PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tesis de Maestría
en Informática

Aceleración de métodos de reducción de modelos dispersos en arquitecturas multi many-core

Ernesto Dufrechou

2015

Ermestp Dufrechou
Aceleración de métodos de reduccoón de modelos dispersos en
Arquitecturas multi/many-core
ISSN 0797-6410
Tesis de Maestría en Informática
Reporte Técnico RT 15-02
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República.
Montevideo, Uruguay, 2015

Autor: Ernesto Dufrechou (edufrechou@fing.edu.uy)

Título: Aceleración de métodos de reducción de modelos dispersos en arquitecturas multi/many-core. (tesis de maestría)

Director de tesis: Pablo Ezzatti, INCO - Facultad de Ingeniería - UdelaR (pezzatti@fing.edu.uy)

Co-Director de tesis: Enrique S. Quintana-Ortí, Universidad Jaume I - Castellón, España (quintana@uji.es)

Fecha: 16 de Diciembre 2014

Resumen: La simulación de procesos naturales mediante herramientas computacionales suelen involucrar modelos numéricos con grandes períodos de tiempo o dimensiones importantes en los sistemas de ecuaciones a resolver. Para abordar este tipo de problemas las técnicas de reducción de modelos (MOR) apuntan a transformar el modelo matemático original de un problema en otro modelo mucho más simple pero que mantenga las principales características matemáticas del modelo original. Muchos métodos de MOR se basan fuertemente en la resolución de sistemas lineales dispersos de gran dimensión, lo cual implica importantes esfuerzos computacionales y motiva el uso de técnicas de HPC.

En los últimos años, las GPU como plataforma de HPC, así como las arquitecturas heterogéneas (por ejemplo procesadores multi-core junto a GPUs) han experimentado una rápida evolución dado su bajo costo económico y alta eficiencia.

Considerando lo anterior, el objetivo principal de la tesis es el desarrollo de un prototipo de biblioteca para la resolución de problemas de MOR dispersos, que permita explotar el paralelismo de arquitecturas híbridas compuestas por procesadores multi-core y GPUs.

Palabras claves: GPU, Reducción de Modelos, Matrices Dispersas, Lyapack.

Universidad de la República
pedeciba informática

Aceleración de métodos de reducción de modelos dispersos en arquitecturas multi/many-core

Tesis de Maestría presentada por

Ernesto Dufrechou

Montevideo, Diciembre de 2014

Director de tesis: Pablo Ezzatti
Co-Director de tesis: Enrique S. Quintana-Ortí

Universidad de la República
pedeciba informática

Aceleración de métodos de reducción de modelos dispersos en arquitecturas multi/many-core

Ernesto Dufrechou

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El estudio de sistemas dinámicos es un componente fundamental de varias áreas de la ingeniería así como también de otras disciplinas, como por ejemplo en Teoría de Control, Procesamiento de Señales, Análisis Estructural o Economía. En algunos casos, el estudio se aborda mediante un enfoque empírico basado en la manipulación de las entradas de dicho sistema y la posterior medición de los resultados con el fin de realizar ajustes [30]. Sin embargo, por lo general, estos sistemas son demasiado complejos, o su construcción muy costosa o peligrosa, por lo que es necesario emplear otro tipo de técnicas. Es por esto que dichos sistemas suelen describirse mediante modelos físicos, y luego, aplicando distintas leyes de la Física, es posible desarrollar sistemas de ecuaciones matemáticas que describan su comportamiento. Estas ecuaciones matemáticas pueden ser de distinta índole, dependiendo del problema que se intente describir. Cuando estas ecuaciones son lineales se habla de un sistema dinámico lineal.

Aunque el modelado de sistemas físicos mediante estas técnicas permite simular el comportamiento de los mismos con la ayuda de una computadora, conforme la dimensión de los sistemas lineales correspondientes crece (lo cual depende de la cantidad de variables involucradas en el modelo), la cantidad de operaciones necesarias para resolver estos sistemas y el costo computacional que esto requiere aumenta considerablemente. Pese al notable avance de la computación en los últimos años, sobre todo en cuanto a la aparición de nuevas plataformas de computación de alto desempeño, los costos computacionales para resolver problemas de gran tamaño son, a veces, prohibitivos.

Es interesante entonces la aplicación de técnicas que permitan realizar simulaciones en tiempos razonables con el fin de evaluar diseños y analizar el comportamiento de estos sistemas. En este sentido, la reducción del orden de modelos [6] (MOR por sus siglas en inglés) o reducción de modelos, es una herramienta matemática que tiene como objetivo, dado un modelo analítico de un sistema, encontrar otro cuya dimensión sea considerablemente menor, pero que presente un comportamiento similar al del modelo original. Debido a su menor dimensión, este modelo reducido puede ser utilizado en simulaciones posteriores con una menor demanda de recursos de cómputo y tiempo de ejecución pero, a su vez, sin una pérdida significativa en la calidad de los resultados. Por ejemplo, un modelo de 500 ecuaciones puede ser reemplazado por uno de 10 ecuaciones incurriendo en un error global menor a la precisión de la máquina [21]. Este enfoque es particularmente útil en el caso de la discretización de problemas que aparecen en la resolución de ecuaciones diferenciales parciales asociadas a escenarios 3D, las cuales pueden ser reducidas a una cantidad de ecuaciones mucho menor.

Formalmente, un sistema dinámico lineal continuo e invariante en el tiempo puede describirse

mediante dos ecuaciones, 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 $u(t) \in \mathbb{R}^m$ es la señal de entrada, $y(t) \in \mathbb{R}^p$ es la salida del sistema y $x(t) \in \mathbb{R}^n$ es el vector de variables de estado, por lo cual esta representación es conocida como ecuaciones del *espacio de estados*. A su vez, la matriz $A \in \mathbb{R}^{n \times n}$ es llamada matriz de estados del sistema, y en este caso es de orden n.

Considerando esta descripción de un sistema diámico lineal continuo invariante en el tiempo, los métodos de reducción de modelos buscan obtener un segundo sistema descrito por

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

de orden $r \ll n$, que mantenga la información fundamental sobre la dinámica del sistema original. Bajo estas premisas, el nuevo sistema es una aproximación del original, lo que permite reemplazarlo en sucesivos análisis o simulaciones.

Existen distintos métodos matemáticos para transformar un modelo numérico de gran dimensión en otro de dimensión menor. Muchos de estos métodos tienen como etapa fundamental la obtención de los gramianos del sistema, los cuales son la solución de cierto par de ecuaciones algebraicas de Lyapunov. Para obtener estos gramianos, por ejemplo, en la reducción de modelos densos de gran dimensión mediante arquitecturas paralelas, suelen utilizarse enfoques basados en la función signo [23]. Sin embargo, si bien pueden ser adecuados para casos densos, estos enfoques no tienen un buen desempeño cuando las matrices involucradas son dispersas, ni pueden sacar partido de propiedades estructurales particulares de dichas matrices.

Para los casos donde la matriz de estados es grande y dispersa, durante los últimos 15 años se han desarrollado un conjunto de algoritmos basados en el método *Alternating Direction Implicit* (ADI) para resolver ecuaciones en derivadas parciales parabólicas, hiperbólicas y elípticas. Este método, que ha sido ampliamente difundido y utilizado en problemas de reducción de modelos dispersos, actualmente es un componente fundamental de Lyapack [64], una conocida biblioteca para resolver problemas de control en MATLAB, y ha sido reformulado en la biblioteca M.E.S.S. (*Matrix Equations Sparse Solvers*) [73] que se plantea como la sucesora de Lyapack.

Desde el punto de vista computacional, todos estos métodos involucran la aplicación de herramientas numéricas (cálculo de las matrices gramianas, descomposición en valores singulares, etc.) sobre matrices de gran tamaño, por lo que en algunos casos, la aplicación de técnicas de MOR implica un importante esfuerzo computacional. Esto último ha despertado el interés en emplear técnicas del área de la Computación de Alto Desempeño (HPC por sus siglas en inglés). A manera de ejemplo puede mencionarse la biblioteca PLiC [22] (*Parallel Library in Control*), que aplica técnicas de paralelismo en memoria compartida y distribuida para resolver problemas de reducción de modelos. Lamentablemente, la aceleración de problemas mediante este tipo de técnicas llevan asociado un alto costo económico. Al esfuerzo que requiere el desarrollo de las soluciones, se agrega el altísimo costo de las plataformas de HPC (por ejemplo clusters) y el importante consumo energético de las mismas.

En los últimos años ha sido posible observar una rápida evolución de arquitecturas no tradicionales de HPC como los procesadores multi-core, los procesadores gráficos (GPUs por sus siglas en inglés) y las arquitecturas heterogéneas (procesadores multi-core junto a GPUs, ARMs, DSPs, etc.). Entre las razones que sustentan este cambio se encuentran el bajo costo económico de este tipo de arquitecturas, que permite que sean accesibles en forma masiva, y su mayor eficiencia energética.

En particular, la importante demanda impulsada por el mercado de los videojuegos, relacionada con la producción de gráficos 3D más complejos y realistas, ha motivado la evolución de las GPUs para convertirlas en procesadores masivamente paralelos, capaces de ejecutar cientos y hasta miles de hilos concurrentemente, con una impresionante potencia computacional y ancho de banda de memoria.

Teniendo en cuenta el reciente desarrollo de técnicas de MOR para casos dispersos, su considerable costo computacional para tratar problemas de gran tamaño, y las nuevas posibilidades que ofrecen las plataformas de HPC de bajo costo, el *objetivo principal* del presente trabajo es diseñar y desarrollar una biblioteca para resolver este tipo de problemas que incluya técnicas de HPC basadas en arquitecturas híbridas, especialmente aquellas compuestas por procesadores multi-core y GPUs.

Adicionalmente, se requiere que la biblioteca desarrollada sea flexible y fácil de utilizar, manteniendo el espíritu de bibliotecas previas ampliamente utilizadas por ingenieros y científicos no necesariamente expertos en computación.

El camino para la concreción del *objetivo principal* de la tesis plantea los siguientes *objetivos específicos*:

- Realizar un relevamiento del estado del arte referente a problemas de MOR dispersos y computación de alto desempeño, en particular utilizando GPUs.
- En base a lo relevado, realizar el diseño de un prototipo de biblioteca para la resolución de problemas de MOR dispersos que explote arquitecturas híbridas (CPUs multi-core y GPUs).
- Validación del diseño propuesto.
- Teniendo en cuenta la validación realizada, efectuar las correcciones pertinentes en el diseño e implementar finalmente un prototipo de biblioteca de MOR para problemas dispersos sobre arquitecturas híbridas (CPUs multi-core y GPUs).
- Evaluar de la biblioteca desarrollada.

1.1. Estructura de la tesis

Luego de esta breve introducción, el Capítulo 2 versa sobre el problema de reducción de modelos. En primer lugar se introducen algunos conceptos básicos sobre sistemas dinámicos lineales y se describe matemáticamente el problema de reducción de modelos. Tras presentar algunos métodos computacionales para abordar la problemática, se profundiza sobre el método de Truncamiento Balanceado y el método de la Raíz Cuadrada, los cuales son de particular interés en este trabajo. Seguidamente, se describe con más detalle un método para obtener la solución de las ecuaciones de Lyapunov adecuado para matrices dispersas, ya que la solución de estas ecuaciones es un paso fundamental en los algoritmos anteriores. En la sección de trabajo relacionado, se realiza primeramente una reseña sobre la aplicación de técnicas de HPC al problema de reducción de modelos. En segundo lugar se ofrece un relevamiento del estado del arte en la resolución de problemas de reducción de modelos y control utilizando procesadores gráficos. Por último, se describen brevemente dos bibliotecas para problemas de control y reducción de modelos dispersos ampliamente aceptadas y utilizadas por la comunidad.

El Capítulo 3 presenta y describe la solución desarrollada en este trabajo. Comienza con una introducción en la que se describen algunos conceptos sobre el diseño de la biblioteca Lyapack. Tras esto, se detalla el diseño de la extensión de la biblioteca para incluir cómputo basado en GPU.

En las siguientes secciones se describe la implementación de las funciones de desarrolladas para las distintas familias de matrices dispersas cubiertas, a saber, matrices tri-diagonales, generales banda, y dispersas. En el caso de las matrices tri-diagonales, se describe la implementación de un resolutor paralelo para sistemas de ecuaciones lineales basado en el algoritmo de reducción cíclica, y se describe también su integración con la biblioteca Lyapack. En el caso de las matrices banda, se introduce en primer lugar la problemática relacionada con la solución de sistemas lineales de este tipo y cómo se resuelven actualmente mediante la biblioteca LAPACK. Luego, se presenta un conjunto de nuevas rutinas para explotar arquitecturas híbridas (CPU-GPU) desarrolladas en este trabajo para resolver sistemas lineales banda en este tipo de plataformas. Específicamente, se describe un algoritmo híbrido "por bloques" para realizar la factorización LU de la matriz de coeficientes junto con una versión del algoritmo que incorpora técnicas de *look-ahead*, otra rutina para resolver los sistemas lineales generados luego de la factorización, y una rutina que fusiona una etapa de las operaciones anteriores logrando un interesante ahorro en el tiempo de ejecución. En el caso de las matrices dispersas generales se detalla la implementación de un resolutor iterativo basado en los métodos del Gradiente Conjugado y Gradiente Conjugado Estabilizado. Seguidamente, se discute brevemente sobre el uso de precondicionadores en métodos iterativos, y a continuación se describen los trabajos realizados sobre ILUPACK, una biblioteca de métodos iterativos para resolver sistemas de ecuaciones lineales dispersos que incluye un poderoso precondicionador multinivel, con el objetivo de acelerar la aplicación de dicho precondicionador en arquitecturas híbridas.

En el Capítulo 4 se describe la evaluación experimental de la solución propuesta. El mismo comienza con una revisión de las plataformas de cómputo sobre las que se realizaron los experimentos. A continuación, se detalla la evaluación de la solución para cada tipo de matriz descrito anteriormente. Para el caso de banda, primero se realiza una evaluación de las nuevas rutinas presentadas en el Capítulo 3, y luego se evalúa la aplicación de estas rutinas a un caso de reducción de modelos. De forma similar, en el caso de las matrices dispersas generales se realiza, en primer lugar, una evaluación de la versión acelerada con GPU de ILUPACK. Seguidamente, se realiza un estudio con el fin de comprobar las ventajas de utilizar el precondicionador de ILUPACK, comparando distintas estrategias de precondicionado. Por último, se realizan mediciones del desempeño de estas estrategias aplicadas a la solución de las ecuaciones de Lyapunov mediante el método LRCF-ADI.

Posteriormente, en el Capítulo 5 se ofrecen algunas conclusiones, se relacionan los principales resultados obtenidos en forma de publicaciones y se describen las líneas de trabajo futuro.

Finalmente, algunos temas específicos que conforman un marco teórico sobre ciertos aspectos de la tesis se incluyen en forma de anexos. Primero se exponen algunos conceptos sobre computación de propósito general con GPUs. En el segundo anexo se brinda una descripción de las principales bibliotecas de álgebra lineal numérica utilizadas en la tesis. Seguidamente, se ofrece un relevamiento del estado del arte en métodos iterativos utilizando GPUs. Por último se exponen algunos aspectos técnicos acerca de ILUPACK, su precondicionador multinivel y la derivación matemática de su aplicación.

Capítulo 2

Reducción de modelos

El estudio analítico de sistemas físicos se vuelve indispensable en aquellos casos en los que la complejidad del sistema, u otros factores como por ejemplo el costo económico de su construcción, vuelven inviable un enfoque empírico basado en la manipulación de las entradas de dicho sistema y la posterior medición de los resultados con el fin de realizar ajustes [30]. Es por eso que dichos sistemas suelen describirse mediante modelos físicos, y luego, aplicando distintas leyes de la Física es posible desarrollar sistemas de ecuaciones matemáticas que describan el comportamiento de los mismos. Estas ecuaciones matemáticas pueden ser de distinta índole, dependiendo del problema que se intente describir. Cuando estas ecuaciones son lineales se habla de un sistema dinámico lineal.

Todo sistema dinámico lineal puede describirse en base a sus entradas u y salidas como

$$y(t) = \int_{t_0}^{t} G(t,\tau)u(\tau)d\tau. \tag{2.1}$$

Si las variables dependientes del sistema se expresan únicamente en función del tiempo t, el sistema también puede ser expresado por dos ecuaciones matriciales de la siguiente forma:

$$\begin{array}{rcll} \dot{x}(t) & = & A(t)x(t) + B(t)u(t), & t > 0, \qquad x(0) = x_0, \\ y(t) & = & C(t)x(t) + D(t)u(t), & t \geq 0, \end{array} \tag{2.2}$$

En esta representación, $x(t) \in \mathbb{R}^n$ es el vector de variables de estado, por lo cual la representación es conocida como ecuaciones del *espacio de estados*.

Si, adicionalmente, el sistema es invariante en el tiempo, las ecuaciones (2.1) y (2.2) se transforman en:

$$y(t) = \int_{0}^{t} G(t-\tau)u(\tau)d\tau \tag{2.3}$$

y

$$\begin{array}{rcll} \dot{x}(t) & = & Ax(t) + Bu(t), & t > 0, \qquad x(0) = x_0, \\ y(t) & = & Cx(t) + Du(t), & t \geq 0, \end{array} \tag{2.4}$$

Para sistemas dinámicos lineales e invariantes en el tiempo (LTI por sus siglas en inglés) la transformada de Laplace es una herramienta útil a la hora de su diseño y análisis [30]. Aplicando esta transformada sobre la ecuación (2.3) y asumiendo $x_0 = 0$, se obtiene otra representación del sistema de tipo entrada/salida:

$$y(s) = \hat{G}(s)\hat{u}(s), \tag{2.5}$$

donde

$$G(s) = C(sI_n - A)^{-1}B + D, \tag{2.6}$$

es llamada *matriz de transferencia*, siendo I_n la matriz identidad de orden n. Esta representación del sistema es descrita habitualmente como representación *en el dominio de la frecuencia*, en contraste con la primera, que usualmente es descrita como *en el dominio del tiempo*.

Aunque el modelado de sistemas físicos mediante estas técnicas permite simular el comportamiento de los mismos con la ayuda de una computadora, conforme la dimensión de los sistemas dinámicos lineales correspondientes crece (lo cual depende de la cantidad de variables involucradas en el modelo), la cantidad de operaciones necesarias para trabajar con estos sistemas y el costo computacional que esto requiere aumenta considerablemente. Pese al notable avance de la computación en los últimos años, sobre todo en cuanto a la aparición de nuevas plataformas de computación de alto desempeño, los costos computacionales correspondientes a resolver problemas de gran tamaño son, a veces, prohibitivos.

Es interesante entonces la aplicación de técnicas que permitan realizar simulaciones en tiempos razonables con el fin de evaluar diseños y analizar el comportamiento de estos sistemas. En este sentido, la *reducción del orden de modelos*[1] (MOR por sus siglas en inglés) es una herramienta matemática que tiene como objetivo, dado un modelo analítico de un sistema, encontrar otro cuya dimensión sea considerablemente menor pero que presente un comportamiento similar al del modelo original. Debido a su menor dimensión, este modelo reducido puede ser utilizado en simulaciones posteriores con una demanda de recursos de cómputo y tiempo de ejecución inferior pero, a su vez, sin una pérdida significativa en la calidad de los resultados.

Considerando el sistema dinámico LTI de orden n presentado en (2.4) los métodos de reducción de modelos buscan obtener un segundo sistema,

$$\begin{array}{rcl} \dot{x}_r(t) & = & A_r x_r(t) + B_r u(t), \quad t > 0, \quad x_r(0) = \bar{x}_0, \\ y_r(t) & = & C_r x_r(t) + D_r u(t), \quad t \geq 0, \end{array} \tag{2.7}$$

de orden r mucho menor que n, que mantenga la información fundamental sobre la dinámica del sistema original. Bajo estas premisas el nuevo sistema en (2.7) es una aproximación del original en (2.4), lo que permite reemplazar a este último en sucesivos análisis o simulaciones.

Existen varias maneras de determinar la diferencia entre ambos sistemas. Frecuentemente esta diferencia es calculada en términos de la norma L_∞ como:

$$\left\| G - \hat{G} \right\|_{L_\infty} = \sup_{\omega \in \mathbb{R}} \left\| G(\jmath\omega) - \hat{G}(\jmath\omega) \right\|, \tag{2.8}$$

donde G y $\hat{G}$ son las funciones de transferencia de los sistemas original y reducido, $\jmath = \sqrt{-1}$, y $\|\cdot\|$ es la norma espectral.

Existen diversos métodos de reducción de modelos. Aunque muchos de estos métodos proporcionan buenos resultados y son extremadamente populares para sistemas pequeños, requieren la solución de un problema de valores propios de orden n por lo que sus implementaciones estándar son poco prácticas para reducir sistemas grandes (por ejemplo con $n > 1000$) [67].

Una parte importante de los algoritmos de reducción de modelos se basan, de algún modo, en obtener una realización balanceada del sistema. Se entiende por realización una descripción en el espacio de estados (A, B, C, D) de cierta función de transferencia $G(s)$. Una misma función de transferencia puede tener distintas realizaciones, pero existe una realización de orden mínimo (realización mínima) que es única salvo por una transformación de similaridad. Es decir, si (A, B, C, D) y $(\tilde{A}, \tilde{B}, \tilde{C}, \tilde{D})$ son realizaciones mínimas de una función de transferencia $G(s)$, entonces existe una única matriz T no singular tal que $\tilde{A} = T^{-1}AT$,$\tilde{B} = T^{-1}B$,$\tilde{C} = CT$,$\tilde{D} = D$.

[1] En ocasiones se utilizan los términos "reducción de modelos" para nombrar la técnica.

Las soluciones X_B y X_C del par de ecuaciones algebraicas de Lyapunov (ALE por sus siglas en inglés)

$$\begin{aligned} AX_B + X_B A^T &= -BB^T, \\ A^T X_C + X_C A &= -C^T C. \end{aligned} \tag{2.9}$$

son llamadas gramianos de controlabilidad y observabilidad respectivamente. Una realización mínima es además balanceada si

$$X_B = X_C = diag(\sigma_1, ..., \sigma_n), \tag{2.10}$$

donde $\sigma_1 \geq \sigma_2 \geq, ..., \geq \sigma_n$ son los valores singulares de Hankel.

Sin entrar en detalles, estos valores están relacionados estrechamente con la energía de cada estado. Debido a esto, en una realización balanceada los estados pueden ordenarse de acuerdo a su energía y, por lo tanto, a su contribución a las propiedades de entrada/salida del sistema. A los efectos de reducir un modelo, una vez obtenido este orden pueden descartarse aquellos estados cuya contribución sea menor [32].

2.1. Truncamiento balanceado y método de la raíz cuadrada

A continuación se describe brevemente el método de Truncamiento Balanceado, el cual es uno de los más utilizados, debido a que asegura importantes propiedades del sistema de orden reducido, y del cual se derivan otros métodos de interés en este trabajo.

El método de Truncamiento Balanceado (BT) [60] pertenece a la categoría de los métodos de *proyección del espacio de estados*. Esto significa que el sistema de orden reducido es obtenido a través de la proyección del espacio de estados del sistema original de dimensión n en un subespacio de dimensión $r \ll n$. El sistema de orden reducido viene dado entonces por

$$\left[\begin{array}{c|c} \hat{A}_r & \hat{B}_r \\ \hline \hat{C}_r & \hat{D}_r \end{array}\right] = \left[\begin{array}{c|c} S_C^T A S_B & S_C^T B \\ \hline C S_B & D \end{array}\right], \tag{2.11}$$

donde $S_C^T S_B = I_r$. La diferencia entre los métodos pertenecientes a esta categoría radica en la forma en que se construyen S_C y S_B. Dada una descripción del sistema en el espacio de estados $\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]$ mínima (controlable y observable) y estable (con matriz A *Hurwitz*), el método de truncamiento balanceado obtiene $S_B = T_{1..r}$ y $S_C = T_{1..r}^{-T}$ donde mediante el subíndice $1..r$ se denotan las primeras r filas de una matriz T no singular que convierte esta realización en una realización mínima, estable y balanceada. De esta forma se obtiene un sistema reducido de orden r.

Una de las principales ventajas del método BT es que proporciona una cota para la diferencia entre los sistemas, original y reducido, en términos de la norma L_∞. Como la realización producida por el método BT es balanceada sus gramianos son diagonales e iguales entre sí:

$$W_o = W_c = \tilde{\Sigma} = \left[\begin{array}{ccc} \sigma_1 & & \\ & \ddots & \\ & & \sigma_r \end{array}\right].$$

Como se mencionó anteriormente, los valores $\sigma_1, ..., \sigma_r$ son los *valores singulares de Hankel*. Si $G(s) = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]$ y $G_r(s) = \left[\begin{array}{c|c} \hat{A}_r & \hat{B}_r \\ \hline \hat{C}_r & \hat{D}_r \end{array}\right]$, la mencionada cota viene dada por

$$\|G - G_r\|_\infty \leq 2 \sum_{k=r+1}^{n} \sigma_k. \tag{2.12}$$

Es posible demostrar que, para un sistema mínimo, la matriz

$$T = \Sigma^{\frac{1}{2}} U^T R^{-T}, \tag{2.13}$$

donde $W_c = R^T R$ y $RW_o R^T = U\Sigma^2 U^T$ (descomposición en valores singulares), proveen una transformación del espacio de estados balanceada. Sin embargo, la matriz T computada en (2.13) tiende a ser altamente mal condicionada [67], lo cual podría provocar serios errores numéricos al balancear el sistema. Además, esta forma de calcular T restringe el método a realizaciones mínimas.

Para superar estos problemas suele utilizarse un método conocido como *método de la raíz cuadrada* [81] (SRBT por sus siglas en inglés *square root balanced truncation*). El primer paso es obtener una factorización de Cholesky de ambos gramianos, computando la solución del par de ecuaciones de Lyapunov en (2.9).

Una vez obtenidos $W_o = S^T S$ y $W_c = R^T R$, se tiene que

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

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

$$SR^T = \begin{bmatrix} U_1 & U_2 \end{bmatrix} \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix}, \qquad \Sigma_1 = \operatorname{diag}(\sigma_1, \ldots, \sigma_r). \tag{2.14}$$

Los proyectores S_B y S_C son entonces calculados según

$$S_B = R^T V_1 \Sigma_1^{-1/2}, \quad S_C = \Sigma_1^{-1/2} U_1^T S, \tag{2.15}$$

mientras que el modelo de orden reducido viene dado por la realización en (2.11).

Una de las etapas más importantes del algoritmo descrito anteriormente es la obtención de los factores de Cholesky de los gramianos del sistema. Si bien en [81] se utiliza el método de Hammarling [48] para computar los factores de los gramianos, existen varios métodos que pueden ser utilizados indistintamente. Por ejemplo, para la reducción de modelos densos de gran dimensión mediante arquitecturas paralelas, suelen ser atractivos los enfoques basados en la función signo [23]. Sin embargo, si bien pueden ser adecuados para casos densos, métodos directos como el de Hammarling o Bartels-Stewart [9], o iterativos como los basados en la función signo, no tienen un buen desempeño cuando la matriz A es dispersa, ya que no pueden sacar partido de propiedades estructurales particulares de dicha matriz.

Para los casos donde la matriz de transición es grande y dispersa, en [65] se propone un algoritmo iterativo basado en el método *Alternating Direction Implicit* (ADI) para resolver ecuaciones en derivadas parciales parabólicas, hiperbólicas y elípticas. Este método ha sido ampliamente difundido y utilizado en problemas de reducción de modelos dispersos, y actualmente es parte fundamental de Lyapack [64], una biblioteca para resolver ecuaciones de Lyapunov y Riccatti de gran tamaño, así como problemas de reducción de modelos dispersos en MATLAB. En la siguiente sección se describe brevemente dicho algoritmo.

Obtención de los factores de Cholesky de los gramianos - algoritmo LRCF-ADI

La resolución numérica de las ecuaciones algebraicas de Lyapunov presentadas en (2.9) no es un problema exclusivamente referido a reducción de modelos. Dichas ecuaciones están implicadas en áreas tales como el análisis de estabilidad, control óptimo, solución de ecuaciones de Riccatti y obtención de realizaciones balanceadas [6].

Si el sistema es estable, es decir, la matriz A tiene todos sus valores propios en el semiplano complejo izquierdo (matriz Hurwitz), existe una solución $X \in \mathbb{R}^{n\times n}$ que además es única, simétrica y semi-definida positiva. Más aún, si el par (A, B) es controlable, dicha solución es definida positiva.

Cuando estas ecuaciones de Lyapunov están relacionadas con problemas de reducción de modelos, es común que la matriz A del sistema sea dispersa y estructurada, y que la matriz B contenga relativamente pocas columnas. En este caso, de acuerdo con lo expuesto en [66], los valores propios de la solución X decaen rápidamente, por lo que puede aproximarse de forma muy precisa por una matriz semi-definida positiva de rango bajo.

Los métodos de Bartels-Stewart y Hammarling son los principales métodos directos para resolver ecuaciones algebraicas de Lyapunov. El primero es más general, ya que es aplicable a la ecuación de Sylvester[2], mientras que el segundo proporciona resultados más precisos en presencia de errores de redondeo. Ambos métodos requieren computar la forma de Schur de la matriz A y, como consecuencia, no pueden beneficiarse de propiedades estructurales de dicha matriz. El método de Smith [77] y los basados en la función signo [70] son métodos iterativos, pero tampoco pueden beneficiarse de estas propiedades. Sin embargo, como se comentó anteriormente, estos últimos son interesantes a la hora de ser implementados en arquitecturas paralelas [67]. El método ADI es un método iterativo que a menudo proporciona buenos resultados para ALEs dispersas o estructuradas. Su complejidad está directamente relacionada con la estructura de la matriz A y muchas veces es menor que la de los métodos anteriores.

En cuanto al consumo de memoria, todos los métodos anteriores presentan un costo computacional proporcional $O(n^2)$ debido a que deben almacenar explícitamente la solución X densa. Muchas veces esto resulta imposible cuando el orden del sistema original es de dimensión importante.

La única alternativa frente a este problema son los métodos de rango bajo. Para los casos en que la matriz A es dispersa o estructurada y la matriz B tiene pocas columnas, este tipo de métodos son capaces de resolver ALEs de gran tamaño computando una factorización de rango bajo que aproxima la matriz solución X, por lo que no es necesario almacenar dicha solución de forma completa.

El método ADI para las ecuaciones de Lyapunov en (2.9) se describe de forma resumida en el Algoritmo 1.

Algoritmo 1 Método ADI

Input: $A, B, X, p_1, p_2, ..., p_{i_{max}}$
Output: $X = X_{i_{max}}$

1: **for** $i = 1, 2, 3, ..., i_{max}$ **do**
2: $\quad (A + p_i I_n)X_{i-1/2} = -BB^T - Xi - 1(A^T - p_i I_n)$
3: $\quad (A + p_i I_n)X_i^T = -BB^T - X_{i-1/2}^T(A^T - p_i I_n)$
4: **end for**

Este procedimiento genera una sucesión de matrices X_i, con $i = 1, 2, ...$, que converge rápidamente a la solución X de la Ecuación (2.9) siempre y cuando los parámetros p_i del algoritmo sean escogidos apropiadamente. Considerando la discusión anterior, puede lograrse una implementación eficiente de este algoritmo reemplazando el iterador por su descomposición de Cholesky, esto es $X_i = Z_i Z_i^H$, y reformulando el método en base a los factores Z_i. En general, estos factores también poseen bajo rango, aunque es necesario hacer notar que la aproximación de rango bajo $X \approx Z_{i_{max}} Z_{i_{max}}^H$ no es única.

[2]Ecuaciones de la forma: $AX + XB = C$.

Existen distintas formas de computar estos factores de bajo rango. Una de las más eficientes es el algoritmo llamado *Low Rank Cholesky Factor ADI* (LRCF-ADI) introducido en [65]. En el Algoritmo 2 se puede encontrar una extensión del Algoritmo 1 utilizando los factores de bajo rango que implementa dicho método. Notar que no es necesario definir a priori el parámetro i_{max}, sino que pueden aplicarse distintos criterios de parada.

