Solving dense linear systems with hybrid ARM+GPU platforms

Juan P. Silva, Ernesto Dufrechou, Pablo Ezzatti
Instituto de Computación (INCO)
Universidad de la República
Montevideo, Uruguay
{jpsilva,edufrechou,pezzatti}@fing.edu.uy

Enrique S. Quintana-Ortí
Dept. de Ingeniería y Ciencia de los Computadores
Universidad Jaime I
Castellón, Spain
quintana@icc.uji.es

Peter Benner, Alfredo Remón
Max Planck Institute for Dynamics of Complex Technical Systems
Magdeburg, Germany
{benner,remon}@mpi-magdeburg.mpg.de

***Resumen*—The necessity of reducing the energy consumption while improving the computational performance has encouraged the development of new hardware platforms. In this line, hybrid architectures that integrate ARM processors with graphics accelerators offer a positive balance between computing capabilities and energy requirements. However, in order to make an efficient use of this hardware, it is necessary to develop new methods and computational kernels, as well as to adapt existing ones. The solution of linear systems of equations is a basic operation in the solution of different problems. Its relevance and computational cost has motivated an important amount of work, and in consequence, it is possible to find high performance solvers for most hardware platforms. In this work we study the solution of dense linear systems of equations in an NVIDIA Jetson TK1 device via the Gauss-Huard method. The experimental evaluation shows that the new solvers outperform the ones available in the MAGMA library for systems of dimesion $n \leq 6{,}000$.**

***Keywords*—Linear Systems, Gauss-Huard, NVIDIA Jetson K1**

I. Introducción

La resolución de sistemas lineales de la forma $Ax = b$, donde $A \in \mathbb{R}^{n \times n}$ es la matriz de coeficientes, y los vectores $b, x \in \mathbb{R}^n$ representan el vector de términos independientes (vector del lado derecho) y el vector solución, respectivamente, es el principal problema en diversas aplicaciones científicas e ingenieriles [1]. Cuando la matriz de coeficientes A es densa, es decir la mayoría de los coeficientes son distintos de cero, el método más divulgado para la resolución del sistema está basado en la factorización LU (o en otras palabras, en la eliminación Gaussiana) [2]. Para asegurar la estabilidad numérica, este método debe ser completado con el uso de técnicas de pivoteo, en particular, se suele aplicar un pivoteo parcial de filas [3].

El método de la eliminación de Gauss-Jordan (GJE) para la inversión de matrices puede ser fácilmente adaptado a la resolución de sistemas lineales. De nuevo, la estabilidad numérica del método queda asegurada mediante una estrategia de pivoteo parcial por filas. El interés en este algoritmo radica en su alto nivel de paralelismo, permitiendo alcanzar niveles de eficiencia notables en las nuevas arquitecturas hardware que disponen de un elevado número de unidades computacionales [4], [5]. Sin embargo, su aplicación directa a la resolución de sistemas lineales incurre en un costo computacional (n^3 operaciones aritméticas de punto flotante, o flops) claramente superior al requerido por la resolución basada en la factorización LU ($2n^3/3$ flops). Esta debilidad del método fue abordada a finales de los años 70 por P. Huard, quien planteó una variante del método GJE para resolver sistemas lineales [6]. Esta variante, conocida como método Gauss-Huard (GH), presenta un coste computacional análogo al que supone el método basado en la factorización LU. Posteriormente, T. J. Dekker et al. [7] propusieron la inclusión de estrategias de pivoteo por columnas y demostraron que el método resultante presenta condiciones de estabilidad numéricas equivalentes a las ofrecidas por el método basado en la factorización LU utilizando pivoteo por filas.

En los últimos años las arquitecturas de hardware han sufrido notables cambios. Motivados principalmente por las limitaciones en el consumo energético, la estrategia de aumentar la frecuencia de los procesadores ha sido reemplazada por un aumento en el número de unidades computacionales. Tal es el caso de la familia de procesadores Intel Xeon Phi o los procesadores gráficos (GPUs). Estas nuevas arquitecturas presentan desde decenas hasta miles de unidades de cómputo y sus requerimientos energéticos son menores a los de un procesador multi-núcleo de propósito general. En este ámbito destacan las soluciones de NVIDIA que integran en una única placa procesadores de bajo consumo ARM y una tarjeta gráfica también de bajo consumo. Un ejemplo es la tarjeta Jetson TK1, compuesto por un procesador quad-core ARM Cortex A15 y un procesador gráfico Kepler con 192 cores. Este dispositivo presenta un consumo de solo 12 Watt y ofrece unas prestaciones de 300 GFLOPS (miles de millones de flops), en aritmética de simple precisión y de 24 GFLOPS en doble precisión. Extraer las mayores prestaciones de estas nuevas plataformas híbridas exige al programador un esfuerzo de adaptación tanto de los métodos como de las codificaciones.

En este trabajo se estudia la implementación del método de Gauss-Huard con pivoteo por columnas para resolver sistemas lineales sobre una tarjeta Jetson TK1. Las soluciones propuestas son evaluadas y comparadas frente a herramientas que representan el estado del arte en plataformas híbridas CPU-GPU, en particular con OpenBLAS [8] y MAGMA [9]. Los resultados experimentales muestran que las nuevas

