y múltiples (MI, del inglés *multiple instruction*) y datos unitarios (SD, del inglés *single data*) o múltiples (MD, del inglés *multiple data*), especificando cuatro clases, como se puede apreciar en la Tabla 1.1.

		Datos	
		SD	MD
Instrucciones	SI	SISD	SIMD
	MI	MISD	MIMD

Tabla 1.1: Clasificación de arquitecturas según la taxonomía de Flynn.

Dentro de la categoría SISD se encuentran las computadoras secuenciales. En las categorías SIMD y MIMD se encuentran generalmente las computadoras paralelas utilizadas en la actualidad. En los equipos SIMD, todos los procesadores reciben la misma instrucción y operan concurrentemente sobre distintos datos, mientras que en los MIMD las unidades de proceso pueden ejecutar simultáneamente diferentes instrucciones sobre distintos conjuntos de datos, trabajando asíncronamente.

Otra clasificación muy utilizada para arquitecturas con múltiples unidades de cómputo [17] se basa en la organización del sistema de memoria. Esta taxonomía reconoce dos clases de arquitecturas:

- Multiprocesadores de memoria compartida. Todos los procesadores comparten y tienen acceso a la memoria del sistema.

- Multiprocesadores de memoria distribuida. Cada unidad de proceso dispone de una fracción de la memoria total del sistema que solo puede ser accedida por la unidad propietaria. La compartición de datos entre los procesadores se realiza a través del intercambio de mensajes. La categoría también se denomina computadores paralelos débilmente acoplados. Un ejemplo típico de equipos pertenecientes a esta categoría son los clusters beowulf [43], sistemas de desempeño escalable en base a hardware de bajo costo, software libre y que utilizan redes de tecnología estándar para interconectar los nodos.

En la actualidad, otra taxonomía ampliamente extendida propone agrupar los computadores paralelos en base a grandes clases de arquitecturas [17]. A continuación se describen las cinco categorías.

- Multiprocesadores simétricos (SMP, del inglés *Symmetric MultiProcessor*). Son equipos con varias unidades de procesamiento muy acopladas (comparten gran cantidad de los recursos incluida la memoria). También suelen denominarse sistemas de grano grueso.

- Equipos con acceso a memoria no uniforme (NUMA). Estos computadores están compuestos por varios procesadores, cada uno de los cuales posee su propia memoria. La característica que distingue a los computadores de esta clase es que a pesar de que cada procesador posee su propia memoria, se dispone de un direccionamiento de memoria global. Existen equipos que emplean coherencia de caché (cc-NUMA) para facilitar su programación, en contraposición con los que no (nc-NUMA).

- Computadores masivamente paralelos (MPP, del inglés *Massively Parallel Processing*). Son equipos que poseen varias unidades de procesamiento poco acopladas, en los que cada procesador accede a su propia memoria. También son denominados sistemas de grano fino.

- Clusters de PCs. Se constituyen por un conjunto de computadores personales (PCs) interconectados mediante una red.

- Grids. Se componen de un conjunto de equipos autónomos conectados mediante una red. En este caso los equipos pueden ser heterogéneos en cuanto a arquitectura (incluso pueden pertenecer a distintas clases de las antes mencionadas), velocidad, sistema operativo, etc.

Si bien las clasificaciones descritas anteriormente son suficientes a los efectos de este trabajo, se destaca que existen otras clasificaciones en base a distintos criterios: basadas en la forma de trabajo de los procesadores de los equipos (síncrona o asíncrona), el poder de cómputo de las unidades de proceso, la cantidad de procesadores, la topología de la red de interconexiones, etc. [17].

Programación paralela

Como se comentó en el apartado anterior, existen diversas arquitecturas de computadoras para HPC. En consecuencia, existen distintas técnicas de programación paralela, cada una de las cuales se adapta mejor a las características de un tipo específico de arquitectura paralela. La programación paralela consiste, esencialmente, en dividir el problema para su resolución simultánea en diferentes unidades de proceso de forma cooperativa. Para efectuar la división del problema existen dos estrategias principales:

- Descomposición funcional. En este caso se dividen las distintas tareas a realizar por el algoritmo entre las distintas unidades de procesamiento disponibles.

- Descomposición de datos. En esta estrategia todas las unidades de procesamiento disponibles realizan las mismas tareas, pero cada una trabaja sobre un subconjunto de los datos del problema.

Además de las dos estrategias descritas anteriormente, en la práctica son ampliamente utilizadas las estrategias híbridas, que dividen tanto las tareas como los datos entre las diferentes unidades de procesamiento.

Al igual que ocurre con las estrategias de división de problemas, las distintas técnicas de programación paralela se encuentran muy ligadas al hardware subyacente, y en particular, a la forma en que las unidades computacionales pueden comunicarse y compartir información. A continuación se presentan las principales técnicas de programación paralela y su relación con las diferentes arquitecturas.

- Pasaje de mensajes: las unidades de procesamiento se comunican y coordinan entre sí mediante el pasaje explícito de mensajes. Esta técnica es muy utilizada cuando se trabaja sobre hardware poco acoplado, típicamente clusters o sistemas distribuidos.

- Memoria compartida: las unidades de procesamiento trabajan de forma coordinada gracias al uso compartido de la memoria central. Esta técnica es típicamente utilizada sobre multiprocesadores de memoria compartida.

Al programar en paralelo aparecen varios factores específicos que influyen en el rendimiento. Entre ellos destacan:

- Balance de carga: es necesario dividir el trabajo a realizar de forma equitativa entre las distintas unidades de proceso. La distribución será igualitaria si todos los elementos de procesamiento tienen la misma capacidad (es decir, arquitectura homogénea) o se deberá distribuir el trabajo de forma proporcional a las capacidades de cómputo de los componentes si se trata de computadores heterogéneos.

- Buen nivel de concurrencia: es prioritario lograr que todas las unidades de proceso trabajen de forma concurrente, evitando tiempos superfluos de espera y sincronización.

- Bajo sobrecosto: se deben minimizar las tareas de coordinación necesarias para realizar el trabajo en paralelo, por ejemplo, minimizando comunicaciones, trabajo redundante, etc.

Otro aspecto a tener en cuenta al desarrollar programas paralelos es el tratamiento de las tareas no paralelizables, es decir, aquellas tareas que sólo pueden ser realizadas por una única unidad de procesamiento. En este contexto, el tiempo de ejecución de las tareas secuenciales es una cota inferior del tiempo total de ejecución del algoritmo paralelo. El enunciado de este concepto es conocido como ley de Amdahl [20].

Herramientas para el diseño e implementación de aplicaciones paralelas

Así como existen diversas arquitecturas de computadores paralelos y se dispone de diferentes técnicas de programación paralela, también se han implementado diversas herramientas para desarrollar programas bajo el paradigma de la programación paralela y distribuida. Durante muchos años la tendencia por parte de los vendedores de hardware fue desarrollar herramientas específicas para sus equipos, existiendo incluso herramientas diseñadas para un determinado modelo de computador. Sin embargo, en las últimos décadas la tendencia gradualmente fue cambiando hacia el desarrollo de herramientas genéricas, buscándose cada día un mayor equilibrio entre el desempeño computacional y la portabilidad/productividad.

Por lo general, las características de las distintas herramientas están asociadas con alguna técnica particular de programación paralela. Por este motivo, en muchas ocasiones la elección de una herramienta u otra se ve condicionada por la técnica de programación a utilizar, que a su vez generalmente se encuentra supeditada por el hardware subyacente. Esto implica que es imposible afirmar de forma general que una herramienta es mejor que otra, siendo necesario evaluar en cada ocasión, y dependiendo del hardware y estrategia de programación paralela a utilizar, qué herramienta se adapta mejor a la resolución de un problema determinado en una arquitectura concreta.

En este trabajo se reseñan las características más importantes de dos herramientas de uso general, el estándar MPI (*Message Passing Interface*) [111] para plataformas de memoria distribuida y la interfaz OpenMP [9] para memoria compartida.

El estándar MPI

Existen diversas bibliotecas para la programación paralela que implementan el paradigma de pasaje de mensajes. Muchas de las grandes compañías que venden computadoras paralelas han creado sus propias bibliotecas. De forma frecuente estas bibliotecas se encuentran diseñadas y optimizadas para un tipo de arquitectura de hardware en particular, siendo en general de difícil comprensión, poca legibilidad y no portables a otras arquitecturas. En las últimas décadas, con el propósito de mejorar la portabilidad de aplicaciones, se han introducido las especificaciones de tipo estándar. Pese a la potencial disminución de la eficiencia en los programas debido a una adaptación incompleta a las propiedades del hardware, el uso de estas especificaciones se ve compensada en una visión de largo plazo, dado que estas estrategias permiten portar códigos a nuevas arquitecturas; facilitar el intercambio de experiencias, el mantenimiento y legibilidad del software; etc.

El estándar MPI [111] fue definido en el año 1992 por un conjunto de desarrolladores de software y aplicaciones científicas en coordinación con importantes empresas (IBM, Intel), con el objetivo de proporcionar una especificación de biblioteca implementable sobre una variada gama de arquitecturas, permitiendo así diseñar aplicaciones distribuidas portables y eficientes. En el estándar se definen las funcionalidades y diversas características deseables que debe incorporar una biblioteca orientada al pasaje de mensajes. El estándar MPI se basa en la experiencia obtenida con herramientas anteriores, como por ejemplo la biblioteca PVM (*Parallel Virtual Machine*) [91], ampliamente utilizada en los finales de la década de 1980 en el ámbito científico.

Entre las principales características del estándar MPI destacan:

- El paralelismo es explícito, es decir, es definido y controlado en su totalidad por el programador.
- Utiliza como único mecanismo de comunicación entre los procesos el pasaje de mensajes explícito.
- Tiene como objetivo principal el trabajo sobre plataformas de memoria distribuida, aunque incluye primitivas para el trabajo sobre plataformas de memoria compartida.
- Considera como mecanismo de diseño de programas el modelo SPMD (del inglés *single program multiple data*), aunque posee la capacidad de adaptarse de manera sencilla al desarrollo de programas bajo un modelo MIMD.

El estándar completo de MPI posee más de 130 funciones que permiten la sincronización, comunicación y trabajo cooperativo. Entre las tareas que implementan las funciones de MPI, destacan el envío y recepción punto a punto de mensajes (bloqueantes o no), las operaciones de difusión (*broadcasting*, envío de mensajes de un proceso a todos los restantes), las operaciones de reducción (recolectar y reducir datos de varios procesos en uno), y la obtención de información del proceso y entorno de ejecución.

Existen diversas implementaciones de la especificación MPI para trabajar con los lenguajes C y Fortran. Dos de las más difundidas son MPICH [8, 109, 110], del Argonne National Laboratory (ANL) [10] de Estados Unidos, y OpenMPI [89].

La interfaz OpenMP

El éxito del estándar MPI por un lado y el renovado interés por el paradigma de paralelismo de memoria compartida debido al crecimiento de capacidad de los computadores multinúcleo, impulsaron a un grupo de instituciones, fabricantes de computadoras y empresas desarrolladoras de compiladores a proponer estándar para programación sobre memoria compartida. En el año 1997 se presentó la interfaz OpenMP.

El diseño de la interfaz posee como principales características ser portable, escalable y relativamente simple de utilizar para el desarrollo de programas paralelos tanto para pequeñas computadoras como para equipos de gran porte.

Actualmente se dispone de versiones de OpenMP para los lenguajes C/C++ y las distintas versiones de Fortran. Entre las funcionalidades de la interfaz se encuentran el definir secciones paralelas, secciones críticas y bucles paralelos. Además, emplea variables de entorno para controlar la ejecución de un programa; por ejemplo, la cantidad máxima de hilos a utilizar durante la ejecución se especifica mediante la variable $OMP\_NUM\_THREADS$.

Sobre las diversas opciones para trabajar con la interfaz OpenMP se puede profundizar en el trabajo de Narus et al. [166].

Medidas del desempeño computacional

Una primera aproximación intuitiva para medir el desempeño de los computadoras puede obtenerse mediante la observación de la velocidad del procesador o del número de instrucciones que es capaz de ejecutar, evaluando los millones de instrucciones por segundo (MIPS) que puede realizar el equipo. Sin embargo, los MIPS no aportan información para comparar dos computadores que poseen distintas arquitecturas, y sobre todo, distintos juegos de instrucciones.

Una forma, más completa y ampliamente utilizada en computación científica, es medir el desempeño de una computadora o algoritmo evaluando cuántos millones de operaciones de coma flotante puede realizar en un segundo (MFLOPS), o en computadoras más potentes cuántos miles de millones de flops (GFLOPS) pueden realizar.

Otra estrategia orientada a la evaluación de los equipos es emplear alguno de los *benchmark* de uso público. Un *benchmark* es un conjunto de programas que se utilizan para comparar el rendimiento de distintos computadores o algoritmos. Entre los benchmarks de uso más extendido en la comunidad se encuentran el HINT [118], propuesto por Gustafson y Snell, y el Linpack [5], propuesto por Dongarra. Este último benchmark emplea subrutinas de álgebra lineal, como la factorización LU, para evaluar el desempeño de los computadores y es utilizado para confeccionar la lista Top500.

En muchas situaciones o estudios, el interés radica en la evaluación de distintos algoritmos o estrategias para la resolución de un mismo problema. Si durante el estudio se emplea siempre la misma computadora para realizar las ejecuciones, utilizar el tiempo de cómputo como medida de desempeño es una opción válida. El tiempo efectivo de cómputo permite realizar una evaluación de forma global para una arquitectura; esta evaluación será válida independientemente de si el algoritmo utiliza más instrucciones, más operaciones de coma flotante, menos movimientos en la memoria, etc.

El análisis del desempeño computacional de un programa es una tarea sumamente compleja. Cuando se trata de medir el desempeño computacional de un algoritmo que utiliza técnicas de programación paralela, la dificultad es mayor, ya que sumado a la complejidad de evaluar un programa secuencial aparecen nuevos condicionantes como son los tiempos de comunicación entre procesos, los tiempos de sincronización, etc., que tienen que ser evaluados de forma separada.

Otro aspecto a tener en cuenta al medir el desempeño de algoritmos que se ejecutan en paralelo es conocer la ganancia al ejecutarlos en paralelo en relación con la versión secuencial. Esta ganancia es habitualmente conocida como el *speedup* (también se estila utilizar el *speedup* normalizado, conocido como eficiencia computacional). Estos resultados están fuertemente ligados al equipo en el cual se ejecutan. En resumen, para evaluar programas paralelos se puede utilizar:

- El tiempo total de ejecución, comparando los tiempos de ejecución de las distintas versiones de un algoritmo en un mismo equipo.
- El *speedup* y la eficiencia computacional, que se definen en el siguiente apartado.

Existe otra gran cantidad de métricas para evaluar el desempeño de programas de computación paralela. Alguna de las más utilizadas son la escalabilidad, la isoeficiencia y la isovelocidad. Para profundizar sobre el tema se puede consultar en el trabajo publicado por Sahni y Thanvantri [198]. Además, en los últimos años existe un creciente interés por el uso de medidas que incluyan aspectos económicos, como la capacidad de cómputo por dólar invertido, o la capacidad de cómputo por potencia consumida.

Speedup y eficiencia de un algoritmo paralelo

El *speedup* absoluto relaciona el tiempo de ejecución del mejor algoritmo secuencial conocido con el tiempo de ejecución del algoritmo ejecutado en paralelo. La métrica se define por la expresión $S(n) = \frac{Tsec}{T(n)}$ donde $Tsec$ es el tiempo de ejecución del mejor algoritmo secuencial conocido y $T(n)$ representa el tiempo de ejecución del algoritmo paralelo utilizando n unidades de cómputo. La definición del *speedup* absoluto es poco práctica, porque no siempre se conoce el mejor algoritmo secuencial para resolver un problema, y en caso de que éste se conozca, en la mayoría de las ocasiones no se dispone de una implementación para evaluar su ejecución en un equipo determinado. Por este motivo, en general suele utilizarse como base de la comparación el tiempo de ejecución del algoritmo paralelo trabajando en una única unidad de cómputo. La medida resultante, denominada *speedup* algorítmico, se define mediante la expresión $S(n) = \frac{T(1)}{T(n)}$ donde $T(1)$ es el tiempo de ejecución del algoritmo paralelo ejecutando en una única unidad de cómputo.

En ocasiones, no sólo es importante el valor de *speedup* para una cantidad dada de unidades de cómputo, sino que también es relevante observar la variación de la ganancia a medida que crece la cantidad de unidades computacionales utilizadas. En este caso se puede utilizar la métrica eficiencia, que corresponde al valor del *speedup* normalizado por el número de unidades computacionales utilizadas. La métrica se define como $E(n) = \frac{S(n)}{n}$, y dependiendo del modo de calcular el *speedup*, se tendrá un valor de eficiencia absoluta o de eficiencia algorítmica (también denominada eficiencia relativa).

1.2.2. HPC y ALN

Diversas aplicaciones científicas tienen la particularidad de que sus etapas más costosas, desde el punto de vista de tiempo de ejecución, requieren el cómputo de operaciones de ALN. A modo ilustrativo, algunos pocos ejemplos donde se puede corroborar esta afirmación son los problemas de optimización lineal, resolución de ecuaciones diferenciales, problemas de radiosidad en computación gráfica y el tema central de esta tesis, la reducción de modelos. Este hecho ha motivado que el área de ALN concentre, desde varias décadas atrás, gran caudal de trabajo, buscando desarrollar métodos eficientes que exploten las características de las diferentes arquitecturas de computadoras empleadas.

Una política impuesta en el área es el desarrollo de especificaciones de bibliotecas (o interfaces) cuyas implementaciones permiten resolver problemas básicos de ALN, que pueden ser utilizadas para construir métodos que permitan tratar problemas más complejos. También, de esta manera se puede fácilmente optimizar las aplicaciones sin necesidad de grandes modificaciones y tiempos de desarrollo, únicamente empleando implementaciones eficientes de las especificaciones de las bibliotecas que resuelven los núcleos básicos.

Un caso paradigmático de lo antes expuesto es la especificación BLAS (del inglés *Basic Linear Algebra Subprograms*), que incluye en su funcionalidad operaciones básicas de ALN. Buscando replicar este caso se han desarrollado diversos esfuerzos similares, como LAPACK y SCALAPACK. A continuación se presenta una breve reseña de estas bibliotecas.

Existen múltiples bibliotecas de características similares a las presentadas a continuación, así como esfuerzos exitosos para desarrollar soluciones a problemas más específicos. En el repositorio NetLib [1] se encuentra un buen compendio del trabajo en el área, teniendo disponible una amplia gama de bibliotecas, algoritmos y artículos científicos en este ámbito.

BLAS

En 1973, Hanson et al. [121] dieron cuenta de la necesidad de adoptar un conjunto de rutinas básicas para problemas de ALN. Pretendían definir un estándar para las operaciones elementales entre vectores que permitiera mejorar la claridad, portabilidad y modularidad de las rutinas numéricas, simplificando el mantenimiento y desarrollo de futuro software. La idea tuvo un éxito considerable; la especificación BLAS fue desarrollada por varias empresas de fabricación de computadoras, que implementaron el estándar optimizando las operaciones para la arquitectura de sus equipos. Al correr de los años, con la evolución de las arquitecturas y en especial con la difusión de las computadoras vectoriales, la biblioteca BLAS pasó a ser una limitante al momento de mejorar el desempeño de los códigos numéricos. Como consecuencia, en el año 1984 comenzó a discutirse la extensión de los subprogramas incluidos en la primera especificación de BLAS. Tras varias reuniones y publicaciones se presentó la extensión, agregando funciones para las operaciones matriz-vector elementales. Las nuevas funciones se denominaron BLAS-2, reservándose el término de BLAS-1 para las funciones originales. Nuevamente, la evolución de la tecnología, en especial el creciente uso de memoria jerárquica y la difusión de los procesadores de memoria compartida, motivaron la extensión de la especificación, debido a que las funciones de BLAS-1 y BLAS-2 exhibían una relación entre las operaciones de coma flotante realizadas y los movimientos de datos entre los distintos niveles de memoria poco eficiente. La extensión incluyó operaciones de tipo matriz-matriz, que fueron denominadas funciones BLAS-3.

Los tres niveles de BLAS resuelven múltiples operaciones con vectores y varios tipos de matrices (generales y estructuradas) para varios tipos de datos (simple y doble precisión, reales y complejos). En el trabajo de Dongarra et al. [78] se puede obtener una descripción completa de los distintos niveles de funciones de la biblioteca BLAS.

Una extensión sobre la implementación de BLAS consistió en dotarlas de la capacidad de ser ejecutadas en paralelo, sobre un modelo de memoria compartida. En este sentido existen implementaciones que utilizan estrategias multihilo como la presentada por T. Guignon, denominada BLASTH [203], la desarrollada por K. Goto disponible en su sitio web [101] y la incluida en la distribución de Intel, la biblioteca MKL [131].

Otra extensión de la biblioteca BLAS disponible es la que se define para vectores y matrices dispersas. Algunos ejemplos de implementaciones de esta extensión son la NIST Sparse BLAS [14] y la disponible en el repositorio Netlib de Dodson et al. [3].

LAPACK

En el año 1987 se desarrolló en la Universidad de Tennessee, Estados Unidos, un proyecto para implementar una biblioteca que permitiera la resolución de los problemas más comunes de ALN. El esfuerzo se cristalizó en la especificación LAPACK [74] (*Linear Algebra PACKage*), y una implementación de la misma, conocida como implementación de referencia. La especificación incluye funciones para la resolución de sistemas de ecuaciones lineales, aproximación de mínimos cuadrados, búsqueda de valores propios y de valores singulares.

La implementación de referencia de LAPACK utiliza rutinas de BLAS para efectuar las operaciones elementales sobre vectores y matrices. Este hecho permite la portabilidad y adaptación de LAPACK a diferentes arquitecturas sin afectar negativamente su desempeño.

La implementación de referencia de LAPACK está originalmente desarrollada en el lenguaje Fortran 77, es de dominio público y se puede obtener en el repositorio Netlib [2]. Además, existen diversas implementaciones optimizadas, entre las que destaca la incluida en la biblioteca MKL de Intel.

SCALAPACK

El aumento en la utilización de computadoras paralelas de bajo costo, y en especial, de aquellas con memoria distribuida, motivó la evolución de LAPACK a una nueva versión basada en técnicas de programación paralela distribuida; esta nueva biblioteca se denominó SCALAPACK [71] (*SCAlable* LAPACK) y la primera versión fue publicada en el año 1995.

Una de las principales características de la biblioteca SCALAPACK es que su diseño especifica distintos niveles de abstracción, buscando encapsular diferentes tareas en distintas componentes. En la Figura 1.1 se puede observar el diseño de componentes de la biblioteca SCALAPACK.

Al igual que LAPACK, SCALAPACK posee funciones para las operaciones más comunes en el campo del ALN, tales como la factorización LU y la factorización QR. Sin embargo, SCALAPACK está diseñada para su ejecución en plataformas de memoria distribuida, en las que las matrices y vectores se distribuyen entre las unidades de procesamiento de forma cíclica sobre una estructura de comunicación 2D.

Al diseñar la biblioteca SCALAPACK se decidió encapsular las operaciones sobre vectores que se debían realizar de forma distribuida. En este sentido, y debido a la notable influencia que tuvo en la comunidad científica la biblioteca BLAS, se creó la especificación de la biblioteca PBLAS (*Parallel* BLAS) [70]. La especificación de PBLAS posee las mismas operaciones que dispone BLAS, pero en este caso las operaciones están diseñadas para ser ejecutadas en paralelo sobre memoria distribuida mediante el paradigma de pasaje de mensajes (MPI, PVM, etc.).

Las operaciones de comunicaciones que utilizan pasaje de mensajes son resueltas mediante la invocación a funciones de la biblioteca BLACS. BLACS consta de un conjunto de rutinas para la transferencia de datos mediante el pasaje de mensajes específicamente diseñadas para dar soporte a la resolución de operaciones de ALN en plataformas con memoria distribuida. La biblioteca incluye la especificación de rutinas de envío y recepción de mensajes de forma síncrona y asíncrona, tanto para realizar la comunicación de matrices y submatrices entre procesadores, como para realizar envíos en forma de difusión y reducciones globales (sumas, máximo, mínimo, etc.). En el repositorio NetLib se pueden obtener versiones de BLACS implementadas para diversas bibliotecas de pasaje de mensaje (MPI, PVM, NX), y con interfaz para los lenguajes Fortran y C, al igual que una implementación de la biblioteca PBLAS.

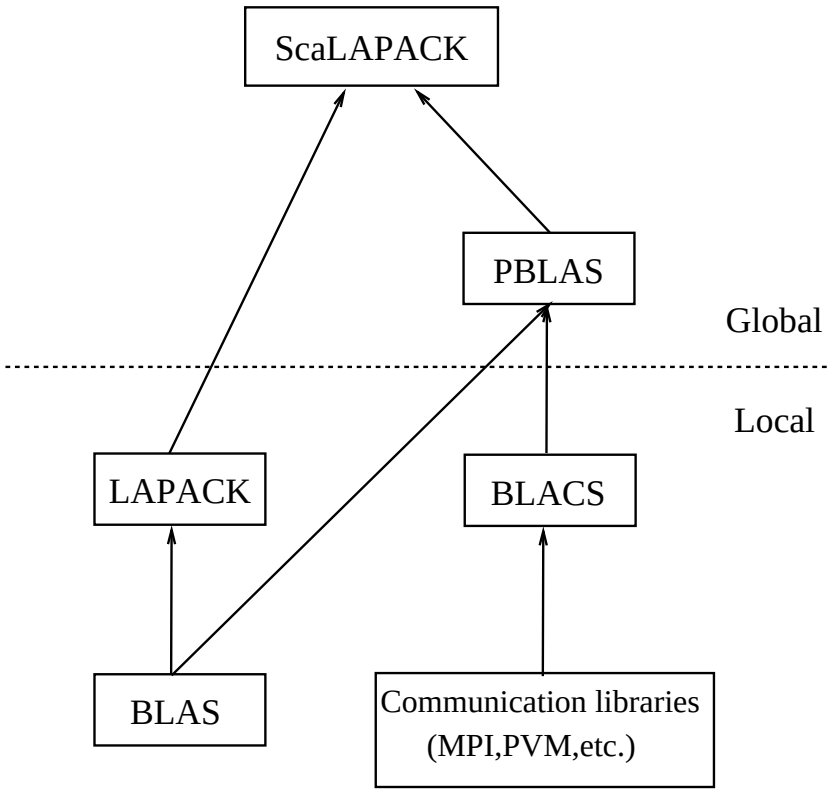

Figura 1.1: Diseño simplificado de la biblioteca SCALAPACK. Extraído de [79].

1.2.3. Aplicación de técnicas de HPC a la reducción de modelos

Uno de los esfuerzos pioneros en el desarrollo de herramientas que incluyan estrategias de HPC para la resolución de problemas de control y en particular de reducción de modelos es la biblioteca SLICOT (*Subroutine Library in Control Theory*)[1] [216], que fue desarrollada a finales de la década de los 90, en el marco del proyecto NICONET (*Numerics in Control Network*) [12]. La biblioteca está implementada en Fortran 77 y también es posible su utilización desde Matlab. Su diseño se basa en el uso de las especificaciones LAPACK y BLAS, por lo cual se puede obtener fácilmente una versión paralela de la biblioteca sobre memoria compartida utilizando una versión multihilo de BLAS. SLICOT es fuertemente utilizada por investigadores del área aún en la actualidad. Los esfuerzos previos a la introducción de SLICOT por desarrollar bibliotecas para la resolución de problemas de control están relevados en el trabajo de Benner et al. [50].

Posteriormente, se desarrollaron diferentes esfuerzos para incorporar estrategias de paralelismo de memoria distribuida a la biblioteca SLICOT, desembocando en la realización de la biblioteca PLiC (*Parallel Library in Control*); los principales trabajos relacionados a este proceso se resumen a continuación.

El trabajo de Blanquer et al. [58] presenta algunas propuestas tendentes a paralelizar la resolución de la ecuación de Lyapunov proponiendo PSLICOT (*Parallel* SLICOT). Posteriormente, uno de los principales esfuerzos se vio plasmado en el desarrollo de la primera versión de la biblioteca PLiC [52] propiamente dicho. La biblioteca PLiC posee dos componentes fundamentales, PLiCOC y PLiCMR (del inglés *Parallel Library in Control: Model Reduction*). La primera da soporte a la re-

[1] Disponible en `http://www.win.tue.nl/niconet/NIC2/slicot.html`.

solución de problemas de control mientras que la segunda componente está especialmente diseñada para la resolución de problemas de reducción de modelos. PLiC incluye rutinas para la resolución de problemas de análisis y diseño de SDLs invariantes en el tiempo y de gran escala sobre arquitecturas de computadoras distribuidas y paralelas. Contiene más de 30 subrutinas que permiten resolver ecuaciones matriciales lineales y cuadráticas (Lyapunov, Sylvester, Stein y la ecuación algebraica de Riccati), reducción de modelos, estabilización parcial, y diversas rutinas de soporte. El modelo de paralelismo emula el utilizado por la biblioteca SCALAPACK. En particular, la resolución de las operaciones de álgebra se hacen invocando a rutinas de SCALAPACK, siempre que es posible, y las comunicaciones se efectúan mediante invocación a rutinas de la biblioteca BLACS.

Otra extensión a la biblioteca PLiC fue el desarrollo de una versión accesible mediante servicios web. Esta propuesta permite resolver problemas de reducción de modelos en sistemas remotos a través de la red, eliminando la necesidad de contar con plataformas de HPC locales para la resolución de este tipo de problemas. En el trabajo de Benner et al. [49] se puede profundizar en la temática. Con el mismo objetivo que el trabajo anterior, facilitar el uso de la biblioteca PLiC, en el trabajo de Galiano et al. [213] se presenta el paquete PyPLiC. En él, se ofrece una interfaz de alto nivel en el lenguaje de *scripting* Python para la biblioteca PLiCMR.

En problemas con modelos dispersos o en presencia de matrices estructuradas banda en los SDL, los métodos LR-ADI [113, 183, 220] son particularmente eficientes. En el trabajo de Remón et al. [188], se aplican diferentes técnicas de paralelismo al método. Los autores presentan dos implementaciones paralelas sobre memoria compartida (arquitecturas SMP) para la factorización LU de matrices de banda. En el trabajo se muestra también cómo aplicar las variantes de factorización desarrolladas para acelerar los métodos LR-ADI en la resolución de problemas de reducción de modelos, consiguiendo porcentajes de mejora superiores al 50 %.

En el caso de los métodos de reducción de modelos basados en sub-espacios de Krylov, existe gran caudal de trabajos referentes a la aplicación de técnicas de HPC [38, 195]. Generalmente, dada la estrategia de resolución de estos métodos, las técnicas de HPC se aplican a la resolución del sistema, es decir el método de Lanczos o alguna de sus variantes.

1.2.4. Utilización de GPUs para la resolución de problemas de propósito general

En los últimos años, las tarjetas gráficas han sufrido un desarrollo explosivo. Movidas por el dinámico mercado de los videojuegos, las empresas que diseñan y fabrican tarjetas (principalmente NVIDIA y ATI) se encuentran en una carrera vertiginosa por conseguir la supremacía tanto en capacidad de cálculo como en bajo costo. Como resultado de esta carrera se puede observar la notoria evolución en la capacidad de cómputo de las tarjetas; por ejemplo el modelo NVIDIA GeForce 6800 Ultra (presentado en abril de 2004) alcanza un máximo teórico de 50-60 GFLOPS, mientras que aproximadamente 30 meses después, la tarjeta NVIDIA GeForce 8800 GTX (de octubre de 2006) supera los 500 GFLOPS. Esta evolución es muy superior a la conseguida por las CPU. Cabe recordar aquí, el enunciado conocido como la ley de Moore, que indica que el poder de cómputo en las CPU se duplica cada año y medio.

Otro aspecto destacado de la evolución de las GPUs es el avance en la versatilidad que han alcanzado. Es necesario mencionar que hace apenas 10 años las GPUs eran dispositivos completamente rígidos en cuanto a su aplicación, mientras que hoy en día permiten un nivel importante de configuración y programación. En lo que respecta a la resolución de problemas de computación gráfica, campo de aplicación para el cual fueron diseñadas originalmente, en sus inicios estos dispositivos solo realizaban ciertos cálculos prefijados del cauce gráfico (*pipeline* gráfico), mientras que los modelos actuales permiten configurar las etapas y especificar las operaciones a realizar en cada etapa del cauce en lenguajes de alto nivel. En el trabajo de Owens et al. [179] se puede profundizar

sobre la evolución de la arquitectura y capacidades de las tarjetas gráficas.

Uno de los sustentos de la evolución impresionante en la capacidad de cómputo de las GPUs es la arquitectura intrínsecamente paralela subyacente, ya que el objetivo para el cual fueron diseñadas las tarjetas gráficas, acelerar el proceso de visualización en pantalla, implica la resolución de problemas explícitamente paralelos.

El aumento en la capacidad de cómputo y el avance en la flexibilidad para la programación de las GPUs antes mencionados implicó un notorio aumento del interés de la comunidad científica por explotar dicho hardware, extendiendo su utilización a la resolución de problemas de propósito general. Así, ha emergido como área de intenso trabajo en los últimos años la denominada GPGPU (del inglés *General Purpose* GPU). Las empresas fabricantes de GPUs han sido receptivas a este interés, y prueba de ello es el fuerte impacto que las carencias detectadas para el avance de GPGPU han tenido en el desarrollo de la arquitectura de estos dispositivos en los últimos años. Un ejemplo de ello es la evolución en la manipulación de números en las tarjetas [141]. En un principio, en las tarjetas gráficas se utilizaban números en 8 bits (año 2001), ya que es más que suficiente para el manejo de los rangos básicos de colores, luego se comenzó a trabajar con 16 bits, posteriormente se lanzaron las tarjetas que utilizan el estándar IEEE de coma flotante simple precisión (32 bits), y recientemente se presentaron las tarjetas que soportan números con doble precisión (64 bits), permitiendo así la resolución de problemas de computación científica donde la precisión numérica es crucial. Además, el fuerte interés de los fabricantes de tarjetas por posicionarse en el uso de GPUs para la resolución de problemas de propósito general, y para HPC en particular, se puede constatar con el lanzamiento por parte de NVIDIA de familias de GPUs específicas para este nicho del mercado.

El constante aumento en la capacidad de cómputo ofrecido por las GPUs y el vertiginoso avance del área se ve frenado por diversos problemas. Entre ellos, la constante evolución del hardware implica que las técnicas de diseño y programación utilizadas deban replantearse de forma casi constante. Además, existen muy pocas herramientas de propósito general para el desarrollo de aplicaciones en GPU, en contraposición al desarrollo de software en arquitecturas de hardware clásicas, en las cuales se dispone de multitud de herramientas y metodologías que han sido revisadas y mejoradas durante varios años en base a la experiencia.

En consecuencia a lo expuesto anteriormente, el uso de GPUs en HPC se ha convertido en una importante área de trabajo que cuenta ya con numerosos y relevantes avances. Es particularmente interesante debido a su bajo costo, en contextos de acceso limitado a grandes (y costosas) plataformas de hardware para HPC. Una de las filosofías que parece adecuada en el contexto de evolución constante de las arquitecturas masivamente paralelas de los procesadores gráficos, es el desarrollo de núcleos básicos de permitan descargar parte del cómputo desde la CPU a la GPU, acelerando así la resolución de problemas más complejos de forma sencilla.

Evolución de la GPGPU

En los primeros años del desarrollo de GPGPU el crecimiento de la capacidad computacional del hardware no fue acompañado por un avance en el software de manejo de las GPUs. Para programar las GPUs era necesario realizar llamadas a las APIs (del inglés *application programming interface*) gráficas como OpenGL [199] y DIRECTX [59]. Esto implicaba la necesidad de conocer profundamente los tópicos de computación gráfica y mapear el problema a resolver en conceptos de computación gráfica. Posteriormente, se empezaron a utilizar diferentes lenguajes de tipo ensamblador específicos de cada modelo, lo que implicaba la existencia de varios lenguajes y una baja portabilidad de las aplicaciones. Para tratar de paliar esta carencia, se presentaron varios lenguajes de alto nivel, que permitían acceder a diferentes modelos de GPUs existentes. Alguna de las herra-

mientas desarrolladas son Brook [64] y Cg [84]. Sin embargo, cada herramienta seguía siendo muy dependiente de la arquitectura de la GPU, el modelo, etc. Posteriormente, en el año 2007, NVIDIA presentó CUDA (del inglés, *Compute Unified Device Architecture*) buscando solucionar algunos de estos inconvenientes. Este salto importante en el software se dio en forma conjunta con un cambio radical en la arquitectura de las GPUs (de NVIDIA), ya que, junto con CUDA se presentó por parte de NVIDIA la arquitectura G80 que, como principal característica, presenta la arquitectura unificada sin distinción ya entre procesadores de píxeles y de vértices. En el Anexo A se profundiza en las arquitecturas de las GPUs.

CUDA

CUDA es una arquitectura de software para el cómputo de propósito general enfocada hacia el cálculo masivamente paralelo y la capacidad de procesamiento de las tarjetas gráficas de NVIDIA. CUDA se compone de una pila de capas de software. Entre estas capas se encuentra un controlador de hardware, una interfaz de programación de aplicaciones y dos bibliotecas matemáticas de alto nivel, CUFFT y CUBLAS (una implementación de BLAS en GPU). El driver CUDA está dedicado a la transferencia de datos entre la GPU y la CPU. Conceptualmente, la arquitectura de CUDA se construye alrededor de una matriz escalable de multiprocesadores (SMs, del inglés *multithreaded Streaming Multiprocessors*), como se ilustra en la Figura 1.2. Un multiprocesador en las GPU actuales (arquitectura G80) consiste de ocho procesadores escalares (SP), así como de elementos adicionales como la unidad de instrucciones multihilo y un chip de memoria compartida. Los multiprocesadores crean, gestionan y ejecutan hilos concurrentemente sin apenas costo de planificación. Para lograr explotar de manera optimizada los recursos de la GPU, CUDA provee un nuevo conjunto de instrucciones accesibles a través de lenguajes de alto nivel como son Fortran y C.

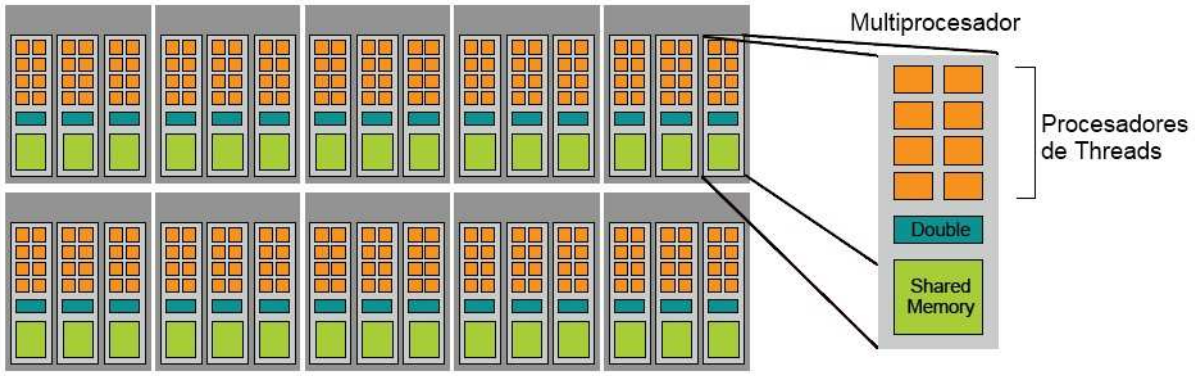

Figura 1.2: Arquitectura CUDA. Extraído de [141].

Esta arquitectura permite abstraer la GPU como un conjunto de multiprocesadores compuestos a su vez por un conjunto de procesadores orientados a la ejecución de hilos. La noción fundamental en el nuevo modelo de ejecución es que cada multiprocesador ejecutará en cada uno de sus procesadores el mismo conjunto de instrucciones, pero sobre distintos datos, es decir, paralelismo bajo el paradigma de programación SPMD. En particular, y debido a la gran cantidad de hilos que manejan, es común referirse al paralelismo ofrecido por las GPUs como SIMT (del inglés *single instruction multiple threads*). Los programas a ejecutar en la GPU se denominan *kernels*, y se organizan en un *grid* tridimiensional de hilos de ejecución (coordenadas x, y y z). A su vez, los hilos

se agrupan en bloques bidimensionales (coordenadas x e y); estas coordenadas se utilizarán para identificar el conjunto de datos sobre el que se quiere que actúe cada hilo. Cada bloque será ejecutado en un multiprocesador en forma independiente (esto es, no puede haber ni comunicación de datos ni sincronización entre los bloques[2]) y cada hilo en un procesador de hilo (entre estos hilos sí puede haber sincronización y comunicación). Al conjunto de hilos que se ejecutan en el mismo instante de tiempo en un mismo bloque se lo denomina *warp*. El tamaño del *warp* es 32 en las arquitecturas que tienen 8 MPs. Organizar la ejecución de esta manera permite escalar sin necesidad de recompilar el programa, dado que siempre (que se creen los hilos suficientes) se utilizarán todos los multiprocesadores disponibles para la ejecución de los bloques, así como todos los procesadores de hilos que los componen.

Jerarquía de Memoria

El acceso a memoria por parte de los hilos es un componente fundamental en el desempeño de un programa sobre CUDA. Las GPUs de NVIDIA con la arquitectura CUDA cuentan con una jerarquía de memoria en la cual los accesos deben ser definidos explícitamente (salvo a los registros y la memoria local); esto es, el programador debe indicar qué tipo de memoria desea utilizar en cada momento. En la Figura 1.3 se presenta un diagrama con los tipos de memoria y su distribución entre las unidades de ejecución. A continuación se ofrece una breve descripción de las diferentes memorias.

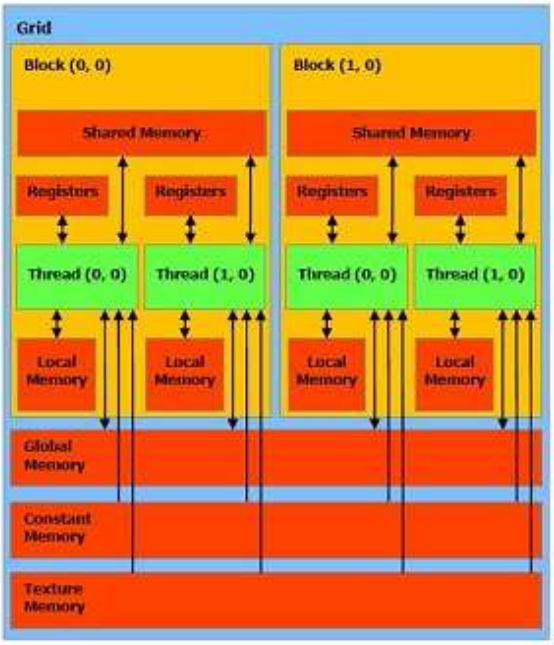

Figura 1.3: Jerarquía de memoria en la arquitectura CUDA. Extraída de [141].

La **memoria global** es común a todos los bloques de hilos que se ejecutan dentro de una tarjeta gráfica. Es la memoria de mayor capacidad y los datos almacenados pueden persistir durante la ejecución de múltiples *kernels*. La memoria global es la más lenta y no posee caché (en tarjetas

[2]Pero sí pueden acceder a memoria global.

de *compute capability* menores a 2.0, este concepto se describe en el Anexo A). Es la única memoria que es visible desde el *host* y permite la escritura desde GPU, lo que la convierte en la única posibilidad para transferir los resultados computados en la GPU a la CPU. Se puede obtener un mejor rendimiento de esta memoria si se emplea la técnica *coalescing*, que implica que hilos consecutivos accedan a posiciones de memoria consecutivas, realizando de esta forma múltiples accesos concurrentes (las características y restricciones del acceso *coalesced* se detallan en el Anexo A).

La **memoria compartida** es una memoria a la cual solo pueden acceder los hilos que corren dentro un mismo bloque. Su capacidad es menor a la memoria global y su ciclo de vida es de una ejecución de *kernel*. La memoria compartida sí posee caché y el costo de acceso es del orden de 100 veces menor que el acceso a memorias globales o locales.

Los **registros** son privados de cada hilo. Es la memoria más rápida pero su tamaño es muy limitado y su uso es determinado por el compilador.

La **memoria local** es privada a cada hilo. Tiene la misma velocidad que la memoria global y su ciclo de vida es de una ejecución de *kernel*. Esta memoria se usa como extensión de los registros.

La **memoria constante** es accesible por todos los hilos. Es de solo lectura para la GPU, pero la CPU puede escribir en ella. Reside en memoria de dispositivo por lo que tiene el mismo tiempo de acceso que la memoria global, pero esta memoria sí posee caché, logrando disminuir los tiempos de acceso en caso de producirse un acierto de caché (caché *hit*).

La **memoria de texturas** es accesible por todos los hilos. Está optimizada para aprovechar la localidad espacial de datos en dos dimensiones y posee caché, pero las escrituras a memoria de textura no se actualizan hasta finalizar la ejecución del *kernel* en curso, por lo cual no se puede acceder a través de memoria de texturas a un dato en el mismo *kernel* una vez que fue modificado.

ALN con GPUs

El tipo de operaciones para el cual fueron concebidas originalmente las tarjetas gráficas, en principio la visualización de gráficos en pantalla, implica en muchos casos resolver operaciones básicas de álgebra, sin embargo esta sección se centra en relevar el uso de las GPUs para la resolución de núcleos de ALN y no como parte del *pipeline* gráfico.

Al comienzo de la década del 2000 se presentaron las primeras aplicaciones de uso de tarjetas gráficas para la resolución de problemas de ALN. Estos trabajos, cargados de ingeniosidad, en muchos casos estaban limitados por las particularidades de la arquitectura de las GPUs y su diseño específico, así como por sus restricciones en cuanto a capacidad de programación. Muchas de estas limitaciones fueron eliminadas o paliadas con la aparición de CUDA y la biblioteca CUBLAS. Estas herramientas han permitido que, en los últimos años, se desarrollaran diversos trabajos que aceleran la resolución de núcleos de ALN, con prestaciones muy superiores a las conseguidas con hardware tradicional de HPC, avanzando así, de ser un área de trabajo artesanal y experimental, a una realidad pujante. A continuación se describen los principales esfuerzos presentados en los últimos 10 años.

El trabajo de Larsen y McAllister [149] podría definirse como el pionero en el área. En el artículo los autores muestran cómo realizar la multiplicación de matrices trabajando con números en la precisión disponible en las GPUs de la época, 8 bits. El algoritmo desarrollado es aplicado a la resolución del problema de búsqueda de componentes conexas sobre matrices de adyacencia. Además de las ingeniosas ideas, el trabajo muestra una gran visión de futuro, ya que los autores predicen la disponibilidad de tarjetas que manejen números en coma flotante con simple precisión (32 bits), algo que en aquella época parecía poco razonable. También en el año 2001, Rumpf y Strzodka [191] utilizan una GPU para resolver sistemas lineales con el método de Jacobi; el marco del trabajo es la resolución de ecuaciones diferenciales mediante discretizaciones con el método de

los elementos finitos (MEF). Los autores muestran cómo sortear diversas dificultades, en particular el uso de sólo 8 bits de precisión y el rango de representación ([0–1]). Otro trabajo destacado de esta primera etapa es el presentado por Bolz et al. [60]. Los autores implementan distintos métodos iterativos de resolución de sistemas lineales, mostrando estrategias para implementar los métodos multigrid, gradiente conjugado (GC) y estrategias combinadas tanto para matrices dispersas como para densas. La implementación del método del GC para matrices dispersas se basa en el producto de una matriz dispersa por un vector (SPMV, del inglés *Sparse Matrix Vector*) sobre GPU. El mismo método, GC, fue implementado por Hillesland et al. [125] sobre Direct X 9.0 y utilizando tarjetas ATI en el contexto de problemas de luminosidad. También es destacable el trabajo de Goodnight et al. [100] donde se resuelven problemas de frontera; en particular, implementando el método de multigrid utilizando los lenguajes Cg y pixel assembler. También en esta primera etapa de desarrollo de GPGPU para ALN se presentó el trabajo de Bajaj et al. [33] donde los autores emplean las GPUs para la resolución de un problema de ecuaciones diferenciales mediante el método de Runge-Kutta de cuarto orden, y emplean como solver de los sistemas lineales el método de Gauss-Seidel. Además, resuelven la ecuación de Poisson resultante de integrar las ecuaciones de Navier-Stokes utilizando el método de Jacobi a bloques sobre una grilla de 256.000 x 256.000. Por último, destaca el artículo de Hall et al. [119] que extiende el trabajo sobre multiplicación de matrices de Larsen y McAllister. Los autores implementan la multiplicación de matrices con una estrategia de múltiples pasadas utilizando los cuatro canales (RGBA) para cargar las matrices a multiplicar en sub-matrices de 2×2 (cada coeficiente en un canal).

Los principales aportes en la segunda etapa del desarrollo de GPGPU para abordar problemas de ALN están caracterizados por la propuesta y desarrollo de diferentes herramientas, estándares y métodos de transformación automática de códigos. Algunos enfoques relevantes son Cg, HLSL y OpenGL Shading, pero todos ellos muy relacionados al cauce gráfico. Con un grado mayor de abstracción destaca Brook, donde se propone ver las GPUs como un procesador de *streaming*. También en el *framework* presentado por Microsoft, Accelerator [73], se propone un lenguaje de alto nivel para trabajar con GPUs, se codifica en C# y se traduce a pixel shader empleando una estrategia de paralelismo a nivel de vector. Además, es necesario también citar la arquitectura propuesta por ATI [182] tendente a explotar el paralelismo de datos basados en una máquina virtual. En el mismo sentido se focaliza el artículo de Gumerov et al. [114] que estudian acceder a las GPUs desde programas codificados en Fortran, y Brodtkovb et al. [63] que presentan ideas similares para Matlab. Otros trabajos relacionados son los de Govindroj et al. [102, 103], donde los autores estudian y presentan un modelo teórico de la memoria de las GPUs y estrategias para explotar esta información.

Desde el año 2007, y con el lanzamiento de CUDA por parte de NVIDIA, la mayoría de los trabajos se han centrado en el uso de dicha tecnología, siendo casi un estándar en los trabajos recientes. A partir de 2009 también se puede apreciar un aumento en la presentación de trabajos usando OpenCL [112], un esfuerzo por crear un estándar que permita desarrollar software válido para GPUs de diferentes fabricantes. Además, esta etapa está caracterizada por la presentación de varios trabajos que consiguen desempeños computacionales importantes, superando ampliamente algoritmos consolidados sobre plataformas tradicionales de HPC; un caso paradigmático de esta etapa es el presentado por Volkov y Demmel [218].

El principal relevamiento sobre utilización de GPUs para la resolución de problemas generales durante las dos primeras etapas del desarrollo es el realizado por Owen et al. [180], extendido posteriormente en el artículo [179]. El trabajo abarca un estudio de la historia de GPGPU, modelos de programación, relevamiento de herramientas de desarrollo y depuración de aplicaciones. Se resumen implementaciones de operaciones básicas como *map, reduce, gather – scatter, stream filtering, sort* y *search*; también se describen las distintas propuestas de estructuras de datos tales como vectores,

matrices, matrices dispersas (estáticas y dinámicas); y aplicaciones genéricas como la resolución de ecuaciones diferenciales, operaciones de ALN, consultas en bases de datos, procesamiento de imágenes y señales, iluminación global y geometría computacional.

En lo que resta de la sección se relevan las principales aportaciones en el área de aceleración de operaciones de ALN utilizando GPUs, agrupando los trabajos por su temática.

Operaciones tipo BLAS

Como se mostró en el Apartado 1.2.2, en ALN es habitual el uso de la especificación BLAS para desarrollar aplicaciones y permitir así acelerar el cómputo de problemas complejos en forma sencilla, legible y portable. En este sentido, gran parte del esfuerzo inicial para acelerar núcleos de ALN con GPU, hasta la presentación en 2007 de CUBLAS por parte de NVIDIA, consistió en la implementación de operaciones incluidas en la especificación BLAS. En esta categoría destacan, además de los trabajos pioneros dedicados a la multiplicación de matrices y el de Dajaj et al. para resolver expresiones lineales, los trabajos que se presentan a continuación. Fatahalian et al. [82] extiende los trabajos en [119, 149] sobre la multiplicación de matrices en GPU; el trabajo se centra en mostrar las restricciones de las GPU del momento, en particular las referentes al ancho de banda en las trasferencias CPU-GPU. Un poco más audaz es el trabajo resumido en el artículo de Krüger et al. [144], uno de los primeros trabajos en delinear el desarrollo de núcleos al estilo BLAS, en contraposición con todo lo realizado anteriormente, donde se desarrollaban soluciones en GPU particulares para cada problema. Los autores implementan funciones para operar con vectores, el producto matriz-vector y funciones de reducción mediante la estrategia multipasada. Utilizan Pixel Shader 2.0 sobre una tarjeta ATI Radeon 9700, los vectores son almacenados en memoria de texturas 2D, organizando las matrices completas por diagonales. El trabajo también ataca problemas con matrices dispersas. Primero, estudiando el uso de matrices banda para las cuales emplean la misma idea que para las matrices densas. Segundo para matrices dispersas no estructuradas, utilizando un formato comprimido. Implementan los métodos del GC y Gauss-Seidel para validar la propuesta. Posteriormente, en 2005, los mismos autores extienden el trabajo [145] utilizando DirectX 9.0 como herramienta de desarrollo y cambian el formato de almacenamiento de las matrices. Las matrices dispersas no estructuradas son almacenadas con una estrategia de bloques con 4 elementos por fila que posee ciertas similitudes con el formato comprimido a bloque y *jagged diagonal*. Validan los desarrollos implementando el método del GC para resolver la ecuación de Poisson y ecuaciones diferenciales mediante un esquema de Crank-Nicholson. Otro autor que estudió de forma temprana el desarrollo de núcleos tipo BLAS sobre GPU es el de Moravánsky [162]. En el trabajo se propone utilizar la memoria de texturas para almacenar las matrices, manejando los cuatro canales de colores (RGBA) para generar bloques de 4×1. Emplea DirectX 9.0 y Shader para implementar las rutinas. Presenta algunas implementaciones, como las rutinas para el producto interno de vectores, localizar el elemento de mayor valor y la multiplicación de matrices. Validan la propuesta aplicando las rutinas desarrolladas para implementar el método del GC en contextos de optimización lineal.

Tras la aparición de la primera versión de la biblioteca CUBLAS, muchos trabajos se centraron en presentar mejoras sobre las rutinas desarrolladas por NVIDIA, algunos de los cuales han sido incluidos en posteriores versiones de la biblioteca. En este sentido, destaca el trabajo de Barrachina et al. [34] donde los autores evalúan el desempeño de rutinas de CUBLAS, y posteriormente muestran cómo conseguir mejoras en base a diferentes estrategias, inteligentes particionamientos, implementaciones híbridas de la multiplicación de matrices (GEMM), modificaciones de la estructura del núcleo de resolución de sistemas lineales triangulares (TRSM), y en especial la inclusión de estrategias de *padding*. También Volkov y Demmel [218] presentan estrategias para la aceleración de la multiplicación de matrices. Los autores validan sus desarrollos acelerando los métodos de

factorización LU y QR; su rutina fue incluida en la versión 2.1 de CUBLAS. Basándose en el trabajo de Volkov y Demmel, L. Chien [69] mostró cómo alcanzar mejoras (entre un 10 % y un 15 %) sobre la arquitectura Tesla; la propuesta se basó en modificar el código ensamblado de las rutinas. Más cercano en el tiempo, Nath et al. [169] presentaron una versión optimizada de la multiplicación de matrices para la arquitectura Fermi; esta propuesta fue incluida en la versión 3.2 de CUBLAS. Otras operaciones presentes en CUBLAS también han sido abordadas por la comunidad, por ejemplo el producto matriz-vector es estudiado en el trabajo de Fujimoto [88], que presenta una implementación de la rutina SGEMV hasta 15 veces más rápida que la disponible en CUBLAS 1.1 en una tarjeta de NVIDIA 8800 GTX. El artículo incluye el código de la rutina propuesta. Con el mismo objetivo, Nath et al. [167] muestran la aceleración de diversos núcleos básicos mediante el uso de la la biblioteca MAGMA [7].

Los veloces avances en el hardware y la gran dependencia de los métodos desarrollados respecto a la arquitectura subyacente motivaron a varios autores a estudiar técnicas de autoconfiguración–optimización. En esta línea se centra el trabajo de Jiang y Skir [134], que presentan un *framework* para la operación de multiplicación de matrices aplicando ideas similares a las ofrecidas en la biblioteca ATLAS [221] (una implementación de BLAS que se autoconfigura, adaptando su funcionamiento a las características de la plataforma en la que se ejecuta). El *framework* intenta optimizar, entre otros, los parámetros de *multi-render-targets*, empaquetado de datos, número de pasadas y de bifurcación. Ideas similares de autoconfiguración para la operación GEMM fueron posteriormente expuestas por Li et al. en [155]. En el mismo sentido, pero para el núcleo de resolución de sistemas triangulares, destaca el artículo de Du et al. [80].

