UNIVERSIDAD JAUME I DE CASTELLÓN
E. S. DE TECNOLOGÍA Y CIENCIAS EXPERIMENTALES

CONSUMO ENERGÉTICO DE MÉTODOS ITERATIVOS PARA SISTEMAS DISPERSOS EN PROCESADORES GRÁFICOS

CASTELLÓN DE LA PLANA, DICIEMBRE 2016

TESIS DOCTORAL PRESENTADA POR: JOAQUÍN PÉREZ BADENES
DIRIGIDA POR: JOSÉ IGNACIO ALIAGA ESTELLÉS
ENRIQUE S. QUINTANA ORTÍ

UNIVERSIDAD JAUME I DE CASTELLÓN
E. S. DE TECNOLOGÍA Y CIENCIAS EXPERIMENTALES

CONSUMO ENERGÉTICO DE MÉTODOS ITERATIVOS PARA SISTEMAS DISPERSOS EN PROCESADORES GRÁFICOS

JOAQUÍN PÉREZ BADENES

Índice general

Índice general **vii**

Lista de figuras **xii**

Lista de tablas **xiv**

1 Introducción **1**
1.1 Motivación . . . 1
1.2 Objetivos de la Tesis Doctoral . . . 4
1.3 Estructura de la Memoria de Tesis . . . 5

2 Métodos iterativos para la resolución de sistemas de ecuaciones dispersos **11**
2.1 Notación y definiciones . . . 11
2.1.1 Nociones básicas sobre vectores y matrices . . . 11
2.1.2 Operaciones básicas sobre vectores y matrices . . . 13
2.1.3 Valores y vectores propios de una matriz . . . 13
2.1.4 Normas vectoriales y matriciales . . . 14
2.1.5 Condicionamiento de una matriz . . . 15
2.1.6 Tipos de matrices . . . 15
2.1.7 Matrices dispersas . . . 16
2.1.7.1 Formatos de almacenamiento para matrices dispersas . . . 16
2.1.7.2 Formato disperso de fila comprimida CSR . . . 17
2.1.7.3 Formato disperso ELLPACK/ITPACK ELL . . . 18
2.2 Resolución de sistemas de ecuaciones lineales . . . 19
2.2.1 Clasificación de métodos para la resolución de SEL . . . 19
2.2.2 Métodos de proyección . . . 21
2.2.3 Método CG . . . 22
2.2.4 Método BICG . . . 24
2.2.5 Método BICGSTAB . . . 26
2.3 Técnicas básicas de precondicionado . . . 28
2.4 Las GPUs en computación de altas prestaciones (HPC) . . . 31
2.4.1 Computación Heterogénea . . . 32
2.4.2 CUDA . . . 34

2.4.2.1 Modelo de programación CUDA . . . 34
2.4.2.2 Jerarquía de memoria en CUDA . . . 36
2.4.2.3 Modelo de ejecución CUDA . . . 37
2.4.2.4 Interfaz de programación de CUDA . . . 37
2.4.3 Arquitecturas de GPUs . . . 42
2.4.3.1 Antecedentes (arquitecturas Tesla G80 y GT200) . . . 44
2.4.3.2 Fermi (GF100) . . . 45
2.4.3.2.1 Optimización en Fermi . . . 47
2.4.3.3 Kepler GK110 . . . 50
2.4.3.3.1 Optimización en Kepler . . . 54

3 Performance-energy trade-off in linear solvers for multiprocessors 61
3.1 Introduction . . . 61
3.2 Background and Experimental Setup . . . 63
3.2.1 Krylov-based iterative solvers . . . 63
3.2.2 Matrix benchmarks . . . 64
3.2.3 Hardware Setup and Compilers . . . 64
3.3 Basic CG Method relying on Compiler Optimization . . . 65
3.3.1 Basic implementation of the CG method . . . 65
3.3.2 Search for the optimal configuration . . . 66
3.3.2.1 Optimization with respect to time . . . 67
3.3.2.2 Optimization with respect to net energy . . . 69
3.4 Hardware-Aware Optimization of the CG Method . . . 71
3.4.1 Optimizing the matrix layout for SpMV . . . 72
3.4.1.1 Multicore processors . . . 72
3.4.1.2 Graphics accelerators . . . 73
3.4.2 Reformulating the CG algorithm for GPUs . . . 74
3.4.3 Search for the optimal configuration . . . 74
3.4.3.1 Optimization with respect to time . . . 75
3.4.3.2 Optimization with respect to net energy . . . 76
3.4.4 Complementary analyses . . . 79
3.4.4.1 Algorithmic optimization potential . . . 79
3.4.5 Single precision performance through iterative refinement . . . 79
3.5 Summary and Future Work . . . 80

4 Reformulated CG for linear systems on GPUs 85
4.1 Introduction . . . 85
4.2 Iterative Solution of Sparse Linear Systems . . . 86
4.3 Environment Setup . . . 87
4.3.1 Test matrices . . . 87
4.3.2 Hardware platform and arithmetic . . . 87
4.4 CUBLAS Implementation of the CG method . . . 89
4.4.1 Using CUDA blocking mode for energy efficiency . . . 89
4.4.2 Merging CUDA kernels to regain performance . . . 90
4.4.2.1 Avoiding synchronizations and data transfers . . . 91
4.4.2.2 Merged CUDA code . . . 91
4.4.3 Performance and energy comparison . . . 95
4.5 Conclusions . . . 98

4.6 Fe de erratas . . . 100

5 Systematic fusion of CUDA kernels for linear systems 101
5.1 Introduction . . . 101
5.2 Related Work . . . 102
5.3 Systematic Kernel Fusion for Sparse Iterative Solvers . . . 103
5.3.1 Overview of Iterative Solvers for Sparse Linear Systems . . . 103
5.3.2 Characterization of GPU kernels for sparse iterative solvers . . . 103
5.3.3 Fusion of GPU kernels . . . 104
5.3.4 Fusions in BiCG . . . 105
5.3.5 Fusions in CG and BiCGStab . . . 107
5.4 Experimental Evaluation . . . 108
5.5 Concluding Remarks . . . 110

6 Harnessing CUDA DP in sparse linear systems 113
6.1 Fusions in the CG method . . . 114
6.1.1 Overview . . . 114
6.1.2 Merging kernels in CG . . . 115
6.2 Exploiting DP to Enhance CG . . . 116
6.2.1 Two-stage dynamic DOT . . . 116
6.2.2 Two-stage dynamic AXPY/XPAY . . . 116
6.3 Experimental Evaluation . . . 117
6.4 Concluding Remarks . . . 120

7 Conclusiones y líneas abiertas de investigación 123
7.1 Análisis de arquitecturas paralelas . . . 123
7.2 Fusionado de *kernels* CUDA . . . 125
7.2.1 Primera aproximación . . . 125
7.2.2 Metodología de fusionado de *kernels* . . . 127
7.3 Paralelismo dinámico . . . 130
7.4 Difusión de los resultados del trabajo . . . 130
7.5 Trabajo futuro . . . 131

Lista de figuras

1.1 Patrón de dispersidad (elementos no nulos) de matrices dispersas asociadas a una discretización por elementos finitos (izquierda) y una red de caráter social (derecha). El primer caso corresponde al ejemplo `Boeing/bcsstk35` de la colección UFMC de Tim Davies, recogida en la Universidad de Florida, y modela un componente de un motor de aviación. El segundo caso corresponde al ejemplo `SNAP/cit-HepTh` de la misma colección, y representa el número de citas entre artículos de física de altas energías publicados en el repositorio `Arxiv`. 2

1.2 Evolución de los procesadores durante los últimos 40 años en número de transistores (*Transistors*), rendimiento computacional de una única hebra de ejecución en el test SpecINT (*Single-thread Performance*), frecuencia de reloj (*Frequency*), potencia disipada (*Typical Power*), y número de núcleos (*Number of cores*). Esta gráfica fue elaborada (y actualizada) por K. Rupp a partir de datos de M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, y C. Batten. 3

2.1 Matrices: (a) diagonal, (b) triangular superior, (c) triangular inferior, (d) tridiagonal. El símbolo $\star$ representa cualquier valor. 16

2.2 Matriz *A*, representada mediante el esquema de almacenamiento CSR, utilizando una indexación de acuerdo al lenguaje "*C*" (*0-indexing*). 17

2.3 Implementación secuencial en lenguaje *"C"* de la operación SpMV con el formato de almacenamiento CSR. La matriz tiene *num_filas* filas, y está almacenada haciendo uso de los vectores (AA, JA, IA). El vector multiplicador se denota como *x*, mientras que el vector resultado del producto se denota como *y*. 17

2.4 Matriz *A*, representada mediante el esquema de almacenada ELL, utilizando una indexación de acuerdo al lenguaje "*C*" (*0-indexing*). 18

2.5 Implementación secuencial en lenguaje *"C"* de la operación SpMV con el formato de almacenamiento ELL. La matriz tiene *num_filas* filas, *num_cols_por_fil*a entradas por fila (como máximo) y está formada por los vectores (AA, JA). El vector multiplicador se denota como *x*, mientras que el vector resultado del producto se denota como *y*. 19

2.6 Algoritmo del método del gradiente conjugado, donde $\tau_{\text{máx}}$ es la cota inferior de la norma del residuo. 24

2.7 Algoritmo del método del gradiente biconjugado, donde $\tau_{\text{máx}}$ es la cota inferior de la norma del residuo. 26

2.8 Algoritmo del método del gradiente biconjugado estabilizado, donde $\tau_{\text{máx}}$ es la cota inferior de la norma del residuo. 28

2.9 Algoritmo del método del gradiente conjugado precondicionado, donde $\tau_{máx}$ es la cota inferior de la norma del residuo. 29
2.10 Algoritmo del método del gradiente biconjugado precondicionionado, donde $\tau_{máx}$ es la cota inferior de la norma del residuo. 30
2.11 Algoritmo del método del gradiente biconjugado estabilizado precondicionado, donde $\tau_{máx}$ es la cota inferior de la norma del residuo. 31
2.13 Sistemas basados en memoria compartida **(a)**, en memoria distribuida **(b)** y en cluster de multiprocesadores provistos de GPUs **(c)**. 33
2.14 Arquitectura básica de una GPU, con 8 procesadores escalares (SP) por SM, una unidad de doble precisión (DP), 16 KB de memoria compartida por SM, y 32 KB (8K entradas de 32 bits) de registros por SM. 35
2.15 Pila de capas software de la arquitectura CUDA. 38
2.16 Tres *grids* equivalentes con distintas dimensiones que permiten mapear los *threads* que los definen sobre los elementos de un vector de tamaño igual a 36. 39
2.17 Implementación básica de una operación AXPY diseñada para tres definiciones de *grid* de distintas dimensiones. 40
2.18 Código principal ejecutado en el *host*, en el que se realizan llamadas a tres versiones distintas de *kernel* de la operación AXPY, que se corresponden con tres *grids* de dimensiones 2D_2D, 2D_1D y 1D_1D. 41
2.19 Acceso y operaciones aritméticas que el *kernel* AXPY realiza sobre sus operandos situados en la memoria global de la GPU, al definir un *grid* 1D-1D. 42
2.21 Caption for LOF 44
2.22 La arquitectura Fermi dispone de 16 SMs distribuidos en torno a una caché L2 (banda azul claro). Cada SM dispone de un ampliado fichero de registros (banda azul oscuro), además contiene dos planificadores de envío de instrucciones (banda naranja), con una unidad de emisión de instrucciones cada una de ellas (banda roja), varias unidades de ejecución (rectángulos en verde) y una unidad configurable de "memoria compartida / caché L1" (banda azul claro) [1] 46
2.23 Diagrama completo de bloques del chip de Kepler GK110 [2] 50
2.24 Caption for LOF 53
2.25 Resumen de las mejoras introducidas en Kepler GK110 comparadas con Fermi. 55

3.1 Mathematical formulation of the CG method. The names inside parenthesis, except for SpMV which refers to the sparse matrix-vector kernel, identify the routine from the level-1 BLAS that offers the corresponding functionality. 63
3.2 Simplified loop-body of the basic CG implementation using double precision (DP). 64
3.3 Basic dense and sparse matrix storage formats. The memory demand corresponds to the grey areas. 67
3.4 Sparse matrix-vector product using the CSR and ELLPACK formats (`SpMV_csr` and `SpMV_ell`, respectively). 68
3.5 Comparison of performance (left) and energy efficiency (right), measured, respectively, in terms of GFLOPS and GFLOPS/W, when optimizing the basic CG implementations with respect to run time (top) or net energy (bottom). 71
3.6 Visualizing the BCSR, CSB, and SELL-P formats. Note that choosing the block size 2 for BCSR and SELL-P, as well as the block size 4 for CSB, requires adding a zero row to the original matrix. Furthermore, padding the SELL-P format to a row length divisible by 2 requires explicit storage of a few additional zeros. 72

3.7 Aggregation of kernels to transform the basic implementation into a merged implementation. Here, the invocation to the sparse matrix-vector kernel SPMV has to be replaced by the appropriate GPU routine, depending on the sparse matrix format. 74
3.8 Comparison of performance (left) and energy efficiency (right), measured, respectively, in terms of GFLOPS and GFLOPS/W, when optimizing the tuned CG implementations with respect to run time (top) or net energy (bottom). 77

4.1 Algorithmic description of the CG method. 87
4.2 Sparsity plots of the test matrices. 88
4.3 Simplified fragment of the CG-CUBLAS implementation based on the scalar kernel SSPMV. The implementation based on the vector kernel only differs in that routine VSPMV (with the same parameters) is invoked instead of SSPMV. 89
4.4 Variation of time and CPU energy of the CG-CUBLAS implementation when the blocking mode is set instead of the default polling mode. 91
4.5 Power (in Watts) dissipated by the CG-CUBLAS implementation, using the polling and blocking modes, for the AUDIKW_1 matrix. 92
4.6 Aggregation of kernels to transform the CG-CUBLAS implementation (using SSPMV) into the CG-merged implementation. The aggregation for the CG-CUBLAS implementation based on VSPMV is analogous. 92
4.7 `fusion_3` employed in the CG-merged code. Note that this kernel employs a block-size of 256. 93
4.8 Implementation of the kernel `fusion_4_1` and routine `fusion_4` employed in the CG-merged code. Note that `fusion_4_1` is based on a block-size of 128 threads. 94
4.9 Symmetric reduction scheme with a thread block size of 128. For larger systems, the reduction has to be applied iteratively. 95
4.10 Comparison of time (top), CPU energy (middle) and total energy (bottom) of the merged implementations, in polling (left) and blocking (right) modes, using the polling-CUBLAS code as reference. 97
4.11 Execution time, CPU energy and total energy, averaged for all matrices in the benchmark collection. 98
4.12 En la izquierda se muestra el código de la Figura 4.7, y en la derecha su código corregido, en el que se ha eliminado la línea 13. 100

5.1 Algorithmic formulation of the preconditioned BiCG method. 103
5.2 Dependencies between GPU kernels and fusions (left and right, respectively) for the preconditioned BiCG solver with SPMV based on the scalar CSR or ELL format. 105
5.3 Fusions of GPU kernels in the preconditioned CG and BiCGStab solvers (left and right, respectively) with SPMV based on the scalar CSR or ELL format. The colors of the nodes match those employed for the preconditioned BiCG solver, and identify the same four types of operations: SPMV, DOT, AXPY/XPAY and JPRED. 107
5.4 Execution time and energy consumption for CG, BiCG and BiCGStab solvers (top, middle and bottom, resp.) without and with preconditioner (left and right, resp.). 109

6.1 Algorithmic formulation of the CG method. In general, we use Greek letters for scalars, lowercase for vectors and uppercase for matrices. Here, $\tau_{\max}$ is an upper bound on the relative residual for the computed approximation to the solution. 115
6.2 Fusions for the CG solver with SPMV based on the scalar CSR or ELL format. 115
6.3 Implementation of kernel $\text{DOT}^{\text{D}}_{\text{ini}}$. 117

6.4 Implementation of kernel $\text{DOT}^{\text{D}}_{\text{fin}}$. 118
6.5 Implementation of $\text{DOT}^{\text{D}}_{\text{ini}}$. 119
6.6 Implementation of kernel AXPY^{D}. 120
6.7 Variations (in %) of execution time and energy consumption of the CG solver (left and right, respectively) with respect to the baseline. 121

Lista de tablas

2.1 Evolución de las características más relevantes de las GPUs desde G80 hasta Kepler. 43
2.2 Kepler GK110. 51

3.1 Description and properties of the test matrices (top) and the corresponding sparsity plots (bottom). 65
3.2 Multithreaded architectures. For the GPU systems (FER, KEP, and QDR), the idle power includes the consumption of both the server and the accelerator. 66
3.3 Storage overhead of the test matrices when using ELLPACK or SELL-P format. n_z^{FORMAT} refers to the explicitly stored elements (n_z nonzero elements and the explicitly stored zeros for padding). 67
3.4 Optimal hardware parameter configuration when optimizing the general-purpose architectures for runtime or energy efficiency using the basic implementations. In the labels, c denotes the number of cores, f the frequency (in MHz), T the time per iteration (in seconds), and E_{net} the net energy per iteration (in Joules). 69
3.5 Optimal hardware parameter configuration when optimizing the specialized architectures for runtime or energy efficiency using the basic implementations. In the labels, c denotes the number of cores, f the frequency (in MHz), T the time per iteration (in seconds), and E_{net} the net energy per iteration (in Joules). 70
3.6 Optimal hardware parameter configuration when optimizing the general-purpose architectures for runtime or energy efficiency using the tuned implementations. In the labels, c denotes the number of cores, f the frequency (in MHz), s the sparse matrix layout/SpMV implementation (0 for CSR+MKL, 1(2) for BCSR+MKL with block size `1b`=2, 1(3) for BCSR+MKL with block size `1b`=3, and 2 for CSB), T the time per iteration (in seconds), and E_{net} the net energy per iteration (in Joules). 75
3.7 Optimal hardware parameter configuration when optimizing the specialized architectures for runtime or energy efficiency using the tuned implementations. In the labels, c denotes the number of cores, f the frequency (in MHz), s the sparse matrix layout/SpMV implementation (0 for CSR, 1 for ELLPACK, 2 for ELLR-T, and 3 for SELL-P), T the time per iteration (in seconds), and E_{net} the net energy per iteration (in Joules). 76
3.8 Average GFLOPS and GFLOPS/W rates of the DP tuned implementations normalized with respect to the "fastest" (highest GFLOPS rate when optimizing w.r.t. time) and "most energy-efficient" (highest GFLOPS/W rate when optimizing w.r.t. energy) architectures, KEP and A15 respectively. 78

3.9 Speed-ups resulting from the algorithmic optimizations. 78
3.10 Speed-ups resulting from the use of the SP tuned implementations instead of the DP versions. 80

4.1 Description and properties of the test matrices. 88
4.2 Execution time of the two SpMV kernels and the CG-CUBLAS implementations built on top of them. 90
4.3 Execution time, CPU energy (Cenergy) and total energy (Tenergy) consumption of the different CG implementations in polling (top) and (blocking) mode. 96

5.1 Types of access to the vector inputs/output of the GPU kernels. 104
5.2 Description and properties of the test matrices from the UFMC (left) and the 3D Laplace problem (right). In the matrix names, FEM_3DTH2 corresponds to the "FEM 3D nonlinear thermal problem". 108

6.1 Description and properties of the test matrices from the UFMC and the 3D Laplace problem. 118
6.2 Minimum, maximum and average variations (in %) of execution time and energy consumption for CG with respect to the baseline. 120

7.1 En la izquierda de la tabla se muestran las fusiones efectuadas en el CG al utilizar el algoritmo SSpMV, mientras que en la derecha se muestran las efectuadas al utilizar el algoritmo VSpMV. Ambos esquemas se corresponden con los del trabajo presentado en [ICPP:2013] [183] (Capítulo 4). 127
7.2 Para cada método iterativo, la tabla muestra el número de *kernels* que lo compone, y el número de *macrokernels* que la aplicación del *"método de fusionado de kernels CUDA"* ha obtenido. Estos resultados corresponden al trabajo presentado en [EPAR:2015] [184] (Capítulo 5). 129

Lo último que uno sabe es por donde empezar.

Blaise Pascal

Hazlo simple, tan simple como sea posible, pero no más

Albert Einstein

Resumen

Hoy en día, en las aplicaciones científicas y de ingeniería surgen con mucha frecuencia problemas de resolución de sistemas de ecuaciones lineales dispersos de gran dimensión. El incesante crecimiento de los tamaños de estos problemas, entre otros factores, propicia un creciente incremento del consumo energético en los grandes centros de cálculo. Las restricciones energéticas imperantes, y la concienciación con el respeto al medio ambiente, propició la aparición de ***Green Computing***, con el objetivo de diseñar aplicaciones conscientes de la energía y considerar la eficiencia energética (GFLOPS/W) como un parámetro prioritario. Tradicionalmente, se ha considerado que el rendimiento es directamente proporcional al consumo energético, pero la aplicación de estas nuevas técnicas permite incrementar el rendimiento que requieren las aplicaciones científicas con un consumo energético ajustado.

Los objetivos de esta Tesis Doctoral se enfocan hacia el estudio y desarrollo de técnicas de ahorro de energía en sistemas de cómputo heterogéneo, CPU-GPU, para diseñar aplicaciones que consigan minimizar el consumo energético sin sacrificar el rendimiento, en la resolución de sistemas lineales dispersos mediante métodos iterativos. Para ello se explota que las GPUs muestran un ahorro de energía muy significativo cuando se sincronizan con la CPU en modo *blocking*, aunque su rendimiento disminuye de forma considerable. Por el contrario, las GPUs alcanzan máximas prestaciones cuando están sincronizadas en modo *polling*, pero ello supone un sumidero de energía por parte de la CPU al permanecer en un estado activo. El diseño de *"técnicas de fusionado de kernels CUDA"*, propicia una reducción del número de *kernels*, eliminándose tiempos de lanzamiento de *kernels*, y tiempos de transferencia de información entre los espacios de memoria de la CPU y la GPU. Además, si la GPU está sincronizada en modo *blocking*, la reducción del número de *kernels* también decrementa el número de cambios de un estado de bajo consumo energético de la CPU hasta otro activo, eliminando los correspondientes tiempos de espera de recuperación de la CPU, que permite que el coste de ejecución del modo *blocking* sea muy parecido al del modo *polling*.

Debido a su complejidad, las *"técnicas de fusionado de kernels CUDA"*, sólo son utilizadas por programadores expertos, por lo que el diseño de una metodología ayudará de forma sistemática al proceso de *"fusionado de kernels CUDA"*. Esta tarea es la principal aportación de esta Tesis Doctoral, extendiendo su aplicación sobre arquitecturas de GPU que soporten paralelismo dinámico, que permite optimizar resolutores para que puedan ejecutarse de forma desacoplada de la CPU. Los resultados experimentales obtenidos validan mejoras destacables tanto en rendimiento como en eficiencia energética, obteniendo un compromiso entre rendimiento y consumo energético constante y equilibrado.

Agradecimientos

En una Tesis Doctoral sobre computación de altas prestaciones, todo gira alrededor de tecnicismos. Sin embargo, el apartado de agradecimientos es el único que fundamentalmente es emotivo, y más en mi caso, ya que durante el último periodo de la escritura de esta Tesis Doctoral tuve un bache en mi salud. Sin la comprensión, apoyo, ayuda y esfuerzo adicional de mis directores, este trabajo no podría haberse concluido. Gracias por sus conocimientos, su rigurosidad y por sus correcciones. Eternamente agradecido a los Drs. José Ignacio Aliaga Estellés y Enrique S. Quintana Ortí.

Deseo expresar mi más sincero agradecimiento:

A la Dra. Maribel Castillo Catalán, quien supuso para mí el primer contacto con el apasionante mundo de las GPUs.

Al Dr. Francisco Daniel Igual Peña por su ayuda cuando caían los equipos y los reiniciaba lo antes posible, ya durante mi periodo de estudios de Master.

Al Dr. Manuel Dolz Zaragozá por su valiosa ayuda con las mediciones de potencia y energía en los equipos utilizados.

A todos los profesores que he tenido en los distintos cursos de Master y Doctorado, por su gran nivel docente e inducirme a tener un espíritu investigador, y en general a todos los compañeros y componentes del departamento.

A mi familia (cuántas veces me he perdido una comida familiar), a mis padres, y en especial a mi madre, Isabel, quien me ha animado para terminar esta aventura de ciencia, una mujer sabia.

CAPÍTULO 1

Introducción

1.1 Motivación

Las operaciones de álgebra lineal en general, y la solución de sistemas de ecuaciones lineales en particular, aparecen de manera ubicua en un amplio abanico de aplicaciones científicas e ingenieriles clásicas así como, más recientemente, en el procesado de grandes volúmenes de datos (*big data*) [10, 16, 29]. Como ejemplo, este tipo de problemas subyace en los motores de búsqueda en la web (*PageRank* de Google), los algoritmos de análisis de redes sociales (agrupamiento y particionado de grafos, detección de comunidades y anomalías, etc.), la generación de modelos económicos, los algoritmos de recuperación de información, y los métodos de elementos finitos para ecuaciones diferenciales parciales. Entre otros muchos problemas de ingeniería, la discretización por elementos finitos sustenta el análisis de la resistencia de estructuras de hormigón, la estimación de la órbita de los electrones, la evaluación del campo gravitatorio terrestre, la simulación de circuitos integrados, el estudio de las propiedades de nanocristales semiconductores, la detección de oclusiones en vasos sanguíneos mediante resonancia magnética, y la simulación del comportamiento de componentes estructurales de aviación. Esta variada relación de casos de aplicación no es más que una pequeña muestra de escenarios en los que resulta necesario resolver un sistema de ecuaciones lineales. En todos estos casos, la resolución de esta operación básica del álgebra lineal a menudo supone la parte computacionalmente más costosa del problema.

En un número elevado de aplicaciones, la matriz coeficiente del sistema de ecuaciones lineales que debe tratarse presenta una gran cantidad de elementos nulos. Bajo estas circunstancias, el aprovechamiento de esta estructura especial puede reducir considerablemente los costes computacionales y de almacenamiento para resolver el problema. Cuando los elementos no nulos de la matriz presentan un patrón irregular de distribución, nos encontramos con un sistema de ecuaciones lineales *disperso*. (Un ejemplo contrapuesto al caso disperso lo constituye, por ejemplo, un problema *banda*, donde los elementos no nulos de la matriz coeficiente se concentran en un pequeño número de diagonales de la matriz alrededor de su diagonal principal.) La Figura 1.1 ilustra el patrón de distribución de elementos no nulos de dos matrices dispersas provenientes de aplicaciones muy diferentes.

El interés de las aplicaciones mencionadas anteriormente ha dado pie al diseño, implementación y evaluación de numerosos formatos de almacenamiento, algoritmos y bibliotecas para resolver problemas de álgebra lineal dispersa en procesadores de propósito general, siguiendo la evolución de las arquitecturas de computador desde los tiempos de Von Neumann.

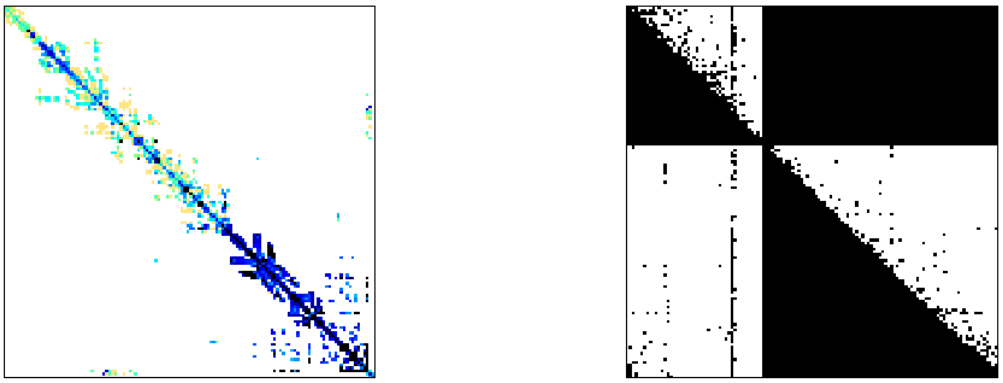


Figura 1.1: Patrón de dispersidad (elementos no nulos) de matrices dispersas asociadas a una discretización por elementos finitos (izquierda) y una red de caráter social (derecha). El primer caso corresponde al ejemplo `Boeing/bcsstk35` de la colección UFMC de Tim Davies, recogida en la Universidad de Florida, y modela un componente de un motor de aviación. El segundo caso corresponde al ejemplo `SNAP/cit-HepTh` de la misma colección, y representa el número de citas entre artículos de física de altas energías publicados en el repositorio `Arxiv`.

A mediados de la década pasada, las limitaciones del paralelismo a nivel de instrucción y la creciente disparidad entre los rendimientos del procesador y de la memoria, entre otros factores, promovieron la introducción de las arquitecturas multinúcleo (*multicore*), compuestas por varios *cores*, o procesadores completos, como vía para seguir mejorando las prestaciones de los computadores [22]. Este cambio, a su vez, motivó el rediseño y adaptación de los algoritmos de álgebra lineal dispersa. En la actualidad, la investigación en esta línea está lo suficientemente asentada como para haber quedado recogida en forma de un conjunto de códigos optimizados para la solución de este tipo de problemas en procesadores multinúcleo [9, 11, 12, 30], que se encuentran integrados en productos comerciales como las bibliotecas MKL de Intel y ESSL de IBM.

En los últimos años, ha surgido un creciente interés por el uso de coprocesadores para operaciones que requieran cálculos intensivos. Entre este tipo de *aceleradores*, los procesadores gráficos, o GPUs (*graphics processing units*), son un tipo de arquitectura masivamente paralela que ha logrado un reconocimiento importante como acelerador de algunas aplicaciones computacionalmente costosas, que exhiben un patrón de concurrencia conocido como *paralelismo de datos* [23]. Entre las características de las GPUs que las han aupado a esta posición de relevancia resaltan la existencia de estándares *de facto* para su programación, como CUDA [28], OpenCL [26] y OpenACC [1], así como la integración, en las últimas generaciones de este tipo de coprocesador, de miles de unidades de cálculo en coma flotante.

A día de hoy, el principal fabricante de procesadores gráficos es NVIDIA, compañía que impulsó la arquitectura CUDA y el modelo de programación asociado basado en paralelismo de datos, clave para la expansión de las GPUs en computación científica. NVIDIA dispone además de un rico ecosistema de soluciones software, propias y desarrolladas por otros, en forma de bibliotecas para álgebra lineal densa (CUBLAS, MAGMA, CULA), álgebra lineal dispersa (CUSPARSE), procesamiento de señal (CUFFT), y un largo etcétera [27].

Desafortunadamente, los avances en el número de transistores que es posible integrar en un circuito, al dictado de la Ley de Moore [25], no están siendo acompañados por una reducción proporcional en la potencia disipada por la tecnología CMOS, en lo que se conoce como el fin de la Ley de Dennard [13, 18].

Este hecho se encuentra entre los factores que contribuyeron a la introducción de los procesadores multinúcleo, en un primer momento, y más recientemente, a la adopción más general de aceleradores, como las GPUs, debido a su favorable relación entre consumo (disipación de potencia) y rendimiento computacional (productividad o desempeño, en términos de operaciones por segundo). En el momento de redactar esta memoria (noviembre de 2015), las clasificaciones Top500 y Green500 [2, 21] recogen en sus primeras posiciones supercomputadores construidos a partir de alguna tecnología estándar de propósito general (procesador multinúcleo) aderezados con algún tipo de acelerador, en muchos casos una GPU.

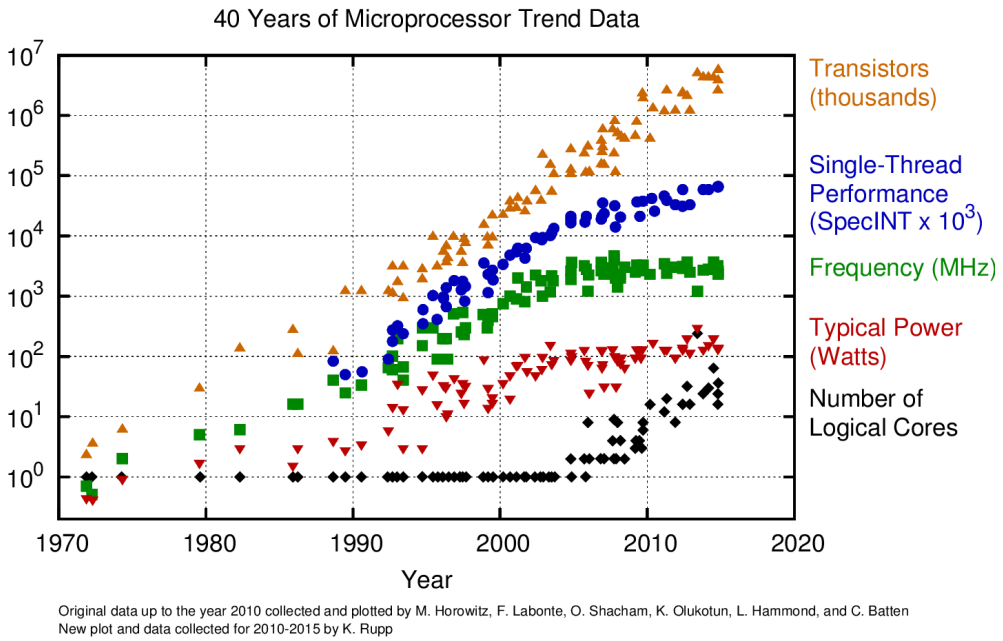


Figura 1.2: Evolución de los procesadores durante los últimos 40 años en número de transistores (*Transistors*), rendimiento computacional de una única hebra de ejecución en el test SpecINT (*Single-thread Performance*), frecuencia de reloj (*Frequency*), potencia disipada (*Typical Power*), y número de núcleos (*Number of cores*). Esta gráfica fue elaborada (y actualizada) por K. Rupp a partir de datos de M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, y C. Batten.

El consumo energético se ha convertido en estos últimos años en uno de los factores que más constriñen las prestaciones de los computadores actuales, debido a la imposibilidad de disipar el calor en circuitos CMOS que funcionen a una frecuencia elevada [17, 18, 19, 24]. La Figura 1.2 muestra claramente esta situación. Aunque este gráfico expone un incremento continuado en el número de núcleos de procesamiento, también muestra que, en general, estos diseños sacrifican a cambio la frecuencia de operación, de forma que la potencia disipada por el procesador raramente supera los 140 Vatios. Además, en los supercomputadores este consumo energético se traduce en un coste económico, que ronda alrededor del millón de euros por MVatio (millón de vatios) y año. A menor escala el consumo energético tiene también importantes repercusiones negativas, que por ejemplo se traducen en la imposibilidad de alimentar todos los componentes de un circuito, en el caso de diseños empotrados de muy bajo consumo alimentados por fuentes de baja potencia, o en una menor autonomía, en el caso de dispositivos móviles (teléfonos, tablets) y portables (ordenadores portátiles).

A pesar de la importancia del consumo energético, hasta la fecha solo se han desarrollado un reducido número de investigaciones que equiparan esta métrica a la productividad. Sin embargo, existe un buen nú-

mero de formas de combinar estos dos factores, consumo y productividad. Por ejemplo, para una aplicación que requiere un tiempo de respuesta determinado, el objetivo puede consistir en minimizar el consumo bajo la premisa de no exceder el tiempo fijado. En cambio, cuando no existe esta limitación, el objetivo puede fijarse en minimizar el consumo, independientemente del tiempo de ejecución. Sin embargo, hay que tener en cuenta que el consumo de energía es proporcional al tiempo de ejecución, con lo que un aumento de este último factor también afecta negativamente al consumo. Otras posibilidades son minimizar una métrica que combine energía y rendimiento computacional, como el EDP (*energy-delay product*) o alguna de sus variantes [8].

En el caso particular de bibliotecas para álgebra lineal densa, unos pocos trabajos recientes han perseguido reducir el consumo energético, manteniendo el rendimiento computacional, a través de la modulación de la frecuencia de reloj del procesador (*dynamic voltage-frequency scaling* o DVFS [20]), la variación del grado de concurrencia, o la eliminación de esperas activas en el propio código de la aplicación o en el entorno (*runtime*) encargado de su ejecución [5, 14]. Algunas de estas técnicas han demostrado ser también válidas para la ejecución de algoritmos de resolución (resolutores) de sistemas de ecuaciones lineales dispersos sobre procesadores multinúcleo y coprocesadores como el Intel Xeon Phi [4].

La adaptación de estas técnicas de reducción de consumo en la resolución de sistemas de ecuaciones lineales dispersos, o la formulación de otras nuevas particularmente adaptadas a la arquitectura de los procesadores gráficos era, hasta el inicio de este trabajo, un campo por explorar. La identificación de esta carencia nos llevó a plantear los objetivos de investigación, que se exponen en la siguiente sección de la memoria, dirigidos a paliar esta deficiencia.

1.2 Objetivos de la Tesis Doctoral

El objetivo general de esta tesis es el *diseño, desarrollo y evaluación de nuevos resolutores de sistemas de ecuaciones lineales para GPUs, que mejoren la eficiencia energética de este tipo de cálculos sin sacrificar su rendimiento computacional.*

Para abordar este objetivo general, nos centramos en el uso de métodos iterativos, equipados con algún tipo de precondicionador sencillo [29]. Comparados con los métodos directos [16], los resolutores iterativos presentan una implementación mucho más sencilla, tanto secuencial como paralela, y no sufren los problemas de llenado que aparecen cuando un método directo se aplica a la solución de una discretización por EF 3D. Otra característica especialmente atractiva de los métodos iterativos escogidos para este estudio es un patrón de acceso a memoria irregular, y una intensidad computacional baja (relación entre operaciones aritméticas y accesos a memoria), lo que los hace representativos de las aplicaciones científicas que se ejecutan en los grandes centros de supercomputación actuales [15].