978-1-4673-9143-6/15/$31.00

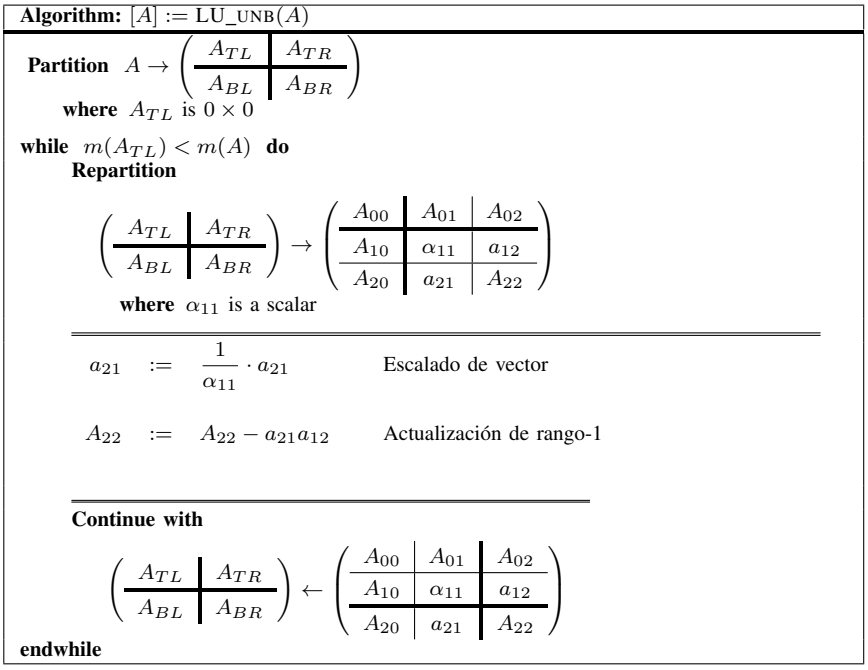

Figura 1. Algoritmo escalar (sin bloques) e *in-place* para computar la factorización LU de la matriz A. Al terminar el algoritmo, A está sobre-escrita con los factores L y U.

variantes son capaces de competir con los resolutores incluídos en las bibliotecas mencionadas, e incluso llegan a mejorar sus resultados en la resolución de sistemas con dimensión $n \leq 6{,}000$.

El resto del artículo se estructura de la siguiente manera. En la Seccción II se describen tres métodos para resolver sistemas lineales, la factorización LU, el método basado en GJE y el método de GH. A continuación, en la Sección III se presentan las diferentes versiones propuestas, y su evaluación experimental se analiza en la Sección IV. Finalmente, en la Sección V, se resumen las principales conclusiones y se discuten algunas líneas de trabajo futuro.

II. Resolución de sistemas lineales densos

En esta sección se muestran diferentes métodos para resolver sistemas lineales densos. En computación científica hay dos grandes familias de métodos para resolver sistemas lineales de la forma:

$$Ax = b \tag{1}$$

donde la matriz $A \in \mathbb{R}^{n \times n}$ y $x, b \in \mathbb{R}^n$, los métodos directos y los iterativos (ver por ejemplo [2]). La primera familia está compuesta por métodos determinísticos que computan la solución exacta (a excepción de errores numéricos), mientras que los métodos iterativos parten de una solución inicial y avanzan en sucesivas aproximaciones convergiendo a la solución real. Cuando la solución alcanzada es suficientemente próxima a la solución exacta el método iterativo se detiene. Desde el punto de vista computacional, generalmente los métodos directos están basados en operaciones de tipo BLAS3, permitiendo un uso eficiente de la jerarquía de memoria en las plataformas hardware actuales. Por el contrario, los métodos iterativos están basados en operaciones BLAS2, y por lo tanto ofrecen prestaciones menores. No obstante, si la convergencia es rápida o no se requiere gran precisión en la solución, es posible que un método iterativo presente un coste computacional menor y por lo tanto pueda ser más rápido que un método directo. Este trabajo se centra en los métodos directos para la resolución de sistemas lineales de ecuaciones densos.

II-A. Resolución de sistemas lineales mediante la factorización LU

El método tradicional para resolver sistemas lineales densos se despreden de la eliminación Gaussiana. Específicamente, esta técnica tiene como primer paso la factorización LU de la matriz A. Para mantener la estabilidad numérica durante la factorización se emplean técnicas de pivoteo por filas [2]. En la práctica la factorización calculada toma la forma $PA = LU$, donde $P \in \mathbb{R}^{n \times n}$ es una matriz de permutación y los factores $L, U \in \mathbb{R}^{n \times n}$ son, respectivamente, triangular inferior unitario y triangular superior. Esta operación presenta un coste espacial de aproximadamente $3n^2$ flops, aunque se puede reducir a $n^2 + n$ si la matriz de permutaciones, P, se almacena de forma implícita mediante un vector y los valores de los factores reemplazan a los de la matriz A[1]. Una vez calculada la factorización de A, el sistema original es equivalente a $LUx = (Pb) = \hat{b}$, y x puede ser obtendio mediante la resolución sucesiva de los sistemas lineales triangulares $Ly = \hat{b}$ y $Ux = y$.

En este método la mayor parte del costo computacional está concentrado en la factorización de la matriz A. La Figura 1 muestra un pseudocódigo de una versión escalar *in-place* de

[1]La diagonal de L no se almacena explícitamente ya que esta formada únicamente por unos.

