Resolución de ecuaciones de Lyapunov dispersas en GPU

Ernesto Dufrechou[1], Pablo Ezzatti[2], Enrique S. Quintana-Ortí[3] y Alfredo Remón[4]

Resumen— **Lyapack es una conocida biblioteca para resolver problemas en teoría de control como, por ejemplo, ecuaciones de Lyapunov o Riccati, y su aplicación en reducción de modelos y problemas de control óptimo. Esta herramienta tiene un diseño sencillo y extensible. Además, está implementada en Matlab, lo cual facilita su comprensión, modificación y verificación, incluso por usuarios no expertos. Sin embargo, en algunos casos Matlab no ofrece un desempeño óptimo al resolver este tipo de problemas. El presente trabajo pretende mejorar el desempeño de Lyapack sin alterar su diseño y usabilidad. En los últimos años los coprocesadores gráficos (GPUs) han experimentado una importante evolución, que ha permitido y facilitado su empleo para resolver problemas de propósito general y, en particular, para acelerar modelos numéricos y problemas de computación científica. En este artículo se presenta una extensión de Lyapack que incorpora el uso de GPUs, y se evalúan las rutinas implementadas para la resolución de ecuaciones de Lyapunov tridiagonales. Los resultados obtenidos demuestran que puede obtenerse una significativa reducción del tiempo de ejecución, en comparación con la versión original de la biblioteca, mientras se mantienen las ventajas ofrecidas por Matlab en cuanto a flexibilidad y usabilidad.**

Palabras clave— **Teoría de control, GPU, Matlab, Matrices dispersas**

I. Introducción

La resolución de ecuaciones matriciales lineales y cuadráticas, aplicadas a problemas de reducción de modelos o control óptimo lineal-cuadrático, es uno de los problemas numéricos más importantes en el campo de la teoría de control [10]. Problemas de este tipo surgen frecuentemente al modelar procesos físicos mediante ecuaciones diferenciales parciales. El alto coste computacional de estas técnicas propició la aparición de diversas bibliotecas software especializadas que, mediante el uso de técnicas de computación de alto desempeño (HPC), aceleran la resolución de este tipo de problemas. En este sentido, Lyapack es una conocida biblioteca que contiene rutinas para resolver ecuaciones matriciales dispersas de gran tamaño. La herramienta está implementada en Matlab, lo cual facilita su comprensión, modificación y verificación, incluso por usuarios no expertos, presentando además un diseño sencillo y extensible. Sin embargo, en algunos casos Matlab no ofrece un desempeño óptimo al resolver este tipo de problemas.

Los elevados tiempos de ejecución que insumen las técnicas de resolución de ecuaciones matriciales y los métodos de reducción de modelos para sistemas de gran tamaño motivan el uso de estrategias y arquitecturas de HPC. Sin embargo, el uso de arquitecturas HPC implica importantes esfuerzos económicos, tanto en su adquisición con en su mantenimiento. Por otro lado, aunque Matlab puede utilizar bibliotecas paralelas de alto desempeño, como por ejemplo versiones multi-hebra de BLAS [14], este enfoque no es del todo útil al tratarse de problemas con matrices dispersas.

Una estrategia que ha ganado importancia en los últimos años es el empleo de coprocesadores gráficos (GPUs) para resolver problemas de propósito general. Estos dispositivos han experimentado una importante evolución, tanto cuantitativa (en lo que se refiere a poder de cómputo) como cualitativa (mejoras en diseño, herramientas y facilidad para programación). Desde la aparición de CUDA (Compute Unified Device Architecture) [13], desarrollado por NVIDIA en 2007, las GPUs perdieron su especifidad, convertiéndose en dispositivos útiles para el cómputo de aplicaciones de propósito general. El elevado número de unidades computacionales disponible en las GPUs, la alta velocidad de acceso a la memoria y la utilización del paradigma de programación SPMD, convierten a las GPUs en dispositivos especialmente efectivos en la computación de operaciones algebráicas. Además, su bajo coste las convierten en las arquitecturas HPC de mayor distribución. Son ya numerosas las investigaciones que han utilizado GPUs para la resolución de problemas de ALN con notables resultados [2], [11], [20].