Aunque existen precondicionadores muy sofisticados, que permiten acelerar notablemente la convergencia de los métodos iterativos, en esta tesis no se aborda su utilización. La razón principal es que, al igual que sucede con los formatos de almacenamiento, la bondad de un precondicionador específico está fuertemente acoplada con una aplicación en particular, por lo que la especialización de nuestro estudio podría reducir su aplicabilidad. En definitiva, los mecanismos, técnicas y estrategias de ahorro de energía propuestos como parte de esta tesis doctoral se evalúan haciendo uso, en todo caso, de un precondicionador simple, si bien es esperable que los resultados sean extrapolables a otros resolutores equipados con precondicionadores más sofisticados.

El objetivo general se descompone en cuatro objetivos más específicos, que se concretan a continuación.

Caracterización del rendimiento. Como punto de partida a la realización de la tesis doctoral, se propone la realización de un estudio experimental, que evalúe de manera exhaustiva implementaciones básicas y optimizadas de un método iterativo representativo, sobre un amplio rango de arquitecturas multinúcleo, así como aceleradores de tipo GPU.

Algunos estudios anteriores sobre sistemas híbridos equipados con un procesador de propósito general (o CPU) y una GPU, identificaron una fuente imporante de ineficiencia energética en las esperas activas de la CPU mientras los cálculos proceden en la GPU [14]. En el caso de los métodos iterativos convencionales para sistemas de ecuaciones lineales dispersos, existe el agravante de que éstos se descomponen en un conjunto de *núcleos computacionales* (*kernels* CUDA) con una intensidad computacional especialmente baja. Para abordar este problema concreto, planteamos los siguientes objetivos específicos para la tesis.

Reducción del número de núcleos computacionales de la GPU. Se estudiará la viabilidad e impacto de la *fusión* de núcleos computacionales, a fin de reducir su número, y en consecuencia minimizar el sobrecoste asociado a su invocación y a la transferencia de información entre los espacios de direcciones de la CPU y la GPU. Aunque la fusión es una técnica bien conocida en el ámbito de los compiladores [3], su aplicación para la mejora energética de resolutores de álgebra lineal dispersa sobre aceleradores gráficos, al ser un problema complejo y muy específico, no ha sido abordada en profundidad previamente.

Cabe destacar que este objetivo no pretende ser un mero trabajo de desarrollo y validación, destinado a producir implementaciones energéticamente eficientes de los resolutores conocidos. Por contra, el propósito es identificar las condiciones que posibilitan la fusión de dos núcleos computacionales, resultando en una caracterización sistemática que permita, eventualmente, la automatización de este proceso mediante un compilador.

Ejecución desacoplada del resolutor iterativo. Como complemento a la fusión de núcleos computacionales, se plantea la posibilidad de producir una ejecución total o parcialmente desacoplada del resolutor en la GPU, explotando el *paralelismo dinámico*, introducido recientemente en el modelo CUDA de NVIDIA (CUDA-DP) [28]. En concreto, esta técnica permite que un núcleo computacional, ejecutándose en la GPU, pueda invocar directamente a otro(s), sin intervención directa de la CPU. Este objetivo es ortogonal a la reducción del número de núcleos computacionales, de modo que es posible combinar ambas estrategias para producir un resolutor con un menor número de núcleos computacionales que, además, sean invocados directamente desde la propia GPU.

En último lugar, a fin de mejorar las prestaciones de los resolutores iterativos, se plantea el estudio y optimización de algunos de los núcleos computacionales que aparecen en estas rutinas en el siguiente objetivo.

Optimización de núcleos computacionales para álgebra lineal dispersa. Una operación especialmente exigente en el caso de la GPU es la *reducción* asociada al producto escalar de dos vectores, por lo que su implementación debe realizarse de manera muy cuidadosa. Esta circunstancia presenta el agravante de que tanto la fusión de núcleos computacionales, como el aprovechamiento de CUDA-DP, pueden influir sobre las prestaciones computacionales y energéticas de esta operación en particular.

1.3 Estructura de la Memoria de Tesis

El resto de la memoria de tesis está estructurada en 6 capítulos, con los contenidos que se reseñan a continuación. El Capítulo 2 realiza una revisión de conceptos y métodos fundamentales del álgebra lineal, centrándose en la resolución de problemas dispersos, y en los métodos por computador.

Los capítulos centrales de la tesis, desde el Capítulo 3 hasta el 6, corresponden a artículos científicos que han sido publicados o aceptados para su publicación en revistas o actas de congresos internacionales con revisión por pares. Cada uno de los capítulos es autocontenido, presentando una motivación y una revisión (en algunos casos implícita) del estado del arte, una exposición de sus contribuciones originales,

una descripción y validación experimental de la propuesta, y un conjunto de conclusiones. A continuación se describe someramente el contenido de estos capítulos.

Capítulo 3

> **[CCPE:2015]** *J. I. Aliaga, H. Anzt, M. Castillo, J. C. Fernández, G. León,* **J. Pérez***, E. S. Quintana-Ortí.*
> *"Unveiling the performance-energy trade-off in iterative linear system solvers for multithreaded processors".*
> *Concurrency and Computation: Practice & Experience, Vol. 27(4), pp. 885-904, 2015.*
> *ISSN: 1532-0626.*
> *Factor de impacto: 0.997 (JCR 2014). Posición 46 de 102 revistas en la categoría* Computer Science, Theory & Methods.

Este trabajo presenta un completo estudio experimental del "rendimiento" del método del gradiente conjugado (CG) [29], una pieza clave en la resolución de sistemas lineales dispersos cuando la matriz coeficiente es simétrica y definida positiva. El estudio comprende plataformas de experimentación basadas en una buena muestra de las arquitecturas multinúcleo y aceleradores gráficos utilizados en la actualidad para computación de altas prestaciones o en sistemas de bajo consumo. A fin de ofrecer unos resultados más representativos, la evaluación experimental incluye tanto implementaciones básicas del método CG como variantes que integran optimizaciones manuales del código, en función de la arquitectura de destino, que van más allá de las posibilidades de un compilador.

Desde el punto de vista práctico, el estudio considera tres métricas para evaluar el comportamiento del algoritmo: rendimiento computacional, disipación de potencia y consumo de energía, que ofrecen otros tantos prismas con los que valorar la eficacia de una implementación/arquitectura. Por ejemplo, el rendimiento computacional está directamente relacionado con el tiempo de ejecución; la potencia disipada puede limitar las condiciones de ejecución del algoritmo en un sistema donde el suministro de potencia está restringido, pudiendo obligar a reducir la frecuencia de ejecución; y del consumo de energía se desprende no solo un coste económico, sino también limitaciones en cuanto a autonomía en dispositivos móviles.

Los resultados de este estudio, aplicados a un subconjunto de la colección UFMC, exponen la posición predominante de los aceleradores gráficos como arquitectura para la solución de problemas del álgebra lineal dispersa desde el punto de vista del rendimiento computacional, así como sus ventajas en términos de eficiencia energética.

Capítulo 4

> **[ICPP:2013]** *J. I. Aliaga, H. Anzt,* **J. Pérez***, E. S. Quintana-Ortí.*
> *"Reformulated Conjugate Gradient for the energy-aware solution of linear systems on GPUs".*
> *42nd International Conference on Parallel Processing – ICPP 2013, pp. 320-329. Lyon (Francia), 2013.*
> *ISBN: 0190-3918/13.*
> *Ratio de aceptación de trabajos: 30.6 %.*

Las investigaciones del grupo *High Performance Computing & Architectures* de la *Universitat Jaume I*, en colaboración con miembros del *Engineering & Mathematics Computing Lab* del *Karlsruhe Institute of Technology*, identificaron a la CPU como un sumidero importante de energía durante la ejecución de métodos asíncronos de solución de sistemas lineales dispersos en servidores híbridos CPU-GPU [6, 7].

En el trabajo [ICPP:2013] se demuestra cómo, mediante el rediseño adecuado del método CG, es posible obtener una nueva formulación del algoritmo que aprovecha de manera más eficiente los mecanismos de ahorro de energía inherentes en la CPU, sin disminuir por ello el rendimiento computacional. La clave de este resultado radica en una cuidadosa fusión de núcleos computacionales (*kernels*) CUDA, que desencadena como efecto periodos de inactividad más prolongados en la CPU, mientras los

cálculos se desarrollan en la GPU, y permiten explotar de manera más eficiente los C-estados del procesador general, vía una espera pasiva.

El rediseño del método CG propuesto en el trabajo [ICPP:2013] requiere identificar los *kernels* que deben fundirse, además de un esfuerzo adicional de implementación, al apartarse de la via tradicional de implementación de métodos iterativos fundamentada en el uso de rutinas de bibliotecas estándar (lo que imposibilita las fusiones). A cambio, los resultados con un subconjunto de casos de la colección UFMC, muestran ligeros beneficios desde la perspectiva de rendimiento computacional, que resultan mucho más notables en términos de ahorro energético.

Capítulo 5

> **[EPAR:2015]** *J. I. Aliaga,* **J. Pérez**, *E. S. Quintana-Ortí.*
> *"Systematic fusion of CUDA kernels for iterative sparse linear system solvers".*
> *Lecture Notes in Computer Science 9233, Euro-Par 2015, pp. 675-686, Viena (Austria), 2015.*
> *ISBN: 978-3-662-48095-3.*
> *Ratio de aceptación de trabajos: 26.8 %.*

Este trabajo contiene el resultado más relevante de la tesis doctoral, al convertir la tarea de fusión de *kernels*, desde un proceso reservado a un experto en un procedimiento sistemático, mediante la definición de criterios para la caracterización de *kernels* que permiten determinar la viabilidad de las fusiones. En algo más de detalle, los criterios de fusión identificados en [EPAR:2015] se basan en un estudio de las dependencias entre *kernels* y el tipo de acceso que cada *kernel* realiza los datos de entrada y de salida.

La generalidad de los principios identificados se valida en el trabajo mediante su aplicación en un conjunto representativo de métodos iterativos para la resolución de sistemas lineales dispersos. En este punto, cabe recordar que el propósito de la fusión de *kernels* es reducir de manera significativa su número, en vista de los beneficios ya demostrados en [ICPP:2015]. Si bien no se ha demostrado, esperamos que estos principios permitan caracterizar cualquier *kernel* CUDA y, por tanto, sean aplicables a la optimización de otros algoritmos compuestos por *kernels* CUDA.

Capítulo 6

> **[PARC:2015]** *J. I. Aliaga, D. Davidović,* **J. Pérez**, *E. S. Quintana-Ortí.*
> *"Harnessing CUDA dynamic parallelism in the solution of sparse linear systems".*
> *Parallel Computing – ParCo2015, (aceptado y pdte. de publicación). Edinburgo (Reino Unido), 2015.*

La fusión de *kernels* es un propósito deseable, al reducir el número de sincronizaciones entre CPU y GPU, pero no siempre posible. Cuando la sincronización no es debida a la necesidad de comunicar un dato entre CPU y GPU, una alternativa a la fusión de *kernels* es el aprovechamiento de la tecnología (CUDA-DP). El trabajo [PARC:2015] demuestra beneficios similares entre las dos técnicas, con un menor esfuerzo de programación con CUDA-DP, centrando principalmente en el diseño de *kernels* especializados para el producto escalar de vectores.

La tesis se cierra con el Capítulo 7 de conclusiones, que recuerda brevemente la motivación de esta tesis doctoral, enumera sus principales contribuciones y resultados cualitativos, e incluye una compilación de líneas de investigación abiertas, a abordar en el marco de la aceleración de métodos numéricos en procesadores gráficos.

Bibliografía

[1] OpenACC directives for accelerators. http://www.openacc.org/, 2015.

[2] The top500 list, Nov. 2015. http://www.top500.org.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Gradiance, 2006.

[4] J. I. Aliaga, R. M. Badia, M. Barreda, M. Bollhöfer, and E. S. Quintana-Ortí. Leveraging task-parallelism with OmpSs in ILUPACK's preconditioned CG method. In *26th Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 262–269, 2014.

[5] J. I. Aliaga, M. Barreda, M. A. Casta no, M. F. Dolz, and E. S. Quintana-Ortí. Adapting concurrency throttling and voltage-frequency scaling for dense eigensolvers. *The Journal of Supercomputing*, 2015. DOI:10.1007/s11227-015-1600-z.

[6] H. Anzt, V. Heuveline, J. I. Aliaga, M. Castillo, J.C. Fernández, R. Mayo, and E. S. Quintana-Ortí. Analysis and optimization of power consumption in the iterative solution of sparse linear systems on multi-core and many-core platforms. In *Int. Green Computing Conference and Workshops (IGCC)*, pages 1–6, 2011.

[7] H. Anzt, B. Rocker, and V. Heuveline. Energy efficiency of mixed precision iterative refinement methods using hybrid hardware platforms. *Computer Science - Research and Development*, 25(3):141–148, 2010.

[8] C. Bekas and A. Curioni. A new energy aware performance metric. *Computer Science - Research and Development*, 25(3-4):187–195, 2010.

[9] A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *IEEE Int. Parallel & Distributed Processing Symposium*, (IPDPS), pages 721–733, 2011.

[10] D. Buono, J.A. Gunnels, X. Que, F. Checconi, F. Petrini, Tai-Ching Tuan, and C. Long. Optimizing sparse linear algebra for large-scale graph analytics. *Computer*, 48(8):26–34, 2015.

[11] Jee Whan Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, volume 45, pages 115–126, 2010.

[12] The CSB library, Nov. 2014. http://gauss.cs.ucsb.edu/~aydin/csb/html/.

[13] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[14] M. F. Dolz. *Energy-aware matrix computations on multithreaded architectures*. PhD thesis, Departmento de Ingenería y Ciencia de Computadores, Universidad Jaume I, 2015.

[15] J. Dongarra and M. A. Heroux. Toward a new metric for ranking high performance computing systems. Sandia Report SAND2013-4744, Sandia National Laboratories, June 2013.

[16] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1986.

[17] M. Duranton, K. De Bosschere, A. Cohen, J. Maebe, and H. Munk. HiPEAC vision 2015, 2015. https://www.hipeac.org/publications/vision/.

[18] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer architecture*, ISCA'11, pages 365–376, 2011.

[19] S. H. Fuller and L. I. Millett. *The Future of Computing Performance: Game Over or Next Level?* National Research Council of the National Academies, 2011.

[20] A. Genser, C. Bachmann, C. Steger, R. Weiss, and J. Haid. Power emulation based DVFS efficiency investigations for embedded systems. In *International Symposium on System on Chip (SoC), 2010*, pages 173 –178, sept. 2010.

[21] The Green500 list, Nov. 2015. http://www.green500.org.

[22] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Pub., San Francisco, 2011.

[23] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing Design and Analysis of Algorithms*. The benjamin / Cummings Publishing Company, Inc., 1994.

[24] R. Lucas. Top ten Exascale research challenges, 2014. http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf.

[25] G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[26] A. Munshi, editor. *The OpenCL Specification Version 2.0*. Khronos OpenCL Working Group, 2014.

[27] NVIDIA. GPU-accelerated libraries, 2015. https://developer.nvidia.com/gpu-accelerated-libraries.

[28] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2.3.1 edition, August 2009.

[29] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[30] S. Williams, N. Bell, J. Choi, M. Garland, L. Oliker, and R. Vuduc. Sparse matrix vector multiplication on multicore and accelerator systems. In Jakub Kurzak, David A. Bader, and Jack Dongarra, editors, *Scientific Computing with Multicore Processors and Accelerators*. CRC Press, 2010.

CAPÍTULO 2

Métodos iterativos para la resolución de sistemas de ecuaciones dispersos

En este capítulo se presentan los fundamentos matemáticos utilizados en el desarrollo de esta Tesis Doctoral, describiendo, de forma breve, la nomenclatura utilizada en algunas operaciones algebraicas, las propiedades matriciales y vectoriales más importantes, y algunos métodos de resolución de sistemas de ecuaciones lineales (SEL), concretamente los métodos iterativos sobre los que se ha desarrollado el trabajo de investigación. Además, se incluye una descripción de la arquitectura de las GPUs de NVIDIA, con una introducción a su programación con CUDA a través de la operación AXPY, así como una descripción de la evolución que las GPUs de NVIDIA han tenido desde G80 hasta Kepler, describiendo con más detalle aquellas que han sido utilizadas en el desarrollo del trabajo de investigación, y proporcionando pautas de optimización sobre su programación.

2.1 Notación y definiciones

Las Matemáticas son una ciencia formal que, a partir de unas verdades evidentes llamadas axiomas, y estableciendo un razonamiento lógico, deduce y estudia relaciones entre entidades abstractas (números, símbolos, figuras geométricas, etc.). La Matemática Computacional se apoya en las Ciencias Matemáticas, para el desarrollo y aplicación de métodos matemáticos avanzados a los problemas que aparecen en ingeniería, tecnología, industria, en la empresa, las administraciones y un largo etcétera. En este apartado, se revisan los fundamentos matemáticos requeridos para la comprensión de los conceptos desarrollados en está tesis. Para un estudio más detallado de estos fundamentos pueden consultarse los textos [39, 45, 47, 53].

2.1.1 Nociones básicas sobre vectores y matrices

***Definición* 2.1** *Un vector es una tabla (array) unidimensional de números (entradas o componentes del vector) pertenecientes a un cuerpo $\mathbb{K}$, siendo éste en general el cuerpo $\mathbb{C}$ (números complejos) o $\mathbb{R}$ (números reales), cuyos elementos están ordenados en una columna de dimensión $n \times 1$ (vector columna) u ordenados en una fila de dimensión $1 \times n$ (vector fila), en donde $n \in \mathbb{N} - \{0\}$. El conjunto de vectores de tamaño n se representa como $V_n(\mathbb{K})$. Los vectores se suelen representar mediante letras minúsculas con o sin algún carácter sobre la propia letra: $v, \vec{v}, \widetilde{v}, \bar{v}, ...$, y sus elementos o componentes se representan con la misma letra y un subíndice numérico, indicando su orden.*

La forma de representar un vector es la siguiente:

$$\textit{vector fila}: \ v^T = (v_1, v_2, v_3, ..., v_n); \ \textit{vector columna}: \ v = (v_1, v_2, v_3, ..., v_n)^T = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{pmatrix}.$$

***Definición* 2.2** *Una matriz es una tabla bidimensional de números (entradas) ordenados en filas y columnas. Cuando una matriz tiene m filas y n columnas, se dice que su dimensión es $m \times n$, en donde $m,n \in \mathbb{N} - \{0\}$. El conjunto de todas las matrices de tamaño $m \times n$ se representa como $\mathscr{M}_{m\times n}(\mathbb{K})$, donde $\mathbb{K}$ es el campo o cuerpo (estructura algebraica) al cual pertenecen las entradas, siendo éste en general $\mathbb{C}$ (números complejos) o $\mathbb{R}$ (números reales). Las matrices se suelen representar por letras mayúsculas: A, B, C,..., y sus elementos se denotan con la misma letra en minúsculas y con subíndices numéricos indicando su orden.*

La forma de representar una matriz es la siguiente:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{pmatrix},$$

que se denota en forma abreviada como $A_{m\times n} = A^{m\times n} = (a_{ij})_{m\times n}$ o bien por $A = (a_{ij})$, con i=1,...,m y j=1,...,n. En cualquiera de los casos, se dice que la matriz es de orden $m \times n$, que el índice i hace referencia a la fila del elemento de la matriz y el índice j a la columna.

***Definición* 2.3** *La matriz identidad se define cono una matriz cuadrada ($m = n$) en la que todos los elementos de su diagonal principal ($a_{i,j}$ tal que $i = j$) son uno, y el resto de elementos son cero. Estas matrices se representan por la letra mayúscula I, de modo que una matriz identidad de tamaño n se representa como $I_{n\times n}$, o de forma más simplificada como I_n.*

***Definición* 2.4** *La transpuesta de una matriz $A = (a_{ij})$ de orden $m \times n$ se representa por A^T, en donde A^T se construye a partir de la matriz A intercambiando filas por columnas; es decir $A^T = (a_{ji})$ con i=1,...,m y j=1,...,n.*

***Definición* 2.5** *Una matriz A es simétrica cuando se cumple que $A = A^T$; es decir, para todo par de elementos a_{ij} y a_{ji} pertenecientes a la matriz A, entonces $a_{ij} = a_{ji}$.*

***Definición* 2.6** *Una matriz cuadrada A de orden $n \times n$ se dice que es invertible cuando existe otra matriz cuadrada B de orden $n \times n$ tal que $AB = BA = I_n$. Entonces la matriz B puede representarse como A^{-1}, ya que denota a la inversa de A.*

***Definición* 2.7** *Una matriz A cuadrada de dimensión $n \times n$ es ortogonal, si su transpuesta coincide con su inversa; es decir, si $A^T = A^{-1}$ o, alternativamente, si $AA^T = I$.*

***Definición* 2.8** *Sea una matriz A simétrica de dimensión $n \times n$ y sea un vector x de dimensión n, ambos definidos en $\mathbb{R}$. Se dice que A es definida positiva si para todos los vectores no nulos $x \in \mathbb{R}^n$, se cumple que $x^*Ax > 0$.*

2.1.2 Operaciones básicas sobre vectores y matrices

***Operación* 2.1** *Dadas dos matrices $A = (a_{ij})$ y $B = (b_{ij})$, con A y $B \in \mathbb{R}^{m\times n}$, la suma de estas matrices obtiene una nueva matriz $C = A + B$, con $C \in \mathbb{R}^{m\times n}$ y $C = (c_{ij}) = (a_{ij} + b_{ij})$.*

***Operación* 2.2** *Dadas dos matrices $A = (a_{ij})$ y $B = (b_{ij})$, con A y $B \in \mathbb{R}^{m\times n}$, la resta de estas matrices obtiene una nueva matriz $C = A - B$, con $C \in \mathbb{R}^{m\times n}$ y $C = (c_{ij}) = (a_{ij} - b_{ij})$.*

***Operación* 2.3** *Dado un escalar α, y una matriz $A = (a_{ij})$, con $\alpha \in \mathbb{R}$ y $A \in \mathbb{R}^{m\times n}$, el producto del escalar y la matriz obtiene una nueva matriz $B = \alpha A$, de forma que $B \in \mathbb{R}^{m\times n}$ y $B = (b_{ij}) = \alpha(a_{ij})$.*

***Operación* 2.4** *Dada una matriz A de dimensión $m \times n$ y un vector de dimensión n, el producto Ax es un nuevo vector de tamaño m, cumpliéndose que, para calcular su i-ésima entrada, se procede de la forma siguiente:*

$$(Ax)_i = \sum_{j=1}^{n} A_{ij}x_j.$$

***Operación* 2.5** *Dadas dos matrices $A_{m\times n}$ y $B_{n\times p}$, el producto de ambas es una nueva matriz $C = AB$, donde C es una matriz de tamaño $m \times p$, cumpliéndose que, para calcular su (ij)-ésima entrada, se procede de la forma siguiente:*

$$(AB)_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj},$$

Por último, indicar que esta operación no es conmutativa, es decir $AB \neq BA$.

2.1.3 Valores y vectores propios de una matriz

***Definición* 2.9** *Sea A una matriz cuadrada de dimensión $n \times n$, que representa un operador lineal. Se denominan vectores propios, o autovectores, de este operador a aquellos vectores no nulos que, al ser transformados por el operador lineal, dan lugar a un escalado de dichos vectores, lo que implica que sus direcciones no cambian. Este escalar, representado por λ, recibe el nombre de valor propio, autovalor o valor característico. Es decir, un valor propio de A es un escalar $\lambda \neq 0$, tal que existe un vector $v \neq 0$ denominado vector propio que cumple que $Av = \lambda v$.*

El cálculo algebraico de los valores y vectores propios de una matriz, cuando ésta actúa como operador lineal, aplicando el teorema de Cayley-Hamilton se resume en:

1. Cálculo del polinomio característico de la matriz, mediante la expresión $p_A(\lambda) = |A - \lambda I_n|$.
2. Búsqueda de las raíces del polinomio característico tal que $p_A(\lambda) = 0$. A las raíces encontradas se les denomina valores propios de la matriz. Al número de veces con que aparece una misma raíz se le denomina multiplicidad algebraica.
3. Para cada valor propio λ, se obtiene el conjunto de vectores propios asociados v_λ a partir de la expresión $(A - \lambda I_n)x = 0$. El subespacio propio de A asociado a λ es el conjunto v_λ formado por todos los vectores $x \in \mathbb{R}^n$ que cumplen la expresión anterior. Al número de vectores propios asociados a un valor propio, se le denomina multiplicidad geométrica.

Una aplicación directa del calculo de valores y vectores propios, es la diagonalización de matrices. Una matriz es diagonalizable si y sólo si, para cada valor propio, la multiplicidad algebraica coincide con la multiplicidad geométrica de los vectores propios asociados. Si una matriz es diagonalizable, puede expresarse como $A = V\Lambda V^{-1}$, donde Λ es una matriz diagonal compuesta por los valores propios en su diagonal principal, y V es una matriz de paso compuesta por los vectores propios asociados a cada valor propio.

2.1.4 Normas vectoriales y matriciales

Las normas proporcionan una métrica dentro de los espacios vectoriales. Son una medida de la misma forma que la longitud lo es, a través de los valores absolutos sobre la recta real, y con ellas se pretende analizar el tamaño de entes multicomponentes, como vectores y matrices. Además son métricas muy útiles para estudiar los errores en problemas numéricos, ya que permiten medir distancias entre vectores y matrices.

***Definición* 2.10** *La norma de un vector $x \in \mathbb{R}^n$ es una función $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, cuya representación matemática viene determinada por $\|x\|$, satisface las siguientes propiedades:*

1. $\|x\| \geqslant 0$, *y* $\|x\| = 0$ *si y sólo si* $x = 0$.
2. $\|\alpha x\| = |\alpha|\|x\|, \forall \alpha \in \mathbb{R}$.
3. $\|x+y\| \leq \|x\| + \|y\|, \forall x, y \in \mathbb{R}^n$ *(Desigualdad triangular).*

La cantidad $\|x\|$ se llama magnitud o longitud del vector x, definiéndose la distancia entre dos vectores $x,\ y \in \mathbb{R}^n$ respecto a la norma como $\|x-y\|$.

Las p-normas son muy empleadas en computación, por lo que se utilizan diferentes subíndices en la doble barra derecha para representar distintos tipos de normas vectoriales, siendo definidas como:

$$\|x\|_p = (|x_1|^p + ... + |x_n|^p)^{\frac{1}{p}},\ con\ p \geqslant 1.$$

Las normas vectoriales más importantes pertenecientes a esta familia de normas, para $p = 1$, $p = 2$ y $p = \infty$ son:

$$\|x\|_1 = |x_1| + ... + |x_n| \quad , \quad \textit{(suma de magnitudes)}.$$
$$\|x\|_2 = \sqrt{|x_1|^2 + ... + |x_n|^2} = \sqrt{x^T x} \quad , \quad \textit{(norma Euclídea)}.$$
$$\|x\|_\infty = \text{máx}(|x_1|, |x_2|, ..., |x_n|) = \max_{1 \leq p \leq n} |x_i| \quad , \quad \textit{(norma del máximo)}.$$

La norma Euclídea corresponde con el concepto intuitivo de longitud o distancia respecto al origen. Si $x = (x_1, x_2, x_3)$, su norma-2 representa la longitud de una recta que une los puntos $(0,0,0)$ y (x_1, x_2, x_3).

***Definición* 2.11** *Una norma matricial sobre el conjunto de las matrices $m \times n$ es una función de valor real definida como $f : \mathbb{R}^{m \times n} \longrightarrow \mathbb{R}$, y que satisface las siguientes propiedades para todas las matrices A y B de dimensión $m \times n$ y para todos los números $\alpha \in \mathbb{R}$:*

1. $\|A\| \geqslant 0$, *y* $\|A\| = 0$ *si y sólo si* $A = 0$.
2. $\|\alpha A\| = |\alpha|\|A\|$.
3. $\|A+B\| \leq \|A\| + \|B\|$.
4. $\|AB\| \leq \|A\|\|B\|$.

El valor $\|A\|$ se denomina magnitud de la matriz, definiéndose la distancia entre dos matrices $A,\ B \in \mathbb{R}^{m \times n}$ respecto a la norma matricial como $\|A-B\|$. En álgebra numérica, se utilizan diferentes subíndices en la doble barra derecha para representar distintos tipos de normas matriciales. Las más utilizadas son:

$$\textit{Norma de Frobenius} : \|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2}, \quad \textit{Norma-1} : \|A\|_1 = \max_{1 \leq j \leq n} \left(\sum_{i=1}^{m} |a_{ij}| \right),$$
$$\textit{Norma-}\infty : \|A\|_\infty = \max_{1 \leq i \leq m} \left(\sum_{j=1}^{n} |a_{ij}| \right), \quad \textit{Norma-2} : \|A\|_2 = \sqrt{\rho(A^T A)}.$$

La Norma-2 es la norma euclídea, o norma espectral, donde $\rho(A^TA)$ es el radio espectral de A^TA, e indica el máximo de los valores propios de la matriz A^TA en valor absoluto, verificándose que $\rho(A) \leq \|A\|$ para cualquier norma.

2.1.5 Condicionamiento de una matriz

Durante la resolución de un sistema de ecuaciones, es posible que se produzcan pequeños cambios en los coeficientes, siendo necesario conocer qué impacto producen estos cambios en el cálculo de la solución. Cuando estos errores son pequeños se dice que el sistema está bien condicionado, mientras que se dirá que está mal condicionado si estos errores son grandes. Para determinar qué tipo de sistema está asociado una aplicación concreta, se estudia el número de condición de la matriz de coeficientes.

***Definición* 2.12** *Sea una matriz A cuadrada de dimensión $n \times n$. Se define su número de condición como $cond(A) = \|A\| \times \|A^{-1}\|$. donde $\|A\|$ es una norma de A.*

Entre las propiedades más destacables del número de condición aparecen las siguientes:

1. $cond(A) \geq 1$.
2. $cond(A^{-1}) = cond(A)$.
3. $cond(\lambda A) = cond(A), \forall \lambda \neq 0$.

A partir de ellas se puede afirmar que un sistema está mal condicionado si la matriz de coeficientes del sistema cumple que $\|A\|\|A^{-1}\|$ se diferencia notablemente de 1.

2.1.6 Tipos de matrices

El tipo de matriz viene determinado por la distribución de sus entradas. A continuación se comentan algunos tipos de matrices, destacando aquellas que son mencionadas en el desarrollo de esta tesis.

***Definición* 2.13** *Una matriz A es diagonal, si es cuadrada $A \in \mathbb{R}^{n\times n}$ y todos los elementos de la matriz son ceros excepto los que pertenecen a su diagonal principal; es decir $a_{ij} = 0$, si $i \neq j$ con $i = 1,..,n$ y $j = 1,..,n$. Su representación matemática es de la forma: $A = diag(a_{ij},...,a_{nn})$, (ver Figura 2.1 a).*

***Definición* 2.14** *Una matriz A es triangular superior, si es cuadrada $A \in \mathbb{R}^{n\times n}$, y todos los elementos de la matriz por debajo de su diagonal principal son ceros; es decir $a_{ij} = 0$, si $i > j$ con $i = 1,..,n$ y $j = 1,..,n$, (ver Figura 2.1 b).*

***Definición* 2.15** *Una matriz A es triangular inferior, si es cuadrada $A \in \mathbb{R}^{n\times n}$, y todos los elementos de la matriz por encima de su diagonal principal son ceros; es decir $a_{ij} = 0$, si $i < j$ con $i = 1,..,n$ y $j = 1,..,n$, (ver Figura 2.1 c).*

***Definición* 2.16** *Se define una matriz tridiagonal, como aquella en la que todos sus elementos son nulos excepto los que cumplen que $|j - i| > 1$. La representación matemática de éstas matrices viene dada por la expresión: $A = tridiag(a_{ij-1}, a_{ij}, a_{ij+1})$, (ver Figura 2.1 d).*

$$\begin{pmatrix} \star & 0 & 0 & 0 \\ 0 & \star & 0 & 0 \\ 0 & 0 & \star & 0 \\ 0 & 0 & 0 & \star \end{pmatrix} \begin{pmatrix} \star & \star & \star & \star \\ 0 & \star & \star & \star \\ 0 & 0 & \star & \star \\ 0 & 0 & 0 & \star \end{pmatrix} \begin{pmatrix} \star & 0 & 0 & 0 \\ \star & \star & 0 & 0 \\ \star & \star & \star & 0 \\ \star & \star & \star & \star \end{pmatrix} \begin{pmatrix} \star & \star & 0 & 0 \\ \star & \star & \star & 0 \\ 0 & \star & \star & \star \\ 0 & 0 & \star & \star \end{pmatrix}$$

(a) (b) (c) (d)

Figura 2.1: Matrices: (a) diagonal, (b) triangular superior, (c) triangular inferior, (d) tridiagonal. El símbolo $\star$ representa cualquier valor.

2.1.7 Matrices dispersas

Inicialmente, se considera que una matriz es dispersa cuando el porcentaje de elementos no nulos es elevado, aunque de un modo más formal, se dice que una matriz es dispersa cuando el número de entradas no nulas es un múltiplo de la dimensión de la matriz, $nnz(A) = O(n)$, o bien cuando la densidad de una matriz es menor que un determinado valor umbral, $nnz(A)/n^2 < umbral$. Este último criterio, aplicado tanto a la matriz como a las filas o columnas de la matriz, será el utilizado para decidir la estructura de datos utilizada para almacenar la matriz y el algoritmo que resuelva una determinada operación algebraica. Desde un punto de vista más práctico, se puede considerar que una matriz es dispersa sólo si vale la pena manejarla como tal en lo que se refiere a rendimiento y eficiencia computacional, utilizando para ello estructuras de datos y algoritmos específicos.

2.1.7.1 Formatos de almacenamiento para matrices dispersas

Puesto que existe una gran cantidad de tipos de matrices dispersas, se dispone de una gran variedad de formatos (esquemas) de almacenamiento de matrices dispersas, donde cada uno de los cuales explota una, o varias, de las características de las matrices dispersas. La operación a resolver y la arquitectura de la máquina sobre la cual se va a ejecutar determinarán el esquema de almacenamiento elegido. Seguidamente, se muestra un listado de los formatos de almacenamiento más relevantes [32, 37, 43, 48, 55]:

- **DNS** *(Dense format).*
- **DIA** *(Diagonal format).*
- **ELL** *(Ellpack-Itpack Generalized Diagonal format).*
- **ELLR** *(ELL format variant, R stores the maximum number of elements per row).*
- **ELLR-T** *(ELLR format variant, special format for graphics units).*
- **CSR** *(Compressed Sparse Row format).*
- **CSC** *(Compressed Sparse Column format).*
- **COO** *(Coordinate format).*
- **HYBRID** *(ELL/COO format).*
- **PKT** *(Packet format).*
- **BND** *(Linpack Banded format).*
- **BSR** *(Block Sparse Row format).*
- **MSR** *(Modified Compressed Sparse Row format).*
- **SKK** *(Symmetric Skyline format).*
- **NSK** *(Non Symmetric Skyline format).*
- **LNK** *(Linked list storage format).*
- **JAD** *(The Jagged Diagonal format).*
- **SSS** *(The Symmetric Sparse Skyline format).*
- **USS** *(The Unsymmetric Sparse Skyline format).*
- **BVR** *(Variable Block Row format).*

$$A = \begin{pmatrix} 1 & 2 & 0 & 4 \\ 9 & 0 & 0 & 5 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 9 & 1 \end{pmatrix} \quad \begin{array}{lcl} \text{AA} & = & (1\ \ 2\ \ 4\ \ 9\ \ 5\ \ 1\ \ 9\ \ 1) \\ \text{JA} & = & (0\ \ 1\ \ 3\ \ 0\ \ 3\ \ 3\ \ 2\ \ 3) \\ \text{IA} & = & (0\ \ 3\ \ 5\ \ 6\ \ 8) \end{array}$$

Figura 2.2: Matriz *A*, representada mediante el esquema de almacenamiento CSR, utilizando una indexación de acuerdo al lenguaje "*C*" (*0-indexing*).

```
1  void SpMV_CSR( int num_filas, float *AA, int *JA, int *IA, float *x, float *y) {
2    int fila, j, inicio_fila, fin_fila;
3    float val;
4
5    for( fila = 0; fila < num_filas; fila++ ){
6      val = 0;
7      inicio_fila = IA[fila];
8      fin_fila = IA[fila + 1];
9      for( j = ininio_fila; j < fin_fila; j++ ){
10       val += AA[j] * x[JA[j]];
11     }
12     y[fila] = val;
13   }
14   return;
15 }
```

Figura 2.3: Implementación secuencial en lenguaje *"C"* de la operación SpMV con el formato de almacenamiento CSR. La matriz tiene *num_filas* filas, y está almacenada haciendo uso de los vectores (AA, JA, IA). El vector multiplicador se denota como *x*, mientras que el vector resultado del producto se denota como *y*.

A continuación, se explican en detalle los dos formatos de almacenamiento de matrices dispersas utilizados en el desarrollo de esta tesis: CSR y ELL.

2.1.7.2 Formato disperso de fila comprimida CSR

El formato CSR es uno de los más populares para almacenar matrices dispersas, por ser simple y compacto, además de poder ser utilizado para almacenar matrices dispersas de cualquier tipo. La definición formal de este esquema es la siguiente:

***Definición* 2.17** *Dada una matriz dispersa $A_{n\times n}$, en la que nnz es el número de elementos no nulos, para almacenarla en formato CSR son necesarios tres vectores:*

- *AA, de dimensión nnz, contiene los elementos no nulos de A ordenados por filas, en donde los elementos de cada fila no están necesariamente ordenados.*
- *JA, de dimensión nnz, contiene los índices columna de los elementos de AA.*
- *IA, de dimensión $n+1$, contiene elementos que indican para cada posición, el índice donde empieza una fila en los otros dos vectores (AA, JA). El último valor de IA siempre apunta al primer valor fuera de rango.*

En la parte izquierda de la Figura 2.2 se muestra una matriz *A* en formato denso, mientras que en su parte derecha se muestra como se representaría la misma matriz utilizando el esquema de almacenamiento CSR, en el que sólo se han almacenado las entradas no nulas. Por su parte, la Figura 2.3 muestra cómo resolver un producto matriz-vector, en el que se observa que el acceso sobre el vector *y* sólo requiere una indirección, mientras que el acceso al vector *x* es más costoso ya que requiere dos pasos de indirección.