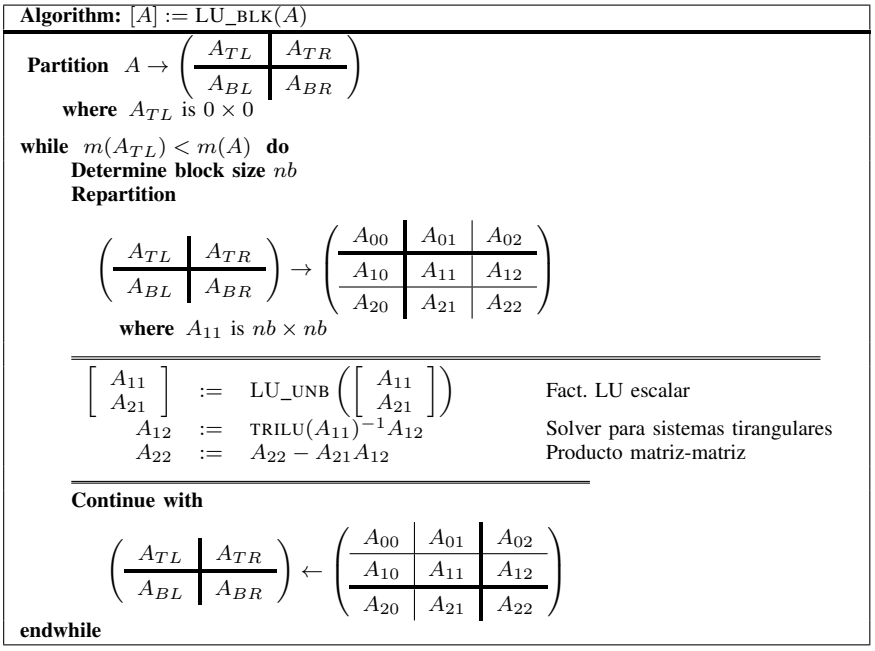

Figura 2. Algoritmo a bloques e *in-place* para computar la factorización LU de la matriz A. Al terminar el algoritmo, A está sobre-escrita con los factores L y U.

la eliminación Gaussiana para computar la factorización LU usando la notación FLAME: por más detalles ver [10], [11]. En la figura, $m(A)$ toma como valor el número de columnas de la matriz A. Por simplicidad no se incluye en la descripción del algoritmo la aplicación de pivoteo. Por otro lado, la Figura 2 presenta la versión a bloques del método. En general, nb se conoce como el tamaño de bloque del algoritmo. Este tipo de variantes a bloques, ofrecen en general mayor desempeño en las arquitecturas de hardware modernas debido a que se mejora la intensidad computacional [2].

El método basado en la factorización LU presenta ciertas características no muy propicias para su implementación sobre arquitecturas de hardware masivamente paralelas y que impactan en forma negativa en el desempeño alcanzado.

- Se trata de un método con tres etapas que deben ser ejecutadas secuencialmente, a saber: la factorización LU, la resolución del sistema lineal triangular inferior y la resolución el sistema lineal triangular superior. Esta característica implica un mínimo de tres puntos de sincronización.
- Se requiere de la resolución de dos sistemas lineales triangulares que, si bien son operaciones de tipo BLAS3, implican una cantidad importante de dependencias de datos y, por lo tanto, ofrecen un nivel inferior de concurrencia. Además, durante el proceso de factorización también aparecen factores triangulares (de pequeña dimensión).

No obstante, estos inconvenientes toman menor relevancia cuando la dimensión de A crece, dado que la factorización concentra la mayor parte del costo computacional.

LAPACK [12] es una especificación que recoge las principales operaciones de álgebra lineal numérica (ALN), y se dispone de implementaciones de altas prestaciones para las principales plataformas de cómputo. Estas implementaciones incluyen técnicas de computación de alto desempeño (HPC) así como mejoras específicas para cada procesador. La funcionalidad ofrecida por LAPACK cubre las necesidades requeridas para los distintos pasos del método. En particular, la rutina GETRF computa la factorización LU (aplicando pivoteo parcial por filas) de una matriz no singular, mientras que la rutina GETRS permite resolver sistemas lineales triangulares almacenados en la forma que los computa la rutina GETRF. Por último, LAPACK incluye la rutina GESV para la resolución de sistemas lineales que simplemente realiza las correspondientes llamadas a GETRF y GETRS.

II-B. El método de Gauss-Jordan

Otra alternativa para la resolución de sistemas lineales densos es la aplicación del método de Gauss-Jordan (GJE). Este algoritmo calcula la inversa de una matriz, pero puede adaptarse fácilmente para la solución de sistemas lineales. Además, al incluir técnicas de pivoteo por filas presenta unas propiedades matemáticas similares a las del método basado en la eliminación Gaussiana [3]. La resolución de sistemas lineales con este método implica formar la matriz $\hat{A} = [A, b]$ y aplicar el método GJE sobre la misma. Este método recorre las columnas de la matriz de izquierda a derecha actualizando, en cada iteración, la columna activa y también el resto los elementos de $\hat{A}$. En su aplicación a la resolución de sistemas lineales, sólo los elementos a la derecha de la columna activa necesitan ser actualizados, reduciendo así el coste del algoritmo a la mitad. Una vez concluido el proceso, la solución del sistema lineal (es decir, el vector x) está almacenada en la última columna de la matriz extendida.

La Figura 3 presenta la versión a bloques del método GJE para resolver el sistema lineal formado por la matriz A usando, nuevamente, la notación de FLAME y la representación de Matlab para expresar la concatenación por columnas

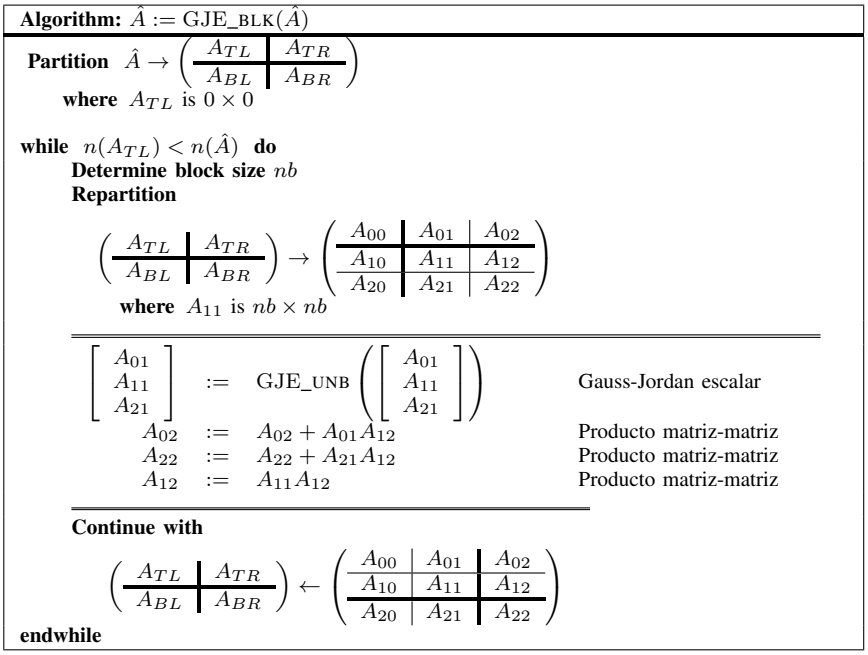

Figura 3. Algoritmo de Gauss-Jordan a bloques para resolver el sistema lineal $Ax = b$. A la entrada, $\hat{A} = [A, b]$, y cuando se termina la última columna de $\hat{A}$ está sobre-escrita con la solución x.