Algoritmo 2 Método LRCF-ADI

Input: $A, B, p_1, p_2, ..., p_{i_{max}}$
Output: $Z = Z_{i_{max}} \in \mathbb{C}^{n,t_{i_{max}}}$ tal que $ZZ^H \approx X$

1: $V_1 = \sqrt{-2\Re\{p_1\}}(A + p_1 I_n)^{-1} B$
2: $Z_1 = V_1$
3: **for** $i = 1, 2, 3, ..., i_{max}$ **do**
4: $\quad V_i = \sqrt{\Re\{p_i\}/\Re\{p_{i-1}\}}(V_{i-1} - (p_i + \bar{p}_{i-1})(A + p_i I_n)^{-1} V_{i-1})$
5: $\quad Z_i = Z_{i-1} V_i$
6: **end for**

La implementación del Algoritmo 2 de [65] requiere que el conjunto de parámetros $\mathcal{P} = p_1, ..., p_{i_{max}}$ esté compuesto por números reales negativos o pares de complejos conjugados con parte real negativa. Esto se debe a que si $X_i = Z_i Z_i^H$ es generada por un conjunto de parámetros como el anterior, X_i es real; de lo contrario, Z_i no lo es. Si bien en [18] se propone un algoritmo más sofisticado para computar la solución de las ecuaciones de Lyapunov, la implementación tomada como referencia (presentada en [65]), en caso de que el factor $Z = Z_{i_{max}}$ sea complejo, computa un factor $\hat{Z}$ real a partir del anterior, tal que $ZZ^H = \hat{Z}\hat{Z}^T$.

La determinación del conjunto de parámetros óptimos (posiblemente subóptimos) para el método ADI está relacionada con el problema de optimización descrito en (2.16), donde $\sigma(A)$ denota el espectro de la matriz A [78].

$$\min_{\mathcal{P}} \left\{ \max_{t \in \sigma(A)} \frac{(t - p_1)...(t - p_l)}{(t + p_1)...(t + p_l)} \right\} \tag{2.16}$$

En general, $\sigma(A)$ es desconocido y su cómputo es costoso cuando la matriz A es de gran dimensión. Además, aún conociendo el espectro o cotas para el mismo, no se dispone de un algoritmo para calcular el conjunto óptimo de parámetros $\mathcal{P}$ en tiempo polinomial. El algoritmo utilizado por la implementación de referencia para computar un conjunto de parámetros subóptimos de desplazamiento para el método ADI es de carácter heurístico.

En primer lugar, este algoritmo aproxima el espectro por medio de un par de procesos de Arnoldi como el que se describe en el Algoritmo 3. El primero de este par de procesos es aplicado sobre la matriz A y devuelve un conjunto de valores que aproximan los valores propios que se encuentran, por lo general, lejos del origen. El segundo de los procesos se aplica sobre la matriz A^{-1} y es utilizado para obtener una aproximación de valores propios cercanos al origen, los cuales son de suma importancia para el problema de optimización descrito en (2.16). Las aproximaciones de los valores propios de A producidas por la iteración de Arnoldi son comúnmente llamadas valores de Ritz.

En segundo lugar, se escoge un subconjunto de estos valores de Ritz mediante una heurística que proporciona una solución subóptima del problema en (2.16). El orden en el cual esta heurística devuelve el conjunto de parámetros es tal que los parámetros que se relacionan con una reducción mayor del error en el método ADI son aplicados primero. Una explicación más detallada sobre este algoritmo puede encontrarse en [65].

Algoritmo 3 Ortogonalización de Arnoldi

Input: vector v_1 de norma 1.
Output: matriz H Hessemberg cuyos valores propios son los valores de Ritz.

1: **for** $k = 1, 2, \ldots$ **do**
2: $q_k = Aq_{k-1}$
3: **for** $j = 1, \ldots, k-1$ **do**
4: $h_{j,k-1} = q_j * q_k$
5: $q_k = q_k - h_{j,k-1} * q_j$
6: **end for**
7: $h_{k,k-1} = q_j * q_k$
8: $q_k = q_k / h_{k,k-1}$
9: **end for**

Es importante notar que en los Algoritmos 2 y 3 no es necesario almacenar ni calcular explícitamente las matrices A^{-1} y $(A - p_i)^{-1}$. Además, esto sería inviable dado que la inversa de una matriz dispersa es, por lo general, una matriz densa, por lo cual sería imposible almacenarla en el caso de que se tratase de una matriz de dimensión importante. Por consiguiente, el paso 3 del Algoritmo 2 involucra la resolución de un sistema de ecuaciones lineales de la forma $(A+p_iI)X = V$, donde V es una matriz rectangular de pocas columnas (igual a la cantidad de columnas de la matriz B), mientras que el paso 5 del Algoritmo 3, cuando se aplica a A^{-1}, debe resolver $Ax = v$, donde x y v son vectores. La resolución de estos sistemas de ecuaciones se realiza en cada paso de ambas iteraciones, por lo cual se convierte en el principal cuello de botella, desde el punto de vista computacional, de estos algoritmos.

2.2. Trabajo relacionado

En esta sección se presenta un relevamiento sobre la aplicación de técnicas de HPC al problema de reducción de modelos. En segundo lugar, se describe el estado del arte en la resolución de problemas de reducción de modelos y control utilizando procesadores gráficos. Por último, se describen brevemente dos bibliotecas para problemas de control y reducción de modelos dispersos ampliamente aceptadas y utilizadas por la comunidad.

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

Hasta fines de la década de los 90, la resolución computacional de problemas de control estuvo basada en distintos paquetes de software para diseño de sistemas de control asistido por computadora (CACSD por sus siglas en inglés). Sin embargo, estos enfoques se vieron limitados por la ausencia de una biblioteca robusta y moderna, que proporcionara una implementación eficiente y algoritmos numéricamente estables para las operaciones básicas realizadas por estos sistemas. Algunos esfuerzos relativos al desarrollo de bibliotecas para la resolución de problemas de control se recogen en el trabajo de Benner et al. [20].

Atendiendo esta necesidad, en el marco del proyecto NICONET (*Numerics in Control Network*) [20], se desarrolló la biblioteca SLICOT (*Subroutine Library in Control Theory*)[3] [84]. SLICOT es una herramienta, implementada en Fortran 77, que provee operaciones básicas frecuente-

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

mente usadas en Teoría de Control, con la intención de ofrecer una base sobre la cual construir nuevos paquetes de software para el diseño de sistemas asistido por computadora. Con el objetivo de ser numéricamente robusta y eficiente desde el punto de vista computacional, SLICOT basa su diseño en la máxima utilización posible de las especificaciones LAPACK [5] y BLAS [34] (ver Anexo B). De esta manera, una implementación eficiente multi-hilo de SLICOT en arquitecturas de memoria compartida se desprende de la simple utilización de una implementación multi-hilo de BLAS. Con el objetivo de mejorar su usabilidad y facilitar la experimentación con la herramienta, SLICOT provee una interfaz que permite su uso desde MATLAB, combinando la flexibilidad y amplia aceptación de este software con el mayor desempeño ofrecido por la implementación en FORTRAN de las rutinas. SLICOT es fuertemente utilizada por investigadores del área en la actualidad.

Muchos esfuerzos fueron dedicados posteriormente a la incorporación de paralelismo para arquitecturas de memoria distribuida a la biblioteca SLICOT. Por ejemplo, el trabajo de Blanquer et al. [25] presenta propuestas relacionadas con paralelizar la resolución de la ecuación de Lyapunov en SLICOT, sugiriendo la creación de la herramienta PSLICOT (*Parallel* SLICOT). El resultado de estos esfuerzos se vio finalmente plasmado en el desarrollo de la biblioteca PLiC (*Parallel Library in Control*) [22].

PLiC incluye rutinas fundamentales para el análisis y diseño de sistemas dinámicos lineales, invariantes en el tiempo y de gran escala, sobre arquitecturas paralelas distribuidas. La herramienta está formada por dos componentes fundamentales, PLiCOC, que brinda rutinas relacionadas con la resolución de diversos problemas en teoría de control, y PLiCMR, especialmente diseñada para problemas de reducción de modelos. El diseño para soportar el paradigma de paralelismo de memoria distribuida se basa fuertemente en la la biblioteca SCALAPACK [24] (también descrita en el Anexo B). De hecho, las operaciones de álgebra se resuelven invocando a rutinas de SCALAPACK siempre que es posible, efectuando las comunicaciones por medio de la biblioteca BLACS [35].

Algunos esfuerzos han sido realizados con el objetivo de facilitar el uso de la biblioteca PLiC. Por ejemplo en [19] se describe una interfaz web que permite resolver problemas de forma remota sin la necesidad de contar con plataformas de HPC locales, mientras que en [39] se presenta una interfaz escrita en Python para la biblioteca PLiCMR.

En el campo de reducción de modelos dispersos, es importante destacar los trabajos de Remón et al. [69] sobre el método LR-ADI para matrices banda. En dicho trabajo se presentan implementaciones paralelas sobre plataformas de memoria compartida para la factorización LU de matrices de banda, mostrando cómo es posible utilizar las rutinas desarrolladas para acelerar el método LR-ADI para este tipo de matrices, obteniendo factores de mejora superiores al 50 %.

En forma más reciente, en [37], los autores estudian la aceleración del método BT sobre matrices simétricas definidas positivas sobre arquitecturas multinúcleo.

2.2.2. Reducción de modelos y problemas de control con GPUs

Los métodos de reducción de modelos suelen involucrar de forma muy importante operaciones básicas del álgebra lineal numérica (ALN). Debido a los importantes resultados obtenidos en los últimos años en la aceleración de operaciones de ALN densa utilizando procesadores gráficos, no es sorprendente que la aceleración de métodos de reducción de modelos densos en arquitecturas de HPC, en particular procesadores gráficos, concentren aún gran parte del esfuerzo de la comunidad. Sin embargo, no existen actualmente propuestas (más allá de las incluidas en el presente trabajo) para resolver problemas de reducción de modelos dispersos en GPU, y no se han encontrado trabajos que discutan este tema durante la revisión del estado del arte.

En lo referente a problemas densos, uno de los primeros esfuerzos en el área de reducción de

modelos utilizando procesadores gráficos es el de Benner et al. [16], donde los autores implementan la función signo con aritmética de precisión simple (la mayor precisión disponible en ese momento) utilizando una GPU NVIDIA C1060. Este trabajo fue extendido posteriormente en [13] para resolver ecuaciones de Lyapunov utilizando técnicas de precisión mixta para obtener resultados de calidad (respecto a errores numéricos) similar a la obtenida al utilizar aritmética de doble precisión.

En cuanto a los métodos de reducción de modelos, los autores de [12] implementaron el método BT en una arquitectura híbrida compuesta por una GPU y una CPU. En [17] los autores implementan el método *Balanced Stochastic Truncation* (BST) incluyendo la resolución de una ecuación algebraica de Riccati (ARE por sus siglas en inglés). Por otro lado, un resolutor de ecuaciones diferenciales de Riccati en GPU es presentado en [14] y posteriormente extendido en [15] para utilizar plataformas con más de una GPU. El uso de múltiples GPU permite a los autores alcanzar factores de aceleración de hasta un factor $\times 20$ en relación con implementaciones para procesadores multi-core, además de posibilitar la resolución de instancias con dimensión (número de estados) de hasta 34K.

Por último, en [68], se presentan implementaciones de algunos métodos para la resolución de la ecuación de Lyapunov en GPU, en particular, para el método de Bartels-Stewart, el método de Schur, el método ADI y la función signo. Los autores comparan sus resultados con rutinas análogas codificadas en MATLAB y con la función `lyap` provista por esta herramienta, logrando aceleraciones de hasta $\times 4$ para la solución de la ecuación de Lyapunov. Sin embargo, la evaluación de los métodos se realiza para problemas de pequeña a mediana dimensión ($n < 4000$) debido a las restricciones de memoria impuestas por el acelerador.

2.2.3. Biblioteca Lyapack

Lyapack es un paquete MATLAB (*toolbox*) que ofrece una importante variedad de rutinas para la solución numérica de ecuaciones de Lyapunov y Riccati, así como problemas de reducción de modelos y control óptimo. Es particularmente apropiado para problemas dispersos de gran dimensión y, en especial, para sistemas dinámicos donde la matriz principal es estructurada. Más allá de sus funcionalidades, Lyapack se distingue por presentar una interfaz amigable para el usuario y una arquitectura modular flexible, la cual facilita su extensión y mantenimiento.

Lyapack implementa un resolutor eficiente para las ecuaciones de Lyapunov basado en el método LRCF-ADI presentado en el Algoritmo 2, el cual es parte fundamental para el resto de los problemas abordados por el paquete. Específicamente, el resolutor de ecuaciones de Riccati y los métodos de reducción de modelos y control óptimo son construidos sobre el resolutor LRCF-ADI. En consecuencia, el desempeño de los métodos incluidos en Lyapack está fuertemente condicionado por el desempeño del resolutor de ecuaciones de Lyapunov.

Desde el punto de vista de la implementación, las rutinas principales de Lyapack realizan las operaciones más costosas computacionalmente en términos de tres operaciones matriciales básicas (BMOs por sus siglas en inglés):

- Producto matricial ($Z := AY$).
- Solución de sistemas de ecuaciones lineales ($Z := A^{-1}Y$).
- Solución de sistemas de ecuaciones lineales desplazados ($Z := (A + pI_n)^{-1}Y$).

En todos los casos $A \in \mathbb{R}^{m\times n}$ es la matriz (dispersa) de coeficientes de la ecuación de Lyapunov, mientras que $Y \in \mathbb{R}^{n\times c}$ es una matriz densa con $c \ll n$ y p es un escalar.

La estrategia de Lyapack para poder tratar matrices de coeficientes con distintas estructuras está basada en una *reverse communication interface*, también utilizada por herramientas como

ARPACK [57] y PETSc [8]. De esta manera, es responsabilidad del usuario proveer un kernel eficiente para el cómputo de cada BMO, que sea capaz de aprovechar las propiedades estructurales de dichas matrices en una arquitectura objetivo específica. Estas funciones son llamadas *user-supplied functions* o USFs.

En resumen, el uso de USFs se traduce en un diseño inspirado en el concepto de polimorfismo que se maneja en programación orientada a objetos: las distintas matrices de coeficientes A son almacenadas en estructuras de datos ocultas, y el propósito de las USFs es manipular estas estructuras, creándolas, destruyéndolas y calculando las correspondientes BMOs según el tipo de matriz. Consecuentemente, el usuario es capaz de definir rutinas específicas, ajustadas para explotar las características particulares del problema y de la plataforma objetivo, sin necesidad de manipular los métodos numéricos implementados en las rutinas principales de Lyapack.

M.E.S.S

En los últimos años se ha avanzado significativamente en el desarrollo de la biblioteca M.E.S.S. (*Matrix Equations Sparse Solvers*)[4], una evolución de Lyapack que incorpora mejoras en algunos métodos así como diversos cambios en su arquitectura. Adicionalmente, se encuentra en desarrollo una versión implementada en C de la biblioteca, orientada a resolver problemas de gran escala de forma eficiente. También se han hecho esfuerzos enfocados a incluir el uso de GPUs en la biblioteca, sin embargo estos esfuerzos no han superado el estado de prototipo.

2.3. Resumen

En este capítulo se presentaron algunos conceptos sobre el problema de reducción del orden de modelos. Respecto a su solución computacional, se mencionan algunos métodos y se profundiza sobre el método de Truncamiento Balanceado y método de la Raíz Cuadrada. Dichos métodos requieren la solución de un par de ecuaciones matriciales de Lyapunov. Cuando la matriz de estados del sistema dinámico es grande y dispersa, estas ecuaciones pueden ser resueltas mediante el método LRCF-ADI, un algoritmo iterativo que tiene como paso fundamental la solución de sistemas de ecuaciones lineales dispersos que tienen, como matriz de coeficientes, la matriz de estados del sistema desplazada por ciertos parámetros calculados previamente mediante una heurística.

En la sección de trabajo relacionado, se realiza una reseña sobre la aplicación de técnicas de HPC y, en particular, de procesadores gráficos en el problema de reducción de modelos. Por último se realizó una breve presentación de Lyapack, así como también de su sucesora M.E.S.S, dos bibliotecas para problemas de control y reducción de modelos dispersos utilizadas por la comunidad.

[4]Disponible en http://svncsc.mpi-magdeburg.mpg.de/trac/messtrac

Capítulo 3

Aceleración de Lyapack en arquitecturas híbridas

El objetivo principal de este trabajo es producir un entorno para la resolución de problemas de reducción de modelos que aproveche los beneficios brindados por las arquitecturas computacionales híbridas, específicamente aquellas conformadas por unidades de procesamiento gráfico y CPUs multi-core. Como se ha descrito anteriormente, Lyapack es una poderosa biblioteca para reducción de modelos caracterizada, además, por su diseño sencillo y flexible, así como por presentar una buena usabilidad. Es interesante, entonces, extender Lyapack con el fin de invocar código que pueda explotar las mencionadas arquitecturas sin realizar grandes cambios en lo que refiere a la forma de uso.

Precisamente, la filosofía de diseño subyacente en Lyapack brinda la posibilidad de alcanzar este objetivo por medio de la implementación en CUDA de las USFs apropiadas. Con este fin se utiliza la *Mex API* de MATLAB, la cual provee interoperabilidad entre MATLAB y otros lenguajes de programación, como por ejemplo C, Fortran o, en nuestro caso, CUDA. Es importante notar que esta aproximación también permite invocar códigos híbridos que ejecuten distintas tareas en GPU y CPU de forma concurrente.

3.1. Diseño de la extensión de Lyapack con GPU

La arquitectura software de la propuesta se encuentra organizada en tres componentes, los cuales se describen a continuación y se ilustran en la Figura 3.1.

`Biblioteca GPU:` Este componente se encarga, principalmente, de mantener referencias a bibliotecas comúnmente utilizadas por las rutinas de álgebra lineal aceleradas con GPU, como CUBLAS y CUSPARSE. Dichas bibliotecas requieren una inicialización y una finalización, ya que reservan memoria tanto en el dispositivo como en la CPU, y crean un manejador a través del cual es posible acceder indirectamente a las estructuras de memoria que forman el *contexto* de la biblioteca. Es por esto que esta biblioteca proporciona funciones que deben invocarse durante la etapa de inicialización y finalización del framework para garantizar un uso correcto de la memoria, compatible con el manejo de memoria que realiza MATLAB. Además, esta biblioteca implementa funciones útiles de propósito general que pueden ser invocadas por los demás módulos del framework.

`Interfaz:` Esta capa utiliza la Mex API de MATLAB para proveer interoperabilidad entre dicha herramienta y las funciones implementadas en otros lenguajes de programación. Está formada

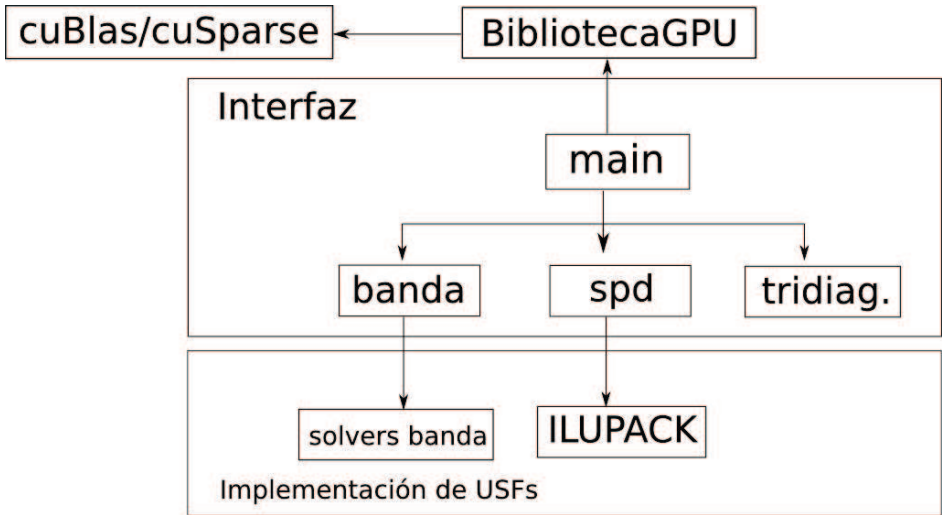

Figura 3.1: Diseño de la extensión de Lyapack para utilizar GPUs.

básicamente por dos componentes principales, un módulo principal que se encarga de recibir cualquier comunicación por parte de MATLAB, así como también de la inicialización y finalización del framework, y un módulo específico por cada conjunto de USFs implementado. El módulo principal provee una interfaz mínima que debe ser implementada por cualquier módulo que desee extender la herramienta para algún tipo específico de USFs. En particular, debe implementarse una función de limpieza que libere correctamente la memoria, tanto en el dispositivo como en la CPU, en los casos antes mencionados, y una operación que procese las llamadas a funciones efectuadas desde MATLAB.

El resto de los módulos reciben los parámetros de la llamada a función que se origina en MATLAB, transforman los datos recibidos del formato manejado por MATLAB al formato adecuado, transfieren los datos desde la memoria de la CPU a la de la GPU (de ser necesario) e invocan los kernels específicos que correspondan. Luego de completada la operación, se transfieren los datos del resultado a la memoria de la CPU (de ser necesario), se transforma el resultado a los formatos manejados por MATLAB y se retorna el control (y el resultado) a la capa superior de Lyapack.

`Implementación de USFs y bibliotecas específicas:` Este componente implementa los kernels necesarios para realizar las USFs correspondientes a cada tipo de matriz. En este sentido, deben proporcionarse funciones que implementen el producto matricial y la resolución de sistemas lineales (posiblemente desplazados) ya sea en la CPU, en GPU o utilizando ambos dispositivos. El diseño del framework permite que estas funciones sean implementadas por bibliotecas de terceros de forma sencilla, ya que cada grupo de USFs se vincula por separado, por lo que pueden utilizarse distintas bibliotecas para distintos grupos de USFs. Por ejemplo, se pueden tener grupos de USFs para el mismo tipo de matriz que resuelvan los sistemas lineales utilizando distintas implementaciones de BLAS y LAPACK. Esta flexibilidad no agrega demasiada complejidad a la extensión de la herramienta.

Como se mencionó anteriormente, para lograr invocar programas implementados en C y CUDA se utilizó la interfaz MEX de MATLAB. Esta interfaz consiste en una API que brinda interoperabilidad entre MATLAB y otros lenguajes de programación. Específicamente, para llamar a una función

C/CUDA desde MATLAB, se debe primero generar un archivo Mex con el nombre de la función, ya que MATLAB busca la implementación de las funciones invocadas según los nombres de los archivos que se encuentran en la variable interna `PATH`. Este archivo es en realidad un archivo fuente C/CUDA, compilado de cierta manera, que incluye la Mex API y contiene una función llamada "Mex Function". La Mex Function es el único punto de acceso al código que se encuentra en el archivo fuente. Es su responsabilidad proveer la implementación de la función que será llamada desde MATLAB, o llamar a una función que la implemente. Hay exactamente una Mex Function por cada archivo binario Mex. Dicha función se encuentra codificada en el módulo principal de la interfaz, componente denotado como "main" en la Figura 3.1.

Como el nombre del archivo Mex debe ser el nombre de la función que se desea llamar desde Matlab, inicialmente debería haber un archivo por cada función. Esto significa que compartir datos entre distintas funciones no es trivial. Sin embargo, si es necesario, existen varias formas de hacerlo. Una de ellas consiste en tener un único archivo Mex que reciba todas las llamadas. Para poder llamar a un mismo archivo con distintos nombres de función, es necesario crear enlaces simbólicos al archivo principal con dichos nombres. Una vez que la Mex Function es ejecutada, es necesario verificar el nombre con el cual la función fue invocada mediante una función provista por la Mex API (*mexFunctionName()*). Luego se decide qué código es ejecutado mediante un bloque condicional, como por ejemplo *if/then*.

Para cada tipo de matriz abordado, se genera un archivo binario Mex diferente, que contiene una Mex Function y la implementación de las USFs para dicho tipo de matriz. Los enlaces simbólicos se encargan de que la llamada a función, que se origina en la consola de MATLAB, sea derivada al archivo binario Mex correcto. Tener un archivo binario distinto para cada tipo de matriz brinda flexibilidad en cuanto al uso de distintas bibliotecas para implementar distintos tipos de USFs; por ejemplo, permite utilizar distintas versiones de LAPACK y BLAS para distintos ejecutables. La mecánica de este sistema se esquematiza en la Figura 3.2.

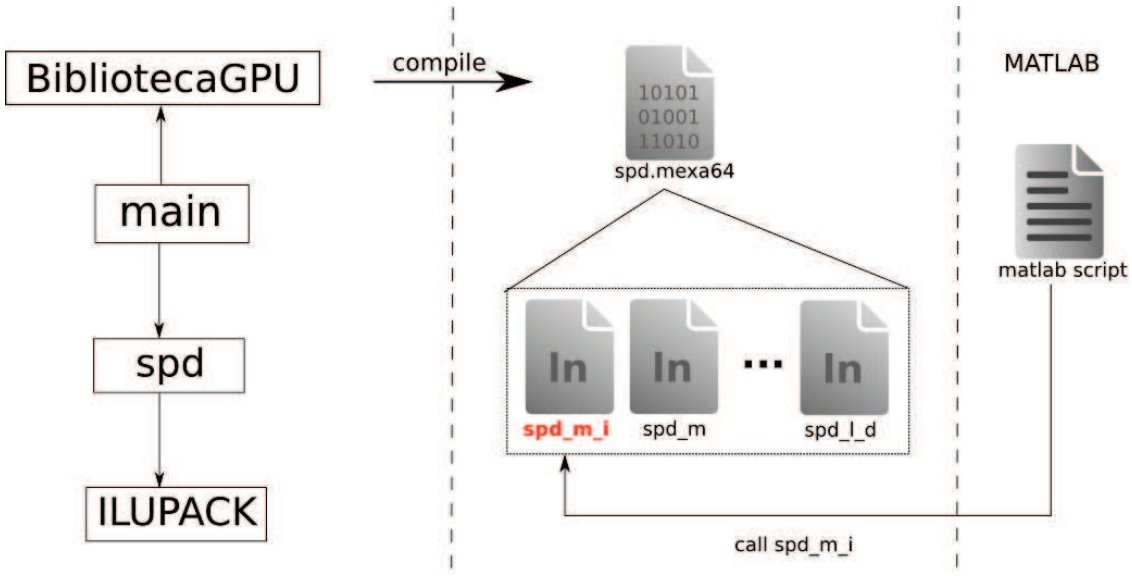

Figura 3.2: Esquema de funcionamiento de la extensión de Lyapack con la interfaz Mex de MATLAB. El ejemplo ilustra el caso de las USFs correspondientes al caso simétrico y definido positivo (SPD) pero es análogo para los demás casos.

En la versión original de la biblioteca Lyapack, las USFs están separadas en archivos que

comparten un prefijo o nombre base que identifica el conjunto de USFs para un determinado tipo de matriz (por ejemplo “as” para matrices simétricas, “au” para no simétricas, etc.).

Al nombre básico de las rutinas se le agregan algunas extensiones según el siguiente criterio:

[nombre base] [ext_1] or [nombre base] [ext_1] [ext_2]

Cinco extensiones distintas son posibles. Su significado es el siguiente:

- ext_1 = m: multiplicación de matrices.
- ext_1 = l: solución de sistemas de ecuaciones lineales.
- ext_1 = s: solución de sistemas de ecuaciones lineales desplazados.
- ext_1 = pre: preprocesamiento.
- ext_1 = pst: postprocesamiento.

Para algunos casos de USFs no existen rutinas de pre y postprocesamiento ya que no son necesarias. Si [ext_1] = pre o pst, no hay segunda extensión. Si [ext_1] = m, l, o s, las posibles extensiones son:

- ext_2 = i: inicialización de los datos requeridos por las BMOs correspondientes.
- no ext_2: la rutina efectivamente realiza la BMO.
- ext_2 = d: destrucción de los datos generados por la rutina de inicialización ([ext_2] = i).

Pueden relacionarse estas ideas con conceptos de programación orientada a objetos, como constructores y destructores.

Es importante notar que las USFs con [ext_1] = pre o pst serán llamadas únicamente desde el programa principal (la rutina escrita por el usuario) y que las USFs con [ext_2] = i serán llamadas frecuentemente desde el programa principal, aunque no siempre. Por el contrario, los tres tipos restantes de USFs ([ext_1] = m, l, or s) son llamados internamente por las rutinas de Lyapack.

Ninguna de estas funciones mantiene dependencias de datos con funciones de distinto prefijo, mientras que estas dependencias sí ocurren con funciones del mismo prefijo, por ejemplo, la función pref_m, usualmente depende de la función pref_m_i. En escenarios típicos se suele trabajar con un tipo particular de matriz y no con varios de ellos, por lo cual no es interesante compartir datos entre funciones que pertenecen a distintos tipos de matriz.

Una característica que cabe destacar, es que la propuesta apunta a reducir el volumen de transferencias de datos entre los espacios de memoria de la CPU y la GPU por medio de la reutilización de los datos que ya se encuentran en la memoria de la GPU. Por ejemplo, dado que los sistemas lineales que se resuelven comparten la matriz de coeficientes, dicha matriz se transfiere una sola vez. Mas aún, al resolver los sistemas desplazados durante la iteración LRCF-ADI, dado que estos sistemas comparten la matriz de coeficientes a excepción de la diagonal principal, se aplican distintas estrategias, según el tipo de matriz, para transferir la matriz A una única vez y construir/resolver los sistemas desplazados directamente en GPU. Es importante notar que este nivel de optimización es difícil de alcanzar mediante herramientas que realizan rutinas básicas en GPU de forma automática, por ejemplo el paquete Jacket de AccelerEyes [53] o el Parallel Computing Toolbox [76] de MATLAB.

3.2. Implementación de USFs para distintas familias de matrices

Como se ha mencionado anteriormente, Lyapack delega en el usuario la responsabilidad de implementar las operaciones matriciales básicas (BMOs) correspondientes a cada estructura de matriz. Sin embargo, el paquete provee implementaciones de sus USFs para diversos grupos de matrices. Específicamente, cinco grupos de USFs son provistas:

1. Matrices simétricas, resolviendo los sistemas de ecuaciones lineales por medio de la factorización de Cholesky.

2. Matrices dispersas no simétricas, utilizando la descomposición LU para resolver los sistemas.

3. Matrices dispersas no simétricas, utilizando el método QMR (del inglés *Quasi Minimal Residual*) con precondicionador ILU0 para resolver los sistemas de ecuaciones lineales.

4. Matrices derivadas de sistemas dinámicos generalizados, en los que las matrices que describen el sistema son simétricas y definidas. En este caso los sistemas de ecuaciones lineales se resuelven mediante la factorización de Cholesky.

5. Matrices derivadas de sistemas dinámicos generalizados, en los que las matrices que describen el sistema son posiblemente no simétricas. Los sistemas de ecuaciones lineales se resuelven mediante la factorización LU.