2.1.7.3 Formato disperso ELLPACK/ITPACK ELL

El formato ELLPACK/ITPACK, denominado de forma reducida como ELL, es muy general, por lo que se puede utilizar para almacenar cualquier tipo de matriz. La principal ventaja de este formato es la utilización de estructuras bidimensionales, de modo que el valor de la fila es parte intrínseca de la estructura, lo que permite reducir el número de estructuras necesarias. La definición formal de este esquema se enuncia a continuación:

***Definición* 2.18** *Dada una matriz dispersa $A_{n\times n}$, para almacenarla en formato ELL se requiere, en primer lugar, calcular el valor $num\_cols\_por\_fila$ que contiene el máximo del número de elementos no nulos de cada fila, a partir del cual se definen dos matrices:*

- *Datos es una matriz de dimensión $n \times num\_cols\_por\_fila$, que almacena en las primeras posiciones de cada fila los valores no nulos de la correspondiente fila de la matriz. Las posiciones no utilizadas en cada fila contienen un valor no significativo.*
- *Índices es una matriz de dimensión $n \times num\_cols\_por\_fila$, que almacena en las primeras posiciones de cada fila los índices de columnas de los correspondientes posiciones de la matriz Datos. Las posiciones no utilizadas en cada fila contienen un valor no significativo.*

En algunos casos, el acceso por columnas de estas matrices mejora las prestaciones de los algoritmos, para lo cual se definen los siguientes vectores:

- *AA, de dimensión $n \times num\_cols\_por\_fila$, contiene los elementos de la matriz "Datos" almacenados por columna.*
- *JA, de dimensión $n \times num\_cols\_por\_fila$, contiene los elementos de la matriz "Indices columna" almacenados por columna.*

La principal desventaja de este formato es que se almacenan más elementos de los estrictamente necesarios, como puede observarse en la Figura 2.4. Este problema se agrava cuanto mayor es la distancia entre el valor de *num_cols_por_fila* y el número medio de elementos no nulos por fila. En cambio, este formato es muy eficiente dado que el acceso a la información sólo requiere un paso de indirección, como puede observarse en la Figura 2.5 en el que se calcula el producto matriz-vector usando este formato de almacenamiento. Por esta razón, si el número de elementos no significativo es reducido, el algoritmo de la Figura 2.5 es mucho más eficiente que el incluido en la Figura 2.3.

$$A = \begin{pmatrix} 1 & 2 & 0 & 4 \\ 9 & 0 & 0 & 5 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 9 & 1 \end{pmatrix} \qquad AA = \begin{pmatrix} 1 & 2 & 4 \\ 9 & 5 & \star \\ 1 & \star & \star \\ 9 & 1 & \star \end{pmatrix} \qquad JA = \begin{pmatrix} 0 & 1 & 3 \\ 0 & 3 & \star \\ 3 & \star & \star \\ 2 & 3 & \star \end{pmatrix}$$

(a) Matriz dispersa. (b) Datos. (c) Índices.

Figura 2.4: Matriz *A*, representada mediante el esquema de almacenada ELL, utilizando una indexación de acuerdo al lenguaje "*C*" (*0-indexing*).

```
1  void SpMV_ELL( int num_filas, int num_cols_por_fila, float *AA, int *JA,
2                 float *x, float *y) {
3    int fila, j, col;
4    float acum, val;
5
6    for( fila = 0; fila < num_filas; fila++ ){
7      acum = 0;
8      for( j = 0; j < num_cols_por_fila; j++ ){
9        col = JA[num_filas * j + fila];
10       val = AA[num_filas * j + fila];
11       if(val!=0){
12         acum += val * x[col];
13       }
14     }
15     y[fila] = acum;
16   }
17   return;
18 }
```

Figura 2.5: Implementación secuencial en lenguaje *"C"* de la operación SpMV con el formato de almacenamiento ELL. La matriz tiene *num_filas* filas, *num_cols_por_fil*a entradas por fila (como máximo) y está formada por los vectores (AA, JA). El vector multiplicador se denota como x, mientras que el vector resultado del producto se denota como y.

2.2 Resolución de sistemas de ecuaciones lineales

En muchas aplicaciones relacionadas con la ciencia y la ingeniería, la resolución de SEL suele ser la etapa más costosa desde el punto de vista computacional. La creciente complejidad de estas aplicaciones ha aumentado la necesidad de acelerar la resolución de los SEL manteniendo unos niveles de precisión y fiabilidad adecuados. La elección del mejor método para la resolución de un SEL debe considerar las características de la matriz, tanto estructurales (densa o dispersa, simétrica o no simétrica, ...) como numéricas (definida positiva o no, número de condición, ...).

En esta sección, tras clasificar los diferentes métodos que permiten resolver un SEL, se presentan los métodos utilizados en el desarrollo de esta tesis. En adelante, se considera que la representación matricial de un sistema con m ecuaciones lineales y con n incógnitas es la siguiente:

$$Ax = b \;\Rightarrow\; \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots \;\; \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}. \tag{2.1}$$

en donde $x = (x_1, \ldots, x_n)$ es el vector de incógnitas, $b = (b_1, \ldots, b_m)$ representa el vector de términos independientes, y a_{ij} son los coeficientes de la matriz.

2.2.1 Clasificación de métodos para la resolución de SEL

Existen dos grandes clases que agrupan los diferentes métodos que resuelven un SEL: los métodos directos y los métodos iterativos. La principal diferencia entre ambos métodos es que, en los primeros, el coste del algoritmo se conoce de antemano, mientras que en los segundos las propiedades de la matriz pueden variar el número de iteraciones y por tanto el coste final del método. Seguidamente se presentan las características más importantes de cada una de estas clases.

Los **métodos directos** transforman el problema $Ax = b$ en otro más sencillo, cuya solución resulta menos costosa, siendo habitual la obtención de un sistema triangular [36]. Esta transformación se puede realizar

mediante alguna de las variantes del método de Gauss (factorización LU, factorización de Cholesky, método de Gauss-Jordan, ...), con o sin pivotamiento, o utilizando alguna transformación ortogonal (transformaciones de Householder, rotaciones de Givens, ...) que permita obtener una factorización QR. El hecho de que, tanto la transformación como la posterior resolución del sistema reducido se calculen en un número de pasos conocidos a priori, y que estos métodos presenten una gran robustez numérica, los hace especialmente útiles para resolver sistemas donde la matriz de coeficientes es densa. Esa es la razón por la que durante muchos años, especialmente entre los años 60 y los años 80, estos métodos han sido los más utilizados para la resolución de SEL.

Cuando se trabaja con sistemas en los que la matriz de coeficientes es dispersa y de gran dimensión, la utilización de los métodos directos suele introducir nuevos elementos no nulos en el proceso de transformación que puede aumentar el coste del proceso. Con el objeto de reducir este "llenado" (*fill-in*), la aplicación de los métodos directos sobre matrices dispersas se descompone en 4 fases: ordenación, factorización simbólica, factorización numérica y solución del sistema triangular. El objetivo básico de las dos primeras fases es limitar el nivel de llenado en el proceso de transformación, buscando una permutación inicial del sistema (ordenación) que lleve a una estructura final del sistema reducido (factorización simbólica). Por desgracia, la introducción de estas fases limita la aplicación del pivotamiento reduciendo con ello la robustez numérica de los métodos, lo que puede aconsejar la utilización de otras alternativas.

Los **métodos iterativos** [32, 50] construyen una sucesión de vectores que converge a la solución del sistema; es decir, aplican una serie de transformaciones algebraicas sobre una solución inicial que va mejorando hasta obtener otra nueva con el nivel de precisión adecuado. Además de estar expuestos a errores de redondeo, estos métodos también pueden sufrir errores de truncamiento, por lo que no se puede asegurar que un SEL arbitrario pueda ser resuelto por un método iterativo determinado. Para resolver esta incertidumbre, es posible utilizar una solución inicial que esté próxima a la solución final, bien al ser obtenida como solución de un SEL similar o bien por ser la solución obtenida por otro método del mismo SEL. También es posible utilizar algún tipo de precondicionador que, al ser aplicado sobre el SEL, modifique sus propiedades numéricas, aumentado de forma considerable la fiabilidad y eficiencia del proceso. Es importante indicar que el precondicionador no se aplica explícitamente sobre la matriz de coeficientes del SEL, ya que podría modificar su estructura, sino que se introduce una nueva transformación en el proceso iterativo que se encarga de aplicar el precondicionador en la resolución del SEL. Por último, respecto de las ventajas de los métodos iterativos sobre los métodos directos, indicar que los primeros presentan una gran escalabilidad cuando el problema crece y, además, destaca la facilidad en el diseño e implementación de versiones paralelas.

Existe una gran variedad de métodos iterativos, cada uno con unas características específicas, que se pueden agrupar como sigue:

- Métodos iterativos estacionarios clásicos. El periodo de tiempo en donde se concentraron los esfuerzos de investigación abarca el intervalo entre los años 50 y 70. Algunos de los métodos más relevantes pertenecientes a esta familia son Jacobi, Gauss-Seidel y el método de sobrerelajación (SOR) [36, 50].
- Métodos iterativos no estacionarios o de proyección. Su periodo de mayor actividad investigadora comienza a mediados de los años 70. La mayoría se basan en la generación de subespacios de Krylov y, a diferencia de los anteriores, las variables asociadas contienen información que varía en cada iteración. Algunos de los métodos más relevantes pertenecientes a esta familia son CG, BICG, BICGSTAB y el GMRES [42, 49].
- Métodos multinivel. Surgen con la idea de mejorar la eficiencia de los métodos iterativos estacionarios clásicos. Su periodo de actividad investigadora comienza a mediados de los años 70. Estos métodos también pueden utilizarse como precondicionadores de los métodos basados en subespacios de Krylov. Una excelente exposición de las ideas fundamentales de los métodos multinivel y su aplicación puede consultarse en [33, 35, 41].

- Métodos de descomposición del dominio. Comprende una familia de técnicas que resuelven los SEL obtenidos como la discretización de problemas gobernados por ecuaciones diferenciales en derivadas parciales (EDP) y que se populariza a partir de los años 80. Se basan en el paradigma de "Dividir, combinar y vencer", para lo cual la resolución del SEL se divide en subproblemas casi independientes. En la actualidad, se usan principalmente como precondicionadores, considerándose, matemáticamente, como una extensión de precondicionadores simples por bloques como los de Jacobi o Gauss-Seidel [50]. Más información acerca de esta familia de métodos puede encontrarse en [46, 52].

2.2.2 Métodos de proyección

Los métodos de proyección se definen como una metodología que permite extraer una aproximación a la solución de un SEL a partir de la información existente en un subespacio. Los métodos basados en la generación de subespacios de Krylov se encuentran dentro de esta familia, y en cada etapa producen una proyección ortogonal sobre un subespacio de Krylov generado por alguna variante de los métodos de Arnoldi o de Lanczos [32, 50]. Sin embargo no son los únicos, ya que la mayoría de los métodos iterativos que se emplean en la actualidad, para resolver SEL dispersos y de gran dimensión, utilizan algún proceso de proyección, como es el caso del método de Gauss-Seidel en el que el subespacio sobre el que se proyecta es el generado por el vector unitario e_i [50].

Un método de proyección parte de un sistema lineal $Ax = b$ de dimensión $n \times n$, y de un subespacio vectorial $\mathscr{K}$ de dimensión $k < n$, generado por una base de vectores $V = \{v_1, \ldots, v_k\}$, una aproximación a la solución $\widehat{x}$ dentro del subespacio $\mathscr{K}$, para lo cual $\widehat{x} = Vy$, en donde y también es de dimensión k. Uno de los métodos más aceptados para calcular el vector y es tomar como restricción que el vector residuo, $r = b - A\widehat{x}$, sea ortogonal a otro subespacio $\mathscr{L}$ generado por una base de vectores $W = \{w_1, \ldots, w_k\}$, de forma que

$$W^T \cdot (b - AVy) = 0\,. \tag{2.2}$$

Despejando y de la expresión (2.2), se obtiene un sistema $k \times k$ lineal, $y = (W^T AV)^{-1} \cdot W^T b$, en donde $W^T AV$ debe ser no singular. El Algoritmo 2.1 muestra el esquema básico de un método de proyección.

Algoritmo 2.1 Prototipo de método de proyección

1: **mientras** no convergencia **hacer**
2: Seleccionar dos subespacios $\mathscr{K}$ y $\mathscr{L}$
3: Escoger las bases $V = \{v_1, \ldots, v_k\}$ y $W = \{w_1, \ldots, w_k\}$
4: $r := b - Ax$
5: $y := (W^T AV)^{-1} W^T r$
6: $x := x + Vy$
7: **fin mientras**

La relación existente entre los dos subespacios implicados, $\mathscr{K}$ (*subespacio de búsqueda*) y $\mathscr{L}$ (*subespacio de restricciones*), define el tipo de proyección empleado. Cuando $\mathscr{K} = \mathscr{L}$ se dice que se utilizan *proyecciones ortogonales* mientras que si $\mathscr{K} \neq \mathscr{L}$ se dice que se utilizan *proyecciones oblicuas*. Comentar además que en la búsqueda de una aproximación a la solución, por cada paso que se realiza en el proceso de proyección, la dimensión de los subespacios se incrementa en uno.

Para mostrar cómo se introducen los subespacios de Krylov como alternativa dentro de los métodos de proyección, supongamos que x_0 es una aproximación inicial de $Ax = b$, con un residuo inicial $r_0 = b - Ax_0$, y sea $\bar{x}$ la solución de

$$A\bar{x} = r_0\,. \tag{2.3}$$

A partir de la expresión del residuo inicial, se obtiene que $Ax_0 = b - r_0$, y corrigiendo la aproximación inicial x_0 con $\bar{x}$ se obtiene la solución exacta

$$A(x_0 + \bar{x}) = Ax_0 + A\bar{x} = b - r_0 + r_0 = b \, . \tag{2.4}$$

La expresión en (2.4) sugiere calcular $\bar{x}$ mediante aproximaciones sucesivas, tal que la solución computada se aproxime cada vez con mayor precisión a la de la ecuación $Ax = b$, lo que es posible utilizando los subespacios de Krylov. En estos métodos se calcula, en cada iteración, una aproximación x_k explorando el subespacio

$$x_k = x_0 + \mathscr{K}_k(A, r_0) = x_0 + span\{r_0, Ar_0, A^2 r_0, \ldots, A^{k-1} r_0\} \ , \ k = 1, 2, 3, \ldots \, , \tag{2.5}$$

en donde x_0 es una solución inicial arbitraria, la expresión $K_k(A, r_0)$ representa el subespacio de Krylov de dimensión k generado por la matriz A, y el residuo inicial es $r_0 = b - Ax_0$, cumpliéndose que los vectores $r_0, Ar_0, \ldots, A^{k-1} r_0$ son los que generan el subespacio $span\{r_0, Ar_0, \ldots, A^{k-1} r_0\}$.

Las diferentes formas de definir restricciones para aproximar x_k llevan a construir diferentes métodos de proyección basados en la generación de subespacios de Krylov:

1. En la aproximación de Ritz-Galerkin se impone que el residuo $r_k = b - Ax_k$ sea ortogonal al subespacio $\mathscr{K}_k(A, r_0)$, es decir que $(b - Ax_k) \perp \mathscr{K}_k(A, r_0)$.

2. En la aproximación del mínimo residuo, se impone que el residuo $r_k = b - Ax_k$ sea ortogonal al subespacio $A \cdot \mathscr{K}_k(A, r_0)$, o lo que es lo mismo, la norma del residuo de $x_k, \| \, b - Ax_k \, \|_2$, sea mínima sobre el subespacio $\mathscr{K}_k(A, r_0)$.

3. En la aproximación de Petrov-Galerkin se impone que el residuo de $r_k = b - Ax_k$ sea ortogonal a un subespacio de dimensión k, denotado por $\mathscr{L}_k$, y construido como $\mathscr{L}_k = \mathscr{K}_k(A^T, r_0)$, es decir que $(b - Ax_k) \perp \mathscr{L}_k$.

4. En la aproximación del mínimo error se impone que $\| \, x - x_k \, \|_2$ sea mínimo sobre todos los vectores del subespacio $A^T \cdot \mathscr{K}_k(A^T, r_0)$.

Dentro del enfoque de Ritz-Galerkin se incluyen el método del gradiente conjugado (CG), el método de Lanczos, o el método FOM y sus variantes IOM y DIOM. Los métodos más importantes incluidos dentro del enfoque del mínimo residuo son GMRES, MINRES y ORTHODIR. Para el enfoque de Petrov-Galerkin los más representativos son los métodos Bi-CG y el QMR. Con el enfoque del mínimo error, los métodos más representativos son SYMMLQ y GMERR. A partir de todos estas aproximaciones, se han desarrollado métodos que extraen ideas combinadas de los cuatro enfoques, y que han dado lugar a otros métodos iterativos no menos populares como lo son CGS, Bi-CGSTAB, FGMRES y GMRESR, sus detalles de implementación, bases matemáticas y análisis numérico pueden encontrarse en [31, 32, 34, 50].

2.2.3 Método CG

El algoritmo del Gradiente Conjugado (CG) fue propuesto originalmente en diciembre de 1952 por Hestenes y Stiefel[42], y es uno de los métodos iterativos más conocidos para resolver sistemas lineales dispersos simétricos y definidos positivos. El método realiza una proyección ortogonal sobre un subespacio de Krylov $\mathscr{K}_k(A, r_0)$, donde r_0 es el residuo inicial, cumpliéndose que los residuos satisfacen la condición de Ritz-Galerkin.

Existen diferentes alternativas para deducir el CG, una de las cuales asume que se han completado **k etapas del método de Lanczos**. Como resultado de este proceso iterativo, se obtiene una matriz tridiagonal T_k al aplicar la base ortonormal V_k del subespacio de Krylov $\mathscr{K}_k(A, r_0)$ sobre la matriz A [50],

$$\mathscr{K}_k(A, r_0) = span\{V_k\} \ , \ V_k^T V_k = I_k \ \Rightarrow \ V_k^T A V_k = T_k \ , \ T_k \, es \, tridiagonal \, .$$

La derivación del método parte de la factorización LU de la matriz tridiagonal, $T_k = L_k U_k$,

$$\begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & . & . & . & \\ & & \beta_{k-1} & \alpha_{k-1} & \beta_k \\ & & & \beta_k & \alpha_k \end{pmatrix} = \begin{pmatrix} 1 & & & & \\ \lambda_2 & 1 & & & \\ & . & . & & \\ & & \lambda_{k-1} & 1 & \\ & & & \lambda_k & 1 \end{pmatrix} \times \begin{pmatrix} \eta_1 & \beta_2 & & & \\ & \eta_2 & \beta_3 & & \\ & & . & . & \\ & & & \eta_{k-1} & \beta_k \\ & & & & \eta_k \end{pmatrix} . \tag{2.6}$$

Aplicando la expresión incluida en el Algoritmo 2.1, la aproximación a la solución se expresa como

$$x_k = x_0 + V_k (V_k^T A V_k)^{-1} V_k^T r = x_0 + V_k T_k^{-1} (V_k^T r) = x_0 + V_k U_k^{-1} L_k^{-1} (\beta_1 e_1) \, ,$$

y agrupando convenientemente los operandos, se modifica la anterior expresión como

$$P_k = V_k U_k^{-1} \;\; , \;\; z_k = L_k^{-1} (\beta_1 e_1) \;\; \Rightarrow \;\; x_k = x_0 + P_k z_k \, .$$

La definición de P_k permite obtener una recurrencia que relaciona los vectores de V_k y P_k,

$$p_k = \eta_k^{-1} [v_k - \beta_k p_{k-1}] \, ,$$

en donde los escalares se obtienen directamente a partir de (2.6),

$$\lambda_k = \frac{\beta_k}{\eta_{k-1}} \;\; , \;\; \eta_k = \alpha_k - \lambda_k \beta_k \, . \tag{2.7}$$

Por su parte, la definición de z_k y la siguiente descomposición

$$z_k = \begin{bmatrix} z_{k-1} \\ \hline \varsigma_k \end{bmatrix} \;\; , \;\; \varsigma_k = -\lambda_k \varsigma_{k-1} \, ,$$

permiten obtener una nueva recurrencia para actualizar la solución del sistema:

$$x_k = x_{k-1} + \varsigma_k p_k \, .$$

Estas expresiones son la base para definir una variante del método de Lanczos para resolver un SEL.

Es importante destacar que este método se obtiene aplicando la eliminación Gaussiana sin pivotamiento sobre un sistema tridiagonal, por lo que podría fallar en el caso que el pivote elegido no fuese adecuado. Existen alternativas que resuelven este problema, como el uso de la factorización QR, la factorización LQ, o la incorporación de pivotamiento, cuyo uso permite obtener otros métodos, como el Mínimo Residuo, LQ-Simétrico … [50].

Otra alternativa para derivar el método CG, explota **las propiedades numéricas de los vectores** implicados. Por construcción, es sabido que los vectores residuos son ortogonales, dado que $r_k = \sigma_k v_{k+1}$, pero la ortogonalidad de los vectores auxiliares, p_k, no es conocida. Aplicando la definición de P_k sobre el producto $P_k^T A P_k$,

$$P_k^T A P_k = (V_k U_k^{-1})^T A V_k U_k^{-1} = U_k^{-T} V_k^T A V_k U_k^{-1} = U_k^{-T} T_k U_k^{-1} = U_k^{-T} L_k \, ,$$

se obtiene que el producto inicial, cuyo resultado debe ser una matriz simétrica, es igual al producto de dos matrices triangulares inferiores, que también es triangular inferior. La única posibilidad de que ambas condiciones se cumplan es que el producto inicial sea una matriz diagonal, y por tanto se cumple que los vectores auxiliares son A-ortogonales, o conjugados.

Estas dos propiedades pueden ser utilizadas para derivar el método CG, que parte de las recurrencias que relacionan los vectores auxiliares con los vectores solución y con los vectores residuo:

$$x_{k+1} = x_k + \alpha_k p_k \;\; , \;\; r_{k+1} = r_k - \alpha_k A p_k \, . \tag{2.8}$$

Inicialización $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$	
while $(\tau_j > \tau_{\text{máx}})$	Bucle asociado al método CG
O1. $v_j := Ap_j$	O1. SPMV
O2. $\alpha_j := \sigma_j/(p_j^T v_j)$	O2. DOT
O3. $x_{j+1} := x_j + \alpha_j p_j$	O3. AXPY
O4. $r_{j+1} := r_j - \alpha_j v_j$	O4. AXPY
O5. $\zeta_j := r_{j+1}^T r_{j+1}$	O5. DOT
O6. $\beta_j := \zeta_j/\sigma_j$	O6. Operación escalar
O7. $\sigma_{j+1} := \zeta_j$	O7. Operación escalar
O8. $p_{j+1} := r_{j+1} + \beta_j p_j$	O8. XPAY (similar a AXPY)
O9. $\tau_{j+1} :=\| r_{j+1} \|_2= \sqrt{\sigma_{j+1}}$	O9. 2-norma del vector (sqrt)
$j := j+1$	
end while	

Figura 2.6: Algoritmo del método del gradiente conjugado, donde $\tau_{\text{máx}}$ es la cota inferior de la norma del residuo.

Aplicando la ortogonalidad de los residuos, se obtiene cómo calcular el valor de α_k:

$$r_k^T r_{k+1} = r_k^T (r_k - \alpha_k A p_k) = 0 \;\Rightarrow\; \alpha_k = \frac{r_k^T r_k}{r_k^T A p_k} \,. \tag{2.9}$$

Por su parte, los vectores auxiliares, p_{k+1}, se definen como una combinación lineal de r_{k+1} y p_k:

$$p_{k+1} = r_{k+1} + \beta_k p_k \,, \tag{2.10}$$

a partir de la cual es posible derivar una expresión que modifica la definición de α_k:

$$r_k^T A p_k = (p_k - \beta_{k-1} p_{k-1})^T A p_k = p_k^T A p_k \;\Rightarrow\; \alpha_k = \frac{r_k^T r_k}{p_k^T A p_k} \,. \tag{2.11}$$

Aprovechando la A-ortogonalidad de los vectores auxiliares, se puede multiplicar (2.10) por Ap_k para calcular el valor

$$\beta_k = -\frac{r_{k+1}^T A p_k}{p_k^T A p_k} \,, \tag{2.12}$$

sobre la cual se puede aplicar la definición de los vectores residuos que aparece en (2.8), obteniendo

$$A p_k = \frac{1}{\alpha_k}(r_{k+1} - r_k) \;\Rightarrow\; \beta_k = -\frac{1}{\alpha_k}\frac{r_{k+1}^T (r_{k+1} - r_k)}{(p_k^T A p_k)} = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \,. \tag{2.13}$$

La Figura 2.6 muestra el algoritmo resultante, en el que los escalares utilizados, α_k y β_k, no coinciden con los elementos de T_k.

2.2.4 Método BICG

El método del gradiente biconjugado (BICG), propuesto por Lanczos en 1952 [44], se deriva del algoritmo de biortogonalización de Lanczos, de forma análoga a cómo el algoritmo del CG se deriva del algoritmo de Lanczos [50]. Este algoritmo está indicado para resolver sistemas donde las matrices snn no singulares y no simétricas, resolviendo tanto el sistema original $Ax = b$ como el sistema $A^T \widetilde{x} = \widetilde{b}$, aunque este último es

ignorado en la formulación del algoritmo. Para ello, se realiza una proyección oblicua que involucra a los subespacios de Krylov $\mathscr{K}_k(A,r_0)$ y $\mathscr{K}_k(A^T,r_0)$, donde r_0 es el residuo inicial asociado al sistema $Ax=b$, cumpliéndose que los residuos cumplen la condición de Petrov-Galerkin. Una rigurosa explicación de las bases matemáticas en las que se sustenta el algoritmo del BICG puede encontrarse en [38].

La derivación del método BICG puede realizarse de modo similar a como se dedujo el método CG, ejecutando **k etapas del método de Lanczos para matrices no simétricas**. La principal diferencia con el caso anterior es que ahora se generan dos subespacios de Krylov diferentes, $\mathscr{K}_k(A,r_0)$ y $\mathscr{K}_k(A^T,\widetilde{r}_0)$,

$$\left.\begin{array}{rcl}\mathscr{K}_k(A,r_0) & = & span\{V_k\}\\ \mathscr{K}_k(A^T,\widetilde{r}_0) & = & span\{W_k\}\end{array}\right\}\ ,\ W_k^TV_k=I_k\ \Rightarrow\ W_k^TAV_k=T_k\ ,\ T_k\,es\ tridiagonal\ .$$

No existe ninguna regla para elegir cuáles deben ser los vectores iniciales de los dos subespacios de Krylov, salvo que no pueden ser ortogonales, esto es $(\widetilde{r}_0^Tr_0\neq 0)$, siendo una buena elección que ambos vectores sean iguales, $(r_0=\widetilde{r}_0)$. Siguiendo el planteamiento seguido en CG, la primera etapa para derivar el método debe calcular la factorización LU de T_k, que presenta la siguiente estructura

$$\begin{pmatrix}\alpha_1 & \gamma_2 & & & \\ \beta_2 & \alpha_2 & \gamma_3 & & \\ & . & . & . & \\ & & \beta_{k-1} & \alpha_{k-1} & \gamma_k \\ & & & \beta_k & \alpha_k\end{pmatrix}=\begin{pmatrix}1 & & & & \\ \lambda_2 & 1 & & & \\ & . & . & & \\ & & \lambda_{k-1} & 1 & \\ & & & \lambda_k & 1\end{pmatrix}\times\begin{pmatrix}\eta_1 & \gamma_2 & & & \\ & \eta_2 & \gamma_3 & & \\ & & . & . & \\ & & & \eta_{k-1} & \gamma_k \\ & & & & \eta_k\end{pmatrix}. \tag{2.14}$$

La aplicación de la expresión del Algoritmo 2.1, que define la aproximación a la solución, da lugar a

$$x_k=x_0+V_k(W^TAV)^{-1}W^Tr=x_0+V_kT_k^{-1}(W_k^Tr)=x_0+V_kU_k^{-1}L_k^{-1}(\beta_1e_1)\ ,$$

que al agrupar convenientemente permite obtener,

$$P_k=V_kU_k^{-1}\ \ ,\ \ z_k=L_k^{-1}(\beta_1e_1)\ \ \Rightarrow\ \ x_k=x_0+P_kz_k\ .$$

A partir de esta expresión es posible definir una recurrencia asociada a la solución del sistema, tal y como se realizó en el método CG. De modo similar se podría haber calculado una recurrencia para calcular la solución del sistema $A^T\widetilde{x}=\widetilde{b}$:

$$\widetilde{P}_k=W_kL_k^{-T}\ \ ,\ \ \widetilde{z}_k=U_k^{-T}(\widetilde{\beta}_1e_1)\ \ \Rightarrow\ \ \widetilde{x}_k=\widetilde{x}_0+\widetilde{P}_k\widetilde{z}_k\ ,$$

cumpliéndose que los vectores que forman P_k y $\widetilde{P}_k$ son A-ortogonales,

$$\widetilde{P}^TAP_k=L_k^{-1}W_k^TAV_kU_k^{-1}=L_k^{-1}T_kU_k^{-1}=I.$$

A partir de las recurrencias de las soluciones y de la A-ortogonalidad de P_k y $\widetilde{P}_k$, es posible derivar el algoritmo asociado al método BICG, tal y como aparece en la Figura 2.7.

Una implementación real del método BICG debería considerar todas las posibles condiciones en las que éste podría finalizar. En el caso de que se complete alguno de los subespacios de Krylov, el vector residuo correspondiente será nulo lo que obliga a finalizar el método. Este hecho se considera positivo cuando se finaliza la secuencia asociada al sistema a resolver r_j, porque se ha generado completamente la matriz T_k y por tanto la solución x_k es la solución del sistema. Por esta razón se convierte en la condición de parada del método, como puede observarse en la Figura 2.7. En cambio, se considerará como un fallo si finaliza la secuencia $\widetilde{r}_j$, puesto que la solución x_k no coincide con la solución del sistema. El método también puede finalizar prematuramente si, en una etapa, los nuevos vectores generados en cada secuencia son ortogonales

Inicialización $r_0, \widetilde{r}_0, p_0, \widetilde{p}_0, x_0, \sigma_0, \tau_0; j := 0$	
while $(\tau_j > \tau_{\text{máx}})$	Bucle asociado al método BICG
O1. $v_j := Ap_j$	O1. SPMV
O2. $\alpha_j := \sigma_j / (\widetilde{p}_j^T v_j)$	O2. DOT
O3. $x_{j+1} := x_j + \alpha_j p_j$	O3. AXPY
O4. $r_{j+1} := r_j - \alpha_j v_j$	O4. AXPY
O5. $w_j := A^T \widetilde{p}_j$	O5. SPMV
O6. $\widetilde{r}_{j+1} := \widetilde{r}_j - \alpha_j w_j$	O6. AXPY
O7. $\zeta_j := \widetilde{r}_{j+1}^T r_{j+1}$	O7. DOT
O8. $\beta_j := \zeta_j / \sigma_j$	O8. Operación escalar
O9. $\sigma_{j+1} = \zeta_j$	O9. Operación escalar
O10. $p_{j+1} := r_{j+1} + \beta_j p_j$	O10. XPAY (similar a AXPY)
O11. $\widetilde{p}_{j+1} := \widetilde{r}_{j+1} + \beta_j \widetilde{p}_j$	O11. XPAY (similar a AXPY)
O12. $\tau_{j+1} := \| r_{j+1} \|_2$	O12. 2-norma del vector (DOT + sqrt)
$j := j + 1$	
end while	

Figura 2.7: Algoritmo del método del gradiente biconjugado, donde $\tau_{\text{máx}}$ es la cota inferior de la norma del residuo.

$(\widetilde{r}_j^T r_j = 0)$. En la bibliografía, esta situación se denomina "serious breakdown" y es habitual cuando se genera más de una secuencia de Krylov, requiriendo la aplicación de técnicas especificas, como "look-ahead", para resolverlas. Una tercera causa de finalización anticipada del método BICG, se produce cuando un elemento diagonal de T_k es nulo, es decir cuando $\widetilde{p}^T v_j = 0$, en cuyo caso falló la factorización LU. Como en el caso del método CG. la solución a este problema requiere la utilización de la factorización QR o la factorización LQ, dando lugar a los métodos QMR y LQ [50].

2.2.5 Método BICGSTAB

Uno de los mayores inconvenientes del método BICG es el hecho de requerir operaciones con la matriz del sistema y su transpuesta, A y A^T. Por una parte, la optimización de estas operaciones puede aconsejar almacenar la matriz dos veces, duplicando el coste espacial del método. Además, en algunos casos la matriz no es disponible de modo explícito, por lo que el manejo de su transpuesta no es posible, imposibilitando la utilización del método BICG. Estas razones, entre otras, motivaron el desarrollo de un conjunto de variantes que no operan con la transpuesta, como es el caso del método del Gradiente Conjugado al Cuadrado (CGS), desarrollado por Sonneveld en 1984, y del método del Gradiente Biconjugado Estabilizado (BICGSTAB) propuesto, originariamente, por Van der Vorst en 1992 [54]. Otros documentos de relevancia en donde se puede encontrar una extensa explicación del método son [40, 51].

El método **CGS** considera que sólo se desea resolver el sistema $Ax = b$, por lo que los vectores relacionados con la resolución de $A^T \widetilde{x} = \widetilde{b}$, $\widetilde{r}$ y $\widetilde{p}$, sólo se utilizan para obtener los escalares calculados en cada iteración. Un análisis del método BICG permite definir los residuos y los vectores conjugados asociados a los sistemas mediante un par de polinomios asociados a las matrices que definen ambos SEL,

$$\left.\begin{array}{rcl} Ax = b & \Rightarrow & r_k = \phi_k(A) r_0 \ , \quad p_k = \pi_k(A) r_0 \\ A^T \widetilde{x} = \widetilde{b} & \Rightarrow & \widetilde{r}_k = \phi_k(A^T) \widetilde{r}_0 \ , \quad \widetilde{p}_k = \pi_k(A^T) \widetilde{r}_0 \end{array}\right\} \ , \ \textit{donde } \phi_k(0) = 1$$

de modo que los escalares pueden definirse como

$$\begin{aligned}\alpha_k &= \frac{\widetilde{r}_k^T r_k}{\widetilde{p}_k^T A p_k} = \frac{(\phi_k(A^T)\widetilde{r}_0)^T \phi_k(A) r_0}{(\pi_k(A^T)\widetilde{r}_0)^T A \pi_k(A) r_0} = \frac{\widetilde{r}_0^T \phi_k^2(A) r_0}{\widetilde{r}_0^T A \pi_k^2(A) r_0}\,,\\ \beta_k &= \frac{\widetilde{r}_{k+1}^T r_{k+1}}{\widetilde{r}_k^T r_k} = \frac{(\phi_{k+1}(A^T)\widetilde{r}_0)^T \phi_{k+1}(A) r_0}{\phi_k(A^T)\widetilde{r}_0)^T \phi_k(A) r_0} = \frac{\widetilde{r}_0^T \phi_{k+1}^2(A) r_0}{\widetilde{r}_0^T \phi_k^2(A) r_0}\,.\end{aligned}$$

Posteriormente, se transforman las recurrencias asociadas a los residuos y los vectores conjugados en términos de polinomios, y se calculan las expresiones que definen el cuadrado de los polinomios, que son la base para definir los residuos y los vectores conjugados asociados al método:

$$\begin{aligned} r_{k+1} = r_k - \alpha_k A p_k &\Rightarrow \phi_{k+1} = \phi_k - \alpha_k A \pi_k \Rightarrow \phi_{k+1}^2 = (\phi_k - \alpha_k A \pi_k)^2 \Rightarrow \hat{r}_k = \phi_k^2(A) r_0\,,\\ p_{k+1} = r_{k+1} + \beta_k p_k &\Rightarrow \pi_{k+1} = \phi_{k+1} + \beta_k \pi_k \Rightarrow \pi_{k+1}^2 = (\phi_{k+1} + \beta_k \pi_k)^2 \Rightarrow \hat{p}_k = \pi_k^2(A) r_0\,,\end{aligned}$$

y por tanto,

$$\alpha_k = \frac{\widetilde{r}_0^T \hat{r}_k}{\widetilde{r}_0^T A \hat{p}_k}\;,\;\; \beta_k = \frac{\widetilde{r}_0^T \hat{r}_{k+1}}{\widetilde{r}_0^T \hat{r}_k}\,.$$

Este método funciona bien en muchos casos, pero el hecho de duplicar los polinomios magnifica los errores de redondeo lo que genera grandes variaciones en los residuos, y por tanto el cálculo de su norma puede ser bastante inexacto [50].

El método **BICGSTAB** intenta resolver la convergencia irregular del método CGS, combinando ideas de los métodos BICG y GMRES. La idea básica de este método es utilizar un polinomio alternativo en la generación de residuos y vectores conjugados asociados al sistema $A^T\widetilde{x} = \widetilde{b}$, sacando partido al hecho que este sistema no debe ser resuelto. La definición de este polinomio, y su relación con los residuos y vectores conjugados del sistema $Ax = b$, es la siguiente:

$$\psi_{k+1}(A) = (1 - \omega_k A)\psi_k(A) = \Rightarrow \left\{ \begin{array}{rcl} \hat{r}_k = \psi_k(A)\phi_k(A) r_0 & \Rightarrow & \hat{r}_{k+1} = (1 - \omega_k A)(\hat{r}_k - \alpha_k A \hat{p}_k)\,,\\ \hat{p}_k = \psi_k(A)\pi_k(A) r_0 & \Rightarrow & \hat{p}_{k+1} = \hat{r}_{k+1} + \beta_k (I - \omega_k A)\hat{p}_k\,. \end{array} \right.$$

Por su parte, el cálculo de los escalares requiere la modificación de las expresiones definidas en el método BICG para que incluyan los nuevos vectores, dando lugar a las siguientes expresiones:

$$\hat{\rho}_k(A) = \widetilde{r}_0^T \hat{r}_k \;\Rightarrow\; \alpha_k = \frac{\widetilde{r}_0^T \hat{r}_k}{\widetilde{r}_0^T A \hat{p}_k} = \frac{\hat{\rho}_k}{\widetilde{r}_0^T A \hat{p}_k} \quad , \quad \beta_k = \frac{\hat{\rho}_{k+1}}{\hat{\rho}_k} \cdot \frac{\alpha_k}{\omega_k}\,.$$

La elección de ω_k determina las propiedades numéricas del método, pudiéndose definir un conjunto de métodos que únicamente varían en el modo en el que se calcula este escalar. En el caso del método BICGSTAB clásico, este escalar se obtiene al aplicar una etapa del método GMRES por lo que se calcula como

$$s_k = \hat{r}_k - \alpha_k A \hat{p}_k \;\Rightarrow\; \hat{r}_{k+1} = (1 - \omega_k A) s_k\,,\; \omega_k = \frac{s_k^T A s_k}{(A s_k)^T A s_k}\,.$$

Respecto de la actualización de la solución, ésta se puede obtener directamente de la definición del residuo de la siguiente forma:

$$\hat{r}_{k+1} = \hat{r}_k - \alpha_k A \hat{p}_k - \omega_k A s_k \;\Rightarrow\; x_{k+1} = x_k + \alpha_k \hat{p}_k + \omega_k s_k\,.$$

La Figura 2.8 muestra el algoritmo resultante.