de matrices. Si bien típicamente se le aplican técnicas de pivoteo parcial por filas al método para alcanzar la estabilidad numérica, por simplicidad no se agregan en el esquema. Durante las actualizaciones, el método de GJE utiliza únicamente matrices rectangulares. Este algoritmo es muy conveniente para su implementación sobre plataformas hardware paralelas, ya que realiza el grueso de los cálculos en forma de productos de matrices. El producto de matrices es una operación BLAS3 con un elevado nivel de concurrencia que ofrece magníficas prestaciones en las arquitecturas hardware actuales. Sin embargo, el método implica un coste de n^3 flops, mientras que el basado en la factorización LU necesita de $2/3n^3$ operaciones de punto flotante. En consecuencia, este método no puede competir con el método basado en la LU en la resolución de sistemas de gran dimensión

Una descripción de la versión escalar del método GJE, que es invocada en la versión a bloques, se puede encontrar en [13]. Es importante hacer notar que, cuando se termina el proceso, el espacio de almacenamiento donde originalmente se tenía la matriz A se sobreescribe con la matriz transformada (es decir es un método *in-place*).

II-C. El método de Gauss-Huard

Si bien el método de Gauss-Jordan presenta diferentes bondades que le permiten explotar de manera eficiente los recursos hardware de las plataformas masivamente paralelas, su empleo para la resolución de sistemas lineales implica un sobrecoste que lo hace inviable frente a otros métodos, como por ejemplo el basado en la factorización LU, que presenta un coste un 50 % menor. El inconveniente del sobrecoste fue abordado y resuelto por P. Huard a finales de los años 70. En [6], P. Huard presentó una variante del método GJE para resolver sistemas lineales. Este método, conocido como Gauss-Huard (GH) presenta un costo computacional análogo al requerido por la resolución de sistemas lineales mediante la eliminación Gaussiana. Posteriormente, T. J. Dekker et al. [14], [7] propusieron la inclusión de estrategias de pivoteo por columnas y demostraron que el método resultante ofrece unas condiciones de estabilidad numéricas equivalentes al del uso de la factorización LU (utilizando pivoteo por filas).

La Figura 4 describe de forma algorítmica el método GH para la resolución del sistema lineal denso definido como $Ax = b$. Al igual que en los casos anteriores, por simplicidad no se describe la aplicación de la técnica de pivoteo. En cualquier caso, la inclusión del pivoteo por columna precisa de cambios mínimos; en particular, antes de realizar la diagonalización de $\left[\hat{\alpha}_{11},\ \hat{a}_{12}^T\right]$, se busca en este vector la entrada de mayor magnitud (excluyendo el último elemento, que contiene un valor del vector b) y la columna de $\hat{A}$ correspondiente a dicha entrada es intercambiada con la columna $\hat{A}$ que contiene el valor de la diagonal $\hat{\alpha}_{11}$.

Una versión a bloques del método GH fue introducida posteriormente en el contexto de paralelismo de memoria distribuida (sistemas de pasaje de mensajes en [15]). Desafortunadamente, los autores no realizaron ninguna evaluación experimental de la implementación, y únicamente indicaron que las prestaciones alcanzadas por este algoritmo se preveían similares a las presentadas por resolutores basados en la factorización LU. En la Figura 5 se describe el método de GH a bloques, el algoritmo procesa nb columnas de la matriz en cada iteración del bucle.

III. Implementación de Gauss-Huard eficiente para arquitecturas híbridas

Las plataformas de hardware híbridas del tipo CPU-GPU han alcanzado una posición relevante en el área de la computación de altas prestaciones, y en especial en el campo de

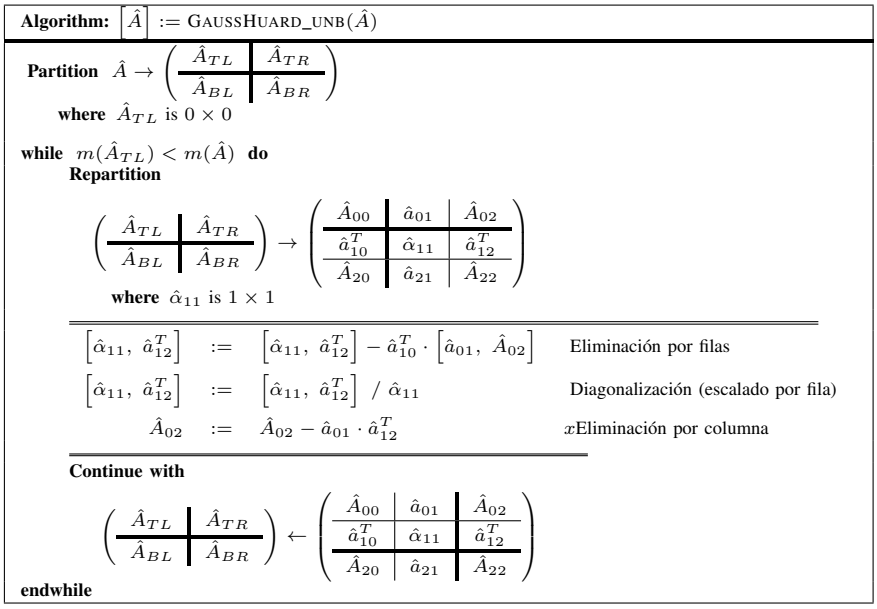

Figura 4. Algoritmo Gauss-Huard escalar (sin bloques) para resolver el sistema lineal $Ax = b$. A la entrada, $\hat{A} = [A, b]$, y cuando se termina la última columna de $\hat{A}$ está sobre-escrita con la solución x.