Otra línea de investigación importante la constituye el uso híbrido CPU-GPU para resolver operaciones básicas de ALN. Uno de los primeros trabajos en esa línea es el de Ohsblima et al. [178]. Discuten primero un marco teórico (muy simple) para estimar el tiempo de ejecución en CPU y GPU, luego utilizan la herramienta propuesta para dividir los problemas y planificar su ejecución (parte en CPU y parte en GPU) de forma que se minimice el tiempo total de ejecución. Utilizan como caso de prueba la rutina GEMM de BLAS para la multiplicación de matrices, aplicando un particionado sobre una de las matrices. Fatica [83] extiende estas ideas para desarrollar una versión del benchmark LINPACK que permite incluir GPUs como elementos de cómputo.

Gran parte del avance en lo que respecta a la implementación de BLAS sobre GPU se compendia en el Capítulo 4 del libro [168].

Factorización LU y operaciones relacionadas

En el trabajo de Galoppo et al. [90] se puede encontrar un buen relevamiento de los primeros trabajos de GPU y *stream programming* para computación científica; además se exponen relaciones de precios y capacidades de cómputo de distintas GPUs y CPUs; también se revisan aspectos de arquitectura de las GPUs y su influencia en los cálculos. Sin embargo, el principal aporte del trabajo es la implementación (una de las pioneras) de métodos para la resolución de sistemas lineales densos, en particular se presentan los métodos de Gauss-Jordan, factorización LU y factorización LU con pivotamiento parcial y completo. Los autores ofrecen un conjunto amplio de pruebas sobre los algoritmos y estudian la influencia de las características de las GPUs en el desempeño de los métodos. Agunas de las características objeto de estudio son el ancho de banda y los tamaños de las memorias. En el mismo año, Ino et al. [130] presentan una implementación de la factorización LU en GPU utilizando estrategias *right–looking*. Posteriormente, Jung y O'Leary [135, 136] presentan una implementación de la factorización de Cholesky sobre una GPU. Los autores proponen almacenar las matrices (simétricas) en un formato comprimido rectangular propuesto por Gunnel y Gustavson [115]. Además aplican su rutina de factorización a la resolución de problemas de optimización

mediante el método del punto interior.

Posterior a la presentación de CUBLAS destaca el trabajo de Ries et al. [189], donde se estudia la inversión de matrices triangulares sobre GPU. Esta operación es necesaria para computar la matriz inversa mediante la factorización LU. También sobre esta operación se centra el trabajo de Fabregat y Bientinesi [81]. Implementaciones eficientes de la factorización LU y la factorización de Cholesky también fueron presentadas por Barrachina et al. [35, 37]. En el estudio se incluye la evaluación de estrategias de cómputo mixto GPU+CPU, *padding* y precisión mixta con refinamiento iterativo. Ideas similares son estudiadas en los trabajos de Baboulin et al. [30, 207, 210]; en estos trabajos se propone acelerar la factorización LU modificando el pivotamiento por un método que implica menor costo computacional (aunque sufre de ciertos problemas de estabilidad). Finalmente, los autores extienden los trabajos para acelerar la factorización de Cholesky mediante el uso de planificación dinámica [157]. Posteriormente, Tomov et al. [208, 209] mostraron implementaciones de la factorización de Cholesky, utilizando la rutina para el producto de matrices desarrollada por Volkov y Demmel, que alcanzaban 160 GFLOPS. Adicionalmente, Volkov y Demmel [219] extienden su trabajo previo presentando implementaciones de la factorización LU basadas en una implementación optimizada de la función GEMM alcanzando los 180 GFLOPS sobre una tarjeta NVIDIA GTX 8800. Además es uno de los primeros trabajos en aprovechar dos GPUs conectadas a un mismo *host* para realizar la factorización LU. En la misma área de trabajo se centra el trabajo de Ltaief et al. [157].

Utilizando otro enfoque, Marqués et al. [158] muestran como explotar el uso de GPUs para la resolución de sistemas lineales en métodos que incluyen técnicas de computación *out–of–core*.

Descomposiciones QR y SVD

Además de los trabajos [30, 90, 218] que abordan la factorización QR como campo de prueba de implementaciones de operaciones básicas de BLAS, Kerr et al. [140] implementan la factorización QR en GPU. En el trabajo los autores evalúan los métodos de Gram-Schmidt modificado, Givens, Householder y Householder a bloques desde el punto de vista de capacidad de ejecución en paralelo. El cuello de botella de la implementación es el producto matriz-vector, por lo cual desarrollan una implementación de dicha operación más eficiente (para esta área de aplicación) que la disponible en CUBLAS 2.0. La propuesta fue validada resolviendo problemas de mínimos cuadrados sobre tarjetas NVIDIA 9800 GX2 y GTX 280.

Otro trabajo interesante es el de Bondhugula et al. [61], donde se propone una estrategia mixta para la SVD, donde la primera etapa de bidiagonalización se realiza mediante transformaciones de Householder en GPU; posteriormente la diagonalización de la matriz bidiagonal se computa en la CPU mediante el método de *shifted*-QR.

Para el caso particular de las matrices tridiagonales simétricas, Lessig et al. [153] presenta la implementación con CUDA del método de bisección (utilizando la secuencia de Sturm [117]). No obstante, el algoritmo presentado tiene ciertas restricciones (por ejemplo en los tamaños de las matrices).

En el mismo ámbito destaca el trabajo de Andreecut [23] que implementa métodos en GPU para evaluar el análisis de componentes principales basado en ortogonalizaciones de Gram–Schmidt y la SVD truncada, así como el de Tomov et al. [209] sobre el uso de una GPU para la reducción de Hessenberg mediante un algoritmo híbrido. Estos procesos son de gran utilidad para el cómputo de valores propios y para el cálculo de la SVD. Los autores utilizan transformaciones WY y versiones optimizadas de la rutina GEMV para alcanzar importantes mejoras en el tiempo de ejecución de la transformación al compararse con la rutina disponible en LAPACK sobre CPU.

Estrategias de precisión mixta

El uso de técnicas de precisión mixta en algoritmos numéricos es un tópico largamente estudiado en el pasado, pero la relación en el costo de resolver los problemas con aritmética de simple y doble precisión en los procesadores modernos, actualmente en el entorno de 2×, restó interés a estas estrategias. Sin embargo, con la difusión de las GPUs, que en un principio no manipulaban números en doble precisión y posteriormente ofrecían aritmética en doble precisión pero a un costo de entre 8× y 10×, diversos investigadores retomaron el tema del uso de técnicas de precisión mixta, tanto para alcanzar doble precisión como para acelerar las aplicaciones sobre GPU. En la actualidad, este factor en las GPUs de última generación se ha reducido nuevamente a 2×; no obstante, el trabajo con simple precisión sigue siendo interesante ya que permiten paliar parcialmente el problema de la limitación de memoria en las GPUs, reduce considerablemente el tiempo de comunicación entre el *host* y la GPU y, un factor 2×, se puede traducir en cientos de GFLOPS.

Uno de los trabajos pioneros es el de Göeddeke et al. [96], que muestra un ejemplo de utilización de las GPUs para resolver sistemas lineales, así como diferentes estrategias para mejorar el desempeño de la aplicación. Se trabaja con el método de Jacobi para la resolución de un sistema lineal con 9 bandas. Posteriormente, en el trabajo de los mismos autores [97] se comparan implementaciones de algoritmos utilizando doble precisión nativa, emulada (empleando dos números de precisión simple) y técnicas de precisión mixta utilizando refinamiento iterativo. Los autores realizan propuestas generales pero en el artículo se trabaja sobre los métodos GC y multigrid para la resolución de sistemas lineales. En el trabajo se puede encontrar una excelente introducción a los temas de computación científica utilizando precisión emulada y mixta. Los autores exponen un *framework* para trabajar con precisión mixta, que validan con diversos resultados, tanto sobre GPUs como sobre FPGAs. A continuación, Göddeke y Strzodka presentan [94] una extensión del trabajo previo para estudiar el efecto de emplear diferentes precisiones, incluyendo doble precisión nativa de una GPU NVIDIA GTX 280. Utilizan las métricas de GFLOPS/Watt y GFLOPS/Euro para evaluar los resultados obtenidos. Trabajan sobre los métodos del GC y multigrid para la resolución de un problema abordado con el MEF, donde los sistemas lineales a resolver son de 9 bandas con 1 millón de incógnitas. Basándose en los trabajos anteriores, los autores desarrollan una herramienta para resolver ecuaciones diferenciales utilizando discretizaciones con el MEF explotando paralelismo de dos niveles [95], aprovechan un cluster donde se hace descomposición de dominio y en cada nodo se utiliza el paralelismo de la GPU para resolver cada subdominio mediante el método multigrid.

Otros trabajos ya mencionados que abordan la temática de precisión mixta son el de Barrachina et al. [34], en el que los autores parten de los resultados de un trabajo previo para generar rutinas que computan las factorizaciones de Cholesky y LU y comparan los resultados obtenidos al trabajar con aritmética de doble precisión con los alcanzados al operar en simple precisión y aumentar la precisión de la respuesta aplicando métodos de refinamiento iterativo. También el de Baboulin et al. [30], que aborda el uso de precisión mixta sobre las factorizaciones LU y de Cholesky y el método iterativo FGMRES-GMRES, pero en este caso la arquitectura de destino es un procesador Cell.

Enfoques diferentes evaluados recientemente son el de Anzt et al. [27] que estudia la relación entre el uso de precisión mixta y el consumo energético para la ejecución de métodos iterativos de resolución de sistemas lineales, así como, el aporte de Daichi y Mukunoki [164] donde se estudia la emulación por software de operaciones de BLAS con precisión cuádruple.

Uso de múltiples GPUs

Como se mencionó anteriormente, Volkov y Demmel presentaron posiblemente el primer trabajo sobre el uso de más de una GPU conectada a un *host* para la resolución de núcleos de ALN

experimentando de forma preliminar con el uso de dos GPUs para la factorización LU.

En cuanto al uso sistemático de múltiples GPUs, los trabajos pioneros son el de Castillo et al. [68], donde se presenta una extensión del *framework* FLAME [116] buscando mejorar las prestaciones utilizando una planificación (*sheduling*) dinámica para la resolución de problemas de ALN densos, incluyendo GPUs entre las unidades de procesamiento. En particular, los autores reportan resultados para la factorización de Cholesky sobre cuatro GPUs NVIDIA (una NVIDIA Tesla S870), mostrando desempeños de 180 GFLOPS al resolver sistemas de dimensión en el entorno de 10.000. El trabajo fue extendido en el artículo de Quintana et al. [187] alcanzando desempeños de 424 GFLOPS para el mismo problema. Posteriormente, Ayguadé et al. [29] presentaron una idea similar pero sobre el modelo StarSs. También relacionado está el esfuerzo de Fogué et al. [86] que extienden la biblioteca PLAPACK para utilizar GPUs como parte de la plataforma destino, implementan la factorización de Cholesky sobre un cluster con 16 nodos (intel Xeon) con 32 GPUs (NVIDIA FX 5800) superando los 3.000 GFLOPS.

Basados en PLASMA y MAGMA, Ltaief et al. [157] estudiaron estrategias híbridas sobre computadoras multiprocesador y GPUs para implementar la factorización Cholesky y alcanzan desempeños superiores al TeraFLOP por segundo. Luego, Agullo et al. [19] extienden trabajos previos presentando la factorización LU. El trabajo se centra en mejorar el núcleo híbrido de factorización LU y el efecto de las arquitecturas Fermi. En particular, aceleran la factorización utilizando un método de resolución de sistemas triangulares que emplea técnicas que no han demostrado estabilidad numérica. Los mismos autores, extienden el estudio del despacho dinámico en el trabajo [18].

Otros trabajos relacionados y que emplean múltiples GPUs son el de Jang et al. [133] para la reconstrucción de tomografías; Neic et al. [170] sobre métodos multigrid algebraicos para la resolución de sistemas lineales en clusters con nodos que incluyen GPUs; Griebel y Zaspel [108] que utilizan múltiples GPUs para la resolución de las ecuaciones de Navier-Stokes incompresible; y el artículo de Diamos y Yalamachili [77] que presenta el estudio de ejecución de núcleos en forma especulativa sobre múltiples GPUs.

Otras aplicaciones

Además de los trabajos descritos anteriormente, se han presentado diversos artículos con trabajos relacionados con la aceleración de operaciones de ALN o de computación científica en GPU. Algunos de los más destacados se mencionan a continuación.

En el trabajo de Brodtkorb [63], se presenta una interfaz para utilizar la GPU como coprocesador matemático para MatLab. Se propone utilizar planificación dinámica, con múltiples hilos de ejecución y colas para administrar las tareas. Implementan los métodos de multiplicación de matrices, eliminación de Gauss-Jordan, factorización LU con pivotamiento y resolución de sistemas lineales tridiagonales. Utilizan OpenGL y explotan la capacidad de multicanal para almacenar las matrices en bloques de 2×2. El trabajo es una síntesis de la tesis de maestría desarrollada por Brodtkorb [62]. También el esfuerzo de Barrachina et al. [36] para el desarrollo de una API que permite invocar operaciones de ALN mediante un lenguaje de *script* de tipo Matlab. Otro esfuerzo es el de Stratlon y Stone [201], que presentan un *framework*, MCUDA, para transferir automáticamente códigos desarrollados en CUDA a contextos de CPUs multiprocesador de memoria compartida.

El trabajo de Lefohn et al. [152] ofrece un interesante estudio del acceso a diferentes tipos de datos en GPU. Las optimizaciones de códigos en GPUs son estudiadas por Ryo et al. [193], evalúan la utilización de registros, tamaños de bloques de hilos, etc. Experimentan sobre los problemas de multiplicación de matrices densas, imágenes de resonancia magnéticas (MRI) y suma de diferencias absolutas. Proponen como trabajo futuro el desarrollo de herramientas automáticas de optimización

de código. Adicionalmente, los mismos autores extienden el estudio [192] para evaluar las capacidades de CUDA sobre una tarjeta GTX 8800. También presentan un relevamiento de herramientas para desarrollar aplicaciones sobre GPU.

Bauluet et al. [65] resuelven en su trabajo el problema de *all-pairs shortest-path* basándose en la multiplicación de matrices, y obtienen resultados muy buenos al compararlos con algoritmos de referencia en el área. Realizan una completa evaluación de los tiempos de cómputo, costos económicos y consumo energético. En este sentido incluyen ideas de recursión y la versión de Volkov y Demmel de la operación GEMM [218].

Algunos trabajos que evalúan las transformaciones rápidas de Fourier (FFT, del inglés *fast Fourier trasnform*) sobre GPU son los aportes de Moreland y Anglel [163], Nukada et al. [172] y Nukada y Matsuoka [171].

Por último, destacar el trabajo de Michalakes and Vachharajani [159], en el que se estudia transformar parte de un modelo numérico de fluidos muy utilizado (el WRF [15] del *National Center for Atmospheric Research*) para ser ejecutado sobre una GPU utilizando CUDA. Plantean la traducción automática de código Fortran a código CUDA, aportando además resultados prometedores.

Resumen del relevamiento

En los últimos años el avance en el uso de las GPUs para acelerar la resolución de núcleos de ALN ha sido relevante, principalmente en el caso de operaciones de ALN densa para las cuales la arquitectura de las GPUs se adapta mejor que para matrices dispersas. Esto ha permitido que actualmente se disponga de rutinas para GPUs que alcanzan desempeños competitivos, y muchas veces superiores a los obtenidos sobre otras arquitectura de HPC.

Como resultado de los trabajos científicos relevados, los desarrollos formales de bibliotecas o herramientas que exploten las GPUs para la resolución de núcleos básicos de ALN del tipo BLAS y LAPACK, además de la ofrecida por NVIDIA en CUBLAS (que disponen de implementaciones de la mayoría de las operaciones de la especificación BLAS), se pueden agrupar en torno a tres grandes proyectos:

- FLAME [116]: se extiende el paquete original con implementaciones de diversas rutinas de BLAS, LAPACK y otras para la resolución de problemas de álgebra matricial para ejecutar en GPU.

- MAGMA [7]: ofrece implementaciones de operaciones de BLAS y LAPACK en GPU que utilizan estrategias híbridas de cómputo, transferencias sincrónas y asincrónas y uso de memoria no paginable.

- CULA [129, 6]: se dispone de implementaciones de algunos de los núcleos de la especificación LAPACK sobre GPU así como algunas rutinas de BLAS, utilizando estrategias híbridas.

Destacar que en el relevamiento realizado no se localizaron propuestas de uso de GPUs en el campo de reducción de modelos ni para la inversión de matrices, operación básica para el cálculo de la función signo matricial.

1.3. Entorno de experimentación

Algunas consideraciones generales de los experimentos realizados en el presente trabajo son:

- Todos los resultados presentados corresponden a experimentos que se realizaron utilizando aritmética de coma flotante IEEE. En la mayoría de los casos se utilizan números en simple precisión aunque en otros pocos se evalúa preliminarmente el uso de aritmética en doble precisión sobre GPUs.
- En las rutinas que usan la GPU los costos de transferencia están incluidos en los tiempos de ejecución y el cálculo del desempeño asociado.
- En los distintos experimentos presentados se evaluaron matrices con dimensión entre 1.000 y 64.000.
- En los algoritmos por bloques se ha experimentado con tamaños de bloque entre 16 y 1.024, pero únicamente se muestran los resultados correspondientes al tamaño de bloque que ha reportado los mejores desempeños.

1.3.1. Plataformas de ejecución

Durante el trabajo se utilizaron distintas plataformas hardware para evaluar los códigos desarrollados; a continuación se enumeran las máquinas empleadas y sus principales características.

- HARRISON

  La plataforma está formada por un computador con procesador Pentium DualCore E5200 a 2.50 GHz conectado mediante un bus PCI-Express a una tarjeta gráfica NVIDIA 9800 GTX+. La Tabla 1.2 ofrece más detalles de este equipo.

Procesadores	#cores	Frecuencia (MHz)	L2 caché (MB)	Memoria (MB)
Pentium DualCore E5200	2	2500	2	2048
NVIDIA 9800 GTX+	128	738	–	512

Tabla 1.2: Características de la plataforma HARRISON.

  Se dispone de la biblioteca MKL versión 10.2.1 como implementación de BLAS y LAPACK en CPU y la versión 2.1 de la biblioteca CUBLAS para GPU.

- PECO

  El equipo cuenta con dos procesadores Intel Xeon QuadCore conectados mediante un bus PCI-Express a una GPU Tesla C1060 de NVIDIA. La Tabla 1.3 ofrece más detalles de esta plataforma.

Procesadores	#cores	Frecuencia (GHz)	L2 caché (MB)	Memoria (GB)
Intel Xeon QuadCore E5405	(4×2)8	2.3	12	24
NVIDIA TESLA C1060	240	1.3	–	4

Tabla 1.3: Características de la plataforma PECO.

  Se utiliza la versión 10.1 de la biblioteca MKL de Intel, que implementa BLAS y LAPACK en versiones multihilo para procesadores de propósito general; mientras que en la GPU se utiliza

la implementación CUBLAS (versión 2.1) desarrollada por NVIDIA de la especificación BLAS. Salvo mención explícita, en el procesador de propósito general se utilizan 8 hilos, uno por cada núcleo disponible en la arquitectura.

- PRODAN[3]

  Esta plataforma está compuesta por 2 procesadores QuadCore a 2.27 GHz y 48 GB de memoria RAM, conectado mediante buses PCI a cuatro GPUs Tesla C1060. La Tabla 1.4 extiende los detalles de esta plataforma.

Procesadores	#proc.	#cores (por proc.)	Frecuencia (GHz)	L2 caché (MB)	Memoria (GB)
Intel Xeon QuadCore E5530	2	4	2.27	8	48
NVIDIA TESLA C1060	4	240	1.3	–	(4x4)16

Tabla 1.4: Características de la plataforma PRODAN.

En los procesadores de propósito general se utiliza la versión 11.1 de la biblioteca MKL como implementación multihilo de las especificaciones BLAS y LAPACK, mientras que sobre las GPU se utiliza la biblioteca CUBLAS versión 3.0.

- ZAPE

  Consiste en un procesador AMD 9550 (Phenom) QuadCore a 2.2 GHz conectado mediante un bus PCI-Express a una tarjeta gráfica NVIDIA GTX480 (con arquitectura Fermi). La Tabla 1.5 extiende los detalles de la plataforma de hardware.

Procesadores	#cores	Frecuencia (GHz)	L2 caché (MB)	Memoria (GB)
AMD Phenom QuadCore 9550	4	2.2	0.5	4
NVIDIA GTX 480	480	1.4	–	1.5

Tabla 1.5: Características de la plataforma ZAPE.

En esta plataforma se encuentran disponibles la implementación de Intel de las especificiones BLAS y LAPACK, Intel MKL en su versión 10.1, y las bibliotecas de ALN para GPU NVIDIA CUBLAS (versión 3.0.14) y MAGMA (versión 0.2).

- PECO-II

  El equipo cuenta con dos procesadores Intel Xeon QuadCore conectados mediante un bus PCI-Express a una GPU C2050 de NVIDIA. La Tabla 1.6 ofrece más detalles de esta plataforma.

  Se utiliza la versión 10.1 de la biblioteca MKL de Intel, que implementa BLAS y LAPACK en versiones multihilo para procesadores de propósito general; mientras que en la GPU se utiliza la implementación CUBLAS (versión 3.1) desarrollada por NVIDIA de la especificación BLAS.

[3]Equipo financiado por la Comisión Sectorial de Investigación Científica (CSIC), Universidad de la República, Uruguay.

Procesadores	#cores	Frecuencia (GHz)	L2 caché (MB)	Memoria (GB)
Intel Xeon QuadCore E5405	(4×2)8	2.3	12	24
NVIDIA TESLA C2050	448	1.15	–	3

Tabla 1.6: Características de la plataforma PECO-II.

- YUCA

  El equipo incluye cuatro procesadores Intel Xeon X7550 con 8 cores por procesador (un total de 32 cores) a 2.0 GHz conectados mediante un bus PCI-Express 2.0 a una tarjeta NVIDIA C2050 que posee un total de 448 cores. En la Tabla 1.7 se pueden encontrar más detalles de la plataforma.

Procesadores	#cores	Frecuencia (GHz)	L2 caché (MB)	Memoria (GB)
Intel Xeon X7550	(8×4) 32	2.0	18	124
NVIDIA C2050	448	1.15	–	6

Tabla 1.7: Características de la plataforma YUCA.

Se utilizan los núcleos computacionales de la biblioteca Intel MKL 11.0 en su versión multihilo como implementación de las especificaciones BLAS y LAPACK, mientras que sobre las GPUs se utiliza la biblioteca NVIDIA CUBLAS 3.2.

Capítulo 2

El problema de Reducción de Modelos

Como se mencionó en el Capítulo 1, diversos fenómenos físicos se pueden modelar matemáticamente mediante ecuaciones diferenciales derivando, en algunos casos, en sistemas dinámicos. En la fabricación de un dispositivo electrónico esto permite, por ejemplo, estudiar el efecto de diferentes diseños sin la necesidad de construir realmente el dispositivo. Esta metodología de trabajo simplifica de forma importante los procesos de optimización y diseño, y por ende, disminuye los costos asociados a estos procesos. La resolución de esta clase de modelos matemáticos generalmente necesita un tratamiento numérico utilizando una computadora. Además, a medida que se tratan problemas de dimensiones mayores, las necesidades de cómputo crecen notoriamente. Si bien, en los últimos años el desarrollo de la computación ha sido vertiginoso, el interés por resolver problemas cada vez más complejos, de mayor dimensión y con más precisión suele superar con creces el poder de cómputo ofrecido por las computadoras modernas. El interés de contar con modelos matemáticos que permitan realizar simulaciones, evaluar posibles diseños, estudiar impactos, etc., y a su vez la necesidad que estos modelos sean tratables en un tiempo aceptable, da origen al campo de trabajo de la reducción de modelos. Estas técnicas buscan, dado un modelo matemático, encontrar otro cuya dimensión sea considerablemente menor pero que describa el fenómeno modelado con la precisión similar a la del modelo original. De esta forma, es posible utilizar el modelo reducido en posteriores estudios.

Los métodos de reducción de modelos más populares se pueden agrupar en dos grandes familias: los métodos de aproximación basados en descomposiciones en valores singulares (SVD) [154, 184] y los de aproximación por subespacios de Krylov [87, 127, 126, 132, 194]. Este trabajo se centra en los métodos basados en la SVD, ya que garantizan la preservación de importantes propiedades del sistema original como la estabilidad y la pasividad [24]. Además, este tipo de métodos proporciona una cota del error introducido por el modelo reducido, lo cual permite desarrollar métodos adaptativos, que reducen la dimensión del sistema en función de un margen de error requerido.

El capítulo se estructura de la forma que se presenta a continuación. Primero se describen algunos conceptos básicos de teoría de control y se presenta el problema de la reducción de modelos. En particular, se estudian los métodos basados en la SVD, que tienen como etapa más costosa la resolución de ecuaciones de Lyapunov. Luego, se describe la técnica para la resolución de una ecuación de Lyapunov mediante la función signo y se profundiza en diversas estrategias de aceleración y disminución del espacio de almacenamiento para dicha función. Posteriormente, se aborda la reducción de modelos en el caso de sistemas generalizados. Por último, se introducen los métodos de error relativo para este mismo problema, en particular, el método de truncamiento estocástico balanceado. Estos métodos implican la resolución de una ecuación de Riccati, para lo cual se estudian las aproximaciones basadas en la función signo y en el método de Newton.

2.1. Reducción de modelos

La formulación clásica de un sistema dinámico lineal (SDL) continuo e invariante en el tiempo, mediante el modelo de espacio de estados [185], queda definida por dos ecuaciones matriciales de la siguiente forma:

$$\begin{array}{rcll} \dot{x}(t) & = & Ax(t) + Bu(t), & t > 0, \quad x(0) = x_0, \\ y(t) & = & Cx(t) + Du(t), & t \geq 0, \end{array} \tag{2.1}$$

donde $x(t) \in \mathbb{R}^n$ denota el vector de variables de estado, $x(0) = x_0$ es el estado inicial del sistema, $u(t) \in \mathbb{R}^m$ es el vector de entradas o controles, $y(t) \in \mathbb{R}^p$ es el vector de salidas, $A \in \mathbb{R}^{n \times n}$ se conoce como la matriz de estados, $B \in \mathbb{R}^{n \times m}$ es la matriz de entradas, $C \in \mathbb{R}^{p \times n}$ es la matriz de salidas, y $D \in \mathbb{R}^{p \times m}$. Además, n es el orden (o la dimensión del espacio de estados) del sistema. La función de transferencia matricial asociada a (2.1), deducida de tomar su transformada de Laplace y asumiendo $x_0 = 0$, está definida como

$$G(s) = C(sI_n - A)^{-1}B + D, \tag{2.2}$$

donde I_n es la matriz identidad de orden n.

El modelo de representación en el espacio de estados ha permitido la aplicación de numerosos resultados provenientes del álgebra lineal numérica (ALN) al ámbito de la teoría de control [137, 185]. Así, los nuevos métodos desarrollados para este modelo permiten abordar, de manera numéricamente estable, problemas de grandes dimensiones que serían difícilmente tratables usando otro tipo de técnicas [150, 185].

Considerando el SDL de orden n en (2.1) y la matriz de función de transferencia asociada en (2.2), los métodos de reducción de modelos buscan obtener un segundo SDL,

$$\begin{array}{rcll} \dot{x}_r(t) & = & A_r x_r(t) + B_r u(t), & t > 0, \quad x_r(0) = \bar{x}_0, \\ y_r(t) & = & C_r x_r(t) + D_r u(t), & t \geq 0, \end{array} \tag{2.3}$$

de orden r mucho menor que n, y con función de transferencia:

$$G_r(s) = C_r(sI_r - A_r)^{-1}B_r + D_r, \tag{2.4}$$

que mantenga la información fundamental sobre la dinámica del sistema original; en particular, el objetivo es que las funciones $G_r(s)$ y $G(s)$ presenten un comportamiento similar (en otras palabras, que $\|y - y_r\|$ sea "pequeño"). Bajo estas premisas, el nuevo SDL en (2.3), de orden $r \ll n$, es una aproximación del SDL original en (2.1), lo que permite reemplazarlo en diferentes análisis o simulaciones posteriores.

Esta sustitución puede ser de vital importancia, entre otras, en las siguientes circunstancias:

- Cuando los recursos disponibles no permiten operar con sistemas de gran dimensión (por ejemplo, en dispositivos empotrados con una memoria limitada).
- Cuando la aplicación impone un tiempo de respuesta máximo que es imposible de alcanzar si se opera con sistemas de gran dimensión (por ejemplo, en entornos que operan en tiempo real).

Como se mencionó anteriormente, los métodos de reducción de modelos más populares se pueden agrupar en dos grandes familias: los métodos de aproximación basados en descomposiciones SVD [154, 184] y los métodos de aproximación por subespacios de Krylov [87, 127, 126, 132, 194].

Entre los métodos de reducción de modelos basados en aproximación por SVD se pueden encontrar dos clases, los métodos de error absoluto y los de error relativo. Entre los primeros destacan los algoritmos de truncamiento balanceado (BT, del inglés *balanced truncation*) [161, 197, 206, 215], los algoritmos de aproximación de perturbaciones singulares (*singular perturbation approximation*) [156] y los algoritmos de aproximación de la norma de Hankel (*Hankel-norm approximation*) [93]. Todos estos métodos presentan, como problema computacional principal, la resolución de un par de ecuaciones matriciales (lineales) de Lyapunov que tienen por operandos las matrices de estados, entradas y salidas del SDL. Entre los métodos de error relativo destaca el método de truncamiento estocástico balanceado (BST, del inglés *balanced stochastic truncation*) [76]. En estos métodos, el problema computacional clave es la solución de una ecuación de Lyapunov y una ecuación matricial (cuadrática) de Riccati.

Los métodos de aproximación por subespacios de Krylov básicamente requieren el cálculo de algún tipo de factorización de la matriz de controlabilidad (o alcanzabilidad) del SDL, definida por:

$$R_k(A,B) = \left[B,\ AB,\ (A)^2B,\ \ldots,(A)^{k-1}B\right].$$

Esta factorización se obtiene, habitualmente, mediante el algoritmo iterativo de Arnoldi o alguna de sus variantes [99, 196].

Este trabajo se centra en los métodos SVD y, más concretamente, en los métodos de error absoluto BT, ya que es una de las técnicas de reducción de modelos más popular en teoría de control. Si bien los métodos basados en la SVD en general presentan un mayor costo computacional que la aproximación por subespacios de Krylov, los primeros tienen como ventaja principal que garantizan la preservación de importantes propiedades del sistema original, como la estabilidad y la pasividad [24]. Además, este tipo de métodos proporciona una cota del error introducido por el modelo reducido, lo cual permite desarrollar métodos adaptativos que reducen la dimensión del sistema en función de un margen de error requerido. En forma complementaria, el trabajo abarca también una aproximación primaria a las técnicas de error relativo, estudiando el método BST.

2.1.1. Métodos basados en la SVD

La controlabilidad de un sistema es un concepto similar a encontrar el gramiano de controlabilidad. En particular, los métodos de reducción de modelos BT se basan en encontrar un sistema de coordenadas apropiado para el espacio de estados, en el cual los gramianos de controlabilidad y observabilidad del sistema, $W_c \in \mathbb{R}^{n\times n}$ y $W_o \in \mathbb{R}^{n\times n}$ respectivamente, son diagonales e iguales. Es más, el método BT explota el hecho de que los gramianos del SDL en (2.1) pueden ser obtenidos como la solución de las siguientes ecuaciones acopladas de Lyapunov:

$$AW_c + W_cA^T + BB^T = 0, \quad A^TW_o + W_oA + C^TC = 0. \tag{2.5}$$

Esta propiedad se puede aplicar siempre y cuando el SDL sea estable, es decir, si todos sus polos (valores propios de A) están en el semiplano complejo izquierdo (tienen parte real negativa). En otras palabras, la matriz A debe ser estable (o *Hurwitz*), es decir, el espectro de A, denotado por $\Lambda(A)$, debe satisfacer $\Lambda(A) \subset \mathbb{C}_-$. De la teoría de estabilidad de Lyapunov (consultar, por ejemplo, [146, Capítulo 13]) se deriva que, para una matriz A estable, las ecuaciones duales de Lyapunov en (2.5) tienen soluciones semidefinidas positivas y únicas W_c y W_o.

Además, se establece que si la matriz W_c es definida positiva, el sistema es controlable, y si la matriz W_o es definida positiva, el sistema es observable.

Los conceptos de controlabilidad y observabilidad son equivalentes a minimizar el sistema, por lo cual, para sistemas mínimos todos los valores propios del producto W_cW_o son números

reales estrictamente positivos. Las raíces cuadradas de estos valores propios, denotadas en orden decreciente como

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n > 0,$$

se conocen como valores singulares de Hankel[1] (HSVs, del inglés *Hankel singular values*) del SDL y son invariantes a transformaciones de coordenadas del espacio de estados.

Una vez modificado el sistema de coordenadas, transformando los gramianos en matrices diagonales con las entradas positivas y en orden decreciente, el modelo de orden reducido se obtiene truncando los estados correspondientes a los $n - r$ HSVs menores.

El método directo para hallar la solución de la ecuación de Lyapunov $AW_c + W_cA^T + BB^T = 0$ implica resolver un sistema lineal $K_cx = q$, donde $K_c = I_n \otimes A^T + A^T \otimes I_n \in \mathbb{R}^{n^2 \times n^2}$, $\otimes$ representa el producto de Kronecker, $q = \text{vec}(BB^T)$, $x = \text{vec}(W_c)$, y $\text{vec}(M)$ es el vector que se obtiene al disponer las columnas de la matriz M de forma consecutiva. De manera análoga, la solución de $A^TW_o + W_oA + C^TC = 0$ puede obtenerse del sistema lineal $K_oy = p$, con $K_o = I_n \otimes A + A \otimes I_n$, $p = \text{vec}(C^TC)$ e $y = \text{vec}(W_o)$. Esta estrategia es prohibitiva por su elevado costo, ya que la resolución de estos sistemas de ecuaciones lineales precisa $\mathcal{O}(n^6)$ operaciones aritméticas en coma flotante (flops) y almacenamiento para $\mathcal{O}(n^4)$ números. Por esta razón, se han desarrollado diversas alternativas para abordar el problema. A continuación se describen las técnicas computacionales más difundidas para la resolución de ecuaciones de Lyapunov.

El método de Bartels-Stewart [39] se basa en transformar la matriz coeficiente de la ecuación a una forma triangular superior por bloques (en concreto, la forma real de Schur) mediante transformaciones de semejanza. Luego es necesario ir resolviendo ecuaciones de Lyapunov con matrices de dimensión 1 ó 2 (dependiendo del tamaño de bloque); en el primer caso se resuelve directamente el sistema; en cambio, si las matrices son 2×2, se puede utilizar el producto de Kronecker para hallar la solución. Este proceso se repite para cada una de las columnas de la solución. Otro método utilizado para resolver ecuaciones matriciales lineales es el de Hessenberg-Schur [98]; sin embargo, en el caso de las ecuaciones de Lyapunov este método es equivalente al de Bartels-Stewart. Para la resolución de las ecuaciones de Lyapunov destaca también el método de Hammarling [120], una variante ingeniosa del método de Bartels-Stewart que obtiene de forma directa el factor de Cholesky de la solución. En estos métodos existe una última etapa en la que se recupera la solución de la ecuación original, antes de la reducción de la matriz coeficientes. Estos métodos tienen como característica su elevada exigencia computacional, $\mathcal{O}(n^3)$ flops, y un alto requerimiento de almacenamiento, $\mathcal{O}(n^2)$ números. Además, las operaciones involucradas en estos métodos (en particular, la reducción a la forma real de Schur) no son propicias para la aplicación de técnicas de computación de alto desempeño (HPC) y menos sobre arquitecturas masivamente paralelas.

Otra técnica muy extendida para la resolución de ecuaciones de Lyapunov con matriz de coeficientes estable es el método de la función signo [190]. El estudio de la tesis se centra en la optimización de este método para el cómputo de los factores de rango bajo de los gramianos. Si bien este método presenta un costo aritmético ($\mathcal{O}(n^3)$ flops) y espacial ($\mathcal{O}(n^2)$ números) similar al de los métodos comentados anteriormente, sus características permiten la aplicación de técnicas de programación paralela y de computación de altas prestaciones. Por lo tanto, esta alternativa es más apropiada para su implementación sobre las nuevas arquitecturas de hardware que disponen decenas, o incluso centenas, de unidades computacionales.

El problema de la reducción de modelos para SDL discretos puede formularse de forma análoga y la mayoría de los métodos discutidos en este trabajo son aplicables a este tipo de sistemas. Para una descripción detallada sobre técnicas para sistemas discretos, consultar el trabajo de Obinata y Anderson [177]. Asimismo existen técnicas específicas para afrontar los casos en que las matrices que

[1] Notar la similitud de la técnica con el cálculo de los valores singulares de una matriz.

definen el SDL son dispersas; estas técnicas obtienen el modelo reducido con un costo proporcional a la cantidad de elementos no nulos de la matriz A. Entre estos métodos destacan los basados en Smith [113] y ADI (del inglés *Alternating Direction Implicit*) [183, 220]. Este trabajo únicamente aborda la reducción de modelos de SDL continuos, con matriz de coeficiente densa.

2.1.2. Realizaciones balanceadas

Una realización de un SDL está definida por el conjunto de matrices (A, B, C, D), asociado a la Ec. (2.1). En general, un SDL tiene infinitas realizaciones y su función de transferencia es invariante ante transformaciones en el espacio de estados. Dada la transformación

$$\mathcal{T} : \begin{cases} x & \rightarrow & Tx, \\ (A, B, C, D) & \rightarrow & (TAT^{-1}, TB, CT^{-1}, D), \end{cases} \tag{2.6}$$

con un simple cálculo se puede demostrar que

$$(CT^{-1})(sI_n - TAT^{-1})^{-1}(TB) + D = C(sI_n - A)^{-1}B + D = G(s)$$

En base a lo anterior, se puede deducir que la representación asociada a un SDL no es única. Es más, cualquier adición en los estados que no modifique la relación entre las entradas y salidas del SDL (es decir, que para una entrada u, la misma salida y es alcanzada por ambos sistemas), conduce a una realización válida del mismo SDL. De esta forma, el orden de un sistema puede ser arbitrariamente extendido sin modificar el mapeo entrada-salida. Por otro lado, para cada sistema existe una cantidad mínima de estados, $\hat{n}$, necesarios para describir por completo el comportamiento de la relación entrada-salida establecida en el sistema. Este número se denomina *grado de McMillan* del sistema y la realización $(\hat{A}, \hat{B}, \hat{C}, \hat{D})$ de orden $\hat{n}$ es una *realización mínima* del sistema. Es importante notar que el grado de McMillan es único y la transformación del espacio de estados del sistema en (2.6) conduce a otra realización mínima del mismo.

Encontrar una realización mínima para un sistema dado, donde los estados redundantes (no mínimos) son eliminados del sistema, puede considerarse como un primer paso en la resolución del problema de reducción de modelos,

Aunque las realizaciones no son únicas, un SDL estable tiene un conjunto de estados invariantes con respecto a la transformación del espacio de estados que provee una buena vía para buscar el modelo de orden reducido. Además, como se comentó anteriormente, si la matriz A es estable, las ecuaciones de Lyapunov en (2.5) tienen sendas soluciones únicas y semidefinidas positivas, W_c y W_o. Dado que controlabilidad y observabilidad son equivalentes a minimizar el sistema, y que para un sistema mínimo todos los valores propios del producto W_cW_o son estrictamente positivos, se pueden calcular los HSVs del SDL que son invariantes en el sistema; esto se puede corroborar fácilmente. Considérese

$$(\hat{A}, \hat{B}, \hat{C}, D) = (TAT^{-1}, TB, CT^{-1}, D);$$

la realización transformada con la ecuación de controlablidad asociada

$$0 = \hat{A}\hat{W}_c + \hat{W}_c\hat{A}^T + \hat{B}\hat{B}^T = TAT^{-1}\hat{W}_c + \hat{W}_cT^{-T}A^TT^T + TBB^TT^T,$$

es equivalente a

$$0 = A(T^{-1}\hat{W}_cT^{-T}) + (T^{-1}\hat{W}_cT^{-T})A^T + BB^T.$$

En consecuencia, la unicidad de la solución de la ecuación de Lyapunov (ver, por ejemplo [146]) implica que $\hat{W}_c = TW_cT^T$, $\hat{W}_o = T^{-T}W_oT^{-1}$ y, por lo tanto,

$$\hat{W}_c\hat{W}_o = TW_cW_oT^{-1},$$

mostrando que $\Lambda\left(\hat{W}_c\hat{W}_o\right) = \Lambda\left(W_cW_o\right) = \{\sigma_1^2, \ldots, \sigma_n^2\}$. La extensión del espacio de estados mediante estados no minimales implica agregar HSVs de magnitud cero, mientras que los restantes HSVs distintos de cero se mantienen invariantes.

Una clase de realizaciones importante (que induce el nombre de la técnica) son las *realizaciones balanceadas*. Una realización (A, B, C, D) se denomina *balanceada* si y solo si

$$W_c = W_o = \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \end{bmatrix};$$

esto es, los gramianos de controlabilidad y observabilidad son diagonales e iguales entre sí, con los HSVs ordenados de forma decreciente como entradas de sus respectivas diagonales.

Para la realización mínima ($n = \hat{n}$) siempre existe una transformación balanceada en el espacio de estados de la forma (2.6), con una matriz no singular $T_b \in \mathbb{R}^{n\times n}$. Para un sistema no mínimo ($n > \hat{n}$), los gramianos pueden ser transformados en matrices diagonales, de forma que

$$\hat{W}_c\hat{W}_o = \operatorname{diag}(\sigma_1^2, \ldots, \sigma_{\hat{n}}^2, 0, \ldots, 0).$$

Usando una realización balanceada obtenida mediante la matriz de transformación T_b, los HSVs permiten una interpretación de la energía de los estados; ver por ejemplo el trabajo de Van Dooren [214] para un estudio detallado. Específicamente, la energía mínima necesaria para alcanzar el estado x^0 es

$$\operatorname{ínf}_{\substack{u\in\mathcal{L}_2(-\infty,0] \\ x(0)=x^0}} \int_{-\infty}^{0} u(t)^T u(t)\, dt = (x^0)^T W_c^{-1} x^0 = (\hat{x}^0)^T \hat{W}_c^{-1} \hat{x}^0 = \sum_{k=1}^{n} \frac{1}{\sigma_k} \hat{x}_k^2,$$

donde $\hat{x}^0 := \begin{bmatrix} \hat{x}_1 \\ \vdots \\ \hat{x}_n \end{bmatrix} = T_b x^0$, ya que los HSVs menores se corresponden con estados que son difíciles de alcanzar. La energía de salida resultante de un estado inicial x^0 y $u(t) \equiv 0$ para $t > 0$ viene dada por

$$\|y\|_2^2 = \int_0^{\infty} y(t)^T y(t)\, dt = x_0^T W_o x_0 = (\hat{x}^0)^T \hat{W}_o \hat{x}^0 = \sum_{k=1}^{n} \sigma_k \hat{x}_j^2;$$

aquí los HSVs mayores se corresponden con los estados que contienen la mayor parte de la energía del sistema. La transferencia de energía desde las entradas a las salidas puede ser computada mediante la siguiente expresión

$$E = \sup_{\substack{u\in\mathcal{L}_2(-\infty,0] \\ x(0)=x^0}} \frac{\|y\|_2^2}{\int_{-\infty}^{0} u(t)^T u(t)\, dt} = \frac{(x^0)^T W_o x^0}{(x^0)^T W_c^{-1} x^0} = \frac{(\bar{x}^0)^T W_c^{\frac{1}{2}} W_o W_c^{\frac{1}{2}} \bar{x}^0}{(\bar{x}^0)^T \bar{x}^0},$$

donde $\bar{x}^0 = W_c^{-\frac{1}{2}} x^0$. Entonces, los HSVs $\left(\Lambda\left(W_cW_o\right)\right)^{\frac{1}{2}} = \left(\Lambda\left(W_c^{\frac{1}{2}} W_o W_c^{\frac{1}{2}}\right)\right)^{\frac{1}{2}}$, siendo éste un buen estimador de cuán implicado está un estado en la transferencia de energía desde las entradas a las salidas.

En resumen, es posible obtener un modelo de orden reducido que aproxime al original mediante la eliminación de los estados menos controlables o menos observables, manteniendo los estados

que contienen la mayor parte de la energía, ya que éstos son los que están más implicados en la transferencia de energía desde las entradas a las salidas. Esto es lo que se consigue manteniendo los estados correspondientes a los HSVs de mayor valor, y es exactamente la idea que persigue al método de BT que se describe a continuación.

2.1.3. Truncamiento balanceado

El objetivo en el método de truncamiento balanceado es computar una realización balanceada

$$(TAT^{-1}, TB, CT^{-1}, D) = \left(\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}, \begin{bmatrix} C_1 & C_2 \end{bmatrix}, D \right), \tag{2.7}$$

donde $A_{11} \in \mathbb{R}^{r\times r}$, $B_1 \in \mathbb{R}^{r\times m}$, $C_1 \in \mathbb{R}^{p\times r}$, con $r \leq \hat{n}$, y entonces definir el modelo de orden reducido mediante la realización truncada

$$(\hat{A}, \hat{B}, \hat{C}, \hat{D}) = (A_{11}, B_1, C_1, D). \tag{2.8}$$

Esta técnica se trata en forma exhaustiva, entre otros, en los trabajos de Moore [161], y Mullins y Roberts [165].

Tomando los resultados de [93, 161, 206], se pueden resumir las siguientes propiedades de aplicar la técnica de truncamiento balanceado.

Dada (A, B, C, D), una realización de un SDL estable con función de transferencia $G(s)$, y dado un modelo $(\hat{A}, \hat{B}, \hat{C}, \hat{D})$ de orden reducido r con función de transferencia asociada $\hat{G}(s)$ que lo aproxima, computado como en (2.7)–(2.8):

a) El sistema de orden reducido $\hat{G}$ es balanceado, mínimo y estable, si sus gramianos son

$$W_o = W_c = \tilde{\Sigma} = \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_r \end{bmatrix}.$$

b) El error absoluto de la transferencia sobre el modelo reducido está acotado por

$$\|G - G_r\|_\infty \leq 2 \sum_{k=r+1}^{\hat{n}} \sigma_k. \tag{2.9}$$

c) Si $r = \hat{n}$, entonces (2.8) es una realización mínima de G y $G = G_r$.

Particularmente importante es la cota del error en (2.9), ya que permite una elección adaptativa de la dimensión del modelo de orden reducido (r) que satisfaga un umbral de tolerancia preestablecido.

Es posible demostrar que, para un sistema controlable y observable (mínimo), por ejemplo un sistema con gramianos no singulares, la matriz

$$T = \Sigma^{\frac{1}{2}} U^T R^{-T} \tag{2.10}$$

donde $W_c = R^T R$ y $R W_o R^T = U \Sigma^2 U^T$ (descomposición en valores singulares), provee una transformación del espacio de estados balanceada. Además, a partir de un sistema no mínimo (2.8) puede

computarse un SDL de orden reducido sin la necesidad de calcular la matriz T completamente. Dado $W_o = S^T S$ y $W_c = R^T R$, entonces

$$S^{-T}(W_c W_o)S^T = (SR^T)(SR^T)^T = (U\Sigma V^T)(V\Sigma U^T) = U\Sigma^2 U^T,$$

por lo cual U y Σ pueden ser computados mediante una SVD de la matriz SR^T,

$$SR^T = \begin{bmatrix} U_1 & U_2 \end{bmatrix} \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix}, \qquad \Sigma_1 = \mathrm{diag}(\sigma_1, \ldots, \sigma_r). \tag{2.11}$$

De este modo, los proyectores T_l y T_r pueden ser obtenidos según

$$T_l = \Sigma_1^{-1/2} V_1^T R, \quad T_r = S^T U_1 \Sigma_1^{-1/2}, \tag{2.12}$$

para finalmente obtener el nuevo modelo como

$$\hat{A} = T_l A T_r, \quad \hat{B} = T_l B, \quad \hat{C} = C T_r \quad y \quad \hat{D} = D. \tag{2.13}$$

Este método es equivalente a computar en primera instancia una realización mínima de (2.1), luego balancear el sistema (2.7) con T como se presenta en (2.10), y finalmente truncar la realización balanceada como en (2.8). En particular, las realizaciones obtenidas en (2.8) y (2.13) son la misma, puesto que T_l contiene las primeras r filas de T y T_r las primeras r columnas de T^{-1}; ambas partes de T son necesarias para computar A_{11}, B_1 y C_1 en (2.7). Cabe destacar que el producto $T_r T_l$ es un proyector en el subespacio r-dimensional del espacio de estados y el modelo de orden reducido obtenido mediante (2.13) puede ser visto como una proyección de la dinámica del sistema en el subespacio de estados.

El algoritmo descrito en las Ecs. (2.12) y (2.13) es comúnmente referido como el método SRBT (del inglés *square root balanced truncation*). En los textos de referencia [151, 206] y varios otros que tratan el truncamiento balanceado, se asume que las matrices S y R son los factores de Cholesky de los gramianos del sistema. En el artículo de Benner et al. [54] se muestra que los factores de Cholesky de rango bajo de los gramianos del sistema pueden ser igualmente utilizados. Esto permite implementaciones más eficientes del algoritmo de BT, especialmente cuando $\hat{n} \ll n$. Además, el rango numérico bajo de los gramianos implica un rápido decaimiento de los valores propios asociados y, por consiguiente, un rápido decaimiento de los HSVs. Esta propiedad se estudia con mayor profundidad más adelante en el capítulo.

Si el sistema original es pronunciadamente desbalanceado (en este caso, la matriz de transformación del espacio de estados T está mal condicionada), es interesante el algoritmo sin raíz cuadrada (BFSR, del inglés *balancing free square-root*) para BT propuesto por Varga [215], que permite, en presencia de errores de redondeo, obtener modelos de orden reducido de mejor precisión. Este método combina la implementación del método SR formulado en [151, 206] y la reducción de modelos sin balanceo (*balancing-free model reduction*) propuesta por Safanov y Chiang [197]. El algoritmo BFSR solamente difiere de la implementación de SR en el procedimiento de obtención de los proyectores T_l y T_r a partir de la factorización SVD (Ec. (2.11)) de SR^T, y en que en este método el modelo de orden reducido no es balanceado. La idea principal es que, para computar el modelo de orden reducido, es suficiente usar bases ortogonales para $\mathrm{Im}(T_l)$ e $\mathrm{Im}(T_r)$, donde $\mathrm{Im}(M)$ denota al conjunto imagen de la transformación M, que pueden obtenerse calculando las siguientes factorizaciones QR:

$$S^T U_1 = [P_1 \; P_2] \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix}, \qquad R^T V_1 = [Q_1 \; Q_2] \begin{bmatrix} \bar{R} \\ 0 \end{bmatrix}, \tag{2.14}$$

donde las columnas de P_1, $Q_1 \in \mathbb{R}^{n\times r}$ son ortonormales, mientras que $\hat{R}$, $\bar{R} \in \mathbb{R}^{r\times r}$ son matrices triangulares superiores. El sistema de orden reducido queda definido según (2.13) con

$$T_l = (Q_1^T P_1)^{-1} Q_1^T, \qquad T_r = P_1, \tag{2.15}$$

donde el factor $(Q_1^T P_1)^{-1}$ es necesario para preservar la condición de proyector de la matriz $T_r T_l$.

El error absoluto de la realización de orden r computada mediante la implementación BFSR de truncamiento balanceado satisface la misma cota superior del error en (2.9).

2.2. La función signo

En 1971, Roberts [190] propuso la función signo matricial y mostró como utilizarla para resolver las ecuaciones de Sylvester y Lyapunov. El método ha sido aplicado con éxito en posteriores trabajos [41, 75, 128].

Considérese la matriz $Z \in \mathbb{R}^{n\times n}$ sin valores propios en el eje imaginario (esto es, $\Lambda(Z) \cap \jmath\mathbb{R} = \emptyset$), donde su descomposición de Jordan [99] está definida por

$$Z = T^{-1} \begin{pmatrix} J_- & 0 \\ 0 & J_+ \end{pmatrix} T, \tag{2.16}$$

y los valores propios de $J_- \in \mathbb{R}^{j\times j}$ /$J_+ \in \mathbb{R}^{(n-j)\times(n-j)}$ tienen parte real negativa/positiva [99]. La función signo matricial de Z queda definida por

$$\mathrm{fsign}(Z) = T^{-1} \begin{pmatrix} -I_j & 0 \\ 0 & I_{n-j} \end{pmatrix} T. \tag{2.17}$$

Existen esquemas iterativos simples para computar la función signo. El más difundido es la iteración de Newton, definida de la siguiente forma:

$$Z_0 = Z, \tag{2.18}$$

$$Z_{k+1} = \frac{1}{2}(Z_k + Z_k^{-1}), \quad k = 0, 1, 2, \ldots. \tag{2.19}$$

Esta estrategia es particularmente atractiva por su simplicidad, eficiencia y desempeño computacional, así como por presentar una convergencia asintóticamente cuadrática [53, 66].

Si bien se pueden encontrar en la literatura diversos esquemas iterativos para computar la función signo con interesantes propiedades, como por ejemplo una acelerada convergencia y la posibilidad de emplear técnicas de programación paralela en su implementación (consultar el trabajo de Kenney y Laub [139] para un estudio cabal del tema), la iteración básica de Newton aparece como uno de los métodos más robustos y que ofrece mejores tiempos de cómputo, tanto en implementaciones secuenciales como paralelas. En particular, la iteración de Newton en (2.19) solo requiere calcular sumas e inversiones de matrices, siendo ambas operaciones ampliamente estudiadas por la comunidad científica que trabaja en computación de altas prestaciones, y contando con gran caudal de desarrollo.

Otra forma de considerar el cómputo basado en la función signo es como un *método de proyección espectral*, ya que

$$P_- = \frac{1}{2}(I_n - \mathrm{fsign}(Z)), \tag{2.20}$$

es un proyector espectral en el subespacio estable Z-invariante. Además, $P_+ = (I_n + \mathrm{fsign}(Z))/2$ es un proyector espectral en el subespacio Z-invariante correspondiente a los valores propios en el

semiplano complejo derecho (parte real positiva). Sin embargo, cabe resaltar que P_- y P_+ no son proyectores ortogonales sino proyectores oblicuos al subespacio Z-invariante complementario.

A continuación, se resumen algunas propiedades importantes de la función signo. Dada una matriz $Z \in \mathbb{R}^{n\times n}$, con $\Lambda(Z) \cap \jmath\mathbb{R} = \emptyset$, entonces:

a) $(\text{fsign}(Z))^2 = I_n$;

b) $\text{fsign}\left(T^{-1}ZT\right) = T^{-1}\text{fsign}(Z)\,T$ para todas las matrices $T \in \mathbb{R}^{n\times n}$ no singulares;

c) $\text{fsign}\left(Z^T\right) = \text{fsign}(Z)^T$;

d) si p_+ y p_- denotan el número de valores propios de Z con parte real positiva y negativa respectivamente, entonces:

$$p_+ = \frac{1}{2}(n + \text{tr}(\text{fsign}(Z))), \quad p_- = \frac{1}{2}(n - \text{tr}(\text{fsign}(Z))),$$

donde $\text{tr}(M)$ define la traza de una matriz M;

e) si Z es estable, entonces

$$\text{fsign}(Z) = -I_n, \quad \text{fsign}(-Z) = I_n.$$

La función signo es criticada por varias razones, siendo la más prominente la necesidad de computar explícitamente la matriz inversa en cada paso, lo que conlleva un alto costo computacional. Además, este método no está definido en contextos con valores propios puramente imaginarios y puede tener problemas de condicionamiento cuando las matrices poseen valores propios cercanos al eje imaginario. Sin embargo, esto se da únicamente cuando existen valores propios con parte imaginaria de magnitud menor a la raíz cuadrada de la precisión de la máquina, es decir, si se trabaja con números en doble precisión la magnitud relativa de los valores propios debería ser menor a 10^{-8}. Habitualmente, en aplicaciones de control como las consideradas en este trabajo, los polos se encuentran apartados del eje imaginario, lo que permite emplear el método de la función signo. Por otro lado, el método de la función signo generalmente está mejor condicionado que las opciones basadas en el complemento de Schur, dado que la primera separa los espacios estables de los anti-estables mientras que el método de Schur esencialmente implica separar n subespacios. Para un análisis exhaustivo del cómputo de subespacios invariantes basados en la función signo, consultar los trabajos [32, 67]. El condicionamiento de la forma de Schur y de la forma triangular a bloques (como la computada por la función signo) es discutido en el trabajo de Konstantinov et al. [142]. Es más, en las aplicaciones consideradas en el área de control, en general $\text{cond}(\text{fsign}(Z)) = 1$ ya que Z es estable o anti-estable, y por lo tanto, el cómputo de $\text{fsign}(Z)$ es un problema bien condicionado.