Inicialización $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$	
while $(\tau_j > \tau_{\text{máx}})$	Bucle asociado al método BICGSTAB
O1. $v_j := Ap_j$	O1. SPMV
O2. $\alpha_j := \sigma_j / r_0^T v_j$	O2. DOT
O3. $s_j := r_j - \alpha_j v_j$	O3. AXPY-like
O4. $\tau_j := \| s_j \|_2$	O4. 2-norma del vector (DOT + sqrt)
if $(\tau_j < \tau_{max})$ **then**	
O5. $x_{j+1} := x_j + \alpha_j p_j$	O5. AXPY
break	
end if	
O6. $v_j := As_j$	O6. SPMV
O7. $\rho_j := (v_j^T s_j)/(v_j^T v_j)$	O7. DOT + DOT
O8. $x_{j+1} := x_j + \alpha_j p_j + \rho_j s_j$	O8. AXPY + AXPY
O9. $r_{j+1} := s_j - \rho_j v_j$	O9. XPAY (similar a AXPY)
O10. $\zeta_j := r_0^T r_{j+1}$	O10. DOT
O11. $\beta_j := (\zeta_j / \sigma_j) * (\alpha_j / \rho_j)$	O11. Operación escalar
O12. $\sigma_{j+1} := \zeta_j$	O12. Operación escalar
O13. $p_{j+1} := r_{j+1} + \beta_j (p_j - \rho_j v_j)$	O13. XPAY + XPAY (similares a AXPY)
O14. $\tau_{j+1} := \| r_{j+1} \|_2$	O14. 2-norma del vector (DOT + sqrt)
$j := j + 1$	
end while	

Figura 2.8: Algoritmo del método del gradiente biconjugado estabilizado, donde $\tau_{\text{máx}}$ es la cota inferior de la norma del residuo.

Respecto de la finalización del método, BICGSTAB mantiene las condiciones ya comentadas psrs BICG e incluye una nueva finalización anticipada, relacionada con la etapa de GMRES, y más concretamente con el cálculo de ω_k.

2.3 Técnicas básicas de precondicionado

Los métodos iterativos convergen de forma rápida cuando la matriz A que representa al sistema de ecuaciones es muy parecida a la matriz identidad. Desafortunadamente las matrices que definen un problema real no suelen cumplir esta condición. Una alternativa sería modificar el sistema a través de un precondicionador M, de modo que la matriz resultante sí cumpla la condición,

$$Ax = b \;\Rightarrow\; M^{-1}Ax = M^{-1}b\,.$$

La resolución iterativa de un SEL puede presentar errores de redondeo y de truncamiento, en especial para matrices mal condicionadas, en las que pequeñas perturbaciones en la matriz, o en el vector de términos independientes, pueden producir grandes errores en la solución. El uso de un precondicionador permite cambiar el número de condición de un sistema, mejorando con ello las propiedades numéricas, y limitando el impacto de las pequeñas perturbaciones, si se selecciona M tal que , puesto que

$$cond(M^{-1}A) < cond(A)\,.$$

La mejor opción sería conseguir que el número de condición del sistema condicionado fuese igual a 1, para lo cual la matriz asociada debería ser igual a I, y por tanto $M = A^{-1}$. Esta opción no resulta práctica, ya que

O0. $A \rightarrow M$	O0. Cálculo del Precondicionador
Inicialización $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$	
while $(\tau_j > \tau_{\text{máx}})$	Bucle asociado al método CG
O1. $v_j := Ap_j$	O1. SPMV
O2. $\alpha_j := \sigma_j/(p_j^T v_j)$	O2. DOT
O3. $x_{j+1} := x_j + \alpha_j p_j$	O3. AXPY
O4. $r_{j+1} := r_j - \alpha_j v_j$	O4. AXPY
O5. $z_{j+1} := M^{-1} r_{j+1}$	O5. Aplicación del Precondicionador
O6. $\zeta_j := r_{j+1}^T z_{j+1}$	O6. DOT
O7. $\beta_j := \zeta_j/\sigma_j$	O7. Operación escalar
O8. $\sigma_{j+1} := \zeta_j$	O8. Operación escalar
O9. $p_{j+1} := z_{j+1} + \beta_j p_j$	O9. XPAY (similar a AXPY)
O10. $\tau_{j+1} := \| r_{j+1} \|_2$	O10. 2-norma del vector (DOT + sqrt)
$j := j+1$	
end while	

Figura 2.9: Algoritmo del método del gradiente conjugado precondicionado, donde $\tau_{\text{máx}}$ es la cota inferior de la norma del residuo.

el cálculo de la inversa de la matriz es una operación muy costosa, y además la solución del sistema original se calcularía de modo inmediato a partir de ella, por lo que el método iterativo no sería necesario. Es por ello que se buscan otras opciones más rápidas que permitan acelerar la convergencia de los métodos iterativos.

Cuando se trabaja con matrices simétricas y definidas positivas, puede ser interesante que la aplicación del precondicionador mantenga estas propiedades. Una solución es calcular la descomposición de Cholesky del precondicionador y aplicarla sobre el sistema como

$$Ax = b \ , \ M = LL^T \ \ \Rightarrow \ \ (L^{-1}AL^{-T})y = (L^{-1}b) \ , \ y = L^T x \ .$$

Otra opción para preservar la simetría es explotar las propiedades del producto interno respecto a una matriz de dos vectores, definido como una generalización del producto escalar:

$$A = A^T \ \ \Rightarrow \ \ (x,y)_A = y^T Ax = (Ax,y) = (x,Ay) \ .$$

Aprovechando que $M^{-1}A$ es autoadjunto ("self-adjoint") para el producto interno respecto de M,

$$A = A^T \ , \ M = M^T \ \ \Rightarrow \ \ (M^{-1}Ax,y)_M = (Ax,y) = (x,Ay) = (x,MM^{-1}Ay) = (x,M^{-1}Ay)_M \ ,$$

es posible sustituir todos los productos escalares en el método CG por productos internos respecto de M. Así, si se define z_j como el residuo del sistema precondicionado, los escalares asociados al método se podrían calcular como

$$z_j = M{-}1r_j \ \ \Rightarrow \ \ \alpha_k = \frac{(z_k,z_k)_M}{(M{-}1Ap_k,p_k)_M} = \frac{(r_k,z_k)}{(Ap_k,p_k)} \ , \ \beta_k = \frac{(z_{k+1},z_{k+1})_M}{(z_k,z_k)_M} = \frac{(r_{k+1},z_{k+1})}{(r_k,z_k)} \ .$$

Estas expresiones son la base para definir las versiones precondicionadas del CG, BICG y BICGSTAB que aparecen, respectivamente, en las Figuras 2.9, 2.10 y 2.11.

Existe una gran variedad de métodos para el cálculo del precondicionador de un SEL, y en muchos casos la elección del mejor precondicionador debe considerar, además de la reducción en el número de etapas, tanto el coste de cálculo de M como el de su aplicación en cada una de las iteraciones del método.

O0. $A \rightarrow M$	O0. Cálculo del Precondicionador
Inicialización $r_0, \widetilde{r}_0, p_0, \widetilde{p}_0, x_0, \sigma_0, \tau_0; j := 0$	
while $(\tau_j > \tau_{\text{máx}})$	Bucle asociado al método BICG
O1. $v_j := Ap_j$	O1. SPMV
O2. $\alpha_j := \sigma_j / (\widetilde{p}_j^T v_j)$	O2. DOT
O3. $x_{j+1} := x_j + \alpha_j p_j$	O3. AXPY
O4. $r_{j+1} := r_j - \alpha_j v_j$	O4. AXPY
O5. $z_{j+1} := M^{-1} r_{j+1}$	O5. Aplicación del Precondicionador
O6. $w_j := A^T \widetilde{p}_j$	O6. SPMV
O7. $\widetilde{r}_{j+1} := \widetilde{r}_j - \alpha_j w_j$	O7. AXPY
O8. $\widetilde{z}_{j+1} := M^{-1} \widetilde{r}_{j+1}$	O8. Aplicación del Precondicionador
O9. $\zeta_j := \widetilde{r}_{j+1}^T z_{j+1}$	O9. DOT
O10. $\beta_j := \zeta_j / \sigma_j$	O10. Operación escalar
O11. $\sigma_{j+1} = \zeta_j$	O11. Operación escalar
O12. $p_{j+1} := z_{j+1} + \beta_j p_j$	O12. XPAY (similar a AXPY)
O13. $\widetilde{p}_{j+1} := \widetilde{z}_{j+1} + \beta_j \widetilde{p}_j$	O13. XPAY (similar a AXPY)
O14. $\tau_{j+1} := \| r_{j+1} \|_2$	O14. 2-norma del vector (DOT + sqrt)
$j := j + 1$	
end while	

Figura 2.10: Algoritmo del método del gradiente biconjugado precondicionionado, donde $\tau_{\text{máx}}$ es la cota inferior de la norma del residuo.

La razón es que, en algunos casos, el sobrecoste incluido por el precondicionador más complejo excede el coste de resolución utilizando un precondicionador más sencillo, aún cuando el número de iteraciones de este último sea mucho mayor. Los métodos más sencillos construyen el precondicionador a partir de un particionado de la matriz del sistema, $A = D - E - F$ donde D incluye sólo la diagonal de la matriz, mientras que E y F contienen, respectivamente, los elementos en los triángulos inferior y superior de A. Algunos de los métodos resultantes son los siguientes:

$$M_{\text{Jacobi}} = D \ , \ M_{\text{Gauss-Seidel}} = D - E \ , \ M_{\text{SSOR}} = (D - \omega E) D^{-1} (D - \omega F) \ .$$

En el caso de que $\omega = 1$, el método SSOR se convierte en el método de Gauss-Seidel simétrico (SGS), en el que un manejo apropiado de las expresiones permite definir M como el producto de una matriz triangular inferior unidad L, y una matriz triangular superior U:

$$M_{\text{SGS}} = (D - E) D^{-1} (D - F) = LU \ , \ L = (D - \omega E) D^{-1} = (I - E D^{-1}) \ , \ U = (D - F) \ .$$

Para el caso de matrices dispersas, los factores obtenidos, L y U, tienen la misma estructura que los triángulos inferior y superior de la matriz A, pero su aplicación sobre el SEL no asegura que el número de condición de la matriz precondicionada esté próxima a 1, principalmente porque la matriz residuo puede ser muy grande:

$$A = LU + R \ \Rightarrow \ R = A - LU = D - E - F - (I - E D^{-1})(D - F) = -E D^{-1} F \ .$$

La obtención de un precondicionador como el producto de dos factores fue el primer paso para definir una familia de métodos, que calculan una descomposición LU incompleta de A, y utilizan los factores obtenidos para definir el precondicionador. Algunos de estos métodos, denominados genéricamente como ILU(k), controlan el llenado de los factores, de modo que sólo algunos elementos que eran cero en A aparecen como

O0. $A \rightarrow M$	O0. Cálculo del Precondicionador
Inicialización $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$	
while $(\tau_j > \tau_{\text{máx}})$	Bucle asociado al método BICGSTAB
O1. $v_j := Ap_j$	O1. SPMV
O2. $\alpha_j := \sigma_j / r_0^T v_j$	O2. DOT
O3. $s_j := r_j - \alpha_j v_j$	O3. AXPY-like
O4. $\tau_j := \| s_j \|_2$	O4. 2-norma del vector (DOT + sqrt)
if $(\tau_j < \tau_{max})$ **then**	
O5. $x_{j+1} := x_j + \alpha_j p_j$	O5. AXPY
break	
end if	
O6. $s_j := M^{-1} s_j$	O6. Aplicación del Precondicionador
O7. $v_j := As_j$	O7. SPMV
O8. $\rho_j := (v_j^T s_j)/(v_j^T v_j)$	O8. DOT + DOT
O9. $x_{j+1} := x_j + \alpha_j p_j + \rho_j s_j$	O9. AXPY + AXPY
O10. $r_{j+1} := s_j - \rho_j v_j$	O10. XPAY (similar a AXPY)
O11. $\zeta_j := r_0^T r_{j+1}$	O11. DOT
O12. $\beta_j := (\zeta_j / \sigma_j) * (\alpha_j / \rho_j)$	O12. Operación escalar
O13. $\sigma_{j+1} := \zeta_j$	O13. Operación escalar
O14. $p_{j+1} := r_{j+1} + \beta_j (p_j - \rho_j v_j)$	O14. XPAY + XPAY (similares a AXPY)
O15. $p_{j+1} := M^{-1} p_{j+1}$	O15. Aplicación del Precondicionador
O16. $\tau_{j+1} := \| r_{j+1} \|_2$	O16. 2-norma del vector (DOT + sqrt)
$j := j + 1$	
end while	

Figura 2.11: Algoritmo del método del gradiente biconjugado estabilizado precondicionado, donde $\tau_{\text{máx}}$ es la cota inferior de la norma del residuo.

no nulos en los factores. El control se realiza a partir del parámetro k, de modo que en ILU(0) no se permite ningún llenado, mientras que ILU(1) sólo se permite el llenado producido a partir de posiciones que en A eran no nulas, y en ILU(2) sólo se permite el llenado a partir de posiciones no nulas el ILU(1). Otros métodos, denominados ILUT(p,τ) controlan que ningún elemento de los factores, en valor absoluto, sea menor que τ, admitiendo, como máximo, p elementos en cada fila, o columna, de los factores, y eliminando los de menor valor absoluto.

2.4 Las GPUs en computación de altas prestaciones (HPC)

Hoy en día, en computación paralela existen dos líneas de desarrollo de arquitecturas, *multicore* y *many-core*. La línea de arquitectura *multicore* (por ejemplo: procesador Intel Core i7 con 4 núcleos y 8 hilos de ejecución [59]), se centra en mantener y mejorar la velocidad de códigos secuenciales en cada uno de sus procesadores, puesto que el desarrollo efectivo de programas comerciales paralelos para estas arquitecturas aún están en una fase temprana. La razón principal consiste en que una migración de todo el software desarrollado supone altos costes para los desarrolladores de software. En cambio, la línea de arquitectura *manycore* requiere una ejecución de códigos con un alto grado de paralelismo. Un ejemplo de ello son las GPUs, cuya principal característica es la gran cantidad de elementos de procesamiento o núcleos de cómputo de que disponen, con arquitecturas relativamente simples, y memorias caché de reducido tamaño. La Figura

2.12 muestra una comparativa entre el espacio de chip dedicado para los distintos bloques funcionales de una CPU y una GPU. Seguidamente, se describen los modelos HPC en dónde están presentes las GPUs, así como cuestiones relativas a su arquitectura, entorno de programación, modelo de ejecución y optimizaciones más relevantes para los modelos de GPUs utilizados en el desarrollo de los trabajos que dan lugar a está Tesis Doctoral.

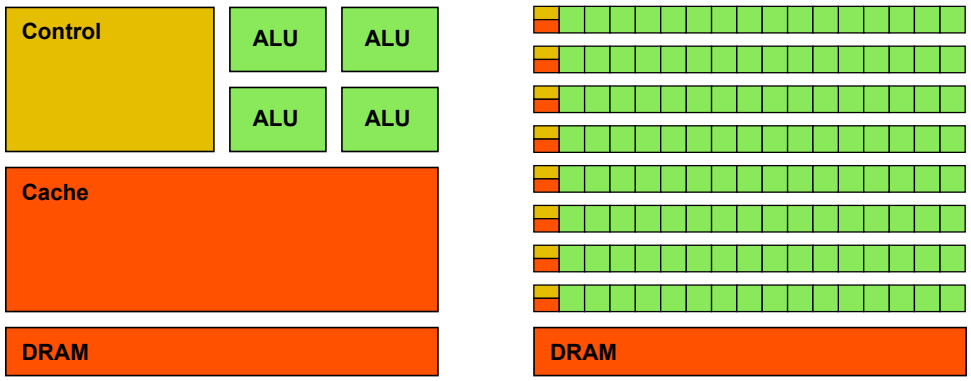


Figura 2.12: Espacio físico (transistores) dedicado a procesamiento (ALU, Unidad Lógica Aritmética), control y elementos de caché, en comparación entre la arquitectura de una CPU (izquierda) con una GPU (derecha) [1] .

2.4.1 Computación Heterogénea

La taxonomía de Flynn [56] permitió clasificar las arquitecturas de computadores en función del flujo de instrucciones y del flujo de datos, dando lugar a cuatro alternativas:

- SISD (*Single Instruction - Single Data*).
- SIMD (*Single Instruction - Multiple Data*).
- MISD (*Multiple Instruction - Single Data*).
- MIMD (*Multiple Instruction - Multiple Data*).

Durante muchos años, los ordenadores personales (OP) se asociaron al modelo SISD, mientras que los modelos SIMD y MIMD eran utilizados únicamente en grandes servidores de cálculo. En los OPs, el incremento de la frecuencia de reloj fue el método utilizado para mejorar las prestaciones, hasta que el aumento de la temperatura de sus componentes obligó a la búsqueda de otras alternativas. La solución que se ha impuesto ha sido la incorporación de varios núcleos computacionales en los OPs, que se ajustan en mayor grado al modelo MIMD. Por su parte, los servidores fueron desarrollados utilizando de modo alternativo las dos variantes del modelo MIMD: los multiprocesadores con memoria compartida (MMC) (ver Figura 2.13a) y los multiprocesadores con memoria distribuida (MMD) (ver Figura 2.13b), hasta que se evidenció que la mejor solución era la que combinaba las características de ambas arquitecturas, en la que los nodos de un MMD se ajustaban al modelo MMC, como es el caso de un clúster de sistemas multinúcleo. Por su parte, el modelo SIMD, que inicialmente fue utilizado para el desarrollo de servidores, como es el caso de los procesadores matriciales, rápidamente fue utilizado para el desarrollo de unidades de cómputo específico

[1] Figura extraída del documento de NVIDIA [91].

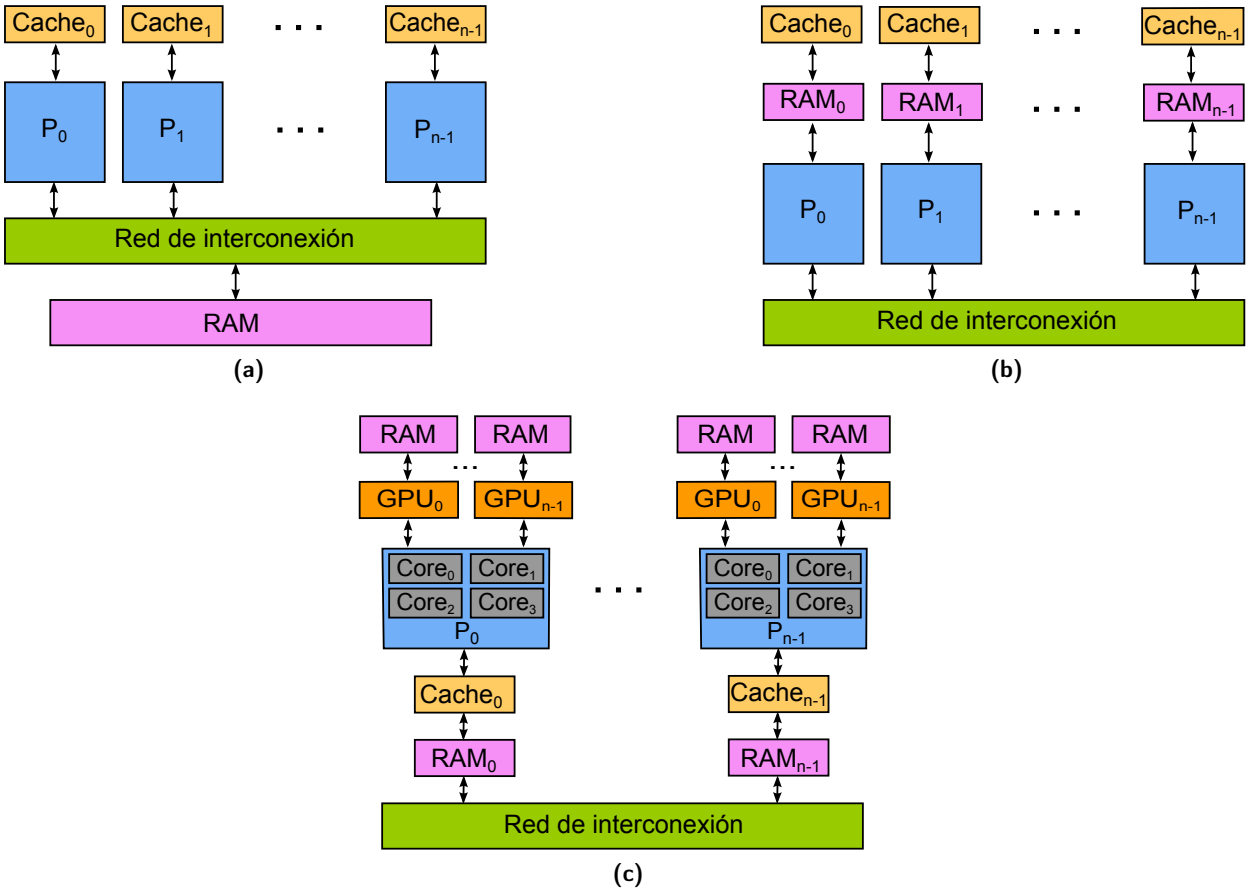


Figura 2.13: Sistemas basados en memoria compartida **(a)**, en memoria distribuida **(b)** y en cluster de multiprocesadores provistos de GPUs **(c)**.

como las unidades vectoriales y las GPUs. La fuerte inversión realizada en la evolución de las GPUs, por parte de empresas relacionadas con el desarrollo de videojuegos, dio lugar a dispositivos de muy alta capacidad computacional, por lo que se consideró su utilización en otros ámbitos. Este es el caso del cálculo científico, donde dio lugar al GPGPU (*General Purpose* GPU), en la que la GPU actúa como coprocesador de la CPU. Tanto ha sido el éxito de este modelo, que se ha integrado en los grandes servidores, y en las primeras posiciones de lista TOP500 [57] y Green500 [58], aparecen algunos sistemas cuyos nodos están formados con MMC y una o varias GPUs (ver Figura 2.13c).

La arquitectura de una GPU está profundamente segmentada y se adecúa a arquitecturas caracterizadas como SIMD (*Single-Instruction, Multiple-Data*), lo que, atendiendo a su modelo de programación, NVIDIA denomina SIMT (*Single-Instruction, Multiple-Thread*) [60]. Las GPUs son emplazadas en tarjetas con RAM dinámica (DRAM) de alta velocidad, y conectadas al computador vía un interface de Entrada/Salida (I/O) de alta velocidad, típicamente un bus PCIe.

Las características de la arquitectura de las GPUs están determinadas históricamente por los requisitos de las aplicaciones gráficas, que son diferentes de los que suelen requerir las tareas computacionales de propósito general. Como la arquitectura cuenta con un gran número de núcleos, y un ancho de banda de memoria relativamente reducido por núcleo, aquellos algoritmos que exponen un alto grado de paralelismo, así como una alta relación de cálculos por dato, se constituyen como candidatos ideales para ser acelerados de forma eficiente por las GPUs. No obstante, también se pueden aplicar otras restricciones, como por ejemplo,

la cantidad de memoria requerida o los patrones de acceso a ésta. Hay que tener en cuenta que, algoritmos en áreas tan diversas como las finanzas [61, 62], química y física [63, 64], dinámica de fluidos [65, 66], álgebra computacional [67, 68], análisis de imágenes [69, 70], etc., han conseguido alcanzar considerables aceleraciones cuando sus códigos se han portado a GPUs, multiplicándolos por un factor elevado con respecto a su ejecución tradicional sobre CPUs.

Las empresas que actualmente lideran el mercado de las GPUs son NVIDIA [71] y AMD [72] (tarjetas gráficas ATI). Por su parte, NVIDIA es la actual líder en el campo de GPGPU, principalmente debido al uso del *framework* que proporciona como apoyo a la programación con CUDA. Éste permite la programación de GPUs en lenguaje C (con la ayuda de algunas extensiones), de tareas de computación de propósito general, acercando su modelo de programación de forma sencilla a programadores no expertos en lenguajes específicos para gráficos, como OpenGL [73, 74] o Cg [75, 76]. Una alternativa a CUDA es OpenCL (*Open Computing Language*) [77], que es un estándar de programación de propósito general en sistemas heterogéneos, con un interfaz de programación de aplicaciones (API) de uso abierto para realizar GPGPU independientemente del dispositivo a utilizar. Con OpenCL, una misma porción de código se puede ejecutar en dispositivos de NVIDIA o AMD, así como también en cualquier CPU habitual y otros dispositivos tales como los procesadores basados en IBM [78] o ARM [79], aunque, en la práctica, es posible que se requiera algún ajuste específico del código para el dispositivo en aras de una mayor eficiencia. Por último, señalar que recientemente han surgido otros entornos de programación de alto nivel, como por ejemplo OpenACC [80], que se ha desarrollado para llevar al cómputo GPU más allá de la era CUDA [81]. En cualquier caso, para el trabajo de investigación desarrollado en esta Tesis Doctoral se utilizó el API de CUDA, debido a que su nivel de madurez es mayor que el API de OpenCL, y además existen evaluaciones que evidencian un rendimiento superior de CUDA [82, 83].

2.4.2 CUDA

CUDA (*Computer Unified Device Architecture*) [84] es el entorno que permite realizar GPU *Computing* en los aceleradores de NVIDIA. Proporciona un conjunto de bibliotecas y herramientas, que permiten el uso de las GPUs de NVIDIA como coprocesadores para acelerar ciertas partes de un programa, generalmente aquellas que presentan un alto coste computacional y un alto nivel de paralelismo de datos. Con cada nueva versión, CUDA extiende su API para hacer más fácil la realización de ciertas operaciones, y añade soporte para las nuevas características introducidas en las sucesivas generaciones de arquitecturas GPU, que se identifican por su revisión de capacidad de cómputo (*Compute Capability*). En los siguientes apartados se describen los elementos de paralelismo propios de los que dispone CUDA, que posibilitan a la GPU desplegar toda su potencia de ejecución masivamente paralela.

2.4.2.1 Modelo de programación CUDA

Una GPU, está compuesta de multiprocesadores (SM), formados por núcleos CUDA sobre los que se implementa el modelo SIMT, a los que se les llama (SP), ver Figura 2.14 . Los SP son procesadores escalares sencillos que comparten una unidad de control, y una pequeña memoria tan rápida como una caché (memoria compartida). Todos los SMs comparten un mismo espacio de memoria global (compuesto por la DRAM de vídeo), con un acceso de elevada latencia pero con gran ancho de banda.

En CUDA, los cálculos son distribuidos a través de una malla (*grid*) de bloques de hilos o hebras (*threads*), en donde todos los bloques tienen el mismo tamaño (número de *threads*) y dimensión. En términos generales, un *grid* puede ser visto como una representación lógica de la propia GPU, un bloque como un SM, y un *thread* como cada uno de los distintos SP. A continuación se detalla el significado de los términos manejados en la programación de GPUs con CUDA.

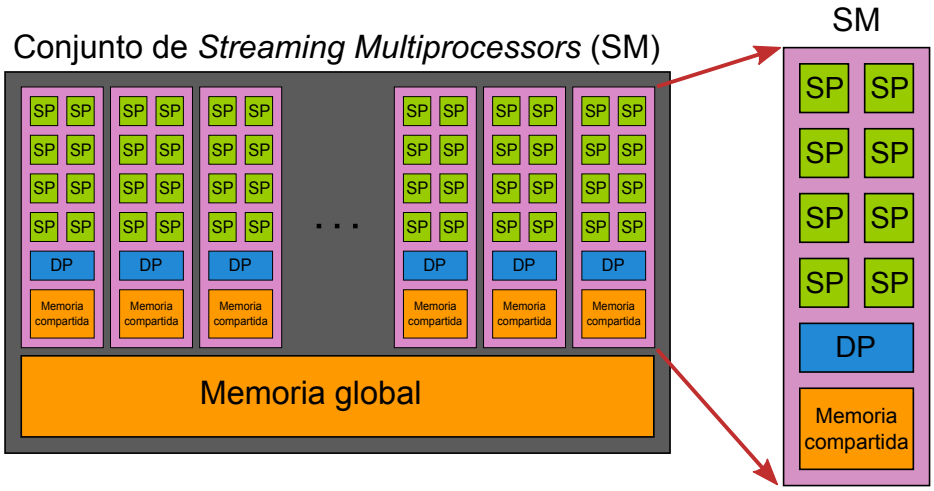


Figura 2.14: Arquitectura básica de una GPU, con 8 procesadores escalares (SP) por SM, una unidad de doble precisión (DP), 16 KB de memoria compartida por SM, y 32 KB (8K entradas de 32 bits) de registros por SM.

- *Threads*: Forman las unidades elementales de ejecución y se deben mapear (asignar) sobre núcleos de procesamiento hardware de forma lógica, a través de las herramientas de programación que proporciona CUDA. Un *thread* se ejecuta sobre un único SP.
- *Warp*: Constituye la unidad mínima de ejecución, y está formado por 32 *threads* consecutivos.
- Bloque: Son agrupaciones de *threads* que han sido asignados a un SM. Los bloques se ejecutan cuando lo permiten los recursos disponibles, en particular, la cantidad requerida de registros o de memoria compartida, y se planifican mediante *warps*. Los bloques se pueden definir con 1, 2 ó 3 dimensiones. Los *threads* que componen un bloque pueden cooperar y sincronizarse a través de la memoria compartida de bloque.
- *Grid*: Es la unidad de trabajo en que se descompone la ejecución de un *kernel*, y está formado por una agrupación de bloques con un mismo tamaño y dimensión. Los *grid* se pueden definir con 1 ó 2 dimensiones, y a partir de la capacidad de cómputo 2.0 hasta con 3 dimensiones. Los bloques de un *grid*, no pueden sincronizarse entre ellos, pero todos los *threads* de un *grid* se pueden comunicar a través de la memoria global.
- *Stream*: Se corresponde con una cola de trabajo en la que todos los *kernels* que lo compone se ejecutan secuencialmente, aunque *kernels* lanzados en "*streams*" diferentes pueden ser ejecutados en paralelo. Salvo que se especifique lo contrario, los *kernels* se lanzan en el *stream* 0.
- *Kernel*: Es una función que contiene una porción de código CUDA, que es ejecutada por todos los *threads* del *grid*, cada uno de los cuales actúa sobre un área diferente de datos que se define a partir de los identificadores de *thread* y de bloque. Junto a la llamada de un *kernel*, se debe especificar la dimensión de su *grid* (`dimG` = número de bloques y `dimB` = número de *threads* por bloque), así como la cantidad de memoria compartida necesaria (parámetro opcional en bytes) y su pertenencia a un determinado *stream* (por defecto al *stream* 0). Las variables `dimG` y `dimB` son del tipo `dim3`, cuyo contenido es accesible de la forma: (`var.x`, `var.y`, `var.z`). Éstas pueden representar 1, 2 ó 3 dimensiones, para definir los tamaños de *grid* y de bloque mediante números enteros, aunque para

1 dimensión se puede utilizar el tipo `int` o `unsigned int`. La definición y llamada de un *grid* se efectúa como sigue:

```
dim3 dimG(nx, ny, nz), dimB(nx, ny, nz);
kernel<<<dimG, dimB, [memCompartida], [numStream]>>>(lista de parámetros);
```

2.4.2.2 Jerarquía de memoria en CUDA

CUDA [84] establece, de acuerdo a la arquitectura de la GPU, distintos niveles de memoria, cada uno con un propósito diferente:

- La memoria de registros, que está situada en el propio chip, se corresponde con los elementos de memoria más pequeños dentro de la arquitectura. Su número está limitado por la capacidad de cálculo del modelo de GPU, y dado que los registros se asignan a cada *thread*, tienen un ciclo de vida igual al del mismo.
- La memoria local, al igual que la de registros, se asigna a cada *thread* y tienen su mismo ciclo de vida, sin embargo este tipo de memoria se encuentra físicamente en la memoria global del dispositivo, por lo que tiene un coste de acceso elevado.
- La memoria compartida, que está dentro del chip, es asignada a cada bloque, presentando un ciclo de vida igual al ciclo de un bloque. Siempre y cuando no hayan conflictos entre bancos (consultar la capacidad de cómputo de la GPU [84]), el coste de acceso a memoria compartida para todos los *theads* de un *warp* es equivalente al acceso a registro, pero en caso contrario el acceso se serializa. Esta memoria también se utiliza para transferir información entre los *threads* de un bloque, y así colaborar para realizar tareas intrabloque en un *kernel*.
- La memoria global (de dispositivo o GPU), que tiene una latencia mayor por situarse fuera del chip, es accesible por todos los *threads* de un *grid*, pero su vida útil está condicionada a las operaciones de reserva y liberación de memoria efectuadas desde el programa ejecutado en el *host*. El espacio de memoria global se subdivide en tres espacios bien diferenciados:
  - La memoria global propiamente dicha permite accesos tanto en modo escritura como en modo lectura. Su acceso es de alta latencia, pero esta condición ha mejorado con la introducción del manejo de ciertos niveles de caché a partir de la capacidad de cómputo 2.x. El patrón de su acceso determina la latencia asociada. Así, cuando se realiza una petición a memoria global, ésta es ejecutada como mínimo por un *warp*, aunque el número de transacciones necesarias para atender dicha petición depende de las restricciones de la capacidad de cómputo del dispositivo. Aquellas condiciones que evitan la serialización de estas operaciones se pueden consultar en [84], y aunque eran muy estrictas al principio, en las GPUs más modernas (capacidad de computo 2.x o superior) sólo se obliga a que todos los *threads* de un *warp* accedan a palabras de tamaño igual o menor a 4 bytes de un mismo segmento de memoria, aún cuando su acceso no sea consecutivo.
  - La memoria de texturas permite realizar escrituras por parte del *host* y sólo lecturas por parte del dispositivo, con la ventaja de disponer de caché *on-chip* por SM muy rápida. Este espacio de memoria está optimizado para aprovechar la localidad de datos definidos en 2D, mientras que para otros usos puede ser incluso más lenta que la memoria global del dispositivo.
  - La memoria de constantes permite realizar escrituras por parte del *host* y sólo lecturas por parte del dispositivo. Su tamaño está limitado a 64 KB por SM y dispone de una caché muy eficiente. Es ideal para su uso en aquellas situaciones en las que todos los *threads* de un *warp* acceden

a la misma dirección de memoria; en otro caso, los accesos se serializan. Su uso está indicado, por ejemplo, para almacenar coeficientes utilizados en repetidas ocasiones por una fórmula matemática.

- Aunque la memoria de *host* (CPU) no forma parte de la jerarquía de memoria CUDA, vale la pena comentar ciertos aspectos a cerca de su funcionamiento en este punto. La memoria de *host* no es accesible por los *threads* de CUDA, a no ser que (desde CUDA 4.0 y mejorado en CUDA 6.0) se establezca un espacio de direcciones virtual unificado (UVA) entre la CPU y la GPU, con las ventajas y restricciones que ello conlleva [84, 90]. Con respecto a su ciclo de vida, está supeditado a la acción de su reserva y liberación por parte del programador.

2.4.2.3 Modelo de ejecución CUDA

Cuando se lanza un *kernel* sobre una GPU, sus SMs ejecutan los bloques que forman el *grid* en tiempo compartido de modo que un bloque siempre se ejecuta en el mismo SM. Además, los bloques siempre se ejecutan en el mismo conjunto de procesadores escalares SPs, que pueden ser compartidos entre varios bloques. La ejecución de un bloque se descompone en *warps*, cuyos *threads* siempre ocupan los mismos SPs. En el caso de que todos los *threads* de un *warp* ejecuten una misma instrucción sobre diferentes datos se alcanzan las máximas prestaciones. En el caso contrario (instrucción condicional) la ejecución se serializa, con la consiguiente reducción de prestaciones. Las dimensiones de *grid* y de bloque, el tamaño de la memoria compartida y el número de registros utilizados, son factores que deben ser cuidadosamente escogidos, para maximizar la ocupación de los recursos de la GPU (*GPU Occupancy*)[86], y alcanzar el máximo rendimiento al ejecutar un *kernel*.

Los *kernels* son ejecutados por la GPU de forma asíncrona, por lo que existen órdenes en CUDA que, ejecutadas sobre el *host*, permiten sincronizar el trabajo de la GPU y del *host* en ese punto. Estos mecanismos, son muy útiles para efectuar medidas de tiempos de ejecución de los trabajos que ejecuta la GPU. Cuando el *host* lanza un *kernel* sobre la GPU, el controlador de dispositivo transfiere el código que define el *kernel*, mientras que el *host* queda a la espera de la respuesta de la GPU. Si el *host* está continuamente consultando al controlador de dispositivo para detectar cuando ha finalizado la GPU, se dice que se realiza una espera activa, o *polling*. En cambio, si es el controlador de dispositivo el que avisa al *host* que ha finalizado, se dice que este último realiza una espera pasiva, o *blocking*. Respecto a la sincronización dentro de la GPU, mencionar que existen mecanismos de sincronización explícitos dentro del ámbito de los *threads* que componen un bloque, y que no es necesario aplicarlo en el ámbito de los *threads* que componen un *warp*, dado que su ejecución es sincronizada. En cambio, no existe ninguna sincronización entre los bloques de un *kernel*, salvo a la finalización del kernel, en la que se incluye una sincronización implícita. Además, las transferencias de datos entre la CPU y la GPU son síncronas por defecto, a no ser que se implemente mediante *streams* un modelo de transferencias asíncronas, permitiendo solapar cálculo y comunicación. Para este propósito la memoria del *host* debe estar mapeada en el espacio de direcciones del dispositivo, utilizando *Page-Locked Host Memory* también conocido como *Pinned Memory*.