Los grupos de USFs provistos por Lyapack intentan ser generales y, de hecho, abarcan una gran variedad de problemas. Con el objetivo de mantener esta generalidad en la versión de la biblioteca acelerada con GPUs, se han re-implementado los conjuntos de USFs para el caso de matrices dispersas generales, tanto simétricas como no simétricas. Sin embargo, la utilización de las GPUs para acelerar Lyapack está motivada fundamentalmente en la reducción de modelos de gran dimensión. En estos casos suele ser impráctica la resolución de los sistemas subyacentes mediante métodos directos como la factorización LU (o de Cholesky en el caso simétrico). Esto se debe a que, por un lado, cuando se practican dichas factorizaciones con una matriz dispersa, los factores experimentan un llenado[1], que en algunos casos puede convertirlos en matrices densas (si bien es común reordenar los coeficientes de la matriz para mitigar este efecto). En este caso los requerimientos de almacenamiento pueden aumentar dramáticamente conforme la dimensión del problema crece. Por otro lado, el costo computacional de resolver estos sistemas de gran dimensión mediante métodos directos es muy elevado, y a veces prohibitivo. Por estas razones, en la implementación de la biblioteca que incorpora cálculos con GPU, se optó por implementar USFs para matrices dispersas generales, tanto simétricas como no simétricas, pero utilizando métodos iterativos para resolver los sistemas de ecuaciones lineales. Para tratar sistemas simétricos y definidos positivos se ha implementado el método del *Gradiente Conjugado Precondicionado*, mientras que para sistemas no simétricos se optó por el método del *Gradiente BiConjugado Estabilizado* (BiCgStab), ambos de la familia de métodos basados en subespacios de Krylov.

Si bien estos métodos iterativos son una herramienta poderosa para resolver sistemas de ecuaciones lineales dispersos generales, existen casos en los que la matriz posee una estructura particular, por lo que frecuentemente existen enfoques más adecuados. Con la intención de aprovechar estas propiedades, se ha incluido una implementación en GPU del conjunto de USFs para matrices tri-diagonales y para matrices generales banda. En el caso de las matrices tri-diagonales se utiliza un método basado en el algoritmo de Reducción Cíclica [51] (CR, por sus siglas en inglés) para

[1]Problema conocido como *fill-in*. Para más información el lector puede referirse a [36, 42].

resolver los sistemas lineales subyacentes, mientras que en el caso banda se utiliza un nuevo método híbrido que utiliza CPU y GPU para resolver este tipo de sistemas, haciendo un uso eficiente de las operaciones provistas por BLAS (en este caso CUBLAS).

3.2.1. Implementación para matrices tri-diagonales

La operación con mayor costo computacional en la solución numérica de ecuaciones de Lyapunov con matriz de coeficientes tri-diagonal mediante el método LR-ADI es la solución de sistemas (desplazados) de ecuaciones lineales. Por lo tanto, es interesante acelerar la solución de estos sistemas computándolos en la GPU, por lo que se elaboró un *kernel* específico en GPU para la solución de sistemas tri-diagonales utilizando aritmética real.

La implementación está basada en la propuesta en [1] cuando todas las operaciones pueden ser realizadas en aritmética real, mientras que cuando se requiere trabajar con aritmética compleja, se utiliza la operación `tsgv` de la biblioteca CUSPARSE.

Por otra parte, la rutina original emplea el operador "\" de MATLAB (el cual, en este caso, se basa en el algoritmo de Thomas [44]). Dada su simplicidad, la multiplicación de una matriz tri-diagonal por un vector es computada en CPU tanto en la versión original de la biblioteca como en la versión acelerada con GPU.

El resolutor tri-diagonal emplea una estructura compacta formada por tres arreglos, uno para cada diagonal de elementos no nulos de la matriz. Este formato de almacenamiento de datos requiere una etapa de preprocesamiento para formar los sistemas desplazados de forma explícita. Esto es realizado de forma eficiente mediante un *kernel* implementado en CUDA, ya que es una operación altamente paralelizable y requiere un acceso de datos alineado.

Como se mencionó anteriormente, para resolver los sistemas lineales desplazados se utiliza una variante del método de Reducción Cíclica para GPU (siguiendo las ideas expuestas en [1]). Este enfoque puede ser descompuesto en tres pasos: reducción, resolución y sustitución. En la primera etapa, la matriz de coeficientes del sistema es reducida sucesivamente, eliminando la mitad de las incógnitas en cada paso. Este proceso se repite hasta que el sistema de ecuaciones es lo suficientemente pequeño. Cuando esto sucede, se resuelve el sistema, por ejemplo, utilizando el método de Thomas. Luego se realiza la etapa de sustitución, en la que los valores previamente calculados son introducidos en las ecuaciones, y un nuevo conjunto de incógnitas que serán resueltas en el paso siguiente es identificado.

La solución de sistemas tri-diagonales es una operación que presenta una cantidad modesta de operaciones en punto flotante en relación a la cantidad de accesos a memoria. Una de las ventajas de esta implementación es que saca partido del importante ancho de banda de memoria que brinda el dispositivo realizando accesos unificados o *coalesced*. Para lograr esto, la matriz se almacena en la memoria de la GPU como dos bloques de filas contiguas. La rutina toma la matriz de entrada y reordena las filas de salida en cada paso. Como consecuencia, las filas pares se almacenan en la parte inferior de la matriz, mientras que las impares se almacenan en la parte superior.

Como se ha mencionado, la etapa de reducción se detiene cuando el sistema transformado alcanza cierta dimensión. Luego, un único hilo ejecuta el algoritmo de Thomas, sacando partido de la memoria compartida del dispositivo. La dimensión óptima en la cual la etapa de reducción debe detenerse es específica para cada problema y hardware utilizado, determinándose empíricamente.

La implementación descrita se comparó con la que está presente en la biblioteca CUSPARSE [63]. Los resultados mostraron que la variante desarrollada supera levemente la implementación de CUSPARSE, por lo que el resolutor de CUSPARSE es utilizado solo en el caso de que la solución de los problemas requiera aritmética compleja, mientras que la implementación desarrollada se utiliza para los demás casos.

3.2.2. Implementación para matrices banda generales

Los sistemas lineales con matrices cuyos coeficientes no nulos se encuentran exclusivamente en un reducido número de diagonales (en comparación con la dimensión de la matriz), adyacentes a la diagonal principal de la matriz, aparecen en una gran variedad de aplicaciones. A manera de ejemplo, pueden mencionarse problemas que surgen en el análisis de elementos finitos en mecánica estructural, métodos de descomposición del dominio para ecuaciones diferenciales en derivadas parciales en ingeniería civil, así como problemas de control y teoría de sistemas. En estos casos, explotar la estructura particular de las matrices conlleva enormes beneficios, tanto en la utilización de la memoria como en la cantidad de operaciones. En consecuencia, la biblioteca LAPACK [5] propone un algoritmo para computar esta operación que, cuando se trabaja con una implementación paralela de BLAS, provee una forma eficiente de resolver el problema en arquitecturas multi-core de propósito general.

Formalmente, el problema consiste en resolver un sistema lineal del tipo

$$A\,X = B, \tag{3.1}$$

donde $A \in \mathbb{R}^{n\times n}$ es una matriz con ancho de banda superior e inferior k_u y k_l respectivamente, $B \in \mathbb{R}^{n\times m}$ contiene una colección de m vectores que forman el término independiente (usualmente $m \ll n$), y $X \in \mathbb{R}^{n\times m}$ es la solución buscada. Este problema puede ser resuelto con LAPACK en dos pasos. Primero, la matriz de coeficientes A se factoriza en dos factores triangulares $L, U \in \mathbb{R}^{n\times n}$ usando la rutina GBTRF, la cual realiza una factorización LU con pivoteo parcial. A continuación, la solución X es obtenida utilizando la rutina GBTRS, la cual resuelve los dos sistemas triangulares correspondientes a las matrices L y U.

Con el fin de comprender las principales limitaciones que experimentan estas rutinas, las cuales justifican el desarrollo de los nuevos algoritmos, en los siguientes apartados se describe la estrategia utilizada por LAPACK para realizar la factorización LU y la subsiguiente resolución de un sistema lineal banda. Posteriormente, se detallan las variantes propuestas para dichos métodos, así como una implementación híbrida del algoritmo, que realiza cómputos tanto CPU como en GPU.

Factorización de matrices banda utilizando LAPACK

LAPACK incluye dos variantes para el cálculo de la factorización LU de una matriz banda, GBTF2 y GBTRF, las cuales representan variantes "sin bloques" y "por bloques" del algoritmo respectivamente. Tal como se mencionaba anteriormente, el interés de este trabajo se centra en la resolución de sistemas de gran dimensión, por lo que a continuación se analiza el algoritmo por bloques, más eficiente para este tipo de problemas.

La rutina GBTRF calcula una *factorización LU con pivoteo parcial al estilo de LINPACK*

$$L_{n-2}^{-1} \cdot P_{n-2} \cdots L_1^{-1} \cdot P_1 \cdot L_0^{-1} \cdot P_0 \cdot A = U \tag{3.2}$$

donde $P_0,\ P_1, \ldots, P_{n-2} \in \mathbb{R}^{n\times n}$ son matrices de permutación, $L_0,\ L_1, \ldots, L_{n-2} \in \mathbb{R}^{n\times n}$ son transformaciones Gaussianas, y $U \in \mathbb{R}^{n\times n}$ es triangular superior y banda, con ancho de banda superior $k_l + k_u$. Respecto a la representación de los datos, LAPACK utiliza un formato empaquetado para almacenar matrices banda, el cual se ilustra en la Figura 3.3. En la figura se aprecia cómo se realiza el almacenamiento de la matriz banda y cómo los coeficientes correspondientes a los factores L y U se almacenan luego de la factorización, sobreescribiendo la matriz original. Notar que la matriz A es almacenada junto con k_l filas adicionales, inicializadas a ceros, destinadas a almacenar el *fill-in* generado durante la factorización. Una vez completada la misma, las entradas del factor U sobreescriben las entradas del triángulo superior de A (aquellas que se encuentran sobre la diagonal

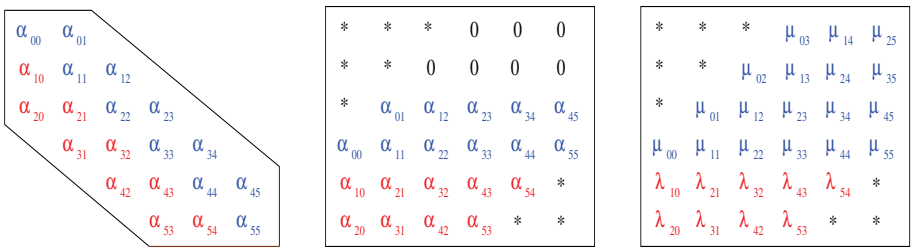

Figura 3.3: Matriz banda de tamaño 6×6 con anchos de banda inferior y superior $k_l = 2$ y $k_u = 1$, respectivamente (izquierda); formato de almacenamiento empaquetado utilizado por LAPACK (centro); resultado de la factorización LU donde $\mu_{i,j}$ y $\lambda_{i,j}$ representan, respectivamente, las entradas del factor triangular superior U y los multiplicadores de las transformaciones Gaussianas (derecha).

principal) mientras que el triángulo estrictamente inferior de A se reemplaza por los multiplicadores que definen las transformaciones Gaussianas.

A continuación se asume que el tamaño de bloque (b) utilizado internamente por el algoritmo, es un entero múltiplo de k_l y k_u, y se considera la siguiente partición:

$$A = \left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & A_{MR} \\ \hline & A_{BM} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} A_{00} & A_{01} & A_{02} & & \\ \hline A_{10} & A_{11} & A_{12} & A_{13} & \\ \hline A_{20} & A_{21} & A_{22} & A_{23} & A_{24} \\ \hline & A_{31} & A_{32} & A_{33} & A_{34} \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right), \tag{3.3}$$

donde $A_{TL}, A_{00} \in \mathbb{R}^{k \times k}$, (con k múltiplo de b,) $A_{11}, A_{33} \in \mathbb{R}^{b \times b}$, y $A_{22} \in \mathbb{R}^{l \times u}$, con $l = k_l - b$ y $u = k_u + k_l - b$.

La rutina GBTRF realiza la factorización de la matriz "hacia la derecha" (variante conocida como *right-looking factorization* [44]) lo cual significa que, antes de comenzar la iteración k/b del algoritmo, A_{TL} ya ha sido factorizada, A_{ML} y A_{TM} han sido reemplazados por los bloques correspondientes de U, A_{MM} ha sido actualizada y el resto de los bloques permanecen intactos. Es importante notar que, con este particionamiento, A_{31} es triangular superior, mientras que A_{13} es triangular inferior. Además, a esta altura del procedimiento, A_{13}, A_{23}, A_{24}, y A_{34} contienen solo ceros.

Durante una iteración dada del algoritmo, la rutina realiza las siguientes operaciones (las anotaciones a la derecha de cada operación corresponden al nombre de la rutina de BLAS que la realiza):

1. Obtener $W_{31} :=$ TRIU(A_{31}), una copia de triángulo superior de A_{31}, y computar la factorización LU con pivoteo parcial

$$P_1 \begin{pmatrix} A_{11} \\ A_{21} \\ W_{31} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \\ L_{31} \end{pmatrix} U_{11}. \tag{3.4}$$

Los bloques de L y U se sobreescriben con los bloques correspondientes de A y W_{31}.

2. Aplicar las permutaciones en P_1 a las restantes columnas de la matriz:

$$\begin{pmatrix} A_{12} \\ A_{22} \\ A_{32} \end{pmatrix} := P_1 \begin{pmatrix} A_{12} \\ A_{22} \\ A_{32} \end{pmatrix} \quad \text{and} \quad (\text{LASWP}) \tag{3.5}$$

$$\begin{pmatrix} A_{13} \\ A_{23} \\ A_{33} \end{pmatrix} := P_1 \begin{pmatrix} A_{13} \\ A_{23} \\ A_{33} \end{pmatrix}. \tag{3.6}$$

   Las permutaciones sobre el bloque A_{13} en (3.6) deben ser realizadas con cuidado, ya que sólo su parte triangular inferior está almacenada físicamente. Como resultado de las permutaciones, A_{13}, que originalmente contenía únicamente ceros, puede pasar a ser triangular inferior.

3. Relizar las actualizaciones:

$$A_{12}(= U_{12}) := L_{11}^{-1} A_{12}, \quad (\text{TRSM}) \tag{3.7}$$

$$A_{22} := A_{22} - L_{21} U_{12}, \quad (\text{GEMM}) \tag{3.8}$$

$$A_{32} := A_{32} - L_{31} U_{12}. \quad (\text{GEMM}) \tag{3.9}$$

4. Obtener una copia de la parte triangular inferior de A_{13}, $W_{13} := \text{TRIL}(A_{13})$, y computar las actualizaciones

$$W_{13}(= U_{13}) := L_{11}^{-1} W_{13}, \quad (\text{TRSM}) \tag{3.10}$$

$$A_{23} := A_{23} - L_{21} W_{13}, \quad (\text{GEMM}) \tag{3.11}$$

$$A_{33} := A_{33} - L_{31} W_{13}; \quad (\text{GEMM}) \tag{3.12}$$

   reestableciendo $A_{13} := \text{TRIL}(W_{13})$.

5. Deshacer las permutaciones realizadas en $\left[L_{11}^T, L_{21}^T, W_{31}^T\right]^T$ para que estos bloques almacenen los multiplicadores utilizados por la factorización LU en (3.6) y W_{31} sea triangular superior. Copiar nuevamente $A_{31} := \text{TRIU}(W_{31})$.

Tras realizar estas operaciones, A_{TL} (la parte que ya ha sido factorizada) crece b filas/columnas de forma que

$$A = \left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & A_{MR} \\ \hline & A_{BM} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c|c|c} A_{00} & A_{01} & A_{02} & & \\ \hline A_{10} & A_{11} & A_{12} & A_{13} & \\ \hline A_{20} & A_{21} & A_{22} & A_{23} & A_{24} \\ \hline & A_{31} & A_{32} & A_{33} & A_{34} \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right), \tag{3.13}$$

es decir, $A_{TL} \in \mathbb{R}^{(k+b)\times(k+b)}$ para la siguiente iteración.

Asumiendo que $b \ll k_u, k_l$ (en CPU, para optimizar el uso de caché, $b \approx 32$ o 64), la actualización de A_{22} implica la mayor parte de las operaciones de punto flotante (flops). Esta operación puede expresarse en términos de multiplicaciones matriciales, una operación que exhibe un alto grado de paralelismo y, por lo tanto, es esperable un buen rendimiento de la rutina GBTRF siempre y cuando se utilice una versión eficiente de la rutina GEMM de BLAS.

Por otro lado, el algoritmo presenta dos desventajas en cuanto a su implementación en arquitecturas paralelas:

- La estructura triangular del bloque A_{13}/U_{13} no es explotada en (3.10)–(3.12) ya que no existe una rutina específica de BLAS para realizar dicha operación. Como consecuencia, es necesario mantener un espacio de trabajo adicional, así como dos copias adicionales. Además una cantidad considerable de flops se desperdician en operaciones que involucran elementos nulos.

- El formato de almacenamiento empleado por LAPACK y la falta de rutinas especializadas de BLAS obligan a que las actualizaciones que se realizan durante la iteración sean divididas en pequeñas operaciones que ofrecen escaso paralelismo.

Solución de los sistemas triangulares

Dada la factorización LU calculada por la rutina GBTRF en (3.2), GBTRS realiza la resolución de los sistemas triangulares banda resultantes para, finalmente, obtener la solución de (3.1). La rutina realiza las siguientes operaciones:

1. Para $i = 0, 1, \ldots, n-2$, (en ese orden) se aplican secuencialmente las matrices de permutación P_i al término independiente B, y la actualización de la matriz con los multiplicadores correspondientes en L_i:
$$\begin{array}{rcll} B & := & P_i B, & (\text{SWAP}) \\ B & := & L_i^{-1} B. & (\text{GER}) \end{array} \tag{3.14}$$

2. Para $j = 1, 2, \ldots, m$ resolver el sistema triangular con matriz de coeficientes U y término independiente dado por la j-ésima columna de B (B_j)
$$B_j := U^{-1} B_j. \qquad (\text{TBSV}) \tag{3.15}$$

A pesar que la operación resuelta por GBTRS pertenece conceptualmente al nivel 3 de BLAS, en esta implementación está expresada enteramente en términos de operaciones menos eficientes, pertenecientes al nivel 2 de BLAS. Esto es debido al formato de almacenamiento adoptado, que implica no formar el factor L explícitamente, y a la falta de una rutina particular en la especificación de BLAS para resolver un sistema triangular banda con múltiples vectores en el término independiente.

Nuevos resolutores híbridos (CPU-GPU) para sistemas lineales banda.

El algoritmo empleado por la rutina GBTRF invoca, en cada iteración, a la rutina TRSM dos veces y a la rutina GEMM cuatro veces (pasos 3 y 4). El particionamiento del trabajo es consecuencia del esquema particular que utiliza LAPACK para almacenar matrices banda. Sin embargo, como el grado de concurrencia que se extrae de la operación depende del uso de rutinas multi-hilo de BLAS, la fragmentación del trabajo en pequeñas operaciones limita el desempeño de las mismas. En arquitecturas similares a la de las GPUs, donde los cálculos deben realizarse sobre grandes conjuntos de datos y es necesario un número considerable de flops para explotar correctamente las capacidades del hardware, este tipo de fragmentación es particularmente desaconsejable.

De forma similar, la rutina GBTRS está fuertemente basada en operaciones correspondientes al nivel 2 de BLAS, por ejemplo, resolviendo los sistemas triangulares para cada columna del

término independiente por separado. Esta fragmentación de las tareas provoca, nuevamente, una disminución en el grado de concurrencia alcanzable.

Considerando lo descrito en los párrafos anteriores, a continuación se describe una variante de las rutinas GBTRF y GBTRS más adecuada para ser computada por una GPU. Esta variante se basa en un cambio mínimo en el formato de almacenamiento y el reordenamiento de algunas tareas que permite su fusión, así como el uso eficiente de las rutinas del nivel 3 de BLAS, mejorando el grado de concurrencia de estas operaciones.

Rutina GBTRF+M

La idea principal de la nueva rutina consiste en modificar ligeramente el formato empaquetado para matrices banda utilizado por LAPACK para almacenar la matriz A (ver Figura 3.3-centro), agregando b filas adicionales en la parte inferior de la matriz, con todas sus entradas inicializadas a ceros. De esta forma, los pasos 1 a 4 en la implementación original de GBTRF son transformados de la siguiente manera:

1. En el primer paso, se calcula la factorización LU con pivoteo parcial

$$P_1 \begin{pmatrix} A_{11} \\ A_{21} \\ A_{31} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \\ L_{31} \end{pmatrix} U_{11}, \tag{3.16}$$

   y los bloques L y U sobreescriben los bloques correspondientes de la matriz A. Ya no es necesario reservar memoria para W_{31}, así como realizar copias desde y hacia este espacio de memoria, ya que las filas adicionales son capaces de almacenar la parte estrictamente inferior de L_{31}.

2. Aplicar las permutaciones en P_1 a las restantes columnas de la matriz:

$$\begin{pmatrix} A_{12} & A_{13} \\ A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix} := P_1 \begin{pmatrix} A_{12} & A_{13} \\ A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix} \quad \text{(LASWP)}. \tag{3.17}$$

   Como las filas adicionales agregadas a la matriz junto con las k_l superdiagonales propias de la estructura original son suficientes para alojar el llenado que puede generarse en el bloque A_{13} debido a las permutaciones, éstas pueden realizarse mediante una sola invocación a la rutina LASWP.

3. Realizar las actualizaciones:

$$(A_{12},\ A_{13})\,(=(U_{12},\ U_{13})) := L_{11}^{-1}\,(A_{12},\ A_{13})\,, \quad \text{(TRSM)}, \tag{3.18}$$

$$\begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix} := \begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} (U_{12},\ U_{13}) \quad \text{(GEMM)}. \tag{3.19}$$

   El sistema triangular inferior en (3.18) tiene como resultado un bloque triangular inferior en A_{13}.

4. Deshacer las permutaciones realizadas en $\left[L_{11}^T, L_{21}^T, W_{31}^T\right]^T$ para que estos bloques almacenen los multiplicadores utilizados por la factorización LU en (3.6) y W_{31} sea triangular superior.

Como se puede observar en la descripción de las nuevas operaciones, esta reorganización del algoritmo es rica en productos matriciales y, por lo tanto, más adecuada para arquitecturas masivamente paralelas.

La implementación saca provecho de esta mejor concurrencia con el propósito de explotar las capacidades de la arquitectura híbrida CPU-GPU de forma eficiente. En particular, la factorización del panel estrecho de b columnas, la que presenta un modesto costo computacional, así como un escaso nivel de paralelismo, es realizada por la CPU, mientras que las permutaciones en (3.17) y actualizaciones en (3.18)–(3.19), que presentan mayor grado de paralelismo, son realizadas en la GPU.

De esta forma, cada operación se ejecuta en el dispositivo más conveniente, sacando máximo partido de la arquitectura a cambio de un numero moderado de transferencias de datos entre CPU y GPU. Particularmente, se requiere una fase de inicialización en la que la matriz A es transferida al dispositivo antes de comenzar la factorización. Luego, durante cada iteración del algoritmo es necesario transferir datos en dos oportunidades:

1. Las entradas de $[A_{11}^T, A_{21}^T, A_{31}^T]^T$ son transferidas a GPU después de la factorización de esta submatriz en (3.16).

2. A su vez, las entradas que formarán la submatriz $[A_{11}^T, A_{21}^T, A_{31}^T]^T$ durante la siguiente iteración son transferidas a la CPU tras su actualización como parte de (3.19).

La cantidad de datos transferida es moderada en relación con la cantidad de operaciones de punto flotante, ya que la cantidad de filas del bloque $[A_{11}^T\, A_{21}^T\, A_{31}^T]^T$ es $(k_u + k_l + k_l)$ mientras que la cantidad de columnas es b. Adicionalmente, al finalizar el algoritmo, la matriz factorizada se encuentra tanto en CPU como en GPU, lo que permite su utilización en ambos dispositivos para cálculos subsiguientes.

Rutina GBTRF+LA

La rutina GBTRF+LA es una variante de la rutina GBTRF+M que incorpora el uso de técnicas de *look-ahead* [79], buscando además superponer parte de los cálculos realizados por la CPU y la GPU. Concretamente, consiste en reordenar estos cálculos de la siguiente manera: las actualizaciones en (3.17)–(3.19), que involucran $k_u + k_l$ columnas de la matriz, son divididas por columnas, de forma que tras calcularse las primeras b columnas, que contienen los elementos actualizados que forman $[A_{11}^T, A_{21}^T, A_{31}]^T$ en la siguiente iteración, éstas son enviadas a la CPU, donde la factorización de este bloque puede realizarse de forma concurrente con la actualización de las restantes $k_u + k_l - b$ columnas correspondientes a (3.17)–(3.19) en la GPU.

Esta variante requiere mínimos cambios en los códigos originales. A pesar de que implica la ejecución de *kernels* con un moderado número de flops, permite realizar cálculos en ambos dispositivos de forma concurrente, reportando mejores resultados en aquellos casos en que $b \ll k_u + k_l$.

Rutina GBTRS+M

La principal desventaja de la rutina GBTRS de LAPACK es la ausencia de rutinas de BLAS-3 en su implementación. Esto se debe a la adopción del formato de almacenamiento empaquetado para las matrices banda, y a la falta de las rutinas apropiadas de BLAS. Desafortunadamente, las modificaciones introducidas en el formato de almacenamiento todavía limitan la utilización de rutinas BLAS-3. Dado que la matriz L no se forma explícitamente, las actualizaciones en (3.14) deben realizarse en términos de actualizaciones de rango uno (rutina GER de BLAS-2). Sin embargo, es posible utilizar rutinas del nivel 3 para la posterior resolución de los sistemas triangulares

en (3.15). En este sentido, se desarrolló una rutina, llamada TBSM de acuerdo con la nomenclatura de LAPACK, que realiza esta operación por medio de rutinas BLAS-3 (principalmente productos matriciales). La operación se realizó, entonces, en dos etapas: primero se actualiza B de la misma forma que en (3.14), para luego resolver los sistemas triangulares banda para todos los vectores B_j mediante una sola invocación a la rutina TBSM, descrita brevemente a continuación.

Rutina TBSM

Considérese la siguiente partición del término independiente B, el cual se remplaza con la solución X de (3.1) una vez finalizada la rutina,

$$B = \left(\begin{array}{c} B_T \\ \hline B_M \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \\ \hline B_3 \\ \hline B_4 \end{array}\right), \tag{3.20}$$

donde B_B, B_4 tienen ambas k filas (con k múltiplo de b), B_1, B_3 tienen b filas cada una, y B_2 tiene $u = k_u + k_l - b$ filas. En esta expresión, B_B representa la parte del término independiente que ya ha sido reemplazada por la parte correspondiente de la solución X.

Considérese también la siguiente partición para la matriz triangular superior U resultante de la factorización previa

$$U = \left(\begin{array}{c|c|c} U_{TL} & U_{TM} & \\ \hline & U_{MM} & U_{MR} \\ \hline & & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} U_{00} & U_{01} & U_{02} & & \\ \hline & U_{11} & U_{12} & U_{13} & \\ \hline & & U_{22} & U_{23} & U_{24} \\ \hline & & & U_{33} & U_{34} \\ \hline & & & & U_{44} \end{array}\right), \tag{3.21}$$

donde $U_{BR},\ U_{44} \in \mathbb{R}^{k\times k}$; $U_{13}, U_{33} \in \mathbb{R}^{b\times b}$ son bloques triangulares inferior y superior respectivamente, y $U_{22} \in \mathbb{R}^{u\times u}$.

Con esta partición, cada iteración del algoritmo realiza las siguientes operaciones:

$$B_3 := U_{33}^{-1} B_3, \qquad \text{(TRSM)} \tag{3.22}$$
$$B_2 := B_2 - U_{23} B_3, \qquad \text{(GEMM)} \tag{3.23}$$
$$B_1 := B_1 - U_{13} B_3. \tag{3.24}$$

La última actualización involucra una matriz triangular y puede ser realizada por medio de la rutina de BLAS TRMM. Sin embargo, esto requiere un espacio de almacenamiento auxiliar,$W_{13} \in \mathbb{R}^{b\times m}$, ya que esta rutina de BLAS realiza productos de la forma $M := U_{13}\ M$. En consecuencia se realizan las siguientes operaciones para (3.24):

$$W_{13} := B_3, \tag{3.25}$$
$$W_{13} := U_{13} W_{13}, \qquad \text{(TRMM)} \tag{3.26}$$
$$B_1 := B_1 - W_{13}. \tag{3.27}$$

Tras completarse, los límites de la partición para B y U son desplazados de acuerdo con:

$$B = \left(\begin{array}{c} B_T \\ \hline B_M \\ \hline B_B \end{array} \right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \\ \hline B_3 \\ \hline B_4 \end{array} \right),$$

$$U = \left(\begin{array}{c|c|c} U_{TL} & U_{TM} & \\ \hline & U_{MM} & U_{MR} \\ \hline & & U_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c|c|c|c} U_{00} & U_{01} & U_{02} & & \\ \hline & U_{11} & U_{12} & U_{13} & \\ \hline & & U_{22} & U_{23} & U_{24} \\ \hline & & & U_{33} & U_{34} \\ \hline & & & & U_{44} \end{array} \right). \tag{3.28}$$

Rutina MERGE

Con el objetivo de explotar al máximo las capacidades brindadas por la plataforma híbrida por medio de la ejecución de operaciones en CPU y GPU de forma concurrente, se propuso una variante de la rutina GBTRF que combina la etapa de actualización en GPU de la factorización LU de la matriz de coeficientes con la solución del primer sistema lineal triangular banda en la CPU.

Originalmente, la solución de ambos sistemas triangulares banda (superior e inferior) se ejecutaba de forma posterior a la factorización LU. En otras palabras, los multiplicadores correspondientes a la matriz L y la matriz U se calculaban completamente antes de resolver los sistemas $LY = B$ y $UX = Y$.

El nuevo enfoque combina cierta parte de la solución de los sistemas triangulares banda con la factorización LU, de forma que, en cada paso de la iteración encargada de computar L y U, una vez que se obtienen los bloques correspondientes de L mediante la Ecuación (3.16), la actualización de la matriz A en GPU (Ecuaciones (3.18) y (3.19)) y la solución y actualización parcial de la matriz B (Ecuación (3.14)) en la CPU, se calclulan de forma concurrente.

Aplicando este enfoque es posible lograr un mejor aprovechamiento de la localidad de datos, ya que se anticipa la solución del sistema lineal que involucra a L cuando el procedimiento de la factorización aún está manipulando estos bloques, y de la plataforma híbrida, al superponer dos operaciones en procesadores diferentes, ocultando el costo computacional de la operación que consuma menos tiempo.

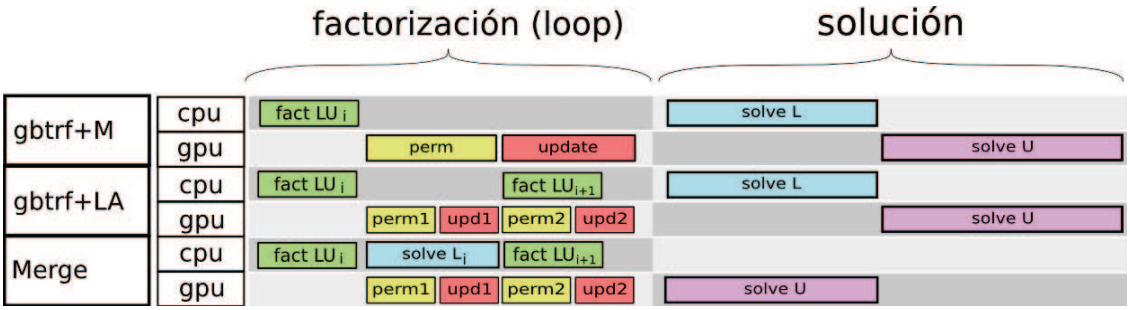

Figura 3.4: Diagrama de las distintas versiones del resolutor banda híbrido (CPU-GPU).

3.2.3. Implementación para matrices dispersas generales

La implementación de un conjunto de USFs para matrices dispersas no estructuradas es imprescindible en cualquier biblioteca que incluya rutinas para reducción de modelos dispersos. Sin embargo, como se podrá observar a continuación, principalmente para el caso de matrices dispersas de gran dimensión, la implementación de estas operaciones no es una tarea trivial y presenta algunos problemas de difícil solución.