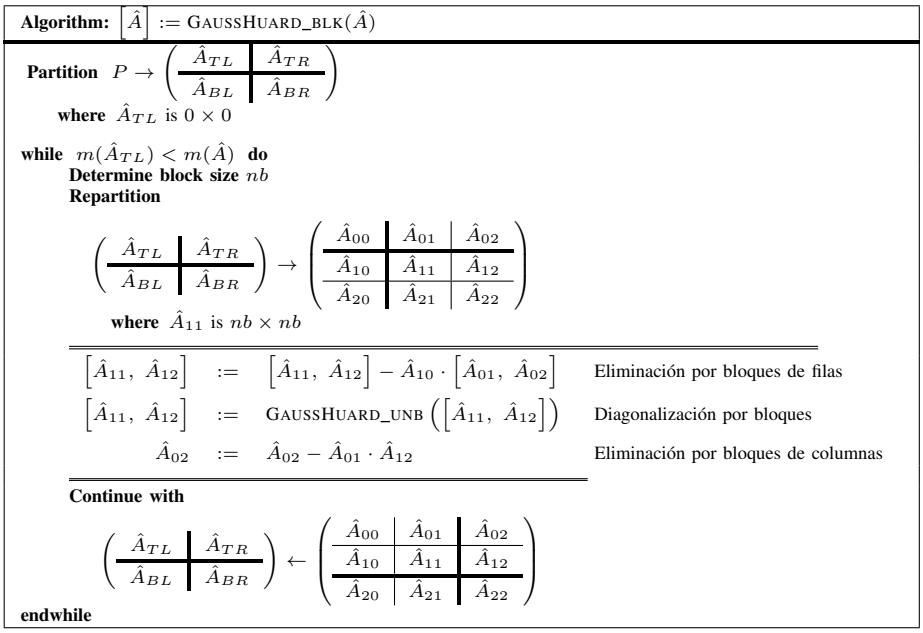

Figura 5. Algoritmo de Gauss-Huard por bloques para resolver el sistema lineal $Ax = b$. En la entrada, $\hat{A} = [A, b]$, y a la salida la última columna de $\hat{A}$ está sobre-escrita con la solución x.

aplicaciones de computación numérica y de álgebra lineal. Esta relevancia ha sido refrendada con la aparición de distintas bibliotecas numéricas específicas para estas plataformas. Entre otras, destacan MAGMA [16], PETSc [17] o ViennaCL [18], que actualmente incluyen soporte para la resolución de sistemas lineales (densos o dispersos) de manera eficiente en plataformas con arquitectura híbridas conformadas por un procesador de propósito general y un acelerador hardware (como por ejemplo un procesador gráfico o GPU). Adicionalmente, las GPUs destacan por sus altas prestaciones y por su bajo consumo energético, lo cual ha motivado el desarrollo de plataformas que aunan procesadores de propósito general de bajo consumo (como los creados por ARM) con GPUs. Estas plataformas ofrecen unas prestaciones destacables y al mismo tiempo un consumo energético reducido. Estas plataformas ya están siendo empleadas para la creación de potentes computadores de bajo consumo, como por ejemplo el supercomputador Tibidabo del Barcelona Supercomputer Center [19].

En este contexto, el presente trabajo presenta y evalúa diferentes implementaciones del método GH sobre una arquitectura híbrida CPU-GPU de bajo consumo energético. Específicamente, se presentan dos variantes, una que se ejecuta

completamente en la GPU y otra que hace uso de la arquitectura híbrida completa. Las variantes descomponen el Algoritmo GAUSSHUARD_BLK en una colección de operaciones, y dependiendo de la variante, se ejecutan en forma completa en la GPU (en el caso GH_{gpu}) o se ejecuta cada operación del algoritmo en el dispositivo más apropiado (versión GH_{hib}).

En ambas variantes, la matriz extendida ($\hat{A} = [A, b]$) se transfiere inicialmente desde el espacio de memoria de la CPU a la memoria de la GPU. De esta manera se evita, en cada iteración del algoritmo a bloques, realizar la transferencia del panel $[\hat{a}_{11}, \hat{a}_{12}]$ de CPU a GPU. Dado que en la arquitectura destino realizar una única transferencia de un gran volumen de datos en general es más eficiente que hacer varias transferencias de pocos datos, con esta estratégia se prevé ahorrar tanto tiempo de ejecución como energía.

III-A. Variante en GPU (GH_{gpu}*)*

Teniendo en cuenta el modesto poder computacional que ofrecen los procesadores tipo ARM que están incluidos en la arquitectura objetivo (en comparación con el ofrecido por la GPU), en esta variante se busca realizar todas las operaciones en la GPU. De esta manera, además, se evitan las costosas transferencias de datos entre la CPU y la GPU en cada paso de la iteración. En mayor detalle, esta variante se basa en el Algoritmo GAUSSHUARD_BLK, utilizando los núcleos computacionales que ofrece la biblioteca CUBLAS, que no es más que una implementación de BLAS desarrollada por NVIDIA para sus GPUs.