2.4.2.4 Interfaz de programación de CUDA

NVIDIA proporciona actualmente dos APIs para escribir programas CUDA: el interfaz de bajo nivel llamado CUDA *Driver* [87] y el de más alto nivel llamado CUDA *Runtime* [88] (ver Figura 2.15). El API de alto nivel maneja las utilidades definidas en el API de bajo nivel, ofreciendo un conjunto de extensiones del lenguaje *C*, conocidas como *C for CUDA*, así como funciones con prefijo `cuda*()`. Sus extensiones introducen una sintaxis nueva, que proporciona una manera más sencilla de especificar los parámetros de ejecución de un *kernel* (tamaños de *grid*, tamaños de bloque, etc.). Además permiten la definición de un *kernel* como una función habitual de *C*, proporcionando funcionalidades que facilitan el manejo por código

del dispositivo, que permite una inicialización implícita, además de manejo de contextos y de módulos. Por su parte, el API de bajo nivel contiene funciones con prefijo `cu*()`, ofreciendo una gama más amplia de funcionalidades siendo las más destacables, el que no depende de bibliotecas de tiempo de ejecución o *runtime library*, y que posee más control sobre los dispositivos, puesto que un *thread* de CPU puede controlar múltiples GPUs. Además no existen extensiones *C* en el código del *host*, lo que permite utilizar cualquier compilador disponible en el *host*, posibilitando el uso de *Parallel Thread eXecution (PTX) [89] Just-in-Time (JIT) compilation*. Por el contrario, no permite emulación en el dispositivo, y sus códigos son más complejos y detallados.

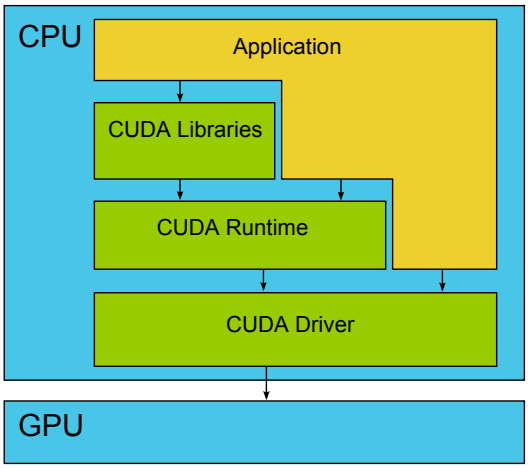


Figura 2.15: Pila de capas software de la arquitectura CUDA.

Seguidamente, se detalla un ejemplo de programa en CUDA para mostrar el funcionamiento básico del entorno de programación proporcionado por el *Runtime API* de CUDA [88]. Para ello, se describe una implementación básica de la operación AXPY, que aparece en numerosas ocasiones en los métodos iterativos sobre los que se han realizado todas las pruebas e investigaciones desarrolladas en esta Tesis Doctoral. La operación AXPY utiliza dos vectores x e y, y un escalar a, representándose como:

$$y := ax + y, \tag{2.15}$$

Antes de abordar la programación de su código, es necesario comentar que la memoria global de la GPU se puede abstraer como una zona lineal de almacén de bits, de forma que las matrices se almacenan una fila tras otra como un vector, y los vectores como una sola fila. Para definir un *grid*, antes se debe conocer la necesidad de elementos de cómputo necesarios para manejar los datos y cálculos, cuestión que depende exclusivamente del tamaño de los datos y del algoritmo que se vaya a implementar para manejarlos. En el caso de la implementación básica de AXPY es de un *thread* por cada elemento del vector, de tal forma que el número de *threads* del *grid* debe ser mayor o igual que el tamaño del vector. Una vez conocido el número de *threads* que componen el *grid*, se decide su organización lógica a través de la elección del tamaño de bloque y el número de bloques, que se puede realizar de varias maneras con respecto su dimensionalidad, siendo equivalentes a efectos de obtención de resultados correctos. La Figura 2.16a izquierda muestra la organización del *grid* en 2D con bloques 2D, cuyos índices, tanto de bloque como de *thread*, se numeran por columnas, estableciéndose a través de las fórmulas allí detalladas una relación lineal entre los *threads* de cada bloque y los elementos del vector organizado por filas como una matriz (Figura 2.16a derecha). Las fórmulas en cada caso calculan el desplazamiento en el espacio unidimensional, bidimensional (o tridimensional en

caso de utilizarse esta posibilidad). La Figuras 2.16b y 2.16c organizan los *grids* como (*grid*_bloque) 2D_1D y 1D_1D respectivamente.

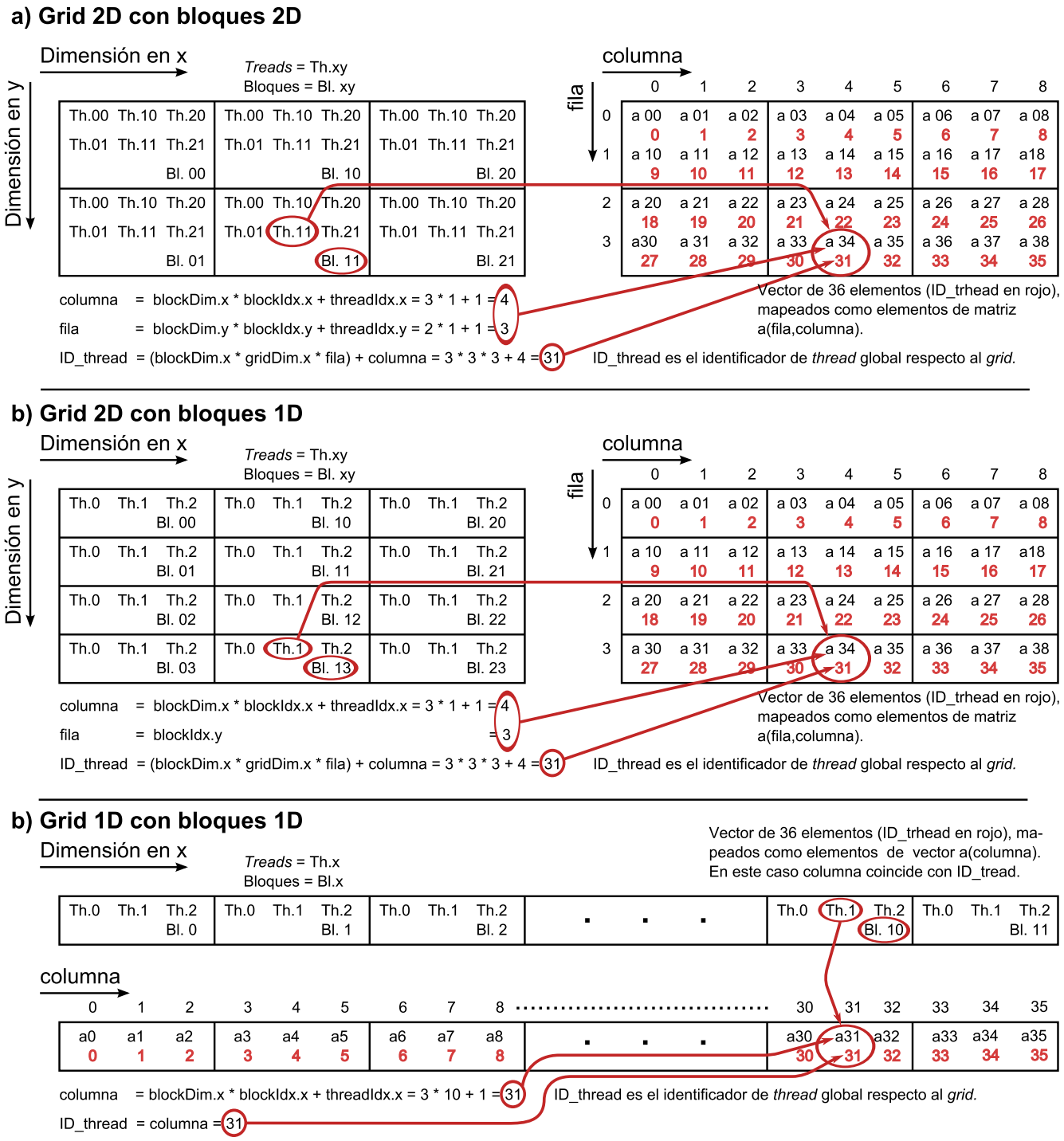


Figura 2.16: Tres *grids* equivalentes con distintas dimensiones que permiten mapear los *threads* que los definen sobre los elementos de un vector de tamaño igual a 36.

```
1  // AXPY definida con un grid 2D con bloques 2D
2  __global__ void axpy_2D_2D(float *a, float *x, float *y, unsigned int N){
3
4    unsigned int columna   =  blockDim.x * blockIdx.x + threadIdx.x;
5    unsigned int fila      =  blockDim.y * blockIdx.y + threadIdx.y;
6    unsigned int ID_thread = (blockDim.x * gridDmim.x * fila) + columna;
7
8    if (ID_thread < N)
9      y [ID_thread] = *a * x[ID_thread] + y[ID_thread];
10 }
11 // AXPY definida con un grid 2D con bloques 1D
12 __global__ void axpy_2D_1D(float *a, float *x, float *y, unsigned int N){
13
14   unsigned int columna   =  blockDim.x * blockIdx.x + threadIdx.x;
15   unsigned int fila      =  blockIdx.y;
16   unsigned int ID_thread = (blockDim.x * gridDmim.x * fila) + columna;
17
18   if (ID_thread < N)
19     y[ID_thread] = *a * x[ID_thread] + y[ID_thread];
20 }
21 // AXPY definida con un grid 1D con bloques 1D
22 __global__ void axpy_1D_1D(float *a, float *x, float *y, unsigned int N){
23
24   unsigned int ID_thread = blockDim.x * blockIdx.x + threadIdx.x;
25
26   if (ID_thread < N)
27     y[ID_thread] = *a * x[ID_thread] + y[ID_thread];
28 }
```

Figura 2.17: Implementación básica de una operación AXPY diseñada para tres definiciones de *grid* de distintas dimensiones.

Cuando se diseña un *kernel*, hay que estudiar qué configuración de *grid* es la más adecuada, ya que un *kernel* puede requerir un número mayor de bloques por dimensión que el permitido por la capacidad de cómputo del modelo de la GPU sobre la que se va a implementar. Para dar solución a este problema, se pueden configurar *grids* equivalentes de diferentes dimensiones (ver Figura 2.16/a/b/c), y analizar sus prestaciones, lo que lleva aparejado leves cambios en la implementación del algoritmo, en lo que se refiere al mapeado que sus *threads* realizan sobre los datos que manejan. En función de la dimensión del *grid*, se procede de forma distinta para calcular los índices de indexado a los vectores que maneja, a partir de variables específicas de CUDA (ver Figura 2.17).

Antes de ejecutar los *kernels*, se efectúa la reserva de memoria en el *host* y la inicialización de sus variables (ver Figura 2.18, líneas 7-13), y, posteriormente, se reserva memoria global para las variables que se alojarán en el dispositivo (ver Figura 2.18, líneas 18-21). En este momento ya se pueden transferir los datos desde el *host* a la GPU (ver Figura 2.18, líneas 23-26), y ejecutar los *kernels* que implementan las operaciones. Antes de realizar la llamada de cada *kernel*, se debe establecer la dimensión de su *grid*, ejecutando las correspondientes operaciones en el dispositivo (ver Figura 2.18, líneas 28-31). Tras finalizar, el resultado es devuelto desde la GPU al *host* (ver Figura 2.18, líneas 33-34), y finalmente se libera la memoria tanto en el *host* como en el dispositivo (ver Figura 2.18, líneas 36-40).

En la Figura 2.19 se muestra el acceso que realiza cada *thread* de una implementación básica del *kernel* AXPY sobre sus operandos almacenados en la memoria global de la GPU, al definir un *grid* 1D-1D, sobre vectores con un tamaño N = 1.000.000 elementos. Suponiendo un tamaño de bloque de 256 *threads*, el número de bloques necesario se calcula de la siguiente manera:

```
unsigned int dimB = 256;
```

```
1  // longitud de los vectores
2  #define N 36;
3
4  // variables en la CPU
5  float *a_h = NULL; float *x_h = NULL; float *y_h = NULL;
6
7  // reserva de memoria en la CPU
8  a_h = (float*)malloc(sizeof(float));
9  x_h = (float*)malloc(N * sizeof(float));
10 y_h = (float*)malloc(N * sizeof(float));
11
12 //inicializacion de las varaibles en la CPU
13 inicializacion(a_h, x_h, y_h, N);
14
15 // variables en la GPU
16 float *a_d = NULL; float *x_d = NULL; float *y_d = NULL;
17
18 // reserva de memoria la CPU
19 cudaMalloc(&a_d, sizeof(float));
20 cudaMalloc(&x_d, N * sizeof(float));
21 cudaMalloc(&y_d, N * sizeof(float));
22
23 // trasferencia de datos desde la CPU a la GPU
24 cudaMemcpy(a_d, a_h, sizeof(float), cudaMemcpyHostToDevice);
25 cudaMemcpy(x_d, x_h, N * sizeof(float), cudaMemcpyHostToDevice);
26 cudaMemcpy(y_d, y_h, N * sizeof(float), cudaMemcpyHostToDevice);
27
28 // lanzado de los kernels en la GPU
29 Dim3 G_a(3,2); B_a(3,2); axpy_D2_D2<<<G_a, B_a>>>(a_d, x_d, y_d, N);
30 Dim3 G_b(3,4); B_b(3);   axpy_D2_D1<<<G_b, B_b>>>(a_d, x_d, y_d, N);
31 Dim3 G_c(12);  B_c(3);   axpy_D1_D1<<<G_c, B_c>>>(a_d, x_d, y_d, N);
32
33 // trasferencia de datos desde el dispositivo al CPU
34 cudaMemcpy(y_d, y_h, N * sizeof(float), cudaMemcpyHostToDevice);
35
36 // desalojo de memoria en la CPU
37 free(a_h); free(x_h); free(y_h);
38
39 // desalojo de memoria de la GPU
40 cudaFree(a_d); cudaFree(x_d); cudaFree(y_d);
```

Figura 2.18: Código principal ejecutado en el *host*, en el que se realizan llamadas a tres versiones distintas de *kernel* de la operación AXPY, que se corresponden con tres *grids* de dimensiones 2D_2D, 2D_1D y 1D_1D.

```
unsigned int dimG = (unsigned int) ceil((float)N / dimB);
```

De esta forma la llamada al *kernel* queda definida como sigue:

```
axpy_D1_D1<<<dimG, dimB>>> (a_d, x_d, y_d, N) ;
```

Es importante destacar (ver la Figura 2.19) que el k-ésimo *thread* (numerados con relación al *grid*) toma la posición k-ésima del vector (x) y del vector (y), realiza la operación accediendo al escalar α, y deja el resultado en la misma posición del vector (y), de modo que los *threads* realizan un acceso coalescente sobre la memoria global.

Para finalizar este punto, resta señalar que, como parte del entorno de trabajo (*framenwork*) de CUDA, también se proporciona un manejador de compilación (*compiler driver*) nvcc [85], que es el encargado de dirigir la compilación CUDA (la cual, no está totalmente documentada). El compilador utiliza un conjunto

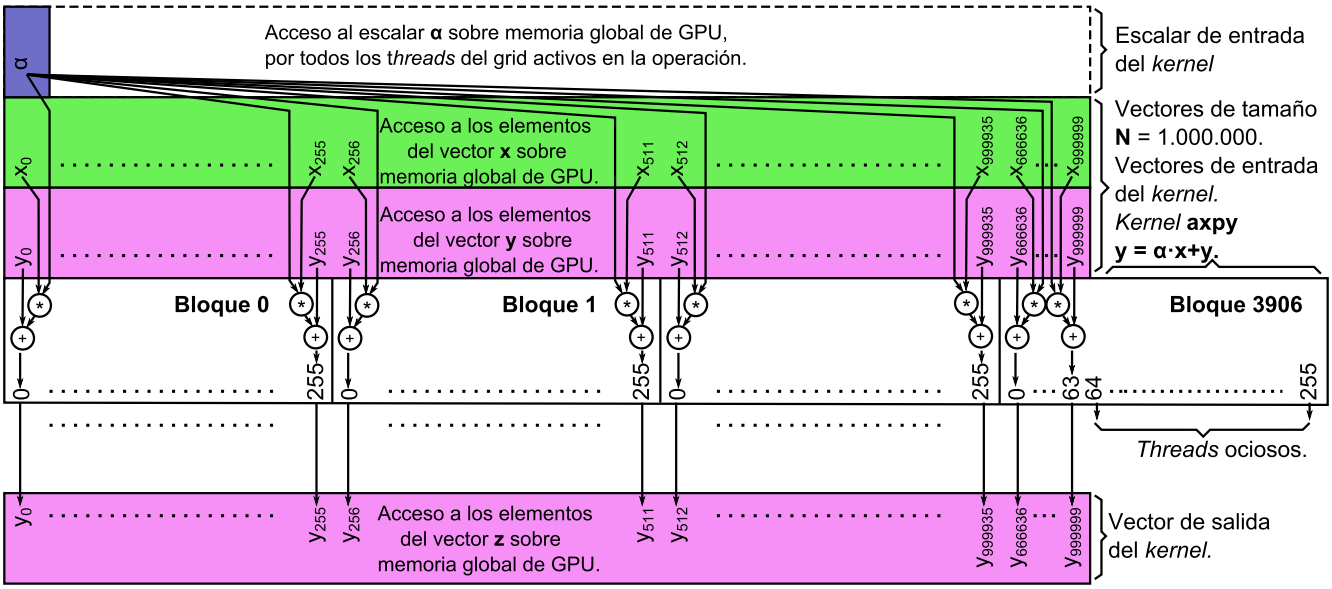


Figura 2.19: Acceso y operaciones aritméticas que el *kernel* AXPY realiza sobre sus operandos situados en la memoria global de la GPU, al definir un *grid* 1D-1D.

de utilidades, incluyendo los compiladores nativos de C/C++ instalados en el *host*, para separar los códigos de dispositivo y de *host* y compilarlos, por lo que cualquier archivo fuente que hace uso de las extensiones de *C* tiene que ser compilado con estas utilidades.

2.4.3 Arquitecturas de GPUs

Los dispositivos gráficos han evolucionado de modo vertiginoso en los últimos tiempos. Desde el nacimiento de CUDA a finales de 2006, y la 1ª generación Tesla chip G80, ya han visto la luz varias generaciones de arquitecturas de aceleradores gráficos dedicados a GPGPU: 2ª generación Tesla chip GT200 (2008), Fermi (2010), Kepler (2012) y Maxwell (2014), esperándose en el momento de la redacción de esta Tesis Doctoral la próxima aparición de Pascal, así como las siguientes generaciones: Volta, Einstein, etc.

Para identificar a los diferentes modelos de arquitectura, NVIDIA asigna un número de versión a cada generación de dispositivos. Este número, denominado *CUDA Compute Capability* o *CCC*, es utilizado por las aplicaciones en tiempo de ejecución para determinar qué características de hardware y/o instrucciones están disponibles en la GPU en uso, y de esta forma poder mejorar el rendimiento adaptando las aplicaciones a los recursos de cada modelo. El *CCC* está formado por dos números **(x.y)**, que representan lo siguiente:

- **(x)** es el número de versión principal y determina la generación: 1, 2, 3 para Tesla, Fermi y Kepler, respectivamente.
- **(y)** es el número de versión secundaria y representa la mejora incremental de la arquitectura de núcleo.

En la Tabla 2.1 se muestran las características más relevantes de las arquitecturas G80, GT200, Fermi y Kepler, con el objetivo de poder comparar su evolución. Seguidamente se detallan cada una de ellas, haciendo hincapié en las dos últimas a través de la descripción de sus características más relevantes, con el fin de optimizar su rendimiento mediante la descripción de pautas básicas en el diseño y programación de

kernels CUDA, como por ejemplo, la elección adecuada del tamaño y dimensión de *grid* y de bloque, tipos de acceso a memoria global y compartida, número de registros, etc.

	C870 Tesla-G80	C1060 Tesla-GT200	C2050 (Fermi)	Tesla K20 (Kepler)
Número de transistores	681 millones	1,4 billones	3,2 billones	7,1 billones
Área de chip, mm²	480	576	526	561
Tecnología de fabricación, nm	90	65	40	28
Consumo, W	170,9	187,8	247	225
Clock Shader, GHz	1,35	1,296	1,150	0,706
Memoria global, GB	1,5 GDDR3-bus 384 bits	1-4 GDDR3-bus 512 bits	3 GDDR5-bus 384 bits	5 GDDR5-bus 320 bits
Ancho de banda de la Memoria Global, GBPS	76.8	102.4	ECC-off 144	ECC-off 208
Transferencias CPU-GPU	PCIex16	PCIe 2.0x16	PCIe 2.0x16	PCIe 2.0x16
TCPs	8	10	-	-
SMs por TCP	2	3	-	-
GPC	-	-	4	5
SMX por GPC	-	-	4	3
SPs por SM o SMX	8	8	32	192
Multiprocesadores/GPU	16	30	14	13
Número total de SPs	128	240	448	2.496
Precisión simple, GFLOPS	500	933,120	1.030,4	1.175
Precisión doble, GFLOPS	No	77.76	515.2	3.524
Planificadores de *warps*	1	1	2	4
CUDA Compute capability	1.0	1.3	2.0	3.5
Hilos por *warp*	32	32	32	32
Bloques por Multiprocesador	8	8	8	16
Hilos por bloque	512	512	1.024	1.024
Hilos por Multiprocesador	768	1.024	1.536	2.048
Warps activos por Multi.	24	32	48	64
Registros de 32 bits por Mult.	8K	16K	32K	64K
Memoria compartida por Mult.	16 KB	16 KB	16 KB / 48 KB 48 KB / 16 KB	16 KB / 48 KB 32 KB / 32 KB 48 KB / 16 KB
Caché L1 por multiprocesador	No	No		
Caché L2 por GPU	No	No	768 KB	1.536 KB
Ancho del bus de direcciones	32	32	64	64
Correcciones de errores DRAM ECC	No	No	Si (configurable)	Si (configurable)
Paralelismo Dinámico	No	No	No	SI
Hyper-Q	No	No	No	SI

Tabla 2.1: Evolución de las características más relevantes de las GPUs desde G80 hasta Kepler.

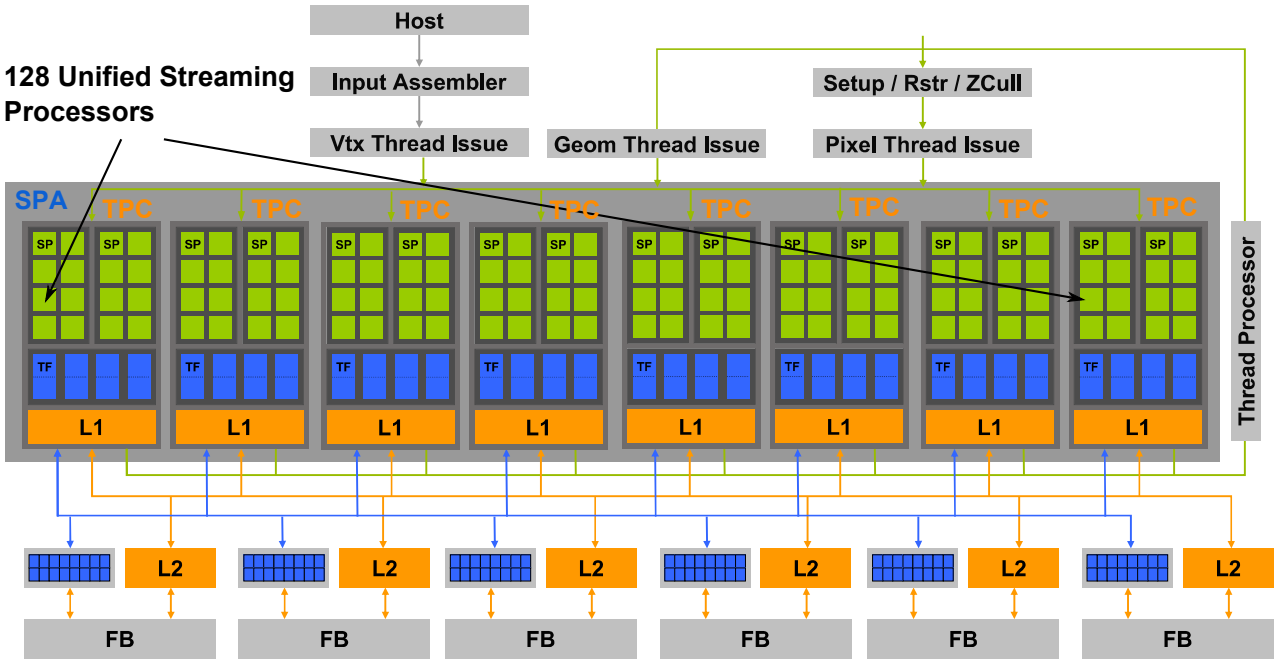


Figura 2.20: Arquitectura del chip GT80 [2] .

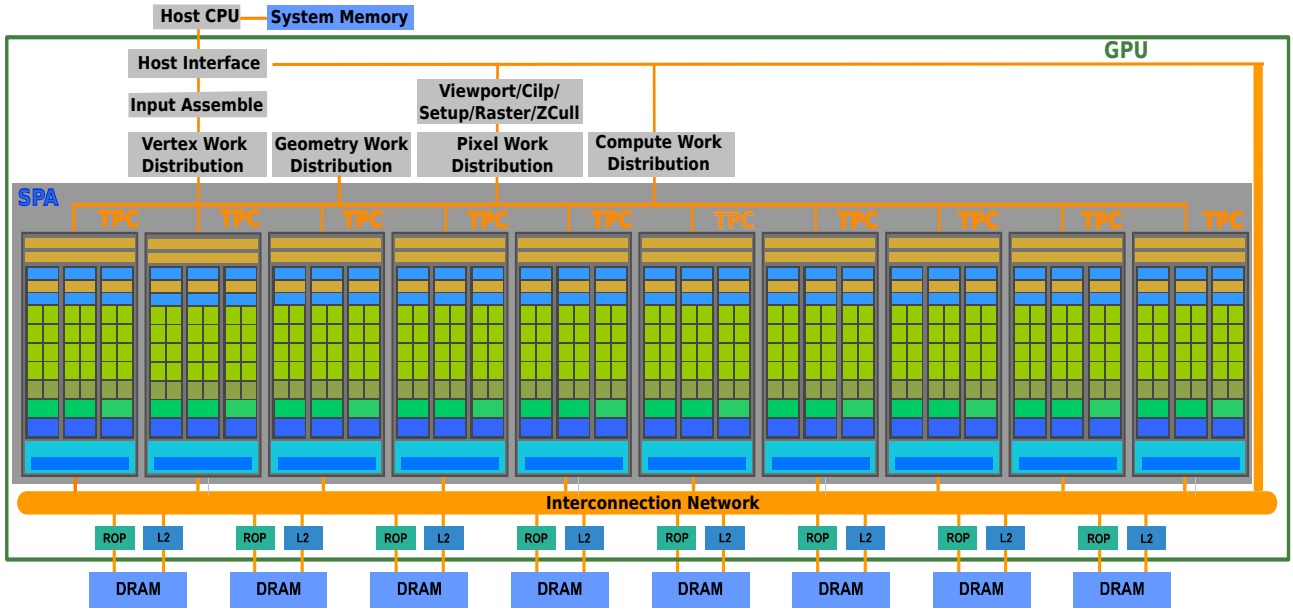


Figura 2.21: Arquitectura del chip GT200 [3] .

2.4.3.1 Antecedentes (arquitecturas Tesla G80 y GT200)

La primera arquitectura de NVIDIA que soporta CUDA se basa en el chip G80 (noviembre de 2006), y su implementación llevó a la práctica el concepto de arquitectura unificada para GPUs. Así, G80 presenta una arquitectura unificada a nivel de *shaders*, en donde no existe división a nivel de hardware entre procesadores

[2] Figura extraída del documento de NVIDIA [91].
[3] Figura extraída del documento de NVIDIA [91].

de vértices y procesadores de fragmentos, lo que permite su uso para GPGPU a través del cambio del modelo lineal de *pipeline* de sus antecesoras a un modelo retroalimentado o de lazo.

La arquitectura del chip G80 está compuesta por una primera estructura, llamada SPA (*Streaming Processor Array*) (ver Figura 2.20), que agrupa 8 unidades de procesamiento independientes llamados *Threads Processing Clusters* (TPC), cada uno de los cuales contiene 2 multiprocesadores de *streaming* (SMs), organizados como 8 procesadores de streaming o escalares (SPs) también llamados CUDA *cores*, que soportan emisión dual de operaciones MAD y MUL. Así, las GPUs G80 tienen un total de 128 SPs, 8K entradas a registros de 32 bits, y 16 KB de memoria compartida por SM.

Posteriormente, aparecen los procesadores GT200 (2008), cuya arquitectura no difiere en exceso de la de G80, ya que simplemente es una revisión en la que se mejoran diferentes aspectos, como la ampliación de algunos recursos y el área de chip. Así, la Figura 2.21 muestra los cambios estructurales, mientras que la Tabla 2.1 muestra las mejoras que presenta ésta evolución. Las mejoras más relevantes que se producen desde el chip G80 al GT200 se pueden resumir en los siguientes 5 puntos:

- **Incrementa la cantidad de SPs**. El número de TCPs se incrementa de 8 a 10. Además, cada TPC agrupa 3 SMs en lugar de los 2 que agrupa G80, con lo que suman un total de 30 SMs, que con 8 SP cada uno, supone un aumento desde 128 SPs en G80 hasta 240 SPs en GT200.
- **Incrementa la cantidad de *threads* activos**. G80 permite 24 *warps* activos por SM, mientras que GT200 permite 32 lo que representa una mejora desde 768 *threads* activos por SM a 1.024 *threads*.
- **Dobla el tamaño del fichero de registros**. G80 dispone de un fichero de registros de 8K entradas de 32 bits por SM, mientras que en GT200 se incrementa a 16K por SM, lo que permite ejecutar un mayor número de *threads* sobre el chip en un momento dado.
- **Añade soporte a operaciones de coma flotante de doble precisión**. En cada SM se añade una unidad para efectuar operaciones de doble precisión fp64, ajustandose al estándar IEEE 754-1985.
- **Mejora el acceso a la memoria**. Se añade hardware específico que mejora el acceso eficiente a la memoria (*memory access coalescing*).

2.4.3.2 Fermi (GF100)

Fermi se presenta en 2010, con un diseño radicalmente diferente con respecto de sus antecesoras, (ver la Figura 2.22). En esta nueva arquitectura el TCP de las generaciones Tesla desaparece, y la nueva GPU se descompone en 4 grandes bloques llamados GPC (*Graphics Processing Cluster*), lo que la hace una arquitectura más modular. Además, cada uno de los GPC está compuesto por 4 SMs, que a su vez contienen 32 SPs, sumando un total de 512 SPs. Los SMs (de tercera generación) se convierten en unidades mucho más complejas (ver la Figura 2.22 izquierda), con varias innovaciones que los hacen más eficientes. Al contrario de la estrategia de mejora intergeneracional que NVIDIA sigue en Tesla, en Fermi decide reducir el número de SMs y aumentar el número de SPs por SM. En consecuencia, Fermi dispone de tres tipos de unidades de procesamiento:

- **Unidades de enteros y de coma flotante**. Cada CUDA *core* o SP dispone de una unidad de enteros (ALU) y otra de coma flotante (FPU) ajustándose al estándar IEE 754-2008. El ALU de enteros soporta una precisión de 32/64 bits, incluyendo operaciones booleanas del tipo, move, convert, shift, compare, etc. Por su parte, la FPU suministra instrucciones fusionadas multiplicar-sumar (FMA), tanto para aritmética de simple precisión como de doble precisión, mejorando la precisión de las instrucciones (MAD) de GT200, y además, cada SM puede realizar 16 operaciones FMA por ciclo de reloj.

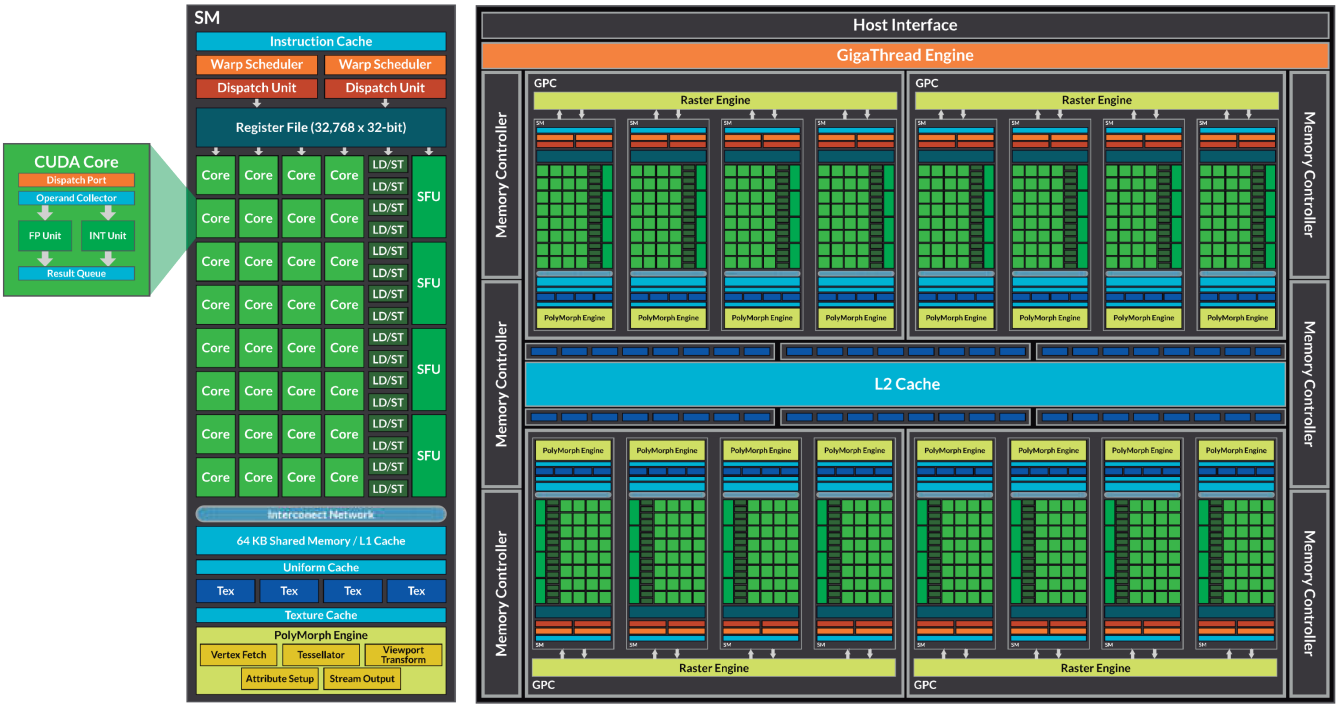


Figura 2.22: La arquitectura Fermi dispone de 16 SMs distribuidos en torno a una caché L2 (banda azul claro). Cada SM dispone de un ampliado fichero de registros (banda azul oscuro), además contiene dos planificadores de envío de instrucciones (banda naranja), con una unidad de emisión de instrucciones cada una de ellas (banda roja), varias unidades de ejecución (rectángulos en verde) y una unidad configurable de "memoria compartida / caché L1" (banda azul claro) [4] .

- **Unidades *Load/Store***. Dispone de 16 unidades *Load/Store* que permiten calcular direcciones de memoria origen/destino a 16 threads por golpe de reloj.
- **Unidades de funciones especiales (SFU)**. Dispone de cuatro unidades para el cálculo rápido de funciones complejas, como seno, coseno, etc. Cada SFU ejecuta una instrucción por *thread* y por ciclo de reloj, lo que permite que un *warp* ejecute una función de este tipo en 8 ciclos de reloj. Además el *pipeline* de las SFUs está desacoplado de la unidad de emisión de instrucciones (*Dispatch Unit*) por lo que se pueden emitir instrucciones a otras unidades mientras las SFUs están ocupadas.

Además, Fermi dispone de nuevas unidades de trabajo, que proporcionan mejoras de rendimiento muy significativas, tal y como se describen seguidamente:

- ***Dual Warp Scheduler***. En el *front-end* de cada SM, Fermi dispone de 2 planificadores de *warps* independientes con 1 unidad de emisión de instrucciones por cada uno, permitiendo que 2 *warps* se ejecuten a la vez. El planificador dual de *warps* proporciona una instrucción a cada grupo de 16 SP, 16 unidades de *Load/Store*, o 4 SFUs. Puesto que la doble emisión de *warps* hace que éstos se ejecuten de forma independiente, no es necesario que el planificador compruebe las dependencias entre cada flujo de instrucciones, lo que permite emitir pares de instrucciones del tipo: de enteros, de coma flotante, o

[4] Figura extraída del documento de NVIDIA [95].

mezclas con operaciones de enteros, coma flotante, *load*, *store* e instrucciones SFU. Únicamente las instrucciones de doble precisión no soportan una emisión dual con alguna otra operación.

- ***Fermi Memory Hierarchy***. Una de las principales mejoras respecto a la anterior generación es la renovada jerarquía de memoria. Cada SM en Fermi dispone de 64 KB de memoria en el chip que se puede configurar de dos modos diferentes: 16 KB de memoria compartida y 48 KB de caché L1 o viceversa. El primer modo ayuda a la optimización de algoritmos cuyos accesos a memoria no se conocen de antemano, mientras que el segundo modo funciona mejor para algoritmos con accesos a la memoria bien definidos. Por otra parte esta generación incorpora 768 KB de caché L2 común a todos los SMs del dispositivo.