Como se mencionó al principio de este capítulo, Lyapack basa sus algoritmos para reducción de modelos dispersos en tres operaciones básicas, a saber, el producto matricial, la solución de sistemas de ecuaciones lineales y la solución de sistemas de ecuaciones lineales desplazados. Si bien en el caso de los productos matriciales existen implementaciones en GPU maduras y eficientes, las principales dificultades se relacionan con la resolución de sistemas de ecuaciones lineales dispersos y, en particular, con los sistemas desplazados.

Aunque bien los formatos de almacenamiento para matrices dispersas hacen posible almacenar matrices de gran dimensión con un costo de memoria aceptable, la utilización de métodos directos para resolver sistemas dispersos requieren un importante uso de memoria. Adicionalmente, suelen ser lentos y extremadamente difíciles de implementar en arquitecturas masivamente paralelas [72]. En cambio, los métodos iterativos son una opción atractiva para resolver este tipo de problemas, dada su simplicidad y bajo costo tanto computacional como de memoria.

Dentro de los métodos iterativos, los métodos basados en subespacios de Krylov ocupan un lugar prominente. De hecho, existen cientos de variaciones de estos métodos, y muchos esfuerzos se han dedicado a la implementación eficiente de estos algoritmos en GPU. En este sentido se ha realizado una profunda revisión del estado del arte, que puede consultarse en el Anexo C.

En este trabajo se decidió implementar dos de los métodos iterativos para la solución de sistemas de ecuaciones lineales dispersos más populares y ampliamente utilizados, a saber, el método del Gradiente Conjugado [49] (CG por sus siglas en inglés), para matrices de coeficientes simétricas y definidas, y el método del Gradiente Bi-Conjugado Estabilizado [83] (Bi-CGStab por sus siglas en inglés), para matrices posiblemente no simétricas e indefinidas.

Implementación del resolutor disperso

Los métodos CG y Bi-CGStab son muy similares entre sí. Ambos se basan en generar una proyección sobre el espacio de Krylov $\mathcal{K}_n = (A, v_0)$, donde n es la dimensión del problema, salvo que en el caso de Bi-CGStab esta proyección es también ortogonal al subespacio de Krylov descrito por $\mathcal{K}_n = (A^T, w_0)$.

Ambos métodos, ver Algoritmos 4 y 5, son procesos iterativos cuya operación principal es la multiplicación de una matriz dispersa por un vector, realizada además en cada paso de la iteración (dos veces en el caso de BiCGStab). Por lo tanto, una buena implementación de esta operación es fundamental para obtener un resolutor iterativo para matrices dispersas eficiente.

La implementación en GPU realizada para estos métodos está basada en el enfoque descrito en [61] para la solución de sistemas lineales dispersos en GPU. Este algoritmo es adecuado para la ejecución en el dispositivo y provoca una aceleración considerable del resolutor basado en GPU para la ecuación de Lyapunov. En esta implementación, el producto matriz-vector es calculado por la GPU, y las matrices dispersas se almacenan en formato comprimido por filas (CSR) [72]. Además, la matriz de coeficientes A se transfiere desde la memoria de la CPU a la GPU una única vez durante la etapa de inicialización del algoritmo, mientras que las sucesivas matrices de tipo $(A + \sigma I)$, en el caso de los sistemas desplazados, se construyen en GPU.

La construcción de dichas matrices puede ser realizada de distintas formas. Un enfoque ingenuo

consistiría en sumar σ a todos los elementos de la diagonal, pero en el caso de existir ceros en la diagonal de A esto implica modificar estructuralmente la matriz, por lo que podría ser necesario realizar un cambio en la cantidad de memoria reservada para la matriz en la GPU, así como un reordenamiento y actualización de los vectores de índices que componen el formato CSR. Una alternativa consiste en mantener una copia de A y, simultáneamente, reservar un espacio distinto para mantener $(A + \sigma I)$, pero esto podría significar un incremento notorio en los requerimientos de memoria, así como cómputos innecesarios. Un enfoque menos drástico es guardar la matriz A almacenando explícitamente los ceros existentes en su diagonal. Además de significar un potencial desperdicio de memoria, este enfoque necesariamente realiza cómputos inútiles con elementos nulos de la matriz. Para evitar estos inconvenientes, se ha expresado el cómputo del producto matriz vector $(A + \sigma I)y$ como dos operaciones separadas $(Ay) + (\sigma Iy) = Ay + \sigma y$, de forma que el producto matriz-vector resultante, así como el posterior escalado, pueden ser realizados fácilmente mediante las rutinas CSRMV y AXPY de las bibliotecas CUSPARSE y CUBLAS respectivamente.

El resto de las operaciones en los Algoritmos 4 y 5 son realizadas por medio de las rutinas de CUBLAS correspondientes.

Algoritmo 4 Gradiente Conjugado Precondicionado

Input: A, M, b, σ
Output: x

1: $x_0 := 0$
2: $r_0 := b - Ax_0$
3: $z_0 := M^{-1}r_0$
4: $d_0 := z_0$
5: $\beta_0 := r_0^T z_0$
6: $\tau_0 :=\| r_0 \|_2$
7: $k := 0$
8: **while** $(\tau_k > \tau_{\text{máx}})$ **do**
9: $w_k := Ad_k$
10: **if** $\sigma \neq 0$ **then**
11: $w_k = w_k + \sigma d_k$
12: **end if**
13: $\rho_k := \beta_k / d_k^T w_k$
14: $x_{k+1} := x_k + \rho_k d_k$
15: $r_{k+1} := r_k - \rho_k w_k$
16: $z_{k+1} := M^{-1} r_{k+1}$
17: $\beta_{k+1} := r_{k+1}^T z_{k+1}$
18: $\alpha_k := \beta_{k+1} / \beta_k$
19: $d_{k+1} := z_{k+1} + \alpha_k d_k$
20: $\tau_{k+1} :=\| r_{k+1} \|_2$
21: $k := k + 1$
22: **end while**

Algoritmo 5 Gradiente Bi-Conjugado Estabilizado

Input: A, b, σ

Output: x

1: $r_0 = b - Ax_0$
2: $\hat{r}_0 = r_0$
3: $\rho_0 = \alpha = \omega_0 = 1$
4: $v_0 = p_0 = 0$
5: **for** $i = 1, 2, 3, \ldots$ **do**
6: $\quad \rho_i = (\hat{r}_0, r_{i-1})$
7: $\quad \beta = (\rho_i/\rho_{i-1})(\alpha/\omega_{i-1})$
8: $\quad p_i = r_{i-1} + \beta(p_{i-1} - \omega_{i-1}v_{i-1})$
9: $\quad v_i = Ap_i$
10: $\quad$ **if** $\sigma \neq 0$ **then**
11: $\quad\quad v_i = v_i + \sigma p_i$
12: $\quad$ **end if**
13: $\quad \alpha = \rho_i/(\hat{r}_0, v_i)$
14: $\quad s = r_{i-1} - \alpha v_i$
15: $\quad t = As$
16: $\quad$ **if** $\sigma \neq 0$ **then**
17: $\quad\quad t = t + \sigma s$
18: $\quad$ **end if**
19: $\quad \omega_i = (t, s)/(t, t)$
20: $\quad x_i = x_{i-1} + \alpha p_{i-1} - \omega_i s$
21: $\quad$ **if** x_i se acerca lo suficiente a la solución **then** stop
22: $\quad$ **else**
23: $\quad\quad r_i = s - \omega_i t$
24: $\quad$ **end if**
25: **end for**

Utilización de precondicionadores

Una de las desventajas principales que presentan los métodos iterativos frente a los métodos directos para la solución de sistemas lineales es su falta de robustez y su susceptibilidad a errores numéricos, introducidos inevitablemente por la representación de números reales mediante el sistema de punto flotante. Adicionalmente, el efecto de estos errores sobre los cálculos que se realizan durante los métodos iterativos varía de acuerdo con las características del problema. Lo que sucede en la práctica es que, para una gran cantidad de problemas, este tipo de métodos no convergen o toman demasiadas iteraciones (y tiempo de ejecución) en converger a una solución aceptable [72].

En estos casos, un enfoque ampliamente utilizado es la aplicación de un precondicionador o matriz de precondicionado. La idea consiste en reemplazar el sistema de ecuaciones original por uno de igual solución, pero cuyas propiedades numéricas lo hagan menos susceptible a los efectos de los errores de redondeo, de forma que pueda ser resuelto por un computador mediante un método iterativo. Por ejemplo, un sistema precondicionado por la izquierda (*left preconditioning*) consiste en resolver un sistema de tipo $M^{-1}AX = M^{-1}B$. En estos casos, la matriz M^{-1} suele asemejarse, en algún sentido, a la matriz A^{-1} (notar que en el caso que $A^{-1} = M^{-1}$ el sistema se resuelve directamente).

Existe una enorme variedad de métodos de precondicionado, precondicionadores y métodos iterativos para resolver sistemas lineales, cada uno con ventajas y desventajas en cuanto a costo computacional, convergencia y rango de problemas a los que es aplicable. Adicionalmente, ningún método de precondicionado logra los mejores resultados para todo tipo de problemas. Sin embargo, los precondicionadores basados en factorizaciones matriciales incompletas son una herramienta ampliamente utilizada [72]. Este tipo de precondicionadores se basa en calcular una factorización de la matriz A permitiendo solamente cierto grado de llenado, manteniendo controlados los requerimientos de memoria y la complejidad de la aplicación del precondicionador durante el método iterativo, mientras se espera que el producto de dichos factores se asemeje a la matriz A y, más importante, que el producto de sus inversas se asemeje a A^{-1}.

Un caso particular de estos métodos es la factorización LU incompleta (ILU). La variante que no permite llenado (ILU0) se encuentra implementada en GPU y está disponible como parte de la biblioteca CUSPARSE. Sin embargo, si bien este precondicionador es bastante simple y general, existen otros más sofisticados que ofrecen resultados superiores para una gran variedad de problemas.

En este sentido, ILUPACK[2] es un paquete para la resolución de sistemas lineales dispersos mediante métodos iterativos que se encuentra en constante desarrollo; ver Anexo D. Este software integra métodos multinivel para la factorización LU incompleta basada en la inversa (*inverse-based multilevel ILU*) para el precondicionado de sistemas lineales generales, hermíticos, simétricos indefinidos y definidos positivos, en combinación con métodos de Krylov para la resolución de dichos sistemas.

ILUPACK ha sido aplicado con éxito en diversos problemas dispersos de gran escala, algunos de ellos compuestos por millones de ecuaciones [26, 74]. En contraste con otros precondicionadores basados en la factorización LU incompleta (ILU) la efectividad de ILUPACK se basa en mantener un control estricto sobre el crecimiento de la norma de los inversos de los factores triangulares [26, 75, 43]. El lector interesado en una descripción minuciosa de los métodos iterativos precondicionados puede consultar [72].

[2]Disponible en `http://ilupack.tu-bs.de`.

Implementación del resolutor disperso utilizando el precondicionador de ILUPACK

Considerando que la resolución de sistemas de ecuaciones lineales es la operación más costosa, desde el punto de vista computacional, de los algoritmos utilizados en este trabajo para la reducción de modelos dispersos, y que ILUPACK es un software que muestra un desarrollo continuo, así como una amplia utilización, representando el estado del arte en cuanto a métodos iterativos precondicionados, es particularmente interesante el desarrollo de una versión que incluya la utilización de procesadores gráficos en los métodos implementados en dicho software. Adicionalmente, la aceleración de los métodos incluidos en ILUPACK es transferible a diversas áreas, trascendiendo su aplicación a la reducción de modelos.

Si bien ILUPACK permite resolver tanto sistemas simétricos como no simétricos, la complejidad de la herramienta provoca que en este trabajo se haya abordado únicamente la paralelización de ILUPACK para sistemas lineales simétricos y definidos positivos (SPD). A diferencia de las propuestas desarrolladas en [2, 3], la estrategia elegida para incluir técnicas de paralelismo no obtiene mayor concurrencia a nivel de tarea mediante el incremento de operaciones de punto flotante, sino que explota el paralelismo inherente a las principales operaciones que componen el resolutor iterativo de ILUPACK, realizando las mismas en el procesador gráfico. Trabajando de esta manera es posible acelerar el resolutor iterativo de ILUPACK manteniendo la precisión[3] y las propiedades de convergencia que presenta la biblioteca.

ILUPACK provee dos funcionalidades principales: la construcción de un precondicionador multinivel mediante una factorización LU incompleta basada en una estimación de la inversa de los factores, y la solución de sistemas de ecuaciones lineales mediante diversos métodos iterativos utilizando este precondicionador. Teniendo en cuenta que el interés en ILUPACK desde el punto de vista de este trabajo es acelerar el resolutor iterativo basado en el método del Gradiente Conjugado Precondicionado (PCG) utilizando el precondicionador multinivel calculado por la herramienta con el fin de acelerar la resolución de las ecuaciones de Lyapunov para matrices dispersas generales, el esfuerzo se ha centrado en paralelizar la aplicación del precondicionador, dejando de lado la construcción del mismo.

En el Anexo D el lector interesado puede encontrar una descripción más completa de la herramienta ILUPACK, así como una derivación matemática de las operaciones que componen la aplicación del precondicionador multinivel. Para comprender los detalles acerca de la implementación que se describen a continuación, únicamente es necesario tener en cuenta que la aplicación del precondicionador multinivel es un algoritmo recursivo, de tantos pasos como niveles tenga el precondicionador calculado, en el que, en cada paso, se realizan dos multiplicaciones de matriz dispersa por vector, y se resuelven dos sistemas lineales de tipo $LDL^T x = b$. Adicionalmente, es posible distinguir otros tres tipos de operación: escalado diagonal, permutación de vectores, y actualizaciones de tipo $x := a - b$.

En cuanto a las estructuras de datos, ILUPACK almacena los factores LDL^T en un formato CSR-extendido, donde las entradas en la parte estrictamente triangular inferior de L (el factor triangular inferior unidad) se almacenan en formato CSR, y la matriz diagonal D se almacena en un vector separado.

El resto de este apartado discute la paralelización de la aplicación del precondicionador multinivel de ILUPACK.

[3] Salvo por los errores de punto flotante.

Propuestas

A continuación se presentan dos versiones del resolutor basado en ILUPACK acelerado con GPU. La primera versión simplemente delega en la GPU la resolución de sistemas lineales triangulares y multiplicaciones de matriz dispersa por vector que aparecen durante la aplicación de cada nivel del precondicionador (ver Anexo D), y que son los principales cuellos de botella, desde el punto de vista computacional, en la mayoría de los escenarios. La segunda versión explota la capacidad de la GPU para abordar todas las operaciones presentes en la aplicación del precondicionador, evitando así transferencias de datos entre CPU y GPU, y acelerando también el resto de las operaciones (sobre vectores).

Versión 1

Esta versión apunta a acelerar dos tipos de operación: la solución de sistemas triangulares y el producto de matriz dispersa por vector, usando las primitivas brindadas por la biblioteca CUSPARSE en ambos casos.

Dado el formato de almacenamiento adoptado por ILUPACK para los factores LDL^T en concreto, una variante del formato CSR llamada *extended-CSR*, y que el formato nativo de CUSPARSE es CSR, es necesario realizar una reorganización de los datos antes de poder llamar a la primitiva correspondiente de CUSPARSE. En la implementación actual, este proceso es realizado por la CPU, durante la etapa de construcción del precondicionador.

Tras la transformación, los factores LDL^T se transfieren y almacenan en la GPU como dos matrices $\hat{L}$ y $\hat{D} = D^{-1}$, donde $\hat{L}$ se almacena en formato CSR. Ya que CUSPARSE provee un resolutor de sistemas lineales triangulares para matrices en este formato, el sistema $LDL^T x = b$ se procesa resolviendo primero $\hat{L}y = b$ para y, luego computando $z = \hat{D}y$, y por último resolviendo $\hat{L}^T x = z$ para x. Esto se realiza mediante dos llamados a la rutina `cusparseDcsrsv_solve`, con el argumento apropiado para operar con $\hat{L}$ o su traspuesta $\hat{L}^T$, y un sencillo *kernel* en GPU para realizar la multiplicación de la matriz diagonal. Para reducir el costo de estas operaciones, la fase de análisis requerida por los resolutor triangulares es realizada una única vez por cada nivel del precondicionador durante la etapa de construcción.

Las multiplicaciones de matriz dispersa por vector (`spmv`) que necesitan ser calculadas durante la aplicación del precondicionador en cada nivel son de la forma $x := Fv$ y $x := F^T v$. La matriz F es almacenada en formato CSR por ILUPACK y, por lo tanto, no es necesaria ninguna reorganización de los datos previa a la invocación de la rutina de CUSPARSE. Sin embargo, tanto la matriz como su transpuesta están involucradas en los cálculos, por lo que pueden seguirse al menos dos estrategias distintas. La primera consiste en utilizar el parámetro que brinda la rutina `cusparseDcsrmv` de CUSPARSE para invocar la multiplicación de un vector con la matriz transpuesta de matriz que se pasa por parámetro. Siguiendo esta estrategia es posible almacenar únicamente la matriz F (o F^T) en la GPU, pero realizar ambas operaciones simplemente cambiando dicho parámetro. Sin embargo, existe lógicamente una diferencia algorítmica entre la implementación de ambas operaciones en GPU. Una evaluación preliminar de este enfoque reveló resultados numéricamente distintos que alteraban la precisión del resultado global al terminar las iteraciones del resolutor. El manual de CUSPARSE advierte al usuario de que esta rutina produce resultados ligeramente distintos para ejecuciones diferentes con los mismos parámetros, al ser utilizada con el parámetro que habilita la multiplicación por la transpuesta. Si, por el contrario, es necesario que los resultados sean iguales, el manual desestima el uso de este parámetro. Adicionalmente, el desempeño de esta rutina para el caso transpuesto es considerablemente menor que el caso no transpuesto.

Por estas razones, y a pesar de utilizar más memoria, se optó por el enfoque de almacenar

tanto F como F^T en la GPU, utilizando una primitiva de la propia biblioteca CUSPARSE para transponer la matriz, lo cual se realiza una única vez por cada nivel, durante la etapa de construcción del precondicionador.

Versión 2

En la mayor parte de los problemas, la solución de sistemas lineales triangulares y la función `spmv` que aparecen en cada nivel de la aplicación del precondicionador son las operaciones que consumen la mayor parte del tiempo de cómputo. En la primera versión, solo se migraron estas operaciones a la GPU. Sin embargo, aunque las restantes operaciones vectoriales que involucra el algoritmo no poseen un alto costo computacional, si se realizan en CPU, es necesario transferir los resultados a la GPU una vez que estos se requieren, así como transferir el resultado de los sistemas triangulares y la función `spmv` a la CPU. Además, como el algoritmo recursivo que aplica el precondicionador multinivel consiste en una secuencia estricta de pasos (donde la concurrencia se extrae de operaciones que componen cada uno de estos pasos) no es posible solapar cálculos y transferencias.

Por estos motivos, se realizó un esfuerzo adicional en portar por completo la aplicación del precondicionador a la GPU. Esto implica que el residuo r_{k+1} es transferido desde la memoria de la CPU a la GPU antes de que el precondicionador sea aplicado, luego la aplicación entera del precondicionador se realiza por la GPU para obtener $z_{k+1} := M^{-1}r_{k+1}$, y por último el residuo precondicionado z_{k+1} se transfere a la memoria de la CPU una vez completada la operación.

Para hacer esto posible, se implementaron tres *kernels* para GPU adicionales:

- El *kernel de escalado diagonal* multiplica cada elemento de un vector de entrada por el elemento correspondiente de un vector de escalado. Esto equivale a multiplicar el vector de entrada por una matriz diagonal cuya diagonal es el vector de escalado.
- El *kernel de ordenamiento* reordena un vector de entrada v_{in} aplicando una permutación contenida en un vector p, y produce un vector v_{out}, cuyas entradas $v_{out}[i] := v_{in}[p(i)]$.
- El tercer kernel simplemente implementa la resta $c := a - b$ donde a, b y c son vectores.

Realizando estas operaciones en la GPU se evitan transferencias innecesarias y se reducen los tiempos de ejecución significativamente, especialmente para aquellos casos donde el tamaño de los vectores es bastante grande en comparación con el número de elementos no nulos de los factores triangulares.

3.3. Resumen

En el presente capítulo se ha descrito el desarrollo de un prototipo de biblioteca para resolver problemas de reducción de modelos dispersos capaz de explotar los beneficios de arquitecturas híbridas compuestas por procesadores mutli-núcleo y GPUs. El diseño del prototipo se basa en extender la biblioteca de control Lyapack, implementada en MATLAB, por medio de las llamadas *User Supplied Functions* (USFs). De esta forma, es posible incluir cómputo basado en GPU sin realizar grandes cambios en el diseño original de la biblioteca, manteniendo las características de la misma en cuanto a la sencillez de su interfaz, así como su facilidad de uso. La interfaz Mex de MATLAB permite la interoperabilidad entre las rutinas de Lyapack y las rutinas aceleradas con GPU.

Como parte del trabajo, se han implementado USFs para tres familias distintas de matrices dispersas, a saber, matrices tri-diagonales, generales banda, y dispersas. En el caso de las matrices

tri-diagonales se realizó la implementación de un resolutor paralelo para sistemas de ecuaciones lineales basado en reducción cíclica y su integración con Lyapack. En el caso de las matrices banda se desarrolló un conjunto de nuevas rutinas híbridas (CPU-GPU) para explotar el poder de cómputo de este tipo de arquitecturas en la solución de sistemas lineales banda. Específicamente, en este capítulo se presentó un algoritmo híbrido "a bloques" para realizar la factorización LU de la matriz de coeficientes junto con una versión del algoritmo que incorpora técnicas de *look-ahead*, otra rutina para resolver los sistemas lineales generados por de la factorización, y una rutina que fusiona una etapa de las operaciones anteriores logrando un interesante ahorro en el tiempo de ejecución. En el caso de las matrices dispersas se describe la implementación de un resolutor iterativo para sistemas de ecuaciones lineales basado en los métodos del Gradiente Conjugado y Gradiente Conjugado Estabilizado, ya que la utilización de métodos directos para este tipo de problemas presenta importantes limitaciones en cuanto a memoria y tiempo de ejecución al trabajar con instancias de gran tamaño. Sin embargo, una de las desventajas de los métodos iterativos es su suceptibilidad a errores numéricos, por lo que frecuentemente es necesario el uso de precondicionadores. En este sentido se trabajó sobre ILUPACK, un paquete de métodos iterativos para resolver sistemas de ecuaciones lineales que incluye un poderoso precondicionador multinivel, con el objetivo de acelerar la aplicación de dicho precondicionador en arquitecturas híbridas. Se presentaron dos nuevas implementaciones que aceleran en GPU el método del Gradiente Conjugado disponible en ILUPACK, computando las operaciones más costosas durante la etapa de aplicación del precondicionador multinivel en el procesador gráfico. La primera variante computa la solución de sistemas triangulares y productos matriz-vector en la GPU, mientras que la segunda variante realiza la aplicación del precondicionador completamente en la GPU, disminuyendo el volumen de transferencias de datos entre las memorias de la CPU y GPU.

Capítulo 4

Evaluación Experimental

En este capítulo se describe la evaluación experimental de la solución presentada en el Capítulo 3. En la Sección 4.1 se describen las plataformas de cómputo sobre las que se realizaron los experimentos. Tras esta descripción, en la Sección 4.2 se detalla la evaluación de la propuesta para cada familia de USFs implementada, cada una asociada a distintos tipos de matrices (tri-diagonales, banda y dispersas generales).

Para el caso de la evaluación de las USFs implementadas para matrices tri-diagonales, la evaluación se centra en la resolución de sistemas de ecuaciones tri-diagonales que, como se podrá apreciar más adelante, es la operación fundamental del algoritmo LRCF-ADI para este tipo de matrices. Dicha evaluación se realiza sobre dos casos de prueba, uno que requiere trabajar con aritmética real, y otro que exige el uso de aritmética con números complejos.

Respecto a la evaluación experimental de la implementación del conjunto de USFs de Lyapack para matrices banda generales, se analiza por separado el desempeño computacional de las nuevas rutinas para la factorización LU de matrices banda (GBTRF+M y GBTRF+LA) así como dos implementaciones del resolutor triangular banda que se describió en el Apartado 3.2.2 del Capítulo 3: una destinada a ejecutarse en CPU, y la otra destinada a ejecutarse en el acelerador (GBTRS+M_{CPU} y GBTRS+M_{GPU}). El desempeño computacional de estas rutinas se compara con el de las rutinas análogas en la versión 11.1 de la biblioteca Intel MKL ($\text{GBTRF}_{\text{Intel}}$ y $\text{GBTRS}_{\text{Intel}}$). Además se analiza el impacto de estas nuevas rutinas sobre el algoritmo utilizado para resolver la ecuación de Lyapunov para un caso de reducción de modelos.

Por su parte, la evaluación para el caso de matrices dispersas generales se realizó en dos etapas. Yendo de lo más particular hacia lo más general, se evaluó primero la nueva rutina desarrollada en GPU para la aplicación del precondicionador multinivel de la biblioteca ILUPACK. Los resultados obtenidos se contrastan con la versión secuencial de dicha herramienta (v2.1), tanto en tiempo de ejecución como en la calidad de la solución obtenida, para asegurar que las nuevas rutinas no provocan el deterioro de la calidad del precondicionador. Seguidamente, se evalúa la implementación de los resolutores dispersos incorporados en Lyapack sobre casos de reducción de modelos. Las pruebas realizadas buscan proporcionar un panorama sobre las distintas alternativas al utilizar métodos iterativos y precondicionadores en conjunto con el algoritmo LRCF-ADI. Con este objetivo se evaluó la solución de la ecuación de Lyapunov para un caso de reducción de modelos, resolviendo los sistemas lineales dispersos mediante el resolutor iterativo en GPU sin precondicionador, utilizando una estrategia clásica de precondicionado (ILU0) con GPU implementada en la biblioteca CUSPARSE, y utilizando el nuevo precondicionador de ILUPACK acelerado con GPUs.

Todas estas secciones comienzan con una descripción de los casos de prueba utilizados.

4.1. Descripción de las plataformas de experimentación

La evaluación se realizó utilizando dos plataformas de hardware distintas, equipadas con tarjetas gráficas NVIDIA de generaciones recientes ("Kepler" en el caso de Buenaventura y "Fermi" en el caso de Enrico). Los procesadores gráficos utilizados pertenecen a la línea "Tesla" fabricada por NVIDIA cuyos diseños se encuentran orientados hacia el área de HPC. Entre otras características, los picos de rendimiento en aritmética de doble precisión de estos modelos son ampliamente superiores a los de los procesadores de las gamas "GTX"(orientada al mercado de los videojuegos) o "Quadro"(orientadas a aplicaciones de visualización profesional).

La tarjeta perteneciente a la arquitectura "Kepler" (generación que sucede a "Fermi") es una NVIDIA Tesla K20. Cuenta con 2496 núcleos CUDA que funcionan a una frecuencia de reloj de 706 MHz, alcanzando un pico teórico para aritmética de doble precisión de 1.17 Tflops. Además presenta un ancho de banda de memoria de 208 Gbytes/s. Por su parte, la tarjeta en Enrico es una NVIDIA Tesla C2070, con 448 núcleos CUDA que funcionan a una frecuencia de 1.15 GHz, presentando un pico teórico de cálculo en doble precisión de 515 Gflops. El ancho de banda de memoria es de 144 Gbytes/s.

Las características principales de ambas plataformas se describen en la Tabla 4.1.

Se utilizó la versión R2011b de MATLAB (que hace uso de una versión reciente de la biblioteca Intel MKL) y los compiladores GNU `gcc` 4.4 para los códigos en C; y NVIDIA CUDA 5.x la compilación y ejecución de los códigos en CUDA.

Para los experimentos realizados con los casos banda se utilizó la versión 11.1 de la biblioteca Intel MKL.

	Buenaventura	Enrico
GPU	Nvidia K20	Nvidia S2070
CUDA cores	2,496	448
Frequency (GHz)	0.71	1.15
GPU memory (GB)	6	5
CPU	Intel i3-3220	Intel i7-2600
CPU cores	2	4
Frequency (GHz)	3.4	3.3
L3 cache (MB)	3	8
Memory (GB)	16	8
O.S.	CentOS Rel. 6.4	CentOS Rel. 6.2
C/Fortran	`gcc` v4.4.7	`gcc` v4.4.6
CUDA/CUBLAS	5.0	5.5
Intel MKL	11.1	11.1

Tabla 4.1: Hardware y software empleado para los experimentos.

Todos los experimentos se realizaron utilizando el estándar IEEE de aritmética en coma flotante de doble precisión. Para el caso tri-diagonal también se utilizaron números complejos con doble precisión.

4.2. Evaluación de USFs para matrices tri-diagonales

La siguiente evaluación solo considera la etapa del proceso correspondiente a la resolución de las ecuaciones de Lyapunov, dejando de lado el costo relacionado con el cálculo de los parámetros de desplazamiento ya que, para este caso, este tiempo es poco importante en comparación con el del método LRCF-ADI.

Para lograr una comparación justa, todos los tiempos incluyen las transferencias de datos entre CPU y GPU para las rutinas aceleradas con CUDA, mientras que la versión original de Lyapack es ejecutada utilizando el poder de cómputo completo del host, en este caso Enrico (un procesador Intel con 4 hilos, uno por cada núcleo).

4.2.1. Casos de prueba

- **Benchmark** HEAT**.** Este ejemplo modela la difusión de calor en un filamento delgado (unidimensional) con una única fuente de calor [29]. El sistema está parametrizado por un escalar cuyo valor, en este caso, es $\alpha = 0{,}1$. El dominio espacial se discretiza en segmentos de largo $h = \frac{1}{n+1}$. Existe una única fuente de calor ubicada en 1/3 del largo del filamento (es decir $m = 1$). Se han definido tres instancias del problema, que se diferencian únicamente en la dimensión de la discretización: $\mathrm{H_S}$, $\mathrm{H_M}$ y $\mathrm{H_L}$ para $n = 524{,}288$, 786,432 y 1,048,576, respectivamente.

- **Benchmark** CIRCUIT**.** Las matrices en este ejemplo modelan un circuito RLC (circuito que contiene una resistencia eléctrica, una bobina y un condensador) de n_0 secciones interconectado en cascada, resultando en un sistema con $n = 2n_0$ estados y una única entrada ($m = 1$) [46]. El sistema está parametrizado por los escalares $R = 0{,}1$; $\bar{R} = 1{,}0$; $C = 0{,}1$; y $L = 0{,}1$. Se evalúan tres instancias del problema, $\mathrm{C_S}$, $\mathrm{C_M}$ y $\mathrm{C_L}$, con la misma dimensión que los casos correspondientes del benchmark HEAT.

En todos los casos se utilizó el número por defecto de parámetros de desplazamiento ($l = 10$).

4.2.2. Experimentación

La parte superior de la Tabla 4.2 muestra el tiempo de ejecución (en segundos) requerido para resolver las ecuaciones de Lyapunov asociadas con las tres instancias del problema HEAT, empleando la rutina original y la nueva rutina implementada en CUDA, así como los factores de aceleración de la nueva rutina respecto a la original en la plataforma Enrico. Este ejemplo involucra solo aritmética de números reales, ya que ninguno de los parámetros de desplazamiento devueltos es un número complejo.

Los resultados resumidos en la Tabla 4.2 muestran que la nueva rutina supera a la original por un factor superior a $\times 10$, ilustrando claramente los beneficios de utilizar la GPU para la solución de los sistemas lineales tri-diagonales.

La rutina acelerada con GPUs y la original se diferencian únicamente en que la primera utiliza la GPU para resolver los sistemas de ecuaciones lineales tri-diagonales. Para medir con más precisión la contribución de esta operación particular en el costo total del algoritmo, la Tabla 4.3 muestra el porcentaje de tiempo dedicado a las diferentes etapas de ambas rutinas, donde se observa que el resolutor tri-diagonal se acelera en un factor de $\times 20$. Por lo tanto, mientras en la rutina original la solución de sistemas tri-diagonales requería aproximadamente un 95 % del tiempo total de ejecución, en la nueva rutina acelerada con GPUs este porcentaje desciende hasta aproximadamente un 50 %.