Como se comentó en la sección anterior, el algoritmo GAUSSHUARD_BLK concentra la carga computacional en los productos de matrices. Para alcanzar buenas prestaciones durante la ejecución de esta operación en la GPU se precisa que las matrices implicadas no sean pequeñas, lo que llevado al algoritmo GAUSSHUARD_BLK implica que el valor de nb no sea demasiado pequeño. Por otro lado, aumentar nb da mayor relevancia a la actualización del bloque $[\hat{A}_{11}, \hat{A}_{12}]$, que es una operación BLAS-2 que presenta prestaciones limitadas. La solución implementada en GH_{gpu} es el uso de múltiples tamaños de bloque de manera anidada. Específicamente, en primera instancia el algoritmo GAUSSHUARD_BLK incluye como resolutor para el bloque central el mismo GAUSSHUARD_BLK, convirtiendo esta actualización en una operación BLAS-3. Para alcanzar buenas prestaciones en ambas implementaciones de GAUSSHUARD_BLK, nb debe tomar un valor lo suficientemente alto. De esta forma se mejora la intensidad computacional y se alcanzan mejores niveles de desempeños en las arquitecturas modernas [20].

III-B. Variante híbrida (GH_{hib}*)*

Esta nueva variante utiliza todos los recursos disponibles en la plataforma. Así, las operaciones que presentan mayor intensidad computacional se ejecutan en la GPU mientras que las restantes se ejecutan en la CPU. Específicamente, las operaciones de actualización (multiplicación de matrices) que exiben alta intensidad computacional y buenos niveles de concurrencia permiten aprovechar la arquitectura de la GPU. En contraste, la diagonalización del bloque presenta un moderado costo computacional y sus niveles de concurrencia están fuertemente limitados por las altas dependencias de datos. Por lo tanto, se puede esperar mejores prestaciones si estas operaciones son ejecutadas en el procesador ARM que si se ejecutan sobre la GPU. Alcanzar mejores prestaciones en estas operaciones permitirá, así mismo, aumentar el valor de nb y por ende mejorar las prestaciones de los productos de matrices en GAUSSHUARD_BLK. Por contra, la utilización de ambas arquitecturas precisa de comunicaciónes y sincronizaciones entre ambos procesadores, lo que obviamente conlleva un coste. En particular, en cada iteración del Algoritmo GAUSSHUARD_BLK el bloque $[\hat{a}_{11},\ \hat{a}_{12}]$ debe ser transferido desde la memoria de GPU a la de CPU y, tras ser factorizado, debe ser enviado de nuevo a la memoria de la GPU. Por lo tanto, el volumen de transferencias de datos en cada iteración es proporcional a n y nb. Cuando se completa el procedimiento, el vector solución (almacenado en la última columna de la matriz $\hat{A}$) tiene que ser devuelto desde el espacio de memoria de la GPU al de CPU.

Al igual que sucede en GH_{gpu}, la eficiencia de la variante híbrida depende fuertemente de los valores elegidos para nb: valores demasiado pequeños, $nb \approx 32$, o 64, reducen el desempeño de las multiplicaciones de matrices que son derivadas a la GPU, pero asegura que la ejecución del algoritmo esté determinada por el cómputo y no por las transferencias de datos. Por otro lado, valores demasiado grandes de este parámetro afectan negativamente al desempeño del procedimiento completo, ya que mueven una parte significante de las operaciones del algoritmo al procesador ARM (el menos potente). Para abordar este escenario, se propone una implementación que emplea dos tamaños de bloques, nbd and nbg, para las operaciones en el procesador ARM y las opearaciones en la GPU, respectivamente. Esta estrategia permite utilizar el tamaño de bloque óptimo para cada arquitectura independientemente. Concretamente, esta implementación lleva a cabo la diagonalización a bloques de $[\hat{A}_{11}, \hat{A}_{12}]$ en GAUSSHUARD_BLK usando una variante a bloques de este algoritmo, que corre exclusivamente en CPU, con tamaños de bloque tales que $nbd < nb$. Por encima de esto, el Algoritmo GAUSSHUARD_BLK opera con el tamaño de bloque $nbg = nb$, que determina una de las tres dimensiones de la multiplicación de matrices involucrada en la eliminación bloque-fila y bloque-columna.

En esta variante, los núcleos básicos en GPU son resueltos mediante la invocación de operaciones de la biblioteca CUBLAS. En el ARM, las operaciones de tipo BLAS-3 son abordadas mediante el uso de los núcleos computacionales en la biblioteca OpenBLAS. Por otro lado, y debido a su mejor desempeño, las operaciones de tipo BLAS-1 y BLAS-2 en ARM son resueltas mediante códigos desarrollados *ad-hoc* y paralelizados mediante OpenMP.

Adicionalmente, la transferencia inicial de la matriz a la memoria de la GPU se realiza de forma simultánea con la diagonalización del primer panel en el procesador ARM, reduciendo de esta forma el impacto negativo de las transferencias en el desempeño.

IV. Evaluación Experimental

En esta sección se presenta la evaluación experimental de los resolutores basados en el Algoritmo GH presentados en la sección anterior. Los desempeños obtenidos son comparados

Tabla I. TIEMPO DE EJECUCIÓN (EN SEGUNDOS) Y DESEMPEÑO (EN BILLONES DE FLOPS/SEGUNDOS O GFLOPS) OBTENIDOS POR LOS CUATRO SOLVER EVALUADOS PARA PROBLEMAS QUE VARÍAN EN LA DIMENSIÓN n.

	GH_{hib}		GH_{gpu}		MAGMA		OpenBLAS	
	$GFLOPS$	$T_{ej.}$(s)	$GFLOPS$	$T_{ej.}$(s)	$GFLOPS$	$T_{ej.}$(s)	$GFLOPS$	$T_{ej.}$(s)
1K	1.0	0.721	1.9	0.368	0.60	1.126	3.4	0.211
2K	3.5	1.637	2.3	2.526	2.2	2.640	3.4	1.689
3K	5.5	3.499	1.7	11.703	3.1	6.147	3.8	5.077
4K	6.7	6.867	2.8	16.214	4.3	10.772	3.6	12.730
5K	7.5	11.923	2.6	34.781	5.8	15.449	3.9	23.017
6K	7.8	19.928	3.1	49.820	7.1	21.651	3.7	42.197
7K	8.0	30.663	1.5	160.056	8.3	29.749	4.0	62.090

con los ofrecidos por los resolutores basados en la factorización LU de las bibliotecas MAGMA [9] y OpenBLAS. Todos las rutinas emplean aritmética de doble precisión.