- **Ejecución concurrente de *kernels***. Fermi permite la ejecución concurrente de *kernels*, donde diferentes *kernels* de una aplicación en un mismo contexto se pueden ejecutar en la GPU al mismo tiempo, lo que permite a los programas que ejecutan una serie de *kernels* pequeños, utilizar completamente los núcleos computacionales de toda la GPU. Sin embargo, los *kernels* que pertenecen a aplicaciones de distintos contextos, aunque se procesen secuencialmente, se pueden ejecutar con buena eficiencia gracias a un cambio de contexto muy mejorado.

- ***GigaThread^{TM} Thread Scheduler***. Esta unidad es la responsable de distribuir el trabajo, creando y suministrando bloques de *threads* a los SMs, y proporcionando soporte para la planificación y ejecución de *kernels* concurrentes dentro una misma aplicación y contexto.

Con respecto a las mejoras alcanzadas comparadas con GT200:

- Fermi dobla el número de registros por SM (32K entradas de 32 bits).
- Incrementa el número máximo de *threads* en ejecución por bloque a 1.024, y el número máximo de *threads* activos por SM a 1.536, lo que supone que puedan estar activos en la GPU un total de 24.576 *threads*.
- Mejora el rendimiento en doble precisión (multiplicándolo por 8).
- Incorpora mecanismos para la detección y corrección de errores de datos en memoria (ECC, *Error Correcting Code*).
- Da soporte al manejo de un espacio unificado de direcciones (*Unified Address Space)*.
- Incrementa la rapidez de los cambios de contexto (10x más rápidas).
- Incrementa la velocidad de las operaciones atómicas (20x más rápidas), gracias a unidades hardware específicas en combinación con la introducción en la jerarquía de memoria de un segundo nivel de caché L2.

2.4.3.2.1 Optimización en Fermi

Para poder optimizar *kernels* CUDA, es necesario conocer de forma exhaustiva la arquitectura de la GPU de NVIDIA sobre las que se ejecutan. Con este objetivo, la optimización de *kernels* se puede dividir en 5 aspectos bien diferenciados, y que serán tratados seguidamente:

- **Conflictos en el acceso a memoria global**. El efecto *Partition Camping* [92, 93] se produce cuando *threads* concurrentes de un SM solicitan segmentos de memoria diferentes pertenecientes a un mismo

banco de memoria global, provocando una serialización de las transacciones. En Fermi, la memoria global está distribuida en 6 bancos con sus correspondientes controladores independientes, y el problema de *Partition Camping* que ocurre cuando se solicitan en repetidas ocasiones los mismos segmentos de transacción, fue reducido con la introducción en su jerarquía de memoria de dos niveles de caché L1/L2, en la que se permite tanto desactivar la caché L1 como ampliar su tamaño a costa del tamaño de la memoria compartida. En esta arquitectura, el tamaño de la caché L2 es de 768 KB, mientras que el tamaño conjunto de la caché L1 y de la memoria compartida es de 64 KB, pudiendo admitir su reparto con las configuraciones (16-48) KB o (48-16) KB. Por otra parte, el tamaño de los segmentos de las transacciones de la memoria global puede ser de 128 o de 32 Bytes, siendo el primer caso válido únicamente si la caché L1 está activa (configuración establecida por defecto).

- **Ocupación de los recursos de la GPU**. Incrementar el rendimiento en una GPU es un factor que está directamente relacionado con maximizar la ocupación (unidades de computo utilizadas). Existen tres factores a saber que afectan a esta cuestión, el tamaño de bloque, el número de registros utilizados, y el uso de memoria compartida:

  – En el primer caso, se debe intentar tener activos la máxima cantidad de *warps* por SM que permita la arquitectura (en el caso de Femi, de 48). Una ocupación óptima ayuda a ocultar latencias en la GPU, como por ejemplo, cuando se realizan accesos a la memoria global para leer aquellos datos que requiera alguna instrucción para que pueda continuar su ejecución. Con el fin de conseguir una ocupación máxima, se debe estudiar tanto la dimensión como la geometría de bloque y de *grid*. Teniendo en cuenta que Fermi soporta 1.024 *threads* por bloque, y 1.536 *threads* activos por SM, con un máximo de 8 bloques ejecutándose de forma concurrente, el número de *threads* por bloque que maximiza la ocupación del SM debe ser igual o inferior a 1.024 y divisor entero de 1.536. Además se debe cumplir que 1.536 dividido por el tamaño de bloque seleccionado no puede ser superior a 8 (máximo número de bloques concurrentes por SM). Los tamaños de bloque que cumplen con estas premisas son: 192, 256, 384, 512 y 768. En el caso de que haya *kernels* que requieran un número de *threads* inferior a (192 $\times$ el número de SMs de la GPU), puede ser conveniente escoger tamaños de bloque inferiores, y así permitir a la arquitectura distribuir mejor la carga de trabajo por todos los SMs disponibles [94] .

  – En el segundo caso, los registros permiten mantener variables locales a cada *thread* con muy baja latencia, pero en Fermi su número por SM está limitado a 32.768 registros. Puesto que en un SM no pueden haber más de 1.536 *threads* activos, cada *thread* está limitado a utilizar 21 registros como máximo.

  – La memoria compartida también es un recurso escaso (por defecto en Fermi es de 48 KB por SM). Cuando se ejecuta un kernel, esta memoria es compartida por los bloques activos que puedan haber en cada SM, por lo que también es un factor determinante en el grado de su ocupación.

- **Acceso coalescente a los datos**. La coalescencia es una técnica de diseño algorítmico que permite disminuir el número de transacciones cuando un *warp* realiza una petición a la memoria global del dispositivo. Esta técnica es especialmente importante aplicarla en códigos que tienen un elevado ratio de operaciones de acceso a la memoria global por operación aritmética. La razón es el alto coste de los accesos a la memoria global, por lo que hay que evitar que un *warp* tenga que realizar varias transacciones a memoria para acceder a sus datos (serialización de transacciones). En Fermi, por defecto, las transacciones son de segmentos de 128 bytes, lo que implica que, si los *threads* de un *warp* acceden a elementos de más de 4 bytes, se generen varias peticiones. Si cada *thread* accede a elementos de 4 bytes, y el *thread* cero está alineado con segmentos de 128 bytes, la operación se resuelve en una sola

petición (en Fermi, incluso con permutaciones en el orden de acceso a los datos dentro del mismo segmento). Una forma de asegurar que los accesos a la memoria global se resuelvan mediante una petición (al margen del efecto de *partition Camping*), es utilizar la coalescencia, que permite que los *threads* consecutivos de un *warp* accedan a direcciones consecutivas, cuando soliciten segmentos de memoria global, evitando accesos serializados, que conllevan una alta penalización en el rendimiento. Una manera de facilitar la implementación de la coalescencia es buscar estructuras de datos alineadas con el número de *threads* de un *warp*, y con el tamaño de los segmentos de transacción. Además, para reducir la probabilidad de aparición del efecto de *Partition Camping*, es conveniente usar estructuras de datos que estén alineadas con el número de bancos o controladores de memoria global del dispositivo. Una buena elección que facilita esta técnica, consiste en escoger *arrays* multidimensionales de (enteros o reales) en los que su última dimensión sea múltiplo del tamaño de *warp* y del número de bancos. En algoritmos que implementen operaciones con un ratio de una o más operaciones de acceso a la memoria por cada operación aritmética, la elección de bloques con 192 *threads* maximiza la ocupación, puesto que con 8 bloques activos por SM, se consiguen ejecutar el máximo número permitido de *threads* activos por SM (1.536 *threads*). Con esta elección también se consigue ejecutar el máximo número de *warps* que pueden estar activos en un SM (48 *warps* en Fermi), posibilitando más oportunidades para ocultar latencias en la GPU. Sin embargo este tamaño no es óptimo en todos los casos, y a veces es necesario ajustarlo experimentalmente. Por ejemplo, en la operación de multiplicación de matrices, aún con accesos coalescentes, el ratio del número de accesos/operaciones se inclina hacia una mayor carga computacional por *thread*, lo que implica reutilizar segmentos ya solicitados. En estos casos aumentar el tamaño de bloque hasta 768 *threads* supone mayores oportunidades de reutilización de datos y por lo tanto una disminución del número de segmentos de transacción solicitados.

La velocidad de acceso a la memoria compartida es similar al acceso a un registro, que de media es 100 veces más rápido que el acceso a la memoria global, por lo que es conveniente que la memoria compartida se reutilice tanto como se pueda, diseñando para ello algoritmos que cumplan esta premisa. La memoria compartida se encuentra en el multiprocesador y, en Fermi, está dividida en 32 bancos con tamaños homogéneos, que pueden ser accedidos simultáneamente si no existen conflictos de acceso. Los bancos están organizados de forma que palabras consecutivas de 32 bits se asignen a bancos consecutivos, y tienen un ancho de banda de 32 bits cada 2 ciclos de reloj. Cuando se realizan al mismo tiempo dos accesos a un mismo banco por *threads* distintos de un mismo *warp*, se produce un conflicto que se resuelve mediante dos accesos serializados (capacidad de computo 2.x), produciéndose un decremento del máximo ancho de banda posible. Es por ello que resulta importante realizar diseños algorítmicos que eviten al máximo estos conflictos, siendo de especial interés la aplicación de diseños algorítmicos que implementen coalescencia.

- **Accesos no coalescentes**. Ciertos algoritmos, por su naturaleza, no pueden evitar realizar accesos a memoria global no coalescentes, motivo por el que se ve incrementado tanto el número de segmentos de transacción solicitados por un *warp*, como la probabilidad de que se produzca *partition Camping*. En Fermi, este hecho se puede aliviar de dos formas:

  - La primera opción es desactivar la caché L1, obligando a que los tamaños de los segmentos de transacción sean de 32 KB (normalmente de 128 KB). Al ser más pequeños, cuando predominen los accesos no coalescentes, la probabilidad de finalización con un número menor de peticiones a memoria global será mayor.
  - La segunda posibilidad es reducir el tamaño de bloque, lo que, posibilita que en un mismo periodo de tiempo se produzca una reducción del número de segmentos solicitados, aunque también se reduzca la ocupación del SM.

- **Geometría de bloque**. La geometría de bloque también es un factor a tener en cuenta, puesto que, aunque existen configuraciones geométricas de *grid* equivalentes, con el mismo ratio de ocupación (ver Figura 2.17), éstas pueden tener un impacto distinto sobre la coalescencia, los conflictos en los bancos de memoria, y provocar cuellos de botella en los accesos a la memoria global, produciendo con ello accesos serializados y una pérdida constatable de rendimiento. Para diseños de algoritmos que se ajusten al aprovechamiento de la técnica de coalescencia, es interesante señalar que los mejores resultados se obtienen con bloques de *threads* con un número de columnas múltiplo del tamaño de *warp*. De esta forma, las peticiones a memoria global estarán alineadas con segmentos de transacción completos.

Figura 2.23: Diagrama completo de bloques del chip de Kepler GK110 [5] .

2.4.3.3 Kepler GK110

Kepler se presentó en 2012, como una revisión muy mejorada de Fermi, en donde NVIDIA puso el foco sobre todo en mejorar la eficiencia energética; la Tabla 2.1 muestra las principales características del modelo K20 comparadas con sus antecesoras. Siguiendo la tendencia introducida por Fermi con respecto GT200, Kepler también aumentó el número de CUDA *cores* por SM y redujo la cantidad de SMs, que en Kepler se renombran como SMX. Además, se conservó la modularidad de Fermi, ya que siguió agrupando los SMX en GPCs, facilitando con ello el diseño y el cambio de prestaciones de sus diferentes modelos. Para ello, el número de CUDA *cores* por SMX es el mismo en las diferentes mejoras que ha tenido la arquitectura, aunque cambiando el número de GPCs y el número de SMX que éste agrupa. A pesar de que GK110 no fue el primer chip que implementó la arquitectura Kepler, este apartado se centra en él, puesto que las GPUs que lo implementan son las más utilizadas. Así, la serie GK110 de NVIDIA (ver Figura 2.23) está estructurada con 6 controladores de memoria global y 5 GPCs, con agrupaciones de 3 SMX por GPC, aunque según

[5] Figura extraída del documento de NVIDIA [96].

	GK110 (k20)	GK110 (k20x)	GK110 (k40x)
C.C.C.	3.5	3.5	3.5
Año de lanzamiento	2012-13	2013	2013-14
SMX	13	14	15
Int *and* fp32 [*cores*/SM]	2496 [192]	2688 [192]	2880 [192]
Fp64 [*cores*/SM]	832 [64]	896 [64]	960 [64]
LSU [*cores*/SM]	416 [32]	448 [32]	480 [32]
SFU [*cores*/SM]	416 [32]	448 [32]	480 [32]
PCI *Express system interface*	PCIe2.0 ×16	PCIe2.0 ×16	PCIe3.0 ×16

Tabla 2.2: Kepler GK110.

modelos, algún GPC puede desactivar algún SMX, de manera que, en total, los modelos k20, k20x y k40 disponen de 13, 14 y 15 SMX, respectivamente (ver la Tabla 2.2). Además Kepler añade nuevas características relacionadas con la capacidad de cálculo, ya que es capaz de proporcionar un rendimiento en doble precisión por encima de 1 TFLOP (ver Tabla 2.1), ajustándose al formato IEEE-754 de 64 bits, mostrando una eficiencia en la ejecución DGEMM mayor al 80 % contra un 60 - 65 % que proporciona Fermi .

Seguidamente se muestran algunas mejoras de la arquitectura Kepler en referencia a su antecesora Fermi, destacando las siguientes diferencias:

- **Nuevos SMX**. Cada uno (ver la Figura 2.24) está formado por las siguientes unidades funcionales:
  - **CUDA *cores***. Kepler dispone de 192 núcleos para aritmética entera y de simple precisión, frente a los 32 de Fermi, lo que significa que, con 13, 14 y 15 SMX en los modelos K20, K20c y K40 respectivamente, dispone de 2.496, 2.688 y 2.880 *cores* frente a los 448 y 512 que tiene Fermi en los chips GF100 y GF110 respectivamente. Además, proporcionan mayores tasas de rendimiento y eficiencia energética, gracias a su innovador diseño, que dedica más espacio a los núcleos de procesamiento que a la lógica de control con una mejora de 3x rendimiento/Vatio.
  - **Unida de Doble precisión (DP)**. Dispone de 64 unidades dedicadas a tal fin.
  - **Unidades de funciones especiales (SFU)**. Dispone de 32 frente a las 4 de Fermi.
  - **Unidades *Load/Store* (LD/ST)**. Dispone de 32 frente a las 16 de Fermi.
  - ***Quadruple Warp Scheduler***. Dispone de un planificador cuádruple de *warps*, con 2 unidades de emisión de instrucciones cada uno, lo que suman un total de 8 unidades de emisión de instrucciones. El planificador cuádruple de Kepler selecciona 4 *warps*, y dos instrucciones independientes por *warp*, pudiendo ser emitidas 8 instrucciones por ciclo, frente a las 2 instrucciones por ciclo que pueden ser emitidas en Fermi. Además, al contrario que Fermi, Kepler permite emparejar instrucciones de doble precisión.
  - ***New ISA Encoding*: 255 *Registers per Thread***. En GK110 se ha cuadruplicado el número de registros a los que cada *thread* puede acceder.

  - ***Shuffle In instruction***. Para mejorar aún más el rendimiento, Kepler implementa este tipo especial de instrucción, que permite a los *threads* de un *warp* en un solo paso compartir datos sin utilizar la memoria compartida. Así, este tipo de instrucciones reducen la cantidad de memoria compartida de bloque. En el caso de FFT, el rendimiento se eleva en un 6 %.
  - ***Atomic Operation***. Las operaciones atómicas son muy importantes en programación paralela, ya que permiten que *threads* concurrentes realicen lecturas/escrituras sobre estructuras de datos compartidos de forma correcta. El rendimiento de una operación atómica sobre direcciones de memoria global es mejorado en un factor 9x para una operación por pulso de reloj. Además expande el soporte a operaciones atómicas de 64 bits sobre memoria global, que se suman a las operaciones que ya soportaba Fermi (atomicAdd, atomicCAS, y atomicExch), y añade: atomicMin, atomicMax, atomicAnd, atomicOr y atomicXor.
  - **Unidad de Texturas**. El rendimiento de la unidad de texturas en Kepler se ha incrementado significativamente en comparación con Fermi, ya que cada unidad SMX contiene 16 unidades de filtrado de texturas, con un incremento de rendimiento de 4x respecto al SM de Fermi GF110. Además se ha mejorado su manejo, pudiendo gestionar el estado de texturas en tiempo de ejecución, a diferencia de Fermi en la que se gestionaba antes de lanzar un *grid* mediante la asignación de *slots* de tamaño fijo.

- **Organización de la jerarquía del subsistema de memoria ampliada**. En la Figura 2.25a/b, se muestra la organización de la jerarquía del subsistema de memoria de Fermi y de Kepler respectivamente, en la que se apreciar como la implementada por Kepler es muy similar a la de Fermi, pero con algunas mejoras que se detallan seguidamente:

  - **L1 / memoria compartida.** La configuración de la memoria compartida en los SMX se hace más flexible, añadiendo a las que ya tiene de Fermi, la posibilidad de repartir los 64 KB en partes iguales entre L1 y la memoria compartida.
  - **48 KB *Read-Only Data Cache***. Además de la caché L1, Kepler añade 48 KB de caché para datos de solo lectura. En Fermi solo era posible acceder a ella mapeando datos como textura. En Kepler es accesible como operaciones de solo lectura sobre datos generales.
  - **Mejora de L2**. Se dobla el tamaño de la caché de segundo nivel L2 hasta 1.536 KB.
  - **ECC (*Error Correcting Code*)**. Fermi ya incorporó la tecnología ECC, asegurando la fortaleza de datos sensibles (como en aplicaciones médicas), aunque su uso reducía sensiblemente el rendimiento del sistema. Kepler mejora este problema en un 66 %.

A partir del modelo K20, la arquitectura Kepler incorpora nuevas características que permiten explotar más posibilidades de paralelismo, y que se describen seguidamente :

- **Paralelismo Dinámico**. En un sistema CPU-GPU, la eficiencia en la ejecución de aplicaciones con un elevado nivel de paralelismo depende del aprovechamiento de los mecanismos que ambas plataformas ofrezcan. Hasta Fermi, la GPU era utilizada básicamente como un potente coprocesador sin autonomía, mientras que la CPU era la encargada de controlar el flujo de ejecuciones en la GPU. La aparición del paralelismo dinámico (ver Figura 2.25d) ha supuesto un gran avance en lo que respecta a la independencia de la GPU como mero coprocesador de la CPU. La CPU siempre ha sido la encargada de lanzar un *kernel* para ser ejecutado en la GPU, pero a partir de Kepler G110, cualquier *kernel* puede tambien lanzar otro *kernel* con una profundidad de hasta 24 generaciones, pudiendo crear, los streams y eventos necesarios, y gestionar las dependencias requeridas para procesar tareas adicionales sin la intervención de la CPU. Así, pueden implementar más variedad de algoritmos paralelos, incluyendo

anidamiento de bucles e implementación de algoritmos recursivos. Con el paralelismo dinámico, se puede hacer que un mayor número de partes de una aplicación se ejecuten íntegramente en la GPU, mientras la CPU puede realizar otras tareas, o bien, puede utilizarse una CPU menos potente para ejecutar la misma carga de trabajo.

- ***Grid Management Unit* (GMU)**. Para habilitar el Paralelismo Dinámico, se requiere un gestor de *grid* y sistema de control de emisión avanzado y flexible (ver Figura 2.25c), en donde la GMU es capaz de pausar la emisión de nuevos *grids*, encolar pendientes y suspender *grids* hasta que estén listos para ser ejecutados, asegurando que los flujos de trabajo en la CPU y GPU son apropiadamente manejados y emitidos.

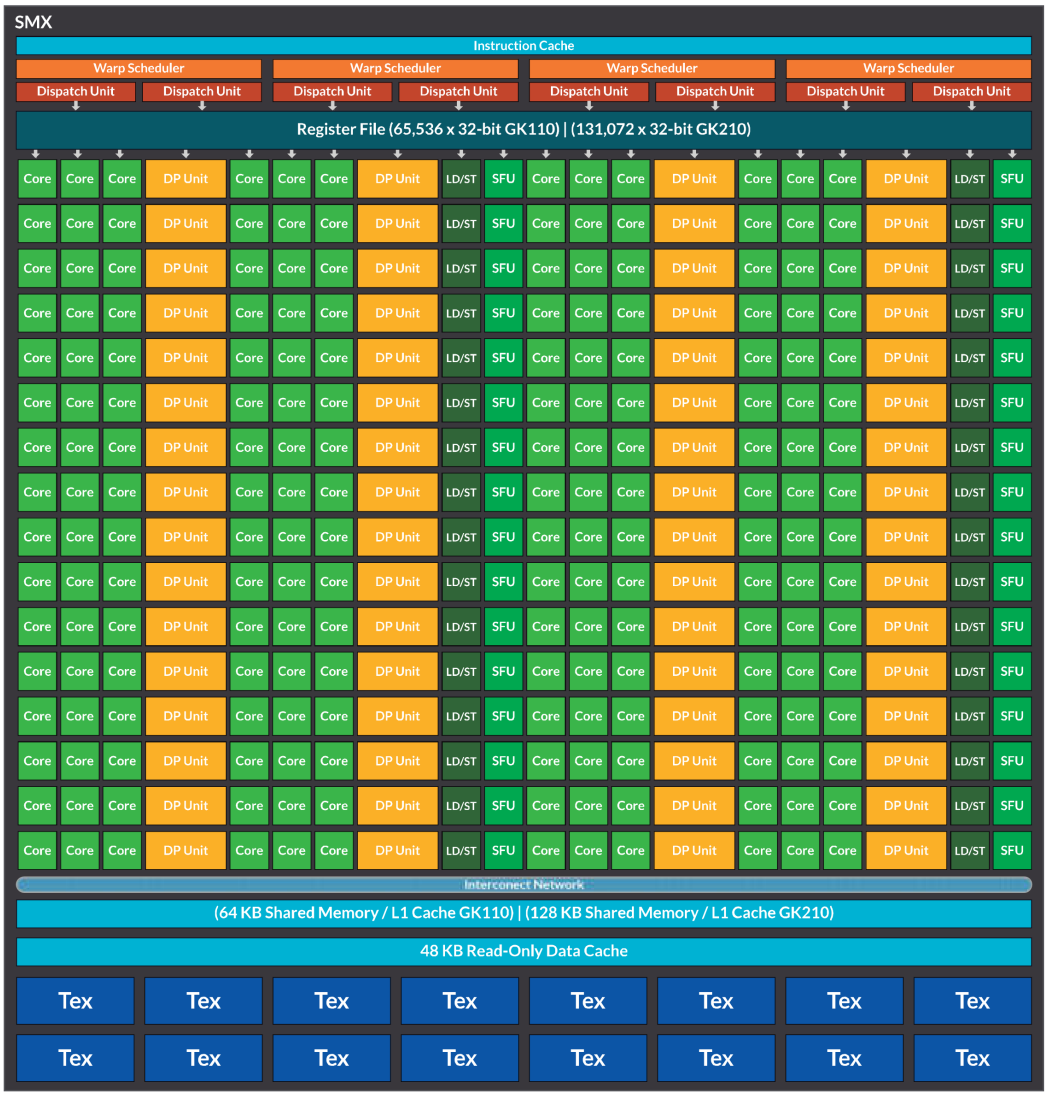


Figura 2.24: SMX de Kepler GK110/GK210. Básicamente dispone de 192 CUDA *cores* para simple precisión, 64 unidades de doble precisión, 32 unidades de funciones especiales (SFU), y 32 unidades (LD/ST) [6] .

[6]Figura extraída del documento de NVIDIA [96] .

- **Hyper-Q**. Fermi permite ejecutar hasta 16 *kernels* concurrentes lanzados desde sus respectivos *streams* dentro de un mismo contexto CUDA, encolándolos dentro de una única cola de trabajo (ver Figura 2.25e). Así, cuando dentro de un *stream* se incluye más de un kernel, aún sin dependencias entre ellos, únicamente el último kernel puede solaparse con los *kernels* de otro *stream* diferente. Este modelo de ejecución causa un efecto de falsas dependencias *intra-stream*, que puede ser aliviado en parte mediante una cuidadosa reorganización de los *kernels*. En Kepler, con la introducción de Hyper-Q, se disponen de hasta 32 colas de trabajo, por lo que los diferentes *stream* pueden ser efectivamente ejecutados en paralelo. Además, Hyper-Q ofrece beneficios significativos para los cálculos paralelos basados en el uso de la interfaz de paso de mensajes (MPI, *Message Passing Interface*) (ver Figura 2.25f), permitiendo hasta 32 flujos simultáneos de gestión de conexiones hardware (tareas MPI) en comparación con un único flujo de tareas permitido por MPI con el esquema de planificación de Fermi. Así, las aplicaciones que con el modelo de Fermi estaban sometidas al efecto de falsas serializaciones a través de tareas, ahora pueden acelerar su ejecución por 32x sin realizar ningún cambio en su código. Simplemente Kepler lo consigue gestionando cada *stream* a través de una cola diferente.

- **NVIDIA GPUDirect™**. Es una nueva capacidad (a partir de CUDA 5.0) que habilita a las GPUs en un único computador, o a las GPUs en diferentes servidores a través de una red, intercambiar directamente datos sin la necesidad de que intervenga para ello la memoria de sistema de la CPU (ver Figura 2.25g). En concreto, GPUDirect v1.0 habilita a los controladores de dispositivos de otras marcas, como los adaptadores de InfiniBand, para establecer una comunicación directa con el controlador de CUDA, siendo innecesario que los datos pasen a través de la CPU. Por otro lado, GPUDirect v2.0 habilita la comunicación de igual a igual (P2P) entre GPUs en un mismo sistema de cómputo, lo que implica también que la CPU quede liberada de la tarea de transferencias de datos. La aplicación de esta nueva capacidad posibilita en algunos casos un 30 % de ahorro en tiempo de comunicación.

2.4.3.3.1 Optimización en Kepler

Kepler es una evolución de Fermi, con más recursos y con muchas características nuevas, pero en lo básico, lo descrito para la optimización de Fermi en el Apartado 2.4.3.2.1 es válido para Kepler, con trabajos que así lo avalan [97], aunque, con algunos matices. Así, la elección del tamaño de bloque y su forma, es una de las decisiones más importantes que el programador debe tomar cuando codifica en CUDA un algoritmo paralelo. La razón es que el tamaño de *grid*, el tamaño de bloque y su geometría, tienen un fuerte impacto en la utilización de los recursos de cómputo de la GPU, y por ende, en rendimiento.

- En Fermi el tamaño de bloque ideal para realizar reducciones (ver Apartado 2.4.3.2.1) es de 192 *threads* por bloque. En Kepler, este dato coincide aunque por otra razón. En concreto el motivo es que cada SMX dispone de 192 CUDA *cores*.

- La gestión del subsistema de memoria que puede realizar el programador es esencialmente la misma, aunque extendiendo el reparto de 64 KB, para situaciones intermedias a las expuestas en el Apartado 2.4.3.2.1.

- Por lo que resta, en la optimización, se tendrán en cuenta las nuevas características que incorpora Kepler, y que ya han sido detalladas.

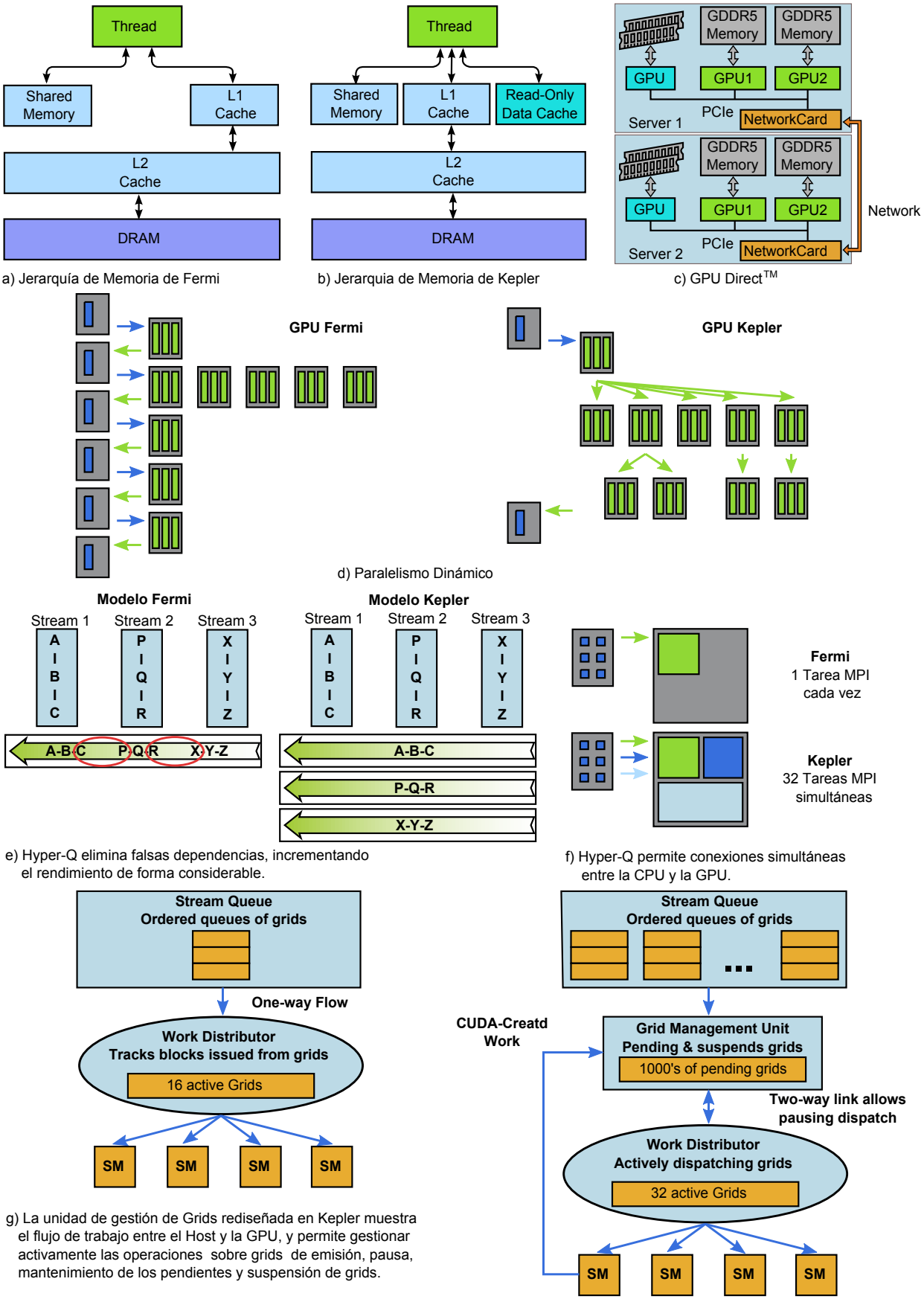


Figura 2.25: Resumen de las mejoras introducidas en Kepler GK110 comparadas con Fermi.

Bibliografía

[31] Owe Axelsson. *Iterative Solution Methods*. Cambridge University Press, New York, NY, USA, 1994.

[32] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[33] Achi Brandt. Multi-Level Adaptive Solutions to Boundary-Value Problems. *Mathematics of Computation*, 31(138):333–390, April 1977.

[34] C. Brezinski. Projection methods for linear systems, 1996.

[35] W. Briggs, V. Henson, and S. McCormick. *A Multigrid Tutorial, Second Edition*. Society for Industrial and Applied Mathematics, second edition, 2000.

[36] T. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006.

[37] Iain S Duff, Albert M Erisman, and John K Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Inc., New York, NY, USA, 1986.

[38] R. Fletcher. Conjugate gradient methods for indefinite systems. In G. Watson, editor, *Numerical Analysis*, volume 506 of *Lecture Notes in Mathematics*, chapter 7, pages 73–89. Springer Berlin / Heidelberg, 1976.

[39] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[40] Martin H. Gutknecht. Variants of BICGSTAB for matrices with complex spectrum. *SIAM J. Sci. Comput.*, 14(5):1020–1033, September 1993.

[41] W. Hackbusch. *Multi-Grid Methods and Applications*. Springer, 2003.

[42] Magnus R. Hestenes and Eduard Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, December 1952.

[43] Gerhard Joubert, Wolfgang Nagel, Frans Peters, and Wolfgang Walter. *Parallel Computing: Software Technology, Algorithms, Architectures & Applications: Proceedings of the International Conference ParCo2003, Dresden, Germany*, volume 13. Elsevier, 2004.

[44] C Lanczos. Solution of systems of linear equations by minimized iterations. *Journal of Research of the National Bureau of Standards*, 49(1):33—-53, July 1952.

[45] D.C. Lay. *Linear Algebra and Its Applications*. Pearson Education, 2002.

[46] Tarek Mathew. *Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations (Lecture Notes in Computational Science and Engineering)*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[47] Clever Moler. *Numerical Computing with MATLAB*. The mathworks edition, 2004.

[48] Youcef Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2, 1994.

[49] Youcef Saad and Martin H Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, July 1986.

[50] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[51] Fokkema Diederik R. Sleijpen Gerard L.G. BiCGstab(*l*) for linear equations involving unsymmetric matrices with complex spectrum. *ETNA. Electronic Transactions on Numerical Analysis [electronic only]*, 1:11–32, 1993.

[52] Barry F. Smith, Petter E. Bjørstad, and William D. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, New York, NY, USA, 1996.

[53] Gilbert Strang. *Introduction to Linear Algebra*. Wesley-Cambridge Press, 3rd edition, 2003.

[54] H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, March 1992.

[55] F. Vázquez, G. Ortega, J. J. Fernández, and E. M. Garzón. Improving the performance of the sparse matrix vector product with GPUs. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 1146–1151, Washington, DC, USA, 2010. IEEE Computer Society.

[56] Patterson, David A. and Hennessy, John L. Computer Organization and Design. Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design). In *Morgan Kaufmann Publishers Inc.*, San Francisco, CA, USA, 2008.

[57] The top500 list, Nov. 2015. `http://www.top500.org`.

[58] The Green500 list, 2014. `http://www.green500.org`.

[59] 6ª generación de procesadores Intel® Core™ i7. `http://www.intel.es/content/www/es/es/processors/core/core-i7-processor.html`.

[60] Whitepaper. NVIDIA's Next Generation. CUDA TM Compute Architecture: FERMI™. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[61] Abbas-Turki, L. A., Vialle, S., Lapeyre, B., and Mercier, P. High dimensional pricing of exotic European contracts on a GPU cluster, and comparison to a CPU cluster. In*IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*(2009), IEEE, pp. 1-8.

[62] Gaikwad, A., and Toke, I. M. GPU based sparse grid technique for solving multidimensional options pricing PDEs. In*Proceedings of the 2nd Workshop on High Performance Computational Finance* (Nov. 2009), D. Daly, M. Eleftheriou, J. E. Moreira, and K. D. Ryu, Eds., ACM.

[63] Genovese, L., Ospici, M., Deutsch, T., Mehaut, J. F., Neelov, A., and Goedecker, S. Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. *The Journal of Chemical Physics 131*, 3 (2009), 034103+.

[64] Playne, D. P., and Hawick, K. A. Data parallel three-dimensional Cahn-Hilliard field equation simulation on GPUs with CUDA. In *International Conference on Parallel and Distributed Processing Techniques and Applications* (2009), H. R. Arabnia, Ed., pp. 104–110.

[65] Brandvik, T., and Pullan, G. Acceleration of a 3D Euler solver using commodity graphics hardware. In *46th AIAA Aerospace Sciences Meeting* (January 2008), American Institute of Aeronautics and Astronautics.

[66] Phillips, E. H., Zhang, Y., Davis, R. L., and Owens, J. D. Rapid aerodynamic performance prediction on a cluster of graphics processing units. In *Proceedings of the 47th AIAA Aerospace Sciences Meeting* (Jan. 2009), no. AIAA 2009-565.

[67] Barrachina, S., Castillo, M., Igual, F. D., Mayo, R., Quintana-Ortí, E. S., and Quintana-Ortí, G. Exploiting the capabilities of modern GPUs for dense matrix In *computations. Concurr. Comput. : Pract. Exper. 21, 18* (2009), 2457–2477.

[68] Quintana-Ortí, G., Igual, F. D., Quintana-Ortí, E. S., and van de Geijn, R. A. Solving dense linear systems on platforms with multiple hardware accelerators. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Feb. 2009), ACM, pp. 121–130.

[69] Hartley, T. D., Catalyurek, U., Ruiz, A., Igual, F., Mayo, R., and Ujaldon, M. Biomedical image analysis on a cooperative cluster of GPUs and multicores. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing* (2008), pp. 15–25.

[70] Luo, Y. C., and Duraiswami, R. Canny edge detection on NVIDIA CUDA. In *Computer Vision on GPU* (2008).

[71] NVIDIA Corporation. World leader in visual computing technologies — NVIDIA. `http://www.nvidia.com,2012`.

[72] Advanced Micro Devices, Inc. Global provider of innovative graphics, processors and media solutions — AMD. `http://www.amd.com,2012`.

[73] Liu, W., Schmidt, B., Voss, G., Schroeder, A., and Muller-Wittig, W. Biosequence database scanning on a GPU. In *20th International Parallel and Distributed Processing Symposium (IPDPS)* (Rhodes Island, Apr. 2006).

[74] Shreiner, D., and OpenGL. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Versions 3.0 and 3.1 (7th Edition)*. Addison-Wesley Professional, August 2009.

[75] Igual, F. D., Mayo, R., and Quintana-Ortí, E. S. Attaining high performance in general-purpose computations on current graphics processors. In *High Performance Computing for Computational Science — VECPAR 2008* (Toulouse, France, June 2008), J. M. L. M. Palma, P. Amestoy, M. J. Dayde, M. Mattoso, and J. C. Lopes, Eds., vol. 5336 of Lecture Notes in Computer Science, Springer-Verlag, pp. 406–419.