Problema	Original CPU	Lyapack+GPU	Speed-up
H_S	1.72	0.17	10.1
H_M	2.53	0.22	11.5
H_L	3.41	0.31	11.0
C_S	2.38	0.31	7.7
C_M	3.64	0.46	8.0
C_L	4.80	0.57	8.4

Tabla 4.2: Tiempo de ejecución (en segundos) y factores de aceleración obtenidos por el resolutor tri-diagonal para los casos de prueba HEAT y CIRCUIT en la plataforma Enrico.

Este resultado demanda claramente la aceleración en GPU de otras etapas del método, lo cual se considera como trabajo futuro.

Problema	Original CPU		Lyapack+GPU		Speed-up
	resolutor	Otros	resolutor	Otros	resolutor
H_S	95.4	4.6	49.3	50.7	20.2
H_M	96.1	3.9	52.1	47.9	21.1
H_L	95.6	4.4	48.4	51.6	21.6

Tabla 4.3: Distribución del tiempo (en porcentajes) entre la resolución de sistemas tri-diagonales (resolutor) y otras etapas (Otros) para el caso de prueba HEAT en la plataforma Enrico.

La parte inferior de la Tabla 4.2 muestra el tiempo de ejecución y la aceleración obtenida por la rutina que incorpora cálculo en GPU, cuando ésta se compara con la rutina original en CPU para los tres modelos evaluados del caso CIRCUIT. En este caso, Lyapack genera parámetros de desplazamiento complejos, por lo que es necesario emplear aritmética de números complejos durante el método LRCF-ADI. En este caso, la rutina acelerada con GPU también supera a la original, esta vez por un factor cercano a $\times 8$, que, además, crece junto con la dimensión del problema. Comparado con el benchmark anterior, los factores de aceleración son claramente inferiores, que puede atribuirse, por un lado, a un bajo desempeño de la rutina `tsgv` de CUSPARSE, pero más importante, a que el volumen de transferencias de datos se duplica. El desarrollo de una implementación para resolutor tri-diagonal propuesto por [1] que incorpore aritmética de números complejos es una de las líneas de trabajo futuro.

4.3. Evaluación de USFs para matrices banda

Al igual que en el caso tri-diagonal, la evaluación del desempeño para matrices banda generales y la comparación con las rutinas correspondientes en CPU se realizó principalmente en la plataforma Enrico, dado que la misma cuenta con una CPU más potente. Sin embargo, con el fin de evaluar desempeño de las nuevas rutinas en la plataforma Buenaventura, se utilizó la misma para estudiar la variante Merge del resolutor lineal y compararla con las versiones anteriores.

4.3.1. Casos de prueba

Los casos de prueba utilizados para la evaluación particular de las nuevas rutinas se componen por 6 sistemas lineales banda, de dimensiones n =12.800, 25.600, 38.400, 51.200, 64.000 y 76.800. Para cada dimensión, se generaron 3 instancias, que varían en el ancho de la banda, con $k_b = k_u = k_l$ =1 %, 2 % y 4 % de n.

La evaluación de las rutinas en la solución de ecuaciones de Lyapunov se realizó sobre dos instancias del problema RAIL de reducción de modelos, que forma parte de la colección de problemas Oberwolfach [54] (ver Tabla 4.4).

Problema	n	$k_u = k_l$	#no-zeros	m
RAIL_S	5,177	139	35,185	7
RAIL_L	20,209	276	139,233	7

Tabla 4.4: Instancias del problema RAIL empleadas en la evaluación extraido de la colección Oberwolfach de problemas de reducción de modelos.

Las matrices de la familia RAIL son dispersas pero no presentan una estructura banda, por lo que fueron reordenadas mediante el método *Reverse Cuthill-McKee* [31] (RCM) para que adoptaran dicha estructura.

4.3.2. Experimentación

La evaluación experimental del caso de banda se realizó en tres etapas. En primer lugar, se evaluaron por separado las rutinas correspondientes al nuevo resolutor de banda. En segundo lugar, se evaluó el desempeño de este resolutor aplicado a un problema de reducción de modelos para una matriz banda, donde el nuevo resolutor es utilizado para resolver los sistemas lineales (desplazados) que surgen durante la obtención de los factores de bajo rango de los gramianos del sistema mediante el algoritmo LRCF-ADI. Por último se evaluó el desempeño del resolutor que realiza la resolución del primer sistema triangular de forma concurrente con la factorización LU. Esta rutina se evaluó únicamente de forma individual y la medición de su impacto sobre problemas de reducción de modelos y, en particular, sobre la resolución de ecuaciones de Lyapunov mediante el algoritmo LRCF-ADI se considera trabajo futuro.

Evaluación del resolutor banda propuesto

La Tabla 4.5 compara las tres versiones de la factorización LU para matrices banda: las dos versiones híbridas implementadas (GBTRF+M y GBTRF+LA) y la rutina GBTRF de la biblioteca MKL ($\text{GBTRF}_{\text{Intel}}$).

Se evaluaron distintos tamaños de bloque (b) para cada rutina pero, para no extender demasiado esta sección, sólo se incluyen los resultados correspondientes a los mejores tamaños de bloque evaluados en cada caso. Los resultados demuestran un desempeño superior de las nuevas implementaciones cuando el volumen de cómputo es grande. Concretamente, ambas implementaciones híbridas superan a la biblioteca MKL para matrices grandes, manteniendo resultados competitivos en los casos pequeños. Esto es de esperar, ya que las rutinas híbridas implican cierta sobrecarga de comunicación que es compensada solamente cuando el problema excede cierta dimensión. En concreto, GBTRF+M y GBTRF+LA son superiores a la rutina correspondiente de MKL para la factorización de matrices con n >25,600 y k_b =2 % de n. Cuando n >51,200, las nuevas variantes superan

a la biblioteca MKL incluso cuando k_b =1 % de n. Para el caso de mayor dimensión, n =76,800, los factores de aceleración obtenidos por GBTRF+LA respecto a la rutina de MKL son de ×2.0, ×3.9 y ×5.5 para k_b =1, 2 y 4 % de n respectivamente. El desempeño obtenido por GBTRF+M es considerablemente menor, reportando factores de aceleración de ×1.9, ×3.5 y ×5.0 para los mismos problemas.

Matriz Dimensión	Ancho de banda $k_b = k_u = k_l$	$\text{GBTRF}_{\text{Intel}}$	GBTRF+M	GBTRF+LA
12,800	1 %	0.066	0.174	0.180
	2 %	0.142	0.240	0.245
	4 %	0.385	0.358	0.341
25,600	1 %	0.313	0.482	0.493
	2 %	0.786	0.701	0.691
	4 %	3.397	1.339	1.231
38,400	1 %	0.684	0.867	0.844
	2 %	2.588	1.502	1.393
	4 %	11.742	3.517	3.407
51,200	1 %	1.898	1.537	1.399
	2 %	6.989	3.131	2.496
	4 %	31.745	7.217	6.627
64,000	1 %	3.104	2.175	2.029
	2 %	12.241	4.465	4.053
	4 %	52.701	12.796	11.660
76,800	1 %	5.749	3.044	2.808
	2 %	24.490	6.914	6.286
	4 %	103.264	20.462	18.769

Tabla 4.5: Tiempo de ejecución (en segundos) para la factorización LU de matrices de banda en la plataforma ENRICO.

Adicionalmente, se comparó la solución de un sistema lineal banda, con un solo vector como término independiente ($m = 1$), utilizando el resolutor banda triangular de MKL, $\text{GBTRS}_{\text{Intel}}$, y las dos implementaciones propuestas para la rutina GBTRS+M (GBTRS+M_{CPU} y GBTRS+M_{GPU}). En este escenario, los tiempos de ejecución son comparables aunque, en general, el desempeño de la biblioteca MKL mostró ser ligeramente mejor. Es importante notar que la mejora de estas rutinas debería ser más importante en los casos en que el término independiente del sistema está formado por varias columnas, es decir, el mismo sistema es resuelto para varios términos independientes ($m > 1$). Este es el caso en diversas aplicaciones de la ingeniería y, en particular, en el algoritmo LRCF-ADI para la solución de las ecuaciones de Lyapunov. También es posible que la implementación de la rutina $\text{GBTRS}_{\text{Intel}}$ difiera de la presentada en la Sección 3.2.2, ya que MKL no es una biblioteca de código abierto, por lo que no se tiene acceso a los detalles sobre su implementación. Particularmente, es probable que MKL utilice rutinas de BLAS-3 para el resolutor triangular banda, lo cual podría explicar la similitud entre el desempeño de MKL y el de las nuevas rutinas.

Considerando los resultados y el reducido impacto de GBTRS el tiempo total del resolutor, se utilizó finalmente la rutina de MKL para la solución de los sistemas triangulares banda. En la Figura 4.1 se aprecian las aceleraciones alcanzadas por la mejor de las rutinas CPU-GPU para la

factorización LU (izquierda) y para el resolutor banda en su totalidad (derecha). En ambos casos la referencia para calcular el factor de aceleración son las rutinas de MKL $\text{GBTRF}_{\text{Intel}}$ y $\text{GBTRS}_{\text{Intel}}$. Como se mencionó anteriormente, la mayor parte de las operaciones de punto flotante corresponden a la factorización por lo que, como se observa claramente en las figuras, los factores de aceleración obtenidos por la factorización y por el resolutor en su totalidad son muy similares.

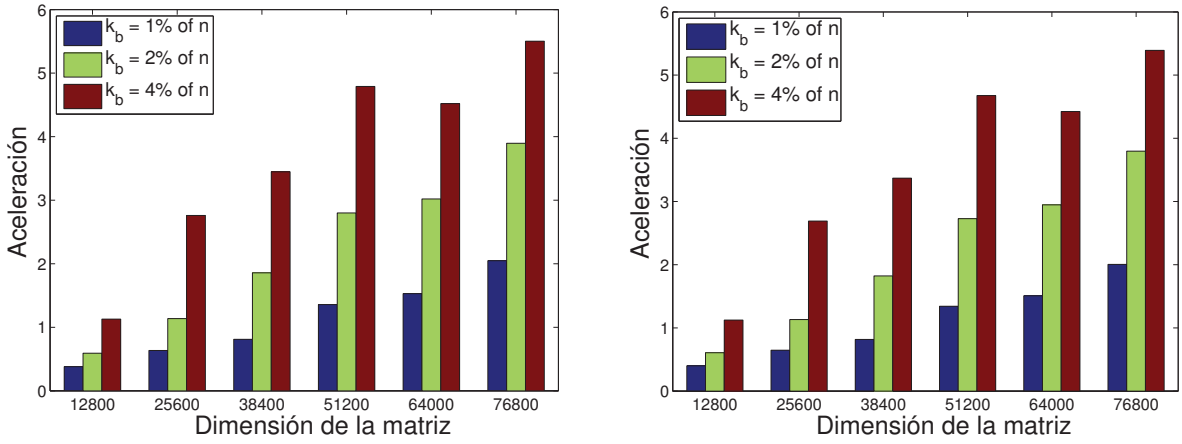

Figura 4.1: Aceleración de las nuevas rutinas híbridas contra la implementación de referencia en MKL para la factorización LU (izquierda) y para el resolutor en su totalidad (derecha) en la plataforma Enrico.

Aplicación al problema de reducción de modelos

Como sucede en el caso de las matrices tri-diagonales, la operación principal del método LRCF-ADI, en cuanto a tiempo de ejecución, corresponde a la solución de sistemas de ecuaciones con un término independiente formado por una pequeña cantidad de columnas (m). Por lo tanto, igual que antes, la aplicación del nuevo resolutor para el caso banda debería acelerar significativamente la resolución de las ecuaciones de Lyapunov.

En la Tabla 4.6 pueden observarse los tiempos de ejecución obtenidos por el resolutor de ecuaciones de Lyapunov basado en MKL y por el resolutor basado en las nuevas rutinas híbridas CPU-GPU. Los resultados muestran que el nuevo resolutor para las ecuaciones de Lyapunov presenta un mejor desempeño que la rutina basada en MKL para ambos problemas, con aceleraciones de $\times 2.23$ para la instancia de menor dimensión y $\times 3.14$ el caso más grande.

Problem	MKL solver	GPU-based solver	Speed-up
RAIL_S	2.34	1.05	2.23
RAIL_L	17.71	5.65	3.14

Tabla 4.6: Tiempo de ejecución (en segundos) y aceleración obtenida utilizando los solvers híbridos (CPU-GPU) en la solución de la ecuación de Lyapunov mediante el algoritmo LRCF-ADI en comparación con sus respectivas implementaciones basadas en MKL, en la plataforma Enrico.

También cabe señalar que el nuevo resolutor de Lyapunov para matrices banda no solamente supera al basado en MKL en la etapa de factorización LU, sino también en la solución de los sistemas

triangulares banda. La razón radica en que, en este caso, el término independiente está formado por m =7 vectores columna, lo que provoca un mejor desempeño de la rutina GBTRS+M_{CPU} sobre la correspondiente rutina de MKL para este tipo de problema.

Evaluación del resolutor MERGE

Los experimentos que conforman la evaluación la rutna MERGE se realizaron sobre la matriz $RAIL_{79841}$ de la colección Oberwolfach [54], ejecutando los experimentos en la plataforma BUENAVENTURA. Se emplearon 5 términos independientes B, cada uno con distinto número de columnas m, con el objetivo de reflejar las dimensiones que aparecen frecuentemente en problemas de reducción de modelos, como por ejemplo el algoritmo LRCF-ADI (ver Algoritmo 2) para resolver las ecuaciones de Lyapunov.

La Tabla 4.7 resume el tiempo de ejecución de tres de los resolutores presentados: el resolutor de la biblioteca MKL (encabezado LAPACK), el resolutor GBTRS+LA y la versión que combina las etapas de factorización y resolución, a la que nos referiremos como Merge.

m	LAPACK	GBTRS+LA_{GPU}	$Merge_{GPU}$
1	3.87	2.60	2.36
8	4.01	2.76	2.37
16	3.94	2.92	2.42
32	3.98	3.29	2.71
48	4.34	3.64	3.02
64	4.39	4.02	3.32
72	5.03	4.23	3.57

Tabla 4.7: Comparación de los tiempos de ejecución (en segundos) correspondientes a las distintas versiones del resolutor banda: rutina de LAPACK, rutina híbrida con look-ahead (GBTRS+LA_{GPU}) y la versión que fusiona la factorización con la resolución del primer sistema triangular ($Merge_{GPU}$) en la plataforma BUENAVENTURA.

Los resultados muestran que el último enfoque produce una reducción del tiempo de ejecución, superando a los dos resolutores restantes (MKL y GBTRS+LA) para todos los casos evaluados, incluso cuando el número de columnas m es pequeño.

4.4. Evaluación de USFs para matrices dispersas generales

Para la evaluación de este conjunto de USFs se utilizaron las dos plataformas de cómputo presentadas en la Sección 4.1.

4.4.1. Casos de prueba

La evaluación del resolutor de ecuaciones de Lyapunov para matrices dispersas generales se realizó sobre los siguientes problemas:

- **Benchmark** POISSON**.** Este caso modela una distribución 2-D de temperatura. La matriz de estados A del modelo se obtiene de la discretización de la ecuación de Poisson con el operador de 5 puntos en una malla $N \times N$, resultando en una matriz dispersa A tri-diagonal por bloques de dimensión N^2. Se generaron cuatro instancias de este problema, con $N = 450$,

724, 886 y 1024. En comparación con los otros dos problemas utilizados en esta sección, las matrices que componen este benchmark son las más pequeñas y dispersas, con un promedio de 5 elementos por fila.

- **Benchmark** LAPLACE**.** Este benchmark corresponde a una discretización en diferencias finitas de una ecuación en derivadas parciales que involucra el operador de Laplace. Las distintas instancias se fabricaron artificialmente con un generador. Se generaron seis matrices de distinta dimensión correspondientes a este problema. En comparación con los demás problemas de esta sección, el tamaño de las matrices de este benchmark va de mediano a grande, con un promedio de 7 elementos no nulos por fila.

- **Benchmark UFSMC.** Dado que los casos anteriores son artificiales, se seleccionaron tres matrices simétricas y definidas positivas de la *University of Florida Sparse Matrix Collection* (UFSMC), correspondientes a la discretización en elementos finitos de problemas estructurales que aparecen en mecánica. La matriz *G3_circuit* corresponde a la simulación de un circuito, la matriz *thermal2* corresponde a la discretización mediante el método de elementos finitos, de un problema de análisis térmico en estado estable; mientras que la matriz *door* corresponde a un problema estructural. Las matrices *G3_circuit* y *thermal2* son medianas en comparación con los demás problemas de esta sección, con promedios de 5 y 7 elementos por fila respectivamente, mientras que la matriz *ldoor* es la de menor dimensión de las tres, así como la más densa, con un promedio de 44 elementos no nulos por fila.

Las matrices que componen cada benchmark se presentan en la Tabla 4.8, junto con su dimensión y cantidad de elementos distintos de cero.

	Matriz	Dimensión n	#no-zeros
Poisson	P450	202,500	1,010,700
	P764	524,176	2,617,984
	P886	784,996	3,921,436
	P1024	1,048,576	5,238,784
Laplace	A050	125,000	860,000
	A100	1,000,000	6,940,000
	A126	2,000,376	13,907,376
	A159	4,019,679	27,986,067
	A200	8,000,000	55,760,000
	A252	16,003,008	111,640,032
UFSMC	LDOOR	952,203	42,493,817
	THERMAL2	1,228,045	8,580,313
	G3_CIRCUIT	1,585,478	7,660,826

Tabla 4.8: Matrices utilizadas en la evaluación del caso SPD.

La evaluación de las distintas implementaciones realizadas del resolutor de ecuaciones de Lyapunov para matrices dispersas generales se realizó utilizando matrices de los tres benchmarks. En todos los casos, la matriz de entrada fue construida como $B = \left[e_1^T,\ 0_{1\times N(N-1)}\right]^T$, con $e_1 \in \mathbb{R}^N$ representando la primer columna de la matriz identidad.

4.4.2. Evaluación de la versión de ILUPACK acelerada con GPU

Tal como se mencionó, de forma previa a la evaluación de las USFs implementadas para matrices dispersas generales, se evaluó la nueva versión basada en GPU de ILUPACK. Para esta evaluación se utilizaron matrices correspondientes a los benchmark LAPLACE y UFSMC, en particular, las matrices A126 a A252 del benchmark LAPLACE y todas las matrices de UFSMC. En los sistemas lineales, todas las entradas del vector derecho b fueron inicializadas en 1, y el PCG fue inicializado con una solución $x_0 \equiv 0$. Para las pruebas, los parámetros que controlan la convergencia del proceso iterativo de ILUPACK, `restol`, se fijó en 10^{-8}.

La Tabla 4.9 compara la implementación original (basada en CPU) de ILUPACK con las dos versiones basadas en GPU (GPUV1 y GPUV2), utilizando todos los casos de prueba mencionados y las dos plataformas de cómputo (ENRICO y BUENAVENTURA). En los resultados se detalla el número de iteraciones que fueron requeridas para la convergencia ("#iter"); los tiempos de ejecución (en milisegundos) correspondientes a la resolución de sistemas LDL^T, `spmv`, transferencias de datos entre CPU y GPU , así como el tiempo total de la aplicación del precondicionador (encabezados como "LDL^T", "`spmv`", "Transf." y "Apl. Prec.", respectivamente); el tiempo total correspondiente al resolutor en milisegundos ("Total PCG time"); y el error relativo del residuo

$$\mathcal{R}(x^*) := \frac{\|b - Ax^*\|_2}{\|x^*\|_2},$$

donde x^* es la solución calculada.

Los resultados en la tabla muestran que ambas versiones basadas en GPU y la versión original en CPU requieren la misma cantidad de iteraciones para converger en todos los casos y plataformas evaluadas. Además, las diferencias entre los residuos obtenidos son extremadamente pequeñas, de una magnitud que puede ser explicada fácilmente por la inevitable introducción de errores de punto flotante. Estos resultados confirman que ambas versiones paralelas conservan las propiedades, tanto numéricas como de convergencia, del resolutor original de ILUPACK.

Desde el punto de vista del desempeño computacional, las nuevas versiones basadas en GPU exhiben una importante reducción en el tiempo de cómputo que requiere la aplicación del precondicionador, en relación con la versión original de ILUPACK para todos los casos de prueba y ambas plataformas.

Particularmente, la Figura 4.2 ilustra los factores de aceleración alcanzados por las versiones basadas en GPU respecto a la original en CPU. Las gráficas reportan que las nuevas versiones superan a la original en factores que varían entre $\times 1.25$ y $\times 5.84$ dependiendo del caso y la plataforma. Por otro lado, la variante GPUV2 claramente supera a la versión GPUV1 en la aplicación del precondicionador multinivel, incurriendo, en todos los casos y plataformas, en un menor sobrecosto debido a la disminución de las transferencias de datos. Más aún, GPUV2 obtiene importantes aceleraciones de hasta $5.84\times$ y $2.92\times$, cuando se compara con la versión en CPU y con GPUV1 respectivamente, para el problema más grande en BUENAVENTURA. Los resultados también muestran mayor escalabilidad para GPUV2.

Finalmente, considerando los resultados desde el punto de vista de las plataformas de hardware, si se enfoca el análisis en la versión GPUV2, la plataforma equipada con una GPU "Fermi" ofrece tiempos de ejecución menores que la equipada con GPU "Kepler" únicamente para el caso más pequeño (LDOOR). Esto es debido a que la arquitectura "Fermi" (448 núcleos CUDA a 1.15 GHz) es más apropiada cuando el problema presenta un menor grado de concurrencia. En otros casos, la plataforma basada en "Kepler" muestra un mejor desempeño.

Plataforma	Matriz	Variante	#iter	LDL^T	spmv	Transf.	Apl. Prec.	Total PCG	$\mathcal{R}(x^*)$
Buenaventura	ldoor	CPU	200	101,713	8,646	-	114,548	136,040	4.845e-07
		GPUV1	200	74,988	1,327	3,154	83,683	103,480	4.963e-07
		GPUV2	200	74,805	1,191	1,074	77,291	96,880	4.963e-07
	thermal2	CPU	186	18,149	5,871	-	30,097	40,060	2.397e-08
		GPUV1	186	7,712	836	3,948	18,751	25,650	2.392e-08
		GPUV2	186	7,525	687	1,263	9,792	19,850	2.392e-08
	G3_Circuit	CPU	75	8,364	3,120	-	14,844	18,290	2.674e-06
		GPUV1	75	3,698	465	2,026	9,568	13,010	2,674e-06
		GPUV2	75	3,603	385	645	4,787	8,200	2.674e-06
	A126	CPU	44	15,122	3,442	-	21,056	23,660	5.143e-09
		GPUV1	44	10,405	498	1,521	14,921	17,580	5.143e-09
		GPUV2	44	10,359	444	469	11,381	14,010	5.143e-09
	A159	CPU	52	37,097	8,616	-	51,973	58,220	2.751e-09
		GPUV1	52	17,430	1,128	3,638	28,466	34,810	2.751e-09
		GPUV2	52	17,351	1,063	1,094	19,751	26,080	2.751e-09
	A200	CPU	76	70,275	30,370	-	124,123	142,880	5.618e-09
		GPUV1	76	20,843	3,673	11,225	59,501	77,850	5.618e-09
		GPUV2	76	20,688	3,532	3,172	28,276	47,020	5.618e-09
	A252	CPU	338	383,617	98,296	-	652,209	815,950	5.721e-08
		GPUV1	338	72,348	5,861	77,414	326,111	495,490	5.721e-08
		GPUV2	338	72,262	5,763	28,240	111,754	279,780	5.721e-08
enrico	ldoor	CPU	200	90,199	7,637	-	101,381	119,800	4.845e-07
		GPUV1	200	74,127	1,476	1,737	80,885	93,030	4.842e-07
		GPUV2	200	74,147	1,373	552	76,337	88,240	4.842e-07
	thermal2	CPU	186	16,021	5,126	-	26,617	34,730	2.397e-08
		GPUV1	186	11,070	1,141	2,179	19,858	28,190	2.302e-08
		GPUV2	186	10,948	987	658	13,024	21,450	2.302e-08
	G3_Circuit	CPU	75	7,138	2,725	-	12,762	15,340	2.674e-06
		GPUV1	75	4,645	616	1,107	9,271	12,920	2.674e-06
		GPUV2	75	4,610	546	339	5,704	8,380	2.675e-06
	A126	CPU	44	13,509	2,972	-	18,688	20,680	5.143e-09
		GPUV1	44	10,751	582	825	14,369	16,430	5.143e-09
		GPUV2	44	10,720	541	247	11,673	13,760	5.143e-09
	A159	CPU	52	32,648	7,424	-	45,671	50,410	2.751e-09
		GPUV1	52	19,680	1,287	1,909	28,489	32,400	2.751e-09
		GPUV2	52	19,814	1,234	575	22,022	26,010	2.751e-09
	A200	CPU	76	60,513	25,626	-	106,540	120,830	5.618e-09
		GPUV1	76	31,815	4,392	5,836	62,485	77,530	5.618e-09
		GPUV2	76	32,245	4,263	1,674	39,536	54,670	5.618e-09
	A252	CPU	338	264,697	74,812	-	476,384	602,160	5.721e-08
		GPUV1	338	106,430	6,625	39,375	290,054	419,590	5.721e-08
		GPUV2	338	106,944	6,527	14,795	136,918	266,690	5.721e-08

Tabla 4.9: Comparación de la versión original de ILUPACK con las dos versiones basadas en GPU en las plataformas Buenaventura y Enrico.

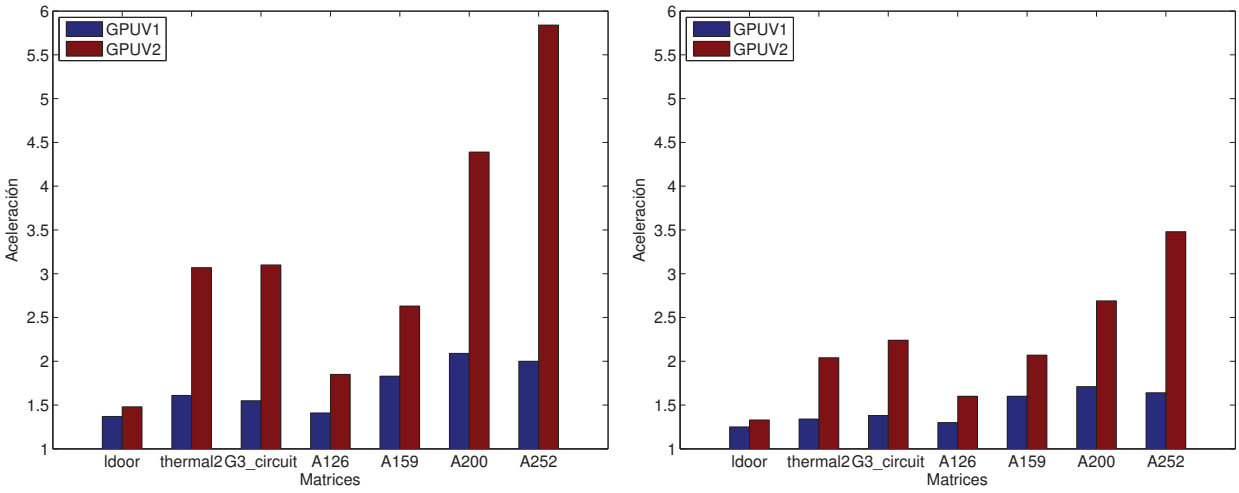

Figura 4.2: Factores de aceleración obtenidos para la aplicación del precondicionador, de las versiones basadas en GPU de ILUPACK, comparando con la versión secuencial basada en CPU, en las plataformas Buenaventura (izquierda) y Enrico (derecha).

Comparación con el precondicionador ILU0 de NVIDIA

Con el objetivo de comprobar la importancia del uso de técnicas de precondicionamiento al aplicar métodos iterativos para la solución de sistemas de ecuaciones lineales, y la efectividad del nuevo precondicionador multinivel de la biblioteca ILUPACK, se evaluó el desempeño de tres implementaciones del método del Gradiente Conjugado, la primera de ellas sin utilizar precondicionador, la segunda utilizando el precondicionador ILU0 de la biblioteca CUSPARSE, y la tercera utilizando la implementación del método PCG que se distribuye en la biblioteca ILUPACK, empleando la versión acelerada en GPU de su precondicionador multinivel.

La Tabla 4.10 ilustra los resultados de esta comparación en la plataforma Buenaventura, mientras que la Tabla 4.11 presenta los resultados de los mismos experimentos realizados en la plataforma Enrico. La columna "Variante" distingue entre las tres implementaciones propuestas del algoritmo del Gradiente Conjugado; la columna "#iter" presenta la cantidad de iteraciones que fueron necesarias para que el método converja; la columna "Tiempo PCG (seg.)" corresponde al tiempo de cómputo en segundos para todas las iteraciones del algoritmo; y la columna "$\mathcal{R}(x^*)$" corresponde al error relativo obtenido en la solución, que se calculó como $\mathcal{R}(x^*) = \frac{\|b-Ax^*\|_2}{\|b\|_2}$, donde x^* es la solución obtenida.

La primera observación que se desprende de los resultados, es que el precondicionador multinivel de la biblioteca ILUPACK es sumamente efectivo para acelerar la convergencia del método del Gradiente Conjugado, siendo la alternativa que presenta el menor número de iteraciones necesarias para alcanzar la convergencia en todos los casos evaluados. Sin embargo, el costo de aplicar este tipo de precondicionador es mucho más alto que el de aplicar otros más simples. En particular y como ejemplo, el costo por iteración para la matriz A159 es 0.08s en el caso de CUSPARSE y 0.45s en el caso de ILUPACK. Es por esto que los resultados muestran un mejor desempeño del precondicionador de CUSPARSE para el benchmark LAPLACE (matrices A050, A100, A126 y A159). Adicionalmente, las propiedades de las matrices utilizadas en dicho benchmark garantizan la convergencia del método iterativo utilizado en un número de iteraciones razonable y, por consiguiente, al no utilizar un precondicionador el costo de la iteración es aún menor. Sin embargo,

Matriz	Variante	#iter	Tiempo PCG (seg.)	$\mathcal{R}(x^*)$
A050	No prec.	134	0.07	8.18e-10
	Cusparse	58	0.31	1.12e-09
	ILUPACK	22	1.40	3.05e-11
A100	No prec.	263	0.70	8.76e-10
	Cusparse	110	2.69	7.68e-10
	ILUPACK	36	7.39	7.72e-10
A126	No prec.	327	1.68	9.86e-10
	Cusparse	132	5.84	8.81e-10
	ILUPACK	43	12.82	4.85e-10
A159	No prec.	395	4.00	9.14e-10
	Cusparse	165	13.65	8.31e-10
	ILUPACK	52	23.76	8.22e-10
THERMAL2	No prec.	4899	16.26	9.97e-10
	Cusparse	2020	92.66	9.87e-10
	ILUPACK	165	18.18	1.22e-09
G3_CIRCUIT	No prec.	8677	27.35	9.52e-10
	Cusparse	376	24.63	8.88e-10
	ILUPACK	72	8.41	5.09e-10
LDOOR	No prec.	10000	97.85	3.21e-06
	Cusparse	10000	1623.60	6.80e-03
	ILUPACK	165	68.67	2.93e-10

Tabla 4.10: Comparación entre tres implementaciones del método del Gradiente Conjugado, una implementación en GPU sin precondicionado, una implementación en GPU utilizando el precondicionador ILU0 de CUSPARSE, y la implementación de ILUPACK acelerada con GPU, en la plataforma BUENAVENTURA.

si bien la versión no precondicionada del método iterativo es la que obtiene mejores resultados en cuanto a tiempo de ejecución, es preciso notar que tanto el número de iteraciones necesarias para que el método converja (debido a la mayor cantidad de errores numéricos) como el costo de cada iteración, aumenta junto con la dimensión de los casos de prueba. Por este motivo, conforme aumenta la dimensión del problema, lograr que el método converja en una menor cantidad de operaciones comienza a ser una ventaja. Por ejemplo, es posible observar que para la matriz A050, la versión basada en la biblioteca ILUPACK es aproximadamente $\times 20$ más lenta que la versión no precondicionada, pero para la matriz A159 la diferencia es de únicamente $\times 6$.