La resolución de ecuaciones matriciales y la reducción de modelos están estrechamente relacionados con operaciones de ALN [12], y debido a los importantes resultados obtenidos en la aceleración de operaciones densas de ALN mediante el uso de GPUs, no es extraño que en los últimos años este tema concentre gran parte del esfuerzo de la comunidad. Sin embargo, en la revisión realizada no se han encontrado propuestas referentes a reducción de modelos dispersos utilizando GPUs. El primer esfuerzo en utilizar GPUs para acelerar problemas de reducción de modelos es el de Benner et al. [7], en donde se implementa sobre una GPU la función de signo de una matriz utilizando simple precisión. Posteriormente el trabajo fue extendido [4], resolviendo las ecuaciones de Lyapunov con técnicas de precisión mixta, obteniendo resultados con calidad similar a doble precisión. Ya en reducción de modelos, los autores del trabajo [3] implementaron el método de *Balanced Truncate* (BT) en una arquitectura híbrida de

[1]Instituto de Computación, Univ. de la República, 11.300-Montevideo (Uruguay) e-mail: `edufrechou@fing.edu.uy`
[2]Instituto de Computación, Univ. de la República, 11.300-Montevideo (Uruguay) e-mail: `pezzatti@fing.edu.uy`
[3]Dpto. Ingeniería y Ciencia de Computadores, Univ. Jaime I, 12.071-Castellón (España) e-mail: `quintana@icc.uji.es`
[4]Dpto. Ingeniería y Ciencia de Computadores, Univ. Jaime I, 12.071-Castellón (España) e-mail: `remon@icc.uji.es`

GPU y CPU. Los métodos de error relativo también han sido implementados en GPUs, en particular el método *Balanced Stochastic Truncation* (BST) fue implementado [8], incluyendo la resolución de una ecuación algebraica de Riccati (EAR). Por su parte, la ecuación diferencial de Riccati (EDR) es resuelta en GPU en el trabajo [5], extendido posteriormente en [6] para explotar plataformas multi-GPU (4 GPUs). En el trabajo referenciado, el uso de 4 GPUs proporciona aceleraciones de hasta 20× respecto a las implementaciónes *multi-core*, y además permite abordar problemas de mayor dimensión. Así, es posible resolver problemas de dimensión 34.000, mientras que con una única GPU la dimensión del mayor problema es 17.000. También en [17] se estudió la aceleración del método *Backward Differentiation Formula* (BDF) para la resolución de la EDR utilizando una GPU.

El presente artículo se organiza de la siguiente manera: Primero se expone una breve introducción acerca de la biblioteca Lyapack en la Sección II. En la Sección III, se propone una arquitectura flexible para la utilización de GPUs con Lyapack y se describe la implementación necesaria para resolver el caso de matrices tridiagonales. Los resultados experimentales se encuentran en la Sección IV. Por último, en la Sección V, se discuten algunas conclusiones y propuestas de trabajo futuro.

II. Biblioteca Lyapack

Lyapack (Lyapunov Package) [19] es una herramienta implementada en Matlab para la resolución de ecuaciones de Lyapunov y Riccati, reducción de modelos y problemas de control óptimo. En particular, está especialmente diseñada para alcanzar buen desempeño en la resolución de problemas que involucran matrices dispersas de gran tamaño.

El objetivo principal de Lyapack, además de resolver correctamente los problemas descritos, es proporcionar a científicos inexpertos en computación numérica una plataforma eficiente y, sobre todo, fácil de usar. Esto motivó el uso de Matlab, en lugar de utilizar otros lenguajes más eficientes para computación científica, como FORTRAN o C.

Desde el punto de vista de HPC, aunque Matlab no ofrece un rendimiento óptimo para resolver los problemas a los que apunta Lyapack, sus rutinas incluyen técnicas, como el uso de variables globales ocultas para almacenar grandes estructuras de datos, que permiten mejorar dicho rendimiento.

Desde el punto de vista matemático, Lyapack implementa una aproximación eficiente para resolver ecuaciones dispersas de Lyapunov basada en el método ADI. En particular, la biblioteca implementa eficientemente el método ADI con Factorización de Cholesky de Rango Bajo (LRCF-ADI) [18]. El resto de los métodos de la biblioteca son construidos sobre el *solver* de las ecuaciones de Lyapunov.