[76] Mark, W. R., Glanville, S. R., Akeley, K., and Kilgard, M. J. Cg: a system for programming graphics hardware in a C-like language. In *SIGGRAPH '03* (New York, NY, USA, 2003), ACM Press, pp. 896–907.

[77] Munshi, A., Ed. *OpenCL 1.0 Specification*. Khronos OpenCL Working Group, 2009.

[78] International Business Machine Corp. IBM - United States. `http://www.ibm.com,2012`.

[79] ARM Ltd. ARM - the architecture for the digital world. `http://www.arm.com,2012`.

[80] Levesque, J. Using OpenACC to develop portable exascale programs. In *Cray Technical Workshop on XK6 Programming* (Oct. 2012).

[81] OpenACC. https://sites.google.com/site/openaccv/openacc. (junio de 2016).

[82] Fang, J., Varbanescu, A., and Sips, H. A comprehensive performance comparison of CUDA and OpenCL. In *2011 International Conference on Parallel Processing (ICPP)* (Sept. 2011), pp. 216–225.

[83] Karimi, K., Dickson, N. G., and Hamze, F. A Performance Comparison of CUDA and OpenCL. *ArXiv e-prints* (May 2010). Online: http://arxiv.org/pdf/1005.2581v2.

[84] NVIDIA Corporation. NVIDIA CUDA C PROGRAMMING GUIDE v5.5, July 2013.

[85] NVIDIA Corporation. NVIDIA CUDA COMPILER DRIVER NVCC v5.5, July 2013.

[86] NVIDIA Corporation. NVIDIA CUDA C BEST PRACTICES GUIDE v5.5, July 2013.

[87] NVIDIA Corporation. NVIDIA CUDA DRIVER API v5.5, July 2013.

[88] NVIDIA Corporation. NVIDIA CUDA RUNTIME API v5.5, July 2013.

[89] NVIDIA Corporation. NVIDIA Parallel Thread Execution ISA v2.3, July 2013.

[90] NVIDIA Corporation. NVIDIA GPUDirect^{TM}. http://developer.nvidia.com/cuda/nvidia-gpudirect, 2012.

[91] NVIDIA Corporation. An Introduction to Modern GPU Architecture. Ashu Rege Director of Developer Technology.

[92] Ashwin M. Aji, Mayank Daga, and Wu-chun Feng. Bounding the effect of partition camping in GPU kernels. In *Proceedings of the 8th ACM International Conference on Computing Frontiers* (CF'11). ACM, New York, NY, USA, 2011, Article 27, 10 pages. DOI=http://dx.doi.org/10.1145/2016604.2016637.

[93] Ruetsch, Greg and Micikevicius, Paulius. Optimizing matrix transpose in CUDA. In *Nvidia CUDA SDK Application Note*, 2009, vol. 18.

[94] Torres, Y., Gonzalez-Escribano, A., & Llanos, D. R. (2012, July). Using Fermi architecture knowledge to speed up CUDA and OpenCL programs. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications* (pp. 617-624). IEEE.

[95] NVIDIA Corporation. Whitepaper: NVIDIA's Next Generation. CUDA Compute Architecture: Fermi, 2009.

[96] NVIDIA Corporation. Whitepaper: NVIDIA's Next Generation. CUDA Compute Architecture: Kepler GK110, 2012.

[97] Torres, Yuri and Gonzalez-Escribano, Arturo and Llanos, Diego R. *uBench: exposing the impact of CUDA block geometry in terms of performance.* In *The Journal of Supercomputing* vol 65, pages 1150-1165, 2013.

CAPÍTULO 3

Unveiling the Performance-Energy Trade-Off in Iterative Linear System Solvers for Multithreaded Processors

J. I. Aliaga, H. Anzt, M. Castillo, J. C. Fernández, G. León, **J. Pérez***, E. S. Quintana-Ortí.*
"Unveiling the performance-energy trade-off in iterative linear system solvers for multithreaded processors".
Concurrency and Computation: Practice & Experience, Vol. 27(4), pp. 885-904, 2015.
ISSN: 1532-0626.

Abstract

In this paper we analyze the interactions occurring in the triangle performance-power-energy for the execution of a pivotal numerical algorithm, the iterative Conjugate Gradient (CG) method, on a diverse collection of parallel multithreaded architectures. This analysis is especially timely in a decade where the power wall has arisen as a major obstacle to build faster processors. Moreover, the CG method has recently been proposed as a complement to the LINPACK benchmark, as this iterative method is argued to be more archetypical of the performance of today's scientific and engineering applications.

To gain insights about the benefits of hands-on optimizations we include runtime and energy efficiency results for both out-of-the-box usage relying exclusively on compiler optimizations, and implementations manually optimized for target architectures, that range from general-purpose and digital signal multicore processors to manycore graphics processing units (GPUs), all representative of current multithreaded systems.

3.1 Introduction

At a rough cost of $1 million USD per MegaWatt (MW) year, current high performance computing (HPC) facilities and large-scale datacenters are painfully aware of the *power wall*. This is recognized as a crucial hurdle by the HPC community, and many ongoing developments towards the world's first exascale system are shaped by the expected power demand and the related energy costs. For instance, a simple look at the Top500 and Green500 lists [98, 118] reveals that an ExaFLOP computer built using the same

technology employed in today's fastest supercomputer, would dissipate more than 520 MW, resulting in an unmistakable call for power-efficient systems and energy proportionality [105, 113, 115, 117, 122].

One important follow-up of the end of Dennard scaling [112] (i.e., the ability to drop the voltage and the current that transistors need to operate reliably as their size shrinks) is the surge of dark silicon [116], and consequently the development of specialized processors composed of heterogeneous core architectures with more favorable performance-power ratios. This trend is visible, e.g., in the Top500 list, with the current top two systems being equipped with NVIDIA GPUs and Intel Xeon Phi accelerators. Against this background, although most manufacturers advertise the power-efficiency of their products by providing theoretical energy specifications, an equitable comparison between different hardware architectures remains difficult. The reason is not only that distinct devices are often designed for different types of computations, but also that they are tailored for either performance or power efficiency. New energy-related metrics have been recently proposed to analyze the balance between these two key figures [107], but the situation becomes increasingly difficult once the different levels of optimization applied to an algorithm enter the picture.

In response to this situation, this paper analyzes the performance-energy trade-offs of an ample variety of multithreaded general-purpose and specialized architectures, using the conjugate gradient (CG) method as a workhorse. This method is a key algorithm for the numerical solution of symmetric positive definite (s.p.d.) sparse linear systems [129]. Furthermore, since the cornerstone of this iterative method is the sparse matrix-vector product (SpMV), an operation that is also crucial for many other numerical methods [103], the significance of the results carries over to many other applications. In addition, the CG method has been recently proposed as a complement to the LINPACK benchmark on the basis of being more representative of the actual performance attained by current scientific and engineering codes that run on HPC platforms [114]. Concretely, the LINPACK benchmark comprises the solution of a (huge) dense system of linear equations via the LU factorization, a compute-bounded operation that is known to deliver a GFLOPS (billions of floating-point arithmetic operations, or flops, per second) rate close to that of the matrix-matrix product and, therefore, the theoretical peak performance of the underlying platform. The CG method, on the other hand, is composed of memory-bound kernels, an attribute shared by a considerably larger number of HPC applications, and offers a much lower GFLOPS throughput than the LINPACK metric.

Our analysis covers two different scenarios. We first review the study in [99], replacing the single-precision (SP) arithmetic with double-precision (DP) experiments, as this is the norm in numerical linear algebra. In [99] we refrained from applying any manual hardware-aware optimizations to the code, but instead evaluated basic SP implementations of the CG method and SpMV, using either the baseline CSR (compressed sparse rows) or ELLPACK sparse matrix formats [129], and relying exclusively on the optimizations applied by the compiler. That scenario thus provided insights on the resource efficiency achieved when running complex numerical codes on large HPC facilities without applying hands-on optimization. In this paper we extend the study significantly by considering an alternative scenario which aims to increase resource efficiency by replacing the standard SpMV with a more sophisticated implementation, and, in the presence of a hardware graphics accelerator, modifying the basic CG algorithm to reduce the overall kernel count [100]. This allows us to also assess the improvement potential of algorithmic modifications on the distinct hardware architectures.

The rest of the paper is structured as follows. In Section 3.2 we briefly introduce the mathematical formulation of the CG method as well as the matrix benchmarks and hardware architectures we evaluate, which include four general-purpose multicore processors (Intel Xeon E5504 and E5-2620, and AMD Opteron 6128 and 6276); a low-power multicore digital signal processor (Texas Instruments C6678); three low-power multicore processors (ARM Cortex A9, Exynos5 Octa, and Intel Atom S1260); and three GPUs with different capabilities (NVIDIA Quadro M1000, Tesla C2050, and Kepler K20). Section 3.3 contains the performance and energy efficiency results obtained from the basic DP implementations of the CG method, where only compiler-intrinsic optimizations are applied. These figures are constrasted next, in Section 3.4, against the results obtained when applying the hands-on optimizations described previously in that section. We finally

conclude the paper in Section 3.5 with a short summary about the findings and a discussion of ideas for future research.

3.2 Background and Experimental Setup

3.2.1 Krylov-based iterative solvers

The CG method [129] is usually the preferred Krylov subspace-based solver to tackle a linear system of the form

$$Ax = b, \tag{3.1}$$

where $A \in \mathbb{R}^{n \times n}$ is sparse s.p.d., $b \in \mathbb{R}^n$ contains the independent terms, and $x \in \mathbb{R}^n$ is the sought-after solution. The method is mathematically formulated in Figure 3.1, where the user-defined parameters *maxres* and *maxiter* set upper bounds, respectively, on the residual for the computed approximation to the solution, x_k, and the maximum number of iterations.

1: $x_0 := 0$ // or any other initial guess
2: $r_0 := b - Ax_0$, 3: $d_0 := r_0$
4: $\beta_0 := r_0^T r_0$, 5: $\tau_0 :=\| r_0 \|_2 = \sqrt{\beta_0}$, 6: $k := 0$
7: **while** $(k < maxiter)$ & $(\tau_k > maxres)$
8: $z_k := Ad_k$ (SPMV)
9: $\rho_k := \beta_k / d_k^T z_k$ (`dot`)
10: $x_{k+1} := x_k + \rho_k d_k$ (`axpy`)
11: $r_{k+1} := r_k - \rho_k z_k$ (`axpy`)
12: $\beta_{k+1} := r_{k+1}^T r_{k+1}$ (`dot`)
13: $\alpha_k := \beta_{k+1} / \beta_k$
14: $d_{k+1} := r_{k+1} + \alpha_k d_k$ (`scal+axpy`)
15: $\tau_{k+1} :=\| r_{k+1} \|_2 = \sqrt{\beta_{k+1}}$
16: $k := k + 1$
17: **end**

Figure 3.1: Mathematical formulation of the CG method. The names inside parenthesis, except for SPMV which refers to the sparse matrix-vector kernel, identify the routine from the level-1 BLAS that offers the corresponding functionality.

In practical applications, the computational cost of the CG method is dominated by the matrix-vector multiplication $z_k := Ad_k$. Given a sparse matrix A with n_z nonzero entries, this operation roughly requires $2n_z$ flops. The sparse matrix-vector product is ubiquitous in scientific computing, being a key operation for the iterative solution of linear systems and eigenproblems as well as the PageRank algorithm, among others [103, 124, 129]. The irregular memory access pattern of this operation, in combination with the limited memory bandwidth of current general-purpose architectures, has resulted in a considerable number of efforts that propose specialized matrix storage layouts as well as optimized implementations for a variety of architectures; see, e.g., [106, 111, 133] and the references therein. Although we target only symmetric systems with our CG implementations for which storage formats and matrix vector routines that efficiently exploit this property exist [129], we refrain from using them. We argue, that by not considering this optimization, our analysis and results can be easily extrapolated to the case of unsymmetric sparse linear system solvers.

In addition to the matrix-vector multiplication, the loop body of the CG method contains several vector operations, with a cost of $O(n)$ flops each, for the updates of x_{k+1}, r_{k+1}, d_{k+1}, and the computation of ρ_k and β_{k+1}. These operations are supported as part of the level-1 BLAS [125]; see Figure 3.2.

```
1  while( ( k < maxiter ) && ( tau > maxres ) ){
2    SpMV ( n, Asparse, d, z );            // z := A * d
3    tmp = ddot ( n, d, 1, z, 1 );         // tmp := d' * z
4    rho = beta / tmp;                     // rho := beta / tmp
5    gamma = beta;                         // gamma := beta
6    daxpy (n,  rho, d, 1, x, 1 );         // x := x + rho * d
7    daxpy (n, -rho, z, 1, r, 1 );         // r := r - rho * z
8    beta = cublasSdot( n, r, 1, r, 1 );   // beta := r' * r
9    alpha = beta / gamma;                 // alpha := beta / gamma
10   dscal (n, alpha, d, 1 );              // d := alpha * d
11   daxpy (n, one, r, 1, d, 1 );          // d := d + r
12   tau = sqrt( beta );                   // tau := sqrt(beta)
13   k++;
14 } // end-while
```

Figure 3.2: Simplified loop-body of the basic CG implementation using double precision (DP).

3.2.2 Matrix benchmarks

For our experiments, we selected six s.p.d. matrices from the University of Florida Matrix Collection (UFMC)[1], corresponding to finite element discretizations of several structural problems arising in mechanics, and an additional case derived from a finite difference discretization of the 3D Laplace problem; see Table 3.1. For these linear systems, the right-hand side vector b was initialized to be consistent with the solution $x \equiv 1$, while the CG iteration was started with the initial guess $x_0 \equiv 0$. Except where stated otherwise, all tests were conducted with DP arithmetic. (We note that the experimentation in [99] was performed using SP arithmetic only.)

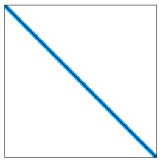
3.2.3 Hardware Setup and Compilers

Table 3.2 lists the main features of the hardware systems and compilers utilized in the experiments. For each multi-core processor we also report the different processor frequencies that were evaluated. Aggressive optimization was applied to all implementations through flag `-O3`.

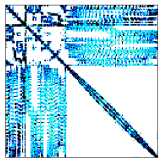
In order to measure power, we leveraged a WATTSUP?PRO wattmeter, connected to the line from the electrical socket to the power supply unit (PSU), with an accuracy of $\pm 1.5\%$ and a rate of 1 sample/sec. These results were collected on a separate server to avoid inteferring with the application performance and consumption. For all test matrices, we based the analysis on the average power draft when executing 1000 CG iterations after a warm up period of 5 minutes using the same test. Since the platforms where the processors are embedded contain other devices —e.g., disks, network interface cards, fans, etc.— on each platform we measured the average idle power and then subtracted the corresponding value (see Table 3.2) from all the samples obtained from the wattmeter. We believe this setup enables a relevant comparison between the energy efficiencies of the different architectures, as proceeding in this manner we only take into account the (net) energy that is drawn for the actual work, but eliminate power sinks such as the inefficiencies of the PSU, the network interface, or the disk.

[1] `http://www.cise.ufl.edu/research/sparse/matrices.`

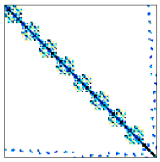
Source	Acronym	Matrix	#nonzeros (n_z)	Size (n)	n_z/n
Laplace	A159	A159	27,986,067	4,019,679	6.96
UFMC	AUDI	AUDIKW_1	77,651,847	943,645	82.28
	BMW	BMWCRA1	10,641,602	148,770	71.53
	CRANK	CRANKSEG_2	14,148,858	63,838	221.63
	F1	F1	26,837,113	343,791	78.06
	INLINE	INLINE_1	38,816,170	503,712	77.06
	LDOOR	LDOOR	42,493,817	952,203	44.62



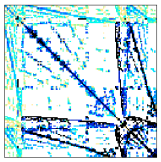
(a) A159



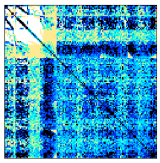
(b) AUDIKW_1



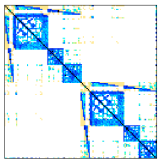
(c) BMWCRA_1



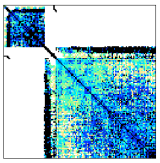
(d) CRANKSEG_2



(e) F1



(f) INLINE_1



(g) LDOOR

Table 3.1: Description and properties of the test matrices (top) and the corresponding sparsity plots (bottom).

3.3 Basic CG Method relying on Compiler Optimization

In this section we analyze a straight-forward implementation of the CG method, with the only optimizations being applied automatically by the compiler, when configuring the hardware for either performance or energy efficiency through adjusting the processor operation frequency and number of active cores. The reference code for all of these baseline implementations is given in Figure 3.2.

3.3.1 Basic implementation of the CG method

In our basic implementations of the sparse matrix-vector product for the CG method, matrix A is stored in the CSR format for multicore architectures, or the ELLPACK format for the GPUs. Both schemes aim to reduce the memory footprint by explicitly storing only the non-zero elements, though the ELLPACK format may store some zero elements for padding all rows to the same length; see Figure 3.3 and Table 3.3 for the evoked overhead. While in general this incurs some additional storage cost, the aligned structure allows for more efficient hardware use when targeting streaming processors like the GPUs [108, 130].

In Figure 3.4 we sketch the sparse matrix-vector product kernels for the CSR and ELLPACK formats. In both routines, `n` refers to the matrix size; the matrix is stowed using arrays `values`, `colind` and, in the case of routine `SpMV_csr`, also array `rowptr` (see Figure 3.3); the input and output vectors of the product $y := Ax$ are, respectively, `x` and `y`; finally, `nzr` refers to the number of entries per row in the ELLPACK-based routine.

Acron.	Architecture	Total #cores	Frequency (GHz) – Idle power (W)	RAM size, type	Compiler
AIL	AMD Opteron 6276 (Interlagos)	8	1.4–167.29, 1.6–167.66 1.8–167.31, 2.1–167.17 2.3–168.90	64GB, DDR3 1.3GHz	`icc` 12.1.3
AMC	AMD Opteron 6128 (Magny-Cours)	8	0.8–107.48, 1.0–109.75, 1.2–114.27, 1.5–121.15, 2.0–130.07	48GB, DDR3 1.3GHz	`icc` 12.1.3
IAT	Intel Atom S1260	2	0.6–41.94, 0.90–41.93, 1.30–41.97, 1.70–41.95 2.0–42.01	8GB, DDR3 1.3GHz	`icc` 12.1.3
INH	Intel Xeon E5504 (Nehalem)	8	1.60–33.43, 1,73–33.43, 1.87–33.43, 2.00–33.43	32GB, DDR3 800MHz	`icc` 12.1.3
ISB	Intel E5-2620 (Sandy-Bridge)	6	1.2–113.00, 1.4–112.96, 1.6–112.77, 1.8–112.87, 2.0–112.85	32GB, DDR3 1.3GHz	`icc` 12.1.3
A9	ARM Cortex A9	4	0.76–10.0, 1.3–10.1	2GB, DDR3L	`gcc` 4.6.3
A15	Exynos5 Octa (ARM Cortex A15 + A7)	4+4	0.25–2.2, 1.6–2.4	2GB, LPDDR3	`gcc` 4.7
FER	Intel Xeon E5520 NVIDIA Tesla C2050 (Fermi)	8 448	1.6–222.0, 2.27–226.0 1.15	24GB, 3GB, GDDR5	`gcc` 4.4.6 `nvcc` 5.5
KEP	Intel Xeon i7-3930K NVIDIA Tesla K20 (Kepler)	6 2,496	1.2–106.30, 3.2–106.50 0.7	24GB, 5GB, GDDR5	`gcc` 4.4.6 `nvcc` 5.5
QDR	ARM Cortex A9 NVIDIA Quadro 1000M	4 96	0.120–11.2, 1.3–12.2 1.4	2GB, DDR3L 2GB, DDR3	`gcc` 4.6.3 `nvcc` 5.5
TIC	Texas Instruments C6678	8	1.0–18.0	512MB, DDR3	`cl6x` 7.4.1

Table 3.2: Multithreaded architectures. For the GPU systems (FER, KEP, and QDR), the idle power includes the consumption of both the server and the accelerator.

In the basic implementation for multicore processors (`SpMV_csr`), concurrency is exploited via the OpenMP application programming interface, with the matrix-vector operation partitioned by rows, and each OpenMP thread being responsible for all the arithmetic operations necessary to update a certain number of entries of `y`. In these architectures, we employed the legacy implementation of BLAS from *netlib*[2] for the level-1 (vector) operations (kernels `sdot`, `daxpy`, `scal` in Figure 3.2). No attempt was made to extract parallelism from these BLAS kernels as, due to their low computational cost, a parallel execution on a conventional multithreaded architecture rarely offers any benefits.

For the GPUs, concurrency is exploited in `SpMV_ell` by rows, with one CUDA thread being responsible for the computation of one element of `y`. On these streaming architectures we used the multithreaded implementation in NVIDIA's CUBLAS, replacing the invocations to `ddot`, `daxpy`, and `dscal` in Figure 3.2 with calls to `cublasDdot`, `cublasDaxpy`, and `cublasDcal`, respectively. (For the resulting code, see also the left-hand side of Figure 3.7.) We consider this a fair comparison as *i)* in general, the time cost of the vector operations is significantly lower than that of the sparse matrix-vector product; and *ii)* due to their reduced cost, there is little opportunity to benefit from a concurrent execution of the vector operations on a multicore processor.

3.3.2 Search for the optimal configuration

Tables 3.4, 3.5, and Figure 3.5 report the performance and net energy (i.e., energy after subtracting the cost of idle power) per iteration, for the different platforms and benchmark cases, obtained with the basic

[2] `http://www.netlib.org`.

DP implementations. (For an analogous study with SP arithmetic, refer to [99].) For each architecture and matrix, we report the combination of core number and frequency that yields the shortest execution time per iteration, together with the corresponding net energy ("optimized w.r.t. time"), as well as the configuration that provides the most energy-efficient result ("optimized w.r.t. energy"). As the tables and figure convey a considerable amount of information, we limit the following analysis to some central aspects.

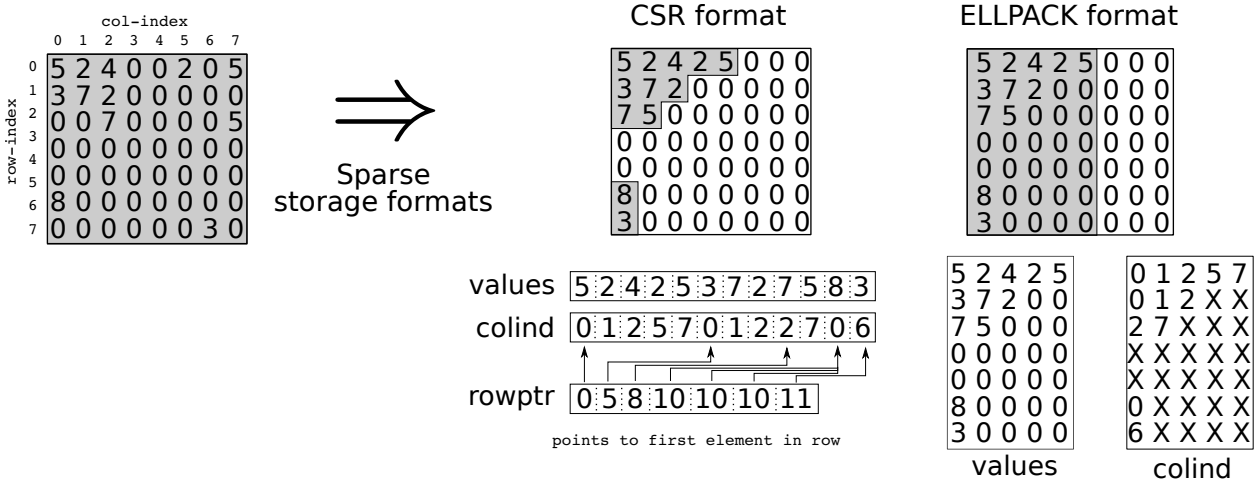


Figure 3.3: Basic dense and sparse matrix storage formats. The memory demand corresponds to the grey areas.

Matrix	#nonzeros (n_z)	Size (n)	n_z/n	n_z^{row}	ELLPACK $n_z^{ELLPACK}$	ELLPACK overhead	SELL-P n_z^{SELL-P}	SELL-P overhead
A159	27,986,067	4,019,679	6.96	7	28,137,753	0.54%	32,157,440	12.97%
AUDI	77,651,847	943,645	82.28	345	325,574,775	76.15%	95,556,416	18.74%
BMW	10,641,602	148,770	71.53	351	52,218,270	79.62%	12,232,960	13.01%
CRANK	14,148,858	63,838	221.63	3423	218,517,474	93.53%	15,991,232	11.52%
F1	26,837,113	343,791	78.06	435	149,549,085	82.05%	33,286,592	19.38%
INLINE	38,816,170	503,712	77.06	843	424,629,216	91.33%	45,603,264	19.27%
LDOOR	42,493,817	952,203	44.62	77	73,319,631	42.04%	52,696,384	19.36%

Table 3.3: Storage overhead of the test matrices when using ELLPACK or SELL-P format. n_z^{FORMAT} refers to the explicitly stored elements (n_z nonzero elements and the explicitly stored zeros for padding).

3.3.2.1 Optimization with respect to time

If aiming for performance using the basic CG implementation, it is not possible to identify an overall winner, as the performance turns out to be highly problem dependent (see the top-left graph in Figure 3.5). For example, the Tesla K20 (KEP) achieves almost 12 GFLOPS for the A159 matrix, a number that is most closely followed by the Tesla C2050 GPU's 7.5 GFLOPS (FER), and about 9 times faster than the best of all the CPUs for that particular case (1.3 GFLOPS attained with the Intel E5-2620, ISB). On the other hand, for the more involved sparsity structure of the CRANK problem, KEP only delivers 1.2 GFLOPS, while

```
1   void SpMV_csr( int n,
2                  int * rowptr, int * colind, double * values,
3                  double * x, double * y ) {
4     int i, j;    double tmp;
5
6     #pragma omp parallel for private ( j, tmp )
7     for ( i = 0; i < n; i++ ) {
8       tmp = 0.0;
9       for ( j = rowptr [ i ]; j < rowptr [ i+1 ]; j++ )
10        tmp +=  values [ j ] * x [ colind[ j ] ];
11      y[i] = tmp;
12    }
13  }
```

```
1   __global__
2   void SpMV_ell( int n, int nzr,
3                  int * colind, double * values,
4                  double * x,   double * y ) {
5     int i, j, k; double tmp;
6
7     i = blockDim.x * blockIdx.x + threadIdx.x;
8     if ( i < n ){
9       tmp = 0.0;
10      for ( j = 0; j < nzr; j++ ) {
11        k = n * j + i;
12        if ( values[ k ] != 0 )
13          tmp += values [ k ] * x [ colind [ k ] ];
14      }
15      y[i] = tmp;
16    }
17  }
```

Figure 3.4: Sparse matrix-vector product using the CSR and ELLPACK formats (`SpMV_csr` and `SpMV_ell`, respectively).

the CPU implementations, using the less problem-dependent CSR format instead of ELLPACK, achieve between 2 and 4 GFLOPS on recent hardware (AIL and ISB, respectively). A closer inspection reveals that the GPU's performance is directly related to the overhead induced by the usage of the ELLPACK format (see Table 3.3). Also, the performance of the older CPU generations (AMC and INH) is less problem-dependent, whereas the other two GPU architectures are similarly sensitive to the matrix structure. Finally, while the low-power architectures —IAT, A9, A15, and TIC— provide the lowest GFLOPS rate for most problems, in some cases the usage of the ELLPACK format on the GPUs may incur significant overhead which their computing power cannot compensate for, such that even the low-power processors may become competitive in those cases (compare e.g., the performance of TIC and QDR for the BMW case).

The performance sensitivity of the architectures to the problem characteristics and the format-dependent overhead also translate into the corresponding energy efficiency, so that the GPUs are only superior to the CPUs for the structured problems A159 and LDOOR (see Figure 3.5 right-top). While the C66780 from Texas Instruments (TIC) also shows some variation on performance as well as the performance-per-watt ratio depending on the matrix structure, its energy efficiency is unmatched by any other architecture. The remaining low-power processors, A9, A15, and IAT, achieve higher GFLOPS/W rates than the conventional general-purpose processors AIL, AMC, and INH, but are only more efficient than GPUs for certain matrix cases. With respect to energy, ISB is at least competitive with the older ARM Cortex A9.

		Optimized w.r.t. time				Optimized w.r.t. net energy			
	Matrix	c	f	T	E_{net}	c	f	T	E_{net}
AIL	A159	8	2300	1.00E−01	1.72E+01	4	2100	1.14E−01	1.53E+01
	AUDI	8	2300	1.07E−01	1.70E+01	8	2300	1.07E−01	1.70E+01
	BMW	8	2300	1.13E−02	1.89E+00	8	2100	1.15E−02	1.82E+00
	CRANK	8	2300	1.42E−02	2.46E+00	8	2100	1.46E−02	2.34E+00
	F1	8	2300	3.82E−02	6.15E+00	8	2300	3.82E−02	6.15E+00
	INLINE	8	2300	4.48E−02	7.77E+00	4	2100	5.24E−02	7.26E+00
	LDOOR	8	2300	6.17E−02	1.09E+01	8	2100	6.25E−02	9.76E+00
AMC	A159	8	2000	2.11E−01	1.94E+01	8	2000	2.11E−01	1.94E+01
	AUDI	8	2000	2.12E−01	2.06E+01	8	2000	2.12E−01	2.06E+01
	BMW	8	2000	1.83E−02	2.01E+00	8	2000	1.83E−02	2.01E+00
	CRANK	8	2000	2.57E−02	2.82E+00	8	2000	2.57E−02	2.82E+00
	F1	8	2000	7.40E−02	7.59E+00	8	2000	7.40E−02	7.59E+00
	INLINE	8	2000	8.15E−02	8.55E+00	8	2000	8.15E−02	8.55E+00
	LDOOR	8	2000	1.02E−01	1.08E+01	8	2000	1.02E−01	1.08E+01
IAT	A159	2	2000	3.01E−01	1.27E+00	1	600	1.25E+00	8.53E−01
	AUDI	2	2000	3.79E−01	1.48E+00	1	600	1.84E+00	9.60E−01
	BMW	2	2000	4.35E−02	1.95E−01	1	600	2.43E−01	1.37E−01
	CRANK	2	2000	5.82E−02	2.43E−01	1	600	3.07E−01	1.43E−01
	F1	2	2000	1.78E−01	6.12E−01	1	600	8.15E−01	3.94E−01
	INLINE	2	2000	1.74E−01	7.23E−01	1	600	9.09E−01	4.24E−01
	LDOOR	2	2000	2.38E−01	9.61E−01	1	600	1.19E+00	6.60E−01
INH	A159	4	1870	1.01E−01	1.01E+01	2	1870	1.03E−01	9.30E+00
	AUDI	8	2000	1.00E−01	1.35E+01	4	1730	1.34E−01	1.28E+01
	BMW	4	1870	1.29E−02	1.37E+00	4	1600	1.33E−02	1.29E+00
	CRANK	4	1870	1.65E−02	1.75E+00	4	1600	1.72E−02	1.64E+00
	F1	8	1730	3.81E−02	4.97E+00	4	1600	5.01E−02	4.72E+00
	INLINE	8	1600	5.00E−02	5.97E+00	4	1600	5.27E−02	5.04E+00
	LDOOR	8	1870	6.84E−02	9.66E+00	4	1600	7.12E−02	6.91E+00
ISB	A159	6	2000	5.56E−02	2.78E+00	2	1200	1.08E−01	1.86E+00
	AUDI	6	2000	6.12E−02	3.10E+00	2	1400	1.30E−01	2.46E+00
	BMW	6	2000	5.51E−03	3.24E−01	6	1200	8.17E−03	2.63E−01
	CRANK	6	2000	7.22E−03	4.56E−01	6	1200	1.11E−02	3.45E−01
	F1	6	2000	1.98E−02	1.04E+00	6	1200	2.92E−02	8.51E−01
	INLINE	6	2000	2.31E−02	1.27E+00	4	1400	3.53E−02	1.05E+00
	LDOOR	6	2000	3.06E−02	1.75E+00	6	1200	4.54E−02	1.45E+00

Table 3.4: Optimal hardware parameter configuration when optimizing the general-purpose architectures for runtime or energy efficiency using the basic implementations. In the labels, c denotes the number of cores, f the frequency (in MHz), T the time per iteration (in seconds), and E_{net} the net energy per iteration (in Joules).

3.3.2.2 Optimization with respect to net energy

The first general observation is that, in many cases, reducing the CPU operating frequency (and voltage) pays off. The FER and KEP GPU-accelerated platforms are two notable exceptions, as rescaling the frequency operation of the host CPU of these systems has negligible impact on the overall performance and energy efficiency of the solver. The behavior is very different for the low-power processors and QDR, where rescaling the CPU frequency can improve the energy efficiency by a wide margin. For instance, reducing the operating frequency of the recent ARM Cortex (A15) from 1600 to 250 MHz improves the efficiency ratio by a factor of 5.5 (see the results for A159 in Table 3.5: from 1.12 to 2.02E−01 GFLOPS/W; also compare the top and bottom plots on the right of Figure 3.5). The results for IAT and A9 (see Tables 3.4

		Optimized w.r.t time				Optimized w.r.t net energy			
	Matrix	c	f	T	E_{net}	c	f	T	E_{net}
A9	A159	4	760	7.15E−01	1.04E+00	4	760	7.15E−01	1.04E+00
	AUDI	4	1300	9.06E−01	2.89E+00	2	760	1.30E+00	1.76E+00
	BMW	4	1300	1.45E−01	5.35E−01	2	760	1.83E−01	2.74E−01
	CRANK	4	1300	1.68E−01	5.15E−01	2	760	2.26E−01	3.05E−01
	F1	4	1300	3.57E−01	1.22E+00	2	760	5.03E−01	6.74E−01
	INLINE	4	1300	4.88E−01	1.72E+00	2	760	6.59E−01	9.88E−01
	LDOOR	4	1300	6.16E−01	2.10E+00	2	760	8.45E−01	1.17E+00
A15	A159	4	1600	1.70E−01	1.12E+00	4	250	8.08E−01	2.02E−01
	AUDI	4	1600	2.11E−01	1.64E+00	4	250	7.87E−01	2.68E−01
	BMW	4	1600	3.12E−02	2.76E−01	4	250	1.10E−01	4.62E−02
	CRANK	4	1600	3.61E−02	3.17E−01	4	250	1.34E−01	5.88E−02
	F1	4	1600	8.44E−02	6.65E−01	4	250	3.11E−01	1.15E−01
	INLINE	4	1600	1.11E−01	9.50E−01	4	250	3.89E−01	1.75E−01
	LDOOR	4	1600	1.44E−01	1.23E+00	4	250	5.21E−01	1.98E−01
FER	A159	1	1600	1.01E−02	1.77E+00	1	1600	1.01E−02	1.77E+00
	BMW	1	1600	1.02E−02	1.69E+00	1	1600	1.02E−02	1.69E+00
	CRANK	1	1600	4.58E−02	7.20E+00	1	1600	4.58E−02	7.20E+00
	F1	1	1600	3.36E−02	5.56E+00	1	1600	3.36E−02	5.56E+00
	LDOOR	1	1600	1.41E−02	2.58E+00	1	1600	1.41E−02	2.58E+00
KEP	A159	1	1200	6.80E−03	6.73E−01	1	1200	6.80E−03	6.73E−01
	AUDI	1	1200	3.53E−02	3.51E+00	1	1200	3.53E−02	3.51E+00
	BMW	1	1200	5.57E−03	5.37E−01	1	1200	5.57E−03	5.37E−01
	CRANK	1	1200	2.26E−02	2.07E+00	1	1200	2.26E−02	2.07E+00
	F1	1	1200	1.68E−02	1.64E+00	1	1200	1.68E−02	1.64E+00
	LDOOR	1	1200	8.57E−03	9.12E−01	1	1200	8.57E−03	9.12E−01
QDR	A159	1	1300	4.54E−02	1.15E+00	1	51	6.06E−02	8.15E−01
	BMW	1	1300	4.56E−02	1.07E+00	1	51	7.36E−02	9.78E−01
TIC	BMW	8	1000	3.06E−02	8.83E−02	4	1000	4.37E−02	1.098E−01
	CRANK	8	1000	4.18E−02	1.10E−01	8	1000	4.18E−02	1.10E−01
	F1	8	1000	1.27E−01	2.69E−01	8	1000	1.27E−01	2.69E−01

Table 3.5: Optimal hardware parameter configuration when optimizing the specialized architectures for runtime or energy efficiency using the basic implementations. In the labels, c denotes the number of cores, f the frequency (in MHz), T the time per iteration (in seconds), and E_{net} the net energy per iteration (in Joules).

and 3.5) also show significant improvements, which reveal that these architectures provide not only a more favorable baseline energy efficiency, but also higher optimization potential compared with the conventional general-purpose CPUs or GPUs. While the TIC provided the highest performance/watt ratio when optimizing for performance, A15 now becomes the overall winner, followed by TIC, IAT, and the older generation Cortex (A9). As the energy efficiency of the graphics accelerators KEP, FER, and QDR is again very problem-dependent, it is difficult to compare them against ISB, but they still deliver higher GFLOPS/W ratios than the older CPU architectures AIL, AMC, and INH. Unfortunately, the factors gained when improving energy efficiency translate into the related execution time, as the reduced Joule-per-iteration values for TIC, A9, A15, and IAT come at the price of significantly higher execution times. However, as these provide the lowest performance in any case, and the GPUs do not allow for too much hardware reconfiguration, the left-top and left-bottom graphs in Figure 3.5 look very similar. Optimizing ISB for either performance or energy efficiency renders improvement factors around 2, significantly higher than those observed for other CPU architectures.