La necesidad de utilizar precondicionadores queda netamente en evidencia para los últimos tres casos, extraidos de la UFSMC. En particular, el caso de la matriz LDOOR no logra converger dentro de las 10000 iteraciones estipuladas como máximo, no solo para la versión no precondicionada sino que tampoco lo logra para la versión precondicionada con el método ILU0. En cambio, la versión basada en ILUPACK produce un resultado en tan solo 165 iteraciones, con un error relativo del orden de 10^{-10}. La versión acelerada en GPU de ILUPACK también logra el mejor desempeño para el caso de la matriz G3_CIRCUIT, mientras que en el caso de la matriz THERMAL2 supera a la versión basada en el precondicionador ILU0, pero, pese a que toma 4899 iteraciones en converger, el bajo

Matriz	Variante	#iter	Tiempo PCG (seg.)	$\mathcal{R}(x^*)$
A050	No prec.	134	0.07	8.17e-10
	Cusparse	58	0.27	8.28e-10
	ILUPACK	22	0.51	3.05e-11
A100	No prec.	263	0.80	8.76e-10
	Cusparse	110	2.92	7.68e-10
	ILUPACK	36	7.72	7.72e-10
A126	No prec.	327	1.95	9.86e-10
	Cusparse	132	6.62	8.81e-10
	ILUPACK	43	19.02	4.85e-10
A159	No prec.	395	4.65	9.14e-10
	Cusparse	165	15.80	8.31e-10
	ILUPACK	52	47.13	8.21e-10
THERMAL2	No prec.	4900	19.04	9.94e-10
	Cusparse	2020	84.73	9.86e-10
	ILUPACK	165	29.97	1.22e-09
G3_CIRCUIT	No prec.	8695	34.11	9.99e-10
	Cusparse	376	21.49	8.88e-10
	ILUPACK	72	13.79	5.09e-10
LDOOR	No prec.	10000	95.73	3.54e-06
	Cusparse	10000	1707.10	4.60e-03
	ILUPACK	165	77.94	2.94e-10

Tabla 4.11: Comparación entre tres implementaciones del método del Gradiente Conjugado, una implementación en GPU sin precondicionado, una implementación en GPU utilizando el precondicionador ILU0 de CUSPARSE, y la implementación de ILUPACK acelerada con GPU, en la plataforma ENRICO.

costo de cada iteración hace que la versión sin precondicionador obtenga el tiempo de ejecución más bajo.

Es importante mencionar que la comparación no es del todo justa, ya que si bien la aplicación del precondicionador es la etapa más importante desde el punto de vista de tiempo computacional, la multiplicación `spmv` del Gradiente Conjugado es también importante, sobre todo en los casos que vinculan sistemas de grandes dimensiones. Esta operación se realiza en GPU en el caso del resolutor implementado con el precondicionador de CUSPARSE, mientras que en el caso de ILUPACK, la misma se realiza en CPU. Trasladar esta operación a GPU, así como otras operaciones del resolutor de la biblioteca ILUPACK se considera como trabajo futuro.

Las diferencias entre los resultados obtenidos para la plataformas BUENAVENTURA y ENRICO son mínimas. Para las implementaciones basadas en la biblioteca CUSPARSE, tanto la versión no precondicionada como la que utiliza ILU0, los tiempos de ejecución son ligeramente menores en la plataforma BUENAVENTURA para las matrices más grandes del benchmark LAPLACE, mientras que para las matrices de la UFSMC, en especial para la matriz THERMAL2, se obtuvo un menor tiempo de ejecución en la plataforma ENRICO. En lo que respecta a la implementación basada en ILUPACK, se observa un mejor desempeño al ejecutar en la plataforma ENRICO (comparado

con el registrado para la plataforma Buenaventura) para las dos instancias más pequeñas del benchmark LAPLACE (aunque para la matriz A100 los tiempos de ejecución son prácticamente iguales), y un peor desempeño para todas las demás instancias. Esto se debe a que, si bien la tarjeta gráfica en la plataforma Enrico tiene un menor número de núcleos CUDA que la GPU K20 de la plataforma Buenaventura, cada uno de estos núcleos es más potente, por lo que, para instancias pequeñas, en las que no se alcanza a explotar el paralelismo que ofrece la tarjeta K20, la tarjeta C2070 de la plataforma Enrico resulta más adecuada.

En síntesis, la comparación realizada refleja que, si bien la aplicación del precondicionador de ILUPACK tiene un costo más alto que la del precondicionador ILU0 de NVIDIA, debido a su mayor complejidad, también produce una sensible reducción en la cantidad de iteraciones. Esto provoca que en muchos casos, especialmente en aquellos de gran dimensión y problemas mal condicionados, el costo total del método PCG con el precondicionador de ILUPACK sea menor que con ILU0.

4.4.3. Evaluación de Reducción de Modelos dispersos generales

Para evaluar el desempeño de la solución sobre problemas de reducción de modelos para el caso de matrices dispersas generales se realizaron diversos experimentos, específicamente sobre matrices simétricas y definidas negativas. Para este tipo de matriz, los valores de Ritz producidos por la iteración de Arnoldi que se describe en el Capítulo 2 son reales; por lo tanto no hay necesidad de operar con aritmética compleja.

Las pruebas realizadas intentan contrastar las ventajas y desventajas de las distintas estrategias que se han seguido para resolver problemas de reducción de modelos dispersos de gran dimensión utilizando Lyapack:

- Utilizando la versión original de Lyapack.
- Utilizando un método iterativo (PCG) precondicionado implementado en GPU para resolver los sistemas de ecuaciones lineales, aplicando como precondicionador la factorización LU incompleta sin llenado (ILU0) disponible en la biblioteca CUSPARSE.
- Utilizando un método iterativo (PCG) precondicionado implementado en CPU y disponible como parte de la biblioteca ILUPACK, para resolver los sistemas de ecuaciones lineales, y como precondicionador la versión acelerada con GPU del precondicionador multinivel de ILUPACK.

Las Tablas 4.12 y 4.13 presentan los resultados obtenidos al resolver una ecuación de Lyapunov de tipo $AX + XA^T + BB^T = 0$, donde A es una de las matrices que componen el benchmark POISSON y B es una matriz de una única columna, sobre las plataformas Buenaventura y Enrico respectivamente. Las filas de ambas tablas se encuentran divididas en tres secciones correspondientes a las distintas implementaciones del resolutor de ecuaciones de Lyapunov para matrices dispersas simétricas y definidas negativas que fueron evaluadas: “Lyapack Original” denota la implementación original en MATLAB de la biblioteca Lyapack; “Lyapack + ILUPACK” refiere a la implementación basada en la biblioteca ILUPACK acelerada con GPU y “Lyapack + CUSPARSE” corresponde a la implementación basada en la biblioteca CUSPARSE. En la columna “Dimensión” se especifica la dimensión del caso de prueba; en la columna “Preproc.” se presenta el tiempo de ejecución insumido por la etapa de preprocesamiento de los sistemas de ecuaciones lineales que, en el caso de la implementación original de la biblioteca Lyapack corresponde a la factorización LU de las matrices para cada parámetro de desplazamiento, y en el caso de las otras dos implementaciones corresponde al cálculo del precondicionador; “lp_param” denota la columna que presenta los tiempos de ejecución correspondientes a la heurística que calcula los parámetros de desplazamiento;

la columna "lrcf_adi" presenta el tiempo de ejecución correspondiente a la rutina que resuelve la ecuación de Lyapunov; "Total sistemas lineales" presenta la suma de tiempo de ejecución dedicada a resolver sistemas lineales (sin contar la etapa de preprocesamiento) y "Total" denota el tiempo de ejecución total (considerando las rutinas `lp_param` y `lrcf_adi`).

versión	Matriz	Preproc.	lp_param	lrcf_adi	Total sistemas lineales	Total
Lyapack Original	P450	5.44	3.31	3.26	5.55	12.00
	P724	17.47	9.55	9.65	16.70	36.67
	P886	36.15	15.58	18.19	29.69	69.92
	P1024	90.80	33.60	526.30	549.00	650.70
Lyapack + ILUPACK	P450	1.56	22.57	4.91	26.50	29.05
	P724	4.14	42.99	14.80	55.26	61.94
	P886	6.22	79.79	29.47	105.54	115.48
	P1024	8.57	97.98	36.83	129.47	143.39
Lyapack + CUSPARSE	P450	0.02	21.86	4.92	25.82	26.78
	P724	0.04	78.76	14.58	90.74	93.35
	P886	0.07	139.01	24.35	159.66	163.36
	P1024	0.10	211.03	36.09	241.99	247.12

Tabla 4.12: Tiempo de ejecución (en segundos) de Lyapack para los modelos dispersos generales del benchmark POISSON en la plataforma Buenaventura.

versión	Matriz	Preproc.	lp_param	lrcf_adi	Total sistemas lineales	Total
Lyapack Original	P450	4.00	2.21	2.26	3.95	8.48
	P724	12.23	6.51	6.65	11.57	25.40
	P886	22.82	10.64	11.22	19.56	44.69
	P1024	29.07	14.14	16.15	27.19	59.36
Lyapack + ILUPACK	P450	1.41	15.29	5.45	20.16	20.73
	P724	3.74	61.52	17.85	77.71	79.37
	P886	5.63	109.48	36.47	143.51	145.95
	P1024	7.74	171.54	45.92	214.18	217.47
Lyapack + CUSPARSE	P450	0.07	87.54	4.56	91.52	92.10
	P724	0.17	231.73	17.59	247.72	249.32
	P886	0.15	356.19	21.21	375.09	377.44
	P1024	0.18	518.70	32.07	547.62	550.77

Tabla 4.13: Tiempo de ejecución (en segundos) de Lyapack para los modelos dispersos generales del benchmark POISSON en la plataforma Enrico.

La primera observación que se desprende de los resultados presentados en la Tabla 4.12, es que la versión original de la biblioteca Lyapack obtiene los mejores resultados para todos los casos, exceptuando el de mayor dimensión. Esta biblioteca resuelve los sistemas lineales involucrados en el algoritmo LRCF-ADI de forma directa, por medio de la factorización de Cholesky y utilizando

bibliotecas altamente optimizadas de ALN en que se basa MATLAB para resolver operaciones matriciales de este tipo. Sin embargo, la utilización de métodos directos resulta una restricción cuando se trata de sistemas de grandes dimensiones, ya que este enfoque requiere grandes cantidades de memoria y, a partir de cierta dimensión, como puede apreciarse en el último caso, los tiempos de ejecución crecen dramáticamente.

La segunda observación inmediata que es posible realizar a partir del análisis de los resultados es que, exceptuando el caso de menor dimensión, el desempeño de la implementación del resolutor de Lyapunov basada en la biblioteca ILUPACK acelerada con GPU supera a la implementación basada en la biblioteca CUSPARSE. La diferencia principal entre el desempeño de ambas implementaciones debe observarse en la columna "lp_param" ya que es únicamente en esta rutina donde se resuelven sistemas lineales por medio de la biblioteca ILUPACK. Como se describe en el Capítulo 2, los sistemas lineales que aparecen en la rutina "lrcf_adi" se encuentran desplazados; es decir, tienen la forma $(A + \sigma_i I)X = B$ donde σ_i es uno de los parámetros calculados por la rutina "lp_param". Esto significa que para resolver estos sistemas utilizando un método iterativo precondicionado debe seguirse alguna de las siguientes estrategias:

- Calcular un precondicionador distinto para cada parámetro de desplazamiento y transferirlo a la memoria de la GPU en el momento adecuado.
- Utilizar el mismo precondicionador para un subconjunto o todos los sistemas desplazados.
- Actualizar de alguna forma el precondicionador calculado para la matriz A de forma que refleje los distintos desplazamientos de la misma.
- No utilizar precondicionadores.

El primer enfoque implica un alto consumo de memoria y tiempo de cómputo. Por un lado, el costo de calcular cada precondicionador es elevado, y por el otro, el costo de transferirlo a la memoria de la GPU es alto también. Dados los requerimientos de almacenamiento, sería inviable, en la mayoría de los casos, alojar todos los precondicionadores en la memoria del dispositivo de forma simultánea, por lo que deberían transferirse a demanda o según alguna estrategia que resulte pertinente para ocultar el costo de la transferencia. Dado que se trabajó con matrices de grandes dimensiones, en muchos de los casos de prueba ya se opera al límite de la capacidad de memoria de la GPU, por lo que este enfoque se descartó. El segundo enfoque se evaluó sin éxito, ya que si bien en algunos casos, utilizar el mismo precondicionador para varios sistemas desplazados reducía la cantidad de iteraciones necesarias para la convergencia, en otros casos casi no lograba reducirlas o incluso las aumentaba levemente. En el caso del precondicionador de la biblioteca ILUPACK, dado el relativamente alto costo computacional de su aplicación, este enfoque es inútil si no se logra reducir significativamente la cantidad de iteraciones necesarias para la convergencia del método iterativo. Dado que el tercer enfoque implica una reformulación matemática del precondicionador para el caso de sistemas desplazados, el mismo queda fuera del alcance de esta tesis y se plantea como trabajo futuro. Por estos motivos, para resolver los sistemas de ecuaciones lineales desplazados se utilizó, tanto en la implementación basada en la biblioteca ILUPACK como en la basada en CUSPARSE, la versión no precondicionada del método del Gradiente Conjugado implementado en GPU. Adicionalmente, se observó que, por lo general, las matrices desplazadas mejoran el número de condición de la matriz original, requiriéndose menos iteraciones para lograr la convergencia del método iterativo.

Pueden observarse algunas pequeñas diferencias en los tiempos de ejecución de la rutina `lrcf_adi` entre las implementaciones basadas en CUSPARSE e ILUPACK, pese que ambas utilizan el mismo resolutor para los sistemas lineales desplazados. Estas diferencias se deben a que, dado que el

Instancia	Memoria (GBytes)
P450	1.2177
P724	3.5915
P886	5.9025
P1024	7.8895

Tabla 4.14: Consumo de memoria del resolutor de Lyapack para benchmark POISSON.

resolutor utilizado en la rutina que calcula los parámetros de desplazamiento sí es distinto, esto provoca que la solución de los sistemas lineales no sea exactamente igual y, por tanto, los parámetros de desplazamiento presentan ligeras diferencias. Esto provoca que las matrices desplazadas tengan distinto número de condición en uno y otro caso, y que el PCG requiera una cantidad de iteraciones distinta para alcanzar la cota de error estipulada (la cual es la misma para ambos casos) para algunos de estos sistemas desplazados.

La Tabla 4.15 muestra los resultados obtenidos al resolver la ecuación de Lyapunov mediante el método LRCF-ADI, utilizando los resolutores iterativos mencionados anteriormente, para los casos de mayor dimensión (y número de elementos distintos de cero). Para estos casos de prueba no fue posible obtener resultados utilizando la versión original de Lyapack, dado que la alta demanda de memoria que presenta esta biblioteca provoca que MATLAB falle con el error "Out of Memory". En la Tabla 4.14 puede observarse el consumo de memoria que se registró para las distintas instancias del benchmark POISSON, mientras que la Figura 4.3 ilustra el crecimiento de los requerimientos de memoria según la dimensión del problema. Este consumo corresponde al almacenamiento de la matriz de coeficientes y de los factores de Cholesky de todos los sistemas lineales a resolver durante la ejecución del método LRCF-ADI para resolver la ecuación de Lyapunov. La biblioteca Lyapack calcula una factorización por cada uno de los parámetros de desplazamiento, almacenando todos los factores como variables globales. Esto provoca que la demanda de memoria se multiplique y se vuelva prohibitiva para algunos casos de gran dimensión. Para estos casos, la aplicación de resolutores basados en métodos iterativos muestra mejores resultados, ya que este tipo de métodos cuenta con la ventaja de mostrar un costo de almacenamiento más reducido.

En lo que respecta a las otras dos implementaciones del resolutor de Lyapunov para matrices dispersas generales, los resultados de la Tabla 4.15 muestran que la implementación basada en la biblioteca CUSPARSE presenta mejores tiempos de ejecución para los casos correspondientes al benchmark LAPLACE, mientras que la implementación basada en la biblioteca ILUPACK acelerada con GPU, obtiene un mejor desempeño para las matrices del benchmark UFSMC. Esto concuerda con lo presentado en la Sección 4.4.2 y se debe a que las matrices del benchmark LAPLACE tienen menor número de condición que las del benchmark UFSMC. El caso de la matriz LDOOR no se presenta en la tabla, dado que la versión no precondicionada del Gradiente Conjugado (utilizada en la rutina `lrcf_adi` para resolver los sistemas desplazados) no logra la convergencia dentro del máximo de iteraciones, por lo que la solución obtenida para la ecuación de Lyapunov no es correcta.

Los resultados obtenidos para las mismas instancias en la plataforma ENRICO pueden observarse en las Tablas 4.13 y 4.16. El mejor desempeño de la versión original de Lyapack se debe a que esta plataforma tiene un procesador de 4 núcleos, en lugar de 2 como es el caso de BUENAVENTURA. Sin embargo, las limitaciones de esta implementación en cuanto al consumo de memoria se mantienen en esta plataforma. Por otro lado, los peores tiempos de ejecución obtenidos por las

versión	Matriz	Preproc.	lp_param	lrcf_adi	Total sistemas lineales	Total
LYAPACK ORIGINAL	A50	130.0	20.0	1290.0	1300.0	1440.0
	A100	Out of memory				
LYAPACK + ILUPACK	A050	4.32	22.88	0.69	22.20	27.89
	A100	38.43	146.49	6.55	147.25	191.46
	A126	79.82	215.45	15.38	221.28	310.65
	A159	167.01	551.42	30.88	562.76	749.31
	thermal2	12.41	246.4705	105.67	346.14	364.56
	G3_circuit	17.57	190.1217	201.18	383.97	408.88
LYAPACK + CUSPARSE	A050	0.03	2.43	1.08	2.94	3.51
	A100	0.11	47.08	6.52	48.75	53.59
	A126	0.29	112.98	15.38	118.81	128.36
	A159	0.83	234.36	35.77	250.14	270.13
	thermal2	0.1269	763.78	105.62	863.39	869.40
	G3_circuit	0.1544	440.75	201.28	634.37	642.03

Tabla 4.15: Evaluación del resolutor de Lyapunov para matrices de los benchmark LAPLACE y UFSMC en la plataforma BUENAVENUTRA.

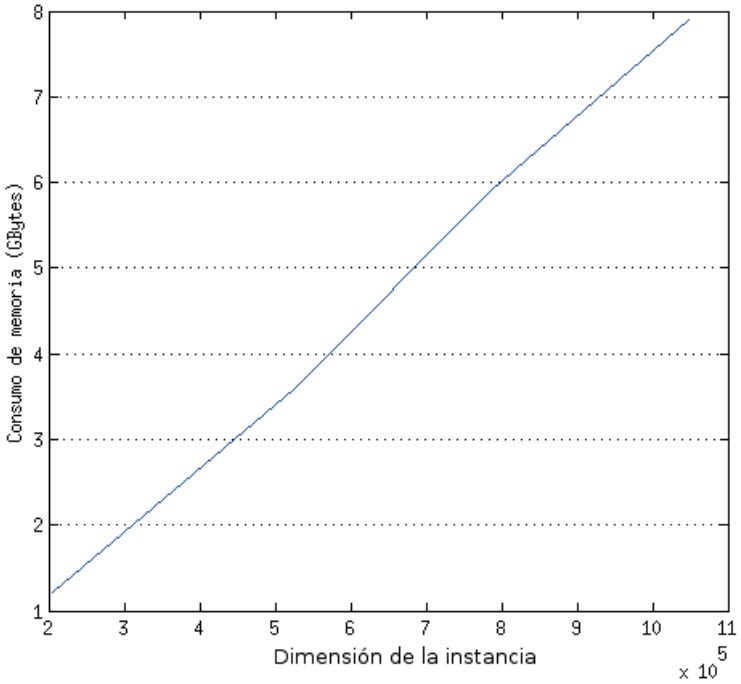

Figura 4.3: Crecimiento del consumo de memoria para el benchmark POISSON en función de la dimensión de las instancias. La densidad de las instancias (cantidad de elementos no nulos por fila/columna) es constante para este benchmark (aproximadamente 5 elementos no nulos por fila).

implementaciones basadas en GPU son consecuencia de que el acelerador instalado en la plataforma

versión	Matriz	Preproc.	lp_param	lrcf_adi	Total sistemas lineales	Total
Lyapack Original	A50	48.93	16.82	25.69	42.13	91.44
	A100	Out of memory				
Lyapack + ILUPACK	A050	3.65	8.60	0.74	8.21	9.34
	A100	34.54	120.19	7.23	124.31	127.42
	A126	72.09	294.91	17.46	306.21	312.36
	A159	150.61	725.07	35.23	747.96	760.30
	thermal2	11.19	451.60	123.11	570.68	574.70
	G3_circuit	15.85	299.23	255.82	550.37	555.06
Lyapack + CUSPARSE	A050	0.03	3.65	0.64	3.93	4.29
	A100	0.15	47.59	7.24	51.63	54.84
	A126	0.29	114.63	17.44	125.91	132.07
	A159	0.57	238.46	41.02	266.46	279.48
	thermal2	0.20	881.30	123.10	1000.50	1004.50
	G3_circuit	0.20	845.60	255.70	1096.60	1101.30

Tabla 4.16: Evaluación del resolutor de Lyapunov para matrices de los benchmark LAPLACE y UFSMC en la plataforma Enrico.

Enrico tiene menos núcleos CUDA que el de Buenaventura. Esto es más notorio para el caso de la implementación basada en CUSPARSE, donde todas las operaciones del resolutor se realizan en la GPU.

4.5. Resumen

En este capítulo se presentó la evaluación experimental del prototipo de biblioteca para MOR en arquitecturas híbridas. La evaluación de la propuesta se realiza por separado para cada familia de USFs implementada.

Para matrices tri-diagonales, el foco de la evaluación se centró en la resolución de sistemas de ecuaciones tri-diagonales, que es la operación fundamental del algoritmo LRCF-ADI para este tipo de problemas. En el caso de las matrices tri-diagonales se implementó una variante basada en GPU (siguiendo las ideas expuestas en [1]) del método de Reducción Cíclica para resolver los sistemas lineales desplazados. Los resultados numéricos muestran una importante aceleración de los sistemas tri-diagonales, que se traduce en una aceleración sustancial del resolutor de ecuaciones de Lyapunov para este tipo de problemas.

Respecto a la evaluación experimental de la implementación del conjunto de USFs de Lyapack para matrices banda generales, los resultados experimentales obtenidos en la plataforma Enrico revelan importantes factores de aceleración cuando se comparan con los correspondientes a la rutina basada en la biblioteca MKL de Intel. Las ventajas de las nuevas rutinas híbridas pueden apreciarse en las solución de las ecuaciones de Lyapunov para matrices banda mediante el método LRCF-ADI.

En el caso de las matrices dispersas generales, la evaluación de la versión acelerada con GPU de la biblioteca ILUPACK revela importantes aceleraciones para matrices de gran dimensión, mientras que la aplicación de los resolutores iterativos basados en GPU para sistemas de ecuaciones lineales permite la resolución de problemas de tamaños que no son abordables mediante la versión original de Lyapack.

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

Como se menciona en el Capítulo 1, el objetivo principal de este trabajo era diseñar y desarrollar una herramienta para resolver problemas de reducción de modelos dispersos que incluya técnicas de HPC basadas en arquitecturas de hardware híbridas, especialmente aquellas compuestas por procesadores multi-core y GPUs. Adicionalmente, se requiere que la biblioteca desarrollada sea flexible y fácil de utilizar, manteniendo el espíritu de bibliotecas previas, ampliamente utilizadas por ingenieros y científicos no necesariamente expertos en computación.

El camino para la concreción del objetivo principal planteaba los siguientes objetivos específicos:

- Realizar un relevamiento del estado del arte referente a problemas de MOR dispersos y computación de alto desempeño, en particular utilizando GPUs. El resultado de este relevamiento se presenta en el Capítulo 2.
- En base a lo relevado, realizar el diseño de un prototipo de biblioteca para la resolución de MOR dispersos que explote arquitecturas híbridas (CPUs multi-core y GPUs). En este sentido, se planteó una extensión de Lyapack, una conocida biblioteca implementada en MATLAB para resolver problemas de reducción de modelos dispersos. El diseño de esta extensión y algunos detalles sobre la implementación de la misma se describen en el Capítulo 3 de la tesis.
- Implementar finalmente un prototipo de biblioteca de MOR para problemas dispersos sobre arquitecturas híbridas (CPUs multi-core y GPUs). Los detalles referentes a la implementación de la extensión de la biblioteca Lyapack se describen en el Capítulo 3 de este trabajo, mientras que la evaluación experimental de los programas desarrollados se presenta en el Capítulo 4.

De esta forma, teniendo en cuenta los puntos anteriores, en este trabajo se ha presentado un nuevo framework para la solución de problemas de reducción de modelos dispersos de gran

dimensión, basado en la biblioteca Lyapack, que explota el potencial de la aceleración con GPU en tanto que mantiene las características de dicha biblioteca en cuanto a su interfaz amigable y su facilidad de uso.

La extensión de la biblioteca Lyapack desarrollada en este trabajo incluye la implementación basada en GPU de las *User Supplied Functions* (USFs) de Lyapack para matrices tri-diagonales, matrices banda generales y matrices dispersas generales.

En el caso de las matrices tri-diagonales se implementó una variante basada en GPU (siguiendo las ideas expuestas en [1]) del método de Reducción Cíclica para resolver los sistemas lineales desplazados. Los resultados numéricos muestra una importante aceleración de los sistemas tri-diagonales (alcanzando un factor $\times 20$ en aritmética real de doble precisión) lo cual se traduce en una aceleración $\times 10$ para el resolutor de ecuaciones de Lyapunov.

Sobre las USFs correspondientes a matrices banda, se presentaron nuevas rutinas híbridas (CPU-GPU) que aceleran la solución de sistemas lineales para este tipo de matrices. Estas rutinas aceleran la factorización LU y la posterior solución de los sistemas triangulares banda resultantes, computando las operaciones más costosas, desde el punto de vista computacional, en la GPU. La primer implementación para la etapa de factorización realiza los cómputos correspondientes a operaciones de BLAS-3 en el acelerador por medio de invocaciones a las rutinas de la biblioteca Nvidia CUBLAS, mientras se intenta, al mismo tiempo, minimizar el volumen de comunicación entre CPU y GPU. La segunda variante de esta rutina incorpora una estrategia de *look-ahead* para superponer la etapa de actualización en GPU con la factorización del siguiente panel en la CPU. Finalmente, una última variante utiliza una estrategia híbrida para combinar la etapa de factorización con la resolución del primer sistema triangular resultante de la misma, provocando un interesante ahorro en el tiempo de cómputo.

Los resultados experimentales obtenidos en la plataforma Enrico utilizando diversos casos de prueba, con matrices banda de dimensiones entre 12,800 y 76,800, y anchos de banda de 1 %, 2 % y 4 % de la dimensión del problema, revelan factores de aceleración para la factorización LU de hasta $\times 6$, cuando se compara con la correspondiente rutina basada en la biblioteca MKL de Intel. Las ventajas de las nuevas rutinas híbridas pueden apreciarse en la solución de las ecuaciones de Lyapunov para matrices banda mediante el método LRCF-ADI, con factores de aceleración de $\times 2$–3 respecto a los resolutores análogos basados en la biblioteca MKL.

Los grupos de USFs provistos por Lyapack intentan ser generales y, de hecho, abarcan una gran variedad de problemas. Con el objetivo de mantener esta generalidad en la versión de la biblioteca acelerada con GPUs, se han implementado los conjuntos de USFs para el caso de matrices dispersas generales, tanto simétricas como no simétricas. Dicha implementación está basada en el uso de métodos iterativos para resolver los sistemas de ecuaciones lineales (desplazados) presentes en el algoritmo LRCF-ADI. La utilización de métodos iterativos en lugar de métodos directos intenta resolver problemas de reducción de modelos de gran dimensión.

En el Capítulo 3 de este trabajo se discute la necesidad de utilizar buenos precondicionadores para resolver los mencionados sistemas lineales mediante métodos iterativos, mientras que durante la evaluación experimental, en el Capítulo 4, los resultados obtenidos para algunas de las matrices dan cuenta de esta necesidad. En este sentido, se realizó un relevamiento del estado del arte en métodos iterativos para la solución de sistemas lineales y se decidió trabajar con la biblioteca ILUPACK, que proporciona implementaciones de los métodos iterativos basados en subespacios de Krylov más difundidos para la solución de sistemas lineales, utilizando un poderoso precondicionador multinivel basado en la técnica de factorización LU incompleta.

Se presentaron dos nuevas implementaciones que aceleran en GPU el método del Gradiente Conjugado (CG) disponible en ILUPACK, computando en el procesador gráfico las operaciones más costosas durante la aplicación del precondicionador multinivel que, a su vez, es el cuello de

botella del método en cuanto a tiempo de cómputo. La primera variante computa la solución de sistemas triangulares y productos matriz-vector en la GPU, por medio de invocaciones a las rutinas correspondientes de las bibliotecas CUBLAS y CUSPARSE. La segunda variante realiza la aplicación del precondicionador completamente en la GPU, disminuyendo el volumen de transferencias de datos entre las memorias de la CPU y GPU. Los resultados experimentales obtenidos utilizando una colección de 7 sistemas de ecuaciones lineales con matriz de coeficientes simétrica y definida positiva, en dos plataformas equipadas con GPUs de las generaciones "Fermi" y "Kepler", reportan factores de aceleración para la aplicación del precondicionador que varían entre $\times 1.25$ y $\times 5.84$.

Se realizaron otras dos implementaciones en GPU del método CG basadas en la biblioteca CUSPARSE, una de ellas sin utilizar precondicionador, y la otra utilizando la implementación en GPU del precondicionador ILU0 que proporciona dicha biblioteca. La comparación de las tres implementaciones muestra los beneficios de utilizar la biblioteca ILUPACK para resolver sistemas lineales de gran dimensión, reduciendo notablemente la cantidad de iteraciones necesarias para alcanzar la convergencia del método CG.

El empleo de estos resolutores lineales iterativos en el método LRCF-ADI para resolver ecuaciones de Lyapunov muestra importantes ventajas. En particular, permite resolver problemas de dimensiones que no era posible abordar con la versión original de Lyapack debido a sus altos requerimientos de memoria.

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.

Revista

- P. BENNER; E. DUFRECHOU; P. EZZATTI; E. S. QUINTANA; A. REMÓN. Extending Lyapack for the Solution of Band Lyapunov Equations on Hybrid CPU-GPU Platforms *Journal of Supercomputing*, 2014, ISSN: 09208542 (Aceptado para publicación).

- E. DUFRECHOU; P. EZZATTI; E. S. QUINTANA; A. REMÓN. Accelerating the Lyapack library using GPUs. *Journal of Supercomputing*, 65(3), pp. 1114-1124, 2013, ISSN: 09208542.

Conferencias internacionales

- P. BENNER; E. DUFRECHOU; P. EZZATTI; P. IGOUNET; E. QUINTANA-ORTÍ; A. REMÓN. Accelerating Band Linear Algebra Operations on GPUs with Application in Model Reduction, *The 14th International Conference on Computational Science and Its Applications (ICCSA 2014)*, Guimaraes, Portugal, 2014.