Desde el punto de vista del software, la herramienta está basada en un diseño sencillo y extensible, que incluye algunas guías para mejorar el rendimiento manteniendo la flexibilidad. Un concepto importante de Lyapack es el de *basic matrix operations* (BMOs). Lyapack clasifica estas operaciones en tres categorías:

- Multiplicaciones, $Y \leftarrow AX$.
- Solución de sistemas de ecuaciones lineales, $Y \leftarrow A^{-1}X$.
- Solución de sistemas de ecuaciones lineales desplazados, $Y \leftarrow (A + p_i I)^{-1}X$.

En las expresiones anteriores, A es una matriz de dispersa de dimensión $n \times n$, X e Y son matrices densas de dimensión $n \times r$, con $r \ll n$, p_i es un escalar e I es la matriz identidad de dimensión n.

Las tres operaciones pueden trabajar sobre la matriz A o A^T.

Estos métodos están implementados en las rutinas principales, con nombres que incluyen el prefijo (lp_), y que constituyen el corazón de la biblioteca. Sin embargo, cada BMO que se utiliza en una rutina principal es resuelta mediante una invocación a una *user-supplied function* (USF).

Las USFs permiten al usuario especifica la rutina a emplear para el cálculo de cada BMO, ya sea proporcionando rutinas disponibles en otras bibliotecas o rutinas implementadas por el propio usuario. Las USFs proporcionan a Lyapack un concepto similar al polimorfismo definido en programación orientada a objetos.

De esta forma, todas las operaciones con la matriz A son realizadas por USFs. A su vez, todos los datos correspondientes a la matriz A son almacenados en variables globales ocultas. El objetivo de las USFs es manipular estas variables globales (crear los datos globales, ejecutar las BMOs y liberar la memoria tras realizar los cálculos), brindando una implementación para las distintas etapas involucradas en la solución de los sistemas de ecuaciones lineales con matrices dispersas. Estas etapas usualmente implican preprocesamiento, reordenamiento de las matrices, precondicionado, factorizaciones y efectivamente la resolución de los sistemas. Desde el punto de vista del diseño, las USFs permiten utilizar diferentes implementaciones de las operaciones básicas, dependiendo del tipo de matriz con el que la rutina deba trabajar.

La biblioteca emplea algunas extensiones que son agregadas al nombre básico de las rutinas siguiendo el siguiente criterio:

[nombre base] [ext_1] o [nombre base] [ext_1] [ext_2]

Cinco valores son posibles en la primera extensión. 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.

En algunos casos no existen rutinas de pre y postprocesamiento, ya que no son necesarias. Si [ext_1]

= pre o pst no hay segunda extensión. En caso contrario, los posibles valores para la segunda extensión 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 (rutina con nombre similar pero en la que [ext_2] = i).

Una vez más se pueden relacionar 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 en el programa principal (la rutina escrita por el usuario) y que las USFs con [ext_2] = i serán llamadas frecuentemente en 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.

III. Implementación

El principal objetivo de este trabajo es desarrollar una extensión de Lyapack que permita explotar la capacidad de cómputo ofrecida por una GPU de forma transparente al usuario, manteniendo con ello la filosofía original de la biblioteca: la simplicidad de uso. Para lograr esto se utilizó la interfaz MEX de Matlab. Esta interfaz consiste en una API que brinda interoperabilidad entre Matlab y otros lenguajes de programación como C, FORTRAN o, como en este trabajo, CUDA.

Para llamar a una función C/CUDA desde Matlab, se debe primero generar un archivo Mex con el nombre de la función. Este archivo es en realidad un archivo fuente C/CUDA, que debe ser compilado de una cierta manera, que incluye la Mex API y que 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 Mex.

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*.

Considerando lo anterior, la plataforma propuesta consiste en dos elementos pricipales:

- Biblioteca_CUDA: Esta biblioteca implementa todas las USFs necesarias en lenguaje CUDA.
- El_Archivo_Mex: Este archivo actúa como interfaz entre Matlab y la Biblioteca CUDA. Su objetivo principal es recibir los parámetros de las funciones Matlab, haciendo la traducción entre la representación de datos de Matlab y C, y llamando a las funciones implementadas en la biblioteca CUDA.

Adicionalmente, se provee un script de Matlab que compila todos los elementos de la solución.