IV-A. Plataforma de ejecución

La plataforma JETSON TK1 [21] se compone de un procesador Tegra K1 SOC que dispone de 2GB de RAM DDR3L 933 MHz. A su vez el procesador Tegra K1 está formado por una GPU con arquitectura Kepler de 192 CUDA cores que alcanza los 13,5 GFlops, y un procesador ARM quad-core Cortex-A15 a 2.32 GHz.

El sistema operativo, es una versión adaptada de Ubuntu 14.04 para la arquitectura ARM. Todos los códigos ejecutados sobre el procesador ARM fueron compilados con el compilador *gcc* v.4.8.2, mientras que los códigos sobre el dispositivo de NVIDIA se compilaron mediante *nvcc* v.6.0.1. Adicionalmente se emplearon rutinas de las bibliotecas OpenBLAS y CUBLAS en las operaciones ejecutadas en la CPU y la GPU respectivamente.

IV-B. Resolutores de referencia

Los nuevos códigos se comparan con dos resolutores de referencia. Ambos implementan el método basado en la factorización LU. El primero es el incluido en la biblioteca OpenBLAS (en su última versión (0.2.14) que incluye optimizaciones específicas para los procesadores de ARM) y emplea únicamente el procesador ARM, que es el procesador menos potente en la plataforma. Por lo tanto se espera que este resolutor ofrezca peores prestaciones. El segundo es el resolutor de sistemas lineales en la biblioteca MAGMA en su versión 1.6.2. Esta implementación hace uso tanto del procesador ARM como del NVIDIA. Además, utiliza la biblioteca OpenBLAS para extraer mejor rendimiento en las operaciones ejecutadas en el procesador ARM.

Como casos de prueba se utilizan sistemas lineales formados por matrices de coeficientes y vectores independientes densos generados de forma aleatoria. Las dimensiones de los sistemas varían entre $n = 1024$ y $n = 7168$. Todas los cálculos se realizan utilizando aritmética de doble precisión.

IV-C. Resultados Experimentales

Los resultados experimentales mostrados en esta sección se corresponden, en todos los casos, con el promedio extraído de 10 ejecuciones independientes. Los tiempos mostrados tanto para las variantes que se ejecutan completamente en la GPU como en las variantes híbridas, incluyen los tiempos de intercambio de información entre los espacios de memoria de los dos procesadores.

En primera instancia, es importante destacar que todas las variantes obtienen resultados numéricos completamente comparables. Es decir, el error residual del vector solución es análogo en todas ellas.

En cuanto a la evaluación del desempeño computacional, la Tabla I resume los tiempos de ejecución ($T_{ej.}$) en segundos y los GFLOPS obtenidos por las cuatro variantes evaluadas, es decir las dos implementaciones desarrolladas, GH_{gpu} y GH_{hib}; y los resolutores considerados el estado del arte en la temática (los proporcionados por las bibliotecas OpenBLAS y MAGMA). Se evaluaron diferentes tamaños de bloque (tanto interno como externo) en las variantes desarrolladas y la utilización de diferente cantidad de hilos en el procesador ARM para todas las variantes. En este sentido, en la tabla sólo se presentan los mejores resultados obtenidos para cada variante (y dimensión de sistema). La Figura 6 representa de forma gráfica los mismos resultados. De las cuatro variantes evaluadas, únicamente GH_{gpu} no es la mejor solución para ninguna de los casos probados. Esta variante emplea el procesador más potente en la plataforma y no incurre en sobrecostes de coordinación de los procesadores. Además presenta un sobrecoste debido a transferencias de datos menor que el de las versiones híbridas, lo que a priori le situaba como un buen candidato a la resolución de sistemas de pequeña dimensión, pero para este tipo de problemas OpenBLAS ofrece mejores prestaciones. Esto se debe a que este caso no presenta un gran coste computacional y, si bien OpenBLAS utiliza sólo el procesador ARM, éste no precisa de transferencia de datos. En los casos $1024 < n < 7168$, GH_{hib} es la variante más rápida. Esto se debe a que, conforme aumenta n, la carga computacional del resolutor también lo hace. En tal caso, el empleo de ambos procesadores le otorga una ventaja sobre las variantes que sólo utilizan uno de los procesadores (es decir, GH_{gpu} y OpenBLAS). Por otro lado, el algoritmo de GH tiene cierta ventaja sobre la rutina de MAGMA. Finalmente, cuando $n \geq 7168$ MAGMA ofrece las mejores prestaciones. Esto se debe al gran esfuerzo de optimización tras esta biblioteca y al algoritmo detrás de cada implementación. En particular, GH requiere menos sincronizaciones y no precisa de la resolución de sistemas triangulares, pero estas ventajas pierden relevancia cuando n aumenta, y en tal caso el alto nivel de optimización de las rutinas en MAGMA es clave.