- J. I. ALIAGA; M. BOLLHOFER; E. DUFRECHOU; P. EZZATTI; E. QUINTANA-ORTÍ. Leveraging Data-Parallelism in ILUPACK using Graphics Processors, *IEEE-13th International Symposium on Parallel and Distributed Computing (ISPDC)*, Toulon, Francia, 2014.

- E. DUFRECHOU; P. EZZATTI; E. QUINTANA-ORTÍ; A. REMÓN. Improving the Solution of Band Linear Systems on Hybrid CPU+GPU Platforms, *International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE 2014)*, Rota-Cadiz, España, 2014.

- E. DUFRECHOU; P. EZZATTI; E. QUINTANA-ORTÍ; A. REMÓN. Towards a many-core Lyapack library, *11th International Conference on Computational and MathematicalMethods in Science and Engineering (CMMSE 2012)*, La Manga, España, 2012.

Conferencias regionales

- E. DUFRECHOU; P. EZZATTI; E. QUINTANA-ORTÍ; A. REMÓN. Efficient Symmetric Band Matrix-Matrix Multiplication on GPUs, *CARLA (HPCLatam) 2014* , Valparaíso, Chile, 2014. (Best Paper)

- P. BENNER; E. DUFRECHOU; P. EZZATTI; E. QUINTANA-ORTÍ; A. REMÓN. Accelerating the General Band Matrix Multiplication Using Graphics Processors, *CLEI 2014*, Montevideo, Uruguay, 2014.

- E. DUFRECHOU; P. EZZATTI; E. QUINTANA-ORTÍ; A. REMÓN. Resolución de ecuaciones de Lyapunov dispresas en GPU, *XXIII Jornadas Sarteco 2012*, Elche, España, 2012.

5.3. Líneas abiertas de investigación

El trabajo realizado ha cubierto el objetivo original de la tesis, consistente en la aceleración de métodos de reducción de modelos dispersos utilizando arquitecturas híbridas compuestas por procesadores multi-hilo y aceleradoras gráficas, pero además ha servido para identificar diversos problemas y extensiones sobre los que merece la pena profundizar:

- El foco de este trabajo, respecto a la aceleración en GPU de las USFs de Lyapack, se ha centrado principalmente en la solución de los sistemas de ecuaciones lineales, ya que estos son el principal cuello de botella del algoritmo LRCF-ADI para resolver las ecuaciones de Lyapunov que aparecen en los métodos de reducción de modelos estudiados. Sin embargo, al acelerar estas rutinas, es posible que otras etapas del resolutor de Lyapunov aumenten su importancia relativa. Por lo tanto, una línea de trabajo futuro es la aceleración de las demás etapas de los algoritmos abordados.

- La rutina que realiza la factorización LU de los sistemas banda concurrentemente con la solución del primer sistema triangular de banda correspondiente (rutina MERGE) se evaluó únicamente de forma individual. La medición de su impacto sobre problemas de reducción de modelos y, en particular, sobre la resolución de ecuaciones de Lyapunov mediante el algoritmo LRCF-ADI se considera trabajo futuro.

- Para obtener una versión completamente funcional de la biblioteca es necesario extender los resolutores desarrollados para trabajar con aritmética de números complejos en caso de ser necesario.

- En relación al trabajo realizado sobre la biblioteca ILUPACK, se prevé migrar las demás etapas del método PCG para trabajar en arquitecturas híbridas, a fin de reducir aún más los costos de comunicación y sincronización entre CPU y GPU, y beneficiarse del poder de cómputo de la GPU para acelerar estas etapas.
- La versión de Lyapack desarrollada para el caso de matrices dispersas generales cuenta con ciertas limitaciones en la resolución de sistemas lineales desplazados. En este sentido, se plantea como línea de trabajo futuro el desarrollo de una estrategia eficiente que permita utilizar métodos iterativos precondicionados para la solución de este tipo de sistemas lineales.
- En relación con la aceleración en GPU de sistemas lineales precondicionados mediante la biblioteca ILUPACK, se considera como trabajo futuro mejorar la aplicación del precondicionador para sistemas no simétricos e indefinidos, así como también abordar sistemas de grandes dimensiones mediante técnicas de paralelismo en plataformas de memoria distribuida y multi-GPUs.
- Se plantea como línea abierta de trabajo encontrar una estrategia eficiente para resolver un conjunto de sistemas lineales desplazados precondicionados mediante la biblioteca ILUPACK.
- Una posibilidad adicional consiste en incorporar los aportes de este trabajo a los esfuerzos realizados sobre la biblioteca M.E.S.S., que se plantea como la sucesora de Lyapack. Este es un desafío plausible de abordar en breve, considerando que evaluaciones preliminares (no formalizadas) mostraron gran adaptación del trabajo realizado a la nueva biblioteca.
- Por último, es interesante evaluar el desempeño de la biblioteca desarrollada en relación al consumo energético.

Anexo A

Computación de propósito general en GPUs

La importante demanda impulsada por el mercado de los videojuegos, relacionada con la producción de gráficos 3D más complejos y realistas, ha motivado la evolución de las GPUs para convertirlas en procesadores masivamente paralelos, capaces de ejecutar cientos y hasta miles de hilos concurrentemente, con una impresionante potencia computacional y ancho de banda de memoria.

Esta evolución en la capacidad de cómputo, la cual es muy distinta a la experimentada por las CPU convencionales (ver Figura A.1) se debe a que las GPU están especialmente diseñadas para aplicaciones altamente paralelas e intensivas en cálculo, dedicando la mayor parte de los transistores a esta tarea, mientras que en la CPU, la mayor parte de ellos está dedicada a la lógica de control, manejo de cachés, etc.

Particularmente, las GPUs son especialmente adecuadas para problemas que exhiben paralelismo de datos, es decir, las mismas operaciones son realizadas sobre distintos elementos del conjunto de datos de forma independiente, y que realizan una alta cantidad de operaciones aritméticas en relación a los accesos a memoria. En este tipo de aplicaciones, como el mismo programa es ejecutado en paralelo por distintos hilos independientes, no es necesaria una compleja lógica de control de flujo, y debido a la alta intensidad computacional, la latencia en el acceso a memoria es ocultada por los cálculos, en lugar de utilizar grandes *cachés* de datos.

Todo lo expuesto anteriormente despertó el interés de algunos pioneros que dedicaron un importante esfuerzo a utilizar estas plataformas para resolver problemas de propósito general, es decir, no relativos a la presentación de gráficos en la pantalla. En esta primera etapa las GPUs todavía estaban dedicadas a la presentación de gráficos y su utilización para otras tareas exigía mapear estas tareas a etapas del pipeline gráfico. Su programación presentaba entonces una dificultad tal que reducía los desarrollos en estas plataformas a algunos pocos experimentos realizados por expertos.

Adicionalmente, la arquitectura que presentaban las GPUs en ese entonces, la cual incluía distintas unidades de procesamiento para distintas etapas del pipeline gráfico, solía dejar recursos ociosos. Por ejemplo, en modelos 3D con geometrías complejas, la carga de trabajo del *vertex shader* era intensiva mientras que la del *pixel shader* era baja, y en escenas con geometrías simples pero texturas complejas, se daba el escenario opuesto.

Atendiendo estos problemas, en Noviembre de 2006[1], NVIDIA anunció el lanzamiento de CUDA, una nueva arquitectura, set de instrucciones y lenguaje de programación, que unifican todas las etapas del pipeline gráfico en un único tipo de procesador paralelo. Este hito marcó el inicio de una nueva concepción de la GPU como coprocesador de propósito general, teniendo desde entonces una alta aceptación, sobre todo por la comunidad científica, dado el bajo costo económico de estas

[1]Microsoft ya había presentado la misma idea en 2005 con su consola de juegos Xbox.

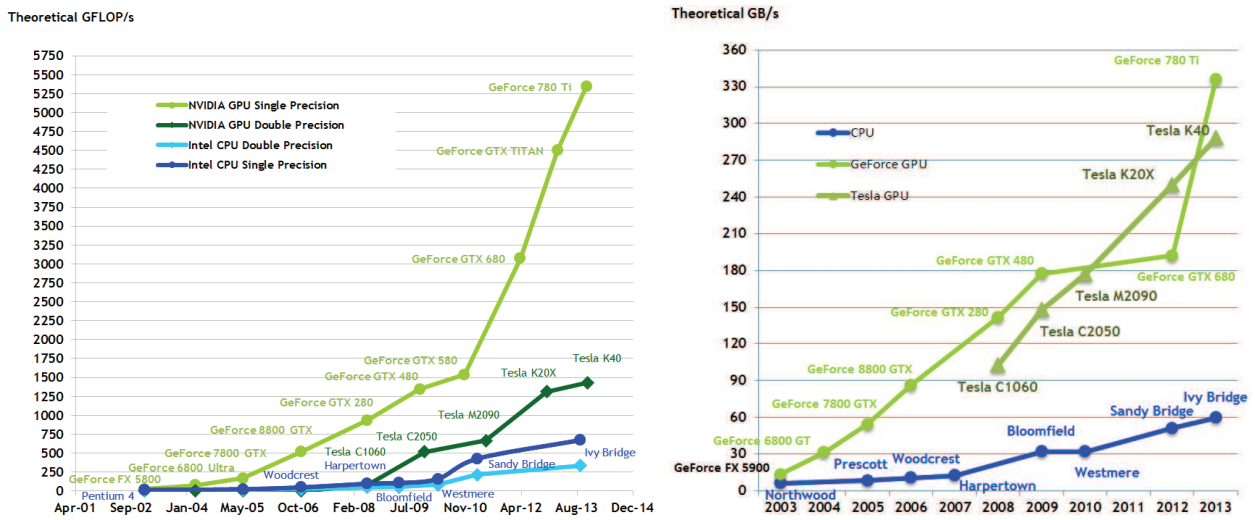

Figura A.1: Evolución en capacidad teórica de cómputo en GFlops (izquierda) y ancho de banda de memoria (derecha) de CPU y GPU desde el año 2003 a la fecha. Extraída de [62].

plataformas en relación con otros tipos de hardware de alto desempeño, su eficiencia energética y el amplio espectro de herramientas desarrolladas hasta la fecha, que facilitan crecientemente su programación para diversas aplicaciones.

A.1. CUDA

La palabra CUDA es una sigla que abrevia *Compute Unified Device Architecture*[2]. Este término es utilizado en sentido amplio, no solamente para describir la nueva arquitectura de hardware presentada en 2006, sino también para referirse al modelo y lenguaje de programación que permite crear aplicaciones que ejecuten en estos dispositivos.

Desde el punto de vista del modelo de programación, la ejecución del programa se distribuye en hilos, los cuales se organizan en una grilla indexada llamada bloque, y a su vez, estos bloques también forman una grilla indexada. A priori, la ejecución de los hilos y bloques es totalmente independiente y puede darse en cualquier orden. El lenguaje de programación consiste en una extensión del lenguaje C, que permite, entre otras cosas, la creación de estas grillas de hilos, la programación de las funciones a ejecutar en el dispositivo (*kernels*) y las transferencias de datos entre la CPU y la GPU.

Por su parte, la arquitectura de hardware se forma entorno a una serie de multi-procesadores paralelos (*Streaming Multiprocessors* o SMs), cada uno formado por varios núcleos. La cantidad de procesadores y núcleos con los que cuenta la GPU varía según las diferentes generaciones de tarjetas.

La memoria del dispositivo se organiza de forma jerárquica. En un primer nivel, existe una memoria global relativamente lenta pero accesible a todos los hilos. Se encuentra fuera de los multiprocesadores, por lo que el acceso a la misma implica una alta latencia, pero también posee un gran ancho de banda. Adicionalmente, a partir de la segunda generación de CUDA, se incluye una memoria caché de segundo nivel. Dentro de cada multiprocesador, existe una memoria de baja

[2]Lo cual se puede traducir como Arquitectura de Dispositivo de Cómputo Unificado.

latencia pero de tamaño reducido. Esta memoria es compartida físicamente por los bloques de hilos que residen en el multiprocesador. Por último, cada multiprocesador contiene un archivo de registros el cual es repartido entre todos los hilos residentes en el multiprocesador. Los registros son la memoria más rápida que brinda la GPU, pero también es la más reducida en tamaño.

Existe cierta correspondencia entre este modelo abstracto y la arquitectura de hardware. Siguiendo el espíritu de la taxonomía de sistemas paralelos propuesta por Flynn en [38], NVIDIA clasifica su arquitectura como *Single Instruction Multiple Thread* o SIMT. A diferencia de la categoría SIMD, en la cual una misma instrucción se ejecuta simultáneamente sobre distintos elementos del conjunto de datos, en SIMT el usuario puede especificar distintos flujos de ejecución para los distintos hilos, aunque dadas las características del hardware, el desempeño es mucho mayor cuando el comportamiento de la aplicación se asemeja al tipo SIMD.

Una vez que comienza la ejecución del programa, cada multiprocesador se encarga de la ejecución concurrente de un grupo fijo de bloques. Los hilos pertenecientes a estos bloques son divididos, planificados y ejecutados en grupos de (hasta el momento) 32 hilos llamados *warps*. La división se realiza siempre de forma que los hilos con índice 0 a 31 forman el primer warp, los de índice 32 a 63 al segundo, y así sucesivamente. La ejecución se organiza de forma que los threads de un mismo warp deben ejecutar la misma instrucción en cada momento. Si, de acuerdo con el flujo de ejecución de distintos hilos del mismo warp, dos hilos divergen y deben ejecutar distintas instrucciones, las mismas se ejecutan de forma serial y los hilos que ejecutan una de las instrucciones quedan inactivos hasta que el resto de los hilos del warp ejecuten la otra instrucción. Por esta razón, la máxima eficiencia es alcanzada cuando todos los hilos de un warp ejecutan la misma instrucción en todo momento, aunque la misma se ejecute sobre distintos elementos del conjunto de datos.

La asignación de recursos a cada warp dentro de un multiprocesador se realiza de forma estática, asignando un segmento del archivo de registros a cada warp, y asignando una sección de la memoria compartida del multiprocesador a cada bloque asignado al mismo. De esta forma, el cambio de contexto entre un warp y otro se realiza sin costo.

Junto con CUDA, NVIDIA provee un siempre creciente conjunto de herramientas, a las que denomina “Ecosistema CUDA”, las cuales están orientadas a complementar la arquitectura mejorando su usabilidad en distintas áreas de aplicación. Entre estas herramientas se encuentran lenguajes como CUDA FORTRAN, PyCuda, APIs como OpenACC o PGI Accelerator Compiler, herramientas de análisis, debugging, y un gran conjunto de bibliotecas aceleradas con GPU, como por ejemplo cuFFT, cuBLAS y cuSPARSE.

A.2. Arquitecturas Fermi y Kepler

Producto del rápido desarrollo de las GPUs que fue mencionado anteriormente, ya son cuatro (Tesla, Fermi, Kepler y la reciente Maxwell) las generaciones de tarjetas gráficas programables que ha lanzado NVIDIA en los últimos ocho años, introduciendo con cada generación importantes cambios en la arquitectura de hardware.

La evaluación experimental de este trabajo (ver Capítulo 4) se realizó principalmente utilizando dos modelos de GPU, la NVIDIA C2070 de la arquitectura Fermi, y la NVIDIA K20 de la arquitectura Kepler. Es importante entonces describir brevemente estas arquitecturas, destacando las principales diferencias entre una y otra, con el objetivo de interpretar correctamente los resultados experimentales en ambas plataformas.

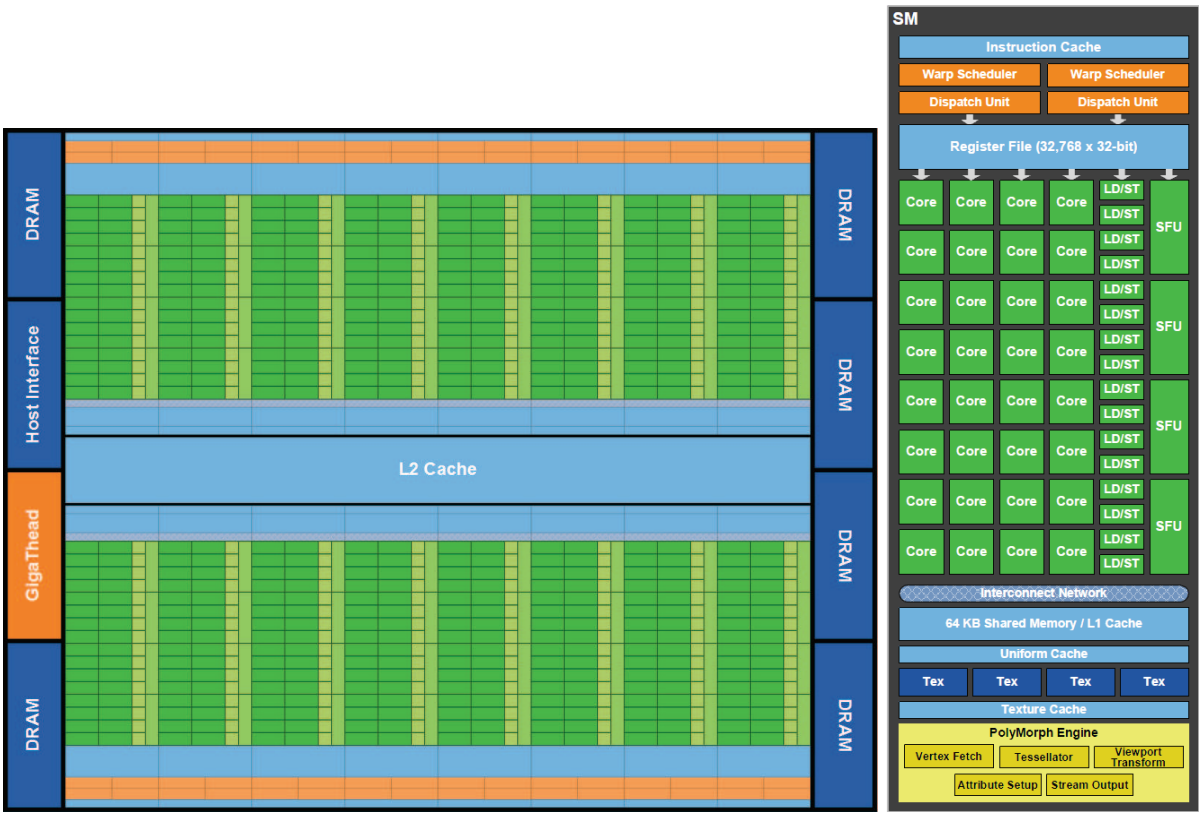

Figura A.2: Diagrama de bloques de la arquitectura Fermi GF100 (izquierda) y su multiprocesador (derecha). Extraído de *NVIDIA Fermi GF100 Architecture Whitepaper*.

Fermi

La arquitectura Fermi, ilustrada en la Figura A.2, presenta hasta 512 núcleos CUDA organizados en 16 multiprocesadores de 32 cores cada uno. La cantidad de núcleos y multiprocesadores varía según el modelo de GPU. La arquitectura soporta hasta 6 GB de memoria RAM GDDR5 con una interfaz de memoria de 384 bits.

Cada núcleo CUDA contiene una Unidad Aritmético-Lógica (ALU) y una unidad de punto flotante (FPU). A diferencia de sus predecesoras (arquitectura Tesla), las GPUs Fermi soportan el estándar IEEE 754-2008 de aritmética en punto flotante, proporcionando la operación *fused multiply-add* (FMA), la cual realiza la multiplicación y adición con un solo paso final de redondeo, evitando así la pérdida de precisión en la adición [62]. Un esquema del diseño del multiprocesador de la arquitectura Fermi puede apreciarse en la Figura A.2 (derecha).

En el caso de la GPU C2070, los núcleos CUDA funcionan a una frecuencia de 1.15GHz, con un consumo energético de 238 W.

Kepler

Luego de el éxito de la arquitectura Fermi, el cual logra mejorar notoriamente el desempeño respecto a la arquitectura antecesora, estableciéndose de esa manera en el mercado de HPC, NVIDIA rediseñó completamente la arquitectura de hardware buscando, no solo maximizar el desempeño computacional, sino también aumentar la eficiencia energética.

Figura A.3: Diagrama de bloques de la arquitectura Kepler GK110. Extraído de *NVIDIA Kepler GK110 Architecture Whitepaper*.

La arquitectura Kepler GK100 (ver Figura A.3) incluye un completo rediseño del multiprocesador, el cual pasa a llamarse SMX. Cada uno de los (como máximo) 15 SMX, incluye 192 núcleos CUDA, que al igual que en Fermi cuentan con una ALU y una FPU que soporta el estándar IEEE 754-2008, 64 unidades de doble precisión, 32 unidades de funciones especiales y 32 unidades de lectura/escritura. En la Figura A.4 puede apreciarse un diagrama de este nuevo multiprocesador.

A diferencia de los SM de arquitecturas previas, los SMX de Kepler utilizan el reloj principal de la GPU en lugar de utilizar el reloj del *shader*, que trabaja al doble de frecuencia. El reloj del shader fue introducido en la arquitectura Tesla G80 y utilizado desde entonces en las arquitecturas Tesla y Fermi. Si bien utilizar un clock más rápido para las unidades de ejecución logra realizar más trabajo con menos unidades de procesamiento, lo cual en principio es una ventaja, pero la circuitería para trabajar con el clock rápido demanda un mayor consumo energético. Por este motivo, la arquitectura Kepler agrega más unidades de procesamiento que operan a una frecuencia menor, obteniendo así un mayor desempeño para muchas aplicaciones, sin que esto signifique un mayor costo energético.

Es importante resaltar que esta arquitectura resulta ventajosa en aplicaciones que tienen un alto grado de paralelismo, capaz de explotar al máximo el gran número de unidades de procesamiento. Para aplicaciones que tienen un nivel de paralelismo más modesto, la arquitectura Fermi, con sus núcleos CUDA de mayor frecuencia, puede llegar a ser más conveniente.

Figura A.4: Diagrama de bloques del multiprocesador de la arquitectura Kepler GK110. Extraído de *NVIDIA Kepler GK110 Architecture Whitepaper*

Anexo B

Bibliotecas de ALN

En el presente anexo se brinda una descripción de las principales bibliotecas de álgebra lineal numérica utilizadas en la tesis.

B.1. BLAS

La necesidad de establecer un conjunto definido de rutinas básicas que sirvieran como bloques de construcción para resolver problemas de ALN fue atendida por primera vez en 1973, cuando Hanson et al. [56] publicaron la primera versión de este conjunto de rutinas, escritas en FORTRAN, bajo el nombre de *Basic Linear Algebra Subprograms* (BLAS). Su objetivo fue definir un estándar para operaciones básicas frecuentes en algoritmos de álgebra lineal numérica, que permitiera mejorar su claridad, portabilidad y modularidad. Desde entonces, BLAS se ha convertido en un estándar *de facto* en el área de ALN.

En sus orígenes, la biblioteca solo contenía operaciones de tipo vectorial, por ejemplo norma vectorial, producto escalar, escalado de vectores, etc. Esto fue denominado posteriormente como nivel 1 de BLAS o BLAS-1. Con el correr de los años, la evolución de las arquitecturas de computadores y, particularmente, la aparición de las computadoras vectoriales y la capacidad de resolver operaciones de BLAS utilizando las operaciones vectoriales de las nuevas arquitecturas, motivó la extensión de la biblioteca. Es así como en 1984 se presentó un nuevo conjunto de rutinas que implementaba operaciones de tipo matriz-vector. Esta extensión de la biblioteca es conocida hasta la actualidad como BLAS-2.

En los años siguientes, el desarrollo de nuevas arquitecturas, en especial el avance en los conceptos de jerarquía de memoria, dieron lugar a algoritmos de ALN eficientes en cuanto al acceso y reutilización de los datos. Esto dio lugar a la creación de nuevas rutinas de BLAS para realizar operaciones de tipo matriz-matriz, donde los algoritmos "a bloques" ocupan un lugar importante. Este conjunto de rutinas se conoce como BLAS-3.

La mayor parte de las rutinas de los tres niveles de BLAS se encuentran especificadas para diversos tipos de matriz (completas, de banda, triangulares) y tipos de datos (simple y doble precisión, números complejos). Una descripción completa de los distintos niveles de la biblioteca BLAS puede encontrarse en [34].

Actualmente, BLAS es una especificación de un subconjunto de rutinas, y existen diversas implementaciones de la misma, producidas por distintos laboratorios y orientadas a un diverso espectro de plataformas de cómputo. La utilización de operaciones de BLAS en aplicaciones, facilita entonces una utilización eficiente de los recursos de cómputo, así como la portabilidad del código de una plataforma a otra. A manera de ejemplo pueden mencionarse las implementaciones de Intel

y NVIDIA, ambos códigos propietarios orientados a ejecutar en arquitecturas multi-core y GPUs NVIDIA respectivamente. La implementación de BLAS de Intel forma parte de la biblioteca *Intel Math Kernel Library* y es considerada por la comunidad como una de las implementaciones más eficientes de BLAS para ejecutar en procesadores Intel. Por su parte, la implementación de NVIDIA, CUBLAS, es la implementación de referencia cuando se trabaja con GPUs NVIDIA, y se describe más adelante en la Sección B.3.

B.2. LAPACK

LAPACK (*Linear Algebra Package*) es una biblioteca que provee una solución eficiente para problemas comunes en el área de ALN. Entre otras cosas, proporciona rutinas para resolver sistemas de ecuaciones lineales, problemas de mínimos cuadrados, valores propios y descomposiciones matriciales como SVD, QR, LU y Cholesky.

La biblioteca fue concebida en 1987, en la Universidad de Tennesse (Estados Unidos) con el objetivo de actualizar las bibliotecas LINPACK [33] y EISPACK [40], las cuales eran ampliamente usadas en ese entonces para resolver problemas de mínimos cuadrados y valores propios sobre máquinas vectoriales. El desarrollo de las arquitecturas con memoria jerárquica y distintos niveles de caché puso en evidencia las dificultades de estas bibliotecas para lograr un uso eficiente del hardware, dado que sus patrones de acceso a memoria no estaban diseñados para este tipo de arquitecturas. En este sentido, LAPACK propuso una reorganización de los algoritmos que hace posible utilizar operaciones con patrones de acceso a memoria "por bloques", más adecuado para las arquitecturas de computadores modernas.

Originalmente escrita en FORTRAN 77, LAPACK cambió a Fortran90 a partir de su versión 3.2, y proporciona rutinas para matrices reales y complejas, densas y de banda, tanto en simple como en doble precisión. Hasta la fecha, LAPACK no brinda soporte a matrices dispersas generales.

Las rutinas de LAPACK están diseñadas para hacer el mayor uso posible de la biblioteca BLAS. Esto facilita la portabilidad de LAPACK a distintos tipos de arquitectura, así como la mejora automática de su desempeño a través de la optimización de BLAS. Más aún, LAPACK busca expresar sus algoritmos en términos de operaciones BLAS-3, sacando máximo provecho de arquitecturas de hardware con varios niveles de caché y procesadores multi-core.

Existen diversas implementaciones optimizadas de la biblioteca, entre las que se destacan la implementación de Intel, que también forma parte de la biblioteca MKL [86]. Además se dispone de extensiones para explotar arquitecturas distribuidas como, por ejemplo, las bibliotecas ScalaPACK [24] y PLAPACK [82].

B.3. CuBLAS

CUBLAS es la implementación sobre GPUs, realizada por NVIDIA, de la biblioteca BLAS. La biblioteca incluye rutinas correspondientes a los tres niveles de BLAS, trabajando sobre matrices densas (posiblemente triangulares) y de banda, reales y complejas, tanto en simple como en doble precisión.

Para utilizar la API de CUBLAS, el usuario debe transferir las matrices y vectores que sean necesarios a la memoria de la GPU, ejecutar las rutinas de CUBLAS correspondientes y luego transferir el resultado nuevamente a la memoria de la CPU. La API incluye rutinas para realizar este tipo de operaciones.

A partir de la versión 6.0 de CUDA, liberada en 2014, la biblioteca incluye una API llamada CUBLASXT API, diseñada para trabajar con múltiples GPUs de forma automática.

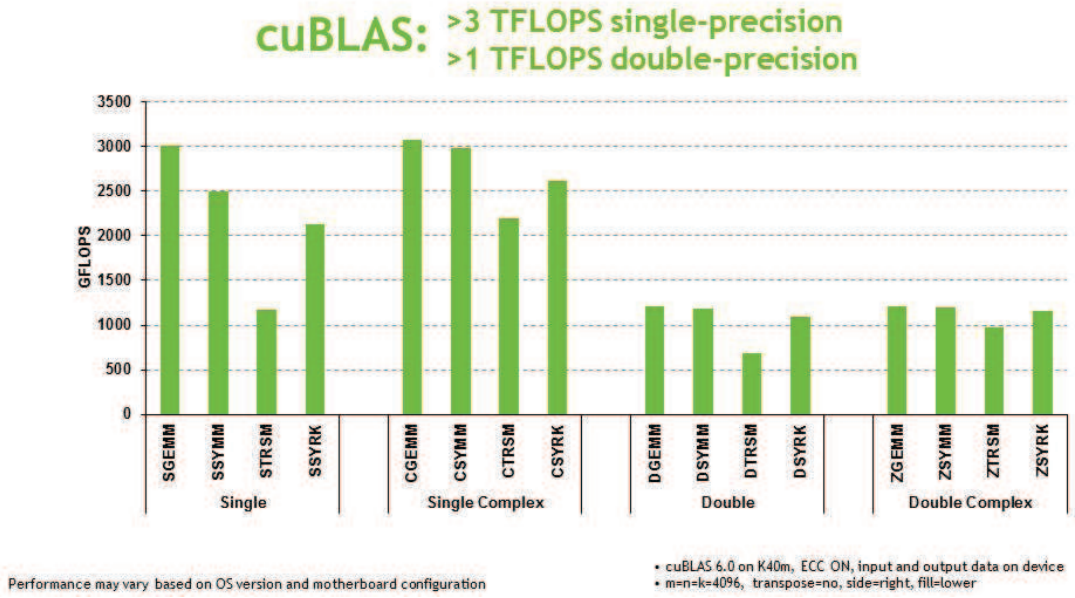

Figura B.1: Desempeño en GFlops de implementación de CUBLAS 6.0 para la multiplicación de matriz simétrica por matriz (SYMM), multiplicación de matrices generales (GEMM), resolución de sistemas triangulares (TRSM) y actualización de rango k para una matriz simétrica (SYRK). Extraído de *https://developer.nvidia.com/cublas*.

Una evaluación del desempeño (en GFlops) de algunas de las principales rutinas de CUBLAS puede apreciarse en la Figura B.1.

B.4. CuSPARSE

La biblioteca cuSPARSE permite explotar el poder de cómputo de las GPU mediante un conjunto de rutinas básicas de álgebra lineal para matrices dispersas. Estas rutinas pueden clasificarse en cuatro categorías:

- Nivel 1: operaciones entre un vector en formato disperso y un vector en formato denso.
- Nivel 2: operaciones entre una matriz en formato disperso y un vector en formato denso.
- Nivel 3: operaciones entre una matriz en formato disperso y un conjunto de vectores en formato denso (que pueden interpretarse como una matriz de pocas columnas).
- Conversión: operaciones relativas a la conversión entre distintos formatos de almacenamiento disperso.

Entre estas operaciones se encuentran la multiplicación de matriz dispersas por vector o conjunto de vectores, la resolución de sistemas triangulares dispersos, y factorizaciones matriciales incompletas LU y Cholesky.