Para cada tipo de matriz abordado, la Biblioteca CUDA implementa las funciones pref_m,l,s_i,-,d de acuerdo con la nomenclatura de Lyapack, donde pref especifica el tipo de matriz. Una desventaja de este enfoque es que cada función aceptará diferentes parámetros dependiendo del tipo de matriz con el que trabaje. Por ejemplo, la forma en que la función recibe la matriz variará en función del formato disperso en la que está almacenada. Esto significa que es difícil proveer un método genérico para llamar a la biblioteca desde el archivo Mex.

La solución adoptada en este trabajo incluye un archivo Mex para cada tipo distinto de almacenamiento de matriz. En Lyapack, las USFs están separadas en archivos que comparten un prefijo, y ninguna de estas funciones mantiene dependencias de datos con funciones de distinto prefijo. 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 único tipo de almacenamiento de matrices, y no con varios de ellos, por lo cual no es interesante compartir datos entre funciones que pertenecen a distintos tipos de matriz.

El archivo Mex tendrá entonces la estructura mostrada en la Figura III.

El archivo Mex actúa como interfaz entre Matlab y la biblioteca C/CUDA. En este sentido, no es adecuado mantener datos específicos de la biblioteca en el nivel del archivo Mex. Mantener las dos capas separadas permite que el mismo archivo Mex pueda trabajar con otras implementaciones de la biblioteca, por ejemplo implementaciones que utilicen OpenMP o MPI.

Como prueba de concepto, se implementaron y evaluaron las ideas previas para el caso particular de las matrices tridiagonales.

En esta implementación, las USFs tienen el siguiente comportamiento:

- tr_m_i: Transfiere la matriz tridiagonal A al dispositivo.
- tr_m: Realiza una multiplicación de matrices ordinaria. Esta operación no representa una fracción significativa del tiempo de ejecución, por lo que se ha optado por utilizar la implementación incluida en Matlab.

```
#include "cuda.h"
#include "mex.h"
#include "cudaLibrary.cuh"

/* Variables globales...  */

void mexFunction(int nlhs,
                 mxArray *plhs[],
                 int nrhs,
                 const mxArray *prhs[])
{

  const char *nm;
  int n,m;
  nm = mexFunctionName ();

  if (strcmp (nm, "pref_m_i") == 0){

    /* Obtener y procesar
    parametros desde Matlab */

    /* Inv. a las funciones de la libreria */

    pref_m_i(...);

  }else if (strcmp (nm, "pref_m") == 0){
    ...
  }else if (strcmp (nm, "pref_m_d") == 0){
    ...
  }else if (strcmp (nm, "pref_l_i") == 0){
    ...
  }else if (strcmp (nm, "pref_l") == 0){
    ...
  }else if (strcmp (nm, "pref_l_d") == 0){
    ...
  }else if (strcmp (nm, "pref_s_i") == 0){
    ...
  }else if (strcmp (nm, "pref_s") == 0){
    ...
  }else if (strcmp (nm, "pref_s_d") == 0){
    ...
  }

}
```

Fig. 1. Estructura del fichero Mex.

- tr_m_d: Libera la memoria que fue reservada en la función tr_m_i.
- tr_l_i: Usualmente algunas de las funciones no son necesarias para el caso que se desea abordar. En nuestro caso, esta función no realiza ninguna tarea, ya que no son necesarias factorizaciones matriciales u otro tipo de procesamiento.
- tr_l: Resuelve un sistema lineal del tipo $Ax = b$, donde A es una matriz tridiagonal. Internamente, esta rutina transfiere el vector b al dispositivo, seguidamente resuelve el sistema $Ax = b$, y finalmente transfiere el vector x al host y lo devuelve a Matlab.
- tr_l_d: No realiza ninguna acción ya que su función sería liberar la memoria reservada en tr_l_i.
- tr_s_i: Almacena un vector de parámetros en el espacio de memoria de la biblioteca CUDA, aunque el vector no es transferido inmediatamente al dispositivo. Estos parámetros son utilizados para resolver los sistemas de ecuaciones lineales desplazados $A + sI = b$.
- tr_s: Resuelve un sistema de ecuaciones lineales desplazado del tipo $A + sI = b$, donde A es tridiagonal. Tal como en tr_l, el vector b es transferido al dispositivo antes de resolver el sistema, y tras resolver el sistema, el vector resultado x es transferido al host.
- tr_s_d Libera la memoria dinámica reservada en la USF tr_s_i.