V. CONCLUSIONES Y TRABAJO FUTURO

En el trabajo se ha abordado la implementación de un resolutor de sistemas lineales densos para arquitecturas híbridas que incluyen procesadores tipo ARM y GPUs. En particular, se estudió la implementación del método de Gauss-Huard sobre el dispositivo JETSON K1 (procesador ARM quad-core + GPU

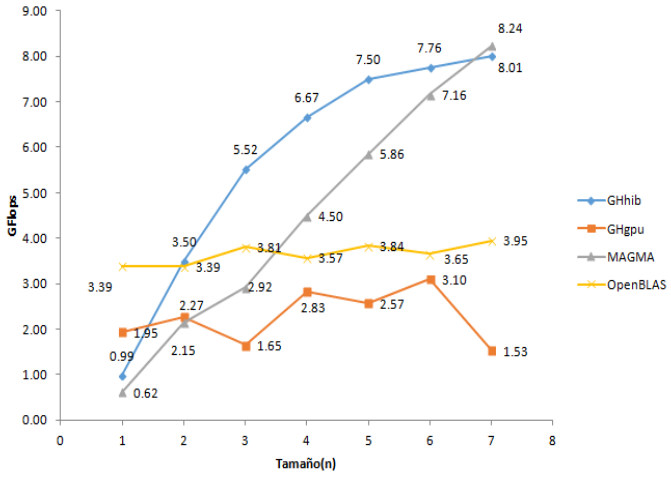

Figura 6. GFLOPS alcanzado por las distintas variantes para resolver sistemas lineales

de 192 CUDA cores).

Se desarrollaron dos variantes del método de Gauss-Huard, una que se ejecuta solo en GPU y otra que hace uso de la plataforma híbrida por completo. La evaluación experimental realizada muestra que la variante híbrida desarrollada supera para los casos pequeños y medianos al resolutor híbrido que es el estado del arte, MAGMA. Para los casos grandes, las desventajas presentes en los resolutores basados en la factorización LU se ven disimuladas por la alta importancia que cobran las operaciones de tipo multiplicación de matrices. Por otro lado, cuando se compara con el resolutor que utiliza únicamente el procesador ARM (OpenBLAS), la variante GH_{hib} es claramente superada para los casos pequeños (situación previsible dado el sobrecosto implicado por las transferencias de datos que no se compensan con las operaciones requeridas) pero ofrece mejores desempeños para los casos medianos y grandes. En resumen, si bien la propuesta es competitiva con el estado del arte, para casos menores a 2048 es mejor utilizar OpenBLAS y, a partir de sistemas de 7168, MAGMA parece ofrecer desempeños algo mejores.

Diferentes aspectos que no fueron cubiertos en el presente trabajo merecen un tratamiento en mayor profundidad. Entre otros, parece importante estudiar la aplicación de otras estrategias de optimización para códigos en arquitecturas híbridas, entre las que se destaca el uso de técnicas de *look-ahead*. En otro sentido, parece también interesante evaluar los resolutores desde un punto de vista de consumo energético, ya que este tipo de plataformas de hardware están especialmente enfocadas a este aspecto.

AGRADECIMIENTOS

Juan Pablo Silva, Ernesto Dufrechou y Pablo Ezzatti agradecen al Programa de Desarrollo de las Ciencias Básicas, y la Agencia Nacional de Investigación e Innovación, Uruguay. Peter Benner y Alfredo Remón agradecen el soporte recibido por el proyecto EHFARS, promovido por el ministerio de Educación e Investigación de Alemania, BMBF.

REFERENCIAS

[1] J. W. Demmel, *Applied Numerical Linear Algebra*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997.

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

[3] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002.

[4] 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 *Euro-Par 2009, Parallel Processing - Workshops*, ser. Lecture Notes in Computer Science, H.-X. Lin, M. Alexander, M. Forsell, A. Knupfer, R. Prodan, L. Sousa, and A. Streit, Eds. Springer-Verlag, 2009, no. 6043, pp. 132–139.

[5] P. Benner, P. Ezzatti, E. S. Quintana-Ortí, and A. Remón, "Matrix inversion on CPU-GPU platforms with applications in control theory," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 8, pp. 1170–1182, 2013. [Online]. Available: http://dx.doi.org/10.1002/cpe.2933

[6] P. Huard, "La méthode simplex sans inverse explicite," *EDB Bull, Direction Études Rech. Sér. C Math. Inform.* 2, pp. 79–98, 1979.

[7] T. J. Dekker, W. Hoffmann, and K. Potma, "Stability of the Gauss-Huard algorithm with partial pivoting," *Computing*, vol. 58, pp. 225–244, 1997.

[8] Z. Xianyi, http://www.openblas.net/.

[9] B. J. Smith, "R package MAGMA: Matrix algebra on GPU and multicore architectures, version 0.2.2," September 3, 2010, [On-line] http://cran.r-project.org/package=magma.

[10] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, "FLAME: Formal linear algebra methods environment," *ACM Trans. Math. Soft.*, vol. 27, no. 4, pp. 422–455, 2001.

[11] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn, "The science of deriving dense linear algebra algorithms," *ACM Trans. Math. Soft.*, vol. 31, no. 1, pp. 1–26, 2005.

[12] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*. Philadelphia: SIAM, 1992.

[13] E. Quintana-Ortí, G. Quintana-Ortí, X. Sun, and R. de Geijn@, "A note on parallel matrix inversion," *SIAM J. Sci. Comput.*, vol. 22, pp. 1762–1771, 2001.

[14] T. J. Dekker, W. Hoffmann, and K. Potma, "Parallel algorithms for solving large linear systems," *Journal of Computational and Applied Mathematics*, vol. 50, no. 1–3, pp. 221–232, 1994.

[15] W. Hoffmann, K. Potma, and G. Pronk, "Solving dense linear systems by Gauss-Huard's method on a distributed memory system," *Future Generation Computer Systems*, vol. 10, no. 2–3, pp. 321–325, 1994.

[16] Univ. of Tennessee, http://icl.cs.utk.edu/magma/.

[17] S. Balay, W. Gropp, L. C. McInnes, and B. Smith, "PETSc 2.0 users manual," Argonne National Laboratory, Tech. Rep. ANL-95/11, October 1996.

[18] TU Wien and FASTMathSciDAC Institute, http://viennacl.sourceforge.net/.

[19] Barcelona Supercomputing Center, http://www.bsc.es.

[20] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[21] NVIDIA, http://developer.download.nvidia.com/embedded/jetson/TK1/docs/Jetson_platform_brief_May2014.pdf.