2.2.1. Aceleración de la convergencia de la función signo

Existen diversas alternativas discutidas en la literatura que permiten acelerar la convergencia de la iteración de Newton para el cálculo de la función signo. Una discusión profunda se puede encontrar tanto en [138] como en [31], donde se ofrece un relevamiento así como una comparativa de los distintos esquemas propuestos. Una estrategia extendida y efectiva es el uso de un factor de escalado, γ, que se aplica en cada iteración del método de la siguiente forma:

$$Z_0 = Z, \tag{2.21}$$

$$Z_{k+1} = \frac{1}{2\gamma_k}(Z_k + \gamma_k^2 Z_k^{-1}), \quad k = 0, 1, 2, \ldots, \tag{2.22}$$

La elección de los diferentes γ_k se discute brevemente a continuación.

***Escalado por determinante* [66]:** el objetivo es minimizar la media de la distancia geométrica de los valores propios de Z_k a 1. Para ello se utiliza el factor de escalado

$$\gamma_k = \det(Z_k)^{\frac{1}{n}}, \tag{2.23}$$

donde $\det(M)$ representa el determinante de la matriz M. Cabe notar que el determinante, $\det(Z_k)$, se puede obtener como un resultado intermedio de los cómputos necesarios para calcular la inversa Z_k^{-1}.

***Escalado por norma* [124]:** esta elección tiene ciertas propiedades de minimización en el contexto del cómputo de descomposiciones polares. Es también beneficiosa sobre los errores de redondeo, ya que iguala las normas de los dos sumandos en el cálculo de la norma finita $(\frac{1}{\gamma_k} Z_k) + (\gamma_k Z_k^{-1})$. En este caso, el escalado está definido por:

$$\gamma_k = \sqrt{\frac{\|Z_k\|_2}{\|Z_k^{-1}\|_2}}. \tag{2.24}$$

siendo $\|M\|_2$ la 2-norma (o espectral) matricial de la matriz M.

Escalado por norma aproximada: dado que la norma espectral es computacionalmente costosa de calcular, en algunos trabajos [124, 138] se sugiere utilizar cotas conocidas de la misma (ver Golub y Van Loan [99]) o aproximar dicha norma por otra más económica de calcular. Por ejemplo, puesto que $\|Z_k\|_2 \leq \sqrt{\|Z_k\|_1 \|Z_k\|_\infty}$, con $\|M\|_1$ y $\|M\|_\infty$ las normas matriciales 1 e infinito respectivamente, se puede utilizar la siguiente expresión:

$$\gamma_k = \sqrt{\frac{\|Z_k\|_1 \|Z_k\|_\infty}{\|Z_k^{-1}\|_1 \|Z_k^{-1}\|_\infty}}. \tag{2.25}$$

Experimentos numéricos y consideraciones analíticas parciales (ver [55]) sugieren que un escalado basado en la norma aproximada es habitualmente una buena alternativa. Además, el cómputo de la 1-norma o la norma infinito puede ser paralelizado fácilmente, por lo cual se presenta como una buena opción en matemática computacional. Sin embargo, la elección de la mejor opción de escalado está fuertemente condicionada por las características de cada problema y, en la práctica, ha de ser realizada en forma empírica.

2.3. Resolución de las ecuaciones de Sylvester y Lyapunov mediante la función signo

En esta sección, en primer lugar se estudia el uso de la función signo para la resolución de la ecuación de Sylvester y luego se extiende la técnica para la resolución de la ecuación de Lyapunov.

La ecuación de Sylvester se define como

$$AX + XB + W = 0, \tag{2.26}$$

donde $A \in \mathbb{R}^{n\times n}, B \in \mathbb{R}^{m\times m}, W, X \in \mathbb{R}^{n\times m}$ y $\Lambda(A) \cap \Lambda(-B) = \emptyset$. Esta propiedad implica que la Ec. (2.26) tiene una solución única [146]. El cálculo

$$\begin{bmatrix} I_n & 0 \\ X & I_m \end{bmatrix} \begin{bmatrix} A & 0 \\ W & -B \end{bmatrix} \begin{bmatrix} I_n & 0 \\ -X & I_m \end{bmatrix} = \begin{bmatrix} A & 0 \\ 0 & -B \end{bmatrix} \tag{2.27}$$

revela que las columnas de $\left[\begin{smallmatrix} I_n \\ -X \end{smallmatrix}\right]$ generan el subespacio invariante de $Z = \left[\begin{smallmatrix} A & 0 \\ W & -B \end{smallmatrix}\right]$ correspondiente a $\Lambda(A)$. Este subespacio, tras aplicar un cambio de base apropiado y el cálculo de la matriz X, puede ser computado mediante la utilización de un proyector espectral sobre su subespacio Z-invariante. La función signo es una herramienta apropiada para resolver este problema siempre y cuando las matrices A y B sean estables, puesto que P_-, definido en la Ec. (2.20), es el proyector espectral requerido. Un análisis de la iteración (2.19) aplicada a Z permite observar que no es necesario generar P_-; en este caso la solución puede ser obtenida directamente de la matriz $\operatorname{fsign}(Z)$. En particular, basándose en la Ec. (2.27) y las propiedades de la función signo resumidas anteriormente, se puede demostrar que:

$$\operatorname{fsign}(Z) = \operatorname{fsign}\left(\begin{bmatrix} A & 0 \\ W & -B \end{bmatrix}\right) = \begin{bmatrix} -I_n & 0 \\ 2X & I_m \end{bmatrix},$$

y por lo tanto la solución de la Ec. (2.26) está dada por el bloque inferior izquierdo del límite dividido entre 2. Es más, la estructura triangular a bloques de Z permite desacoplar la iteración (2.19) como se presenta en el Algoritmo `GECSNW`. Cuando este algoritmo converge, $X_* = \frac{1}{2}\lim_{k\to\infty} W_k$, siendo X_* la solución de (2.26).

Algoritmo `GECSNW`**:**
Iteración básica de Newton para la ecuación de Sylvester

$A_0 \leftarrow A$, $B_0 \leftarrow B$, $W_0 \leftarrow W$
`for` $k = 0, 1, 2, \ldots$ *until convergence*
$A_{k+1} \leftarrow \frac{1}{2\gamma_k}\left(A_k + \gamma_k^2 A_k^{-1}\right)$
$B_{k+1} \leftarrow \frac{1}{2\gamma_k}\left(B_k + \gamma_k^2 B_k^{-1}\right)$
$W_{k+1} \leftarrow \frac{1}{2\gamma_k}\left(W_k + \gamma_k^2 A_k^{-1} W_k B_k^{-1}\right)$

Para considerar la formulación estándar de la ecuación de Lyapunov se deben particularizar los resultados anteriores para la ecuación

$$AX + XA^T + Q \;=\; 0, \tag{2.28}$$

donde $A, Q, X \in \mathbb{R}^{n\times n}$, $Q = Q^T$ y $X = X^T$ es la incógnita. Asumiendo que todos los valores propios de A tienen parte real negativa, se puede aplicar el método de la iteración de Newton (2.18)–(2.19) a la matriz Z,

$$Z \;=\; \begin{bmatrix} A^T & 0 \\ Q & -A \end{bmatrix}, \tag{2.29}$$

de forma que,

$$Z_\infty \;=\; \lim_{k\to\infty} Z_k \;=\; \begin{bmatrix} -I & 0 \\ 2X & I \end{bmatrix}. \tag{2.30}$$

En la práctica, cada iteración con matrices Z_k de dimensión $2n \times 2n$ puede ser reemplazada con dos iteraciones con matrices de dimensión $n \times n$, según se muestra en el Algoritmo `GECLNW`.

Algoritmo `GECLNW`:
Iteración básica de Newton para la ecuación de Lyapunov
$A_0 \leftarrow A$, $Q_0 \leftarrow Q$
`for` $k = 0, 1, 2, \ldots$ *until convergence*
$A_{k+1} \leftarrow \frac{1}{2\gamma_k}\left(A_k + \gamma_k^2 A_k^{-1}\right)$
$Q_{k+1} \leftarrow \frac{1}{2\gamma_k}\left(Q_k + \gamma_k^2 A_k^{-1} Q_k A_k^{-T}\right)$

Una posible condición de parada para el Algoritmo `GECLNW` es:

$$\|A_k + I_n\|_F < \tau_{\text{iter}} \cdot \|A_k\|_F, \tag{2.31}$$

donde, dada la convergencia asintóticamente cuadrática de la iteración, comúnmente se especifica $\tau_{\text{iter}} = 10 \cdot n \cdot \sqrt{\varepsilon}$, siendo ε la precisión de la máquina. Ejecutando dos iteraciones más una vez que el criterio de condición de parada se satisface, en la práctica se asegura la obtención de una solución suficientemente precisa y se evitan problemas de convergencia lenta derivados de un mal condicionamiento del problema (nótese que esta misma discusión puede adaptarse fácilmente al Algoritmo `GECSNW`).

2.3.1. Disminución de requerimientos de memoria

En modelos de gran escala que surgen en problemas de teoría de control, como por ejemplo la reducción de modelos y el control óptimo, el número de estados en las ecuaciones en (2.1) es mucho mayor que el número de entradas ($n \gg m$) y salidas ($n \gg p$). Esto supone que las ecuaciones de Lyapunov frecuentemente exhiban una matriz de términos independientes Q en forma factorizada y que, además, la magnitud de los valores propios de la matriz solución (X) decaiga rápidamente, es decir, la matriz X presenta un rango numérico bajo (ver [26, 104, 184]). La Figura 2.1 (ejemplo extraído del trabajo de Benner et al. [51]) muestra el comportamiento antes descrito para el gramiano de controlabilidad de un SDL estable aleatorio con $n = 500$, $m = 10$, y *margen de estabilidad* (distancia mínima de $\Lambda(A)$ a $\jmath\mathbb{R}$) $\approx 0{,}055$. En este contexto, si n_ε=rank (X) es el rango numérico de X, existe una matriz $S_\varepsilon \in \mathbb{R}^{n\times n_\varepsilon}$ tal que $X \approx S_\varepsilon S_\varepsilon^T$ al nivel de la precisión de la máquina.

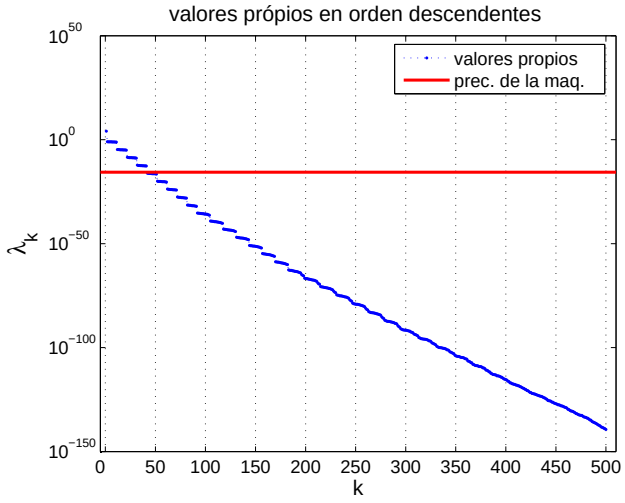

Figura 2.1: Ritmo de decaimiento de los valores propios para el gramiano de controlabiliad de un SDL aleatorio, con $n = 500$, $m = 10$, y margen de estabilidad $\approx 0{,}055$.

En la práctica, algunas aplicaciones no requieren explícitamente la solución de la ecuación, sino que es suficiente con conocer el factor de rango reducido S_ε. En estos casos el Algoritmo `GECLNC` para la solución de la ecuación $AX + XA^T + BB^T = 0$, presentado a continuación, es especialmente atractivo.

Algoritmo `GECLNC`**:**
Iteración factorizada de Newton para la ecuación de Lyapunov
$A_0 \leftarrow A$, $B_0 \leftarrow B$
`for` $k = 0, 1, 2, \ldots$ *until convergence*
$A_{k+1} \leftarrow \frac{1}{2\gamma_k}\left(A_k + \gamma_k^2 A_k^{-1}\right)$
$B_{k+1} \leftarrow \frac{1}{\sqrt{2\gamma_k}}[\, B_k,\ \gamma_k A_k^{-1} B_k \,]$

Esta observación sirve como idea básica en la mayoría de los algoritmos propuestos para resolver ecuaciones de Lyapunov de grandes dimensiones (ver [25, 184]), ya que almacenar S_ε (en el Algoritmo `GECLNC` se utiliza el espacio de memoria empleado por B_k para almacenarlo) es más económico que almacenar X; en concreto, en lugar de almacenar n^2 números sólo se almacenan $(n \times n_\varepsilon)$. En el ejemplo utilizado anteriormente para ilustrar el ritmo de decaimiento de los valores propios, se puede observar un ahorro de un 90 % (500 columnas frente a 50) para el almacenamiento del gramiano de controlabilidad, aunque es habitual encontrar situaciones en las que se reduce la necesidad de almacenamiento hasta en un 99 % [46].

Por otro lado, en el nuevo esquema iterativo el espacio requerido para almacenar B_k se dobla en cada paso. Existen diversas maneras para limitar dicho espacio. La primera técnica, propuesta en [148], trabaja con una matriz de $n \times n$, asignando en B_0 el factor de Cholesky BB^T, computando la factorización QR de $\begin{bmatrix} B_k \\ \gamma_k A_k^{-1} B_k \end{bmatrix}$ en cada iteración, y usando dicho factor R en la siguiente iteración. Una versión un poco más económica de esta técnica se presenta en el trabajo de Benner et al. [53], donde la factorización se aplica siempre y cuando $k \leq \log_2 \frac{n}{p}$, y sólo se computa una factorización

QR en los pasos posteriores. En ambos casos, se puede demostrar que B_k converge a un factor de Cholesky de la solución X de la Ec. (2.28). Sin embargo, esto puede ser mejorado computando la factorización QR con pivotamiento por columnas de B_{k+1}^T en cada iteración,

$$B_{k+1}^T \Pi_{k+1} = U_{k+1} \begin{bmatrix} R_{k+1} \\ 0 \end{bmatrix}, \tag{2.32}$$

donde Π_{k+1} es una matriz de permutación, U_{k+1} es ortogonal, y R_{k+1} es triangular superior. Dado que

$$B_{k+1}B_{k+1}^T = \Pi_{k+1}R_{k+1}^T U_{k+1}^T U_{k+1} R_{k+1}\Pi_{k+1}^T = (\Pi_{k+1}R_{k+1}^T)(R_{k+1}\Pi_{k+1}) = \tilde{R}_{k+1}^T \tilde{R}_{k+1}, \tag{2.33}$$

es posible reemplazar B_{k+1} por $\tilde{R}_{k+1}^T$. La reducción del costo de almacenamiento y cómputo de cada iteración utilizando esta factorización es fuertemente dependiente de las características del problema y, por lo tanto, es necesario evaluarla experimentalmente en cada caso concreto.

2.3.2. Ecuaciones de Lyapunov acopladas

Como se mencionó en el Apartado 2.1.1, el tratamiento del problema de reducción de modelos mediante el método BT implica la resolución de dos ecuaciones estándares de Lyapunov. Una primera aproximación a este problema es ejecutar dos instancias del Algoritmo `GECLNC`, una para la resolución de cada ecuación. Sin embargo, en el artículo de Lang y Lezius [147] se demuestra que no es necesario realizar dos iteraciones de la función signo para resolver las ecuaciones acopladas de Lyapunov. Dicha resolución puede ser unificada, debido a que esencialmente son la misma iteración para las matrices A_k, donde la única diferencia reside en que las matrices de las iteraciones son traspuestas entre ellas; por lo tanto solamente es necesario efectuar una iteración (o mejor dicho, una inversión por paso).

Esta idea fue generalizada y combinada con la iteración de rango bajo en los trabajos [48, 54]. El algoritmo de la función signo `CECLNW` es el resultante de aplicar las técnicas de optimización mencionadas hasta este momento, y obtiene los factores de rango bajo de los gramianos de controlabilidad y observabilidad del SDL.

Algoritmo `CECLNW`:
Iteración factorizada de Newton para las ecuaciones de Lyapunov acopladas

$A_0 \leftarrow A$, $\tilde{S}_0 \leftarrow B^T$, $\tilde{R}_0 \leftarrow C$
`for` $k = 0, 1, 2, \ldots$ *until convergence*
$A_{k+1} \leftarrow \frac{1}{2\gamma_k}\left(A_k + \gamma_k^2 A_k^{-1}\right)$

Computa la descomposición RRQR (*rank-revealing* QR)
$\frac{1}{\sqrt{2\gamma_k}}\left[\tilde{S}_k,\ \ \gamma_k \tilde{S}_k (A_k^{-1})^T\right] = Q_s \begin{bmatrix} U_s \\ 0 \end{bmatrix} \Pi_s$
$\tilde{S}_{k+1} \leftarrow U_s \Pi_s$

Computa la descomposición RRQR
$\frac{1}{\sqrt{2\gamma_k}}\left[\tilde{R}_k,\ \ \gamma_k (\tilde{R}_k A_k^{-1})\right] = Q_r \begin{bmatrix} U_r \\ 0 \end{bmatrix} \Pi_r$
$\tilde{R}_{k+1} \leftarrow U_r \Pi_r$

2.4. Reducción de modelos: caso generalizado

Consideremos ahora el SDL generalizado

$$\begin{array}{rcll} E\dot{x}(t) & = & Ax(t)+Bu(t), & t>0, \qquad x(0)=x_0, \\ y(t) & = & Cx(t)+Du(t), & t\geq 0, \end{array} \tag{2.34}$$

donde $E \in \mathbb{R}^{n\times n}$, $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, $C \in \mathbb{R}^{p\times n}$, $D \in \mathbb{R}^{p\times n}$, con función de transferencia asociada

$$G(s) = C(sE-A)^{-1}B + D. \tag{2.35}$$

El problema de reducción de modelos se puede plantear en este caso como la búsqueda de un segundo SDL,

$$\begin{array}{rcll} E_r\dot{x_r}(t) & = & A_r x_r(t)+B_r u(t), & t>0, \qquad x_r(0)=\hat{x}^0, \\ y_r(t) & = & C_r x_r(t)+D_r u(t), & t\geq 0, \end{array} \tag{2.36}$$

donde $E \in \mathbb{R}^{r\times r}$, $A \in \mathbb{R}^{r\times r}$, $B \in \mathbb{R}^{r\times m}$, $C \in \mathbb{R}^{p\times r}$, $D \in \mathbb{R}^{p\times r}$, $r \ll n$ e $\|y-y_r\|$ "pequeño".

En la resolución del problema de reducción de modelos generalizado son aplicables la mayor parte de los conceptos desarrollados previamente para el caso estándar ($E = I_n$), pero es necesario cuidar ciertos aspectos.

En este contexto, para encontrar los gramianos de controlabilidad y observabilidad, W_c y W_o, es necesario resolver dos ecuaciones de Lyapunov generalizadas de la forma

$$AW_cE^T + EW_cA^T + BB^T = 0, \tag{2.37}$$

$$A^T\tilde{W}_oE + E^T\tilde{W}_oA + C^TC = 0, \tag{2.38}$$

donde $W_o = E^T\tilde{W}_oE$.

Tras aplicar el método de la función signo (con las consiguiente modificaciones, expuestas en el siguiente apartado), se obtienen los factores S y R de rango bajo, donde $W_c = RR^T$ y $W_o = S^TS$. Considérese la descomposición en valores singulares del producto

$$SR^T = U\Sigma V^T = [\,U_1\ U_2\,]\begin{bmatrix}\Sigma_1 & \\ & \Sigma_2\end{bmatrix}[\,V_1\ V_2\,]^T, \tag{2.39}$$

donde U y V son matrices ortogonales, y $\Sigma = \mathrm{diag}\,(\sigma_1,\sigma_2,\ldots,\sigma_n)$ es una matriz diagonal que contiene los valores singulares de SR^T. Dada una partición de Σ en $\Sigma_1 \in \mathbb{R}^{r\times r}$ y $\Sigma_2 \in \mathbb{R}^{(n-r)\times(n-r)}$, y una partición conforme de U y V en la Ec. (2.39), el método SRBT para el caso generalizado determina un modelo (E_r,A_r,B_r,C_r,D_r) de orden r ($r \ll n$) al realizar las siguientes operaciones:

$$\begin{array}{rclrclrcl} E_r & = & T_lET_r, & A_r & = & T_lAT_r, & & & \\ B_r & = & T_lB, & C_r & = & CT_r, & D_r & = & D, \end{array} \tag{2.40}$$

donde

$$T_l = \Sigma_1^{-1/2}V_1^TRE^{-1} \quad \text{y} \quad T_r = S^TU_1\Sigma_1^{-1/2}. \tag{2.41}$$

2.4.1. Ecuaciones de Lyapunov generalizadas

Una primera aproximación para la resolución del problema de reducción de modelos asociado a un SDL generalizado es mediante la transformación que toma como entrada la tupla $(\tilde{E}, \tilde{A}, \tilde{B}, \tilde{C})$, donde el par $\tilde{E}, \tilde{A} \in \mathbb{R}^{n\times n}$ define el haz de matrices de estados, $\tilde{B} \in \mathbb{R}^{n\times m}$ es la matriz de entradas, y $\tilde{C} \in \mathbb{R}^{p\times n}$ es la matriz de salidas, y produce el SDL estándar $(I_n, \tilde{E}^{-1}\tilde{A}, \tilde{E}^{-1}\tilde{B}, \tilde{E}^{-1}\tilde{C})$.

Al aplicar estas operaciones se obtiene un SDL donde los gramianos se derivan de la resolución de las ecuaciones de Lyapunov estándar. Sin embargo, esta aproximación puede presentar problemas de estabilidad numérica [181] por lo cual, en general, no es una vía recomendable.

Otro enfoque, más estable desde el punto de vista numérico, es la resolución directamente de las ecuaciones generalizadas de Lyapunov (Ecs. (2.37)-(2.38)); esto puede ser abordado con el método de la función signo y la iteración de Newton, tal y como muestra el Algoritmo `CGCLNW`.

Algoritmo `CGCLNW`**:**
Iteración factorizada de Newton para las Ecs. de Lyapunov generalizadas acopladas

$A_0 \leftarrow A$, $\tilde{S}_0 \leftarrow B^T$, $\tilde{R}_0 \leftarrow C$
`for` $k = 0, 1, 2, \ldots$ *until convergence*
$A_{k+1} \leftarrow \frac{1}{2\gamma_k}\left(A_k + \gamma_k^2 (E A_k^{-1})E\right)$
Computa la descomposición RRQR
$\frac{1}{\sqrt{2\gamma_k}}\left[\tilde{S}_k,\ \gamma_k \tilde{S}_k (E A_k^{-1})^T\right] = Q_s \begin{bmatrix} U_s \\ 0 \end{bmatrix} \Pi_s$
$\tilde{S}_{k+1} \leftarrow U_s \Pi_s$
Computa la descomposición RRQR
$\frac{1}{\sqrt{2\gamma_k}}\left[\tilde{R}_k,\ \gamma_k (\tilde{R}_k A_k^{-1})E\right] = Q_r \begin{bmatrix} U_r \\ 0 \end{bmatrix} \Pi_r$
$\tilde{R}_{k+1} \leftarrow U_r \Pi_r$

Si el algoritmo converge tras $\tilde{k}_0$ iteraciones, entonces las aproximaciones de rango completo de S y R vienen dadas, respectivamente, por

$$\tilde{S} = \frac{1}{\sqrt{2}} \tilde{S}_{\tilde{k}_0} E^{-T} \quad \text{y} \quad \tilde{R} = \frac{1}{\sqrt{2}} \tilde{R}_{\tilde{k}_0} E^{-1}, \tag{2.42}$$

donde $\tilde{S} \in R^{\tilde{k}_o \times n}$, $\tilde{R} \in R^{\tilde{k}_c \times n}$ y, habitualmente, $\tilde{k}_o, \tilde{k}_c \ll n$, de forma que $W_c = S^T S \approx \tilde{S}^T \tilde{S}$ y $W_o = R^T R \approx \tilde{R}^T \tilde{R}$.

El núcleo del Algoritmo `CGCLNW`, desde el punto de vista del costo computacional, es la etapa que computa $\left(A_k + \gamma_k^2 (E A_k^{-1})E\right)$. A diferencia de lo que sucede en el método para la resolución de las ecuaciones estándar de Lyapunov, donde es necesario calcular la inversa de A_k, en este caso es más conveniente evitar el cálculo explícito de la inversa. En su lugar, resulta más eficiente calcular la factorización LU de la matriz A_k (asumiendo que esta matriz no presenta ninguna estructura particular), y resolver los dos sistemas triangulares correspondientes para encontrar la solución de $XLU = E$. De esta manera se realizan $\frac{2}{3}n^3$ flops para la factorización, n^3 flops para la resolución de cada uno de los dos sistemas triangulares y $2n^3$ flops más para el producto de $E A_k^{-1}$ con E. Frente a esto, la alternativa supone $2n^3$ flops para la inversión de la matriz A_k más $4n^3$ flops adicionales que implica las multiplicaciones de matrices $E A_k^{-1}$ y $(E A_k^{-1})E$.

2.5. Métodos de error relativo

Los métodos de reducción de modelos de error absoluto descritos en las secciones anteriores se basan en minimizar $\|\Delta_a\| = \|G - G_r\|$ para alguna norma. En particular, el método de realizaciones balanceadas proporciona una cota del error Δ_a según lo expuesto en la Ec. (2.9). Esta cota del error solo aporta información sobre la peor desviación del modelo de orden reducido con respecto al sistema original. Sin embargo, no se dispone de ninguna pauta de para qué entradas ocurre ese máximo de desviación. Por ejemplo, el método BT [161, 206] tiende a aproximar con mucha precisión las frecuencias altas ($\lim_{\omega\to\infty} \Delta_a(\jmath\omega) = 0$), mientras que el método de aproximación por perturbación singular [156] no tiene errores en los estados estacionarios (es decir, $G(0) = G_r(0)$) y presenta buenas propiedades de aproximación para las frecuencias bajas.

En diversas aplicaciones es necesario que el sistema de orden reducido provea una aproximación uniforme sobre todo el rango de frecuencias, $0 \le \omega \le \infty$, mientras que otras precisan únicamente de una buena aproximación para un rango de frecuencias preestablecidas. Este es el caso, por ejemplo, si el SDL describe el comportamiento de un controlador de alto orden que tiene que trabajar correctamente en un determinado rango de frecuencias. Este requerimiento puede ser satisfecho por los métodos de error relativo.

Los métodos de error relativo buscan minimizar el error relativo $\|\Delta_r\|_\infty$, definido implícitamente por $G - G_r = G\Delta_r$. Entre éstos, el método de truncamiento estocástico balanceado (BST) [76, 106, 217] y los métodos derivados de él son particularmente relevantes. Sin embargo, teniendo en cuenta su costo computacional y los cálculos involucrados, estos métodos pueden ser aplicados solo en sistemas de tamaño modesto, es decir, modelos donde n es del orden de algunos miles.

En cuanto a los fundamentos matemáticos, BST es un método de reducción de modelos basado en truncar una realización estocásticamente balanceada. Sean $\Phi(s) = G(s)G^T(-s)$ y W un factor espectral de fase mínima de Φ, es decir $\Phi(s) = W^T(-s)W(s)$, con D una matriz de rango completo, $\tilde{D} = DD^T$ definida positiva; entonces, una realización en el espacio de estados mínima (A_W, B_W, C_W, D_W) de W viene dada por [21, 22]:

$$A_W = A, \quad B_W = BD^T + W_c C^T, \quad C_W = \tilde{D}^{-\frac{1}{2}}(C - B_W^T X_W), \quad D_W = \tilde{D}^{\frac{1}{2}}.$$

Aquí, W_c es el gramiano de controlabilidad de $G(s)$, la solución de la ecuación de Lyapunov estándar,

$$AW_c + W_c A^T + BB^T \;=\; 0 \tag{2.43}$$

mientras X_W es el gramiano de observabilidad de $W(s)$, obtenido como la solución estabilizante de la ecuación algebraica de Riccati (EAR)

$$(A - B_W\tilde{D}^{-1}C)^T X + X(A - B_W\tilde{D}^{-1}C) + XB_W\tilde{D}^{-1}B_W^T X + C^T\tilde{D}^{-1}C = 0. \tag{2.44}$$

Es decir, para esta solución particular,

$$\hat{A} = A - B_W\tilde{D}^{-1}C + B_W\tilde{D}^{-1}B_W^T X_W \tag{2.45}$$

es estable. Además, los gramianos W_c y X_W son matrices simétricas definidas (semi-)positivas y admiten descomposiciones $W_c = S^T S$ y $X_W = R^T R$ (factorizaciones de Cholesky). Como en el cómputo de una realización balanceada, puede obtenerse una transformación en el espacio de estados T con los subespacios invariantes dominantes por izquierda y derecha de $W_c X_W$, o como con los subespacios singulares por izquierda y derecha de SR^T, tal que la transformación del sistema dada por $(\tilde{A}, \tilde{B}, \tilde{C}, \tilde{D}) = (T^{-1}AT, T^{-1}B, CT, D)$ tiene el gramiano de controlabiliad $\tilde{W}_c$ que satisface:

$$\tilde{W}_c = T^{-1}W_c T^{-T} = \operatorname{diag}(\sigma_1, \ldots, \sigma_n) = T^T X_W T = \tilde{X}_W, \tag{2.46}$$

donde $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$. Esto es, el gramiano de observabilidad $\tilde{X}_W$ de la realización transformada de factor espectral derecho de $\Phi(s)$ es igual al gramiano de controlablidad de la realización transformada de $G(s)$. La realización de $G(s)$ se conoce como realización estocástica balanceada (BSR, del inglés *balanced stochastic realization*). La reducción de modelos basada en BSR se puede obtener truncando la realización $(\tilde{A}, \tilde{B}, \tilde{C}, \tilde{D})$ a un orden r donde $\sigma_r \gg \sigma_{r+1}$. Las propiedades del método BST se resumen en el siguiente enunciado, que reúne resultados de diversos trabajos [76, 106, 107].

Considérese la función de transferencia matricial del SDL $G(s) = C(sI_n - A)^{-1}B + D$ con A estable y D no singular y suponiendo que

$$(T^{-1}AT, T^{-1}B, CT, D) = \left(\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}, \begin{bmatrix} C_1 & C_2 \end{bmatrix}, D \right)$$

es una BSR tal que satisface (2.46); entonces

$$(A_r, B_r, C_r, D_r) = (A_{11}, B_1, C_1, D)$$

es estable y mínima con las siguientes propiedades:

a) $G_r(s) = C_r(sI_r - A_r)^{-1}B_r + D_r$ satisface la cota de error relativo

$$\|\Delta_r\|_\infty = \|G^{-1}(G - G_r)\|_\infty \leq \prod_{j=r+1}^{n} \frac{1+\sigma_j}{1-\sigma_j} - 1. \tag{2.47}$$

b) $G(s)$ es de fase mínima, es decir, G no tiene ceros en $\mathbb{C}_+$. De igual modo, $G_r(s)$ es de fase mínima.

En este trabajo se aborda el método de la raíz cuadrada, especialmente adecuado cuando se desea trabajar con SR^T en lugar de $W_c X_w$; ver [54]. En el mismo trabajo se puede encontrar un profundo relevamiento de los distintos enfoques para abordar la resolución del método.

Como se mencionó anteriormente, en el método BST es necesario resolver primero la ecuación de Lyapunov (2.43), para lo cual se puede utilizar el método de la función signo, obteniendo la factorización de rango completo de $W_c = S^T S$, es decir, $S \in \mathbb{R}^{n_c \times n}$ donde $n_c = \text{rank}\,(S) = \text{rank}\,(W_c)$. Además, es necesario resolver una ecuación de Riccati, para la cual se puede emplear tanto el método de Newton [28, 122] como el método de la función signo [42, 190]. En los siguientes apartados se describen ambas técnicas y su aplicación en el método BST.

2.5.1. Método de Newton para la ecuación de Riccati

El método de Newton permite resolver una ecuación de Riccati de la forma

$$F^T X + XF - XGX + Q = 0, \tag{2.48}$$

donde $F, G \in \mathbb{R}^{n \times n}$ son matrices simétricas definidas positivas.

El método se basa en una iteración en la cual se corrige la solución obtenida en cada paso, obteniéndose la matriz de corrección mediante la resolución de una ecuación de Lyapunov sobre la matriz F y con término independiente el error de la solución obtenida en el paso previo.

Para acelerar la convergencia de la iteración se puede utilizar la técnica *exact line search* [45], que permite disminuir la cantidad de pasos de la iteración. Hay que destacar el elevado costo computacional que implica cada paso del método, pues es necesario resolver una ecuación de Lyapunov,

por lo que los beneficios que puedan derivarse de una aceleración de la convergencia son claramente relevantes. La técnica se basa en modificar el tamaño del paso, utilizando el óptimo, es decir aquel que minimiza la norma Frobenius del próximo residuo. Esto supone un costo muy inferior al de una sola iteración, según [44] entre un 5 – 10 %. El algoritmo resultante, `GECRNW`, se detalla a continuación. Para especificar el método de Newton estándar basta con fijar la variable $t_k = 1$ para todas las iteraciones k.

Algoritmo `GECRNW`**:**
Método de Newton con *exact line search* para la ecuación de Riccati
$F_0 \leftarrow F$, $X_0 \leftarrow X$
`for` $k = 0, 1, 2, \ldots$ *until convergence*
$F_k \leftarrow F + GX_k$
$R(X_k) \leftarrow F_k^T X_k + X_k F_k - X_k G X_k + Q$
Resolver $F_k^T N_k + N_k F_k + R(X_k) = 0$
Calcular el parámetro t_k
$X_{k+1} \leftarrow X_k + t_k N_k$

Se puede observar que la EAR que se resuelve en el método Newton, Algoritmo `GECRNW`, difiere de la que aparece en el método BST (en 2.44); sin embargo, la Ec. (2.44) se puede transformar sencillamente en la Ec. (2.48) mediante los siguientes cambios de variables:

$$F = A - B_W \tilde{D}^{-1} C, \quad G = B_W \tilde{D}^{-1} B_W^T, \quad Q = C^T \tilde{D}^{-1} C. \tag{2.49}$$

El núcleo de mayor peso en cuanto al costo computacional del método es la resolución de la ecuación de Lyapunov, para lo cual se pueden utilizar las técnicas desarrolladas en este mismo capítulo basadas en la función signo.

2.5.2. Función signo para la ecuación de Riccati

La solución de la ecuación de Riccati (2.48) es una matriz $X \in \mathbb{R}^{n \times n}$ que cumple que los valores propios de la matriz $F - GX$ tienen parte real negativa. La solución de la EAR puede definirse también en términos de los subespacios invariantes del haz $H - \lambda I_{2n}$, donde H es la matriz Hamiltoniana

$$H = \begin{bmatrix} F & G \\ -Q & -F^T \end{bmatrix}. \tag{2.50}$$

Además, se puede demostrar que la matriz formada por una base de los espacios invariantes de la matriz H es la solución de la EAR asociada [47]. Entonces, una forma de calcular esta solución es separando el espectro de H en dos subespacios complementarios (estable y anti-estable). Se puede emplear esta propiedad, combinándola con la función signo, para resolver la primera etapa del procedimiento requerido. A partir de la función signo de H,

$$\operatorname{fsign}(H) = Y = \begin{bmatrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{bmatrix}, \tag{2.51}$$

para encontrar la solución a la EAR únicamente es necesario resolver (por ejemplo mediante aproximaciones de mínimos cuadrados) el siguiente sistema,

$$\begin{bmatrix} Y_{11} \\ Y_{12} + I_n \end{bmatrix} X = \begin{bmatrix} I_n - Y_{10} \\ -Y_{00} \end{bmatrix}. \tag{2.52}$$

El procedimiento resultante de aplicar los pasos antes descritos se resume en el Algoritmo `GECRSG`.

Algoritmo `GECRSG`**:**
Método de la función signo para la ecuación de Riccati

$H_0 \leftarrow \begin{bmatrix} F & G \\ -Q & -F^T \end{bmatrix}$
`for` $k = 0, 1, 2, \ldots$ *until convergence*
$H_{k+1} \leftarrow \frac{1}{2\gamma_k}\left(H_k + \gamma_k^2 H_k^{-1}\right)$
Resolver $\begin{bmatrix} Y_{11} \\ Y_{12} + I_n \end{bmatrix} X = \begin{bmatrix} I_n - Y_{10} \\ -Y_{00} \end{bmatrix}$

2.6. Resumen

En el capítulo se presentan algunos conceptos básicos del problema de reducción de modelos, y en particular, el método BT. Este método precisa, como etapa fundamental desde el punto de vista computacional, la resolución de un par de ecuaciones acopladas de Lyapunov. También se extiende el estudio para abordar de forma primaria el método BST, donde la principal etapa de cómputo es la resolución de una ecuación algebraica de Riccati (EAR), que se puede tratar mediante un algoritmo iterativo que resuelve una ecuación de Lyapunov por paso, o bien mediante el método de la función signo.

La herramienta principal para la resolución de estas ecuaciones matriciales es la iteración de Newton para la función signo, que se basa en el cálculo de operaciones básicas de ALN (fundamentalmente inversión de matrices) de gran exigencia computacional. Por lo tanto, cuando las ecuaciones a resolver son de gran dimensión, la ejecución de estas operaciones precisa del uso de estrategias de computación de alto desempeño.

Capítulo 3

Inversión de matrices utilizando GPUs

En el Capítulo 2 se constató la importancia de la función signo para la resolución de una ecuación de Lyapunov estándar o generalizada (así como las ecuaciones de Riccati y Sylvester) y, en consecuencia, su relevancia para abordar la resolución del problema de reducción de modelos. Una de las estrategias más extendidas para abordar el cálculo de la función signo es la iteración de Newton que, desde el punto de vista computacional, es un método muy costoso, pues requiere calcular la inversión de una matriz en cada iteración.

El alto costo computacional que implica la iteración de Newton para el cálculo de la función signo motiva su tratamiento mediante estrategias de computación de alto desempeño (HPC). En este sentido, y como se mostró en el Capítulo 1, se han presentado diversos trabajos que introducen técnicas de HPC en la resolución del problema de reducción de modelos, generalmente mejorando el desempeño computacional o paralelizando operaciones básicas de álgebra lineal numérica (ALN). Por otro lado, del relevamiento realizado del uso de GPUs para la resolución de problemas de propósito general en el Capítulo 1, se puede concluir que dicha arquitectura de hardware es propicia para la resolución de operaciones de ALN, especialmente cuando se trabaja con matrices densas. Sin embargo, en el relevamiento realizado no se encontraron propuestas previas de uso de GPUs para acelerar la resolución del problema de reducción de modelos, ni trabajos que cubran la aceleración de la inversión de matrices en GPUs.

Por las razones expuestas anteriormente, este capítulo se centra en el estudio de algoritmos que hagan uso de plataformas de hardware que incluyan procesadores de propósito general multinúcleo y GPUs para acelerar el cómputo de la inversión de matrices generales y simétricas definidas positivas (SDP). Entre otras técnicas, se evalúan estrategias de relleno (*padding*), trabajo híbrido CPU-GPU, esquemas concurrentes de ambas arquitecturas, y el uso de múltiples GPUs de forma coordinada.

El capítulo se estructura de la siguiente forma. Primero se describen los métodos desarrollados para la inversión de matrices generales, en particular estudiando las estrategias tradicionales basadas en la factorización LU y en el método de Gauss-Jordan (GJE). Se presentan diversas variantes para cada una de estas dos clases de algoritmos y, posteriormente, se evalúan y validan de forma experimental sobre distintas plataformas de hardware. En segunda instancia se estudian los métodos para la inversión de matrices SDP, estudiando tanto algoritmos basados en la factorización de Cholesky como en extensiones del método de GJE para este tipo de matrices. Por último, se incluyen algunos trabajos introductorios sobre el uso de múltiples GPUs para acelerar el cómputo de la matriz inversa. En este caso, únicamente se estudiaron propuestas basadas en planificación de ejecución estática para matrices generales.

3.1. Inversión de matrices generales

En primera instancia, se aborda la inversión de matrices generales, o sea, aquellas que no poseen una estructura particular (en contraposición por ejemplo con las matrices simétricas, triangulares, banda, etc.).

Entre los métodos para la inversión de matrices generales se pueden especificar dos grandes familias: los métodos basados en la factorización LU y las técnicas basadas en el método de Gauss-Jordan [99]. Ambos métodos poseen el mismo costo computacional, pero se diferencian en el orden que realizan las operaciones y, en consecuencia, en el patrón de acceso a memoria. Para mayor simplicidad, los algoritmos presentados en este documento no incorporan el pivotamiento; no obstante, todos las implementaciones presentadas en esta sección efectúan pivotamiento parcial por filas [200]. En ocasiones se utiliza la notación de MatLab para representar vectores/matrices columna, es decir, se emplea la notación $[A_0;\ A_1;\ A_2]$ para respresentar la matriz $\begin{bmatrix} A_0 \\ A_1 \\ A_2 \end{bmatrix}$.

3.1.1. Métodos basados en la factorización LU

La estrategia tradicional para calcular la inversa de una matriz $A \in \mathbb{R}^{n \times n}$ no singular se basa en la eliminación Gaussiana, y consiste en los siguientes pasos [99]:

1. Computar la factorización LU de la matriz $PA = LU$, donde $P, L, U \in \mathbb{R}^{n \times n}$, P es una matriz de permutación, y L y U son, respectivamente, los factores triangulares inferior y superior.

2. Invertir el factor triangular $U \rightarrow U^{-1}$.

3. Resolver el sistema $XL = U^{-1}$ para X.

4. Deshacer la permutación $A^{-1} = XP$.

LAPACK provee la funcionalidad requerida para los pasos antes descritos. En particular, la rutina GETRF implementa la factorización LU (con pivotamiento parcial de filas) de una matriz no singular (paso 1), mientras que la rutina GETRI calcula la inversa de una matriz A usando la factorización LU obtenida mediante GETRF (pasos 2–4).

En la Figura 3.1 se describe la mecánica de cómputo de la rutina GETRF (para calcular la factorización LU de la matriz; por simplicidad se obvia el pivotamiento de filas) de LAPACK usando la notación FLAME [56, 116]. El método LU_{UNB} presente en el algoritmo de la figura es la factorización LU escalar, consultar [99]. En el algoritmo, $m(M)$ representa el número de filas de la matriz M; el resto de la notación es intuitiva.

El costo computacional del cálculo de la inversa mediante los cuatro pasos descritos anteriormente es de $2n^3$ flops (operaciones aritméticas en coma flotante). El algoritmo recorre la matriz cuatro veces (una por paso) y presenta problemas de mal balanceo de la carga (entre otros, la resolución de sistemas triangulares es un problema intrínsecamente mal balanceado).

A continuación se presentan las distintas versiones propuestas e implementadas para acelerar el cálculo de la matriz inversa basadas en la factorización LU, utilizando el poder de cómputo que ofrecen las GPUs para efectuar los cálculos. Previamente, se describe la implementación sobre CPU para realizar dicha operación, basadas en la biblioteca LAPACK, que oficia de versión de referencia.

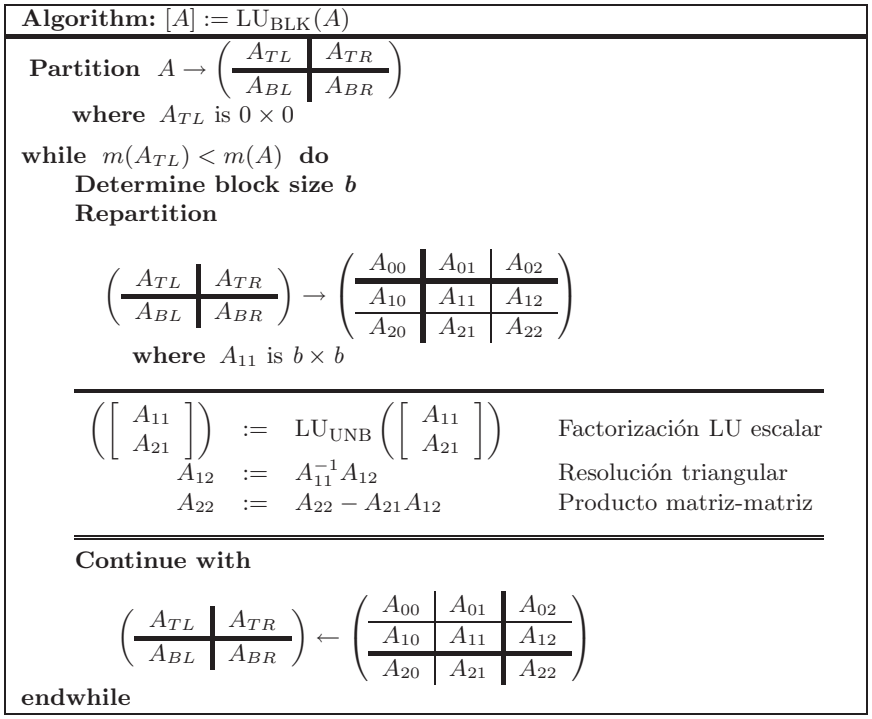

Figura 3.1: Algoritmo por bloques ($\mathrm{LU}_{\mathrm{BLK}}$) para la factorización LU de una matriz.

Implementación sobre multi-núcleo CPU: $\mathrm{LU}_{\mathrm{CPU}}$**.**

En esta variante, versión de referencia, se utilizan las rutinas de la especificación LAPACK, GETRF y GETRI. El paralelismo de esta versión se obtiene del uso de una implementación multi-hilo (*multi-thread*) de los núcleos básicos de BLAS invocados por ambas rutinas, básicamente la resolución de sistemas triangulares y el producto de matrices.

Implementación sobre GPU: $\mathrm{LU}_{\mathrm{GPU}}$**.**

En esta versión se implementaron en GPU las rutinas GETRF y GETRI, utilizando los núcleos computacionales de la biblioteca CUBLAS. También fue necesario implementar las funciones auxiliares GETF2 (para calcular la factorización LU de una matriz mediante operaciones BLAS-2, correspondiente a la factorización LU escalar) y TRTRI (para calcular la inversa de una matriz triangular, en el paso 2), ya que son invocadas por las rutinas GETRF y GETRI, respectivamente.

La metodología de ejecución del algoritmo es la siguiente:

1. Enviar la matriz del espacio de memoria del *host* al de la GPU.

2. Ejecutar las funciones GETRF y GETRI desarrolladas para GPU.

3. Transferir la matriz resultado del espacio de memoria de la GPU al del *host*.

Una primera mejora al algoritmo básico consiste en incluir estrategias de *padding*, es decir, trabajar con matrices cuyas dimensiones sean más propicias para el cómputo en GPU (generalmente múltiplos de 32 ó 64). Esto se puede realizar fácilmente añadiendo filas y columnas de elementos a la matriz original A de forma que se convierte la matriz

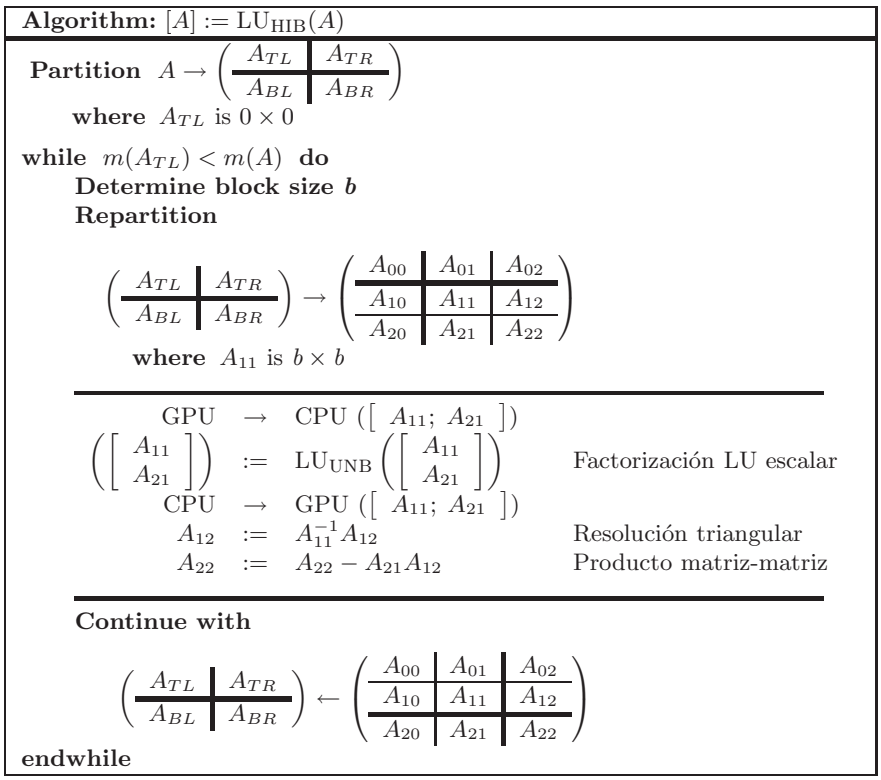

Figura 3.2: Algoritmo por bloques híbrido ($\mathrm{LU}_{\mathrm{HIB}}$) para la factorización LU de una matriz.

$$\tilde{A} = \begin{bmatrix} A & 0 \\ 0 & I_{p-n} \end{bmatrix}$$

de dimensión p, siendo éste el tamaño múltiplo deseado.

Como se presentó anteriormente en [35], esta técnica ha reportado resultados notables, mejorando sustancialmente el desempeño computacional de los métodos a los que se aplicó.

En resumen, esta variante se ejecuta completamente en GPU y utiliza estrategias de *padding*.

Implementación híbrida: $\mathrm{LU}_{\mathrm{HIB}}$.

En esta versión, siguiendo los resultados presentados en el trabajo de Barrachina et al. [37], se implementó una versión híbrida de la rutina GETRF, en la que la CPU y la GPU colaboran en el cómputo de la factorización. En la Figura 3.2 se describe la mecánica de cómputo de la rutina.

La metodología de trabajo es similar a la especificada en la variante $\mathrm{LU}_{\mathrm{GPU}}$, diferenciándose únicamente en la estrategia de cómputo de la factorización LU con pivotamiento. La mecánica que sigue el algoritmo es la siguiente:

1. Enviar la matriz del espacio de memoria del *host* al de la GPU.
2. Enviar el bloque actual de columnas [A_{11};A_{21}] a CPU y factorizarlo en el *host*.
3. Enviar el resultado a GPU.
4. Actualizar la matriz activa en GPU con el bloque actual factorizado (resolución de sistema triangular y multiplicación de matrices).

5. Se repiten los pasos 2 al 4 hasta factorizar por completo la matriz.

6. Ejecutar la función GETRI desarrollada para GPU.

7. Transferir la matriz resultado del espacio de memoria de la GPU al del *host*.

Se destaca de esta versión que la función GETRF utiliza la CPU para realizar las operaciones de grano fino, en particular, la factorización del bloque de columnas actual formado por $[A_{11};A_{21}]$, mientras que la GPU ejecuta las actualizaciones de la submatriz por procesar. La versión se completa con el uso de la rutina GETRI sobre GPU mencionada en el apartado anterior.

Implementación híbrida mejorada: $\mathrm{LU}_{\mathrm{HIB}+}$

Esta implementación se basa en el uso de núcleos de procesamiento sobre GPU de rutinas de BLAS provistos por la biblioteca MAGMA [19]. Las implementaciones en GPU de las rutinas GETRF, TRSM, y GEMM provistas por la biblioteca MAGMA son utilizadas para la obtención de la factorización LU con pivotamiento, la resolución de los sistemas triangulares y las multiplicaciones de matrices respectivamente. Dado que la rutina TRTRI no está disponible en MAGMA, se utiliza [81] la implementación propuesta por investigadores del laboratorio AICES-RWTHD [4]. Dicha rutina emplea estrategias híbridas de cómputo CPU-GPU y ha mostrado altas prestaciones.

Al igual que en la rutina anterior, esta variante requiere como primer paso la transferencia de la matriz a invertir del espacio de memoria de la CPU al de la GPU; ejecutar las operaciones de los cuatro pasos para invertir la matriz mediante la factorización LU utilizando las rutinas comentadas; y, por último, el envío de la matriz inversa al espacio de memoria de la CPU.

Es importante destacar que en esta versión tanto la factorización como la inversión triangular emplean estrategias híbridas de cómputo CPU-GPU, y, por lo tanto, ambas rutinas realizan diversas transferencias parciales durante su ejecución.

3.1.2. Métodos basados en la eliminación de Gauss-Jordan

El algoritmo de eliminación de Gauss-Jordan [92] para la inversión de matrices es esencialmente un reordenamiento de los cómputos realizados por el método de inversión basado en la eliminación Gaussiana, y como se mencionó anteriormente, posee el mismo costo computacional. La Figura 3.3 presenta la versión del algoritmo de GJE por bloques para la inversión de matrices. La descripción de una versión escalar, necesaria para actualizar el bloques de columnas actual ($[A_{01};\ A_{11};\ A_{21}]$), se puede consultar en [186].

Destaca del algoritmo que necesita barrer una única vez la matriz para calcular la inversa (a diferencia del método tradicional, que recorre la matriz al menos 4 veces). Además, la etapa de actualización se basa en operaciones de multiplicación de matrices (operaciones del nivel 3 de BLAS), un tipo de núcleo altamente optimizado y sobre el que las nuevas arquitecturas masivamente paralelas, como las GPUs, ha demostrado alcanzar prestaciones elevadas. Estas razones posicionan al método de GJE como un algoritmo especialmente propicio para su ejecución en GPUs.

A continuación se presentan seis implementaciones del método de GJE sobre las dos arquitecturas en estudio. Las variantes difieren en qué etapas de las rutinas son ejecutadas en CPU y cuáles en GPU, y en el orden (o planificación) en que se efectúan las mismas. Para las versiones que utilizan la CPU se intenta minimizar las comunicaciones entre la memoria de la CPU y la de la GPU, es decir, solamente se emplea la CPU cuando el costo de comunicación introducido es inferior al ahorro provocado por su uso.

Algorithm: $[A] := \mathrm{GJE}_{\mathrm{BLK}}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is 0×0 and A_{BR} is $n \times n$

while $m(A_{TL}) < m(A)$ **do**
Determine block size b
Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

$$\left[\begin{array}{c} A_{01} \\ A_{11} \\ A_{21} \end{array}\right] := \mathrm{GJE}_{\mathrm{UNB}}\left(\left[\begin{array}{c} A_{01} \\ A_{11} \\ A_{21} \end{array}\right]\right) \quad \text{Gauss-Jordan escalar}$$

$$\begin{array}{rcll} A_{00} & := & A_{00} + A_{01}A_{10} & \text{Producto matriz-matriz} \\ A_{20} & := & A_{20} + A_{21}A_{10} & \text{Producto matriz-matriz} \\ A_{10} & := & A_{11}A_{10} & \text{Producto matriz-matriz} \\ A_{02} & := & A_{02} + A_{01}A_{12} & \text{Producto matriz-matriz} \\ A_{22} & := & A_{22} + A_{21}A_{12} & \text{Producto matriz-matriz} \\ A_{12} & := & A_{11}A_{12} & \text{Producto matriz-matriz} \end{array}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

Figura 3.3: Algoritmo de GJE por bloques ($\mathrm{GJE}_{\mathrm{BLK}}$) para la inversión de una matriz.

Implementación sobre multi-núcleo CPU: $\mathrm{GJE}_{\mathrm{CPU}}$**.**

En esta primera variante, todas las operaciones se ejecutan en la CPU invocando rutinas de BLAS. El paralelismo en esta versión se obtiene al utilizar una implementación multi-hilo de dicha biblioteca.

Implementación sobre GPU: $\mathrm{GJE}_{\mathrm{GPU}}$**.**

Esta versión es análoga a $\mathrm{GJE}_{\mathrm{CPU}}$, pero utilizando la GPU para realizar los cálculos. La estructura de la rutina es la siguiente:

1. Inicialmente, la matriz a invertir se transfiere del espacio de memoria de la CPU (o *host*) al de la GPU.

2. Se realizan los cómputos en la GPU mediante la implementación de BLAS sobre esta arquitectura provista por NVIDIA, CUBLAS.

3. Finalmente, se transfiere la matriz inversa desde el espacio de memoria de la GPU al del *host*.

Como se mencionó anteriormente las estrategias de *padding* permiten mejorar el desempeño en GPU de las operaciones matriciales, por lo cual estas técnicas se incluyen en las versiones que utilizan la GPU para computar. Además, esta variante se ejecuta completamente en GPU.

Implementación híbrida: $\mathrm{GJE_{HIB}}$.

Si bien la mayoría de las operaciones presentes en el método de GJE se adaptan bien a la arquitectura de las GPUs, hay operaciones del método que no lo hacen. Éste es el caso de las operaciones de grano fino, que poseen un bajo costo computacional y una alta dependencia de datos, limitando su rendimiento sobre arquitecturas masivamente paralelas como las GPUs. Para resolver este problema, e inspirados en ideas similares a las de [37], se desarrolló una versión del algoritmo que emplea estrategias de cómputo híbridas CPU-GPU, en donde se busca que cada operación sea computada en el hardware que mejor se adapta a su resolución, es decir, sobre el que se obtiene un mejor desempeño. Además, simultáneamente se intenta que la cantidad de datos transferidos, y por lo tanto el tiempo de comunicación, sea mínima. El objetivo es explotar las capacidades de ambas arquitecturas manteniendo controlados los sobrecostos introducidos. En la Figura 3.4 se detalla el algoritmo implementado por esta versión, que se resume en los siguientes pasos:

1. Inicialmente se transfiere la matriz de la memoria del *host* a la memoria de la GPU.
2. Al comienzo de cada iteración del algoritmo las columnas del bloque actual, compuesto por $[A_{01};\ A_{11};\ A_{21}]$, se transfieren de la GPU al *host*.
3. Se procesa el bloque actual de columnas en la CPU.
4. Una vez procesado el bloque de columnas, el resultado es inmediatamente devuelto de la CPU a la GPU.
5. Se actualiza el resto de la matriz en GPU (mediante diversos productos de matrices).
6. Los pasos 2 al 5 se repiten hasta que se haya computado toda la matriz inversa.
7. Finalmente, la inversa es transferida de la GPU al *host*.