Para resolver los sistemas tridiagonales se utilizó el *solver* propuesto por Alfaro et al. [1] para aritmética de doble precisión. En problemas que involucran números complejos, se empleó además del comentado *solver*, la función *tsgv* de la biblioteca CuSparse [15] de nVidia.

IV. Resultados Experimentales

En esta sección se evalúa el rendimiento de la implementación de los *solvers* tridiagonales descritos en la Sección III.

Para evaluar la propuesta se adaptó como caso de prueba el denominado "demo_m11", distribuido con Lyapack. Este caso muestra la resolución de las ecuaciones de Lyapunov. El generador de las matrices fue adaptado para crear problemas artificiales tridiagonales.

El trabajo fue evaluado en una plataforma hardware híbrida compuesta por un microprocesador multinúcleo y una GPU. En particular, los experimentos fueron realizados con un procesador Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz y 16 Gb de memoria RAM, conectados vía bus PCI-e a una tarjeta nVidia C2070. Se utilizó Matlab R2011b y el compilador gcc en su versión 4.4.6. Para procesar el código CUDA se utilizó nvidia CUDA 4.1. Todos los experimentos fueron realizados utilizando aritmética compleja de doble precisión.

La Tabla I compara los tiempos de ejecución (en segundos) que insume resolver las ecuaciones tridiagonales de Lyapunov con la implementación original de Lyapack, que implementa el método LR-ADI y realiza la totalidad de los cómputos en la CPU, con los obtenidos con la nueva versión propuesta, que explota la capacidad de cómputo de la GPU. Adicionalmente, la tabla incluye los valores de aceleración sobre la versión original.

TABLA I
TIEMPOS DE EJECUCION (EN SEGUNDOS) Y ACELERACIÓN OBTENIDA CON LA VERSIÓN LYAPACK+GPU PROPUESTA

Dim. del problema	Lyapack	Lyapack + GPU	Aceleración
64K	0,38	0,07	5,4
128K	0,80	0,12	6,7
256K	1,56	0,22	7,1
512K	3,03	0,42	7,2
1024K	6,09	0,81	7,5

El análisis de los resultados de la Tabla I demuestra que, para la plataforma hardware evaluada, la rutina original toma al menos 5 veces más tiempo que la versión basada en GPU. Además, la aceleración obtenida aumenta con la dimensión del proble-

ma, alcanzando valores de hasta 7,5× para el caso más grande tratado.

Sin embargo, la inclusión de computación basada en GPU no abarca todas las etapas del método LR-ADI. A continuación se analizan los tiempos de ejecución de las distintas etapas lógicas de la versión original y de la basada en GPU, con el fin de identificar los valores reales de aceleración sobre aquellas fases en que la GPU participa. En particular, Lyapack+GPU hace un uso intensivo de la GPU durante la resolución de los sistemas lineales, mientras que el resto de etapas son íntegramente computadas en la CPU. Con esta intención, la Tabla II presenta los tiempos de ejecución de las dos versiones, discriminando los tiempos relacionados con la resolución de sistemas lineales y los tiempos correspondientes al resto de etapas del método LR-ADI.

Los resultados resumidos en la Tabla II muestran que la resolución de sistemas lineales es drásticamente acelerada mediante el empleo de la GPU. Así, la aceleración obtenida en esta fase está en torno a 7× para los casos de menores, y aumenta con la dimensión del problema hasta superar 10× en problemas cuya dimensión es mayor a 512k. Los resultados obtenidos también muestran como, si bien en la implementación original de Lyapack el tiempo dedicado a la resolución de sistemas representa más del 90 % del tiempo total, en Lyapack+GPU el tiempo dedicado a esta fase representa aproximadamente un 65 % del tiempo total.