La mayor parte de las rutinas de CUSPARSE se encuentran implementadas para el formato de almacenamiento disperso conocido como CSR. Adicionalmente, CUSPARSE provee versiones

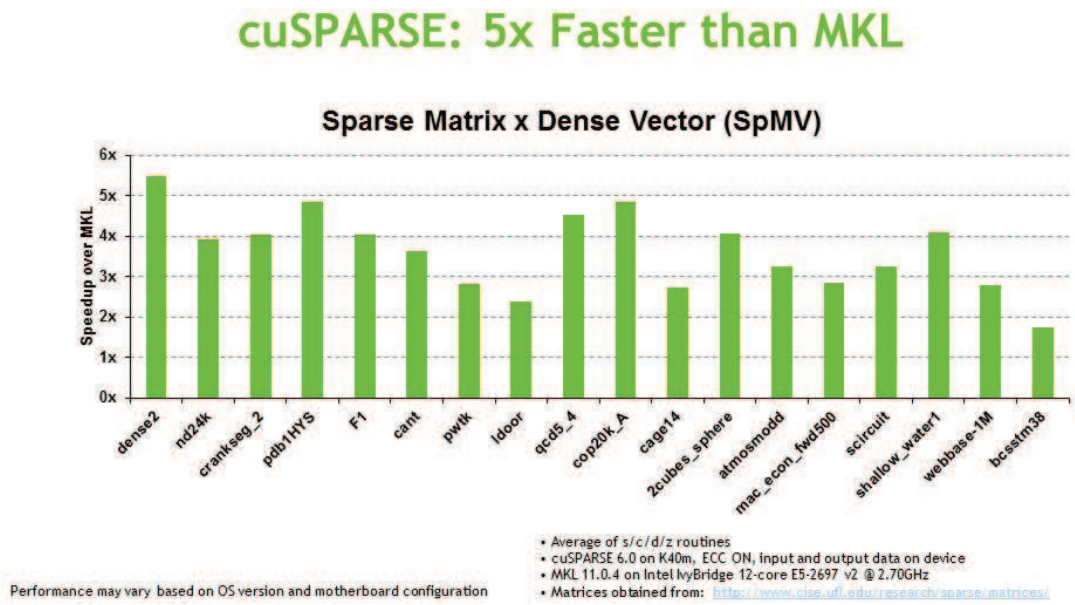

Figura B.2: Comparación entre el desempeño de la multiplicación de matriz dispersa por vector de CUSPARSE y su correspondiente implementación basada en la biblioteca Intel MKL para distintas matrices dispersas.
Extraído de *https://developer.nvidia.com/cuSPARSE*.

de algunas rutinas (especialmente la multiplicación de matriz dispersa por vector) (`spmv`) para los formatos CSC, BCSR, BCSRX e HYB [10, 85].

Al igual que en el caso de CUBLAS, es responsabilidad del usuario reservar la memoria necesaria en el dispositivo y generar o transferir los datos utilizados por las rutinas. La API proporciona funciones para facilitar estas tareas.

Una comparación entre el desempeño de la multiplicación de matriz dispersa por vector de CUSPARSE y su correspondiente implementación basada en la biblioteca Intel MKL puede apreciarse en la Figura B.2.

Anexo C

Métodos dispersos iterativos utilizando GPUs

Lejos de ser un problema restringido al área de reducción de modelos, la solución de sistemas de ecuaciones lineales donde la matriz de coeficientes es dispersa está presente en muy diversas aplicaciones científicas e ingenieriles, representando un cuello de botella, desde el punto de vista computacional, en los problemas que surgen de la discretización de ecuaciones en derivadas parciales (PDEs) y aplicaciones en áreas como física cuántica o simulación de circuitos. Por este motivo, el desarrollo de técnicas eficientes para resolver sistemas lineales es de suma importancia cuando la dimensión de los problemas se vuelve considerable, por ejemplo, debido al aumento de los dispositivos en un circuito integrado que se desea simular.

Mientras que los solvers directos proveen una forma eficiente de atacar cierta clase de problemas, son sumamente inadecuados para otros. Particularmente, los métodos directos son sumamente lentos, y en ocasiones inaplicables, por ejemplo, a problemas 3D debido al excesivo *fill-in* durante la etapa de factorización [41]. En estos casos, técnicas de factorización aproximada en conjunto con métodos iterativos basados en subespacios de Krylov son una alternativa altamente utilizada [72].

Considerando lo expuesto en los párrafos anteriores, en este apartado se describen los diferentes esfuerzos relacionados con el uso de GPU para acelerar los métodos iterativos de resolución de sistemas de ecuaciones lineales.

C.1. Estudio del estado del arte

Como se mencionó anteriormente, las arquitecturas masivamente paralelas de las GPUs actuales son altamente adecuadas para computar operaciones de álgebra lineal densa, ya que este tipo de problemas presentan, frecuentemente, patrones alineados de acceso a memoria y gran intensidad de cómputo. Sin embargo, varios investigadores también han reportado factores de aceleración importantes para problemas que implican la manipulación de matrices dispersas. Concretamente, la aceleración de algoritmos iterativos para la solución de sistemas lineales dispersos usando GPUs, así como métodos basados en subespacios de Krylov, han sido estudiados anteriormente en varias ocasiones. Una cantidad notable de estos esfuerzos considera solamente versiones no precondicionadas de los métodos, en los cuales la multiplicación de matriz dispersa por vector (`spmv`) es la operación que concentra la mayor parte del tiempo de ejecución.

Diversos trabajos pioneros proponen implementaciones en GPU de solvers típicos, incluso antes de que existiese CUDA. Algunos ejemplos incluyen el trabajo de Rumpf y Strzodka [71] resolviendo sistemas lineales mediante el método del Gradiente Conjugado (GC) para métodos de elementos

finitos, el de Bolz et al. [27] utilizando el *pipeline gráfico* para resolver problemas multi-grilla; el de Bajaj et al. [7] acerca de los métodos de Jacobi y Gauss-Seidel; así como los estudios sobre el GC de Goodnight et al. [45], Hillesland et al. [50], y Krüger et al. [55].

Luego de la aparición de CUDA, Buatois et al. [28] integró un simple precondicionador de Jacobi al método del Gradiente Conjugado. Este trabajo saca partido del formato comprimido por filas a bloques (BCSR) de almacenamiento disperso para la implementación de la `spmv`, y presenta *kernels* escritos en CUDA para operaciones de nivel 1 de BLAS como el producto escalar (`dot`) y la adición de vectores, uno de ellos escalado (`saxpy`). Poco tiempo después, Bell y Garland [10] realizaron un completo estudio sobre la `spmv`, evaluando varios formatos de almacenamiento disperso, los cuales formaron la base para el desarrollo de la biblioteca CUSP [11].

Verschoor y Jalba [85] también apuntaron a la mejora de la `spmv` utilizando el formato BCSR, esta vez analizando los efectos de ciertos reordenamientos de los bloques. Estos autores evalúan el *speed-up* total, considerando el tiempo de ejecución promedio para 64 casos de prueba, y reportan mejoras de hasta 1.25× en comparación con la implementación de Bell y Garland. Más aún, el algoritmo presentado en [85] muestra un pobre desempeño para matrices que tienen largos de fila desbalanceados, y los autores muestran que dejando tres casos extremos fuera del promedio, el *speed-up* total asciende a 2.5×. Adicionalmente, a pesar que la implementación de Buatois et al. mostró ser superior en el 33 % de los casos de prueba, Verschoor y Jalba aún obtienen un factor de mejora de 3.7× para el total de los casos y, dejando de lado los tres casos extremos, el factor aumenta a 6.1×.

Ament et al. [4] presentaron un solver paralelo basado en el Gradiente Conjugado para la ecuación de Poisson, optimizado para arquitecturas multi-GPU. Sudan et al. [80] implementan *kernels* en GPU para la `spmv` y el método GMRES precondicionado mediante una factorización LU incompleta (ILU) a bloques, aplicado a simulación de fluidos, mostrando factores de aceleración prometedores al compararlo contra la implementación serial de los algoritmos. Más o menos al mismo tiempo, Gupta completó su tesis de maestría [47] en la cual implementa un método de GC precondicionado con deflación para fluidos burbujeantes, reportando aceleraciones de hasta 20× respecto a la implementación en CPU.

Naumov [61], por su parte, describió un algoritmo para resolver sistemas lineales triangulares en una GPU, la cual es una de las operaciones principales (en cuanto a tiempo de ejecución) de métodos iterativos como CG o GMRES cuando se utilizan precondicionadores basados en ILU. Mas tarde en [61], el mismo autor desarrolla algoritmos para computar las factorizaciones incompletas (LU y Cholesky) con *0 fill-in* en GPU. El desempeño de los algoritmos mencionados depende fuertemente del patrón de dispersión de la matriz de coeficientes.

En [58], Li y Saad optimizaron *kernels* para la `spmv` con distintos formatos de almacenamiento disperso para construir una versión con paralelismo de datos del GC y GMRES. Como el rendimiento del solver de sistemas triangulares en GPU para los métodos GC y GMRES precondicionados con IC e ILU respectivamente resultó ser pobre, los autores proponen un enfoque híbrido, en el cual los sistemas triangulares son resueltos en CPU. Además, también se evalúan otros métodos de precondicionado, por ejemplo *Jacobi*, *multicolored SSOR/ILU0*, y *least squares polynomial*.

Muchas de estas ideas están actualmente implementadas en bibliotecas y frameworks como CUSPARSE o CUSP. En particular, CUSP implementa la `spmv` para los formatos dispersos COO, CSR, DIA, ELL e HYB, junto con algunos métodos de Krylov. CUSPARSE (ver Sección B.4, por su parte, implementa la `spmv` para los formatos disperso CSR, HYB, BCSR y BCSRX, presentando también un solver de sistemas triangulares para los formatos CSR e HYB. A su vez, también integra una rutina que computa el precondicionador ILU0, basada en la implementación de Naumov.

Existen además otras bibliotecas y frameworks para la solución de sistemas lineales dispersos en GPU. CULA Sparse [52] provee implementaciones de métodos como BICG con precondicionador

de Jacobi, BICGStab con Jacobi a Bloques, así como GC y GMRES con ILU0.

Por último, cabe mencionar el caso de PARALUTION [59], una biblioteca que provee implementaciones de varios métodos iterativos dispersos en plataformas multi-core y many-core. Este software contiene métodos de Krylov (GC, BiCGStab, GMRES e IDR), multigrilla (GMG, AMG), iteraciones de punto fijo, esquemas de precisión mixta y precondicionadores paralelos.

En contraste con los trabajos revisados, en esta tesis se propone una implementación con paralelismo a nivel de datos de un solver iterativo basado en el GC, pero equipado con un complejo precondicionador multinivel, el cual se encuentra implementado en ILUPACK. Las propiedades matemáticas de dicho precondicionador lo vuelven una opción sumamente atractiva para la solución de sistemas lineales dispersos, mientras que su paralelización resulta una tarea desafiante.

Anexo D

ILUPACK

ILUPACK provee un completo juego de rutinas implementadas en C y Fortran para resolver sistemas lineales de tipo $Ax = b$ mediante métodos iterativos basados en subespacios de Krylov. Estas rutinas incluyen el cálculo de un precondicionador M y la aplicación de distintos métodos iterativos para resolver el sistema lineal utilizando dicho precondicionador.

La convergencia de los métodos iterativos está estrechamente relacionada con el número de condición del sistema lineal subyacente [72]. Por este motivo es que se intenta, por medio del cómputo de un precondicionador M, obtener un sistema (precondicionado) equivalente al original, con un número de condición más favorable y por lo tanto mejor convergencia. Una conocida técnica de precondicionado para métodos basados en subespacios de Krylov consiste en calcular una factorización LU incompleta (ILU) de la matriz A, es decir, una aproximación de sus factores LU en la que algunos elementos son descartados para así controlar el *fill-in* de los factores resultantes.

El enfoque de precondicionado multinivel propuesto por ILUPACK explota las llamadas "factorizaciones ILU basadas en la matriz inversa" o *inverse-based ILU factorizations*. A diferencia de factorizaciones ILU más clásicas, como las basadas en umbrales (*threshold-based ILU*), este método directamente acota la magnitud del error inducido por el descarte de algunos coeficientes durante la factorización, mejorando tanto su robustez como su escalabilidad, especialmente para aplicaciones relacionadas con PDEs, debido a su conexión con los métodos algebraicos multinivel [2].

La idea clave de este procedimiento es evitar los cálculos con pivotes "numéricamente-difíciles", los cuales son movidos a las últimas filas/columnas de la matriz mediante las permutaciones correspondientes. Para lograr esto, se utiliza la variante de *Crout* de la factorización LU [72], realizando las siguientes operaciones en cada iteración del método aplicado a una matriz A simétrica y definida positiva:

1. Aplicar la transformación correspondiente a la parte de la matriz que ya ha sido factorizada a la columna y fila actuales.

2. Si el pivote actual es "numéricamente dudoso", mover la columna actual al final de la matriz, acumulando estas permutaciones en la matriz P.

3. De otra forma, proceder con la factorización de la fila y columna actual, aplicando ciertas "técnicas de descarte".

Cuando este proceso se completa, se obtiene una factorización ILU parcial de la forma $P^T AP$, y debe computarse un complemento de Schur para la parte de la matriz que aún no ha sido factorizada. Este proceso es repetido recursivamente sobre el complemento de Schur hasta que la matriz está factorizada completamente. Las técnicas de descarte y el cálculo del complemento de

Schur están diseñados para acotar la magnitud de los inversos de los factores triangulares que forman la factorización. Esta propiedad mejora el desempeño numérico del método, ya que dichos inversos están ligados directamente al residuo del sistema precondicionado.

D.1. Aplicación del precondicionador multinivel

A continuación se describe la aplicación de este precondicionador como parte del método iterativo.

El precondicionador multinivel puede ser expresado recursivamente, en un nivel l dado, de la siguiente forma:

$$M_l = D^{-1}\tilde{P}^{-T}P^{-T}\begin{pmatrix} L_B & 0 \\ L_F & I \end{pmatrix}\begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix}\begin{pmatrix} L_B^T & L_F^T \\ 0 & I \end{pmatrix}P^T\tilde{P}^T D^{-1}, \tag{D.1}$$

donde D y $\tilde{P}$ son matrices de permutación; L_B, D_B y L_F son bloques correspondientes a los factores del precondicionador LDL^T multinivel (con L_B triangular inferior unidad y D_B diagonal); y M_{l+1} representa al precondicionador calculado en el nivel $l+1$.

Aplicar el precondicionador en este nivel, es decir, calcular $z := M_l^{-1}r$ (donde se omiten los subíndices de los vectores z y r por simplicidad), requiere la solución del siguiente sistema de ecuaciones lineales:

$$\begin{pmatrix} L_B & 0 \\ L_F & I \end{pmatrix}\begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix}\begin{pmatrix} L_B^T & L_F^T \\ 0 & I \end{pmatrix}P^T\tilde{P}^T D^{-1}z = P^T\tilde{P}^T Dr. \tag{D.2}$$

Al analizar (D.2), reconocemos, en primer lugar, tres transformaciones aplicadas al vector residuo r. Primero, $r' := Dr$ aplica un escalado diagonal a este vector; luego, $r'' := \tilde{P}^T r'$ realiza una primer permutación al resultado y, por último, $\hat{r} := P^T r''$ aplica una segunda permutación.

Una vez que estas transformaciones se han completado, el sistema

$$\begin{pmatrix} L_B & 0 \\ L_F & I \end{pmatrix}\begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix}\begin{pmatrix} L_B^T & L_F^T \\ 0 & I \end{pmatrix}w = \hat{r} \tag{D.3}$$

es resuelto para $w(= P^T\tilde{P}^T D^{-1}z)$ en tres pasos. Primero, resolviendo

$$\begin{pmatrix} L_B & 0 \\ L_F & I \end{pmatrix}y = \hat{r} \tag{D.4}$$

para y; luego resolviendo recursivamente

$$\begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix}x = y \tag{D.5}$$

para x; y finalmente resolviendo

$$\begin{pmatrix} L_B^T & L_F^T \\ 0 & I \end{pmatrix}w = x. \tag{D.6}$$

A su vez, las expresiones en (D.4) y (D.6) también requieren ser resueltas en dos pasos. Asumiendo que los vectores y y $\hat{r}$ son particionados conforme a los bloques de los factores, para (D.4) se obtiene

$$\begin{pmatrix} L_B & 0 \\ L_F & I \end{pmatrix}\begin{pmatrix} y_B \\ y_C \end{pmatrix} = \begin{pmatrix} \hat{r}_B \\ \hat{r}_C \end{pmatrix}. \tag{D.7}$$

Este sistema es posteriormente abordado resolviendo el sistema triangular inferior unidad

$$L_B y_B = \hat{r}_B \tag{D.8}$$

para y_B, y luego calculando,

$$y_C := \hat{r}_C - L_F y_B. \tag{D.9}$$

Particionando los vectores igual que antes, la Ecuación D.5 involucra el producto diagonal

$$x_B := D_B^{-1} y_B, \tag{D.10}$$

y el paso recursivo

$$x_C := M_{l+1}^{-1} y_C. \tag{D.11}$$

No hace falta entrar en detalles acerca del paso base de la recursión ya que, para la inmensa mayoría de los casos de prueba, representa una fracción despreciable del tiempo de cómputo total y por lo tanto no fue abordado en este trabajo.

Por último, luego de una partición análoga, la Ecuación (D.6) puede ser reformulada como

$$w_C := x_C \tag{D.12}$$

y

$$L_B^T w_B = x_B - L_F^T w_C; \tag{D.13}$$

obteniendo z de forma sencilla a través de $z := D(\hat{P}^T(P^T w))$.

Desde un punto de vista práctico, con el propósito de ahorrar memoria, ILUPACK descarta el factor L_F una vez que se calcula cada nivel del precondicionador, manteniendo únicamente una matriz dispersa y rectangular F, frecuentemente mucho más dispersa que L_F, de forma que $L_F = F L_B^{-T} D_B^{-1}$.

Esto cambia (D.9) por

$$y_C := \hat{r}_C - F L_B^{-T} D_B^{-1} y_B, \tag{D.14}$$

lo cual, en combinación con (D.8), produce

$$y_C := \hat{r}_C - F L_B^{-T} D_B^{-1} L_B^{-1} \hat{r}_B. \tag{D.15}$$

Además, (D.13) también cambia por

$$L_B^T w_B = D_B^{-1} y_B - D_B^{-1} L_B^{-1} F^T w_C. \tag{D.16}$$

Ahora, (D.15) puede ser abordada resolviendo primero $L_B D_B L_B^T s_B = \hat{r}_B$ para s_B, y luego obteniendo $y_C := \hat{r}_C - F s_B$.

A su vez, la Ecuación (D.16) puede ser abordada resolviendo $L_B D_B L_B^T \hat{s}_B = F^T w_C$ para $\hat{s}_B$, obteniendo luego $w_B := s_B - \hat{s}_B$.

Bibliografía

[1] P. Alfaro, P. Igounet, and P. Ezzatti. A study on the implementation of tridiagonal systems solvers using a GPU. In H.X. Lin, M. Alexander, M. Forsell, A. Knüpfer, R. Prodan, L. Sousa, and A. Streit, editors, *Proceedings of the XXX International Conference of the Chileans Computer Science Society (SCCC'2011)*. IEEE, 2011.

[2] J. I. Aliaga, M. Bollhöfer, A. F. Martín, and E. S. Quintana-Ortí. Exploiting thread-level parallelism in the iterative solution of sparse linear systems. *Parallel Computing*, 37(3):183–202, 2011.

[3] J. I. Aliaga, M. Bollhöfer, A. F. Martín, and E. S. Quintana-Ortí. Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors. In Kristján Jónasson, editor, *Applied Parallel and Scientific Computing*, volume 7133 of *Lecture Notes in Computer Science*, pages 162–172. 2012.

[4] M. Ament, G. Knittel, D. Weiskopf, and W. Straßer. A parallel preconditioned conjugate gradient solver for the Poisson problem on a multi-GPU platform. In *PDP '10: Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Networkbased Processing*, pages 583–592, 2010.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammerling, A. McKenney, et al. *LAPACK Users' guide*, volume 9. SIAM, 1999.

[6] A. C. Antoulas. *Approximation of Large-Scale Dynamical Systems (Advances in Design and Control) (Advances in Design and Control)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2005.

[7] 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, December 2004.

[8] S Balay, J Brown, K Buschelman, V Eijkhout, W Gropp, D Kaushik, M Knepley, L Curfman McInnes, B Smith, and H Zhang. PETSc users manual, revision 3.4, 2013.

[9] R. H. Bartels and G. W. Stewart. Solution of the matrix equation $AX + XB = C$. *Commun. ACM*, 15(9):820–826, September 1972.

[10] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[11] N. Bell and M. Garland. CUSP: Generic parallel algorithms for sparse matrix and graph computations, 2012. Version 0.3.0.

[12] P. Benner, P. Ezzatti, D. Kressner, E. S. Quintana-Ortí, and A. Remón. Accelerating model reduction of large linear systems with graphics processors. In Kristján Jónasson, editor, *Applied Parallel and Scientific Computing - 10th International Conference, PARA 2010, Reykjavík, Iceland, June 6-9, 2010, Revised Selected Papers, Part II*, volume 7134 of *Lecture Notes in Computer Science*, pages 88–97. Springer, 2010.

[13] P. Benner, P. Ezzatti, D. Kressner, E. S. Quintana-Ortí, and A. Remón. A mixed-precision algorithm for the solution of Lyapunov equations on hybrid CPU-GPU platforms. *Parallel Computing*, 37(8):439–450, 2011.

[14] P. Benner, P. Ezzatti, H. Mena, E. S. Quintana-Ortí, and A. Remón. Solving differential Riccati equations on multi-GPU platforms. In *2nd Meeting on Linear Algebra, Matrix Analysis and Applications ALAMA10*, 2010.

[15] P. Benner, P. Ezzatti, H. Mena, E. S. Quintana-Ortí, and A. Remón. Solving differential Riccati equations on multi-GPU platforms. In *10th International Conference on Computational and Mathematical Methods in Science and Engineering – CMMSE 2011*, pages 178–188, 2011.

[16] P. Benner, P. Ezzatti, E. S. Quintana-Ortí, and A. Remón. Using hybrid CPU-GPU platforms to accelerate the computation of the matrix sign function. In H.X. Lin, M. Alexander, M. Forsell, A. Knüpfer, R. Prodan, L. Sousa, and A. Streit, editors, *7th Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Lecture Notes in Computer Science, Vol. 6043, pages 132–139. Springer-Verlag, 2009.

[17] P. Benner, P. Ezzatti, E. S. Quintana-Ortí, and A. Remón. Accelerating BST methods for model reduction with graphics processors. In *9th Int. Conference on Parallel Processing and Applied Mathematics*, 2011.

[18] P. Benner, J. Li, and T. Penzl. Numerical solution of large-scale Lyapunov equations, Riccati equations, and linear-quadratic optimal control problems. *Numerical Linear Algebra with Applications*, 15(9):755–777, 2008.

[19] P. Benner, R. Mayo, E.S. Quintana-Orti, and G. Quintana-Orti. A service for remote model reduction of very large linear systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 6 pp.–, April 2003.

[20] P. Benner, V. Mehrmann, V. Sima, Sabine Van Huffel, and Andras Varga. SLICOT - a subroutine library in systems and control theory. In *Applied and Computational Control, Signals, and Circuits*, pages 499–539. Birkhäuser, 1997.

[21] P. Benner, V. Mehrmann, and Danny C Sorensen. *Dimension reduction of large-scale systems*, volume 45. Springer, 2005.

[22] 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.

[23] P. Benner, E. S. Quintana-Ortí, and G. Quitana-Ortí. Ba.ced truncation model reduction of large-scale dense systems on parallel computers. *Mathematical and Computer Modelling of Dynamical Systems*, 6(4):383–405, 2000.

[24] L S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, et al. *ScaLAPACK users' guide*, volume 4. SIAM, 1997.

[25] 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.

[26] M. Bollhöfer and Y. Saad. Multilevel preconditioners constructed from inverse–based ILUs. *SIAM J. Sci. Comput.*, 27(5):1627–1650, 2006. special issue on the 8–th Copper Mountain Conference on Iterative Methods.

[27] Jeff Bolz, Ian Farmer, Eitan Grinspun, and P. Schröoder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, July 2003.

[28] L. Buatois, G. Caumon, and B. Levy. Concurrent number cruncher: An efficient sparse linear solver on the GPU. In *High Performance Computation Conference (HPCC), Springer Lecture Notes in Computer Sciences*, 2007.

[29] Y. Chahlaoui and P. Van Dooren. A collection of benchmark examples for model reduction of linear time invariant dynamical systems. SLICOT Working Note 2002–2, February 2002. Available from `http://www.win.tue.nl/niconet/NIC2/reports.html`.

[30] C. Chen. *Linear System Theory and Design*. Oxford University Press, Inc., New York, NY, USA, 3rd edition, 1998.

[31] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

[32] B. De Schutter. Minimal state-space realization in linear system theory: An overview. *J. Comput. Appl. Math.*, 121(1-2):331–354, September 2000.

[33] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia, 1979.

[34] J. J Dongarra, J. Du Croz, S. Hammarling, and I. S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[35] Jack J Dongarra and R Clint Whaley. LAPACK working note 94 a user's guide to the BLACS v1. *Tech. Report*, 1997.

[36] I. S. Duff and G. A. Meurant. The effect of ordering on preconditioned conjugate gradients. *BIT Numerical Mathematics*, 29(4):635–657, 1989.

[37] P. Ezzatti, E.S. Quintana-Ortí, and A. Remón. Efficient model order reduction of large-scale systems on multi-core platforms. In B. Murgante, O. Gervasi, A. Iglesias, D. Taniar, and B. Apduhan, editors, *Computational Science and Its Applications - ICCSA 2011*, volume 6786 of *Lecture Notes in Computer Science*, pages 643–653. Springer Berlin Heidelberg, 2011.

[38] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972.

[39] V. Galiano, A. Martín, H. Migallón, V. Migallón, J. Penadés, 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.

[40] B. S. Garbow. EISPACK - a package of matrix eigensystem routines. *Computer Physics Communications*, 7(4):179 – 184, 1974.

[41] A. George. Nested Dissection of a Regular Finite Element Mesh. *SIAM J. Numer. Anal.*, 10(2):345–363, 1973.

[42] A. George, J. Liu, and E. Ng. Computer solution of sparse linear systems. *Academic Press, Orlando*, 1994.

[43] T. George, A. Gupta, and V. Sarin. An empirical analysis of the performance of preconditioners for spd systems. *ACM Trans. Math. Softw.*, 38(4):24:1–24:30, August 2012.

[44] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2013.

[45] 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.

[46] S. Gugercin and A.C. Antoulas. A survey of ba.cing methods for model reduction. In *Proc. European Control Conference ECC 2003, Cambridge, UK*, 2003.

[47] R. Gupta, M. Bastiaan van G., and C. Vuik. 3d bubbly flow simulation on the GPU - iterative solution of a linear system using sub-domain and level-set deflation. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, volume 0, pages 359–366, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[48] S. J. Hammarling. Numerical solution of the stable, non-negative definite Lyapunov equation. *IMA J. Numer. Anal*, 2:303–323, 1982.

[49] M. R Hestenes. The conjugate gradient method for solving linear systems. In *Proc. Symp. Appl. Math VI, American Mathematical Society*, pages 83–102, 1956.

[50] 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.

[51] R. W. Hockney. A fast direct solution of Poisson's equation using Fourier analysis. *J. ACM*, 12(1):95–113, January 1965.

[52] J. R. Humphrey, D. K. Price, K. E. Spagnol, A. L. Paolini, and E. J. Kelmelis. CULA: Hybrid GPU Accelerated Linear Algebra Routines. In *SPIE Defense and Security Symposium (DSS)*, 2010.

[53] Wen-mei W. Hwu. *GPU Computing Gems Jade Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[54] IMTEK. *Oberwolfach model reduction benchmark collection.* `http://www.imtek.de/simulation/benchmark`, Accedido el 2-9-2014.

[55] 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.

[56] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[57] R.B. Lehoucq, D.C. Sorensen, and C. Yang. *ARPACK Users' Guide: Solution of Large-scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.* Software, Environments, Tools. SIAM, 1998.

[58] R. Li and Y. Saad. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63(2):443–466, 2013.

[59] Dimitar Lukarski. PARALUTION project. `http://www.paralution.com`, Accedido el 2-9-2014.

[60] B. Moore. Principal component analysis in linear systems: Controllability, observability, and model reduction. *Automatic Control, IEEE Transactions on*, 26(1):17–32, Feb 1981.

[61] M. Naumov. Incomplete-LU and Cholesky preconditioned. Iterative methods using CUSPARSE and CUBLAS. Nvidia white paper, 2011.

[62] Nvidia CUDA Nvidia. Programming guide, version 6.0. *Nvidia Corporation, February*, 2014.

[63] Nvidia Corporation. *CUSPARSE User Guide 4.1*, 2012.

[64] T. Penzl. Lyapack: A Matlab toolbox for large Lyapunov and Riccati equations, model reduction problems, and linear-quadratic optimal control problems users' guide (version 1.0).

[65] T. Penzl. A cyclic low-rank smith method for large sparse Lyapunov equations. *SIAM Journal on Scientific Computing*, 21(4):1401–1418, 1999.

[66] T. Penzl. Eigenvalue decay bounds for solutions of Lyapunov equations: the symmetric case. *Systems and Control Letters*, 40(2):139 – 144, 2000.

[67] T. Penzl. Algorithms for model reduction of large dynamical systems. *Linear Algebra and its Applications*, 415(2):322–343, 2006.

[68] D. Raczynski and W. Stanislawski. Controllability and observability gramians parallel computation using GPU. *Journal of Theoretical and Applied Computer Science*, 6:47–66, 2012.

[69] A. Remón, E. S. Quintana-Ortí, and G.orio Quintana-Ortí. Parallel solution of band linear systems in model reduction. In Roman Wyrzykowski, J. 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.

[70] J. D. Roberts. Linear model reduction and solution of the algebraic Riccati equation by use of the sign function. *International Journal of Control*, 32(4):677–687, 1980.

[71] M. Rumpf and R. Strzodka. Using graphics cards for quantized fem computations. In *IASTED Visualization, Imaging and Image Processing Conference*, pages 193–202, 2001.

[72] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[73] J Saak, H Mena, and P Benner. Matrix equation sparse solvers (mess): a Matlab toolbox for the solution of sparse large-scale matrix equations. *Chemnitz University of Technology, Germany*, 2010.

[74] O. Schenk, A. Wächter, and M. Weiser. Inertia Revealing Preconditioning For Large-Scale Nonconvex Constrained Optimization. *SIAM J. Scientific Computing*, 31(2):939–960, 2008.

[75] O. Schenk, A. Wächter, and M. Weiser. Algebraic Multilevel Preconditioner For the Helmholtz Equation In Heterogeneous Media. *SIAM J. Scientific Computing*, 31(2):3781–3805, 2009.

[76] Gaurav Sharma and Jos Martin. Matlab®: A language for parallel computing. *International Journal of Parallel Programming*, 37(1):3–36, 2009.

[77] R. Smith. Matrix equation $XA+BX = C$. *SIAM Journal on Applied Mathematics*, 16(1):198–201, 1968.

[78] G. Starke. Optimal alternating direction implicit parameters for nonsymmetric systems of linear equations. *SIAM Journal on Numerical Analysis*, 28(5):1431–1445, 1991.

[79] 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.

[80] H Sudan, H Klie, R Li, and Y Saad. High performance manycore solvers for reservoir simulation. In *12th European conference on the mathematics of oil recovery*, 2010.

[81] M. S. Tombs and I. Postlethwaite. Truncated ba.ced realization of a stable non-minimal state-space system. *International Journal of Control*, 46(4):1319–1330, 1987.

[82] R. A Van de Geijn. *Using PLAPACK–parallel Linear Algebra Package*. MIT Press, 1997.

[83] H. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.

[84] A. Varga. Task II.B.1 – selection of software for controller reduction. SLICOT Working Note 1999–18, The Working Group on Software (WGS), 1999.

[85] M. Verschoor and A. C. Jalba. Analysis and performance estimation of the conjugate gradient method on multiple GPUs. *Parallel Comput.*, 38(10-11):552–575, October 2012.

[86] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel Math Kernel Library. In *High-Performance Computing on the Intel Xeon Phi*, pages 167–188. Springer International Publishing, 2014.