Destaca en esta versión que solamente la factorización del bloque actual de columnas es ejecutada en la CPU, ya que implica un número reducido de operaciones (esta cantidad depende del tamaño de bloque y habitualmente el tamaño de bloque óptimo es pequeño), para realizar el pivotamiento y operaciones del primer nivel de BLAS, que no se adaptan bien a la arquitectura de las GPUs. La actualización del resto de la matriz (multiplicaciones de matrices) y el pivotamiento fuera de las columnas del bloque actual se realiza en la GPU.

Implementación híbrida y concurrente: $\mathrm{GJE_{CON}}$.

Aunque con la versión anterior del método de GJE se logran mejoras en el desempeño computacional debido a que se ejecuta cada etapa en la arquitectura más conveniente, todos los cómputos se realizan de forma secuencial; es decir, en un momento dado está computando la CPU o la GPU, pero en ningún caso ambos dispositivos realizan cómputos en forma concurrente. Para abatir dicha restricción se diseñó una nueva versión del algoritmo de $\mathrm{GJE_{HIB}}$, basada en reordenar la planificación (*scheduling*) estática de las operaciones, permitiendo así solapar cálculos en ambas arquitecturas.

La especificación del método de GJE por bloques sugiere la utilización de un particionado 3×3; sin embargo, el reordenamiento propuesto en el algoritmo se puede ilustrar mejor mediante el uso de un particionado 3×4, como se muestra en la Figura 3.5, con cuatro paneles de columnas y tres de filas. La estructura de la rutina es la siguiente:

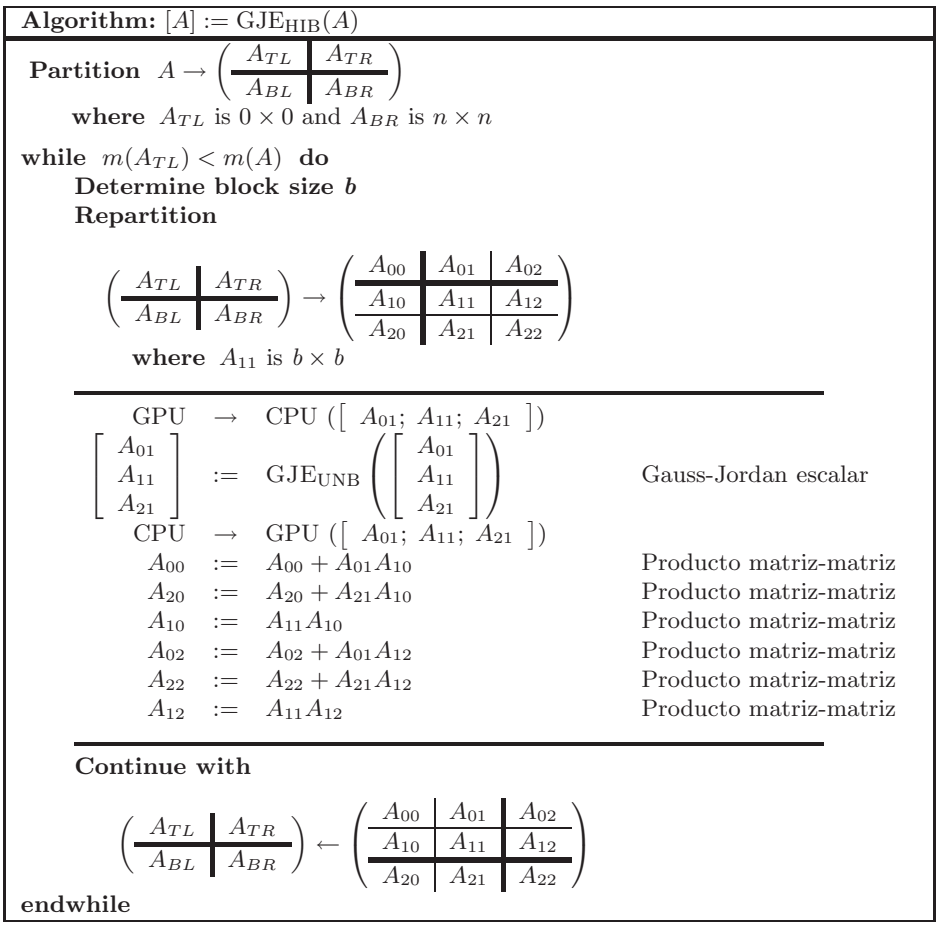

Figura 3.4: Algoritmo de GJE por bloques híbrido ($\mathrm{GJE}_{\mathrm{HIB}}$) para la inversión de una matriz.

1. Se transfiere la matriz del espacio de memoria del *host* al de la GPU.
2. Las columnas del bloque actual, compuesto por $[A_{01};\ A_{11};\ A_{21}]$, se transfieren de la GPU al *host*, donde se factorizan.
3. Una vez factorizado el bloque, el resultado es inmediatamente enviado a la GPU.
4. Se actualiza el panel de columnas $[A_{02}, A_{03};\ A_{12}, A_{13};\ A_{22}, A_{23}]$ y se envía a CPU el bloque $[A_{02};\ A_{12};\ A_{22}]$ (notar que en la siguiente iteración del algoritmo $[A_{02};\ A_{12};\ A_{22}]$ será el bloque actual de columnas).
5. Mientras la CPU factoriza el bloque $[A_{02};\ A_{12};\ A_{22}]$ la GPU actualiza $[A_{00};\ A_{10};\ A_{20}]$.
6. Los pasos 3 al 5 se repiten hasta invertir la matriz por completo.
7. La matriz inversa es finalmente transferida de la GPU al *host*.

En resumen, en esta versión, además de utilizar la plataforma de hardware que más se adapta para la resolución de cada etapa del método de GJE, se solapa la factorización del siguiente bloque de columnas $[A_{02};\ A_{12};\ A_{22}]$ en CPU, con la actualización del panel de columnas $[A_{00};\ A_{10};\ A_{20}]$ en GPU.

Algorithm: $[A] := \mathrm{GJE}_{\mathrm{CON}}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is 0×0 and A_{BR} is $n \times n$

while $m(A_{TL}) < m(A)$ **do**
Determine block size b
Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c} A_{00} & A_{01} & A_{02} & A_{03} \\ \hline A_{10} & A_{11} & A_{12} & A_{13} \\ \hline A_{20} & A_{21} & A_{22} & A_{23} \end{array}\right)$$

where A_{11} is $b \times b$, A_{12} is $b \times b$,

if $m(A_{TL}) == 0$			
GPU	$\rightarrow$	CPU ($[\ A_{01};\ A_{11};\ A_{21}\ ]$)	
endif			
$\left[\begin{array}{c} A_{01} \\ A_{11} \\ A_{21} \end{array}\right]$	:=	$\mathrm{GJE}_{\mathrm{UNB}}\left(\left[\begin{array}{c} A_{01} \\ A_{11} \\ A_{21} \end{array}\right]\right)$	Gauss-Jordan escalar
CPU	$\rightarrow$	GPU ($[\ A_{01};\ A_{11};\ A_{21}\ ]$)	
$[\ A_{02}, A_{03}\ ]$	:=	$[\ A_{02}, A_{03}\ ] + A_{01}\ [\ A_{12}, A_{13}\ ]$	Producto matriz-matriz
$[\ A_{22}, A_{23}\ ]$	:=	$[\ A_{22}, A_{23}\ ] + A_{01}\ [\ A_{12}, A_{13}\ ]$	Producto matriz-matriz
$[\ A_{12}, A_{13}\ ]$	:=	$A_{11}\ [\ A_{12}, A_{13}\ ]$	Producto matriz-matriz
GPU	$\rightarrow$	CPU ($[\ A_{01};\ A_{11};\ A_{21}\ ]$)	
A_{00}	:=	$A_{00} + A_{01}A_{10}$	Producto matriz-matriz
A_{20}	:=	$A_{20} + A_{21}A_{10}$	Producto matriz-matriz
A_{10}	:=	$A_{11}A_{10}$	Producto matriz-matriz

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c|c} A_{00} & A_{01} & A_{02} & A_{03} \\ \hline A_{10} & A_{11} & A_{12} & A_{13} \\ \hline A_{20} & A_{21} & A_{22} & A_{23} \end{array}\right)$$

endwhile

Figura 3.5: Algoritmo de GJE por bloques concurrente ($\mathrm{GJE}_{\mathrm{CON}}$) para la inversión de una matriz.

Implementación concurrente con inclusión de estrategias de look-ahead: $\mathrm{GJE}_{\mathrm{LA}}$**.**

Con el solapamiento de cálculos en ambas plataformas se logran ciertas mejoras en los tiempos de cómputo del algoritmo. Sin embargo, el costo computacional de las etapas que se solapan varía durante la ejecución del algoritmo, no permitiendo alcanzar un nivel de concurrencia constante. En concreto, durante las primeras iteraciones del algoritmo, cuando se procesan los primeros bloques de columnas, la GPU solapa muy poco trabajo, ya que la actualización de $[A_{00};\ A_{10};\ A_{20}]$ requiere pocos cómputos, mientras que en los últimos pasos ocurre lo contrario, ya que la factorización del bloque (ejecutada por la CPU) tiene un costo computacional muy inferior al requerido por la actualización del panel $[A_{00};\ A_{10};\ A_{20}]$ (realizada en la GPU). Además, la sincronización entre las unidades de cómputo depende del tiempo de ejecución de la factorización del bloque actual en CPU y la actualización del panel de columnas $[A_{02};\ A_{12};\ A_{22}]$ en GPU, siendo ambas etapas también de tiempo de cómputo desbalanceado durante la ejecución.

Buscando mejorar la situación descrita en el párrafo anterior, se desarrolló una nueva versión del algoritmo, basada en la versión $\mathrm{GJE}_{\mathrm{CON}}$, pero incluyendo técnicas de tipo *look-ahead* [202].

El objetivo perseguido es priorizar la ejecución de las operaciones que se encuentran en el camino crítico. En la Figura 3.6 se ilustra el algoritmo, que opera como se describe a continuación.

La estrategia del nuevo algoritmo requiere del envío de la matriz desde el espacio de memoria del *host* a la GPU al comienzo de la rutina y la transferencia de GPU a CPU del resultado al finalizar el cálculo de la inversa. La mecánica de los pasos intermedios del algoritmo se refleja en las etapas 2–8 siguientes:

1. Se transfiere la matriz de la memoria del *host* a la memoria de la GPU.
2. El primer bloque de columnas es transferido de la GPU a la CPU.
3. El primer bloque de columnas se factoriza en CPU.
4. El bloque actual $[A_{01};\ A_{11};\ A_{21}]$ es transferido de la CPU a la GPU.
5. El bloque $[A_{02};\ A_{12};\ A_{22}]$ es actualizado en la GPU.
6. Se envía a CPU el bloque $[A_{02};\ A_{12};\ A_{22}]$.
7. Mientras la GPU actualiza los paneles $[A_{00};\ A_{10};\ A_{20}]$ y $[A_{03};\ A_{13};\ A_{23}]$, la CPU procesa el bloque $[A_{02};\ A_{12};\ A_{22}]$.
8. Los pasos 4–7 se repiten hasta completar la inversión de la matriz.
9. Finalmente, la inversa es transferida de la GPU al *host*.

Con la inclusión de la técnica de *look-ahead* se logra mejorar el grado de solapamiento de cómputos entre la CPU y la GPU, y además el camino crítico del algoritmo se ve disminuido al poder comenzar con la sección concurrente tan pronto como se actualiza el bloque de columnas $[A_{02};\ A_{12};\ A_{22}]$ que posee un número fijo de columnas (el tamaño de bloque) durante todo el algoritmo. Esta situación favorece el solapamiento, mejorando la utilización de los recursos y reduciendo el tiempo de ejecución.

Implementación multi-nivel: $\mathrm{GJE}_{\mathrm{MN}}$.

Todos los algoritmos desarrollados se basan en estrategias de división de problemas en bloques. En arquitecturas tradicionales estas estrategias permiten explotar eficientemente los niveles de memoria (y en especial la memoria caché), y en GPU realizar operaciones con estructura uniforme. Esto implica que la elección del tamaño de bloque condiciona fuertemente el desempeño computacional de los algoritmos. En particular, el tamaño de bloque óptimo varía según la operación a ejecutar y la arquitectura sobre la que se ejecuta. En este sentido, las versiones híbridas desarrolladas poseen un problema: al emplear el mismo tamaño de bloque en ambas arquitecturas, este tamaño de bloque es un compromiso entre el óptimo para las etapas computadas en CPU y las computadas en GPU. En consecuencia, el desempeño global de las rutinas se deteriora. En general, el cómputo en CPU en este algoritmo alcanza las mejores prestaciones cuando se utiliza un tamaño de bloque "pequeño", en el entorno de 16-32, mientras que las etapas en GPU alcanzan el mejor desempeño cuando se utilizan tamaños de bloque "grandes", por ejemplo 256 y 512 ofrecen habitualmente buenas opciones. La Tabla 3.1 presenta un ejemplo de los tiempos de ejecución de las diferentes etapas del Algoritmo $\mathrm{GJE}_{\mathrm{LA}}$ para invertir una matriz de dimensión 8.000 en la plataforma PECO (CPU Intel Xeon QuadCore y GPU NVIDIA C1060) usando diferentes tamaños de bloques. Como muestra la tabla, el mejor tiempo global se obtiene con $b = 128$, a pesar de que

Algorithm: $[A] := \mathrm{GJE_{LA}}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is 0×0 and A_{BR} is $n \times n$

while $m(A_{TL}) < m(A)$ **do**
Determine block size b
Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c} A_{00} & A_{01} & A_{02} & A_{03} \\ \hline A_{10} & A_{11} & A_{12} & A_{13} \\ \hline A_{20} & A_{21} & A_{22} & A_{23} \end{array}\right)$$

where A_{11} is $b \times b$, A_{12} is $b \times b$

if $m(A_{TL})$	==	0	
GPU	→	CPU ($[\ A_{01};\ A_{11};\ A_{21}\ ]$)	
endif			
$\begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix}$	:=	$\mathrm{GJE_{UNB}}\left(\begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix}\right)$	Gauss-Jordan escalar
CPU	→	GPU ($[\ A_{01};\ A_{11};\ A_{21}\ ]$)	
A_{02}	:=	$A_{02} + A_{01}A_{12}$	Producto matriz-matriz
A_{22}	:=	$A_{22} + A_{21}A_{12}$	Producto matriz-matriz
A_{12}	:=	$A_{11}A_{12}$	Producto matriz-matriz
GPU	→	CPU ($[\ A_{02};\ A_{12};\ A_{22}\ ]$)	
A_{03}	:=	$A_{03} + A_{01}A_{13}$	Producto matriz-matriz
A_{23}	:=	$A_{23} + A_{21}A_{13}$	Producto matriz-matriz
A_{13}	:=	$A_{11}A_{13}$	Producto matriz-matriz
A_{00}	:=	$A_{00} + A_{01}A_{10}$	Producto matriz-matriz
A_{20}	:=	$A_{20} + A_{21}A_{10}$	Producto matriz-matriz
A_{10}	:=	$A_{11}A_{10}$	Producto matriz-matriz

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c|c} A_{00} & A_{01} & A_{02} & A_{03} \\ \hline A_{10} & A_{11} & A_{12} & A_{13} \\ \hline A_{20} & A_{21} & A_{22} & A_{23} \end{array}\right)$$

endwhile

Figura 3.6: Algoritmo de GJE por bloques con estrategia de *look-ahead* ($\mathrm{GJE_{LA}}$) para la inversión de una matriz.

dicho tamaño de bloque no es el óptimo para ninguna de las dos arquitecturas, siendo 16 y 512 los tamaños de bloque óptimos para la CPU y la GPU respectivamente.

Teniendo en cuenta esto, la variante $\mathrm{GJE_{MN}}$ introduce una modificación en el algoritmo $\mathrm{GJE_{LA}}$, de forma que la nueva versión del método de GJE está basada en el Algoritmo $\mathrm{GJE_{LA}}$, pero utiliza un algoritmo por bloques para factorizar el bloque actual de columnas ($[A_{01};\ A_{11};\ A_{21}]$) en CPU. Esta modificación permite que las rutinas ejecutadas en la CPU puedan operar con un tamaño de bloque propio, b_c, distinto de b. El objetivo es fijar b en un valor que maximice las prestaciones de la GPU y, al mismo tiempo, establecer b_c al valor óptimo para la CPU. La única restricción en los valores de b y b_c es que $b \geq b_c$, pero como se comentó anteriormente, el tamaño de bloque óptimo para la GPU es habitualmente mayor que el de la CPU, por lo que esta limitación no afecta negativamente a las prestaciones.

Tamaño de bloque (b)	Tiempo de CPU (seg.)	Tiempo de GPU + comunicaciones (seg.)	Tiempo total (seg.)
16	0,44	11,50	12,24
32	0,82	5,27	5,95
64	1,61	3,50	4,14
96	2,39	4,20	4,87
128	3,17	3,34	4,06
256	9,33	3,27	10,19
512	26,29	3,23	28,34

Tabla 3.1: Tiempo de cómputo de las distintas etapas del Algoritmo GJE_{LA} para invertir una matriz de dimensión 8.000 usando diferentes tamaños de bloque en la plataforma PECO.

En resumen, esta versión utiliza un tamaño de bloque general para el método de GJE (b), y a su vez emplea otro tamaño de bloque para factorizar el bloque actual de columnas en CPU (b_c). Esta estrategia permite, en primer lugar, aumentar el desempeño alcanzado por ambas arquitecturas, y en segundo lugar, alcanzar un mejor balance entre el tiempo de ejecución de la CPU y de la GPU, resultando en una reducción del tiempo total de ejecución.

3.1.3. Análisis experimental

En esta sección se presenta la evaluación de las diferentes versiones desarrolladas para el cálculo de la inversa de una matriz general. Las implementaciones evaluadas fueron anteriormente descritas en los apartados 3.1.1 y 3.1.2, y se enumeran a continuación:

- Variantes basadas en la factorización LU: LU_{CPU}, LU_{GPU}, LU_{HIB} y $\text{LU}_{\text{HIB}+}$.
- Variantes basadas en el método de GJE: GJE_{CPU}, GJE_{GPU}, GJE_{HIB}, GJE_{CON}, GJE_{LA} y GJE_{MN}.

Debido a que los algoritmos utilizados emplean estrategias de bloques, la primera etapa de cada uno de los experimentos se centró en la búsqueda de los tamaños de bloque óptimos para cada algoritmo y para cada tamaño de matriz utilizado (excepto la implementación LU_{CPU}, puesto que la implementación de LAPACK se asume optimizada). En todos los casos se evaluaron las versiones con los siguientes tamaños de bloque (b): 16, 32, 64, 128, 160, 192, 256, 288 y 320. Para la implementación que incluye multi-nivel, se evaluaron los valores 2, 4, 8, 16 y 32 como tamaño del sub-bloque (b_c). Cuando se utiliza 16 como tamaño de bloque, se evalúa b_c hasta 8, mientras que para el caso $b = 32$ los b_c estudiados incluyen el 16. Los GFLOPS (correspondientes a los tiempos de ejecución de los algoritmos) presentados en el resto de la sección se refieren a los obtenidos utilizando en cada caso el mejor tamaño de bloque.

Versiones básicas sobre HARRISON

El primer conjunto de experimentos se desarrolló sobre la plataforma HARRISON (ver Apartado 1.3.1), un PC de escritorio que no ofrece gran poder de cómputo, pero permite validar las hipótesis formuladas durante el desarrollo de las variantes y evaluar el alcance de las mejoras introducidas en hardware de bajo costo. Por esta razón, en esta plataforma solamente se evaluaron las primeras versiones de los algoritmos.

La Figura 3.7 muestra los GFLOPS alcanzados por los algoritmos de inversión para los distintos tamaños de matrices evaluados utilizando las rutinas basadas en la factorización LU. En particular, se presentan experimentos para matrices con dimensiones entre 1.000 y 7.000 (el máximo soportado por la memoria de la tarjeta utilizada). Observando los resultados se pueden constatar los beneficios obtenidos al utilizar la GPU para computar las operaciones de ALN, ya que las versiones que la emplean logran disminuciones significativas en los tiempos de ejecución, alcanzando valores de aceleración de hasta 4×. En cuanto a la comparación de las estrategias que utilizan la GPU, la estrategia híbrida presenta ventajas poco significativas con respecto a la que utiliza únicamente la GPU.

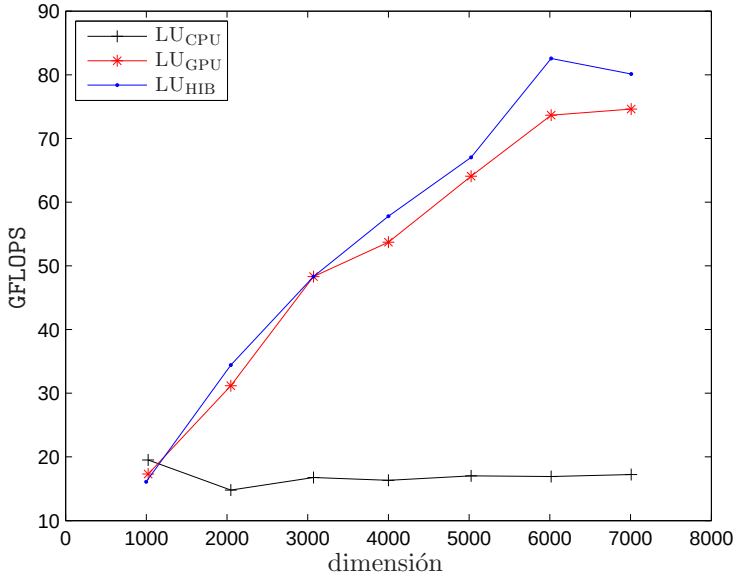

Figura 3.7: Desempeño (expresado en GFLOPS) alcanzado por las variantes de inversión de matrices basadas en la factorización LU sobre HARRISON.

La Figura 3.8 presenta los GFLOPS alcanzados por las versiones basadas en el método de la eliminación de Gauss-Jordan. Analizando los resultados se puede observar una situación similar a la que presentaban los algoritmos basados en la factorización LU, siendo notablemente más rápidas las versiones que utilizan la GPU.

La Figura 3.9 presenta la comparación de las versiones más eficientes de ambas estrategias de inversión. Se puede observar que, para todos los tamaños de matrices, las variantes basadas en el método de GJE superan a las basadas en la factorización LU, siendo la versión GJE_{CON} la que obtiene las mejores prestaciones.

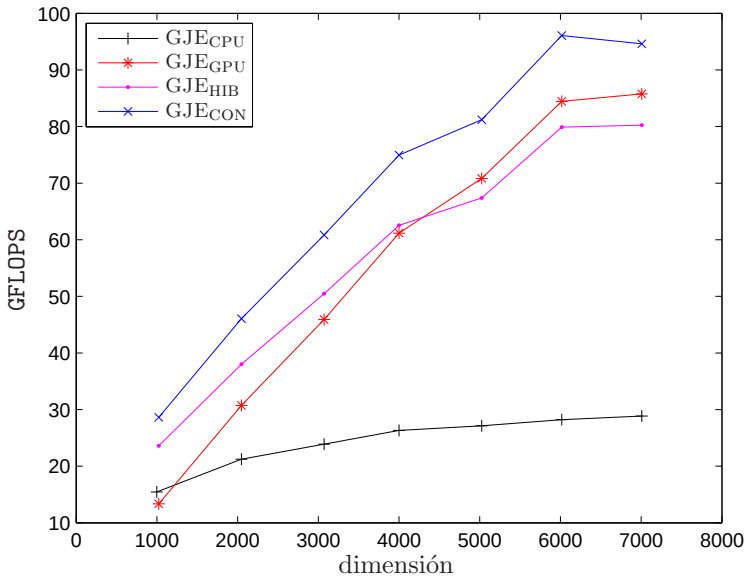

Figura 3.8: Desempeño (expresado en GFLOPS) alcanzado por las variantes de inversión de matrices basadas en el método de GJE sobre HARRISON.

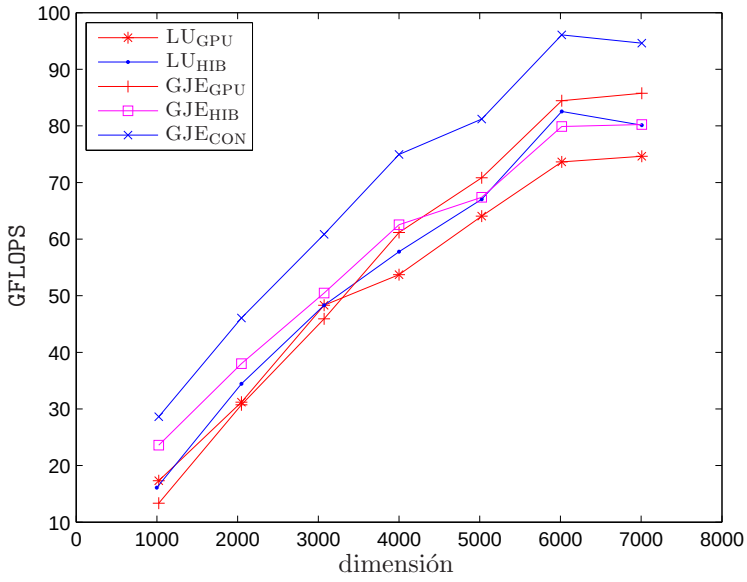

Figura 3.9: Desempeño (expresado en GFLOPS) alcanzado por las variantes de inversión de matrices más eficientes sobre HARRISON.

Versiones básicas sobre PECO

Este segundo conjunto de experimentos incluye las mismas pruebas que el presentado en el apartado anterior, pero sobre una plataforma con mayor capacidad de cómputo, PECO. El objetivo

de estos experimentos es corroborar que las tendencias obtenidas con equipos de bajo porte se mantienen en equipos de alta gama.

La Figura 3.10 muestra el desempeño, medido en GFLOPS, conseguidos por las implementaciones de inversión de matrices basadas en la factorización LU: $\mathrm{LU}_{\mathrm{CPU}}$, $\mathrm{LU}_{\mathrm{GPU}}$ y $\mathrm{LU}_{\mathrm{HIB}}$.

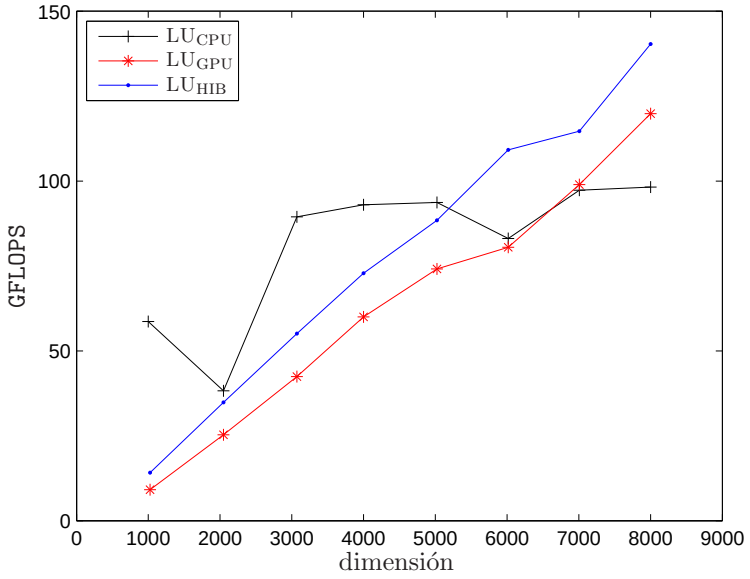

Figura 3.10: Desempeño (expresado en GFLOPS) alcanzado por las variantes de inversión de matrices basadas en la factorización LU sobre PECO.

La variante $\mathrm{LU}_{\mathrm{CPU}}$ alcanza los mejores resultados para matrices pequeñas y medianas. Esto se puede atribuir a la baja cantidad de cómputos necesarios para invertir matrices de esas dimensiones, que no permiten explotar las capacidades de las plataformas masivamente paralelas tipo GPU, y sí a la CPU de altas prestaciones disponible en PECO (Intel Xeon E5405 con 8 núcleos). La versión $\mathrm{LU}_{\mathrm{HIB}}$ es la que presenta mejores resultados para matrices grandes, ya que los problemas de gran dimensión implican montos importantes de operaciones de coma flotante, por lo cual se adaptan mejor a la arquitectura de las GPUs, y permiten conseguir mejores desempeños al ejecutar cada operación en la arquitectura más conveniente.

La Figura 3.11 presenta el desempeño para las implementaciones basadas en el método de GJE. Nuevamente, la implementación $\mathrm{GJE}_{\mathrm{CPU}}$ consigue los mejores resultados para matrices pequeñas, por las mismas razones expuestas anteriormente. Como se justificó en la Sección 3.1.2, los algoritmos tipo GJE se adaptan mejor que los algoritmos basados en la factorización LU a las arquitecturas masivamente paralelas, y como consecuencia, las implementaciones de GJE para GPU presentan mejor desempeño que las basadas en la factorización LU. La variante $\mathrm{GJE}_{\mathrm{HIB}}$ supera a la versión $\mathrm{GJE}_{\mathrm{GPU}}$ al operar sobre matrices pequeñas y medianas (ya que ejecuta cada operación en el dispositivo más conveniente), pero para matrices grandes, el costo de las operaciones de grano fino pasa a ser menos relevante y las ganancias obtenidas por el enfoque híbrido no se compensan con los costos de transferencia de datos. Finalmente, la versión $\mathrm{GJE}_{\mathrm{CON}}$ supera a las versiones anteriores para matrices medias y grandes gracias a la ejecución concurrente en ambas arquitecturas.

En resumen:

- Los códigos para GPU son notoriamente más rápidos en PECO que los códigos para CPU

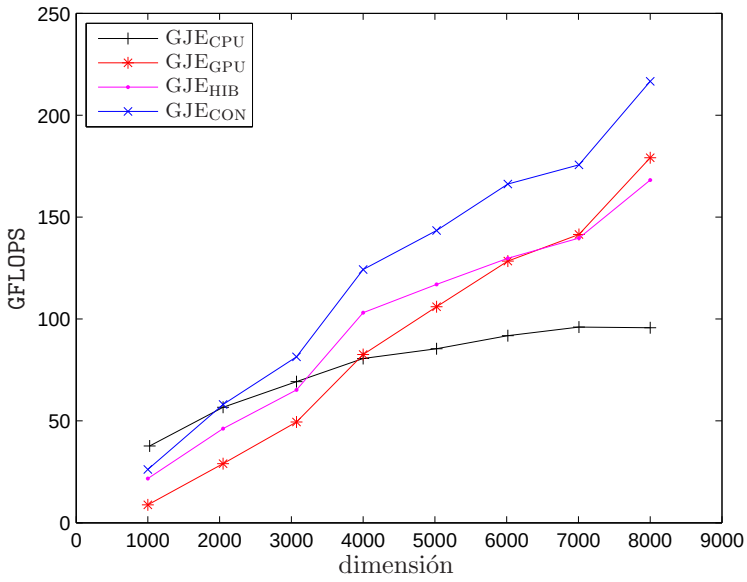

Figura 3.11: Desempeño (expresado en GFLOPS) alcanzado por las variantes de inversión de matrices basadas en el método de GJE sobre PECO.

cuando la dimensión de la matriz es suficientemente grande como para explotar la potencia computacional de la GPU.

- Las ganancias en GPU son menores para el método tradicional basado en la factorización LU, puesto que el método de GJE se adapta mejor a arquitecturas masivamente paralelas como es el caso de las GPUs.
- Las mejores implementaciones de cada algoritmo son las versiones híbridas, demostrando los beneficios de ejecutar cada tarea en el dispositivo que mejor se adapta para su cómputo, superior a los sobrecostos debidos a las comunicaciones.
- La versión GJE_{CON} mostró ser la mejor implementación: es dos veces más rápida que la implementación provista por LAPACK, alcanzando 220 GFLOPS para matrices de dimensión 8.000. Notar que, si bien el pico de desempeño de la GPU utilizada es de 933 GFLOPS, este valor es inalcanzable por la mayoría de las aplicaciones, incluida la inversión de matrices. Una referencia más realista de la capacidad de cómputo de la GPU es el pico de desempeño conseguido por la multiplicación de matrices. En este caso la implementación provista por NVIDIA incluida en la biblioteca CUBLAS alcanza los 350 GFLOPS sobre una GPU Tesla C1060 [218]. Una referencia más realista aún puede ser obtenida comparando las prestaciones con las alcanzadas por operaciones como la factorización LU [219] o la factorización de Cholesky [210], que son operaciones más similares a la inversión de matrices que la propia multiplicación de matrices.

Evaluación final

El tercer conjunto de experimentos incluye la experimentación final de las distintas versiones sobre plataformas de hardware que poseen GPUs de alta gama, PECO y ZAPE. En el primer caso se

utilizan matrices con dimensiones entre 1.000 y 14.000, mientras que en ZAPE se evalúa la inversión de matrices con dimensión entre 1.000 y 13.000, debido a que la cantidad de memoria disponible en esta GPU es algo menor (1,5 GB).

La Figura 3.12 presenta los resultados sobre la plataforma PECO. Las implementaciones basadas en el método de Gauss-Jordan presentan un rendimiento muy elevado, especialmente en la inversión de grandes matrices. La implementación altamente optimizada de LAPACK es claramente la mejor opción para matrices pequeñas, pero para matrices medianas y grandes es la implementación menos eficiente. Debido a la gran cantidad de núcleos de cómputo disponibles en las GPUs, las implementaciones sobre esta arquitectura son muy eficientes para la inversión de grandes matrices, pero el tiempo de transferencia CPU-GPU transforma en ineficiente el método para matrices pequeñas. Las propuestas híbridas ($\text{LU}_{\text{HIB+}}$ y GJE_{MN}) obtienen los mejores resultados para matrices con tamaños medios y grandes ($n > 2.000$). Estas rutinas logran explotar las capacidades de cada plataforma y, al mismo tiempo, mantienen controlado el sobrecosto introducido por las comunicaciones. Ambas implementaciones obtienen resultados similares para matrices pequeñas y medianas, pero la variante GJE_{MN} es más veloz para la inversión de matrices de gran dimensión ($n > 6.000$).

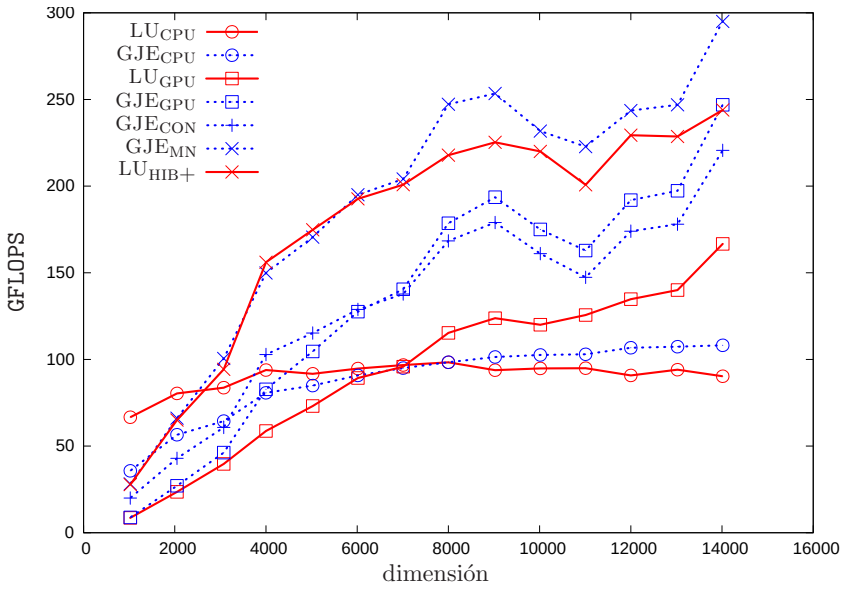

Figura 3.12: Desempeño (expresado en GFLOPS) alcanzado por las variantes de inversión de matrices sobre PECO.

La Figura 3.13 muestra los resultados obtenidos sobre la plataforma ZAPE. Nuevamente, las implementaciones híbridas obtuvieron los mejores resultados, y los algoritmos basados en GJE claramente superan las alternativas basadas en la factorización LU. Las diferencias entre las capacidades de la GPU y de la CPU en ZAPE implican que:

- Las implementaciones tanto híbridas como aquellas que utilizan únicamente la GPU son más rápidas que las implementaciones que únicamente emplean la CPU (excepto en la inversión de matrices muy pequeñas).
- La mejor implementación desarrollada es aproximadamente 10 veces más rápida en ZAPE que la versión basada en LAPACK (mientras que en PECO la mejor implementación es entre 3 y

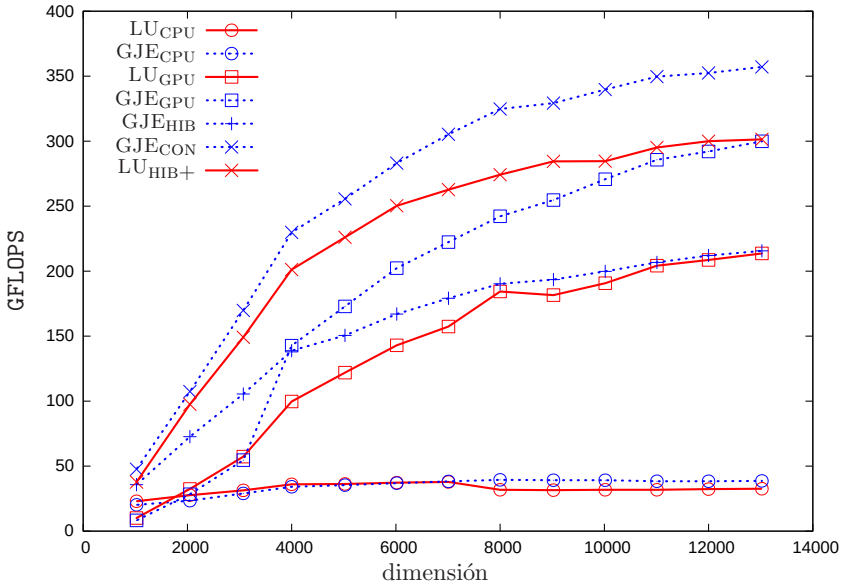

Figura 3.13: Desempeño (expresado en GFLOPS) alcanzado por las variantes de inversión de matrices sobre ZAPE.

4 veces más rápida).

En base a los resultados conseguidos se puede remarcar que la implementación GJE_{MN} supera claramente al resto de las implementaciones, excepto para la inversión de matrices de dimensión reducida en la plataforma PECO, donde LAPACK consigue los mejores desempeños. En definitiva, se puede concluir que la versión GJE_{MN} es capaz de adaptar su ejecución a la plataforma de hardware subyacente utilizada y a la dimensión de la matriz a invertir, brindando un alto desempeño en las diferentes plataformas y para los distintos tamaños de matrices evaluados.

Resumen de las evaluaciones

Se realizaron experimentos en tres plataformas de hardware híbridas distintas que incluyen CPUs y GPUs de capacidades computacionales dispares (*computer capabilities* 1.0, 1.2 y 2.0, concepto que se extiende en el Anexo A).

En cuanto a los resultados, se puede apreciar que para matrices pequeñas (dimensión ≤ 2.000) la implementación basada en la biblioteca MKL sobre CPU consigue los mejores resultados. En el caso de matrices medianas y grandes, las versiones que utilizan GPUs mostraron mejores desempeños que las versiones que utilizan únicamente la CPU. Entre las diferentes implementaciones evaluadas sobre la GPU, la versión híbrida GJE_{MN} ofrece los mejores resultados, puesto que ejecuta cada operación en la arquitectura más conveniente, realiza concurrentemente cómputos en ambos dispositivos, y es capaz de adaptar su ejecución a las características de ambas arquitecturas (gracias a la definición de dos tamaños de bloque).

3.2. Inversión de matrices simétricas y definidas positivas

Las matrices SDP poseen dos propiedades que permiten disminuir la cantidad de cálculos y movimientos de memoria necesarios para llevar adelante el proceso de inversión. Por un lado, la

simetría permite trabajar únicamente sobre la mitad de la matriz, reduciéndose efectivamente el número de cálculos significativamente, n^3 flops, si se comparan con los flops necesarios en los métodos basados en la factorización LU para matrices generales, $2n^3$ flops. Por otro lado, al ser matrices definidas positivas, no es necesario realizar el pivotamiento para preservar la estabilidad numérica del algoritmo.

Al igual que en el caso de las matrices generales, la inversión de matrices SDP puede ser abordada tanto con métodos que proceden en 3 etapas (factorización, inversión de una matriz triangular y producto de matrices triangulares) como basada en el método de GJE. En los siguientes apartados se describen ambas técnicas y se presentan las versiones desarrolladas para la inversión de matrices SDP empleando arquitecturas de cómputo que incluyen GPUs.

3.2.1. Métodos basados en la factorización de Cholesky

La estrategia tradicional para la inversión de una matriz SDP, $A \in \mathbb{R}^{n \times n}$, se basa en la factorización de Cholesky, que descompone la matriz como $A = LL^T$ (o $A = U^T U$), donde $L \in \mathbb{R}^{n \times n}$ es triangular inferior ($U \in \mathbb{R}^{n \times n}$ es triangular superior). En adelante, mostraremos cómo operar utilizando el factor triangular inferior L, aunque el trabajo presentado puede ser fácilmente aplicable al caso en que se opera con el factor triangular superior U. El proceso de inversión consiste en los siguientes tres pasos:

1. Calcular la factorización de Cholesky $A = LL^T$, donde $L \in \mathbb{R}^{n \times n}$ es una matriz triangular inferior.

2. Invertir el factor triangular $L \rightarrow L^{-1}$.

3. Calcular la matriz inversa como $A^{-1} = L^{-1}L^{-T}$.

LAPACK provee la funcionalidad requerida para los pasos antes descritos. En particular, la rutina POTRF implementa la factorización de Cholesky de una matriz SDP (paso 1), mientras que la rutina POTRI calcula la inversa de una matriz A usando la factorización de Cholesky obtenida mediante la invocación de la rutina POTRF (pasos 2–3).

En la Figura 3.14 se describe la mecánica de cómputo de la rutina POTRF de LAPACK para calcular la factorización de Cholesky de la matriz usando la notación FLAME [56, 116]. Con el símbolo "$\star$" se representan las secciones no accedidas de la matriz durante el algoritmo, y la etiqueta TRIL(M) significa que sólo se accede a la sección triangular inferior de la matriz M (TRIU(M) significa que sólo se accede a la sección triangular superior de la matriz).

El método CHOL_{UNB} invocado en el algoritmo de la figura, corresponde a la factorización de Cholesky escalar, y una implementación se puede consultar en [99].

El costo del algoritmo de inversión es n^3 flops. El método aprovecha la simetría de la matriz para reducir a la mitad el número de operaciones necesarias para el cómputo de la factorización con respecto al de factorización de matrices generales. También se pueden reducir los requerimientos de memoria utilizando métodos *in-place*, es decir, métodos que utilizan las matrices de entrada como espacio de trabajo y para la devolución de los resultados, o incluso más si se emplean técnicas de almacenamiento de matrices comprimidas [115]. El método se compone de tres pasos que tienen que ser ejecutados de forma secuencial; esto implica que el paralelismo únicamente se puede extraer dentro de la ejecución de cada paso.

En este apartado se describen tres implementaciones utilizando técnicas de HPC para este algoritmo. Todas las versiones son implementaciones *in-place*, lo que permite reducir los requerimientos de memoria.

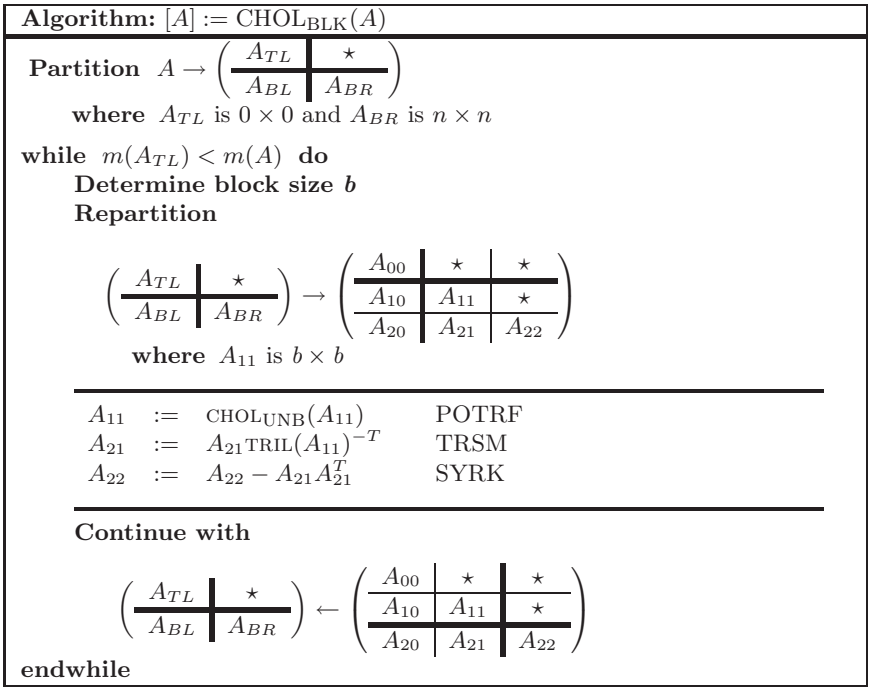

Figura 3.14: Algoritmo por bloques (CHOL_{BLK}) para la factorización de Cholesky de una matriz SDP.

Implementación en CPU multi-núcleo: CHOL_{CPU}**.**

La biblioteca Intel MKL [131] ofrece implementaciones multi-hilo de BLAS y LAPACK, que incluyen las rutinas paralelas necesarias para computar el algoritmo. La rutina POTRF implementa la factorización de Cholesky de una matriz SDP (paso 1), y la rutina POTRI computa la inversa de una matriz factorizada utilizando la rutina POTRF (pasos 2 y 3). El uso de versiones multi-hilo de estas bibliotecas permite introducir paralelismo y alcanzar niveles satisfactorios de eficiencia para la ejecución de las dos rutinas sobre CPUs multi-núcleo.

Implementación en GPU: CHOL_{GPU}**.**

Esta implementación es similar a la presentada previamente, pero diseñada para ser ejecutada sobre la GPU. Se desarrollaron las rutinas POTRF y POTRI para GPU, la implementación de ambas rutinas está basada en el uso de núcleos de la biblioteca CUBLAS de NVIDIA. Además de las dos rutinas comentadas, esta versión precisa de la funcionalidad ofrecida por las rutinas POTF2 (implementa la factorización de Cholesky escalar), TRTRI y TRTRI2 (esta última implementa la inversión de una matriz triangular en forma escalar), LAUUM (calcula la multiplicación de una matriz triangular por su traspuesta utilizando operaciones matriciales) y LAUU2 (calcula la multiplicación de una matriz triangular por su traspuesta utilizando operaciones escalares) de LAPACK, por lo cual también fue necesaria su implementación sobre la GPU.

La implementación CHOL_{GPU} requiere algunas transferencias entre el espacio de memoria de la CPU y de la GPU. En particular, la estructura de la rutina es la siguiente:

1. La matriz a invertir inicialmente es enviada desde el espacio de memoria del *host* al de GPU.

2. La matriz inversa es computada mediante las rutinas para GPU desarrolladas.

3. Finalmente la inversa es transferida desde la GPU a la CPU.

Destaca de esta versión que se ejecuta completamente en la GPU y utiliza estrategias de *padding*.

Implementación híbrida: CHOL_{HIB}**.**

Generalmente, la CPU es más eficiente que la GPU para ejecutar operaciones de grano fino, mientras que las GPUs superan a las CPUs en la ejecución de operaciones de gran dimensión y altamente paralelas. Si bien gran parte de las operaciones del método basado en la factorización de Cholesky son adecuadas para ser resueltas en GPU, ésta también incluye algunas operaciones de grano fino, especialmente durante el cómputo de la factorización de Cholesky del bloque actual, que no lo son. Por esta razón, en esta implementación la CPU y la GPU trabajan de forma coordinada, para calcular la factorización de Cholesky de la matriz, buscando que cada operación se ejecute en la plataforma que más se adapta a sus características, pero al mismo tiempo, buscando evitar el aumento improductivo de comunicaciones. La operativa de la rutina POTRI no cambia con respecto a la presentada en la versión CHOL_{GPU}. La estructura de la rutina es la siguiente:

1. Al comienzo enviar la matriz del espacio de memoria del *host* al de la GPU.
2. Enviar el bloque actual de columnas ($[A_{11}]$) a CPU y factorizarlo.
3. Devolver el bloque factorizado a GPU.
4. Actualizar la matriz activa en GPU con el bloque actual factorizado (resolución de sistema triangular y multiplicación de matrices).
5. Se repiten los pasos 2 al 4 hasta factorizar por completo la matriz.
6. Ejecutar la función POTRI desarrollada para GPU.
7. Transferir la matriz resultado del espacio de memoria de la GPU al del *host*.

3.2.2. Métodos basados en la eliminación de Gauss-Jordan

Como se mostró en el apartado anterior, la estrategia tradicional incluye tres pasos que tienen que ser ejecutados de forma secuencial. Por lo tanto, el método necesita recorrer la matriz al menos tres veces. Como resultado se incrementa el número de accesos a memoria y, en consecuencia, se limita el desempeño del algoritmo. Extrapolando ideas del método de Gauss-Jordan para matrices generales, en el trabajo de Bientinesi et al. [57] se derivan reordenaciones del método que permiten calcular la inversa recorriendo la matriz SDP una única vez. Las reordenaciones propuestas mantienen el uso eficiente de la estructura simétrica de la matriz y de la memoria obteniendo un algoritmo *in-place*.

Las Figuras 3.15 y 3.16 muestran dos algoritmos a bloques basados en las ideas propuestas en [57]. En ambos casos, únicamente la parte superior de la matriz inversa es computada y ésta sobreescribe la matriz inicial. En las figuras, a la derecha de cada operación del algoritmo, se especifica el nombre de la rutina de BLAS necesaria para su cómputo.

El algoritmo en la Figura 3.15 describe una variante en la que el bloque activo, A_{11}, se desplaza sobre la diagonal principal. En esta variante cada iteración del algoritmo implica a lo sumo ocho operaciones.

Tres factores pueden limitar el desempeño de una implementación paralela de este algoritmo. El primero es que, debido a las dependencias de datos entre los operadores, no es posible extraer

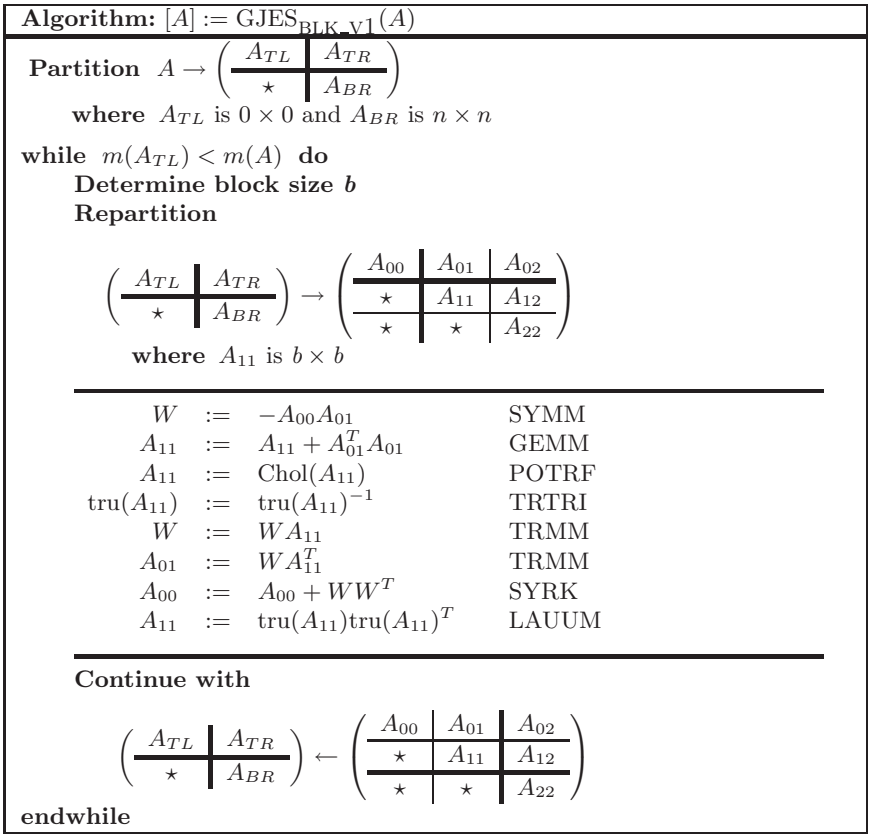

Figura 3.15: Algoritmo para invertir matrices SDP mediante la variante 1 del método de GJE.

paralelismo de la ejecución concurrente de operaciones. El segundo es que la mayoría de las actualizaciones implican trabajar con bloques de tamaño reducido (asumiendo que el tamaño de bloque b es pequeño en comparación con la dimensión de la matriz) y, por lo tanto, presentan también un costo computacional reducido. Por último, las operaciones se ejecutan a medida que evoluciona el método sobre matrices de distintos tamaños, lo que implica ciertos desequilibrios. Estas características limitan significativamente el nivel de paralelismo. Además, esta variante necesita un espacio de almacenamiento extra, W, de tamaño $b \times b$.

La Figura 3.16 muestra la segunda variante propuesta en [57], $\text{GJES}_{\text{BLK\_V2}}$, para la inversión basada en algoritmos de una barrida. En esta versión, todos los elementos de la parte superior de la matriz son actualizados en cada iteración. Esta característica implica que el esfuerzo computacional en cada iteración se mantenga constante durante todo el algoritmo. Al igual que en la variante anterior, la mayoría de las operaciones requieren su ejecución de forma secuencial, permitiendo únicamente explotar el paralelismo de grano fino inherente a cada una de la operaciones. La actualización de los paneles A_{00} y A_{22} concentra el mayor peso computacional del método, ya que el resto de las operaciones involucra tratar con bloques de datos de pequeña dimensión. En resumen, esta variante presenta, con respecto a la versión anterior, dos ventajas:

- No necesita espacio adicional de almacenamiento.
- El costo para completar cada iteración se mantiene constante durante la ejecución del algoritmo.

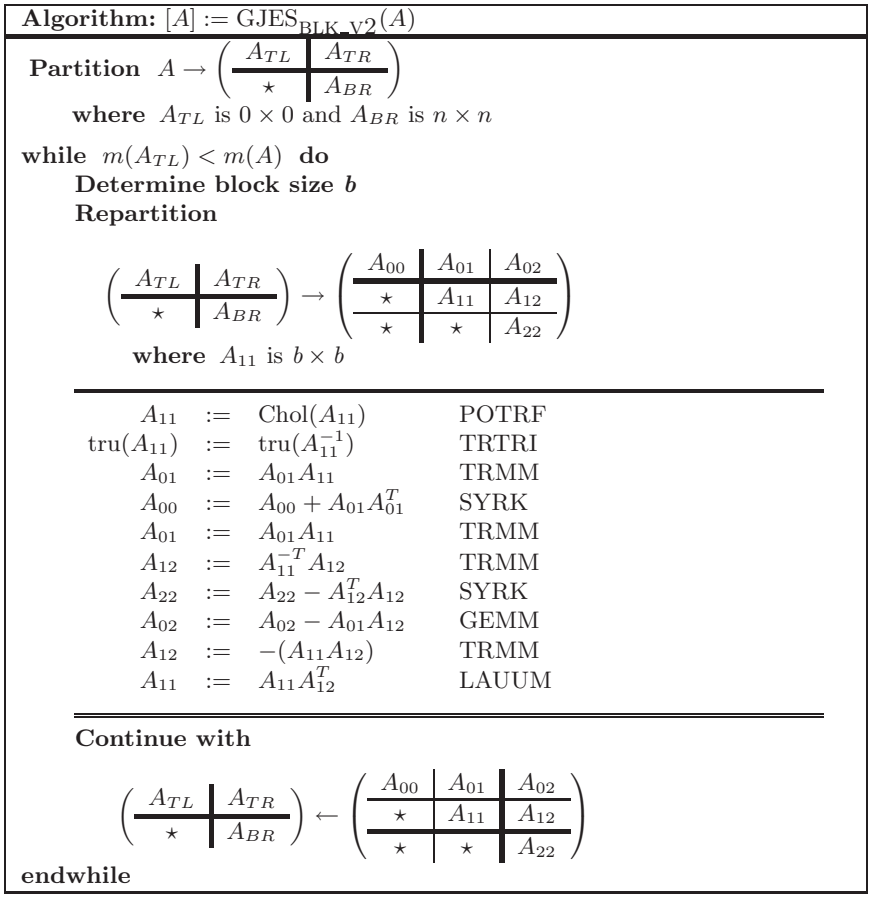

Figura 3.16: Algoritmo para invertir matrices SDP mediante la variante 2 del método de GJE.