Es importante notar que los tiempos de transferencia CPU-GPU están incluidos en los resultados exhibidos anteriormente. Además, el coste del solver tiene orden lineal tanto en cómputo como en transferencias. Teniendo esto en consideración, se plantea una evaluación detallada del tiempo de ejecución de los *solvers* tridiagonales. En este experimento se distinguen los tiempos dedicados a transferencias CPU-GPU y el tiempo de cómputo. Los resultados de esta evaluación se recogen en la Tabla III y muestran que el tiempo dedicado a las transferencias de datos es superior al tiempo de cómputo, de forma que aproximadamente un 65 % del tiempo correspondiente al *solver* en GPU está relacionado con las transferencias de datos, mientras que sólo el 33 % restante está relacionado con los cómputos efectuados en la GPU.

Considerando los resultados obtenidos en este estudio, es posible anticipar los beneficios que aportaría una versión Lyapack+GPU que minimice las transferencias de datos CPU-GPU. En este caso la aceleración obtenida podría ser superior a 13× para el proceso completo, y 30× para la fase de resolución de sistemas.

V. Conclusiones y trabajo futuro

En este trabajo se presenta una extensión de la biblioteca Lyapack para mejorar su desempeño computacional mediante la explotación de las capacidades de cómputo de las GPUs. El diseño e implementación de esta extensión fueron realizados siguiendo la filosofía original de la biblioteca, en la que impera su flexibilidad y sobre todo su facilidad de uso. Adicionalmente, se implementó una instancia de la biblioteca para resolver problemas tridiagonales de acuerdo con la solución propuesta.

Los resultados experimentales muestran una acusada aceleración de los cómputos al utilizar la GPU para resolver ecuaciones de Lyapunov tridiagonales. La versión Lyapack+GPU reporta un factor de aceleración de hasta 7,5× respecto a la implementación original en CPU. Si centramos el análisis a las fases del método en las que la GPU es empleada, estos factores de aceleración alcanzan 10,8× en los problemas de mayor dimensión evaluados.

Otra ventaja de la propuesta es su escalabilidad. Se observó que un incremento en la dimensión del problema reporta también un incremento en los valores de aceleración. Hay que notar que se detecta un sobrecoste elevado debido a las transferencias de datos entre CPU y GPU. Así, casi 2/3 del tiempo dedicado a la resolución de sistemas está dedicado a estas transferencias. Este hecho hace necesario el desarrollo de nuevas versiones que reduzcan el número de datos transferidos. Una versión que implementara el método LR-ADI completamente en la GPU podría reducir el impacto de las transferencias en el tiempo total, reportando aceleraciones sustancialmente superiores a las experimentadas con la implementación Lyapack+GPU.

Es necesario notar que la implementación de los métodos para matrices dispersas no estructuradas, así como para problemas dispersos simétricos generales, no fue discutida. Como trabajo futuro, se espera abordar estos problemas, incluyendo también la aceleración de otras etapas involucradas en un *solver* lineal disperso genérico mediante el uso de GPUs.

Agradecimientos

Los autores quieren agradecer a Martín Pedemonte por el soporte técnico brindado. Ernesto Dufrechou agradece al PEDECIBA por su apoyo financiero. Pablo Ezzatti recibió financiación del proyecto ANII_FMV_2009_3095. Enrique S. Quintana-Ortí y Alfredo Remón recibieron financiación del proyecto TIN2011-23283 del Ministerio de Economía y Competitividad y de FEDER.

Referencias

[1] Alfaro, P., Igounet, P. and Ezzatti, P. : *A study on the implementation of tridiagonal systems solvers using a GPU.* In SCCC 2011, Proceedings of the XXX International Conference of the Chilean Computer Science Society, 2011.

[2] Barrachina, S., Castillo, M., Igual, F., Mayo, R., Quintana-Ortí, E. and Quintana-Ortí, G. *Exploiting the capabilities fo modern GPUs for dense matrix computations.* Concurrency Computat.: Practice and Experience, 21, pp. 2457–2477, 2009.

[3] Benner, P., Ezzatti, P., Kressner, D., Quintana-Ortí, E. S. and Remón, A.: *Accelerating model reduction of large linear systems with graphics processors.* In Lecture Notes in Computer Science, State of the Art in Scientific and Parallel Computing, Springer, 2010.

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

[5] Benner, P., Ezzatti, P., Mena, H., Quintana-Ortí, E. S. and Remón, A.: *Solving differential Riccati equations on*

TABLA II

Tiempo insumido (en segundos) por la resolución de sistemas de ecuaciones lineales y otras etapas para ambas versiones.