Además, como se ha comentado previamente, este método tiene la particularidad de recorrer una única vez la matriz, disminuyendo las pérdidas de desempeño debido a una mala localidad de referencia de datos y accesos innecesarios a memoria.

Implementaciones en CPU: $\text{GJES}_{\text{CPU\_V1}}$ **y** $\text{GJES}_{\text{CPU\_V2}}$**.**

Las dos variantes basadas en el método de GJE para invertir matrices SDP presentadas anteriormente concentran la mayor parte del esfuerzo computacional en términos de productos matriciales (operación perteneciente al tercer nivel de BLAS). En particular, la operación que implica mayor costo es la actualización simétrica de rango k (rutina SYRK de BLAS). La biblioteca MKL brinda implementaciones de alto desempeño para este núcleo computacional y el resto de las operaciones presentes en los algoritmos $\text{GJES}_{\text{BLK\_V1}}$ y $\text{GJES}_{\text{BLK\_V2}}$.

Las rutinas $\text{GJES}_{\text{CPU\_V1}}$ y $\text{GJES}_{\text{CPU\_V2}}$ implementan estos algoritmos usando los núcleos computacionales de MKL. El paralelismo se obtiene gracias al uso de una implementación multihilo de la biblioteca.

Implementaciones en GPU: $\text{GJES}_{\text{GPU\_V1}}$ **y** $\text{GJES}_{\text{GPU\_V2}}$**.**

Dado que la mayoría de las operaciones de los algoritmos basados en el método de GJE para la inversión de matrices SDP pueden ser expresadas en términos de multiplicaciones de matrices,

es de esperar que la implementación del algoritmo en una arquitectura masivamente paralela (como por ejemplo una GPU) alcance altos niveles de desempeño. A continuación se describen las implementaciones sobre GPU de los algoritmos $\text{GJES}_{\text{BLK\_V1}}$ y $\text{GJES}_{\text{BLK\_V2}}$ utilizando técnicas de HPC.

Para la construcción de ambos algoritmos es necesario invocar a las rutinas TRTRI y LAUUM implementadas sobre GPU, descritas al presentar la implementación CHOL_{GPU}. Estas rutinas se emplean en la actualización del bloque A_{11}, que es relativamente pequeño. Por esta razón, es difícil conseguir grandes beneficios al computar estas operaciones en la GPU. Sin embargo, el objetivo de invocar estas rutinas y ejecutar las tareas de grano fino en una GPU es reducir el monto de transferencia de datos entre el espacio de memoria de la GPU y la CPU.

Implementación híbrida: GJES_{HIB}.

Es esperable que la implementación en GPU de los algoritmos basados en GJE para invertir matrices SDP alcance altos niveles de desempeño debido al uso de núcleos computacionales de la biblioteca CUBLAS para el cómputo de operaciones de las tipo multiplicación de matrices. Sin embargo, algunas operaciones presentes en el algoritmo no se adaptan totalmente a las características de las GPUs, en concreto la actualización del bloque A_{11}. Por el contrario, sí se adapta su ejecución a las características de las CPUs y, además, se dispone de implementaciones de estas operaciones altamente optimizadas sobre dicha arquitectura. Por esta razón, se diseñó una versión híbrida que implementa el Algoritmo $\text{GJES}_{\text{BLK\_V2}}$ ejecutando la actualización del bloque A_{11} en CPU y el resto de las operaciones en GPU. En la Figura 3.17 se presenta de forma algorítmica la implementación híbrida.

En esta implementación, cada operación es ejecutada en el dispositivo más conveniente, intentando moderar el tamaño y número de transferencias entre las memorias de ambos dispositivos, buscando así alcanzar el máximo desempeño para cada operación, pero manteniendo el tiempo de comunicaciones bajo. En particular, en cada iteración son necesarias dos transferencias que implican el movimiento de un número reducido de datos, el bloque A_{11}. También se implementó de forma híbrida la variante 1 del algoritmo; sin embargo, en los experimentos (no formalizados) esta variante no ofreció beneficios al compararla con GJES_{HIB}.

Implementación concurrente: GJES_{CON}.

Esta versión, además de realizar cálculos de forma híbrida sobre CPU y GPU, busca la ejecución concurrente en ambas arquitecturas. Es decir, cada operación se procesa sobre la plataforma más adecuada y, además, se computan operaciones en ambas arquitecturas de forma concurrente. La Figura 3.18 describe el algoritmo.

En cada iteración son necesarias tres transferencias de datos, pero cada una de las transferencias implica el movimiento de pocos datos, el bloque A_{11}. Estas transferencias permiten ejecutar cada operación en el hardware que se adapta mejor, permitiendo mejorar el desempeño, y posibilitando además la ejecución de operaciones en forma concurrente en CPU y GPU, agregando así otro nivel de paralelismo y, por ende, obteniendo una mejora en el uso de los recursos de cómputo.

3.2.3. Análisis experimental para matrices SDP

En este apartado se presenta la evaluación experimental de las diferentes versiones implementadas para la inversión de matrices SDP. Las versiones evaluadas son:

- Variantes basadas en la factorización de Cholesky: CHOL_{CPU}, CHOL_{GPU} y CHOL_{HIB}.

Algorithm: $[A] := \mathrm{GJES}_{\mathrm{HIB}}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right)$

where A_{TL} is 0×0 and A_{BR} is $n \times n$

while $m(A_{TL}) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

A_{11}	:=	$\mathrm{Chol}(A_{11})$	(en CPU)
$\mathrm{tru}(A_{11})$	:=	$\mathrm{tru}(A_{11}^{-1})$	(en CPU)
CPU	→	GPU (A_{11})	
A_{01}	:=	$A_{01}A_{11}$	(en GPU)
A_{00}	:=	$A_{00} + A_{01}A_{01}^T$	(en GPU)
A_{01}	:=	$A_{01}A_{11}$	(en GPU)
A_{12}	:=	$A_{11}^{-T}A_{12}$	(en GPU)
A_{22}	:=	$A_{22} - A_{12}^T A_{12}$	(en GPU)
A_{02}	:=	$A_{02} - A_{01}A_{12}$	(en GPU)
A_{12}	:=	$-(A_{11}A_{12})$	(en GPU)
A_{11}	:=	$A_{11}A_{12}^T$	(en CPU)
GPU	→	CPU (A_{11})	

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$$

endwhile

Figura 3.17: Implementación híbrida para la inversión de matrices SDP mediante la variante 2 del algoritmo de GJE sobre una arquitectura híbrida, compuesta por una CPU multinúcleo y una GPU.

- Variantes basadas en el método de GJE: $\mathrm{GJES}_{\mathrm{CPU\_V1}}$, $\mathrm{GJES}_{\mathrm{CPU\_V2}}$, $\mathrm{GJES}_{\mathrm{GPU\_V1}}$, $\mathrm{GJES}_{\mathrm{GPU\_V2}}$, $\mathrm{GJES}_{\mathrm{HIB}}$ y $\mathrm{GJES}_{\mathrm{CON}}$.

La evaluación de los núcleos para invertir matrices SDP implementados se desarrolló sobre la plataforma YUCA (ver Apartado 1.3.1). En primera instancia, y dado que se disponía de una plataforma donde la CPU dispone con gran cantidad de núcleos de cómputo (32), se evaluó el desempeño y escalabilidad de las diferentes implementaciones desarrolladas para CPU. Posteriormente, se estudiaron las versiones desarrolladas sobre GPU.

Los experimentos se realizaron utilizando aritmética de simple precisión para matrices cuya dimensión varía entre 1.000 y 15.000. También se evaluaron los algoritmos con diversos tamaños de bloque (32, 64, 128, 256, 512 y 1.024) pero, por simplicidad, solo se presentan los resultados obtenidos con el mejor tamaño de bloque en cada caso.

Evaluación de implementaciones sobre CPUs

La Figura 3.19 muestra el desempeño alcanzado usando 32 hilos (un hilo por núcleo disponible en la plataforma) por las implementaciones sobre CPU basadas en LAPACK y en GJE descritas en los

Algorithm: $[A] := \text{GJES}_{\text{CON}}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right)$
where A_{TL} is 0×0 and A_{BR} is $n \times n$

while $m(A_{TL}) < m(A)$ **do**
Determine block size b
Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

GPU	$\rightarrow$	CPU (A_{11})	
A_{11}	$:=$	$\text{Chol}(A_{11})$	(en CPU)
$\text{tru}(A_{11})$	$:=$	$\text{tru}(A_{11}^{-1})$	(en CPU)
CPU	$\rightarrow$	GPU (A_{11})	
A_{01}	$:=$	$A_{01}A_{11}$	(en GPU)
A_{00}	$:=$	$A_{00} + A_{01}A_{01}^T$	(en GPU)
A_{01}	$:=$	$A_{01}A_{11}$	(en GPU)
A_{12}	$:=$	$A_{11}^{-T}A_{12}$	(en GPU)
A_{22}	$:=$	$A_{22} - A_{12}^T A_{12}$	(en GPU)
A_{02}	$:=$	$A_{02} - A_{01}A_{12}$	(en GPU)
A_{12}	$:=$	$-(A_{11}A_{12})$	(en GPU)
A_{11}	$:=$	$A_{11}A_{12}^T$	(en CPU)
CPU	$\rightarrow$	GPU (A_{11})	

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$$

endwhile

Figura 3.18: Implementación híbrida y concurrente para la inversión de matrices SDP mediante la variante 2 del algoritmo de GJE sobre una arquitectura híbrida, compuesta por una CPU multinúcleo y una GPU.

apartados 3.2.1 y 3.2.2. La implementación de la primera variante, $\text{GJES}_{\text{CPU\_V1}}$, es notoriamente más eficiente que las basadas en LAPACK, especialmente para matrices grandes. Por ejemplo, para matrices de dimensión 15.000, es aproximadamente 3 veces más rápida. Aun así, el mejor desempeño es obtenido por la implementación de la variante 2 del método ($\text{GJES}_{\text{CPU\_V2}}$). Ésta supera los 300 GFLOPS para matrices de dimensión 15.000, siendo más de 4 veces más rápida que la versión basada en LAPACK. En resumen, ambas implementaciones basadas en el método de GJE ofrecen mejores desempeños que LAPACK pero, dadas sus propiedades, la segunda versión consigue mejores niveles de paralelismo y, por lo tanto, mejores prestaciones.

Una explicación para las grandes diferencias entre los desempeños alcanzados por las variantes basadas en GJE frente a la basada en LAPACK es la mejor adaptación de las primeras al procesamiento paralelo con procesadores multi-núcleo (multi-hilo), pudiendo explotar mejor las 32 unidades de procesamiento disponibles en la plataforma. En este sentido, la Figura 3.20 muestra los resultados para la implementación de la versión $\text{GJES}_{\text{CPU\_V1}}$ utilizando 1, 2, 4, 8, 16 y 32 hilos (izquierda) y los resultados del experimento análogo para la versión $\text{GJES}_{\text{CPU\_V2}}$ (derecha), mientras que la Figura 3.21 presenta la misma evaluación para la rutina CHOL_{CPU}.

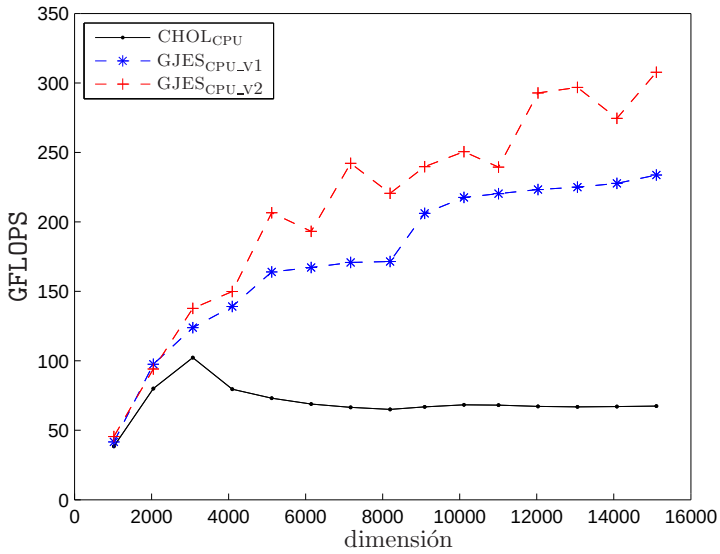

Figura 3.19: Desempeño (expresado en GFLOPS) alcanzado por las variantes de inversión de matrices SDP en CPUs sobre YUCA.

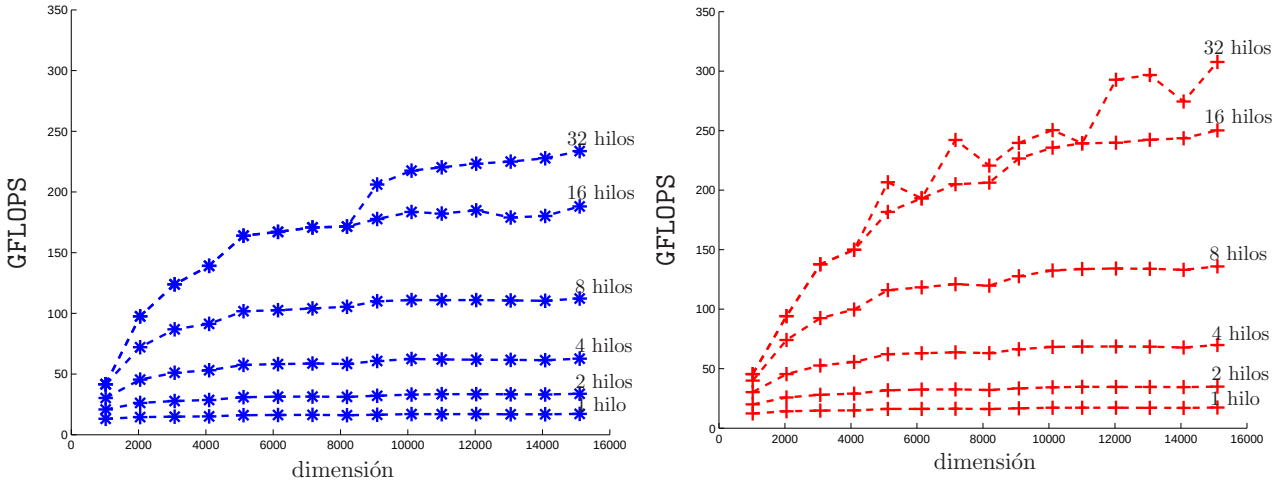

Figura 3.20: Evolución del desempeño de la inversión de matrices al emplear distinta cantidad de hilos con las versiones $\text{GJES}_{\text{CPU\_v1}}$ (izquierda) y $\text{GJES}_{\text{CPU\_v2}}$ (derecha) sobre YUCA.

Observando las figuras, se puede notar que el uso de más hilos incrementa las prestaciones de las implementaciones basadas en GJE considerablemente, excepto para la inversión de matrices de dimensión 8.000 usando más de 16 hilos. Este resultado demuestra la escalabilidad de las versiones basadas en GJE, que consiguen aumentar su rendimiento de manera casi lineal al aumentar los hilos (hasta el último incremento, que muestra cierta saturación). Estos resultados se diferencian ampliamente de los conseguidos por la versión basada en LAPACK en los que, al duplicar los hilos a partir de 4, el rendimiento crece de forma marginal.

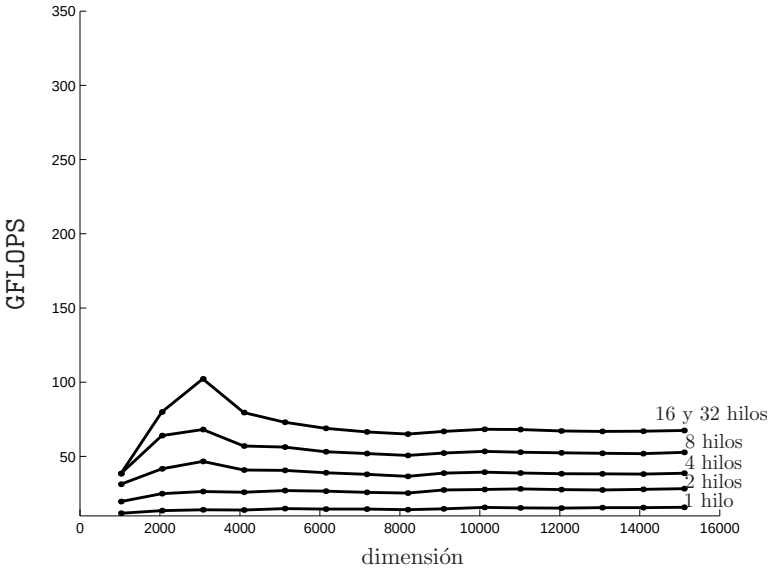

Figura 3.21: Evolución del desempeño de la inversión de matrices al emplear distinta cantidad de hilos con CHOL_{CPU} sobre YUCA.

Evaluación de implementaciones sobre GPU

En primera instancia se evalúan las variantes implementadas sobre GPU basadas en la factorización de Cholesky. La Figura 3.22 muestra los desempeños alcanzados por las tres implementaciones presentadas utilizando la GPU y los 32 núcleos.

Se puede observar que la versión basada en LAPACK, CHOL_{CPU}, obtiene los mejores resultados (luego de alcanzar el desempeño máximo absoluto para matrices de dimensión 4.000) para matrices de dimensión hasta 10.000, mientras que para matrices de dimensiones mayores las variantes que emplean el procesador gráfico superan ampliamente el desempeño alcanzado por CHOL_{CPU}. El enfoque híbrido incrementa el desempeño, superando en aproximadamente un 80 % el enfoque tradicional basado en LAPACK para matrices de dimensión 15.000. Es más, mientras que la versión CHOL_{CPU} parece alcanzar el máximo (local) desempeño para matrices de dimensión 10.000[1], el desempeño de las implementaciones basadas en GPU continúa creciendo incluso para las mayores matrices evaluadas en los experimentos (15.000). A partir de lo anterior, se puede afirmar que el uso de arquitecturas masivamente paralelas, como por ejemplo GPUs, permite mejorar el rendimiento de este tipo de algoritmos para computar matrices de grandes dimensiones.

La Figura 3.23 muestra el desempeño alcanzado por las diferentes implementaciones basadas en el método de GJE. Los mayores niveles de GFLOPS sobre matrices de gran dimensión (mayor que 8.000) son alcanzados por la variante híbrida y concurrente (GJES_{CON}). Como se observó en el apartado anterior, las implementaciones multi-núcleo basadas en GJE también alcanzan buenos desempeños, y en particular, son la mejor opción para matrices medianas (dimensión entre 4.000 y 8.000). En contraste, las implementaciones que utilizan únicamente la GPU ofrecen un desempeño pobre. Esta situación es debida, presumiblemente, al gran número de operaciones de grano fino presentes; estas operaciones no son adecuadas para su ejecución en arquitecturas masivamente paralelas como las GPUs, lo que degrada las prestaciones de estas rutinas. Por otro lado, el procesamiento multi-núcleo sobre CPU es eficiente a la hora de resolver este tipo de operaciones de grano fino a

[1] Luego de obtener un máximo absoluto para matrices de dimensión 3.000.

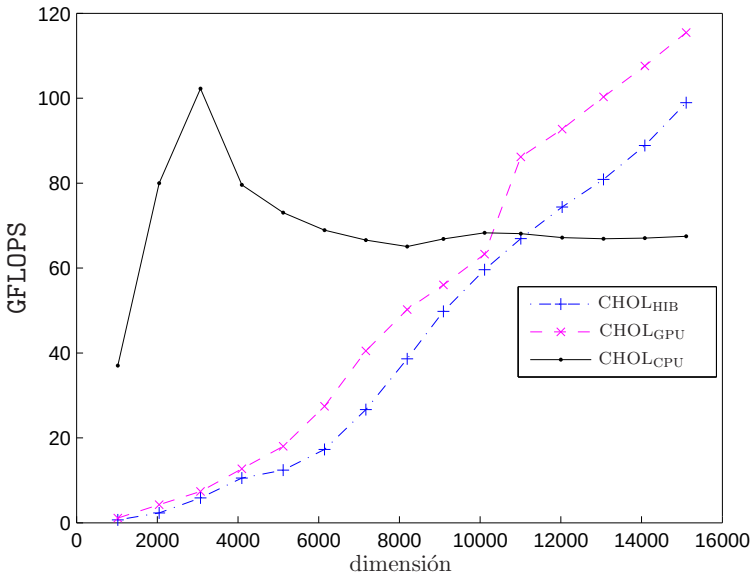

Figura 3.22: Desempeño (expresado en GFLOPS) alcanzado por las variantes de inversión de matrices SDP basadas en la factorización de Cholesky sobre YUCA.

bloques, aunque no es tan eficiente para resolver operaciones que involucran grandes matrices.

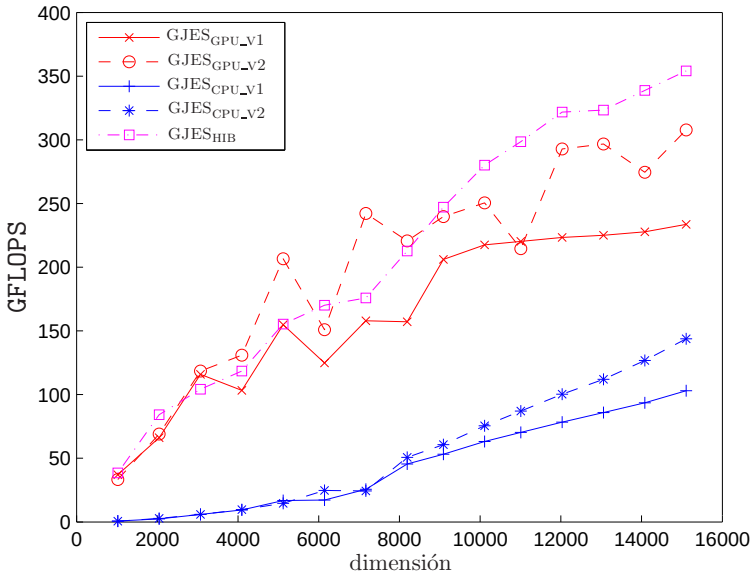

Figura 3.23: Desempeño (expresado en GFLOPS) alcanzado por las variantes de inversión de matrices SDP basadas en GJE sobre YUCA.

Se puede ver en los resultados resumidos en la Figura 3.24 que la implementación $\text{GJES}_{\text{CPU\_v2}}$ alcanza los mejores desempeños utilizando tamaños de bloque grande (256 ó 512), mientras que la

versión $\text{GJES}_{\text{GPU\_V2}}$ alcanza los mejores resultados utilizando tamaños de bloque más pequeños (de 32 a 256). Cuando el tamaño de bloque se reduce, el tiempo de ejecución de las operaciones de bajo costo también decrece. Esta circunstancia permite a los algoritmos híbridos beneficiarse de ambos contextos de ejecución, empleando un tamaño de bloque intermedio, entre 64 y 256, que permita alcanzar una solución de compromiso.

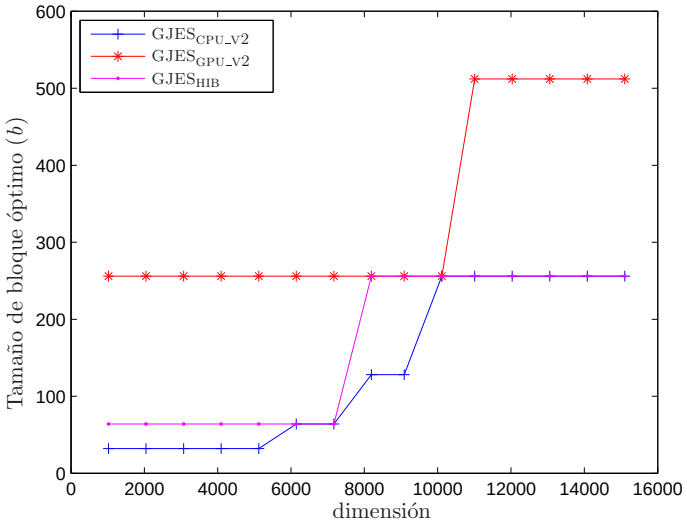

Figura 3.24: Tamaño de bloque óptimo para las implementaciones del método de GJE de inversión de matrices sobre YUCA.

Finalmente, la Figura 3.25 muestra el desempeño de la implementación basada en LAPACK y la mejor implementación para cada arquitectura. Es importante notar que, para ambas arquitecturas, la mejor implementación se basa en los algoritmos tipo GJE. La rutina $\text{GJES}_{\text{GPU\_V2}}$ supera a la variante basada en LAPACK cuando la dimensión de la matriz a invertir es mayor que 7.000, mientras que, con matrices pequeñas, el sobrecosto debido a la transferencia de datos y la ejecución ineficiente de las operaciones de grano fino reducen el desempeño del método de GJE en GPU. Por otro lado, las implementaciones multi-núcleo del método GJE ofrecen un comportamiento irregular, pero que claramente supera a la versión basada en la factorización de Cholesky, CHOL_{CPU}. Para matrices grandes la implementación más rápida es la híbrida y concurrente, GJES_{CON}, ya que a pesar del sobrecosto introducido por el aumento en el número de comunicaciones, esta versión permite ejecutar cada operación en la arquitectura que mejores prestaciones alcanza y además, computa operaciones en forma concurrente en ambas plataformas. Es importante resaltar la escalabilidad mostrada por la variante GJES_{CON}, y en general, por las diferentes implementaciones basadas en el método de GJE.

Resumen de las evaluaciones para matrices SDP

De los resultados obtenidos en los experimentos realizados, se pueden extraer las siguientes concluciones:

- Las implementaciones basadas en el método de GJE mostraron mejores desempeños en contextos de múltiples unidades de cómputo, tanto en arquitectura multi-núcleo como en plataformas híbridas CPU-GPU.

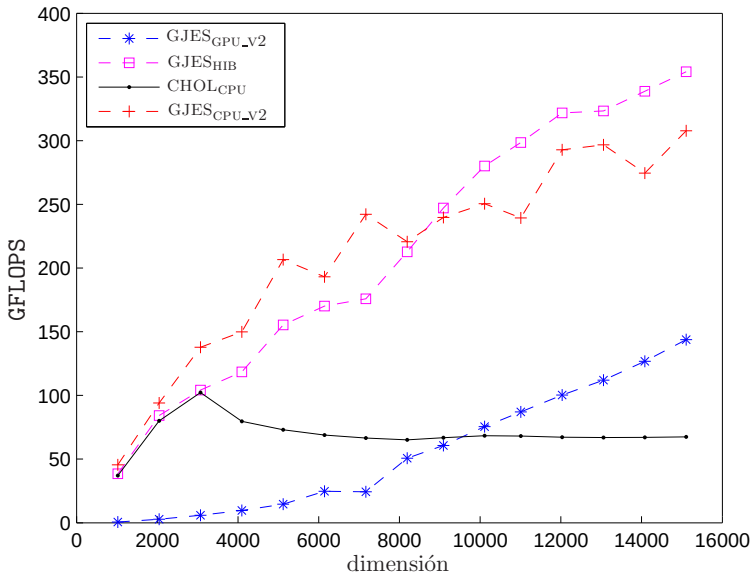

Figura 3.25: Desempeño (expresado en GFLOPS) alcanzado por las variantes de inversión de matrices SDP sobre YUCA.

- Las versiones basadas en el método de GJE mostraron buena capacidad de escalabilidad sobre las plataformas evaluadas.
- La versión híbrida que utiliza de forma concurrente tanto la CPU como la GPU (GJES$_{CON}$) mostró los mejores desempeños, superando en un factor mayor a 5× los de la versión basada en LAPACK para matrices de dimensión 15.000.

3.3. Inversión de matrices con múltiples GPUs

Existen dos objetivos principales para el uso de múltiples GPUs al computar la inversión de matrices. Por un lado, utilizar mayor poder de cómputo permite (al menos teóricamente) disminuir el tiempo de ejecución; por otro lado, utilizar varias GPUs de forma coordinada habilita la inversión de matrices con mayor dimensión, dado que se puede sacar provecho de las memorias de cada GPU para almacenar distintas secciones de la matriz. Para alcanzar el primer objetivo, es necesario explotar en forma eficiente el poder de cómputo de las distintas GPUs y controlar el volumen de envíos de datos y cómputos extra necesarios para coordinar las distintas GPUs. El segundo objetivo implica aprovechar realmente el espacio de memoria de las diferentes GPUs utilizadas, restringiendo en lo posible la replicación de datos y el uso de espacio extra de almacenamiento. Por ejemplo, si se utiliza una tarjeta NVIDIA C1060 que tiene 4 GB de memoria y permite almacenar matrices cuadradas (con números de simple precisión) de dimensión cercana a $32K$, con 4 GPUs (por ejemplo, utilizando una NVIDIA Tesla S1070), la memoria disponible se incrementa hasta los 16 GB, permitiendo duplicar la dimensión de la matriz a procesar (recordar el orden cuadrático en lo que respecta al costo espacial).

En esta sección se presentan las diferentes implementaciones del método de GJE desarrolladas para invertir matrices generales explotando el uso de múltiples GPUs conectadas a un mismo

host. Las implementaciones buscan alcanzar los dos objetivos descritos anteriormente, acelerar la resolución y escalar en la dimensión de los problemas tratables.

Un aspecto importante para poder utilizar múltiples GPUs conectadas a un único computador es la necesidad de emplear varios hilos de ejecución o procesos en la CPU, al menos uno por cada GPU que se desee utilizar. Por ello, en todos los desarrollos presentados en esta sección se utiliza OpenMP (presentada en el Capítulo 1) para cubrir este aspecto.

3.3.1. Implementaciones sobre múltiples GPUs

Las diferentes implementaciones presentadas en esta sección se basan en las conclusiones extraídas a partir del estudio sobre la inversión de matrices generales utilizando una única GPU. Sin embargo, hay particularidades derivadas de disponer de varias GPUs que afectan directamente al diseño de la solución; por ejemplo, se debe estudiar cuidadosamente la planificación de tareas y comunicaciones. A continuación se detallan las diferentes implementaciones desarrolladas.

Primera aproximación: $\mathrm{MGGJE_{CON}}$.

Esta versión para invertir matrices es simplemente la extrapolación al contexto de múltiples GPUs de la versión sobre una única GPU de GJE, $\mathrm{GJE_{CON}}$. La mecánica de la versión es la siguiente:

1. La matriz, de dimensión n, es dividida uniformemente en n_G paneles de columnas, donde n_G es el número de GPUs disponibles. Cada panel de columnas es enviado a una GPU, la primeras $Cant_{col}$ columnas a la primera GPU, las siguientes $Cant_{col}$ columnas a la segunda GPU y así sucesivamente, donde $Cant_{col}$ se define como $\frac{n}{n_G}$ (por simplicidad, se asume que n es un múltiplo exacto de n_G). La Figura 3.26 muestra la distribución sobre 4 GPUs donde cada GPU almacena un cuarto de las columnas de la matriz. Este reparto se conoce habitualmente como distribución por bloques de columnas.

2. El núcleo de cómputo en la CPU trabaja de igual forma que en la versión sobre una GPU, $\mathrm{GJE_{CON}}$, pero en diferentes pasos del algoritmo el bloque actual de columnas reside en una GPU diferente (en el ejemplo mostrado en la Figura 3.26 en la GPU 2). Esto implica que, en cada paso, la CPU debe tomar el bloque actual de columnas desde una GPU específica, aquella que está procesando el bloque actual de columnas.

3. Una vez actualizado en CPU el bloque actual de columnas, éste es copiado a todas las GPUs.

4. La GPU que procesa el bloque de columnas siguiente al actual, actualiza su panel y envía el siguiente bloque de columnas actual a la CPU. El resto de las GPUs simplemente actualizan el conjunto de columnas que almacenan.

5. Por último, cada GPU devuelve su trozo de matriz (inversa) al *host* (y se deshace la permutación de columnas, derivada de la estrategia de pivotamiento, en CPU).

Con esta estrategia de ejecución la sincronización entre las diferentes unidades de cómputo, las distintas GPUs y la CPU, se resume en sincronizar la CPU y una única GPU en cada iteración del algoritmo, aquella donde está alojado el bloque actual de columnas.

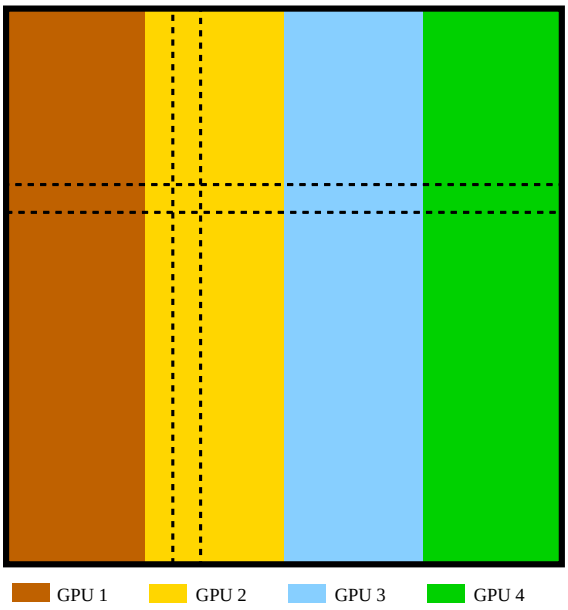

Figura 3.26: Distribución de datos a bloque para el método de GJE sobre 4 GPUs.

En términos de uso de memoria, la versión propuesta implica que cada GPU mantiene su fracción de matriz. Además, para realizar la actualización de las columnas de cada GPU es necesario transferir el bloque actual de columnas a todas las GPUs. En consecuencia, la rutina necesita un espacio auxiliar de memoria, en cada GPU, de tamaño igual a $n \times b$ donde b es el tamaño de bloque. El espacio de memoria en las GPUs necesario para invertir una matriz de dimensiones $n \times n$ utilizando n_G GPUs totaliza:

$$(n \times n) + (n_G \times n \times b). \tag{3.1}$$

Para obtener la cantidad de bytes en caso de trabajar con números en simple precisión es necesario multiplicar por 4 y en doble por 8.

Inclusión de estrategias de Look-Ahead: $\mathrm{MGGJE_{LA}}$**.**

Observando los resultados presentados en el Apartado 3.1.3 para invertir matrices con una única GPU, una primera mejora al algoritmo sobre múltiples GPUs es reordenar los cálculos para mejorar el solapamiento incluyendo estrategias de *look-ahead* (ver Algoritmo $\mathrm{GJE_{LA}}$).

Esta versión es similar a la anterior, $\mathrm{MGGJE_{CON}}$, pero la GPU que almacena el bloque de columnas $[A_{02};\ A_{12};\ A_{22}]$ actualizará en primer lugar dicho bloque de columnas y lo enviará a la CPU, para después actualizar el resto de columnas del panel que almacena. Así se obtiene un mayor solapamiento de cómputo entre la CPU y las distintas GPUs.

Inclusión de estrategias multi-nivel: $\mathrm{MGGJE_{MN}}$.

Otra mejora utilizada en el contexto de inversión de matrices con una única GPU es la utilización de un núcleo de actualización por bloques del bloque actual de columnas en CPU. La inclusión de esta estrategia al utilizar una GPU mostró apreciables mejoras. Además, en el contexto de múltiples GPUs, el desequilibrio entre el tamaño de bloque óptimo de las etapas resueltas en CPU y en GPU se puede ver agudizado, ya que la posibilidad de abordar la inversión de matrices con dimensión mayor potencia la problemática.

Específicamente, esta implementación difiere de la versión anterior en la forma de procesar el bloque actual ($[A_{01};\ A_{11};\ A_{21}]$) en CPU. Como en la versión $\mathrm{MGGJE_{LA}}$, el Algoritmo $\mathrm{GJE_{BLK}}$ es aplicado a la matriz A, pero en este caso la CPU procesa el bloque de columnas $[A_{01};\ A_{11};\ A_{21}]$ usando una instancia del método de GJE por bloques. Así, la CPU ejecuta el método de GJE a bloques sobre $[A_{01};\ A_{11};\ A_{21}]$ usando el tamaño de bloque óptimo para este hardware, mientras que en un nivel superior, el Algoritmo $\mathrm{MGGJE_{LA}}$ es ejecutado con el tamaño de bloque óptimo para las GPUs.

Implementación con distribución cíclica de datos $\mathrm{MGGJE_{DC}}$.

Si se estudian los tiempos de ejecución de la CPU y las distintas GPUs, en la versión anterior, se puede observar que:

- Hay un desequilibrio en los tiempos de ejecución entre las distintas GPUs en cada paso de la iteración, ya que la GPU que está calculando el bloque de columnas $[A_{02};\ A_{12};\ A_{22}]$ emplea un mayor tiempo de ejecución que el resto.
- Por el tipo de distribución de datos empleada, cada GPU realiza durante al menos t pasos consecutivos la misma tarea, donde $t = Cant_{col}/b$. Por ejemplo, durante t pasos la misma GPU almacena y actualiza el bloque de columnas $[A_{02};\ A_{12};\ A_{22}]$; de esta manera el camino crítico del algoritmo está dominado en cada paso por la GPU con más trabajo. Empleando por ejemplo 4 GPUs, en la mayoría de los casos habrá una GPU transfiriendo el bloque actual de columnas y resolviendo los cálculos correspondientes al *look-ahead*, y las otras 3 tarjetas gráficas estarán únicamente actualizando paneles de columnas que no afectan el camino crítico del algoritmo. Solo hay tres excepciones a lo anterior, cuando el bloque actual es el último alojado en una GPU por lo cual la actualización del bloque actual se realiza en una GPU, y la actualización del siguiente bloque de columnas ($[A_{02};\ A_{12};\ A_{22}]$) lo realiza la siguiente GPU.

Una primera aproximación para resolver la problemática descrita anteriormente es utilizar una distribución de datos cíclica por bloques. De esta manera, el primer bloque de b columnas está almacenado en la primera GPU, las siguientes b columnas en la segunda GPU, y así sucesivamente. La Figura 3.27 ejemplifica la distribución cíclica de bloques de columnas sobre 4 GPUs.

Con la nueva distribución de datos, el procesamiento del segundo bloque de columnas en GPU puede comenzar una vez que la CPU factorizó el segundo bloque de columnas; en cuanto al tiempo de procesamiento esto implica el mayor entre 1) que la CPU factorice el primer bloque de columnas, la primera GPU calcule el *look-ahead* y la CPU factorice el segundo bloque y 2) el tiempo para factorizar el primer bloque en CPU más el tiempo de la siguiente GPU (en el ejemplo, la GPU 2) en procesar todo su panel. En contraposición, para comenzar con el segundo bloque de columnas la versión anterior necesitaba que la CPU computara el primer bloque más el tiempo de la GPU actual (en el ejemplo, la GPU 1) para computar el *look-ahead* y enviar el bloque a CPU, y además

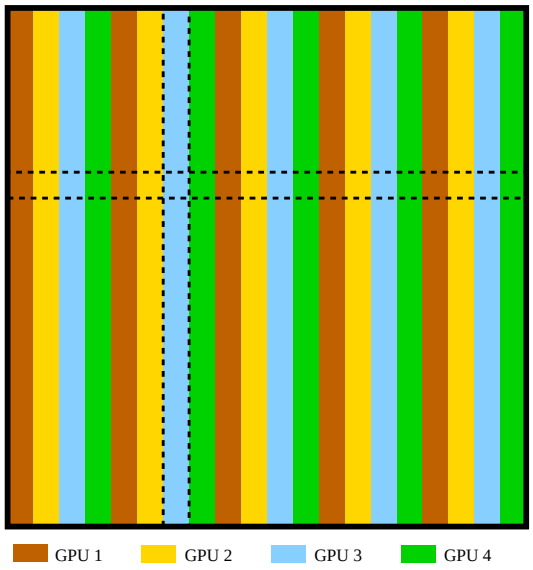

Figura 3.27: Distribución de datos cíclica para el método de GJE sobre 4 GPUs.

actualizar todo el panel (asociado al *look ahead*), lo que implica un tiempo mayor. La Figura 3.28 ilustra esta situación mediante un diagrama de planificación de tareas ejecutadas sobre dos GPUs y una CPU, donde se realiza una distribución por bloques (arriba) y cíclica (abajo).

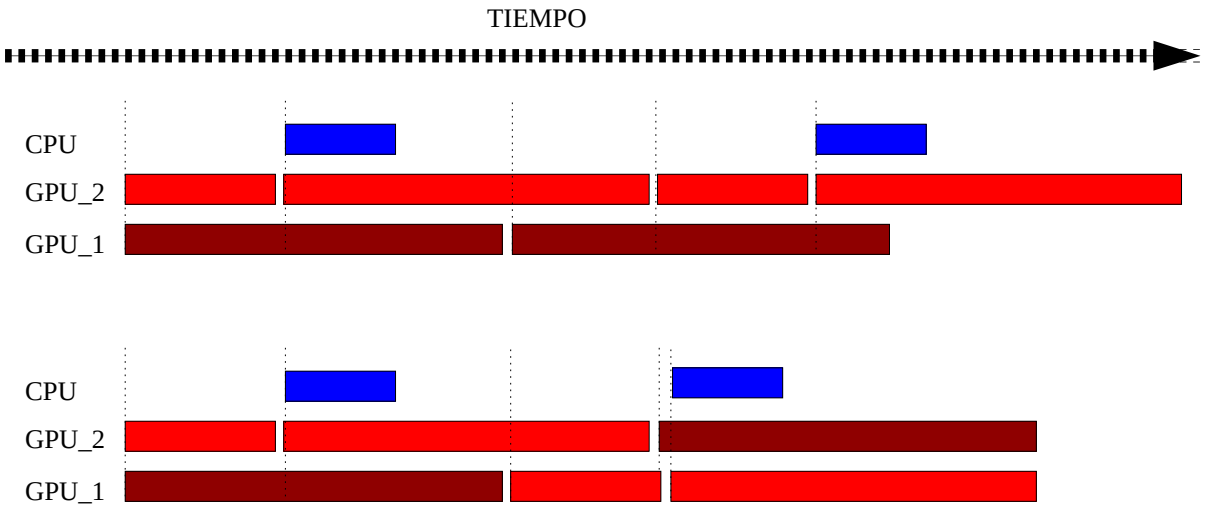

Figura 3.28: Ejemplo de planificación de tareas sobre dos GPUs y una CPU.

Implementación con combinación de operaciones $\mathrm{MGGJE_{CO}}$**.**

Una particularidad de la arquitectura de las GPUs es que, para alcanzar los mejores desempeños, deben trabajar con gran cantidad de datos.

Teniendo en cuenta lo anterior, se puede reordenar la estructura del método buscando combinar las operaciones del Algoritmo $\mathrm{MGGJE_{DC}}$ para realizar menos operaciones pero con matrices de mayor tamaño. La idea se basa en, antes de actualizar el panel de columnas, copiar el bloque de columnas actual $[A_{01};\ A_{11};\ A_{21}]$ a una estructura auxiliar, luego establecer el contenido del bloque actual a 0, y por último actualizar (multiplicar) todos los sub-paneles de forma conjunta. En la Figura 3.29 se presenta un esquema de la metodología del algoritmo; la descripción se centra en explicar la modificación introducida, por lo cual no se brindan detalles de las transferencias de datos ni del uso de la GPU en otras etapas del algoritmo.

Algorithm: $[A] := \mathrm{GJE_{CO}}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is 0×0

while $m(A_{TL}) < m(A)$ **do**
Determine block size b
Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

$\begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix}$	$:=$	$\mathrm{GJE}\left(\begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix}\right)$	Gauss-Jordan
W_1	$:=$	A_{10}	Copia
A_{10}	$:=$	0	
$\begin{bmatrix} A_{00} \\ A_{10} \\ A_{20} \end{bmatrix}$	$:=$	$\begin{bmatrix} A_{00} \\ A_{10} \\ A_{20} \end{bmatrix} + \begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix} W_1$	Producto matriz-matriz
W_2	$:=$	A_{12}	Copia
A_{12}	$:=$	0	
$\begin{bmatrix} A_{02} \\ A_{12} \\ A_{22} \end{bmatrix}$	$:=$	$\begin{bmatrix} A_{02} \\ A_{12} \\ A_{22} \end{bmatrix} + \begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix} W_2$	Producto matriz-matriz

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

Figura 3.29: Algoritmo GJE con combinación de operaciones ($\mathrm{GJE_{CO}}$) para la inversión de una matriz.

En resumen, en esta versión la actualización de los datos de un panel se realiza mediante una única invocación a la rutina GEMM de CUBLAS, en comparación con la versión anterior que implicaba al menos 3 invocaciones.

Implementación con disminución de movimientos de datos $\mathrm{MGGJE_{DM}}$**.**

El reordenamiento de operaciones presentado en el caso anterior implica un aumento en los movimientos de datos y accesos a memoria (copias a estructuras auxiliares, inicialización de valores a 0 e intercambio de valores). En las GPUs, el acceso a memoria introduce retardos importantes. Por dicha razón, a continuación se describe una estrategia para modificar el manejo de datos que permite disminuir la cantidad de accesos a memoria y de movimientos de datos mencionados. La estrategia se basa en unir la etapa de intercambio de datos debido al pivotamiento (*swap*) con la etapa de copia en la estructura auxiliar de $[A_{00};\ A_{10};\ A_{20}]$, y su posterior sobreescritura a 0. El Algoritmo `Fusión de operaciones` presenta la mecánica propuesta.

Algoritmo `Fusión de operaciones`**:**
Implementación de la fusión de operaciones de pivotamiento e inicialización

```
for ii = i  : (i + b)
    ip = IPIV(ii)
    if (ip ≠ ii)
      if ((i ≤ ip)  &  (ip < ii))
        aux(ii) = aux(ip)
        aux(ip) = A(ii)
      else
        aux(ii) = aux(ip)
        A(ip) = A(ii)
      endif
    else
      aux(ii) = A(ii)
    endif
    A(ii) = 0
```

Analizando la nueva estrategia y comparándola con la estrategia original, en la versión anterior para cada fila se producían como mínimo dos movimientos de datos (mover la fila i a la estructura auxiliar y fijar a 0 los datos de la fila), y como peor caso 5 movimientos, los dos anteriores más los tres provocados para realizar el intercambio de datos debido al pivotamiento. Al igual que la anterior, esta nueva estrategia de manejo de las filas implica, en el mejor caso, dos movimientos, pero en el peor caso solo necesita tres movimientos de datos.

3.3.2. Análisis experimental

La evaluación experimental abarcó dos aspectos. Por un lado, se evaluó el desempeño computacional de las diferentes versiones planteadas; por otro, se estudió la capacidad de escalado (tanto en lo que respecta a dimensión de los problemas tratables como en el desempeño alcanzado) de las propuestas frente al aumento de la cantidad de GPUs utilizadas.

La Figura 3.30 muestra la comparación del desempeño computacional en GFLOPS de las distintas versiones implementadas del método de GJE utilizando múltiples GPUs. La comparativa se realizó utilizando los 8 núcleos disponibles en el procesador de propósito general y las 4 GPUs de PRODAN. También se incluye, como referencia, el desempeño de la inversión de matrices alcanzado al utilizar una implementación basada en LAPACK (versión $\mathrm{LU_{CPU}}$ presentada en el Apartado 3.1.1) sobre los 8 núcleos.

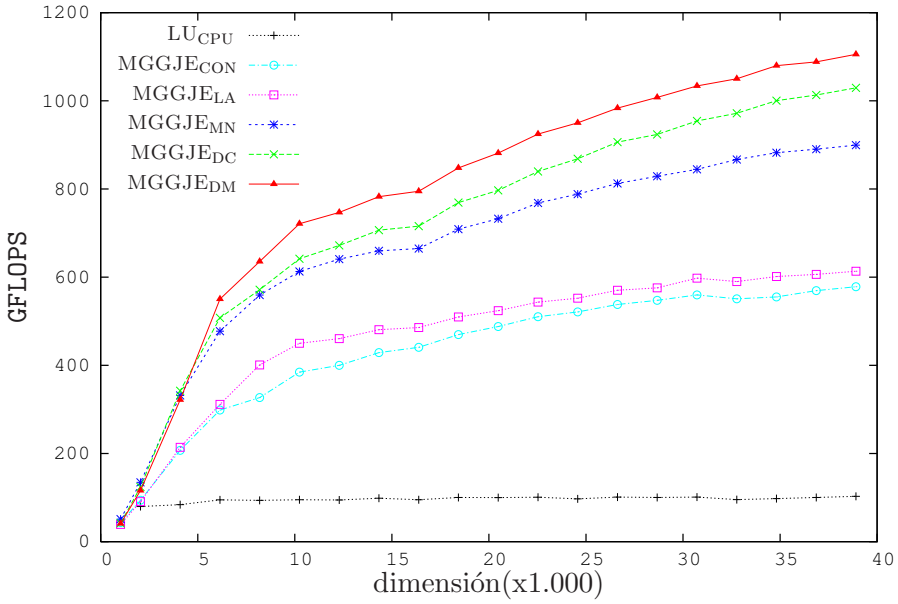

Figura 3.30: Desempeño (expresado en GFLOPS) alcanzado por las variantes de inversión de matrices sobre 4 GPUs de PRODAN.

Los nuevos códigos superan en desempeño a la implementación basada en LAPACK incluso para matrices relativamente pequeñas. La variante $\text{MGGJE}_{\text{CON}}$ ofrece un rendimiento moderado, mostrando ser más de 6 veces más rápida que la versión de LAPACK, pero se encuentra lejos del desempeño teórico ofrecido por las GPUs. La variante MGGJE_{LA} mejora la eficiencia gracias al aumento del solapamiento del cómputo de operaciones en CPU y GPUs. La estrategia multi-nivel introducida en la variante MGGJE_{MN} mejora significativamente el desempeño, alcanzando los 900 GFLOPS. El mejor balance de carga entre las distintas GPUs logrado con la distribución cíclica permite a la versión MGGJE_{DC} incrementar el desempeño en un entorno del 10 %. Finalmente, la modificación introducida por la variante MGGJE_{CO} incorpora nuevas mejoras al desempeño, reduciendo el tiempo de ejecución un 8 %. Los resultados muestran que las modificaciones introducidas por cada nueva implementación se traducen en un incremento notorio del desempeño. En particular, la versión MGGJE_{DM} presenta un desempeño que casi duplica al obtenido con la primera versión, $\text{MGGJE}_{\text{CON}}$, mejorando el desempeño ofrecido por la versión basada en LAPACK en un factor de 12×.

La Figura 3.31 presenta el desempeño en GFLOPS alcanzado por la última versión de GJE sobre múltiples GPUs, MGGJE_{DM}, pero en este caso evaluando su capacidad de escalado ante variaciones en el número de GPUs utilizadas. Se presentan las prestaciones del algoritmo utilizando 1, 2, 3 y 4 GPUs. Los resultados resumidos en dicha figura ilustran la capacidad de escalabilidad de la implementación MGGJE_{DM}, tanto en lo referido a desempeño computacional como a la dimensión de los problemas abordables. Notar que, a medida que aumenta el número de GPUs utilizadas, se dispone de mayor espacio de memoria, y en consecuencia se pueden abordar problemas con mayor dimensión. Con respecto al desempeño, se puede apreciar un aumento lineal en los GFLOPS alcanzados al aumentar la cantidad de GPUs utilizadas.

En cuanto a la capacidad de la propuesta para resolver problemas de dimensiones mayores, los experimentos mostraron gran capacidad de escalado en el tamaño de los problemas abordados;

esto se puede ver en el caso del uso de cuatro GPUs, en la que se invierten matrices de dimensión superior a 60K. Además, se validó la fórmula expresada en la Ec. (3.1), ya que se dispone de 4 GPUs con 4 GB de memoria y se presentan resultados para matrices con dimensión de hasta 62K utilizando un tamaño de bloque de hasta 512.

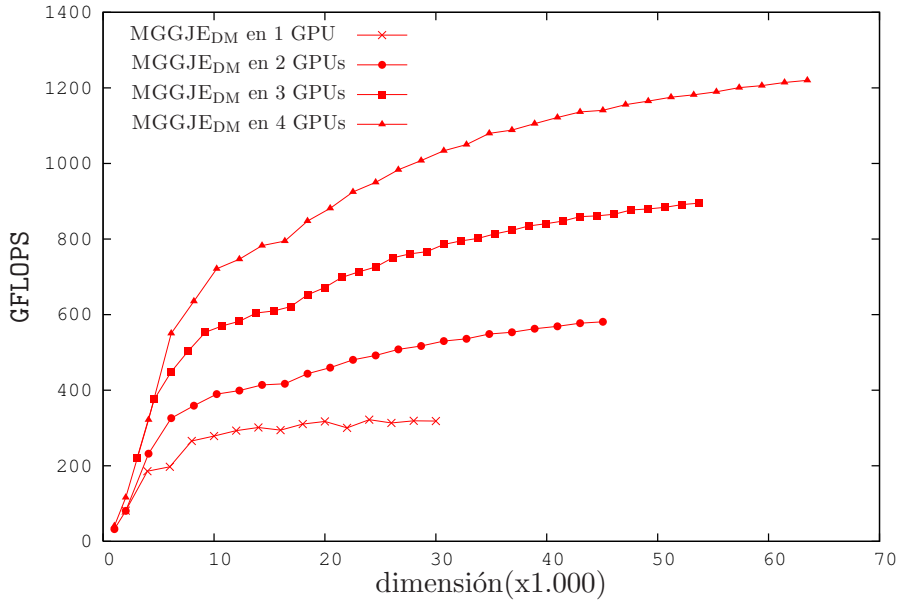

Figura 3.31: Desempeño (expresado en GFLOPS) alcanzado por la variante MGGJE$_{DM}$ para invertir de matrices sobre 1, 2, 3 y 4 GPUs de PRODAN.

En resumen, el uso de múltiples GPUs permitió abordar problemas con dimensión mayor, pero además, la versión MGGJE$_{DM}$ permitió acelerar el proceso de inversión de matrices generales en un factor de más de 12× respecto a la versión basada en LAPACK, alcanzando prestaciones superiores a los 1.200 GFLOPS (1,2 TFLOPS) al invertir matrices de dimensión 62K.

3.4. Resumen

En este capítulo se presentaron diversas estrategias para acelerar la inversión de matrices utilizando arquitecturas masivamente paralelas, en concreto GPUs. Se abordaron algoritmos para invertir tanto matrices generales como matrices SDP.

En el primer caso, se implementaron y validaron 10 rutinas para invertir matrices generales basadas en la factorización LU y el método de GJE sobre tres plataformas de hardware que incluyen configuraciones CPU-GPU con distintas capacidades de cómputo. De los resultados obtenidos se puede deducir que las variantes híbridas y concurrentes, donde se aprovechan las bondades de ambas arquitecturas, alcanzan las mejores prestaciones.

En cuanto al estudio sobre la inversión de matrices SDP, se implementaron 3 versiones basadas en la factorización de Cholesky y 6 versiones basadas en los métodos de GJE. Las implementaciones de los algoritmos basados en el método GJE presentaron muy buenos desempeños y niveles de escalabilidad, tanto en CPU como en plataformas híbridas CPU-GPU.

Por último, se incluyen algunos trabajos preliminares sobre el uso de múltiples GPUs para acelerar la inversión de matrices generales, trabajando únicamente con planificaciones estáticas. Las variantes desarrolladas, trabajando sobre 4 GPUs (del tipo Tesla C1060), alcanzaron desempeños superiores a 1 TFLOPS para invertir matrices de dimensiones superiores a 60K.

Otro aspecto destacable del estudio presentado es que se trabajó sobre 5 plataformas hardware distintas, utilizando configuraciones sumamente dispares tanto en lo que refiere a las CPUs como a las GPUs empleadas, cubriendo así un abanico importante de las capacidades ofrecidas por las GPUs de NVIDIA y en procesadores de propósito general. Además, los distintos métodos propuestos e implementados permiten acelerar la resolución del problema de reducción de modelos, tema que se trata en el siguiente capítulo.

Capítulo 4

Aceleración de la resolución del problema de reducción de modelos utilizando GPUs

En el Capítulo 2 se presentaron diferentes estrategias para abordar el problema de reducción de modelos, con especial atención a los métodos de la familia SVD. En particular, en ese capítulo se profundizó en el método de truncamiento balanceado (BT), que requiere de gran cantidad de operaciones matriciales, la mayor parte de ellas necesarias para la resolución de ecuaciones de Lyapunov. El método de la función signo es uno de los más extendidos para el tratamiento de este tipo de ecuaciones gracias, en gran parte, a su convergencia cuadrática y su sencilla implementación. La etapa fundamental en el método de la función signo, desde el punto de vista de costo computacional, es la inversión de matrices. Adicionalmente, también se presentó cómo extender y aplicar las mismas técnicas a los métodos de truncamiento estocástico balanceado (BST).

En el Capítulo 3 se presentaron distintas estrategias para acelerar la inversión de matrices explotando las cualidades de las plataformas de hardware de alto desempeño empleando procesadores de propósito general multi-núcleo conectados a GPUs.

En este capítulo se presentan propuestas para, utilizando GPUs, acelerar la resolución de las distintas variantes del problema de reducción de modelos y de las ecuaciones matriciales asociadas (Lyapunov, Lyapunov generalizada y Riccati). Para la aceleración de los métodos de reducción de modelos se cubren distintas estrategias, como el uso de núcleos computacionales optimizados para su ejecución sobre GPUs para la inversión de matrices, trabajo solapado CPU-GPU, empleo de métodos de precisión mixta, etc.

El capítulo se estructura de la forma que se presenta a continuación. Primero se describen los casos de prueba utilizados para evaluar y validar las propuestas desarrolladas. En segunda instancia se presenta una propuesta para la resolución del problema estándar de reducción del modelos mediante el método BT. Una característica destacable de esta propuesta es la inclusión de estrategias de precisión mixta, que posibilitan explotar la capacidad de cómputo de la GPU en simple precisión y, al mismo tiempo, conseguir resultados similares, en cuanto a calidad numérica, a los obtenidos al trabajar con números en doble precisión. Posteriormente, se extienden las ideas antes presentadas al caso de un sistema generalizado, es decir, un método basado en la resolución de dos ecuaciones generalizadas de Lyapunov. También se aborda la resolución de la ecuación matricial de Riccati en el contexto de técnicas de reducción de modelos con el método BST. Por último, se resumen algunas líneas de trabajo abordadas de forma preliminar relacionadas con la aceleración de la resolución del problema de reducción de modelos que incluyen transformación de matrices a bidiagonales, técnicas de *out-of-core* y el uso de núcleos para la inversión de matrices SDP sobre GPU.

4.1. Casos de prueba

En teoría de control, y particularmente en la reducción de modelos, es común evaluar las propuestas utilizando casos de prueba como los incluidas en la colección Oberwolfach de la Universidad de Freiburgo [72]. Los estudios presentados en este capítulo utilizan, principalmente, dos de los casos disponibles en dicho repositorio:

- `STEEL_I`: Este modelo aparece en la fabricación de railes de acero [211]. El objetivo es diseñar un sistema de control para el sistema de refrigeración que mantenga un gradiente de temperatura moderado en el proceso de enfriado del metal. O sea, un sistema de control para dispersar fluidos con el objetivo de enfriar el acero de manera controlada, tal que, diversas propiedades del metal (durabilidad, porosidad, etc.) se mantengan dentro de los parámetros de calidad establecidos.

  El modelo matemático corresponde a un problema de control para una ecuación de borde del calor en 2-D. Se utiliza una discretización basada en elementos finitos, con la inclusión de métodos de refinamiento adaptativo de los resultados de la malla. La Figura 4.1 muestra gráficamente el contexto del caso de prueba.

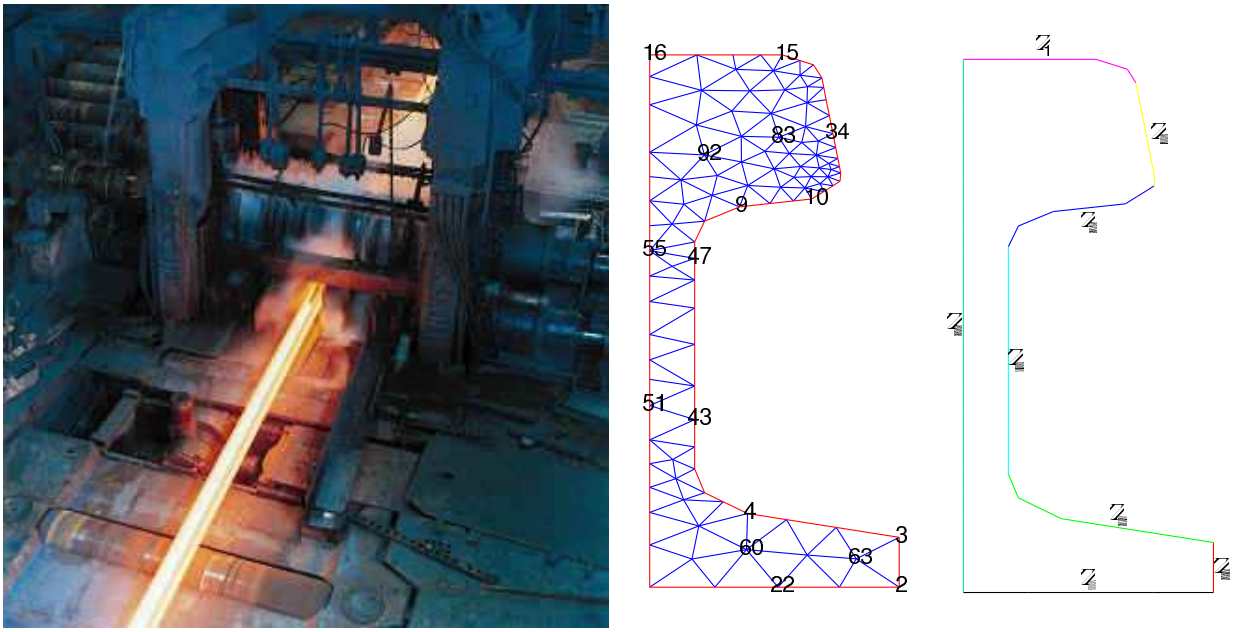

Figura 4.1: Caso práctico del problema `STEEL_I` y ejemplo de discretización de su malla asociada.

Se dispone de cuatro instancias de este caso de prueba, que se diferencian en el grado de refinamiento de la malla utilizada para discretizar el problema (y por lo tanto en la dimensión de las matrices presentes en el modelo). La Tabla 4.1 detalla las dimensiones y las características de las matrices presentes en las intancias.

Casos de prueba	n	m	p
STEEL_I$_{1357}$	1.357	7	6
STEEL_I$_{5177}$	5.177	7	6
STEEL_I$_{20209}$	20.209	7	6
STEEL_I$_{79841}$	79.841	7	6

Tabla 4.1: Dimensión de los sistemas de la familia STEEL_I.

- FLOW_METER: Este es un modelo 2-D de una estructura de tipo anemómetro consistente en un tubo y una fuente pequeña de calor. Las condiciones de borde de Dirichlet son aplicadas al sistema original. La temperatura de referencia se establece en 300 K, y las condiciones de borde de Dirichlet, así como las condiciones iniciales, son establecidas en 0 con respecto a la referencia. Las matrices del SDL de este caso presentan dimensiones n=9.669, $m = 1$ y $p = 5$.

4.2. Problema estándar de reducción de modelos

Como se expuso en el Capítulo 2, la resolución del problema estándar de reducción de modelos requiere encontrar la matriz solución a un par de ecuaciones estándar de Lyapunov duales, siendo la función signo una de las técnicas más difundidas para su tratamiento. En esta sección se describe una implementación para arquitecturas híbridas CPU-GPU de este método, que explota el poder de cómputo tanto de la CPU como de la GPU para resolver la ecuación de Lyapunov. En la estrategia propuesta, se acelera la ejecución de cada etapa utilizando el recurso de hardware más apropiado; las etapas que requieren cómputo intensivo son ejecutadas por la GPU, mientras que los restantes cálculos se efectúan en un procesador multi-núcleo de propósito general.

Además, la propuesta emplea estrategias de cómputo de precisión mixta, planteando una etapa inicial donde se calcula una aproximación de la solución de la ecuación de Lyapunov en aritmética de simple precisión para explotar los altos desempeños alcanzados por las GPUs en este tipo de aritmética, y luego se mejora la aproximación mediante un algoritmo eficiente de refinamiento iterativo. Esta estrategia permite alcanzar resultados comparables, en cuanto a calidad numérica, a los obtenidos trabajando directamente en doble precisión, a un costo computacional sustancialmente inferior.

En resumen, la propuesta se basa en resolver la ecuación de Lyapunov mediante un algoritmo similar a GECLNC (presentado en la Sección 2.3.1), utilizando tanto la CPU como una GPU para obtener una aproximación de la solución, para luego aplicar un método de refinamiento iterativo sobre la CPU. A continuación se profundiza en las diferentes etapas del algoritmo propuesto.

4.2.1. Refinamiento iterativo

La aplicación de un método de bajo costo que permite refinar una solución aproximada a una ecuación de Lyapunov presenta gran interés:

- En primer lugar, la aproximación a la solución puede ser computada utilizando aritmética de simple precisión, lo que permite explotar la gran capacidad de cómputo de las GPUs con este tipo de aritmética. En las arquitecturas GPU anteriores a Fermi, operar con aritmética de simple precisión es aproximadamente 8 veces más rápido que hacerlo en doble precisión[1]. En

[1]Antes de la arquitectura GTX200 no se disponía de aritmética en doble precisión.

la arquitectura Fermi, se reduce esta diferencia a un factor de dos. No obstante, tal diferencia en una arquitectura Fermi significa varios cientos de GFLOPS y, por lo tanto, sigue siendo interesante utilizar aritmética de simple precisión.

- En segundo lugar, una técnica de refinamiento permite reducir el número de iteraciones requeridas por los métodos iterativos para resolver ecuaciones de Lyapunov, como por ejemplo el método de la función signo. De esta forma, costosas iteraciones de estos métodos pueden ser reemplazadas por iteraciones del método de refinamiento, que requieren un costo muy inferior.

En los siguientes párrafos se describe un procedimiento para refinar aproximaciones a las soluciones factorizadas de la ecuación de Lyapunov, basado en la propuesta de Kressner et al. [143].

Idea general

Dado un solver inexacto, `GECLAR`, que devuelve una aproximación del factor de rango bajo, L, de la solución de la ecuación de Lyapunov ($AX + XA^T = -BB^T$) de forma que $X \approx LL^T$, se pretende desarrollar un método iterativo de refinamiento que, a partir de la matriz X, alcance X_n donde X_n es la solución de la ecuación con precisión doble. En este apartado, se entiende que `GECLAR` es el método `GECLNC`, ya sea ejecutando en aritmética de simple precisión y/o detenido de forma temprana. Sin embargo, vale la pena señalar que este método puede aplicarse a otros ámbitos, por ejemplo, para refinar el resultado de la función signo aplicada en aritmética de matrices jerárquica [40, 105].

Dada $L_0 = \texttt{GECLAR}(A, B)$, para mejorar la calidad de la solución aproximada, $X_0 = L_0 L_0^T$, se construye un corrector basado en el residuo

$$\mathcal{R}(L_0) = AL_0L_0^T + L_0L_0^TA^T + BB^T.$$

Este residuo es simétrico pero generalmente indefinido. En los esquemas de refinamiento estándar [123], simplemente es necesario resolver una nueva ecuación de Lyapunov con el factor del lado derecho $-\mathcal{R}(L_0)$ para obtener el término corrector. Sin embargo, esto no es posible en el marco de este trabajo ya que el solver utilizado, `GECLAR`, asume que la matriz del lado derecho es semi-definida negativa. Para salvar esta limitación, se puede descomponer el residuo en dos factores,

$$\mathcal{R}(L_0) = B_+B_+^T - B_-B_-^T. \tag{4.1}$$

La implementación computacional de esta descomposición se discute posteriormente en este apartado.

La descomposición (4.1) permite alcanzar el término corrector, X_c, resolviendo dos ecuaciones de Lyapunov,

$$AX_+ + X_+A^T = -B_+B_+^T \quad \text{y} \quad A(-X_-) + (-X_-)A^T = -B_-B_-^T,$$

con $X_c = X_+ + X_-$. Aproximando $L_+ = \texttt{GECLAR}(B_+)$ y $L_- = \texttt{GECLAR}(B_-)$, por lo tanto, la solución corregida toma la forma $X_1 = L_0L_0^T + L_+L_+^T - L_-L_-^T$. En forma similar a la Ec. (4.1), se descompone X_1 en las partes semi definidas positiva y negativa,

$$X_1 = L_1L_1^T - \widetilde{L}_-\widetilde{L}_-^T, \quad \widetilde{L}_-^TL_1 = 0. \tag{4.2}$$

La restricción de ortogonalidad $\widetilde{L}_-^TL_1 = 0$ asegura que $L_1L_1^T$ es la mejor aproximación simétrica y semidefinida positiva de X_1. Por lo tanto se puede despreciar el termino $\widetilde{L}_-$, y continuar con la iteración con L_1 como nuevo iterador. Este mecanismo se resume en el Algoritmo `GECLRF`, presentado a continuación.

Algorithm `GECLRF`:
Refinamiento iterativo para la solución factorizada de la ecuación de Lyapunov

`for` $k = 0, 1, 2, \ldots$ *until convergence*
$\mathcal{R}(L_k) \leftarrow AL_kL_k^T + L_kL_k^TA^T + BB^T$
$B_+B_+^T - B_-B_-^T =$ Factorizar $\mathcal{R}(L_k)$
$L_+ \leftarrow$ `GECLAR`(B_+)
$L_- \leftarrow$ `GECLAR`(B_-)
$X_{k+1} \leftarrow L_kL_k^T + L_+L_+^T - L_-L_-^T$
$L_{k+1}L_{k+1}^T - \widetilde{L}_-\widetilde{L}_-^T =$ Factorizar X_{k+1}

Notar que, como se muestra en el siguiente apartado, las matrices $\mathcal{R}(L_k)$ y X_{k+1} nunca son explícitamente construidas. Además, la iteración de Newton se corta tan pronto como el residuo relativo $\|\mathcal{R}(L_k)\|_F/\|L_kL_k^T\|_F$ es menor a una tolerancia definida por el usuario. Como se desarrolla en el Capítulo 12 de [123], `GECLRF` alcanza la solución X_n de precisión doble incluso partiendo de una aproximación, L_0, con poca precisión, siempre y cuando $\mathcal{R}(L_k)$ se forme utilizando doble precisión y la ecuación de Lyapunov no esté mal condicionada. Sin embargo, incluso si se busca la solución con un nivel alto de precisión, no es necesario el uso de aritmética en doble precisión para determinar la descomposición de $\mathcal{R}(L_k)$.

Descomposición en partes semi definidas positiva y negativa

En principio, la descomposición presentada en la Ec. (4.1) puede ser directamente obtenida mediante una descomposición espectral de $\mathcal{R}(L_k) = AL_kL_k^T + L_kL_k^TA^T + BB^T$. Sin embargo, desde el punto de vista computacional, ni el cómputo explícito ni la descomposición espectral de $\mathcal{R}(L_k)$ son deseables debido a su alto costo. Un enfoque más eficiente se obtiene reescribiendo el residuo como

$$\mathcal{R}(L_k) = [L_k, AL_k, B] \begin{bmatrix} 0 & I & 0 \\ I & 0 & 0 \\ 0 & 0 & I \end{bmatrix} [L_k, AL_k, B]^T = F \begin{bmatrix} 0 & I & 0 \\ I & 0 & 0 \\ 0 & 0 & I \end{bmatrix} F^T, \tag{4.3}$$

y computando la (económica) factorización QR de la matriz F, $F = UT$, donde las columnas de U son ortogonales y T es una matriz triangular superior, para después computar la descomposición espectral de la matriz (que tiene dimensiones significativamente menores)

$$T \begin{bmatrix} 0 & I & 0 \\ I & 0 & 0 \\ 0 & 0 & I \end{bmatrix} T^T = [Q_+, Q_-, Q_0] \begin{bmatrix} \Lambda_+ & 0 & 0 \\ 0 & \Lambda_- & 0 \\ 0 & 0 & \Lambda_0 \end{bmatrix} [Q_+, Q_-, Q_0]^T,$$

donde Λ_+ contiene los valores propios positivos, Λ_- los valores propios negativos, y Λ_0 los valores propios cero o despreciables de $\mathcal{R}(L_k)$. Como se explicó anteriormente, este paso se puede generar despreciando los valores propios relativamente pequeños. En los experimentos realizados, son despreciados los valores propios de magnitud menor a $\tau_{\text{eig}} = 10^{-4} \cdot \text{máx}(\|\Lambda_+\|_1, \|\Lambda_-\|_1, \|\Lambda_0\|_1)$.

De esta forma, la descomposición (4.1) se obtiene estableciendo

$$L_+ = UQ_+\sqrt{\Lambda_+} \quad y \quad L_- = UQ_-\sqrt{-\Lambda_-}.$$

Notar que $L_+^TL_- = 0$ y, por lo tanto, la descomposición de la Ec. (4.2) puede ser obtenida de forma análoga.

Resumen

La técnica de precisión mixta para la resolución de la ecuación de Lyapunov consta de los siguientes pasos: (i) resolver la ecuación de Lyapunov utilizando aritmética de baja precisión; (ii) evaluar el error cometido; (iii) utilizar la matriz de error para formular (y resolver) dos nuevas ecuaciones de Lyapunov, que tienen como término independiente una matriz formada por las columnas relativas a los valores propios positivos y negativos de la matriz error respectivamente. Notar que se puede utilizar el mismo método para resolver la nueva ecuación de Lyapunov. Esta separación es necesaria ya que el método `GECLNC` solo puede resolver ecuaciones con término independiente Hurwitz. Sin embargo, este cambio no introduce penalizaciones, ya que ambas ecuaciones pueden ser resueltas de forma unificada, para lo cual se precisa un costo ligeramente superior al de una única resolución.

En resumen, el algoritmo para resolver la ecuación de Lyapunov utilizando precisión mixta (`GECLRF`) requiere dos aplicaciones del solver aproximado de Lyapunov (utilizando la nomenclatura anterior, `GECLAR`), lo que se corresponde con una doble ejecución del Algoritmo `GECLNC`. Sin embargo, el costo computacional del Algoritmo `GECLNC` puede ser reducido significativamente si las matrices inversas utilizadas en la iteración de Newton ($A_1^{-1}, A_2^{-1}, \ldots$) ya están disponibles. Para ello, conforme se calculan, se propone almacenar las matrices $A_1^{-1}, \ldots, A_{\bar{k}}^{-1}$ para un número fijo $\bar{k}$ de iteraciones del método de Newton durante la iteración de la función signo. En los experimentos numéricos realizados, se puede observar que el número $\bar{k}$ puede ser elegido con un valor relativamente bajo, de forma que las necesidades de almacenamiento adicionales siguen estando razonablemente acotadas.

4.2.2. Implementación de la iteración de Newton híbrida CPU-GPU

El algoritmo para resolver una ecuación de Lyapunov ($AX + XA^T + BB^T = 0$), `GECLNC`, en esencia se compone de los siguientes núcleos computacionales:

- Inversión de matrices, (A_k^{-1}).
- Producto de matrices, ($A_k^{-1}B_k$).
- Factorización QR con pivotamiento de columnas (factorización RRQR).
- Otras operaciones menores relacionadas con el cómputo de los coeficientes de aceleración γ_k, evaluación de normas matriciales como $\|A_k\|_{1/\infty}$, $\|A_k^{-1}\|_{1/\infty}$, y el test de convergencia.

Seguidamente se describen las soluciones propuestas para resolver cada una de estas etapas sobre una plataforma híbrida CPU-GPU.

Inversión de matrices

Para la inversión de las matrices durante la primera etapa del Algoritmo `GECLRF`, la obtención de la solución aproximada mediante el Algoritmo `GECLNC`, se utiliza la versión $\mathrm{GJE}_{\mathrm{LA}}$ del algoritmo basado en Gauss-Jordan presentado en el Apartado 3.1.2.

Producto de matrices

Considerando el efecto combinado de las siguientes situaciones: (i) en aplicaciones prácticas de control es común que $m \ll n$, (ii) la rápida convergencia de la iteración de Newton, (iii) la posibilidad de parada, en forma aún más temprana, de la iteración de Newton debido al uso de

posterior de estrategias de refinamiento, y (iv) los beneficios de la compresión de columnas, es de esperar que el número de columnas de la matriz B_k se mantenga bajo. Además, como se describió en el Apartado 4.2.1, es aconsejable almacenar la secuencia de inversas A_k^{-1} para su posterior utilización durante el proceso de refinamiento iterativo, y como la memoria en GPU es limitada, es necesario transferir la matriz A_k^{-1} en cada iteración al espacio de memoria de la CPU para su almacenamiento. Por consiguiente, el producto matricial $A_k^{-1}B_k$ puede ser computado eficientemente en CPU sin que ello acarree ningún sobrecosto por comunicaciones; de hecho, computarlo en GPU requeriría la transferencia de la matriz B_k y aumentaría el problema de limitación de memoria de la GPU. Además, es de esperar que el tiempo de ejecución de dicho producto sea pequeño en comparación con el tiempo requerido por la etapa de inversión de la matriz. En base a lo comentado, en esta implementación se computa el producto de matrices ($A_k^{-1}B_k$) en CPU. Opcionalmente, si el tiempo de ejecución de dicho producto matricial llegara a ser significativo, en las primeras etapas de la versión propuesta se puede solapar la inversión de la matriz A_{k+1} usando un núcleo de cómputo (core) de la CPU (para la factorización del bloque actual de columnas) y el cómputo de $A_k^{-1}B_k$ usando el resto de los cores. Los resultados experimentales muestran que esto no es necesario en los casos de estudios evaluados.

Compresión mediante la factorización QR con pivotamiento de columnas

Actualmente no existen implementaciones eficientes de la factorización QR que incluyan estrategias de paralelismo de datos sobre GPUs. Por lo tanto, las opciones para computar el factor comprimido consisten en realizarlo en CPU, usando por ejemplo la rutina `GEQP3` de la biblioteca LAPACK para la factorización QR con pivotamiento de columnas, y extraer paralelismo mediante la invocación de una implementación multihilo o implementar una rutina específica para GPU. El bajo costo computacional involucrado en la resolución de una factorización aboga por la utilización de la versión implementada para CPU. Además, en este caso se utiliza una heurística ligera para estimar el rango del factor triangular resultante de la factorización QR, usando una tolerancia para el rango de $\tau_{\text{rank}} = 10 \cdot \sqrt{n} \cdot \varepsilon$.

Cuando la cantidad de columnas de la matriz B_k es pequeña, el beneficio de emplear técnicas de compresión es nulo. En consecuencia, solo tiene sentido emplear este tipo de técnicas cuando la cantidad de columnas de B es grande o la cantidad de iteraciones del método de Newton necesarias es importante (recordar que en cada paso se duplica la cantidad de columnas). El umbral donde los beneficios aportados por la reducción de columnas supera el costo del cómputo de la factorización correspondiente depende del problema abordado, por lo cual este aspecto es discutido y evaluado empíricamente en cada caso.

4.2.3. Análisis experimental

En este apartado se evalúa la precisión numérica y el desempeño computacional del método para resolver la ecuación de Lyapunov y del refinamiento iterativo propuestos. La plataforma de hardware utilizada para la evaluación es PECO, equipo presentado en el Apartado 1.3.1.

Para la evaluación experimental se utilizaron instancias de los casos de prueba de los problemas `STEEL_I` (en particular $\texttt{STEEL\_I}_{5177}$, ver Tabla 4.1) y `FLOW_METER` antes descritos.

Como los casos de prueba utilizados son instancias de problemas generalizados, se transforma el SDL generalizado $(\tilde{E}, \tilde{A}, \tilde{B}, \tilde{C})$, donde el par $\tilde{E}, \tilde{A} \in \mathbb{R}^{n \times n}$ define el haz de matrices de estados, $\tilde{B} \in \mathbb{R}^{n \times m}$ la matriz de entrada, y $\tilde{C} \in \mathbb{R}^{p \times n}$ la matriz de salida, en el SDL estándar $(I_n, \tilde{E}^{-1}\tilde{A}, \tilde{E}^{-1}\tilde{B}, \tilde{C})$, de donde se deriva la siguiente ecuación de Lyapunov:

$$\begin{aligned}(\tilde{E}^{-1}\tilde{A})X + X(\tilde{E}^{-1}\tilde{A})^T &= -(\tilde{E}^{-1}\tilde{B})(\tilde{E}^{-1}\tilde{B})^T \quad \equiv \\ AX + XA^T &= -BB^T. \end{aligned} \tag{4.4}$$

Esta transformación puede introducir problemas de estabilidad numérica; en tal caso las ecuaciones de Lyapunov generalizadas ($\tilde{A}X\tilde{E}^T + \tilde{E}X\tilde{A}^T = -\tilde{B}\tilde{B}^T$) pueden ser resueltas usando la variante de la función signo presentada en la Sección 2.4. Sin embargo, en esta primera aproximación el interés se centra en la evaluación del desempeño computacional obtenidos por los métodos paralelos sobre ecuaciones estándar de Lyapunov y, por lo tanto, se utilizó la Ec. (4.4); el método para sistemas generalizados puede ser implementado en forma análoga a la que se describe en esta propuesta y se aborda en secciones posteriores.

En un primer experimento se estudió el desempeño de la implementación híbrida CPU-GPU de la iteración de Newton para la solución factorizada de la ecuación de Lyapunov (Algoritmo `GECLNC` en la Sección 2.2 combinado con el procedimiento híbrido de GJE para la inversión de matrices). Recordar que, con el fin de explotar el alto desempeño en el procedimiento de inversión de matrices sobre GPU, todos los cómputos son realizados en aritmética de simple precisión.

La Tabla 4.2 reporta el tiempo de ejecución (en segundos) de los diferentes cómputos y transferencias de datos durante la primera etapa del algoritmo propuesto: inversión de la matriz A_k (A_k^{-1}) mediante la variante híbrida del método de GJE, GJE_{LA}, (denotado como Tiempo de A_k^{-1}), el producto matricial $A_k^{-1}B_k$ en CPU (Tiempo $A_k^{-1}B_k$), la transferencia de la matriz A_k^{-1} desde el espacio de memoria de la GPU al de la CPU (Tiempo de transfer.), el costo total de la iteración (Tiempo de iter.), y el tiempo acumulado (Tiempo acum.). La última columna en la tabla presenta el valor del criterio de convergencia de la iteración (Criterio de conv.), de acuerdo al criterio de parada especificado en la Ec. (2.31). En este experimento no se aplican técnicas de compresión de columnas a la matriz y todos los cálculos se realizan utilizando aritmética de simple precisión.

#iter. k	Tiempo A_k^{-1}	Tiempo $A_k^{-1}B_k$	Tiempo transfer.	Tiempo iter.	Tiempo acum.	Criterio de conv. $\|A_k + I_n\|_F/\sqrt{n}$
`STEEL_I`$_{5177}$						
1	1,108	0,035	0,129	1,6000	1,600	1,443e+00
2	1,091	0,027	0,127	1,5720	3,172	3,570e−01
3	1,090	0,032	0,127	1,5740	4,746	1,837e−02
4	1,090	0,045	0,127	1,5870	6,333	6,756e−05
5	1,090	0,071	0,127	1,6140	7,947	3,358e−08
`FLOW_METER`						
1	7,645	0,105	0,436	9,340	9,340	8,246e+01
2	8,077	0,105	0,437	9,772	19,112	2,119e+00
3	8,063	0,106	0,437	9,757	28,869	4,116e−01
4	8,058	0,106	0,438	9,751	38,620	2,637e−02
5	8,056	0,107	0,437	9,751	48,371	4,503e−03
6	8,056	0,127	0,437	9,770	58,141	5,490e−05
7	8,101	0,174	0,437	9,863	68,004	1,473e−09

Tabla 4.2: Tiempo de ejecución (en segundos) de la versión híbrida CPU+GPU de la iteración de Newton para la resolución de la ecuación de Lyapunov utilizando un factor de Cholesky de rango bajo sobre PECO.

De los resultados presentados en la Tabla 4.2 se puede deducir que gran parte del tiempo de la iteración es empleado en el cómputo de las matrices inversas. Por otro lado, el producto matricial ($A_k^{-1}B_k$) representa entre el 2 % y el 5 % del tiempo, y crece con las iteraciones, al igual que el número de columnas en B_k, que se duplica en cada iteración. Finalmente, el tiempo de transferencia está ligeramente por encima del 10 % del tiempo total para el problema pequeño (STEEL_I$_{5177}$, $n = 5.177$) pero por debajo del 5 % para el problema de tamaño mayor (FLOW_MODEL, $n = 9.669$), lo cual es lógico ya que el ratio cómputo/comunicación es $\frac{\mathcal{O}(n^3+n^2)}{\mathcal{O}(n^2)}$. La última columna de la tabla ilustra la convergencia cuadrática de la iteración de Newton, que se refleja en que 5 y 7 iteraciones son suficientes, respectivamente, para alcanzar la convergencia en precisión simple para los ejemplos STEEL_I$_{5177}$ y FLOW_METER. Aproximadamente 8 y 68 segundos son requeridos para obtener los factores de rango bajo de la solución de las ecuaciones estándar de Lyapunov de dimensiones 5.177 y 9.669 en aritmética de simple precisión.

La Tabla 4.3 ilustra el impacto de la técnica de compresión en la iteración de Newton. Recordar que la inversión de matrices, el tiempo de transferencia, así como el criterio de convergencia dependen exclusivamente de la dimensión de la matriz A_k y, por lo tanto, no se ven afectados por el uso de un procedimiento de compresión. La tabla presenta, para cada iteración, el tiempo requerido para computar el producto matricial (Tiempo $A_k^{-1}B_k$) y el dedicado a la compresión de la matriz resultado $B_{k+1} \leftarrow \frac{1}{\sqrt{2\gamma_k}}\left[B_k,\ A_k^{-1}B_k\right]$ mediante la factorización QR con pivotamiento de columnas (Tiempo de $\bar{R}_{k+1}$); las últimas tres columnas de la tabla contienen el número de columnas original y de los factores comprimidos ($m_{k+1} \rightarrow \bar{m}_{k+1}$), el tiempo de la iteración y el acumulado.

#iter. k	Tiempo de $A_k^{-1}B_k$	Tiempo de $\bar{R}_{k+1}$	$m_{k+1} \rightarrow \bar{m}_{k+1}$	Tiempo de iter.	Tiempo Acum.
STEEL_I$_{5177}$					
1	0,028	0,007	14 → 14	1,594	1,594
2	0,028	0,006	28 → 28	1,577	3,171
3	0,032	0,020	56 → 49	1,594	4,765
4	0,040	0,045	98 → 62	1,628	6,393
5	0,044	0,060	124 → 62	1,647	8,040
FLOW_METER					
1	0,105	0,005	2 → 2	9,349	9,349
2	0,106	0,002	4 → 4	9,776	19,125
3	0,106	0,005	8 → 8	9,764	28,889
4	0,106	0,011	16 → 14	9,763	38,652
5	0,108	0,020	28 → 20	9,774	48,426
6	0,113	0,031	40 → 20	9,789	58,215
7	0,112	0,031	40 → 20	9,790	68,005

Tabla 4.3: Tiempo de ejecución (en segundos) de la técnica de compresión sobre la implementación híbrida CPU+GPU de la iteración de Newton para la resolución de la ecuación de Lyapunov utilizando un factor de Cholesky de rango bajo sobre PECO.

De este experimento se puede concluir que la técnica de compresión genera una reducción importante en el número de columnas del factor B_k para la última iteración. Sin embargo, esta reducción en el espacio de almacenamiento no se traduce en un ahorro en el tiempo de ejecución. Por ejemplo, en la instancia FLOW_METER, la compresión en el paso 6 requiere 0,031 segundos, reduciendo el número de columnas de B_k de 40 a 20, resultando en un tiempo de ejecución de 0,112

segundos para el producto de matrices en el paso 7. Entonces, el efecto neto de usar la compresión es 0,031+0,112 = 0,143 segundos. Por otro lado, en la Tabla 4.2 se puede apreciar que computar el producto matricial en el paso 7 sin compresión requiere 0,174 segundos. Por lo tanto, incluso para el último paso de la iteración, donde la compresión genera las mayores ganancias, la diferencia en el tiempo de ejecución es ínfima. La situación descrita motiva que en los siguientes experimentos solo se aplique el procedimiento de compresión una vez, en el último paso de la iteración de Newton. Cabe resaltar que la dimensión de la matriz solución de la primera etapa (la solución aproximada en simple precisión) afecta el desempeño de la segunda etapa, el refinamiento, ya que la iteración en el refinamiento comienza justamente con la matriz solución aproximada como valor inicial, y en cada paso del refinamiento el número de columnas de la matriz se duplica.

La velocidad de convergencia y el costo computacional de cada iteración del refinamiento iterativo depende de la solución aproximada de partida, y por lo tanto de la cantidad de iteraciones de la función signo realizadas. Es por ello que, en primera instancia, es necesario encontrar el punto de equilibrio entre la cantidad de iteraciones de la función signo y del refinamiento iterativo que proporciona una solución con la precisión necesaria en el menor tiempo de cómputo.

Para encontrar el citado punto de equilibrio, la Figura 4.2 ilustra el ritmo de convergencia del procedimiento de refinamiento iterativo para los dos ejemplos abordados. En particular, las gráficas reportan el residuo relativo $r(L_k) = \|\mathcal{R}(L_k)\|_F / \|L_k L_k^T\|_F$, donde L_k es el factor computado después de k pasos de la aplicación del refinamiento iterativo a la aproximación inicial del factor L_0 computado después de $\bar{k}$ = 2,3,..,7 pasos de la iteración de Newton. Este resultado muestra que una aproximación grosera ($r(L_0) \approx 0{,}01$) es suficiente para alcanzar la convergencia con el refinamiento iterativo. La velocidad de convergencia del método depende de la calidad de la aproximación y del problema en particular, pero puede clasificarse como super-lineal. Por lo tanto, puede esperarse que un número pequeño de iteraciones de refinamiento sean suficientes para alcanzar resultados con calidad similar a doble precisión. En particular, el mejor tiempo de ejecución para el caso $\texttt{STEEL\_I}_{5177}$ se obtiene con 4 pasos de la iteración de Newton seguidas de 3 iteraciones de refinamiento. Esto implica un tiempo de ejecución de 13,089 segundos y un residuo relativo de $r(L_8)$ = 1,573e-14. En el caso `FLOW_METER`, la mejor combinación se obtiene para 6 y 4 pasos de iteración de Newton y refinamiento iterativo respectivamente, que implican un tiempo de ejecución de 71,667 segundos y alcanzar un residuo relativo de $r(L_8)$ = 3,484e-10.

Con el fin de que sirva de referencia, la Tabla 4.4 reporta el tiempo de ejecución y la evolución del criterio de convergencia de la implementación del Algoritmo `GECLNC` en doble precisión, de forma que no es necesario aplicar la técnica de refinamiento. Esto implica computar la inversión de matrices mediante el método de GJE en CPU (variante GJE_{CPU}). En el caso $\texttt{STEEL\_I}_{5177}$, la iteración de Newton converge en 6 pasos, con un tiempo de ejecución de 55,558 segundos, y $r(L_6)$ = 4,455e-15. El caso `FLOW_METER` requiere 8 iteraciones de Newton y 463,122 segundos para converger, y alcanza un residuo $r(L_8)$ = 3,422e-09. En resumen, comparando con esta versión, el algoritmo híbrido alcanza valores de aceleración de 4,24 y 6,46 para los ejemplos $\texttt{STEEL\_I}_{5177}$ y `FLOW_METER` respectivamente, y alcanza un resultado con precisión numérica similar en el residuo relativo al obtenido utilizando únicamente aritmética en doble precisión.

4.2.4. Resumen de la propuesta

En esta sección se ha presentado un algoritmo, basado en la función signo, que utiliza una estrategia de precisión mixta que permite la resolución eficiente de la ecuación estándar de Lyapunov en arquitecturas híbridas CPU-GPU. Este algoritmo permite explotar tanto las prestaciones de los procesadores de propósito general multi-núcleo (CPU), como de las GPUs actuales. En particular, el enorme paralelismo intrínseco de las GPU (en precisión simple) es explotado por una

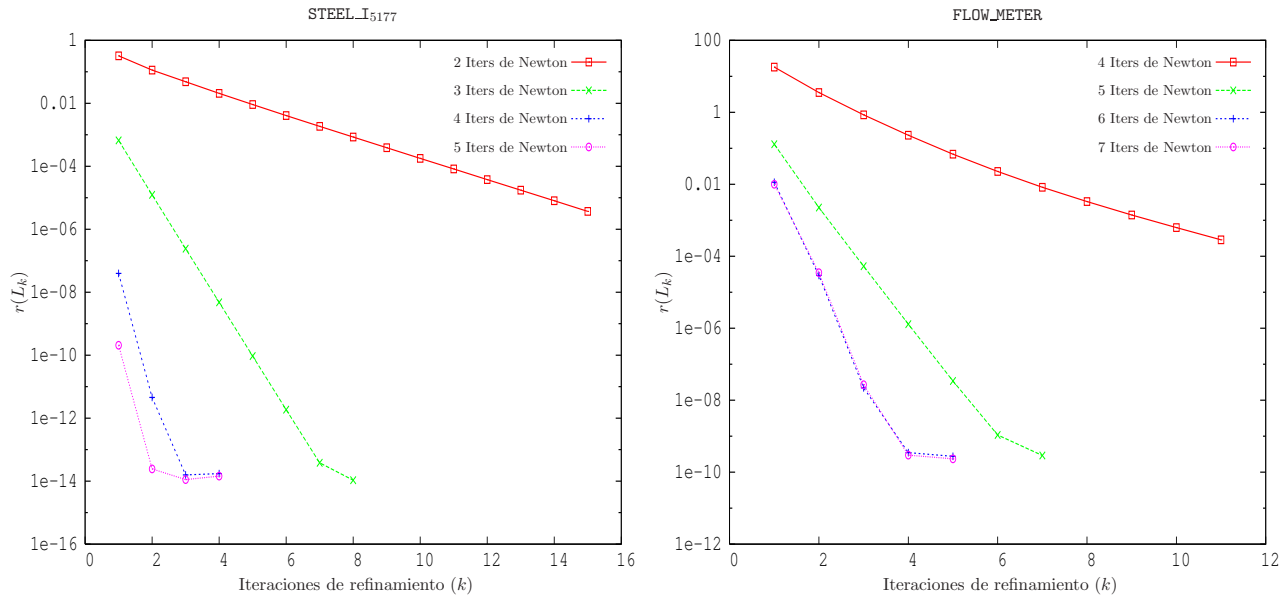

Figura 4.2: Ritmo de convergencia del procedimiento de refinamiento iterativo para los casos STEEL_I$_{5177}$ (izquierda) y FLOW_METER (derecha).

#iter. k	Tiempo A_k^{-1}	Tiempo $A_k^{-1}B_k$	Tiempo iter.	Tiempo. acum.	Criterio de Conv. $\|A_k + I_n\|_F/\sqrt{n}$
STEEL_I$_{5177}$					
1	8,612	0,127	9,299	9,299	1,443e+00
2	8,569	0,051	9,171	18,470	3,570e−01
3	8,587	0,064	9,202	27,672	1,837e−02
4	8,593	0,085	9,229	36,901	6,755e−05
5	8,582	0,136	9,269	46,170	1,382e−09
6	8,585	0,252	9,388	55,558	5,220e−19
FLOW_METER					
1	55,325	0,255	57,541	57,541	8,251e+01
2	55,799	0,512	58,236	115,787	2.124e+00
3	55,693	0,356	57,977	173,753	4,122e−01
4	55,592	0,167	57,686	231,443	2,644e−02
5	55,721	0,187	57,834	289,270	4,511e−03
6	55,689	0,215	57,831	347,115	5,510e−05
7	55,691	0,295	57,913	405,024	4,932e−09
8	55,688	0,485	58,099	463,122	1,091e−17

Tabla 4.4: Tiempo de ejecución (en segundos) de la implementación en doble precisión (en CPU) de la iteración de Newton para resolver la ecuación de Lyapunov utilizando un factor de Cholesky de rango bajo sobre PECO.

implementación híbrida de la rutina de inversión de matrices basada en el algoritmo de GJE. Esta implementación incorpora técnicas de HPC que permiten solapar eficientemente cómputos en CPU y GPU. La inversión de las matrices presentes en la iteración de Newton empleada para resolver la función signo es claramente la etapa más costosa de este esquema iterativo, y por ello se deriva a la GPU.

Una segunda contribución de la propuesta es la inclusión (e implementación) de la técnica de refinamiento iterativo, que puede ser usada para transformar la aproximación del factor de rango bajo de la solución computada con aritmética de simple precisión (u otra estrategia que proporcione una solución aproximada), para alcanzar resultados similares a trabajar directamente con doble precisión, lo que requeriría una mayor exigencia computacional. Ello posibilita explotar la diferencia entre la velocidad de cómputo con simple y doble precisión en las arquitecturas de GPU (un factor 8:1 en GPUs Tesla y alrededor de 2:1 en procesadores de propósito general).

Utilizando las técnicas descritas anteriormente, se obtuvo un solver de precisión mixta que, comparado con los códigos equivalentes para operar únicamente en CPU en doble precisión, alcanza un factor de aceleración de 4,24 y 6,46 para las ecuaciones de Lyapunov de orden 5.177 y 9.669 respectivamente. Esos resultados muestran fehacientemente que dicha plataforma de hardware es válida para abordar la resolución de ecuaciones de Lyapunov que pocos años atrás requerían el uso de clusters de memoria distribuida. Este es un resultado particularmente importante en teoría de control (reducción de modelos, control óptimo, etc.) dado que los ingenieros de control raramente utilizan plataformas de memoria distribuida para su trabajo cotidiano.

Notar que para completar la resolución del problema de reducción de modelos a partir de la solución de la ecuación de Lyapunov, únicamente es necesario computar la SVD de una matriz de dimensión reducida, y algunas multiplicaciones de matrices pequeñas. Este conjunto de operaciones no implican un costo de cómputo relevante, por lo cual pueden ser ejecutadas directamente en CPU utilizando una implementación de la especificación LAPACK, con un tiempo de cómputo moderado.

La conclusión general de esta propuesta es que el uso de las GPUs actuales es un enfoque sumamente prometedor para resolver aplicaciones científicas y de ingeniería donde la mayor parte del costo computacional se concentra en la resolución de operaciones de álgebra lineal (en especial álgebra lineal densa).

4.3. Problema de reducción de modelos generalizados

Como se presentó en el Capítulo 2, el enfoque numérico para transformar las ecuaciones de Lyapunov generalizadas (Ecs. (2.37)-(2.38)) en ecuaciones estándar, pre-multiplicando por la inversa de la matriz E, puede introducir errores numéricos. Por esta razón, en esta sección se aborda la resolución de las ecuaciones de Lyapunov generalizadas específicamente. Ello es posible utilizando una implementación de la variante de la iteración de Newton para la función signo matricial presentada en el Capítulo 2.

Algunas características específicas de la implementación aquí presentada son:

1. Se acelera la convergencia inicial utilizando un factor de escalado, γ_k, definido como $\gamma_k = \sqrt{\|A\|_\infty / \|E A_k^{-1} E\|_\infty}$.

2. Para el criterio de convergencia, se toma un umbral de tolerancia para la iteración, τ, especificado en función de la dimensión del problema y de la precisión de la máquina ε. En particular, para evitar el estancamiento de la iteración se suele especificar $\tau = n \times \sqrt{\varepsilon}$ y se realiza una o dos iteraciones después de que sea satisfecho el criterio de parada para asegurar la calidad de

la solución. Debido a la convergencia asintóticamente cuadrática de la iteración de Newton, esto suele ser suficiente para obtener una solución con la máxima precisión alcanzable.

3. Para la etapa de compresión se puede utilizar la misma estrategia propuesta anteriormente, calculando la descomposición RRQR por medio de la factorización QR tradicional con pivotamiento de columnas más un un estimador del rango confiable.

Un análisis primario del Algoritmo `CGCLNW` permite comprobar que cada iteración requiere las siguientes operaciones: la factorización LU de la matriz A_k ($\frac{2}{3}n^3$ flops), la resolución del sistema EA_k^{-1} ($2n^3$ flops), (recordar que, como se mostró en el Apartado 2.4.1, en el caso generalizado es más conveniente obtener la factorización LU de la matriz A_k y calcular la resolución de los sistemas triangulares EA_k^{-1}); la multiplicación de matrices $(EA_k^{-1})E$ ($2n^3$ flops). Además, si s_k y r_k denotan el número de columnas de $\hat{S}_k$ y $\hat{R}_k$, respectivamente, se necesita un producto matricial de dimensiones $n \times n \times s_k$ para construir $\tilde{S}_k(EA_k^{-1})^T$ ($2n^2 s_k$ flops); la resolución del sistema con r_k vectores de términos independientes para obtener $\tilde{R}_k A_k^{-1}$ ($2n^2 r_k$ flops); y un producto matricial de dimensiones $n \times n \times r_k$ para construir $(\tilde{R}_k A_k^{-1})E$ ($2n^2 r_k$ flops); finalmente, dos factorizaciones QR con pivotamiento de columnas completan la mayor parte del cómputo del algoritmo ($2n(s_k^2 + r_k^2) - \frac{2}{3}(s_k^3 + r_k^3)$ flops). En la cuenta anterior de operaciones aritméticas las matrices S_k y R_k son de rango completo, pero el costo real es menor si se computan factores de rango bajo. También son necesarias otras operaciones menores como, por ejemplo, el cálculo de normas, escalados, etc., que implican incrementos despreciables del costo computacional del algoritmo. Teniendo en cuenta estos costos, en el siguiente apartado se describe un solver eficiente sobre plataformas de HPC compuestas por procesadores multi-núcleo y GPUs.

4.3.1. Implementación híbrida del solver de la ecuación de Lyapunov generalizada

El objetivo de la implementación híbrida, que utiliza tanto la CPU como la GPU para computar, es reducir el tiempo de procesamiento ejecutando cada operación en la arquitectura que se adapta mejor, y además, cuando es posible, solapando cálculos en ambas arquitecturas. Por otro lado, estos objetivos exigen una planificación (*scheduling*) cuidadosa de las tareas para minimizar el sobrecosto (*overhead*) provocado por las comunicaciones entre los espacios de memoria de la CPU y de la GPU, buscando siempre amortizar este costo. A partir de estos conceptos, la propuesta se basa en un algoritmo híbrido que procede de la siguiente manera:

1. Al comienzo de cada iteración, se transfiere la matriz A_k de la CPU a la GPU.

2. Luego, la CPU y GPU cooperan en el cálculo de la factorización LU de la matriz A_k (ver Apartado 3.1.1).

3. La GPU transmite la matriz factorizada a la CPU (es necesario para almacenar las matrices factorizadas, que más tarde serán reutilizadas por la técnica de refinamiento iterativo).

4. La solución del sistema triangular EA_k^{-1} es computada en la GPU, mientras la CPU resuelve el sistema $\tilde{R}_k A_k^{-1}$ (notar que en el paso previo se envió la matriz con la factorización LU a CPU).

5. Posteriormente, el cálculo del producto matricial $(EA_k^{-1})E$ es ejecutado en la GPU mientras la CPU realiza algunas operaciones menores.

El resto de las operaciones son ejecutadas en CPU, dado que requieren un esfuerzo computacional menor. Aunque se estudiaron otras planificaciones, en todos los casos resultaban en mayores tiempos de ejecución.

En cuanto a los movimientos de datos, las transferencias necesarias en cada iteración en esta propuesta son:

1. Enviar la matriz A_k desde la CPU a la GPU para que la GPU (en colaboración con la CPU) compute su factorización LU.
2. Enviar los factores resultantes de la factorización LU de la matriz A_k desde la GPU a la CPU.
3. Enviar la solución de EA_k^{-1} desde la GPU a la CPU.
4. Enviar el resultado de $(EA_k^{-1})E$ desde la GPU a la CPU.

Además de las transferencias de datos antes descritas, el algoritmo incluye otras comunicaciones de menor tamaño, en particular realizadas por el núcleo híbrido que computa la factorización LU de la matriz A_k.

Las tres etapas del método para la resolución de las ecuaciones generalizadas de Lyapunov que se ejecutan en GPU son efectuadas mediante el uso de la implementación híbrida de cómputo CPU-GPU para la factorización LU presentada en el Apartado 3.1.1, la implementación de CUBLAS de la rutina GEMM para la multiplicación de matrices, y un núcleo optimizado para la resolución de sistemas lineales triangulares sobre GPU. En un principio, se utilizó la implementación de NVIDIA para la resolución de los sistemas triangulares, pero dado que este paso era el cuello de botella de la propuesta se decidió desarrollar una rutina para acelerar dicha operación; esta rutina se presenta en el siguiente apartado.

Para el resto de las operaciones, que son computadas en CPU, se invoca a las versiones multihilo de las rutinas de la biblioteca BLAS, incluyendo así estrategias de cómputo paralelo.

Una vez que los factores de Cholesky $\tilde{S}$ y $\tilde{R}$ son computados, es decir, una vez calculada la función signo, las restantes operaciones para obtener el modelo reducido comprenden un producto matricial de dimensiones moderadas ($\tilde{S}^T\tilde{R} \approx SR^T$); la descomposición SVD del resultado (ver (2.39); unos pocos productos matriz-matriz y la resolución de un sistema (ver las Ecs. (2.40)–(2.41)). Todos estos cálculos requieren un número reducido de flops y, por lo tanto, se pueden realizar en la CPU con un tiempo de cómputo moderado. En experimentos realizados en forma preliminar, estas etapas implican menos del 1 % del costo total.

Además, el método presentado en esta sección puede ser extendido utilizando estrategias de refinamiento iterativo (como se mostró para el caso estándar en la Sección 4.2) para mejorar la precisión del resultado obtenido. La metodología de trabajo es análoga a la presentada en esa sección anterior, pero la presencia de la matriz E en el SDL implica modificar levemente el método para el refinamiento iterativo. En particular, en el caso de las ecuaciones generalizadas de Lyapunov, se tienen que almacenar las matrices resultantes de la factorización LU y no las matrices inversas; además, el manejo de la descomposición expresada en la Ec. (4.3) no es válido, ya que se está resolviendo otra ecuación ($AXE^T + EXA^T + BB^T = 0$) y por lo tanto el procedimiento derivado en la sección anterior no contempla este caso, teniendo que construir la matriz F como se presenta a continuación.

$$\mathcal{R}(L_k) = [EL_k, AL_k, B] \begin{bmatrix} 0 & I & 0 \\ I & 0 & 0 \\ 0 & 0 & I \end{bmatrix} [EL_k, AL_k, B]^T =: F \begin{bmatrix} 0 & I & 0 \\ I & 0 & 0 \\ 0 & 0 & I \end{bmatrix} F^T. \tag{4.5}$$

4.3.2. Aceleración de la resolución de sistemas triangulares en GPU

Una evaluación preliminar de los tiempos de ejecución de la propuesta mostró que las tres etapas con mayor costo de cómputo eran la factorización LU (mediante la técnica híbrida), la resolución de sistemas triangulares y la multiplicación de matrices en GPU. Como se presentó en el Capítulo 1, las versiones desarrolladas de la factorización LU sobre GPU consiguen altos desempeños; además el producto de matrices está sumamente optimizado sobre arquitecturas masivamente paralelas. Por esta razón, en este apartado se estudia la aceleración de la resolución de sistemas lineales triangulares en GPU.

La función `TRSM`

BLAS establece la especificación de la función `TRSM` para resolver sistemas lineales triangulares. La función `TRSM` resuelve una de las siguientes ecuaciones,

$$op(A) \cdot X = \alpha \cdot B \quad \text{ó} \quad X \cdot op(A) = \alpha \cdot B, \tag{4.6}$$

donde α es un escalar, X y $B \in \mathbb{R}^{m \times n}$, y $A \in \mathbb{R}^{m \times m}$/ $\mathbb{R}^{n \times n}$ es una matriz triangular. La matriz A puede asumirse con unos en la diagonal o no, puede ser triangular superior o inferior, y el operador $op(A)$ puede ser A o su traspuesta.

La rutina `TRSM` de BLAS permite definir si se resuelve el sistema $op(A) \cdot X = \alpha \cdot B$ o bien $X \cdot op(A) = \alpha \cdot B$ (L o R); especificar si A es una matriz triangular superior o inferior (U o L); definir si $op(A)$ es A o bien su traspuesta (N o T). La combinación de estos tres parámetros de la función implica ocho variantes diferentes de la rutina, dependiendo de los valores utilizados en ellos: `LUN, LUT, LLN, LLT, RUN, RUT, RLN` y `RLT`.

En este trabajo se abordaron únicamente las variantes `RUN` y `RLN` de la rutina `TRSM`, ya que, con estas dos variantes, se puede resolver el sistema $XA_k = B$ utilizando la factorización LU de la matriz A_k. Sin embargo, el proceso para derivar las otras variantes de la rutina es análogo.

Como se comentó en el Capítulo 1, en las arquitecturas de HPC, los accesos a memoria representan una de las operaciones más costosas. El acceso por bloques a los elementos de la matriz en dichas arquitecturas posibilita una reutilización de los datos almacenados en la memoria caché, lo que permite reducir el número de accesos a memoria principal y proporciona un mejor aprovechamiento de la jerarquía de memoria. En la función `TRSM`, se pueden especificar cuatro estrategias de acceso a los elementos por bloques. Por ejemplo, en el caso de la variante `RLN` de la rutina, que permite resolver el sistema $XA = B$ donde $X \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{n \times n}$ es triangular inferior y $B \in \mathbb{R}^{m \times n}$, una forma equivalente es:

$$\left(\begin{array}{c|c} X_0 & X_1 \end{array} \right) \left(\begin{array}{c|c} A_{00} & 0 \\ \hline A_{10} & A_{11} \end{array} \right) = \left(\begin{array}{c|c} B_0 & B_1 \end{array} \right). \tag{4.7}$$

En base a este particionamiento del sistema, es fácil deducir que se puede resolver el sistema $X_1 A_{11} = B_1$ (un sistema también triangular), luego con la solución X_1 actualizar el bloque B_0 como $B_0 - A_{01}X_1$, y posteriormente resolver el nuevo sistema triangular, $X_0 A = (B_0 - A_{01}X_1)$, para encontrar el otro bloque de incógnitas. Estas ideas se pueden extender para dividir la matriz A en $k \times k$ bloques, quedando el algoritmo expresado como la resolución de distintos subproblemas de tamaño $b = n/k$. En la Figura 4.3 se presenta el desarrollo completo para una versión de la rutina `TRSM` a bloques utilizando la notación del proyecto FLAME [212]. En el algoritmo se asume que la solución X sobrescribe la matriz B.

En el caso de trabajar con varios bloques, el sistema se especifica de la siguiente forma:

$$\left(\begin{array}{c|c|c} X_0 & X_1 & X_2 \end{array} \right) \left(\begin{array}{c|c|c} A_{00} & 0 & 0 \\ \hline A_{10} & A_{11} & 0 \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right) = \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array} \right); \tag{4.8}$$

en este caso al resolver X_1 (asumiendo resuelto X_2) se puede actualizar B_1 ($B_1 = B_1 - B_2 A_{21}$) y luego resolver X_1 ($B_1 = X_1 A_{21}$) definiendo así la versión 1 del método, o la versión 2 donde primero se resuelve el sistema X_1 ($B_1 = X_1 A_{21}$) y luego es necesario despejar el resultado de B_0 ($B_0 = B_0 - B_1 A_{10}$). Adicionalmente, se pueden derivar dos versiones más que se basan en dividir la matriz B (y X) en dos submatrices $[B_0;\ B_1]$ y resolver primero el sistema $X_0 A = B_0$ y luego el sistema $X_1 A = B_1$ (versión 3) o en el orden contrario dando lugar a otra variante (versión 4).

Las posibilidaes para la variante `RUN` son análogas, permitiendo generar también cuatro versiones (versiones 1, 2, 3 y 4).

Algorithm: $[B] := \text{TRSM}_{\text{BLK}}(A, B)$

Partition $X \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right)$, $A \rightarrow \left(\begin{array}{c|c} A_{TL} & \\ \hline A_{BL} & A_{BR} \end{array} \right)$

where B_T has 0 rows, A_{TL} is 0×0

while $m(B_T) < m(B)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array} \right), \left(\begin{array}{c|c} A_{TL} & \\ \hline A_{BL} & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & & \\ \hline A_{10} & A_{11} & \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right)$$

where B_1 has b rows; A_{11} is $b \times b$;

$$\begin{array}{lcl} B_1 & := & A_{11}^{-1} \cdot B_1 \\ B_2 & := & B_2 - A_{21} \cdot B_1 \end{array}$$

Continue with

$$\left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array} \right), \left(\begin{array}{c|c} A_{TL} & \\ \hline A_{BL} & A_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & & \\ \hline A_{10} & A_{11} & \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right)$$

endwhile

Figura 4.3: Algoritmo a bloques (TRSM_{BLK}) para la resolución de sistemas triangulares.

En [56] y [116] se puede profundizar sobre la derivación de algoritmos de álgebra lineal a bloques y, en particular, las distintas versiones de la función `TRSM`.

Implementación

Se desarrollaron las cuatro versiones descritas para cada una de las dos operaciones, adoptando la metodología de desarrollo propuesta en el proyecto FLAME. Además, siguiendo las recomendaciones del trabajo [35], las nuevas rutinas emplean estrategias de rellenado (*padding*), aumentando las dimensiones de las matrices A y B al múltiplo de 32 (ó 64) más cercano.

La estrategia para el procesamiento de las rutinas en las diferentes arquitecturas se describe a continuación.

1. Al comienzo de los algoritmos se transfieren las matrices del espacio de memoria de la CPU al de la GPU.
2. La recursión implícita en los algoritmos se transforma en una iteración que es conducida por la CPU, mientras que las operaciones matriciales que concentran el mayor costo computacional, así como la resolución de los bloques finales (paso base de la recursión), se realizan en GPU utilizando las núcleos computacionales proporcionadas por la biblioteca CUBLAS.
3. Por último, se transfiere el resultado al espacio de memoria de la CPU.

Todas las versiones utilizan el esqueleto básico descrito, diferenciándose en la estrategia de particionado de las matrices y el orden en que se ejecutan las operaciones que se realizan durante el segundo paso.