Dim. del Problema	Lyapack		Lyapack+GPU		Aceleración
	system solvers	otras etapas	system solvers	otras etapas	system solvers
64K	0,35	0,03	0,05	0,02	7,4
128K	0,76	0,04	0,08	0,04	7,7
256K	1,48	0,08	0,15	0,07	9,6
512K	2,90	0,13	0,28	0,14	10,2
1024K	5,84	0,25	0,54	0,27	10,8

TABLA III

Tiempo de ejecución (en segundos) de la versión en GPU separando tiempos de transferencia y cómputo.

Dim. del problema	Lyapack+GPU		Lyapack	
	transfer.	cómputo	system solver	Aceleración
64K	0,03	0,02	0,35	17,6
128K	0,05	0,03	0,76	25,3
256K	0,09	0,05	1,48	29,7
512K	0,18	0,09	2,91	32,3
1024K	0,34	0,17	5,85	34,4

multi-GPU platforms. In 2nd Meeting on Linear Algebra, Matrix Analysis and Applications ALAMA10, 2010.

[6] Benner, P., Ezzatti, P., Mena, H., Quintana-Ortí, E. S. and Remón, A.: *Solving differential Riccati equations on multi-GPU platforms*. In 10th International Conference on Computational and Mathematical Methods in Science and Engineering CMMSE11, pp. 178–188, 2011.

[7] Benner, P., Ezzatti, Quintana-Ortí, E. S. and Remón, A.: *Using hybrid CPU-GPU platforms to accelerate the computation of the matrix sign function*. In Euro-Par Workshops, H.-X. Lin, M. Alexander, M. Forsell, A. Knüpfer, R. Prodan, L. Sousa, and A. Streit, eds., vol. 6043 of Lecture Notes in Computer Science, Springer, pp. 132–139, 2009.

[8] Benner, P., Ezzatti, P., Kressner ,D., Quintana-Ortí, E. S. and Remón, A.: *Accelerating BST methods for model reduction with graphics processors*. In Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics, 2011.

[9] Benner, P., Ezzatti, Quintana-Ortí, E. S. and Remón, A.: *High performance matrix inversion of SPD matrices on graphics processors*. In Workshop on Exploitation of Hardware Accelerators WEHA 2011, pp. 640–646, 2011.

[10] Dullerud, G.E. and Paganini, F.G.: *A Course in Robust Control Theory: A Convex Approach*. Texts in Applied Mathematics, Springer, 2000.

[11] Ezzatti, P., Quintana-Ortí, E. S., and Remón, A.: *High Performance Matrix Inversion on a Multi-core Platform with Several GPUs*. In 19th Euromicro International Conference, 2011.

[12] Golub, G and Loan, C. V. *Matrix Computations, 3rd Edition*, The Johns Hopkins University Press, Baltimore, 1996.

[13] Kirk, D. and Hwu, W., *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[14] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh: *Basic Linear Algebra Subprograms for FORTRAN usage*. ACM Trans. Math. Soft., 5, 1979, pp. 308–323.

[15] NVIDIA: *CUSPARSE User Guide 4.1*, 2012.

[16] Owens, J., Luebke,D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. AND Purcell,T.: *A Survey of General-Purpose Computation on Graphics Hardware*. Computer Graphics Forum, vol. 26, num. 1, pp. 80–113", 2007.

[17] Peinado, J., Ibáñez, J. J., Arias, E. and Hernández, V.: *Speeding up solving of differential matrix Riccati equations using GPGPU computing and MATLAB*. Concurrency Computat.: Practice and Experience. 2011, doi: 10.1002/cpe.1835

[18] Penzl, T.: *A Cyclic Low-Rank Smith Method for Large Sparse Lyapunov Equations*. SIAM J. Sci. Comput. 21, 4, 1401-1418, 1999.

[19] Penzl, T.: *LYAPACK A MATLAB Toolbox for Large Lyapunov and Riccati Equations, Model Reduction Problems, and Linear-Quadratic Optimal Control Problems. Users Guide (Version 1.0)*. 2000.

[20] Tomov, S., Dongarra, J., and Baboulin, M.: *Towards dense linear algebra for hybrid GPU accelerated manycore systems*. MIMS EPrint 2009.7, Manchester Institute for Mathematical Sciences, University of Manchester, Manchester, UK.