Experimentación

La plataforma utilizada para llevar adelante los experimentos es PECO; ver Apartado 1.3.1. Para la evaluación experimental se generaron matrices de distintas dimensiones con valores aleatorios. En un estudio inicial se estableció el mejor tamaño de bloque para cada versión del algoritmo y tamaño de matriz empleado. Para ello se realizaron ejecuciones con diferentes tamaños de bloque (16, 32, 64, 128, 192, 256, 320, 384, 448 y 512). Una vez establecido el tamaño de bloque óptimo para cada versión, se comparan los resultados de las mejores versiones para cada variante con los obtenidos por la implementación de CUBLAS (rutina `cublas_strsm`). Finalmente, se compararon las distintas versiones para matrices con dimensiones que varían entre 1.024 y 8.192. En este sentido, en la Figura 4.4 (arriba) se presenta el desempeño alcanzado (en GFLOPS) por cada implementación (únicamente se muestran los resultados para el tamaño de bloque óptimo) de las variantes `RUN` y `RLN` de `TRSM` evaluadas. Para el cálculo de los GFLOPS alcanzados, únicamente se computa el tiempo de ejecución de las funciones en GPU, excluyendo el tiempo de transferencia.

En base a los resultados obtenidos, se puede deducir que la versión 1 de la variante `RUN` es claramente la que obtiene mejores resultados, dándose la misma situación para la variante `RLN`. Las versiones 3 y 4 para ambas variantes presentaron malos desempeños. Una hipótesis para explicar esta situación es que la arquitectura de las GPUs privilegia el cómputo matricial regular (matrices completas), mientras que estas versiones justamente dividen la matriz con patrón regular de datos presente en el cómputo. Sin embargo, estas versiones son, a priori, buenas opciones para extraer paralelismo de "memoria distribuida", así como para su implementación en plataformas con múltiples GPUs. Una vez identificados los tamaños de bloque óptimos y las mejores rutinas implementadas, se compararon los desempeños de las rutinas desarrolladas con el alcanzado por la implementación ofrecida por NVIDIA para GPUs incluida en la biblioteca CUBLAS. Los resultados de estos experimentos se muestran en la Figura 4.4 (abajo), donde se compara el desempeño alcanzado por las dos variantes (`RUN` y `RLN`) propuestas con el de la rutina `cublas_strsm` para diferentes tamaños de matrices. Para el cálculo de los GFLOPS se incluyen tanto los tiempos de transferencia de las matrices como el tiempo de cómputo. Los experimentos se realizaron utilizando matrices de dimensión múltiplo de 32, para evitar que los beneficios del efecto de aplicar *padding* condicionara los resultados. Al observar las gráficas, se pueden apreciar las ventajas de utilizar las estrategias de acceso a bloque para acelerar operaciones de álgebra lineal en GPUs, ya que las rutinas propuestas alcanzan desempeños claramente superiores a los conseguidos por la implementación de CUBLAS; este hecho se resume en la Tabla 4.5, donde se muestran los coeficientes de aceleración de las versiones propuestas para los distintos tamaños de matrices evaluadas al compararse con la implementación de NVIDIA. Se presentan los coeficientes de aceleración, dependiendo de si se tienen en cuenta o no, los tiempos de transferencia de las matrices.

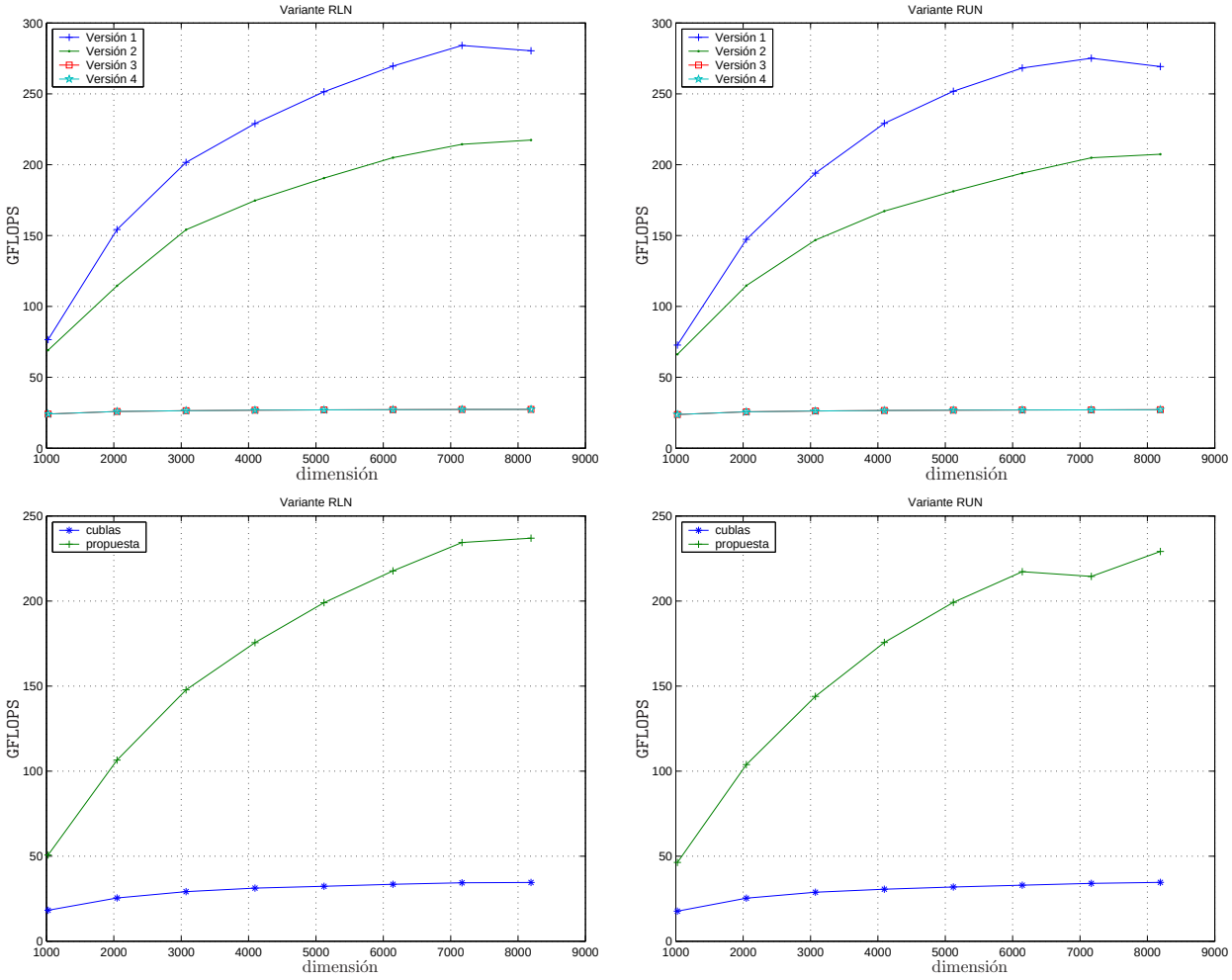

Figura 4.4: Desempeño (expresado en GFLOPS) alcanzado por las distintas versiones implementadas de las variantes `RLN` (arriba-izquierda) y `RUN` (arriba-derecha) y por la rutina `cublas_strsm` de NVIDIA y por las rutinas `TRSM` desarrolladas (abajo) sobre PECO.

Como conclusión de este estudio, se puede decir que las versiones propuestas de las rutinas `TRSM`, en las variantes `RLN` y `RUN`, son notoriamente más rápidas que la versión ofrecida por NVIDIA en CUBLAS.

4.3.3. Análisis experimental

En esta sección se evalúa la precisión numérica y el desempeño computacional del método para la reducción de modelos de SDL generalizados. La plataforma utilizada es PECO; descrita en el Apartado 1.3.1. En los siguientes experimentos, se evalúa el desempeño del método híbrido usando los casos presentados anteriormente, $\texttt{STEEL\_I}_{5177}$ y `FLOW_METER`.

La Tabla 4.6 muestra los resultados obtenidos con el algoritmo híbrido CPU-GPU para la solución de las ecuaciones generalizadas de Lyapunov acopladas para ambos sistemas. Las columnas 2, 3, 4 y 5 de la tabla muestran respectivamente el tiempo consumido (en segundos) por la factorización LU de la matriz A_k, la solución de los cuatro sistemas triangulares implicados en el

dimensión de la matriz	aceleración c/transferencia		aceleración s/transferencia	
	RLN	RUN	RLN	RUN
1.024	2,79	2,62	3,66	3,61
2.048	4,19	4,11	5,57	5,41
3.072	5,07	5,00	6,54	6,36
4.096	5,21	5,74	7,03	7,13
5.120	6,17	6,26	7,52	7,64
6.144	6,51	6,60	7,84	7,93
7.168	6,82	6,31	8,03	7,89
8.192	6,86	6,62	7,94	7,60

Tabla 4.5: Factor de aceleración de la rutina TRSM propuesta sobre PECO.

cómputo de EA_k^{-1} y $\tilde{R}_k A_k^{-1}$, el producto matricial $(EA_k^{-1})E$, y la actualización de los factores $\tilde{S}$ y $\tilde{R}$ (incluyendo el tiempo para todas las transferencias asociadas a cada una de las operaciones). El resto de las columnas muestran el tiempo global por iteración de la implementación híbrida, el tiempo por iteración para el mismo algoritmo implementado sobre CPU, y la evolución del criterio de convergencia.

#Iter k	**Tiempo** $PA_k = LU$	**Tiempo** EA_k^{-1}	**Tiempo** $(EA_k^{-1})E$	**Tiempo** $\tilde{S}_k(EA_k^{-1})$ (compresión)	**Tiempo** iteración	**Tiempo** en CPU	**Criterio de Conv.** $\frac{\|A_k+E\|_F}{\|E\|_F}$
STEEL_I$_{5177}$							
1	0,698	1,041	0,807	0,121	2,958	5,337	2,732e+02
2	0,544	1.023	0,788	0,047	2,618	5,286	2,064e+01
3	0,544	1,023	0,788	0.079	2,650	5,354	3,698e+00
4	0,544	1,023	0,788	0,159	2,732	5,465	1,140e+00
5	0,543	1,023	0,789	0,381	2,955	5,638	3,644e−01
6	0.545	1,023	0,788	0,909	3,486	6,219	7,936e−02
7	0,546	1,022	0,789	1,366	3,946	6,553	8,546e−03
8	0,543	1,023	0,788	1,866	4,442	6,909	5,706e−04
9	0,544	1,184	0.788	2,093	4,670	7,105	1,257e−05
10	0,546	1,209	0,788	2,185	4,767	7,250	7,319e−07
				TIEMPO ACUMULADO	35,224	61,156	
FLOW_METER							
1	3,380	7,7,1	5,183	0,289	17,359	31,516	6,884e+01
2	2,906	7.673	5,116	0,109	16,512	31,580	6,758e+00
3	2,918	7,673	5,116	0,137	16,553	31,725	1,585e+00
4	2,888	7,673	5,116	0,202	16,592	31,970	5,010e−01
5	3,007	7,673	5,115	0,359	16,871	32,126	1,580e−01
6	2,893	7,674	5,116	0,702	17,099	32,329	5,044e−02
7	2,886	7,673	5,116	0,971	17,365	32,525	1,241e−02
8	2,890	7,674	5,116	1,066	17,462	32,842	1,702e−03
9	2,893	7,673	5,117	1,191	17,591	32,896	1,156e−04
10	2,891	7,673	5,115	1,236	16,634	32,997	1,396e−06
11	2,891	7,673	5,116	1,248	17,994	32,881	2,389e−07
				TIEMPO ACUMULADO	188,032	355,387	

Tabla 4.6: Desempeño de la implementación híbrida de la iteración de Newton para la resolución de la ecuación generalizada de Lyapunov con factor de Cholesky de rango bajo sobre PECO.

Estudiando la Tabla 4.6 se puede observar que la mayor parte del tiempo empleado por iteración se consume en la etapa de factorización LU (columna 2), la solución de los cuatro sistemas triangulares (columna 3) y el producto matricial (columna 4). Éstas son las operaciones que, en parte o completamente, son realizadas en GPU.

El número de columnas de los factores $\tilde{S}$ y $\tilde{R}$ se duplica en cada iteración, y en consecuencia, el costo asociado a la actualización de los factores incrementa con las iteraciones. Para mantener el número de columnas de los factores acotado, una factorización RRQR es computada en cada paso. Este enfoque, como se dijo anteriormente, produce importantes ganancias cuando el número de iteraciones requeridas para la convergencia es tan grande como para incrementar notoriamente el tamaño de los factores, como es el caso de los dos problemas considerados en esta sección. El aumento en el número de columnas de $\tilde{S}$ y $\tilde{R}$ resulta en un incremento, entre otros, del tiempo requerido para realizar su actualización (columna 5).

Al comparar los resultados obtenidos con los del mismo algoritmo implementado íntegramente en CPU, se observa que el uso de la GPU proporciona una importante reducción en el tiempo de ejecución de las operaciones con alto costo computacional (la factorización LU, la solución de los sistemas triangulares y el producto matriz-matriz), ganancia que se traslada al tiempo total de ejecución por iteración. Además, mientras algunos cálculos son realizados en la GPU, otros son computados concurrentemente en la CPU. Este segundo nivel de paralelismo disminuye aún más el tiempo total de ejecución del algoritmo.

4.3.4. Resumen de la propuesta

Se presentó un algoritmo paralelo para la reducción de modelos (generalizados) de SDLs de gran dimensión sobre una plataforma de hardware híbrida CPU-GPU. El algoritmo explota las características de ambas arquitecturas, el procesador de propósito general multi-núcleo y la GPU. En particular, se obtuvo una implementación de alto desempeño de la técnica de reducción de modelos BT, donde se explotan dos niveles de paralelismo:

- De grano fino, empleando implementaciones multihilo de los núcleos de BLAS (MKL y CUBLAS) para computar las operaciones más costosas de álgebra lineal (en CPU y GPU respectivamente).

- De grano grueso, ya que las operaciones con mayor costo computacional se procesan en forma cooperativa y solapada en ambos dispositivos.

Ambos niveles de paralelismo ayudan a maximizar la utilización de los recursos, y con ello, a aumentar las prestaciones. Además, del algoritmo propuesto destacan las siguientes características por su aplicabilidad en multitud de aplicaciones científicas:

- El uso de núcleos híbridos para la factorización LU. En la implementación utilizada tanto la GPU como la CPU cooperan para calcular la factorización de la matriz A_k, proporcionando gran eficiencia.

- El nuevo código generado para la resolución de los sistemas triangulares sobre GPU. Esta nueva versión supera notoriamente la implementación de CUBLAS (que es aproximadamente entre un 30 % y 70 % más lenta para los sistemas de los casos $\texttt{STEEL\_I}_{5177}$ y `FLOW_METER`, respectivamente) y permite alcanzar una aceleración importante para una de las etapas que más tiempo consume en el procedimiento de reducción de modelos (en el caso general, cuando en el SDL $E \neq I$).

- El método propuesto permite incluir técnicas de refinamiento iterativo de bajo costo para mejorar la precisión de los resultados (en caso de trabajar con aritmética de simple precisión) empleando, por ejemplo, las técnicas mostradas en el Apartado 4.2.1 y las extensiones derivadas en el Apartado 4.3.1.

Finalmente recalcar que los resultados muestran que la reducción de modelos de sistemas lineales generalizados de gran dimensión pueden ser tratados con este tipo de plataformas de costo moderado en tiempos de cómputo razonable.

4.4. Reducción de modelos utilizando el método BST

El método BST para la reducción de modelos, descrito en el Capítulo 2, consiste en una estrategia de reducción de modelos basada en obtener el modelo de menor orden cuyo error relativo cumple con una cota prefijada. Sin embargo, el alto costo computacional del método limita su aplicación a sistemas de dimensión moderada. En esta sección se presenta una implementación de este método capaz de explotar el poder de cómputo ofrecido por plataformas de hardware que incluyen GPUs. En la propuesta se utiliza el Algoritmo `GECLNC` para resolver la ecuación de Lyapunov que determina el gramiano de controlabilidad, mientras que para resolver la ecuación de Riccati presente en el BST se utiliza el método de la función signo.

Todas las implementaciones propuestas utilizan aritmética de doble precisión. Para ello se desarrolló una implementación con aritmética de doble precisión del algoritmo `GECLAR`, presentado en la Sección 4.2. También se implementaron rutinas para la inversión de matrices basadas en el método de la eliminación de Gauss-Jordan sobre aritmética de doble precisión que explotan el poder de cómputo de la GPU. Estos núcleos para invertir matrices también se emplean para acelerar el método de resolución de la ecuación de Riccati.

Algunos criterios generales de la metodología propuesta se presentan a continuación.

- Se utiliza la expresión presentada en la Sección 4.3 como criterio de convergencia, tanto para el método de resolución de la ecuación de Lyapunov como para el método de resolución de la ecuación de Riccati.

- En la resolución de la ecuación de Lyapunov se incluyen estrategias de compresión para moderar el crecimiento de la dimensión de la matriz B. En particular, las técnicas descritas anteriormente en el capítulo basadas en la factorización RRQR y un estimador de rango ligero desde el punto de vista de costo computacional.

4.4.1. Implementación híbrida del método BST

La implementación del método BST propuesta se puede dividir en cinco etapas que se describen a continuación.

1. Solver de Lyapunov: Como primer paso del método BST es necesario resolver una ecuación de Lyapunov para calcular el gramiano de controlabilidad. El algoritmo empleado en esta etapa es `GECLNC`. La mayor parte del costo computacional de este algoritmo se centra en la inversión de matrices. Para acelerar esta operación se ha diseñado e implementado una rutina híbrida capaz de explotar la totalidad de las unidades computacionales disponibles en el hardware subyacente. Esta rutina, GJE_{MN}, emplea aritmética de doble precisión en ambas arquitecturas y fundamenta sus buenas prestaciones en una distribución inteligente de las

operaciones básicas a ejecutar entre ambas arquitecturas. Esta distribución aumenta la productividad de la CPU y la GPU, permite la computación concurrente en ambas arquitecturas y, al mismo tiempo, mantiene controlado el sobrecosto introducido por las comunicaciones de datos CPU-GPU.

2. Construcción de la matriz: La primera etapa del método para la resolución de la ecuación de Riccati

$$F^T X + XF - XGX + Q \;=\; 0, \tag{4.9}$$

mediante la función signo (ver Apartado 2.5.2) consiste en formar la matriz H tal que

$$H = \begin{bmatrix} F & G \\ -Q & -F^T \end{bmatrix}. \tag{4.10}$$

Esto requiere de diversas multiplicaciones matriz-matriz, en las que participan matrices de dimensión reducida, y que por lo tanto, implican un volumen de cálculos moderado. Por esta razón, y con el espíritu de reducir los sobrecostos debido a las comunicaciones entre la CPU y la GPU, esta etapa se realiza íntegramente en la CPU invocando núcleos computacionales de la biblioteca BLAS.

3. Cálculo de la función signo: Una vez formada la matriz H, se aplica el método de la función signo,

$$\text{fsign}\,(H) = Y = \begin{bmatrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{bmatrix}. \tag{4.11}$$

La implementación de esta etapa se basa en el método para invertir matrices utilizando la GPU comentado anteriormente y en la utilización de la interfaz OpenMP para acelerar la adición de matrices y la evaluación de normas matriciales en la CPU.

Notar que en este paso se manipulan matrices de mayor dimensión que las involucradas en el solver de la ecuación de Lyapunov, particularmente, las matrices presentan una dimensión 2 veces superior. Esto implica un notable incremento de la memoria requerida, ya que se multiplican por 4 los requerimientos de almacenamiento; y además un aumento importante de las necesidades computacionales debido al costo de orden cúbico (en la dimensión de la matriz) de los algoritmos utilizados, de forma que al tratar con matrices de dimensión dos veces superior, el número de operaciones aritméticas se multiplica por 8.

4. Resolución del sistema sobredeterminado: El siguiente paso del método es la resolución del sistema sobredeterminado

$$\begin{bmatrix} Y_{11} \\ Y_{12} + I_n \end{bmatrix} X = \begin{bmatrix} I_n - Y_{10} \\ -Y_{00} \end{bmatrix}, \tag{4.12}$$

aplicando el método de los mínimos cuadrados. Esta etapa se computa en la CPU, ya que el volumen de operaciones involucradas para resolver el sistema es moderado, y además no se dispone de implementaciones eficientes de este tipo de núcleos computacionales sobre GPUs. En particular, se utiliza una implementación multihilo de la rutina `GEQP3` incluida en LAPACK. Otras operaciones menores (normas matriciales, adición de matrices, etc.) son ejecutadas en CPU y paralelizadas mediante OpenMP.

5. Restantes etapas del método BST. Una vez computados los factores de rango bajo del gramiano de controlabilidad y el gramiano de observabilidad mediante la resolución de las ecuaciones de Lyapunov y Riccati respectivamente, para completar el cálculo del nuevo modelo únicamente restan operaciones de costo computacional moderado; ver Sección 2.5. Por esta razón, se pueden ejecutar de forma eficiente en CPU (de nuevo utilizando rutinas de BLAS) evitando introducir nuevos sobrecostos por transferencias de datos.

4.4.2. Análisis experimental

En este apartado se evalúa el desempeño computacional de la implementación del método BST sobre una plataforma de cómputo híbrida CPU-GPU. En particular, la plataforma de hardware utilizada para la evaluación es PECO-II (CPU con 8 núcleos y una GPU Fermi C2050); el equipo fue presentado en el Apartado 1.3.1. No se profundiza en el estudio de la precisión numérica de las diferentes implementaciones debido a que todas ellas realizan las mismas operaciones y en aritmética de doble precisión.

El análisis experimental se centra en la comparación de implementaciones del método BST antes descrito que se diferencian, principalmente, en la plataforma de hardware destino. A saber: una versión secuencial (denominada $\texttt{GECBSG}_{\text{CPU}}$) que se ejecuta completamente en un único núcleo de la CPU (estableciendo `OMP_NUM_THREADS=1`, esta versión es utilizada como versión de referencia); Una versión paralela multihilo ($\texttt{GECBSG}_{\text{MTCPU}}$) que explota todos los núcleos disponibles en la CPU (se establece `OMP_NUM_THREADS=8`, puesto que PECO-II dispone de 8 núcleos); una versión híbrida CPU-GPU ($\texttt{GECBSG}_{\text{GPU}}$) la cual computa operaciones de forma concurrente en la GPU y en todos los núcleos de la CPU.

Para la evaluación experimental se utilizaron dos instancias del caso de prueba `STEEL_I`, en concreto $\texttt{STEEL\_I}_{1357}$ (con dimensiones $n = 1,357$, $m = 7$ y $p = 6$) y $\texttt{STEEL\_I}_{5177}$ ($n = 5,177$, $m = 7$ y $p = 6$).

La Tabla 4.7 resume los tiempos de ejecución (en segundos) obtenidos con las tres variantes evaluadas. El tiempo de ejecución dedicado a la resolución de la ecuación de Lyapunov es mostrado en la columna 2; las columnas 3, 4 y 5 muestran el tiempo requerido para construir la matriz H, computar la función signo para la matriz H y la resolución del sistema sobredeterminado respectivamente; la columna 6 presenta el tiempo total de los diferentes métodos. Los tiempos expresados en la tabla referentes a la versión $\texttt{GECBSG}_{\text{GPU}}$ incluyen en todos los casos los costos de tranferencia de datos CPU-GPU.

De los resultados obtenidos (Tabla 4.7) se puede observar que la mayor parte del tiempo es dedicado a la resolución de la ecuación de Riccati, columnas 3 a 5, especialmente el cómputo de la función signo (fsign (H)), columna 4. El resto del tiempo, consumido por el solver aplicado a la ecuación de Lyapunov (columna 2), no posee una influencia importante en el tiempo total de ejecución.

Un estudio cuidadoso de los tiempos de ejecución de los dos solvers, demuestra que el esfuerzo computacional de los mismos está centrado en el cómputo de la matriz inversa. Esta operación es acelerada en la versión $\texttt{GECBSG}_{\text{MTCPU}}$ utilizando núcleos computacionales multihilo de la bibliotecas MKL. La variante $\texttt{GECBSG}_{\text{GPU}}$ profundiza en la optimización del procedimiento de inversión de matrices usando el método de Gauss-Jordan, más adecuado a las características de la plataforma híbrida empleada, utilizando las técnicas presentadas en el Capítulo 3.

Los tiempos de ejecución reportados para el caso $\texttt{STEEL\_I}_{1357}$ muestran los grandes beneficios alcanzados con el uso de plataformas multi-núcleo (versión $\texttt{GECBSG}_{\text{MTCPU}}$) y de plataformas híbridas (versión $\texttt{GECBSG}_{\text{GPU}}$), las cuales son, respectivamente, 5 y 6 veces más veloces que la versión secuencial. Además, los resultados obtenidos sugieren plantear una nueva versión híbrida, específica

Implementación	Resolución de ec. Lyapunov	Construcción de H	Cómputo de fsign (H)	Resolución del sist. sobredeter.	Tiempo Acumulado
$\texttt{STEEL\_I}_{1357}$					
$\texttt{GECBSG}_{\text{CPU}}$	7,74	0,10	118,15	3,23	129,22
$\texttt{GECBSG}_{\text{MTCPU}}$	1,68	0,05	22,34	0,57	24,64
$\texttt{GECBSG}_{\text{GPU}}$	9,46	0,05	10,93	0,57	21,01
$\texttt{STEEL\_I}_{5177}$					
$\texttt{GECBSG}_{\text{CPU}}$	334,16	1,52	6404,65	325,34	7065,67
$\texttt{GECBSG}_{\text{MTCPU}}$	63,75	0,86	1127,87	25,05	1217,53
$\texttt{GECBSG}_{\text{GPU}}$	26,82	0.,8	292,93	24,92	345,48

Tabla 4.7: Tiempo de ejecución (en segundos) de las distintas implementaciones del método BST para la resolución del problema de reducción de modelos sobre PECO.

para problemas de dimensión moderada, donde la resolución de la ecuación de Lyapunov se realiza completamente en CPU utilizando técnicas de computación multihilo y empleando las GPUs para acelerar la resolución de la ecuación de Riccati. El tiempo de ejecución de esta versión se puede estimar sumando el costo de la variante $\texttt{GECBSG}_{\text{MTCPU}}$ para resolver la ecuación Lyapunov y el de la variante $\texttt{GECBSG}_{\text{GPU}}$ para resolver la ecuación de Ricatti. El tiempo total de esta versión para el caso $\texttt{STEEL\_I}_{1357}$ sería de 13,23 s (1,68 + 11,55), ofreciendo un factor de aceleración con respecto a la versión de referencia superior a 9×.

De los tiempos de ejecución obtenidos para resolver las dos instancias del caso de prueba utilizado, se puede concluir que las diferencias en los tiempos de ejecución se incrementan al aumentar la dimensión de los problemas abordados. Mientras que las versiones $\texttt{GECBSG}_{\text{MTCPU}}$ y $\texttt{GECBSG}_{\text{GPU}}$ son aproximadamente 5 y 6 veces más rápidas que la versión $\texttt{GECBSG}_{\text{CPU}}$ para $\texttt{STEEL\_I}_{1357}$, en el caso $\texttt{STEEL\_I}_{5177}$ esta aceleración se incrementa a 6 para la rutina $\texttt{GECBSG}_{\text{MTCPU}}$ y 20 para la variante $\texttt{GECBSG}_{\text{GPU}}$. Esto se debe a que problemas de mayor dimensión ofrecen un paralelismo intrínseco mayor, permitiendo obtener mayores beneficios al emplear arquitecturas masivamente paralelas como las de las GPUs.

4.4.3. Resumen de la propuesta

Se presentaron dos implementaciones de alto desempeño para la resolución del método BST de reducción de modelos. Las implementaciones se diferencian en el hardware que utilizan para su ejecución. La variante $\texttt{GECBSG}_{\text{MTCPU}}$ está optimizada para su ejecución en plataformas multi-núcleo, mientras que $\texttt{GECBSG}_{\text{GPU}}$ está optimizada para su ejecución en plataformas híbridas compuestas por procesadores de propósito general multi-núcleo conectados a una GPU.

En la implementación $\texttt{GECBSG}_{\text{GPU}}$ se emplean dos niveles de paralelismo: a bajo nivel, se emplean rutinas computacionales multihilo de BLAS (en concreto de sus implementaciones MKL y CUBLAS), que son utilizadas para computar las operaciones básicas de álgebra lineal que concentran la mayor parte del costo computacional del método; un nivel superior, diferentes operaciones son computadas en forma concurrente en ambas arquitecturas, solapando así cómputos en CPU y GPU.

Los resultados obtenidos muestran que los problemas de reducción de modelos de sistemas de grandes dimensiones pueden ser tratados con esta clase de plataformas híbridas de bajo costo en tiempos de cómputo razonable.

4.5. Algunos enfoques tratados de forma preliminar

Además de las propuestas presentadas en las secciones anteriores, durante el proyecto se evaluaron de forma preliminar otras variantes que suponen, o suponían en un principio, ciertas ganancias en la utilización de los recursos de procesamiento. A continuación se describen las principales técnicas abordadas.

4.5.1. Almacenamiento externo de matrices (out-of-core)

El uso de la técnica de refinamiento iterativo, presentada en el Apartado 4.2.1, necesita el almacenamiento de las distintas matrices inversas (o de la factorización LU en el caso generalizado) calculadas durante la resolución de la función signo. Si bien en general la cantidad de memoria principal en las plataformas de cálculo actuales es elevada, el alto requerimiento espacial de este esquema podría exceder la cantidad de memoria disponible en el sistema. Buscando desarrollar estrategias para abatir dicha limitante, se implementó de forma preliminar la inclusión de una estrategia de almacenamiento externo de matrices (*out-of-core*), utilizando un disco duro o cualquier dispositivo de almacenamiento externo, en el algoritmo de reducción de modelos.

Las pruebas realizadas, mostraron que la técnica puede ser aplicada sin introducir penalizaciones importantes, empleando estrategias del tipo *prefetch* en el manejo de las matrices inversas. Con este tipo de técnicas, los datos almacenados en el disco son traídos a memoria principal antes de que sean requeridos para su cómputo, de forma que, mientras se realiza su transferencia a memoria principal, el procesador ejecuta otros cómputos. El objetivo es evitar los retrasos introducidos por las transferencias desde el disco a memoria principal.

La aplicación de técnicas de almacenamiento externo se encuentra ampliamente estudiada en la literatura [204, 205], incluso utilizando GPUs [158], e ideas similares a las evaluadas ya han sido cubiertas mostrando buenos resultados. Por estas razones, esta línea de trabajo no fue estudiada con mayor profundidad, pudiendo utilizarse, en caso de ser necesario, alguno de los resultados exitosos presentados en la literatura.

4.5.2. Transformación a bidiagonal de la matriz E

Una técnica para acelerar la resolución de las ecuaciones de Lyapunov en el caso generalizado es emplear transformaciones de matrices; en particular, se puede transformar la matriz E en bidiagonal como

$$E = UE_bV^T, \tag{4.13}$$

y utilizar las matrices de transformación ortogonales, U y V^T, para reformular las ecuaciones del SDL, como

$$\begin{array}{rcll} U^TEV\dot{x}(t) & = & U^TAVx(t) + U^TBVu(t), & t > 0, \quad x(0) = x_0, \\ y(t) & = & Cx(t) + Du(t), & t \geq 0. \end{array} \tag{4.14}$$

Utilizando las igualdades $\tilde{A} = U^TAV$ y $\tilde{B} = U^TB$ y la Ec. (4.13), el nuevo sistema a resolver queda definido como,

$$\begin{array}{rcll} E_b\dot{x}(t) & = & \tilde{A}x(t) + \tilde{B}u(t), & t > 0, \quad x(0) = x_0, \\ y(t) & = & Cx(t) + Du(t), & t \geq 0, \end{array} \tag{4.15}$$

donde, la matriz E_b es bidiagonal.

La etapa con mayor costo computacional del Algoritmo `CGCLNW` consiste en resolver el sistema $(EA_k^{-1})E$, mediante la factorización LU de A_k, la resolución de sistemas triangulares y la multiplicación de matrices. Aplicando la transformación propuesta al sistema, el producto de matrices se simplifica, resultando en una importante reducción de su costo computacional. Además, esta transformación introduce una notable reducción del costo espacial, ya que el almacenamiento de la matriz E se reduce del $\mathcal{O}(n^2)$ original al $\mathcal{O}(2n)$ necesario para almacenar una matriz cuadrada y bidiagonal de dimensión n.

Esta aproximación se implementó, pero si bien la transformación planteada permite operar con un costo asociado menor, realizar la transformación del sistema empleando las herramientas disponibles para CPU, y más concretamente implementaciones de las rutinas para la bidiagonalización de la especificación LAPACK, implica un costo computacional elevado, convirtiendo este enfoque en prohibitivo para sistemas de las dimensiones tratadas en este trabajo. Es más, algunos experimentos no formalizados mostraron que la resolución de problemas de dimensión media ($\texttt{STEEL\_I}_{5177}$) implica menos tiempo de cómputo que calcular únicamente la transformación de la matriz E asociada.

Las causas expresadas anteriormente postergaron la utilización de esta técnica hasta disponer de núcleos de computación de alto desempeño, posiblemente sobre GPU, que permitan acelerar la transformación expresada en la Ec. (4.13).

4.5.3. Uso de múltiples GPUs para reducción de modelos

En forma preliminar, se evaluó el uso de múltiples GPUs para acelerar la resolución del problema de reducción de modelos. Como se mencionó anteriormente, utilizar múltiples GPUs para la inversión de matrices, además de posibilitar aumentar las dimensiones de los problemas tratados, introduce importantes reducciones en el tiempo. En particular, los experimentos realizados sobre la plataforma PRODAN (no formalizados) utilizando las 4 GPUs disponibles y el problema `STEEL_I`, mediante los casos $\texttt{STEEL\_I}_{5177}$ y $\texttt{STEEL\_I}_{20209}$, permiten observar factores de aceleración de 1,80 y 2,55 con respecto al uso de una única GPU (ver Apartado 4.2.3), mientras que los factores trepan hasta 3,48 y 6,50 al compararlos con la implementación basada en LAPACK sobre CPU.

4.5.4. Utilización de núcleos de inversión de matrices SDP en GPU

Así como en el apartado anterior se presentó la inclusión de métodos que explotan el uso de múltiples GPUs para acelerar la inversión de las matrices durante la función signo, se pueden incluir núcleos para resolver matrices SDPs en los algoritmos de resolución de las ecuaciones de Lyapunov (donde la matriz A es SDP). Notar que, si una matriz es SDP su inversa también lo es, y además, la suma de dos matrices SDP da como resultado otra matriz SDP, por lo cual durante toda la iteración de la función signo se puede trabajar con matrices SDP.

El uso de estos métodos permite obtener doble ganancia, por un lado la aceleración de los métodos ofrece una aceleración directa en los desempeños y, por otro lado, al trabajar con métodos particulares para matrices SDP se necesita realizar la mitad de operaciones que cuando se trabaja con métodos para matrices generales redundando en otro beneficio en cuanto al tiempo de ejecución total.

4.6. Resumen

En el capítulo se han presentado diferentes estrategias para acelerar la resolución del problema de reducción de modelos, tanto para sistemas estándares como para sistemas generalizados. Se incluyen métodos tipo BT y BST utilizando plataformas de hardware que incluyen una o más GPUs conectadas a un procesador (o varios) multi-núcleo de propósito general.

Las implementaciones de los métodos permitió evaluar, entre otras técnicas, el uso de núcleos eficientes para la inversión y factorización de matrices en GPU, esquemas de trabajo concurrente en los que se solapan cálculos en la CPU y en la GPU, técnicas de precisión mixta y el desarrollo de núcleos computacionales de altas prestaciones sobre GPU para operaciones básicas de ALN, como por ejemplo para la resolución de sistemas lineales triangulares. Los resultados obtenidos con las diferentes implementaciones mostraron los beneficios de emplear arquitecturas masivamente paralelas, como las GPUs, para acelerar la resolución del problema de reducción de modelos.

Además, parte de las propuestas manejadas involucra la resolución de sistemas lineales triangulares, por lo cual se estudiaron y presentaron implementaciones que superan ampliamente el desempeño computacional alcanzado por la implementación de la biblioteca CUBLAS en GPU para dicha operación.

Capítulo 5

Conclusiones y líneas abiertas de investigación

En este capítulo se discuten las conclusiones y contribuciones más relevantes del trabajo realizado, junto con una serie de líneas abiertas de investigación. La estructura del capítulo es la siguiente: en la Sección 5.1, tras una breve revisión del problema, se presentan las conclusiones y aportes más importantes del presente trabajo; seguidamente, en la Sección 5.2, se enumeran las publicaciones derivadas del trabajo realizado; y por último, en la Sección 5.3 se presentan posibles extensiones al trabajo.

5.1. Conclusiones y aportes

Diversos fenómenos físicos se pueden modelar matemáticamente mediante ecuaciones diferenciales estableciendo SDLs que emulen el comportamiento del suceso. Esta metodología de trabajo es de gran utilidad en diversas áreas científicas, como el diseño de circuitos o el desarrollo de sistemas de control óptimo. La utilización de esta clase de modelos matemáticos requiere generalmente de un ingente número de operaciones aritméticas; a esta circunstancia se une el hecho de que, a medida que se tratan problemas de mayor complejidad, la dimensión del modelo crece y con ello las necesidades de cómputo. Es por lo tanto necesario el uso de sistemas informáticos para acelerar el proceso.

Si bien en los últimos años el desarrollo del hardware ha sido vertiginoso, el interés por resolver problemas cada vez más complejos, de mayor dimensión y con más precisión, supera con creces el poder de cálculo ofrecido por las computadoras modernas. El interés de contar con modelos matemáticos que permitan realizar simulaciones, evaluar posibles diseños o estudiar el impacto de posibles modificaciones sin necesidad de una implementación física, unido a la necesidad de que estos modelos sean tratables en un tiempo razonable, da origen al problema de la reducción de modelos.

Las técnicas de reducción de modelos buscan, dado un modelo matemático, encontrar otro cuya dimensión sea considerablemente menor pero que describa el fenómeno modelado con precisión similar a la del modelo original. De esta forma el nuevo modelo puede reemplazar al original en posteriores estudios. Los métodos de reducción de modelos más populares se pueden agrupar en dos grandes familias: los métodos basados en la SVD y los métodos de aproximación por subespacios de Krylov. Este trabajo se centra en los métodos basados en la SVD, ya que preservan importantes propiedades numéricas del sistema original, como la estabilidad y la pasividad. Además, estos métodos permiten calcular una cota del error introducido por el modelo reducido, de forma que

es posible desarrollar métodos adaptativos que reducen la dimensión del sistema en función de la precisión requerida.

El costo computacional de estos métodos es habitualmente excesivo para ser tratados en computadores de escritorio, lo que ha motivado la aplicación de técnicas y arquitecturas de HPC. El principal inconveniente que presentan estas técnicas es el alto costo económico del hardware que emplean; sin embargo, en los últimos años han surgido nuevas alternativas de hardware de bajo costo. Por un lado, las computadoras con procesadores multi-núcleo actuales integran decenas de unidades computacionales en un sólo equipo. Por otro lado, los aceleradores de hardware, como las GPUs, pueden funcionar como un hardware secundario y especializado, hacia los que el procesador principal descargue núcleos computacionales intensivos. Las principales características que diferencian a las GPUs de otras arquitecturas son su bajo costo y el alto ratio que relaciona prestaciones y consumo energético. Además, la importante evolución del poder de cómputo de las GPUs, alentada por la industria de los videojuegos, propicia una ventajosa evolución del ratio mencionado frente a la ofrecida por las plataformas de HPC tradicionales. El abaratamiento de las arquitecturas HPC abre nuevos ámbitos a su aplicación, ampliando las necesidades y desafíos de desarrollo de algoritmos y software de alto desempeño capaces de aprovechar de forma eficiente el paralelismo disponible en el hardware.

El trabajo realizado ha afrontado el análisis, desarrollo, implementación y evaluación de algoritmos paralelos capaces de identificar, extraer y aprovechar eficientemente el paralelismo disponible en los métodos para la reducción de modelos sobre plataformas que incluyen procesadores de propósito general multi-núcleo y procesadores gráficos. El estudio experimental realizado demuestra que los nuevos núcleos de ALN y las variantes de los métodos desarrollados, ejecutados en plataformas híbridas integradas por procesadores multi-núcleo y GPUs, ofrecen desempeños superiores a los alcanzados sobre arquitecturas tradicionales de HPC. A continuación se enumeran las contribuciones más importantes del presente trabajo:

- **Revisión del estado del arte en el uso de procesadores gráficos para la aceleración de núcleos de ALN.** Los métodos para la reducción de modelos basados en la SVD implican la ejecución de operaciones básicas de ALN; en particular, el costo computacional de estos métodos se centra en operaciones como la inversión de matrices, la factorización LU, la resolución de sistemas triangulares y la multiplicación de matrices. En los últimos años se han presentado diversos trabajos sobre el uso de GPUs para acelerar estos núcleos básicos. Conocer las herramientas de software disponibles, así como los enfoques y técnicas de programación en el área, es imprescindible para poder hacer una definición y evaluación cabal de los objetivos y el trabajo a desarrollar en la tesis. Es importante destacar que, durante la revisión realizada, no se encontraron trabajos que abordaran específicamente la inversión de matrices generales.

- **Diseño e implementación de ocho nuevas rutinas para la inversión de matrices generales sobre plataformas que incluyen GPUs.** Ante la ausencia de implementaciones para la inversión de matrices generales sobre plataformas con GPUs, y dada la importancia de esta operación para los métodos de resolución del problema de la reducción de modelos, una parte relevante del trabajo se centró en el estudio y optimización esta operación matricial.

  Las implementaciones desarrolladas se pueden dividir, según la plataforma destino, en dos categorías:

  - **Implementaciones sobre una GPU:** Se han desarrollado dos implementaciones para su ejecución íntegra en esta arquitectura, una de ellas basada en la factorización LU

($\mathrm{LU_{GPU}}$) y otra en el algoritmo de Gauss-Jordan ($\mathrm{GJE_{GPU}}$). En ambas se explotan estrategias de *padding* y el uso de núcleos computacionales incluidos en la biblioteca CUBLAS.

- **Implementaciones sobre una plataforma híbrida CPU-GPU:** Se han desarrollado seis implementaciones para esta plataforma. Todas ellas se fundamentan en el reparto de los cómputos entre ambas arquitecturas según sus características, y en la contención del sobrecosto introducido por las comunicaciones CPU-GPU. Dos de las implementaciones se basan en el uso de la factorización LU de la matriz ($\mathrm{LU_{HIB}}$ y $\mathrm{LU_{HIB+}}$). Ambas implementaciones emplean estrategias tipo *padding* para acelerar el acceso a la memoria de la GPU. $\mathrm{LU_{HIB}}$ está basada en el uso de los núcleos computacionales incluidos en CUBLAS, mientras que $\mathrm{LU_{HIB+}}$ se basa en núcleos desarrollados en otros proyectos que han mostrado ser el estado del arte. Otras cuatro variantes se basan en el algoritmo de Gauss-Jordan. La primera implementación, $\mathrm{GJE_{HIB}}$, realiza una distribución inteligente de las operaciones de ALN a ejecutar entre ambas arquitecturas, de forma que cada operación se computa sobre el dispositivo que mejor se adapta a su resolución; además, incluye técnicas básicas de computación de altas prestaciones en GPUs como el *padding*. Partiendo de esta versión se desarrolló la rutina $\mathrm{GJE_{CON}}$, que incluye el cálculo concurrente en ambas arquitecturas y, posteriormente, $\mathrm{GJE_{LA}}$, que añade técnicas de *look-ahead* para reducir el impacto negativo del rendimiento del camino crítico. Finalmente, en la versión $\mathrm{GJE_{MN}}$ se agrega el uso de dos tamaños de bloque, posibilitando que el algoritmo se adapte simultáneamente a las particularidades de ambas arquitecturas. Los resultados obtenidos muestran las buenas prestaciones que se pueden alcanzar en la inversión de matrices generales sobre una arquitectura híbrida CPU-GPU. Los diferentes experimentos realizados demuestran que la rutina más eficiente en la inversión de matrices pequeñas, típicamente matrices de dimensión menor a 2.000, es la basada en la factorización LU empleando únicamente la CPU (rutina $\mathrm{LU_{CPU}}$, basada en los núcleos computacionales de LAPACK); sin embargo, para matrices de dimensión mayor, la versión $\mathrm{GJE_{MN}}$ es la que ofrece mejores desempeños. En particular, sobre la plataforma PECO, esta variante ofrece un factor de aceleración de 3× con respecto a $\mathrm{LU_{CPU}}$ para invertir matrices de dimensión 14.000.

- **Implementación de dos variantes del método de GJE para la inversión de matrices SDP sobre procesadores multi-núcleo de propósito general.** Los métodos específicos para la inversión de matrices SDP, frente al uso de métodos para matrices generales, permiten reducir el costo computacional de la operación a la mitad. Sin embargo, el método tradicional para invertir matrices SDP, basado en la factorización de Cholesky, muestra niveles de desempeño inferior a los métodos para matrices generales sobre arquitecturas tradicionales de HPC. Teniendo en cuenta lo anterior se implementaron y evaluaron las dos variantes del método de GJE para matrices SDP propuestas por Bientinesi et al. [57] sobre plataformas multi-núcleo. Los desempeños de las implementaciones desarrolladas mostraron superar notoriamente la implementación basada en la factorización de Cholesky; además las variantes basadas en el método de GJE presentan una mejor escalabilidad ante el aumento de las unidades de cómputo.

- **Diseño e implementación de cinco rutinas para la inversión de matrices SDP sobre plataformas que incluyen GPUs.** En la actualidad no existen implementaciones de métodos para invertir matrices SDP sobre arquitecturas que incluyan GPUs. Con el fin de acelerar la reducción de modelos en los que la matriz de estados es SDP, se desarrollaron

diversos núcleos computacionales de alto desempeño para esta operación. Al igual que en el caso de la inversión de matrices generales, las variantes implementadas pueden ser agrupadas según su arquitectura destino en dos grupos:

- **Implementaciones sobre una GPU:** Se han implementado tres versiones sobre esta arquitectura, una de ellas basada en el algoritmo tradicional que emplea la factorización de Cholesky de la matriz (CHOL_{GPU}) y las otras dos basadas sendas variantes del algoritmo de GJE para la inversión de matrices SDP propuestas por Bientinesi et al., $\text{GJES}_{\text{GPU\_V1}}$ y $\text{GJES}_{\text{GPU\_V2}}$. Este segundo algoritmo demostró ser más apropiado para la arquitectura masivamente paralela de la GPU.
- **Implementaciones sobre una plataforma híbrida CPU-GPU:** Se han desarrollado tres rutinas para esta plataforma; todas ellas logran un alto desempeño ejecutando cada núcleo computacional sobre la arquitectura más conveniente, al mismo tiempo que limitan el sobrecosto introducido por las comunicaciones CPU-GPU. La primera rutina desarrollada, CHOL_{HIB}, está basada en la factorización de Cholesky, mientras que las versiones GJES_{HIB} y GJES_{CON} implementan la segunda variante del algoritmo GJE, que ha demostrado experimentalmente ser la más apropiada sobre plataformas paralelas. GJES_{HIB} incluye el uso de *padding* para acelerar el acceso a la memoria de la GPU; GJES_{CON} además incluye el cálculo concurrente en CPU y GPU.

- **Diseño e implementación de algoritmos para invertir matrices generales utilizando múltiples GPUs.** El uso de múltiples GPUs requiere un detallado análisis de la estrategia de particionado de datos y operaciones, del balanceo de la carga y de la planificación de las comunicaciones. La plataforma utilizada en este estudio consiste en varias GPUs conectadas a un *host*. Se han desarrollado seis versiones diferentes, en las que varían, además de las técnicas de optimización empleadas, la forma en que datos y operaciones son distribuidas entre las distintas GPUs. La implementación con mejor desempeño superó el TFLOPS (10^{12} flops por segundo) en una plataforma con 4 GPUs para invertir matrices de dimensión 60.000. Al utilizar de forma eficiente las memorias de las distintas GPUs empleadas, las versiones desarrolladas permiten además, abordar la inversión de matrices de mayor dimensión.

- **Diseño e implementación de una variante del Algoritmo `GECLNC` para la reducción de modelos definidos por SDL estándar que explota el uso de GPUs.** La propuesta deriva las etapas más costosas del método para su ejecución en la GPU, empleando implementaciones eficientes de núcleos de ALN de la biblioteca CUBLAS, así como otros desarrollados en el marco de esta tesis; entre ellos destacan las rutinas para la inversión de matrices comentadas anteriormente. Además, se muestra cómo utilizar estrategias de precisión mixta en las que se efectúa el grueso de las operaciones necesarias para la resolución de las ecuaciones de Lyapunov, que concentran la mayor parte del costo computacional del método, usando aritmética de simple precisión, para después aplicar una técnica de refinamiento de la solución que aumenta su precisión hasta alcanzar una solución numéricamente equiparable a la que se obtendría al trabajar directamente con aritmética de doble precisión, pero de una forma computacionalmente más económica.

- **Diseño e implementación de una variante del Algoritmo `CGCLNW` para la reducción de modelos definidos por SDL generalizados que explota el uso de GPUs.** Esta propuesta está basada en dos niveles de paralelismo. En un primer nivel se utilizan implementaciones paralelas (paralelismo a nivel de hilos) para la ejecución de cada operación tanto en la CPU como en la GPU. En el segundo nivel se realizan cómputos concurrentemente en

ambas arquitecturas. Adicionalmente, se describe cómo extender el método de precisión mixta implementado para la rutina `GECLNC` al caso de SDLs generalizados, y con ello los pasos necesarios para su integración en el Algoritmo `CGCLNW`. Otro aporte enmarcado en esta propuesta es la implementación de dos rutinas para la resolución de sistemas lineales triangulares, rutina `TRSM` de BLAS, sobre GPUs. El desempeño de estas rutinas superan ampliamente el ofrecido por la rutina análoga de la biblioteca CUBLAS.

- **Diseño e implementación de una variante del algoritmo para resolver el problema de reducción de modelos mediante el método BST que explota el uso de GPUs.** La rutina propuesta utiliza aritmética en doble precisión, y se basa en el Algoritmo `GECLNC` y en el método de la función signo para resolver ecuaciones de Riccati. Los resultados experimentales han demostrado que problemas de gran dimensión pueden ser tratados en arquitecturas híbridas CPU-GPU con tiempos de respuesta razonables.

Cabe destacar que los experimentos realizados en este trabajo han incluido el uso de varias plataformas de hardware con características dispares, tanto en la capacidad de cómputo de las CPUs como de las GPUs. Esta circunstancia ha permitido comprobar la validez de las técnicas y códigos propuestos sobre un amplio espectro de plataformas.

Para concluir, el presente trabajo ha demostrado que las plataformas híbridas CPU-GPU son sumamente idóneas para computar operaciones de ALN, y por ende, para la reducción de modelos. Estas plataformas permiten, pese a su moderado costo económico, resolver con un tiempo de cómputo razonable problemas que hasta hace pocos años requerían del uso de costosos clusters de computadoras. Esta situación, aunada al vertiginoso avance de los procesadores multi-núcleo y, sobre todo, de los procesadores gráficos, permite conjeturar que, en un futuro no muy lejano, problemas de gran dimensión (con matrices de hasta centenares de miles de filas) podrán ser resueltos en ordenadores con un costo moderado.

5.2. Difusión de los resultados del trabajo

Algunos resultados obtenidos durante el desarrollo de la tesis aparecen publicados en diferentes foros de divulgación científica. A continuación se listan estas publicaciones agrupadas en revistas, actas de congresos internacionales y actas de congresos regionales.

Revistas

- "Using graphics processors to accelerate the computation of the matrix inverse". P. Ezzatti, E. S. Quintana, A. Remón. J. of Supercomputing, 2011 (aceptado y pdte. de publicación). ISSN: 0920-8542. D.O.I.:10.1007/s11227.011.0606.4

- "A mixed-precision algorithm for the solution of Lyapunov equations on hybrid CPU-GPU platforms". P. Benner, P. Ezzatti, D. Kressner, E. S. Quintana, A. Remón. Parallel Computing, 2010 (aceptado y pdte. de publicación).
ISSN: 0167-8191. D.O.I.:10.1016/j.parco.2010.12.002

Actas de conferencias internacionales

- "High performance matix inversion of SPD matrices on graphics processors". P. Benner, P. Ezzatti, E. S. Quintana, A. Remón. Workshop on Exploitation of Hardware Accelerators – WEHA 2011, (aceptado y pdte. de publicación). Estambul (Turquía). 2011. ISBN: –.

- "Efficient model order reduction of large-scale systems on multi-core processors". P. Ezzatti, E. S. Quintana, A. Remón. 11th Int. Conf. on Computational Science and Its Applications – ICCSA 2011, (aceptado y pdte. de publicación en Lecture Notes in Computer Science, vol. 6786). Santander (España). 2011.

- "High performance matrix inversion on a multi-core platform with several GPUs". P. Ezzatti, E. S. Quintana, A. Remón. 19th Euromicro Conference on Parallel, Distributed and Network based Processing – PDP 2011, pp. 87-93. Ayia Napa (Chipre). 2011. ISBN: 1066-6192.

- "Accelerating model reduction of large linear systems with graphics processors". P. Benner, P. Ezzatti, D. Kressner, E. S. Quintana, A. Remón. State of the Art in Scientific and Parallel Computing – PARA 2010, (aceptado y pdte. de publicación). Reykjavik (Iceland). 2010.

- "Using hybrid CPU-GPU platforms to accelerate the computation of the matrix sign function". P. Benner, P. Ezzatti, E. S. Quintana, A. Remón. Lecture Notes in Computer Science 6043, 7th Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks – HeteroPar'09, pp. 132-139, (Eds. H.X. Lin, M. Alexander, M. Forsell, A. Knüpfer, R. Prodan, L. Sousa,A. Streit). Delft (Holanda). 2009. ISBN: 3-642-14121-8.

Actas de conferencias regionales

- "Inversión de matrices en arquitecturas CPU-GPU". P. Ezzatti, E. S. Quintana, A. Remón. XXI Jornadas de Paralelismo, pp. 503-510. Valencia (España). 2010. ISBN: 978-84-92812-49-3.

- "Resolución de sistemas traingulares en tarjetas gráficas (GPU)". P. Ezzatti, E. S. Quintana, A. Remón. XXI Iberian Latin American Congress on Computational Methods in Engineering - CILAMCE 2010. Buenos Aires (Argentina). 2010.

- "Improving the performance of matrix inversion with a tesla GPU". P. Ezzatti, E. S. Quintana, A. Remón. 3rd High-Performance Computing Symposium – HPCLatAm 2010. Buenos Aires (Argentina). 2010.

- "Uso de GPUs para acelerar el cálculo de la matriz inversa". P. Ezzatti, E. S. Quintana, A. Remón. XXXVI Conferencia Latinoamericana en Informática – CLEI 2010. Asunción (Paraguay). 2010.

5.3. Líneas abiertas de investigación

El trabajo realizado ha cubierto el objetivo original de la tesis, la aceleración de métodos de reducción de modelos utilizando procesadores gráficos, pero además ha servido para identificar diversos problemas y extensiones sobre los que merecería la pena profundizar:

- Aunque la aceleración de la inversión de matrices generales y de matrices simétricas y definidas positivas utilizando GPUs ha sido estudiada y diversas implementaciones de altas prestaciones desarrolladas, existe otro conjunto de matrices con propiedades particulares de interés en el campo de la reducción de modelos, se trata de las matrices simétricas indefinidas. Además, un análisis preliminar mostró que las rutinas involucradas en el método de inversión tradicional para las matrices simétricas indefinidas ofrecidas en LAPACK no presentan buenos desempeños.

- Si bien ya se estudió en forma prelimar el uso de múltiples GPUs para acelerar la inversión de matrices generales, diversas líneas de trabajo quedaron pendientes. Entre otras destaca el uso de planificaciones dinámicas de tareas, el diseño de métodos que se beneficien del uso de un mayor número de GPUs, así como emplear otras configuraciones de hardware con mayor capacidad de cómputo y de almacenamiento, como por ejemplo los clusters de computadores en los que cada computador incluya más de una GPU.

- Debido a las características que presentaban las GPUs al inicio del trabajo, éste se centra en el uso de aritmética de punto flotante de simple precisión, y en el uso de estrategias de precisión mixta. Dada la evolución que han sufrido las GPUs durante la ejecución de la tesis, se incluyó también una evaluación del uso de aritmética de doble precisión sobre la arquitectura Fermi. Parece interesante profundizar en ambas líneas, tanto en el empleo de aritmética de doble precisión sobre GPUs, como en el uso de estrategias de precisión mixta donde la etapa de simple precisión y la de doble se procesan en GPU, explotando las importantes mejoras en el desempeño sobre aritmética en doble precisión ofrecidas por las GPUs con arquitectura Fermi.

- Los diferentes métodos y códigos desarrollados fueron implementados con un carácter de prototipo con el fin de estudiar las capacidades de las nuevas arquitecturas de HPC sobre un problema conocido. En base a los buenos resultados obtenidos, y al volumen de las herramientas desarrolladas, parece interesante formalizar los desarrollos para ofrecer a la comunidad una biblioteca de métodos para la reducción de modelos acelerados mediante la utilización de procesadores gráficos.

- También resulta interesante estudiar el uso de GPUs para la aceleración de otros métodos de reducción de modelos. Presentan particular interés aquellos métodos que permiten explotar la estructura dispersa de las matrices que definen el modelo original, reduciendo notablemente el volumen de datos y cómputos necesarios en el proceso de reducción de modelos. En este ámbito, cabe destacar los métodos LR-ADI.

Anexo A

Las GPUs como procesadores de propósito general

En el Capítulo 1 se presentó el potencial que ofrecen las GPUs actuales para computar operaciones de propósito general con un alto grado de paralelismo de datos. Este potencial actual se sustenta en diversos avances en el hardware subyacente (las tarjetas gráficas), en las herramientas software disponibles y en las técnicas para su utilización.

Las cualidades de la arquitectura de las GPUs actuales pretenden, principalmente, acelerar el cómputo de gráficos, objetivo para el cual fueron diseñadas. Para facilitar la comprensión de la arquitectura de las GPUs, en este anexo se ofrece una breve descripción del proceso de cómputo de imágenes, el *pipeline* gráfico (consultar [141, 160] para más detalles). Otro aspecto con gran relevancia en el uso de las GPUs para la resolución de problemas de propósito general es la evolución que ha sufrido el hardware y su impacto en la facilidad y flexibilidad de programación del mismo. Este aspecto también se aborda en el presente anexo, con particular detalle en las arquitecturas de las tarjetas utilizadas en el proyecto, los procesadores de NVIDIA con arquitectura unificada (compatibles con CUDA). La información resumida en este anexo se basa, entre otros, en [141, 160, 180, 179, 173, 174, 175, 176].

La estructura del anexo se describe a continuación. En la siguiente sección se describe el *pipeline* gráfico. Posteriormente, se ofrece un resumen de la evolución histórica de las GPUs. Por último, se describen las principales características de las arquitecturas de las GPUs de NVIDIA utilizadas durante el proyecto, G80, GTX200 y Fermi. También se describe el concepto *compute capability*, directamente relacionado con la evolución de las tarjetas de NVIDIA posteriores a la arquitectura unificada.

A.1. Pipeline gráfico

Las diferentes operaciones presentes en las etapas para visualizar imágenes en pantalla se pueden agrupar en un proceso generalmente conocido como cauce o *pipeline* gráfico. Conviene destacar que, en una pantalla bidimensional, frecuentemente se necesita desplegar imágenes que son proyecciones de escenas tridimensionales. Esto introduce ciertas dificultades que son resueltas mediante costosas operaciones en el *pipeline* gráfico. Con el fin de aliviar a la CPU de la ejecución del *pipeline* gráfico, se propucieron en la década de los 80s las primeras tarjetas gráficas que, en un principio, eran dispositivos diseñados particularmente para el cómputo del *pipeline*.

Las estapas del *pipeline* gráfico clásico se resumen en la Figura A.1. La primera etapa del *pipeline* se encarga de recibir las imágenes de la CPU en algún formato manejable, en particular

una colección de primitivas que generalmente son triángulos (los triángulos, al ser los polígonos más simples que se pueden construir, posibilitan un ágil manejo).

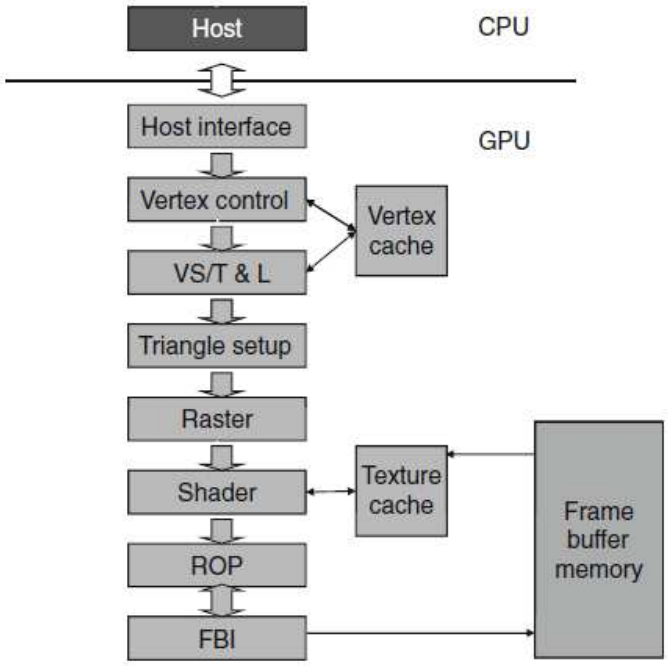

Figura A.1: Diseño simplificado del *pipeline* gráfico. Extraído de [141].

Posteriormente, en el *Vertex control*, cada triángulo se transforma de acuerdo a su posición y orientación respecto al punto de vista deseado. Así, cada uno de los vértices del triángulo se transforma a otra posición, según las características del espacio de vista (eliminando así los vértices no visibles).

A continuación, en la etapa de *VS/T&L* (sombreado de vértices, transformaciones e iluminación) se transforman los vértices y se asignan propiedades a los vértices visibles (colores, normales, tangentes, texturas). En la GPU, el sombreado es realizado por los *vertex shaders*[1].

La siguiente etapa (*Triangle setup*) se concentra en realizar cálculos para las aristas, interpolando colores y otros valores de los vértices.

La etapa *Raster* determina para cada triángulo qué píxeles lo integran. Por cada píxel, en esta etapa se interpola, a partir de los vértices, los valores necesarios para sombrear los píxeles, evaluar la inclusión de colores y determinar la posición de la textura que será pintada en el píxel, entre otros.

En la etapa *Shader* (de sombreado) se determina el color final de cada píxel. Para esto se utilizan técnicas como la interpolación de colores entre vértices y el mapeo de textura e iluminación por píxel (aproximación global). En resumen, los efectos que permiten renderizar imágenes de forma más realista son incorporados en esta etapa. En la GPU esta etapa es computada por los *pixel shaders*.

[1]En tarjetas previas a la arquitectura unificada.

En la etapa de *ROP* (del inglés *raster operation*) se aplican las restricciones finales sobre cada píxel, operaciones como mezcla de colores de triángulos superpuestos, para mejorar los efectos de transparencias y *antialiasing*. También se determinan los objetos visibles para un punto de vista dado y se descartan los píxeles ocultos.

Finalmente, la intefaz del *buffer* de imagen (FBI, del inglés *frame buffer interface*) se encarga de leer y escribir en el *buffer* de imagen.

A.2. Evolución histórica de las tarjetas gráficas

El tipo de cálculos presentes en el *pipeline* es altamente paralelo, ya que es necesario computar varias operaciones de forma independiente sobre los diferentes vértices y píxeles de la escena. Esta característica motivó desde su origen el diseño de dispositivos orientados al cálculo masivamente paralelo para su tratamiento.

En los años 80 se presentaron las primeras tarjetas gráficas de venta masiva, en un principio únicamente capaces de normalizar caracteres y, posteriormente, con capacidad para desplegar imágenes en pantalla, como era el caso de la computadora Amiga (pionera en este aspecto). En los años 90 se presentaron las primeras tarjetas con capacidad de reproducir efectos en 2D/3D. En particular, en 1997, con el desarrollo de la tarjeta Voodoo por parte de la empresa 3DFX, se produjo un salto cualitativo en las capacidades de desplegar gráficos 3D en pantalla. La tarjeta Voodoo fue rápidamente reemplazada por la Voodoo2, de la misma empresa, y seguidamente por las TNT y TNT2 de NVIDIA. Al final del siglo XX, NVIDIA diponía de las GPUs con mejores prestaciones del mercado en la familia de tarjeta gráficas GeForce (incluso apoderándose de parte de la empresa 3DFX).

Durante el periodo entre 1999 y 2006, las GPUs mejoraron su capacidad de programación. En un principio eran dispositivos con capacidad muy limitada de programación y evolucionaron, entre otros aspectos, en el manejo de aritmética de coma flotante y en la capacidad de acceder a las texturas desde los *pixel shaders*. Algunos equipos que marcaron hitos en esta evolución se enumeran a continuación. La GeForce 3 dispuso la primera GPU programable que podía ejecutar *vertex shaders* descritos en la especificación de DirectX 8, en particular pudiendo programar la etapa de *VS/T&L*. La tarjeta ATI Radeon 9700 (2002) introdujo aritmética en coma flotante con 24 bits en los *pixel shaders* (DirectX 9, OpenGL). Posteriormente, la GeForce FX introdujo la especificación de 32 bits para coma flotante. En el 2005 el XBox 360 fue pionera en presentar procesadores unificados para procesar el *pipeline* gráfico: es decir, una sola clase de procesadores para computar las distintas secciones del *pipeline*. Finalizada esta etapa, en el año 2006 las empresas ATI y NVIDIA se disputaban el liderazgo en el mercado de tarjetas con los modelos Radeon y GeForce respectivamente. En la Figura A.2 se presenta la arquitectura de las tarjetas GeForce 7800.

En 2006 NVIDIA presentó la GeForce 8800, que es la primera tarjeta con arquitectura unificada. Este cambio permitió avanzar en el desempeño de las GPUs para el proceso de gráficos, ya que el tratamiento de diferentes imágenes puede concentrar el costo de procesamiento en distintas etapas del *pipeline*: algunas en los cálculos asociados a los vértices y otras en los asociados a los píxeles. Antes, estos dos tipos de cálculos eran computados por los *pixel* y *vertex shaders* respectivamente, por lo que el rendimiento estaba acotado por unos u otros. Con esta nueva arquitectura, fácilmente se puede utilizar una cantidad diferente de procesadores en cada etapa, adaptando así la arquitectura a las necesidades de procesamiento de cada imagen. Además de acelerar el cómputo del *pipeline* gráfico, poseer una arquitectura unificada facilitó enormemente el uso de este hardware para efectuar cómputos de propósito general. Una de las cualidades del nuevo diseño que facilitó su uso para resolver problemas de propósito general consiste en disponer de un proceso orientado a

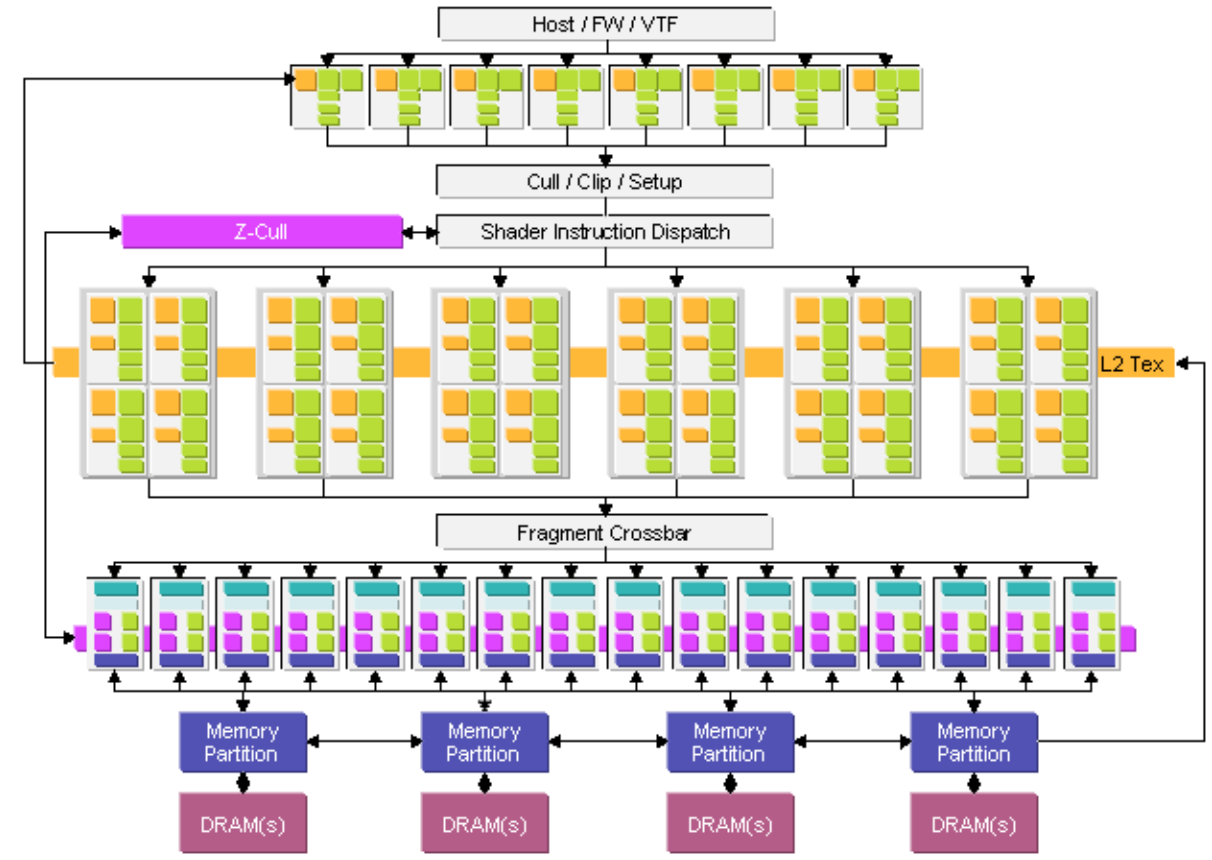

Figura A.2: Arquitectura de la tarjeta GeForce 7800. Extraído de [173].

ciclos (re-entrando los resultados intermedios a los procesadores para computar las distintas etapas del *pipeline*), en contraposición con las propuestas originales orientadas al flujo (que otorgaban el nombre de *pipeline* al proceso).

La evolución histórica de las GPUs culminó en los dispositivos actuales, que en conjunto con las capas de software desarrolladas (por ejemplo las incluidas en CUDA para las tarjetas NVIDIA), permiten ser accedidos como verdaderos multiprocesadores masivamente paralelos bajo el paradigma SPMD.

A.3. Arquitecturas unificadas de NVIDIA

A partir de 2007 con la arquitectura unificada del hardware y las herramientas CUDA para el desarollo de software para las GPUs, el cómputo de propósito general se ha centrado mayoritariamente en las tarjetas de NVIDIA. En particular, en la tesis se emplean GPUs de NVIDIA con arquitectura G80, GTX200 y Fermi, que se describen en los apartados siguientes.

Como se mostró anteriormente, para hacer un uso óptimo de las GPUs, es necesario conocer diferentes características de la tarjeta que se emplea (cantidad de núcleos, tamaño de la memoria, etc.). NVIDIA utiliza un formato estandarizado para especificar estas características, denominado *compute capabilities*. La categorización incluye dos números: los cambios en la primera cifra

implican cambios de generación, mientras en la segunda únicamente una revisión. Las primeras GPUs programables con arquitectura unificada son de *compute capability* 1.0, mientras que en la actualidad (2011) las GPUs pertenecen a la *compute capability* 2.0.

El nivel de *compute capabilty* también especifica diferentes capacidades de acceso *coalescing* a memoria. En el caso de *compute capabilities* 1.0 y 1.1, si los accesos de hilos consecutivos se encuentran separados por una palabra, no se aplica *coalescing*, y las lecturas se serializan en 16 lecturas individuales (el tamaño mínimo de una lectura es de 32 bytes). En el caso de *compute capabilities* 1.2 o superiores los accesos a memoria de hilos consecutivos no precisan ser contiguos, basta con que se encuentren en un mismo segmento de 32, 64 ó 128 bytes. Esto permite que, por ejemplo, si la lectura es de 16 valores, uno por cada hilo del *warp*, de 4 bytes separados 4 bytes entre sí, al traer el segmento entero se resuelven todas las lecturas necesarias en un único acceso a memoria de 128 bytes.

A.3.1. G80

En el Capítulo 1 se introdujo brevemente la arquitectura G80 (al presentar CUDA). La principal característica es la propuesta de arquitectura unificada. Además, con el lanzamiento de CUDA, se puede acceder a estas tarjetas con un nivel mayor de abstracción del hardware.

En la Figura A.3 se presenta un esquema de la arquitectura G80.

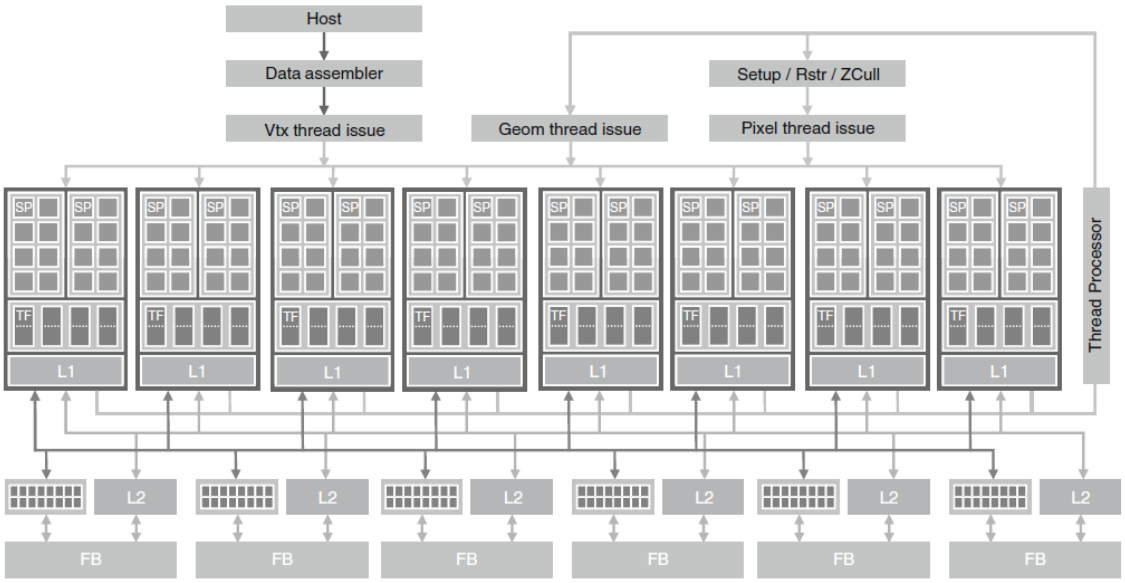

Figura A.3: Arquitectura G80 de NVIDIA. Extraído de [141].

En forma resumida, el diseño se basa en múltiples procesadores que incluyen 8 núcleos de ejecución cada uno. Además, la arquitectura dispone de una jerarquía de memoria que fue descrita en el Apartado 1.2.4.

Algunas tarjetas destacadas de esta clase son la 8800 y la 9800 GTX+.

A.3.2. GTX200

En junio de 2008 NVIDIA lanza la arquitectura GTX200, una revisión menor de la arquitectura G80.

Algunas de las características de esta revisión son:

- Aumento de los cores por tarjeta. Se pasa de un número máximo de 128 a 240 cores por tarjeta.
- Se duplica el espacio para alojar registros por multiprocesador.
- Aumento en la cantidad de hilos que se pueden ejecutar de forma concurrente: más de 30.000.
- Aritmética de doble precisión nativa. Las tarjetas con esta arquitectura permiten operaciones con aritmética de doble precisión.

Entre las tarjetas que disponen de esta arquitectura destaca la C1060, con 240 unidades computacionales y 4GB de memoria.

A.3.3. Fermi

En 2010 NVIDIA presenta una revisión mayor de la arquitectura unificada, la arquitectura Fermi. La propuesta se centra en mejorar algunos aspectos que limitaban el uso de las GPUs para computación científica. Entre otros aspectos se mejora la capacidad de cómputo en aritmética de doble precisión, se introduce memoria caché, se extiende la memoria compartida y se permite configurar a conveniencia la memoria compartida de 64K.

Algunas características mejoradas en esta arquitectura son:

- Se soporta direccionamiento de hasta 6 GB de memoria (GDDR5 DRAM).
- Se dispone de 32 núcleos por multiprocesador. En la Figura A.4 se describe la estructura de un multiprocesador de esta arquitectura.
- Se multiplica por 8 el desempeño en aritmética de doble precisión.
- Se cumple con todo el estándar de IEEE sobre coma flotante, en particular se manejan los números desnormalizados.
- Se dispone de dos despachadores duales de tareas por multiprocesador.
- Se extiende a 64 KB el tamaño de la memoria compartida.
- Se dispone de memoria caché.
- Se puede configurar la memoria compartida, para disponer de 16 KB de memoria compartida y 48 KB de caché o 48 KB para memoria compartida y 16 KB para caché.
- Se dispone de 16 unidades de *load-store* a memoria (16 operaciones de acceso a memoria por ciclo de reloj).
- Se pueden ejecutar 16 *kernels* en paralelo.

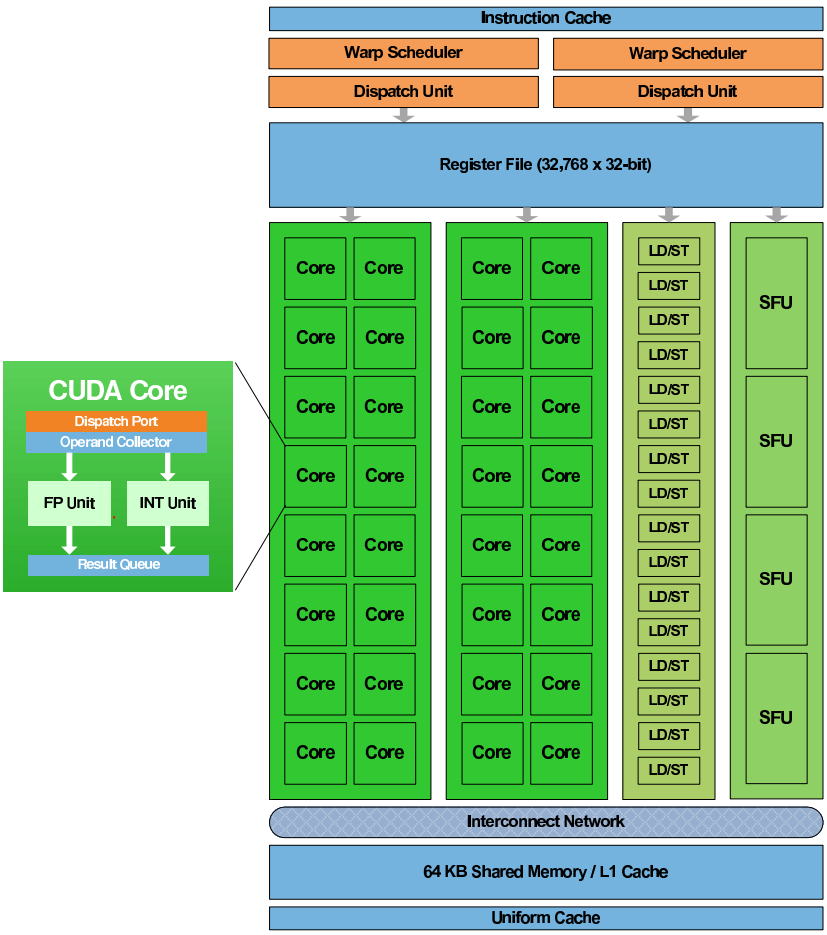

Figura A.4: Arquitectura de un multiprocesador en la arquitectura Fermi. Extraído de [176].

- Se dispone de 4 unidades especiales (SFU), para computar operaciones tipo seno, coseno, raíz cuadrada. Cada SFU ejecuta una instrucción por hilo por ciclo de reloj, por lo tanto un *warp* implica 8 ciclos. Además las SFU ejecutan de forma desacopladas de los núcleos, por lo cual se pueden realizar operaciones de forma concurrente en los núcleos.

Algunas GPUs que poseen esta arquitectura son: GTX 480, C2050 y C2070, esta última con 448 unidades computacionales y 6 GB de memoria RAM.

Bibliografía

[1] Repositorio Netlib. `www.netlib.org/`. Consultado en abril 2011.

[2] Repositorio Netlib, biblioteca LAPack. `www.netlib.org/lapack/`. Consultado en abril 2011.

[3] Repositorio Netlib, biblioteca sparse BLAS. `www.netlib.org/sparse-blas/`. Consultado en abril 2011.

[4] Sitio web de AICES. `www.aices.rwth-aachen.de/`. Consultado en abril 2011.

[5] Sitio web de benchmarks. `www.netlib.org/benchmark/`. Consultado en abril 2011.

[6] Sitio web de CULA. `www.culatools.com/`. Consultado en abril 2011.

[7] Sitio web de la biblioteca MAGMA. `http://icl.cs.utk.edu/magma/index.html`. Consultado en abril 2011.

[8] Sitio web de MPICH. `www-unix.mcs.anl.gov/mpi/mpich/`. Consultado en abril 2011.

[9] Sitio web de OpenMP. `www.openmp.org/drupal/`. Consultado en abril 2011.

[10] Sitio web del ANL. `www.anl.gov/`. Consultado en abril 2011.

[11] Sitio web del China National Supercomputing Center. `www.nscc-tj.gov.cn/en/`. Consultado en abril 2011.

[12] Sitio web del proyecto the Numerics in Control Network NICONET. `www.icm.tu-bs.de/NICONET/`. Consultado en abril 2011.

[13] Sitio web del Tokyo Institute of Technology. `www.titech.ac.jp/english/`. Consultado en abril 2011.

[14] Sito web de NIST Sparse BLAS. `http://math.nist.gov/spblas/`. Consultado en abril 2011.

[15] WRF Users page. `http://www.mmm.ucar.edu/wrf/users/`. Consultado en abril 2011.

[16] Sitio web Top 500 supercomputer, 2011. `www.top500.org/`. Consultado en abril 2011.

[17] J. Aguilar and E. Leiss. *Introducción a la computción paralela*. Editorial: Universidad de los Andes. Mérida – Venezuela, 2004.

[18] E. Agullo, H. Bouwmeester, J. Dongarra, J. Kurzak, J. Langou, and L. Rosenberg. Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. In *VECPAR*, volume 6449 of *Lecture Notes in Computer Science*, pages 129–138. Springer, 2010.

[19] E. Agullo, J. Demmel, J. Dongarra, Kurzak Hadri, B., J. J., Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180, 2009.

[20] G. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS*, pages 483–485, 1967.

[21] B.D.O. Anderson. An algebraic solution to the spectral factorization problem. *IEEE Trans. Automat. Control*, AC-12:410–414, 1967.

[22] B.D.O. Anderson. A system theory criterion for positive real matrices. *SIAM J. Cont.*, 5:171–182, 1967.

[23] M. Andrecut. Parallel GPU implementation of iterative PCA algorithms. *Computing Research Repository*, abs/0811.1081, 2008.

[24] A.C. Antoulas. *Approximation of Large-Scale Dynamical Systems*. SIAM Publications, Philadelphia, PA, 2005.

[25] A.C. Antoulas and D.C. Sorensen. Approximation of large-scale dynamical systems: An overview. *Int. J. Appl. Math. Comp. Sci.*, 11(5):1093–1121, 2001.

[26] A.C. Antoulas, D.C. Sorensen, and Y. Zhou. On the decay rate of Hankel singular values and related issues. *Sys. Control Lett.*, 46(5):323–342, 2002.

[27] H. Anzt, B. Rocker, and V. Heuveline. Energy efficiency of mixed precision iterative refinement methods using hybrid hardware platforms - an evaluation of different solver and hardware configurations. *Computer Science - R&D*, 25(3-4):141–148, 2010.

[28] W.F. Arnold, III and A.J. Laub. Generalized eigenproblem algorithms and software for algebraic Riccati equations. *Proc. IEEE*, 72:1746–1754, 1984.

[29] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An extension of the StarSs programming model for platforms with multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.

[30] M. Baboulin, J. Dongarra, and S. Tomov. Some issues in dense linear algebra for multicore and special purpose architectures. MIMS EPrint 2009.2, Manchester Institute for Mathematical Sciences, University of Manchester, Manchester, UK, jan 2009. Appears also as Technical Report UT-CS-08-615, Department of Computer Science, University of Tennessee, Knoxville, TN, USA, May 2008 and as LAPACK Working Note 200".

[31] Z. Bai and J. Demmel. Design of a parallel nonsymmetric eigenroutine toolbox, Part I. In R.F. Sincovec et al., editor, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 391–398. SIAM, Philadelphia, PA, 1993. *See also:* Tech. Report CSD-92-718, Computer Science Division, University of California, Berkeley, CA 94720.

[32] Z. Bai and J. Demmel. Using the matrix sign function to compute invariant subspaces. *SIAM J. Matrix Anal. Appl.*, 19(1):205–225, 1998.

[33] C. Bajaj, I. Ihm, J. Min, and J. Oh. SIMD optimization of linear expressions for programmable graphics hardware. *Computer Graphics Forum*, 23(4):697–714, Dec. 2004.

[34] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Ortí, and G. Quintana-Ortí. Evaluation and tuning of the level 3 CUBLAS for graphics processors. Technical report, Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaime I, Campus de Riu Sec, s/n 12.071 - Castellón, España, 2008.

[35] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Ortí, and G. Quintana-Ortí. Exploiting the capabilities of modern GPUs for dense matrix. Technical report, Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaime I, Campus de Riu Sec, s/n 12.071 - Castellón, España, 2008.

[36] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Ortí, and G. Quintana-Ortí. Glame@lab: An M-script API for linear algebra operations on graphics processors. Technical report, Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaime I, Campus de Riu Sec, s/n 12.071 - Castellón, España, 2008.

[37] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Solving dense linear systems on graphics processors. In *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 739–748, Berlin, Heidelberg, 2008. Springer-Verlag.

[38] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[39] R.H. Bartels and G.W. Stewart. Solution of the matrix equation $AX + XB = C$: Algorithm 432. *Comm. ACM*, 15:820–826, 1972.

[40] U. Baur and P. Benner. Factorized solution of the Lyapunov equation by using the hierarchical matrix arithmetic. *Proc. Appl. Math. Mech.*, 4(1):658–659, 2004.

[41] A. N. Beavers and E. D. Denman. A new solution method for the Lyapunov matrix equations. *SIAM J. Appl. Math.*, 29:416–421, 1975.

[42] A.N. Beavers and E.D. Denman. A new solution method for quadratic matrix equations. *Mathematical Biosciences*, 20:135–143, 1974.

[43] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, pages 11–14, 1995.

[44] P. Benner. *Contributions to the Numerical Solution of Algebraic Riccati Equations and Related Eigenvalue Problems*. Logos–Verlag, Berlin, Germany, 1997. *Also:* Dissertation, Fakultät für Mathematik, TU Chemnitz–Zwickau, 1997.

[45] P. Benner. Accelerating Newton's method for discrete-time algebraic Riccati equations. In A. Beghi, L. Finesso, and G. Picci, editors, *Mathematical Theory of Networks and Systems*, pages 569–572, Il Poligrafo, Padova, Italy, 1998.

[46] P. Benner. Symplectic balancing of Hamiltonian matrices. *SIAM J. Sci. Comput.*, 22(5):1885–1904, 2001.

[47] P. Benner, R. Byers, E.S. Quintana-Ortí, and G. Quintana-Ortí. Solving algebraic Riccati equations on parallel computers using Newton's method with exact line search. *Parallel Comput.*, 26(10):1345–1368, 2000.

[48] P. Benner, J.M. Claver, and E.S. Quintana-Ortí. Efficient solution of coupled Lyapunov equations via matrix sign function iteration. In A. Dourado et al., editor, *Proc.* 3rd *Portuguese Conf. on Automatic Control CONTROLO'98,* Coimbra, pages 205–210, 1998.

[49] P. Benner, R. Mayo, E. S. Quintana-Ortí, and G. Quintana-Ortí. Enhanced services for remote model reduction of large-scale dense linear systems. In Juha Fagerholm, Juha Haataja, Jari Järvinen, Mikko Lyly, Peter Råback, and Ville Savolainen, editors, *PARA*, volume 2367 of *Lecture Notes in Computer Science*, pages 329–338. Springer, 2002.

[50] P. Benner, V. Mehrmann, V. Sima, S. Van Huffel, and A. Varga. SLICOT - a subroutine library in systems and control theory. In *Applied and Computational Control, Signals, and Circuits*, pages 499–539. Birkhäuser, 1997.

[51] P. Benner and E. S. Quintana-ortí. Model reduction based on spectral projection methods. In *Dimension Reduction of Large-Scale Systems*, pages 5–45. Springer-Verlag, 2005.

[52] P. Benner, E. S. Quintana-Ortí, and G. Quintana-Ortí. A portable subroutine library for solving linear control problems on distributed memory computers. In *Workshop on Wide Area Networks and High Performance Computing*, pages 61–87, London, UK, 1999. Springer-Verlag.

[53] P. Benner and E.S. Quintana-Ortí. Solving stable generalized Lyapunov equations with the matrix sign function. *Numer. Algorithms*, 20(1):75–100, 1999.

[54] P. Benner, E.S. Quintana-Ortí, and G. Quintana-Ortí. Balanced truncation model reduction of large-scale dense systems on parallel computers. *Math. Comput. Model. Dyn. Syst.*, 6(4):383–405, 2000.

[55] P. Benner, E.S. Quintana-Ortí, and G. Quintana-Ortí. Solving linear matrix equations via rational iterative schemes. Technical Report SFB393/04-08, Sonderforschungsbereich 393 *Numerische Simulation auf massiv parallelen Rechnern*, TU Chemnitz, 09107 Chemnitz, FRG, 2004. Available from `http://www.tu-chemnitz.de/sfb393/preprints.html`.

[56] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.*, 31(1):1–26, March 2005.

[57] P. Bientinesi, B. Gunter, and R. A. van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Softw.*, 35(1):1–22, 2008.

[58] I. Blanquer, D. Guerrero, V. Hernández, E. S. Quintana-Ortí, and P. A. Ruiz. Parallel-SLICOT implementation and documentation standards. Technical report, SLICOT Working Note, 1998.

[59] D. Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25:724–734, July 2006.

[60] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[61] V. Bondhugula, N. Govindaraju, and D. Manocha. Fast SVD on Graphics Processors. Technical report, University of North Carolina, 2006.

[62] A. R. Brodtkorb. A MATLAB interface to the GPU. Master's thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, May 2007.

[63] A. R. Brodtkorb. The Graphics Processor as a Mathematical Coprocessor in MATLAB. In *CISIS '08: Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 822–827, Washington, DC, USA, 2008. IEEE Computer Society.

[64] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23:777–786, 2004.

[65] A. Bulu, J. R. Gilberti, and C. Budaka. Gaussian elimination based algorithms on the GPU. Technical report, Computer Science Department, University of California, Santa Barbara, CA 93106-5110, 2008.

[66] R. Byers. Solving the algebraic Riccati equation with the matrix sign function. *Linear Algebra Appl.*, 85:267–279, 1987.

[67] R. Byers, C. He, and V. Mehrmann. The matrix sign function method and the computation of invariant subspaces. *SIAM J. Matrix Anal. Appl.*, 18(3):615–632, 1997.

[68] M. Castillo, E. Chan, F. D. Igual, R. Mayo, Quintana-Orti, Quintana-Orti E. S., R. G., van de Geijn, and F. G. Van Zee. Making programming synonymous with programming for linear algebra libraries, FLAME working note 31. Technical Report Technical Report TR-08-20, The University of Texas at Austin, Department of Computer Sciences., April 2008.

[69] L. S. Chien. Hand Tuned SGEMM on GT200 GPU. Technical report, Department of Mathematics, Tsing Hua University, Taiwan, Feb. 2010.

[70] J. Choi, J. Dongarra, and D. Walker. PB-BLAS: A set of parallel block basic linear algebra subprograms. In *Proc. of the 1994 Scalable High Performance Computing Conference, IEEE Computer Society Press*, 1994.

[71] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Comput. Soc. Press, 1992.

[72] Colección de casos Oberwolfach de la Universidad de Freiburgo, `http://portal.uni-freiburg.de/imteksimulation/downloads/benchmark`. Consultado en abril 2011.

[73] J. Oglesby D. Tarditi, S. Puri. Accelerator: Using data parallelism to program GPUs for general-purpose uses. Technical Report MSR-TR-2005-184, 2005.

[74] J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen. Prospectus for the development of a linear algebra library for high-performance computers. Technical Report ANL/MCS-TM-97, 9700 South Cass Avenue, Argonne, IL 60439-4801, USA, 1987.

[75] E.D. Denman and A.N. Beavers. The matrix sign function and computations in systems. *Appl. Math. Comput.*, 2:63–94, 1976.

[76] U.B. Desai and D. Pal. A transformation approach to stochastic model reduction. *IEEE Trans. Automat. Control*, AC–29:1097–1100, 1984.

[77] G. F. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *PACT*, pages 353–364, 2010.

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

[79] J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Communications of the ACM*, 30(5):403–407, July 1987.

[80] P. Du, P. Luszczek, S. Tomov, and J. Dongarra. Mixed-tool performance analysis on hybrid multicore architectures. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 236–244, Washington, DC, USA, 2010. IEEE Computer Society.

[81] D. Fabregat and P. Bientinesi. Triangular linear system inversion, (unpublished manuscript), 2010.

[82] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137, New York, NY, USA, 2004. ACM.

[83] M. Fatica. Accelerating LINPACK with CUDA on heterogenous clusters. In *GPGPU*, pages 46–51, 2009.

[84] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[85] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, 21:948–960, 1972.

[86] M. Fogue, F. D. Igual, E. S. Quintana-Ortí, and Robert A. van de Geijn. Retargeting PLAPACK to clusters with hardware accelerators. In Waleed W. Smari and John P. McIntire, editors, *HPCS*, pages 444–451. IEEE, 2010.

[87] R. Freund. Model reduction methods based on Krylov subspaces. *Acta Numerica*, 12:267–319, 2003.

[88] N. Fujimoto. Faster matrix-vector multiplication on GeForce 8800GTX. In *IPDPS*, pages 1–8, 2008.

[89] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, Ralph H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[90] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC 05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[91] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.

[92] A. V. Gerbessiotis. Algorithmic and Practical Considerations for Dense Matrix Computations on the BSP Model. PRG-TR 32, Oxford University Computing Laboratory, 1997.

[93] K. Glover. All optimal Hankel-norm approximations of linear multivariable systems and their L^∞ norms. *Internat. J. Control*, 39:1115–1193, 1984.

[94] D. Göddeke and R. Strzodka. Performance and accuracy of hardware–oriented native–, emulated– and mixed–precision solvers in FEM simulations (part 2: Double precision GPUs). Technical report, Fakultät für Mathematik, TU Dortmund, July 2008. Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 370.

[95] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek. Using GPUs to improve multigrid solver performance on a cluster. *Int. J. Comput. Sci. Eng.*, 4(1):36–55, 2008.

[96] D. Göddeke, R. Strzodka, and S. Turek. Accelerating double precision FEM simulations with GPUs. Technical report, Fakultät für Mathematik, TU Dortmund, 2005. Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 292.

[97] D. Göddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in FEM simulations. *Int. J. Parallel Emerg. Distrib. Syst.*, 22(4):221–256, 2007.

[98] G. H. Golub, S. Nash, and C. F. Van Loan. A Hessenberg–Schur method for the problem $AX + XB = C$. *IEEE Trans. Automat. Control*, AC-24:909–913, 1979.

[99] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, third edition, 1996.

[100] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 102–111, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[101] K. Goto and R. A. van de Geijn. On reducing TLB misses in matrix multiplication. Technical Report CS-TR-02-55, Department of Computer Sciences, The University of Texas at Austin, 2002.

[102] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 89, New York, NY, USA, 2006. ACM.

[103] N. K. Govindaraju and D. Manocha. Cache-efficient numerical algorithms using graphics hardware. *Parallel Comput.*, 33(10-11):663–684, 2007.

[104] L. Grasedyck. Existence of a low rank or H-matrix approximant to the solution of a Sylvester equation. *Numer. Lin. Alg. Appl.*, 11:371–389, 2004.

[105] L. Grasedyck, W. Hackbusch, and B.N. Khoromskij. Solution of large scale algebraic matrix Riccati equations by use of hierarchical matrices. *Computing*, 70:121–165, 2003.

[106] M. Green. Balanced stochastic realization. *Linear Algebra Appl.*, 98:211–247, 1988.

[107] M. Green. A relative error bound for balanced stochastic truncation. *IEEE Trans. Automat. Control*, AC-33(10):961–965, 1988.

[108] M. Griebel and P. Zaspel. A multi-GPU accelerated solver for the three-dimensional two-phase incompressible navier-stokes equations. *Computer Science - R&D*, 25(1-2):65–73, 2010.

[109] W. Gropp and E. Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[110] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, sep 1996.

[111] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1994.

[112] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. `www.khronos.org/opencl/`. Consultado en abril 2011.

[113] S. Gugercin, D.C. Sorensen, and A.C. Antoulas. A modified low-rank Smith method for large-scale Lyapunov equations. *Numer. Algorithms*, 32(1):27–55, 2003.

[114] N. Gumerov, R. Duraiswami, and W. Dorland. Efficient personal supercomputing in Fortran 9x on CPU-GPU systems. Technical report, Center for Scientific Computation and Mathematical Modeling (CSCAMM), University of Maryland. CSCAMM-08-27, 2008.

[115] J. A. Gunnels and F. G. Gustavson. A new array format for symmetric and triangular matrices. In *PARA*, volume 3732 of *Lecture Notes in Computer Science*, pages 247–255. Springer, 2004.

[116] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

[117] K. K. Gupta. Solution of eigenvalue problems by sturm sequence method. *International Journal for Numerical Methods in Engineering*, 4(3):379–404, 1972.

[118] J. Gustafson and Q. Snell. Hint: A new way to measure computer performance. In *HICSS (2)*, pages 392–401, 1995.

[119] J. Hall, N. Carr, and J. Hart. Cache and bandwidth aware matrix multiplication on the GPU. Technical report, Technical Report UIUCDCS-R-20032328, University of Illinois, 2003.

[120] S.J. Hammarling. Numerical solution of the stable, non-negative definite Lyapunov equation. *IMA J. Numer. Anal.*, 2:303–323, 1982.

[121] R. Hanson, F. Krogh, and L. Lawson. A proposal for standard linear algebra subprograms. *Jet Propulsion Lab., Pasadena, Calif. TM 33-660*, 5(3), 1973.

[122] G.A. Hewer. An iterative technique for the computation of steady state gains for the discrete optimal regulator. *IEEE Trans. Automat. Control*, AC-16:382–384, 1971.

[123] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[124] N.J. Higham. Computing the polar decomposition—with applications. *SIAM J. Sci. Statist. Comput.*, 7:1160–1174, 1986.

[125] K. E. Hillesland, S. Molinov, and R. Grzeszczuk. Nonlinear optimization framework for image-based modeling on programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[126] M. Hochbruck and G. Starke. Preconditioned Krylov subspace methods for Lyapunov matrix equations. *SIAM J. Matrix Anal. Appl.*, 16(1):156–171, 1995. (See also: IPS Research Report 92–17, ETH Zürich, Switzerland (1992)).

[127] A.S. Hodel, B. Tenison, and K.R. Poolla. Numerical solution of the Lyapunov equation by approximate power iteration. *Linear Algebra Appl.*, 236:205–230, 1996.

[128] W.D. Hoskins, D.S. Meek, and D.J. Walton. The numerical solution of $A'Q + QA = -C$. *IEEE Trans. Automat. Control*, AC-22:882–883, 1977.

[129] J. Humphrey, D. Price, K. Spagnoli, A. Paolini, and E. Kelmelis. CULA: Hybrid GPU accelerated linear algebra routines. In *SPIE Defense and Security Symposium (DSS)*, 2010.

[130] F. Ino, M. Matsui, K. Goda, and K. Hagihara. Performance study of LU decomposition on the programmable GPU. In *HiPC*, pages 83–94, 2005.

[131] Intel Corporation., `http://www.intel.com/`. Consultado en abril 2011.

[132] I.M. Jaimoukha and E.M. Kasenally. Krylov subspace methods for solving large Lyapunov equations. *SIAM J. Numer. Anal.*, 31:227–251, 1994.

[133] B. Jang, D. Kaeli, S. Do, and H. Pien. Multi GPU implementation of iterative tomographic reconstruction algorithms. In *ISBI'09: Proceedings of the Sixth IEEE international conference on Symposium on Biomedical Imaging*, pages 185–188, Piscataway, NJ, USA, 2009. IEEE Press.

[134] C. Jiang and M. Snir. Automatic tuning matrix multiplication performance on graphics hardware. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 185–196, Washington, DC, USA, 2005. IEEE Computer Society.

[135] J. H. Jung. Cholesky decomposition and linear programming on GPU. Technical report, Scholarly Paper Directed by D. O'leary, 2006.

[136] J. H. Jung and D. O'leary. Exploiting structure of symmetric or triangular matrices on a GPU. In *First Workshop on General Purpose Processing on Graphics Processing Units*, Northeastern Univ., Boston, 2007, 2007.

[137] T. Kailath. *Systems Theory*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[138] C. Kenney and A.J. Laub. On scaling Newton's method for polar decomposition and the matrix sign function. *SIAM J. Matrix Anal. Appl.*, 13:688–706, 1992.

[139] C. Kenney and A.J. Laub. The matrix sign function. *IEEE Trans. Automat. Control*, 40(8):1330–1348, 1995.

[140] A. Kerr, D. Campbell, and M. Richards. QR decomposition on GPUs. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 71–78, New York, NY, USA, 2009. ACM.

[141] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[142] M.M. Konstantinov, V. Mehrmann, and P.Hr. Petkov. Perturbation analysis for the Hamiltonian Schur form. *SIAM J. Matrix Anal. Appl.*, 23(2):387–424, 2001.

[143] D. Kressner. Estrategias de refinamiento iterativo para ecuaciones de Lyapunov, (comunicación personal), 2009.

[144] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22:908–916, 2003.

[145] J. Krüger, T. Schiwietz, P. Kipfer, and R. Westermann. Numerical simulations on PC graphics hardware. In *ParSim 2004 (Special Session of EuroPVM/MPI 2004)*, Budapest, Hungary, 2004.

[146] P. Lancaster and M. Tismenetsky. *The Theory of Matrices*. Academic Press, Orlando, 2nd edition, 1985.

[147] W. Lang and U. Lezius. Numerical realization of the balanced reduction of a control problem. In H. Neunzert, editor, *Progress in Industrial Mathematics at ECMI94*, pages 504–512. John Wiley & Sons Ltd and B.G. Teubner, New York and Leipzig, 1996.

[148] V.B. Larin and F.A. Aliev. Construction of square root factor for solution of the Lyapunov matrix equation. *Sys. Control Lett.*, 20:109–112, 1993.

[149] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, pages 55–55, New York, NY, USA, 2001. ACM.

[150] A.J. Laub. Numerical linear algebra aspects of control design computations. *IEEE Trans. Automat. Control*, AC-30:97–108, 1985.

[151] A.J. Laub, M.T. Heath, C.C. Paige, and R.C. Ward. Computation of system balancing transformations and other application of simultaneous diagonalization algorithms. *IEEE Trans. Automat. Control*, 34:115–122, 1987.

[152] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Trans. Graph.*, 25(1):60–99, 2006.

[153] C. Lessig. Eigenvalue computation with CUDA. Technical report, Nvidia Corporation. Available at: `developer.download.nvidia.com/compute/cuda/1_1/Website/projects/eigenvalues/doc/eigenvalues.pdf`, 2007.

[154] J.-R. Li and J. White. Reduction of large circuit models via low rank approximate gramians. *Int. J. Appl. Math. Comp. Sci.*, 11(5):1151–1171, 2001.

[155] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[156] Y. Liu and B.D.O. Anderson. Controller reduction via stable factorization and balancing. *Internat. J. Control*, 44:507–531, 1986.

[157] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra. A scalable high performant Cholesky factorization for multicore with GPU accelerators. In *VECPAR*, volume 6449 of *Lecture Notes in Computer Science*, pages 93–101. Springer, 2010.

[158] M. Marqués, G. Quintana-Ortí, E. S. Quintana-Ortí, and R. A. van de Geijn. Using graphics processors to accelerate the solution of out-of-core linear systems. In *ISPDC*, pages 169–176, 2009.

[159] J. Michalakes and M. Vachharajani. GPU acceleration of numerical weather prediction. In *IPDPS*, pages 1–7, 2008.

[160] T. Möller and E. Haines. *Real-time rendering*. Ak Peters Series. AK Peters, 2002.

[161] B.C. Moore. Principal component analysis in linear systems: Controllability, observability, and model reduction. *IEEE Trans. Automat. Control*, AC-26:17–32, 1981.

[162] A. Moravánszky and N. Ag. Dense matrix algebra on the GPU. In *In Direct3D ShaderX2, Engel W. F., (Ed.). Wordware Publishing*, page 2. NovodeX AG, 2003.

[163] K. Moreland and E. Angel. The FFT on a GPU. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[164] D. Mukunoki and D. Takahashi. Implementation and evaluation of quadruple precision BLAS on GPU. *IPSJ SIG Notes*, 2009(13):1–6, 2009-11-23.

[165] C. Mullis and R.A. Roberts. Synthesis of minimum roundoff noise fixed point digital filters. *IEEE Trans. Circuits and Systems*, CAS-23(9):551–562, 1976.

[166] C. Narus, M. Narus, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[167] R. Nath, S. Tomov, and J. Dongarra. Accelerating GPU kernels for dense linear algebra. In *VECPAR*, volume 6449 of *Lecture Notes in Computer Science*, pages 83–92. Springer, 2010.

[168] R. Nath, S. Tomov, and J. Dongarra. BLAS for GPUs. In J. Kurzak, D. A. Bader, and J. and Dongarra, editors, *Scientific Computing with Multicore and Accelerators*. CRC Press, Dec. 2010.

[169] R. Nath, S. Tomov, and J. Dongarra. An improved MAGMA GEMM for Fermi graphics processing units. *IJHPCA*, 24(4):511–515, 2010.

[170] A. Neic, M. Liebmann, and G. Haas. Algebraic multigrid solver on clusters of CPUs and GPUs. In *Applied Parallel Computing, State of the Art in Scientific Computing, 10th International Workshop, PARA 2010*, 2010.

[171] A. Nukada and S. Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 30:1–30:10, New York, NY, USA, 2009. ACM.

[172] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 5:1–5:11, Piscataway, NJ, USA, 2008. IEEE Press.

[173] Nvidia. *NVIDIA GeForce 7800 GPU Architecture Overview*, 2005.

[174] Nvidia. *NVIDIA GeForce 8800 GPU Architecture Overview*, 2006.

[175] Nvidia. *NVIDIA GeForce GTX 200 GPU Architectural Overview*, 2008.

[176] Nvidia. *NVIDIAs Next Generation CUDA Compute Architecture: Fermi*, 2010.

[177] G. Obinata and B.D.O. Anderson. *Model Reduction for Control System Design*. Communications and Control Engineering Series. Springer-Verlag, London, UK, 2001.

[178] S. Ohshima, K. Kise, T. Katagiri, and T. Yuba. Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment. In *In 7th International Meeting on High Performance Computing for Computational Science (VECPAR 06)*, pages 41–50, 2006.

[179] J. D. Owens, D. Luebke, N. Govindaraju, J. Harris, M.and Kruger, A. E. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[180] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, September 2005.

[181] G. Quintana P. Benner, E. S. Quintana. Solving linear-quadratic optimal control problems on parallel computers. *Optimization Methods & Software*, 23(6):879–909, 2008.

[182] M. Peercy, M. Segal, and D. Gerstmann. A performance-oriented data parallel virtual machine for GPUs. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[183] T. Penzl. A cyclic low rank Smith method for large, sparse Lyapunov equations with applications in model reduction and optimal control. Technical Report SFB393/98-6, Fakultät für Mathematik, TU Chemnitz, 09107 Chemnitz, FRG, 1998. Available from `http://www.tu-chemnitz.de/sfb393/sfb98pr.html`.

[184] T. Penzl. A cyclic low rank Smith method for large sparse Lyapunov equations. *SIAM J. Sci. Comput.*, 21(4):1401–1418, 2000.

[185] P.H. Petkov, N.D. Christov, and M.M. Konstantinov. *Computational Methods for Linear Control Systems*. Prentice-Hall, Hertfordshire, UK, 1991.

[186] E.S. Quintana-Ortí, G. Quintana-Ortí, X. Sun, and R.A. van de Geijn. A note on parallel matrix inversion. *SIAM J. Sci. Comput.*, 22:1762–1771, 2001.

[187] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 121–130, New York, NY, USA, 2008. ACM.

[188] A. Remón, E. Quintana-Ortí, and G. Quintana-Ortí. Parallel solution of band linear systems in model reduction. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 4967 of *Lecture Notes in Computer Science*, pages 678–687. Springer Berlin / Heidelberg, 2008.

[189] F. Ries, T. De Marco, M. Zivieri, and R. Guerrieri. Triangular matrix inversion on graphics processing unit. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 9:1–9:10, New York, NY, USA, 2009. ACM.

[190] J.D. Roberts. Linear model reduction and solution of the algebraic Riccati equation by use of the sign function. *Internat. J. Control*, 32:677–687, 1980. (Reprint of Technical Report No. TR-13, CUED/B-Control, Cambridge University, Engineering Department, 1971).

[191] M. Rumpf and R. Strzodka. Using graphics cards for quantized FEM computations. In *VIIP*, pages 193–202, 2001.

[192] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[193] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, and W. W. Hwu. Program optimization study on a 128-core GPU. In *In The First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[194] Y. Saad. Numerical solution of large Lyapunov equation. In M. A. Kaashoek, J. H. van Schuppen, and A. C. M. Ran, editors, *Signal Processing, Scattering, Operator Theory and Numerical Methods*, pages 503–511. Birkhäuser, 1990.

[195] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[196] J. Saak. Effiziente numerische Lösung eines Optimalsteuerungsproblems für die Abkühlung von stahlprofilen. Diplomarbeit, Fachbereich 3/Mathematik und Informatik, Universität Bremen, D-28334 Bremen, September 2003.

[197] M.G. Safonov and R.Y. Chiang. A Schur method for balanced-truncation model reduction. *IEEE Trans. Automat. Control*, AC–34:729–733, 1989.

[198] S. Sahni and V. Thanvantri. Parallel computing: Performance metrics and models, 1995. Research Report, Computer Science Department, University of Florida.

[199] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, Aug. 2005.

[200] G. W. Stewart. *Matrix Algorithms: Volume 1, Basic Decompositions*. Society for Industrial Mathematics, 1998.

[201] J. A. Stratton, S. Stone, and W. W. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers*, pages 16–30, Berlin, Heidelberg, 2008. Springer-Verlag.

[202] P. Strazdins. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Technical Report TR-CS-98-07, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, 1998.

[203] G. Thomas. BLASTH, a BLAS library for dual SMP computer. In *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*, pages 24–24, Berkeley, CA, USA, 2000. USENIX Association.

[204] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1999.

[205] S. Toledo and F. G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Proceedings of the fourth workshop on I/O in parallel and distributed systems: part of the federated computing research conference*, IOPADS '96, pages 28–40, New York, NY, USA, 1996. ACM.

[206] M.S. Tombs and I. Postlethwaite. Truncated balanced realization of a stable non-minimal state-space system. *Internat. J. Control*, 46(4):1319–1330, 1987.

[207] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. MIMS EPrint 2009.7, Manchester Institute for Mathematical Sciences, University of Manchester, Manchester, UK, jan 2009. Appears also as Technical Report UT-CS-08-632, Department of Computer Science, University of Tennessee, Knoxville, TN, USA, October 2008 and as LAPACK Working Note 210".

[208] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. MIMS EPrint 2009.7, Manchester Institute for Mathematical Sciences, University of Manchester, Manchester, UK, jan 2009.

[209] S. Tomov, R. Nath, and J. Dongarra. Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Computing*, 36(12):645–654, 2010.

[210] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *IPDPS Workshops*, pages 1–8. IEEE, 2010.

[211] F. Tröltzsch and A. Unger. Fast solution of optimal control problems in the selective cooling of steel. *Z. Angew. Math. Mech.*, 81:447–456, 2001.

[212] The University of Texas at Austin, `http://www.cs.utexas.edu/~flame/`. Consultado en abril 2011. *Sitio web de la biblioteca FLAME*.

[213] H. Migallón V. Migallón J. Penadés V. Galiano, A. Martín and E.S. Quintana-Ortí. PyPLiC: A high-level interface to the parallel model reduction library PLiCMR. In B. H. V. Topping,

editor, *Proceedings of the Eleventh International Conference on Civil, Structural and Environmental Engineering Computing*, Stirlingshire, United Kingdom, 2007. Civil-Comp Press. paper 62.

[214] P. Van Dooren. Gramian based model reduction of large-scale dynamical systems. In D.F. Griffiths and G.A. Watson, editors, *Numerical Analysis 1999. Proc. 18th Dundee Biennial Conference on Numerical Analysis*, pages 231–247, London, UK, 2000. Chapman & Hall/CRC.

[215] A. Varga. Efficient minimal realization procedure based on balancing. In *Prepr. of the IMACS Symp. on Modelling and Control of Technological Systems*, volume 2, pages 42–47, 1991.

[216] A. Varga. Task II.B.1 – selection of software for controller reduction. SLICOT Working Note 1999–18, The Working Group on Software (WGS), December 1999. Available from `http://www.win.tue.nl/niconet/NIC2/reports.html`.

[217] A. Varga and K. H. Fasol. A new square–root balancing–free stochastic truncation model reduction algorithm. In *Prepr. 12th IFAC World Congress*, volume 7, pages 153–156, Sydney, Australia, 1993.

[218] V Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[219] V. Volkov and J. Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[220] E.L. Wachspress. ADI iteration parameters for the Sylvester equation, 2000. available from the author.

[221] R. C. Whaley and A Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.