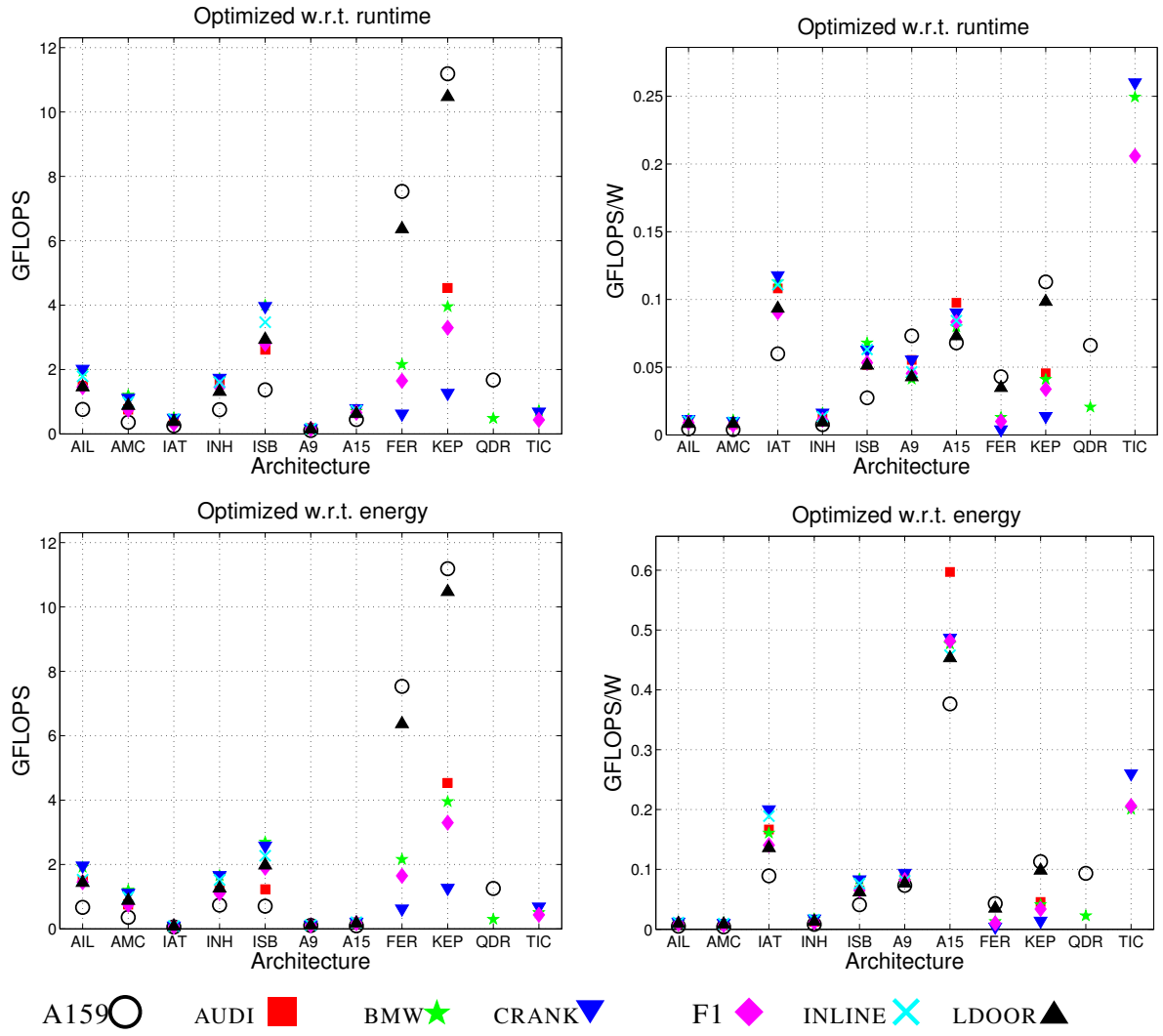


Figure 3.5: Comparison of performance (left) and energy efficiency (right), measured, respectively, in terms of GFLOPS and GFLOPS/W, when optimizing the basic CG implementations with respect to run time (top) or net energy (bottom).

Finally, we mention that optimizing with respect to net energy is not necessarily equivalent to optimizing for the total energy consumption. It may happen that the net energy consumption is reduced by decreasing the CPU frequency, at the cost of an increase of the total energy, as we did not consider the system's baseline power draft (idle power) in this analysis.

3.4 Hardware-Aware Optimization of the CG Method

While the results in the previous section offer insights on the computational and energy efficiencies of the target hardware architectures when running "unoptimized" code, we now address the question of how much improvement can be achieved by tailoring the codes to specific hardware. For this purpose we modify the basic implementations of the CG solver in two ways:

1. We replace the basic sparse matrix formats and the associated baseline matrix-vector product codes (see Figures 3.3 and 3.4) with more sophisticated solutions which can be expected to render higher performance, depending on the target hardware and the matrix benchmark.

2. On the streaming processors, the performance of memory-bound algorithms suffers from the read-/write operations from/to global memory that are required when launching a kernel. For this reason we leverage a variant of the CG algorithm where the SPMV and vector operations are reorganized and merged into several kernels [100], reducing the volume of memory access and the kernel launch overhead.

In the remainder of this section we expose the modifications introduced in the CG method, in the same order as enumerated above, and experimentally evaluate the effect they exert on the performance and energy efficiency of the solver.

3.4.1 Optimizing the matrix layout for SPMV

3.4.1.1 Multicore processors

CSR has become the de facto standard format for sparse matrices thanks to its flexibility and low memory requirements [132]. In particular, in exchange for keeping only the nonzero entries of the matrix, CSR introduces a storage overhead of only $n+n_z$ integer entries to represent the indices or pointers that determine the positions of these elements.

BCSR (block CSR) is a blocked variant of CSR that aims to improve register reuse for SPMV [120], by dividing the matrix into small dense blocks that are kept in consecutive locations in memory, as a sparse collection of dense blocks; see Figure 3.6. While BCSR introduces some storage overhead by padding the dense blocks with zeros, this is potentially compensated for by having to maintain a single index per block only, and by the performance advantage of improved register reuse that serves as motivation for the format. A detailed description of BCSR and a basic implementation of the SPMV operation can be found in [120].

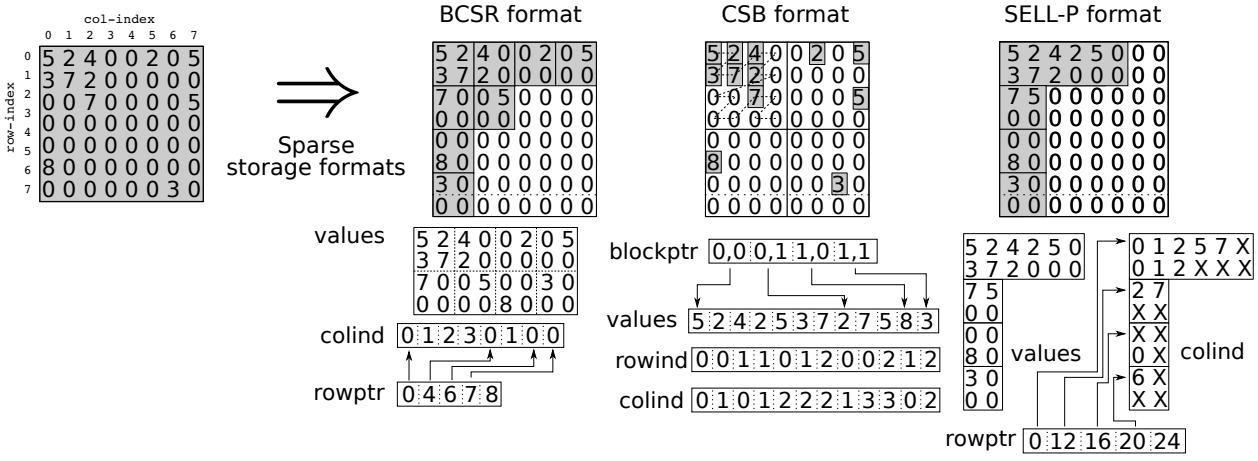


Figure 3.6: Visualizing the BCSR, CSB, and SELL-P formats. Note that choosing the block size 2 for BCSR and SELL-P, as well as the block size 4 for CSB, requires adding a zero row to the original matrix. Furthermore, padding the SELL-P format to a row length divisible by 2 requires explicit storage of a few additional zeros.

In our tailored version of the CG method, we leveraged two multithreaded routines of the SPMV routine in Intel MKL (version 10.3 update 9), based on the CSR and BCSR formats: `mkl_cspblas_dcsrgemv` and `mkl_cspblas_dbsrgemv`, respectively [121]. For the latter code, our tests considered values for the block size (parameter `lb`) equal to 2 and 3, but we only report the best result.

The alternative CSB (compressed sparse blocks) format [110] maintains the matrix as a dense collection of sparse blocks, with the blocks themselves being stored in Z-morton order [128]; see Figure 3.6 for a sketch of this layout. This format exhibits negligible storage overhead compared to CSR, and, more importantly, allows for an efficient parallelization of SPMV on current multicore architectures. We employ the multithreaded routine `bicsb_gespmv` from the CSB library [109] for the SPMV routine, linked to release 1.1 of Intel Cilk Plus and release 4.0 of Intel Thread Building Blocks (TBB).

The optimized implementations of the CG method for the multicore platforms are analogous to the code in Figure 3.2, with the SPMV routine replaced by the appropriate implementation of the matrix-vector product (either one of the MKL routines or the CSB library code), and linked in all cases with the MKL implementation of BLAS [121] for routines `ddot`, `daxpy`, and `dscal`.

3.4.1.2 Graphics accelerators

The ELLPACK format incurs a storage overhead, for the general case, which grows with the differences in the number of nonzero elements per row (see Table 3.3). In those cases, the associated memory and computational overheads may result in poor performance, despite that coalesced memory access is highly beneficial for streaming processors like the GPUs. ELLR-T [130, 131] is a subtle variant of ELLPACK that addresses this problem while maintaining the coalesced memory access pattern of the original layout. In particular, it reduces useless computations with zeros and improves thread load balancing, often resulting in superior performance.

An alternative workaround to reduce memory and computational overhead is to split the original matrix into row blocks before converting these into the ELLPACK format. In the resulting sliced ELLPACK format (SELL or SELL-C where C denotes the size of the row blocks [123, 127]), the overhead is no longer determined by the matrix row containing the largest number of nonzeros, but by the row with the largest number of nonzero elements in the respective block.

In [123], the SELL-C layout is enhanced with an a-priori sorting step, which aims to gather rows with a similar number of nonzeros into the same blocks. While this may improve the performance of SPMV, the impact on complex algorithms is still an open research topic. Another optimization, specific to streaming processors like GPUs, enforces a memory alignment within the SELL-P format that allows for applying the sophisticated matrix-vector kernel proposed in [102]. Although the padding that comes along with this format introduces some zero fill-in, a comparison between the SELL-P format visualized in Figure 3.6 and the plain ELLPACK in Figure 3.3 reveals that the blocking strategy may still render significant memory savings (see also Table 3.3), which directly translate into reduced computational cost and improved runtime performance. Previous experiments with this SELL-P format have shown high performance without impacting the algorithm's stability by applying row-sorting, so we will include it as an option for the GPU implementations. In particular, we apply a default configuration for SELL-P with the blocksize 8 and the row length padded to a multiple of 8.

Compared with the baseline (unoptimized) implementation of the CG solver, the optimized GPU codes consider the simple ELLPACK kernel for SPMV as well, but also include implementations for this operation based on the alternative formats ELLR-T and SELL-P. Furthermore, the optimized implementations do not invoke routines from CUBLAS, but instead reorganize and merge the vector operations in CG as described next.

3.4.2 Reformulating the CG algorithm for GPUs

The performance of memory-bound algorithms, like the CG solver, is determined by the volume and pattern of the memory accesses. On streaming processors, the number reads/writes to global memory usually correlates to the number of kernels launched over the runtime. While this is motivation enough to reduce the number of kernel calls, additional performance benefits may come through reduced CPU/GPU communication and a lower number of kernel launches. Here, we follow the ideas presented in [100] to adopt merged versions of the basic CG method. The resulting algorithm is illustrated in Figure 3.7, where SPMV denotes a generic implementation of the sparse matrix-vector kernel, which is replaced in the actual codes with one of the optimized cases described in the previous subsection.

All the tuned implementations of CG that we evaluate next are part of a pre-release extension of the MAGMA library [126] composed of iterative algorithms for sparse linear algebra. These codes block the CPU threads while the computation is proceeding in the GPU, so as to avoid a power-hungry polling during the execution. This is easily achieved by setting parameter `cudaDeviceBlockingSync` of the CUDA routine `cudaSetDeviceFlags`.

```
while( ( k < maxiter ) && ( res > epsilon ) ){
  SpMV <<<Gs,Bs>>> ( n, Asparse, d, z );
  tmp = cublasDdot ( n, d, 1, z, 1 );
  rho = beta / tmp;
  gamma = beta;
  cublasDaxpy (n,  rho, d, 1, x, 1 );
  cublasDaxpy (n, -rho, z, 1, r, 1 );
  tmp = cublasDdot ( n, r, 1, r, 1 );
  beta = tmp;
  alpha = beta / gamma;
  cublasDscal (n, alpha, d, 1 );
  cublasDaxpy (n, one, r, 1, d, 1 );
  res = sqrt( beta );
  k++;
} // end-while
```

```
while( ( k < maxiter ) && ( res > epsilon ) ){
  scalar_fusion_1 <<<Gs, Bs, Ms>>> ( n, Asparse, d, z,
                                     beta, rho, gamma, vtmp );
  fusion_2 ( Gs, Bs, Ms, beta, rho, vtmp );
  fusion_3 <<<Gs, Bs, Ms>>> ( n, rho, d, x, z, r, vtmp );
  fusion_4 ( Gs, Bs, Ms, vtmp, vtmp2 );
  fusion_5 <<<Gs, Bs>>> ( n, beta, gamma, alpha,
                                                d, r, vtmp );
  cudaMemcopy( &res, beta, sizeof(float),
                                     cudaMemcpyDeviceToHost );
  res = sqrt( res );
  k ++;
} // end-while
```

Figure 3.7: Aggregation of kernels to transform the basic implementation into a merged implementation. Here, the invocation to the sparse matrix-vector kernel SPMV has to be replaced by the appropriate GPU routine, depending on the sparse matrix format.

3.4.3 Search for the optimal configuration

Similarly to Section 3.3, we next analyze the runtime and energy efficiency of the algorithmically-optimized implementations of the CG method when configuring the hardware for either performance or energy efficiency by varying the CPU operation frequency and number of active cores. For completeness, we again provide the detailed information of the optimal configuration for each architecture and matrix combination (Tables 3.6 and 3.7), but in the following analysis we prefer referring to the graphical results. In particular, Figure 3.8 compares, analogously to Figure 3.5, the performance and energy efficiency of the architectures when configuring the hardware execution parameters either for runtime performance or energy efficiency.

At this point, we note that the algorithmic optimizations intrinsic to the data layouts BCSR and CSB, and the tuned implementations of SPMV embedded in the Intel MKL and CSB libraries, could not be

leveraged on A9, A15, and TIC, since these are not x86-based architectures and, therefore, it was not possible to use Intel's MKL, Cilk, or TBB. Hence, in Figure 3.5 we reproduce the data obtained with the basic implementations for these three architectures.

		Optimized w.r.t time					**Optimized w.r.t net energy**				
	Matrix	c	f	s	T	E_{net}	c	f	s	T	E_{net}
AIL	A159	8	2300	2	8.89E−02	1.48E+01	8	2100	2	9.46E−02	1.27E+01
	AUDI	8	2300	1(3)	7.90E−02	1.44E+01	8	2100	1(3)	8.08E−02	1.26E+01
	BMW	8	2300	1(3)	8.46E−03	1.50E+00	8	2100	1(3)	8.58E−03	1.34E+00
	CRANK	8	2300	2	1.29E−02	2.05E+00	8	2300	2	1.29E−02	2.05E+00
	F1	8	2300	1(3)	2.76E−02	4.46E+00	8	2100	1(3)	2.84E−02	4.42E+00
	INLINE	8	2300	1(3)	3.35E−02	5.73E+00	8	2100	1(3)	3.43E−02	5.37E+00
	LDOOR	8	2300	1(2)	5.40E−02	8.76E+00	8	2100	1(2)	5.49E−02	8.63E+00
AMC	A159	8	1500	2	1.69E−01	1.65E+01	8	1500	2	1.69E−01	1.65E+01
	AUDI	8	2000	1(3)	1.67E−01	1.63E+01	8	2000	1(3)	1.67E−01	1.63E+01
	BMW	8	2000	1(3)	1.44E−02	1.52E+00	8	2000	1(3)	1.44E−02	1.52E+00
	CRANK	8	2000	0	2.35E−02	2.56E+00	8	2000	0	2.35E−02	2.56E+00
	F1	8	2000	1(3)	5.59E−02	5.58E+00	8	2000	1(3)	5.59E−02	5.58E+00
	INLINE	8	2000	1(3)	6.48E−02	6.61E+00	8	2000	1(3)	6.48E−02	6.61E+00
	LDOOR	8	2000	1(2)	9.59E−02	9.92E+00	8	2000	1(2)	9.59E−02	9.92E+00
IAT	A159	2	2000	0	2.88E−01	1.33E+00	1	600	2	1.56E+00	7.71E−01
	AUDI	2	2000	0	3.20E−01	1.41E+00	1	600	1(3)	1.64E+00	7.65E−01
	BMW	2	2000	0	3.65E−02	1.83E−01	1	600	1(3)	2.16E−01	1.14E−01
	CRANK	2	2000	1(2)	4.70E−02	2.16E−01	1	600	2	5.40E−01	1.02E−01
	F1	2	2000	1(3)	1.55E−01	5.27E−01	1	600	1(3)	7.17E−01	3.28E−01
	INLINE	2	2000	0	1.50E−01	6.90E−01	1	600	2	1.48E+00	3.73E−01
	LDOOR	2	2000	1(2)	1.92E−01	8.54E−01	1	600	1(2)	8.93E−01	6.18E−01
INH	A159	8	1870	0	7.04E−02	1.03E+01	4	1600	0	8.71E−02	8.38E+00
	AUDI	8	2000	1(3)	6.86E−02	9.82E+00	8	1600	1(3)	6.91E−02	8.57E+00
	BMW	4	2000	1(3)	8.88E−03	9.48E−01	4	1600	1(3)	9.14E−03	8.88E−01
	CRANK	4	1870	1(2)	1.53E−02	1.63E+00	4	1600	1(2)	1.59E−02	1.54E+00
	F1	8	1870	1(3)	2.53E−02	3.62E+00	8	1600	1(3)	2.57E−02	3.30E+00
	INLINE	8	2000	1(3)	3.05E−02	4.35E+00	4	1600	1(3)	3.72E−02	3.61E+00
	LDOOR	8	1870	1(2)	5.46E−02	7.74E+00	4	1600	0	6.56E−02	6.37E+00
ISB	A159	6	2000	0	3.65E−02	2.15E+00	4	1200	0	5.69E−02	1.68E+00
	AUDI	6	2000	1(3)	4.04E−02	2.08E+00	2	1200	1(3)	9.65E−02	1.62E+00
	BMW	6	2000	1(3)	3.57E−03	2.13E−01	6	1200	1(3)	5.15E−03	1.71E−01
	CRANK	6	2000	0	6.45E−03	4.23E−01	6	1200	1(2)	9.67E−03	3.11E−01
	F1	6	2000	1(3)	1.35E−02	6.92E−01	6	1200	1(3)	1.97E−02	5.68E−01
	INLINE	6	2000	1(3)	1.54E−02	8.51E−01	4	1200	1(3)	2.58E−02	6.96E−01
	LDOOR	6	2000	1(2)	2.38E−02	1.40E+00	6	1200	1(2)	3.40E−02	1.16E+00

Table 3.6: Optimal hardware parameter configuration when optimizing the general-purpose architectures for runtime or energy efficiency using the tuned implementations. In the labels, c denotes the number of cores, f the frequency (in MHz), s the sparse matrix layout/SPMV implementation (0 for CSR+MKL, 1(2) for BCSR+MKL with block size `1b`=2, 1(3) for BCSR+MKL with block size `1b`=3, and 2 for CSB), T the time per iteration (in seconds), and E_{net} the net energy per iteration (in Joules).

3.4.3.1 Optimization with respect to time

When comparing the runtime performance of the optimized implementations on the different architectures, the high-end GPUs KEP and FER are the overall winners, independently of whether the hardware execution

		Optimized w.r.t time					Optimized w.r.t net energy				
	Matrix	c	f	s	T	E_{net}	c	f	s	T	E_{net}
FER	A159	1	2270	1	8.22E−03	1.57E+00	1	1600	1	8.24E−03	1.49E+00
	AUDI	1	1600	3	1.58E−02	2.95E+00	1	1600	3	1.58E−02	2.95E+00
	BMW	1	1600	3	2.06E−03	3.73E−01	1	1600	3	2.06−03	3.73E−01
	CRANK	1	1600	3	2.43E−03	4.50E−01	1	1600	3	2.43−03	4.50E−01
	F1	1	1600	3	6.21E−03	1.16E+00	1	1600	3	6.21E−03	1.16E+00
	LDOOR	1	1600	3	9.55E−03	1.75E+00	1	1600	3	9.55E−03	1.75E+00
KEP	A159	1	3200	1	5.29E−03	7.45E−01	1	1200	1	5.30E−03	5.89E−01
	AUDI	1	3200	3	9.37E−03	1.39E+00	1	1200	3	9.38E−03	1.13E+00
	BMW	1	3200	3	1.28E−03	1.82E−01	1	1200	3	1.29E−03	1.47E−01
	CRANK	1	3200	3	1.52E−03	2.18E−01	1	1200	3	1.53E−03	1.76E−01
	F1	1	3200	3	3.63E−03	5.32E−01	1	1200	3	3.64E−03	4.31E−01
	INLINE	1	3200	3	4.69E−03	6.58E−01	1	1200	3	4.71E−03	5.55E−01
	LDOOR	1	3200	3	5.61E−03	8.21E−01	1	1200	3	5.62E−03	6.64E−01
QDR	A159	1	1300	1	3.60E−02	1.01E+00	1	51	1	3.92E−02	5.96E−01
	BMW	1	1300	3	1.03E−02	2.76E−01	1	1300	3	1.03E−02	2.76E−01
	CRANK	1	1300	3	1.18E−02	3.31E−01	1	1300	3	1.18E−02	3.31E−01
	F1	1	1300	3	3.05E−02	8.59E−01	1	51	3	4.05E−02	6.77E−01
	INLINE	1	1300	0	3.42E−01	9.59E+00	1	51	0	3.46E−01	7.73E+00

Table 3.7: Optimal hardware parameter configuration when optimizing the specialized architectures for runtime or energy efficiency using the tuned implementations. In the labels, c denotes the number of cores, f the frequency (in MHz), s the sparse matrix layout/SPMV implementation (0 for CSR, 1 for ELLPACK, 2 for ELLR-T, and 3 for SELL-P), T the time per iteration (in seconds), and E_{net} the net energy per iteration (in Joules).

parameters are tuned for performance or energy efficiency (see the left top and bottom plots in Figure 3.8, respectively). For example, the recent CPU generations —AIL and ISB— are outperformed by NVIDIA Tesla K20 (KEP) by factors up to 10, and the reduced runtime of the GPU implementations directly translates into outstanding energy efficiency. From the energy perspective, all GPU architectures are competitive with the low-power processors IAT, A9, A15, and the Intel Sandy Bridge CPU ISB (see right-top graph in Figure 3.8). Only the TIC achieves about twice the energy efficiency. The most inefficient architectures from this point of view are, like in the non-optimized case, the older general-purpose CPUs AIL, AMC, and INH.

3.4.3.2 Optimization with respect to net energy

As already observed for the basic implementation, modifying the hardware execution parameters for the GPU systems has only negligible impact on performance (compare left top and bottom plots in Figure 3.8). This is different for ISB which outperforms QDR only when aiming for performance. Like in the non-optimized case, improving the energy demand of the CPUs and the low-power architectures comes at the price of a certain level of performance loss. As we could not apply algorithmic optimizations for the latter architectures (A9, A15, TIC), it is not suprising that they again deliver the lowest GFLOPS rates; despite that, A15 and TIC are the overall winners when aiming for low energy demands. Even the algorithmically-optimized CG implementation on the top-end GPUs achieves energy efficiency rates far from those of A15. The IAT and TIC are competitive, while the ISB achieves ratios similar to those of the older A9 and QDR. KEP has a slightly higher, and FER a slightly lower, energy efficiency than Intel's latest CPU generation (ISB), while the AMD CPUs (AIL and AMC) stay, like in the performance-oriented configuration, far behind (see right-bottom plot in Figure 3.8). In contrast to the GPUs, the outstanding energy efficiency of the low-power

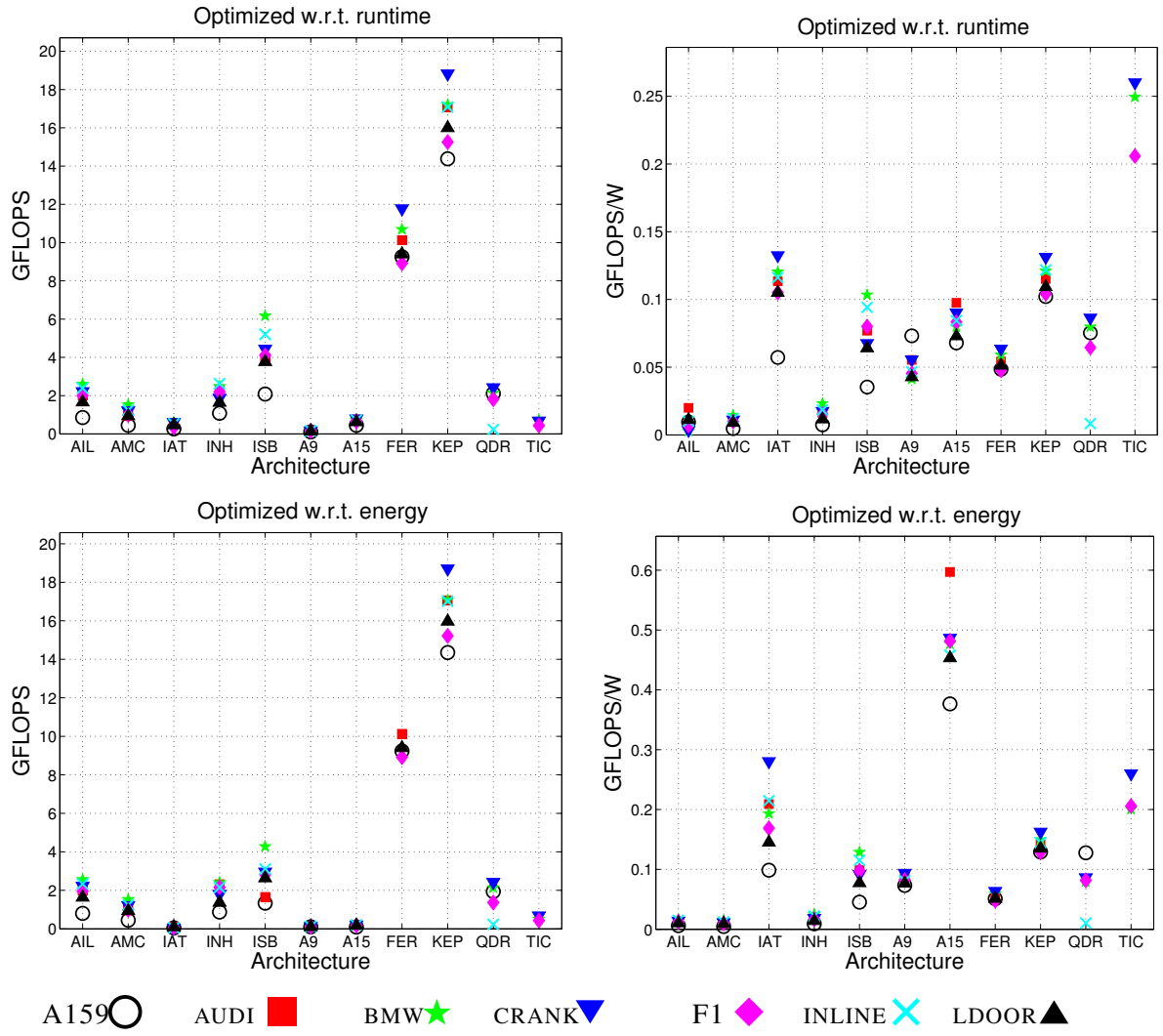


Figure 3.8: Comparison of performance (left) and energy efficiency (right), measured, respectively, in terms of GFLOPS and GFLOPS/W, when optimizing the tuned CG implementations with respect to run time (top) or net energy (bottom).

architectures again comes at the cost of increased runtime, such that KEP outperforms A15 by almost two orders of magnitude.

To summarize this initial study (search for the optimal configuration), next we rank the systems according to their performance and energy efficiency. In order to do this, for each architecture we employ the GFLOPS and GFLOPS/W rates, averaged for all matrix benchmarks, when optimizing for time and for energy efficiency. The purpose and effect of using average values is to collapse the results for all the matrix cases into a single figure-of-merit, which allows an easier (albeit less precise) comparison of the architectures. Table 3.8 shows the results of these metrics, normalized with respect to the fastest architecture when optimizing for time (KEP), and with respect to the most energy-efficient one when optimizing for energy (A15). The results in the second and third columns of the table show that, when optimizing for performance, on average KEP is roughly half an order of magnitude faster than FER (6.04E−01), around one order of magnitude faster than QDR (1.05E−01), and slightly more than two orders of magnitude faster than A9 (9.24E−03). If the goal is runtime performance, KEP consumes less energy than all the other architectures

except TIC. On the other hand, the last two columns of the table compare the energy efficiency of A15 with that of its competitors. Here, the differences (obviously in favor of A15 in all cases) are smaller, and range between close to half an order of magnitude for TIC (4.67E−01) and more than two orders of magnitude for AMC (2.13E−02).

	Optimized w.r.t time		**Optimized w.r.t net energy**	
	Norm. avg. GFLOPS	Norm. avg. GFLOPS/W	Norm. avg. GFLOPS	Norm. avg. GFLOPS/W
AIL	1.18E-01	7.95E-02	1.06E+01	2.61E-02
AMC	6.31E-02	8.84E-02	5.77E+00	2.13E-02
IAT	2.87E-02	9.31E-01	4.20E-01	3.93E-01
INH	1.22E-01	1.36E-01	1.03E+01	3.73E-02
ISB	2.56E-01	6.48E-01	1.48E+01	1.96E-01
A9	9.24E-03	4.47E-01	6.42E-01	1.73E-01
A15	4.06E-02	7.15E-01	1.00E+00	**1.00E+00**
FER	6.05E-01	4.70E-01	3.46E+01	1.14E-01
KEP	**1.00E+00**	1.00E+00	9.10E+01	2.97E-01
QDR	1.05E-01	5.46E-01	8.94E+00	1.62E-01
TIC	3.70E-02	2.07E+00	2.99E+00	4.67E-01

Table 3.8: Average GFLOPS and GFLOPS/W rates of the DP tuned implementations normalized with respect to the "fastest" (highest GFLOPS rate when optimizing w.r.t. time) and "most energy-efficient" (highest GFLOPS/W rate when optimizing w.r.t. energy) architectures, KEP and A15 respectively.

		Optimized w.r.t. time		**Optimized w.r.t. net energy**	
	Matrix	Speed-up in T	Speed-up in E_{net}	Speed-up in T	Speed-up in E_{net}
IAT	A159	1.04	0.95	0.80	1.10
	AUDI	1.18	1.04	1.12	1.25
	BMW	1.19	1.06	1.12	1.20
	CRANK	1.23	1.12	0.56	1.40
	F1	1.14	1.16	1.13	1.20
	INLINE	1.16	1.04	0.61	1.13
	LDOOR	1.23	1.12	1.33	1.06
	AVERAGE	1.17	1.07	0.95	1.19
ISB	A159	1.52	1.29	1.89	1.10
	AUDI	1.51	1.49	1.34	1.51
	BMW	1.54	1.52	1.58	1.53
	CRANK	1.11	1.07	1.14	1.10
	F1	1.46	1.50	1.48	1.49
	INLINE	1.50	1.49	1.36	1.50
	LDOOR	1.28	1.25	1.33	1.25
	AVERAGE	1.42	1.37	1.45	1.36
KEP	A159	1.28	0.90	1.28	1.14
	AUDI	3.76	2.52	3.76	3.10
	BMW	4.35	2.95	4.31	3.65
	CRANK	14.86	9.49	14.77	11.76
	F1	4.62	3.08	4.61	3.80
	LDOOR	1.52	1.11	1.52	1.37
	AVERAGE	5.07	3.34	5.04	4.14

Table 3.9: Speed-ups resulting from the algorithmic optimizations.

3.4.4 Complementary analyses

In the remainder of this section, we elaborate on two additional studies of the impact of algorithmic optimization and the advantages of SP arithmetic, on IAT, ISB, and KEP. We selected these three systems because they all correspond to recent processor architectures and each one represents a different class of the systems included in this study (low-power architectures, server-oriented general-purpose multicore processors, and manycore GPUs, respectively). Finally, we chose IAT instead of the more appealing TIC and A15 because, as noted earlier, we could not apply algorithmic optimizations on the latter two.

3.4.4.1 Algorithmic optimization potential

Table 3.9 assesses the impact of the algorithmic optimizations described at the beginning of this section. A first observation from this table is that significant performance improvements can be achieved through algorithmic optimizations of the implementations. This is especially true for KEP, where we replaced the standard ELLPACK-based implementation of SPMV with a format tailored to each matrix case, and substituted the CUBLAS-based implementation of the CG method with a version using algorithm-specific kernels. At the same time, the improvement potential on GPUs is again very dependent on the matrix characteristics, as we reduce runtime by "only" 28% for the A159 case, but in a factor larger than 14× for CRANK. An explanation for this effect can be found in Table 3.3, showing a drastically reduced storage (and computational) overhead when we replace the ELLPACK with the SELLP kernel for the CRANK test case, while the improvement for the A159, where we keep the ELLPACK format (see Table 3.7), comes exclusively from enhancing the implementation using algorithm-specific kernels. On average, Table 3.9 reveals that replacing the basic CG implementation with an optimized variant improves performance on KEP by a factor of 5.07×. For the CPU based architectures, the reported speedups are lower on average, but are, on the other hand, more consistent. On ISB, for example, the optimization techniques render an average performance improvement of 1.42×, with 1.11×, and 1.54× as upper and lower bounds, respectively; and similar variations when optimizing for energy efficiency. On the low-power architecture IAT, the improvements are slightly lower.

3.4.5 Single precision performance through iterative refinement

It is well-known that iterative refinement, in combination with mixed precision, can render the benefits of a faster SP arithmetic while still delivering DP accuracy for some applications [119]. In particular, while the use of DP arithmetic is in general mandatory for the solution of sparse linear systems, in [101], for instance, it is shown how the use of mixed SP-DP arithmetic and iterative refinement leads to improved execution time and energy consumption for GPU-accelerated solver-platforms. Our last experiment thus aims to evaluate the performance and energy efficiency variations that can be expected when embedding a CG solver, which performs its arithmetic in SP instead of DP, into the mixed precision iterative refinement framework.

Table 3.10 reports the results of this test, comparing the performance and energy efficiency of SP and DP implementations of the optimized CG solvers in terms of their respective speed-ups. In this case, we do not include results for IAT, as the CSB library does not provide the necessary SP implementation of SPMV and, therefore, the collection of matrix cases that could be reported for IAT is limited (on this platform, the CSB solution is optimal for A159, CRANK, and INLINE; see Table 3.6). Furthermore, we found out that, for this particular architecture, in many cases the optimal configuration when operating with DP data differed from that identified for SP arithmetic. On average the variations are quite independent of the target platform, revealing acceleration factors on performance, when optimizing w.r.t. time, that vary between 1.35× (KEP) and 1.60× (ISB). Similar numbers are observed when the optimization is w.r.t. energy. It is, however, interesting to notice that the usage of SP arithmetic is more beneficial to performance on the CPU-based

systems, while it renders larger improvement to the energy efficiency on the KEP GPU. If the hardware execution parameters are configured for energy efficiency, the same effect occurs at a larger scale. This can be explained by the fact that the algorithm is memory-bound, and decreasing frequency (and voltage) has, in general, more effect on the core clock than on the memory clock rate.

		Optimized w.r.t time		**Optimized w.r.t net energy**	
		Speed-up in T	Speed-up in E_{net}	Speed-up in T	Speed-up in E_{net}
ISB	A159	1.72	1.54	1.81	1.54
	AUDI	1.53	1.40	2.34	1.38
	BMW	1.64	1.53	1.51	1.54
	CRANK	1.47	1.54	1.41	1.43
	F1	1.59	1.42	1.48	1.50
	INLINE	1.64	1.48	1.75	1.54
	LDOOR	1.60	1.47	1.47	1.53
	AVERAGE	1.60	1.48	1.68	1.49
KEP	A159	1.52	1.70	1.52	1.72
	AUDI	1.34	1.47	1.34	1.49
	BMW	1.30	1.45	1.29	1.47
	CRANK	1.39	1.51	1.39	1.51
	F1	1.31	1.45	1.31	1.48
	INLINE	1.32	1.40	1.33	1.48
	LDOOR	1.28	1.43	1.28	1.44
	AVERAGE	1.35	1.49	1.35	1.51

Table 3.10: Speed-ups resulting from the use of the SP tuned implementations instead of the DP versions.

3.5 Summary and Future Work

Current processor architectures take different roads toward delivering raw performance and energy efficiency: in the form of low-power, general-purpose multicore designs and digital signal processors (DSPs) for mobile and embedded appliances; power-hungrier, but more versatile, general-purpose multicore architectures for servers; and hardware accelerators like manycore graphics processors (GPUs) or the Intel Xeon Phi. In this paper, we have provided a broad overview about the potential of representative samples of these three different approaches, assessing their performance and energy efficiency using basic and advanced implementations of a crucial numerical algorithm: the iterative CG method. We observed that the flops-per-watt rate of manycore systems, like the GPUs from NVIDIA, can be matched by low-power devices such as the Intel Atom, the ARM A9, or a DSP from Texas Instruments; and clearly outperformed by the more recent ARM A15. While GPUs traditionally deliver high energy efficiency with outstanding performance, the less energy hungry architectures provide it less cores, a lower power dissipation and/or smaller memories. This reduces the suitability of these inexpensive architectures for general-purpose computing, but makes them appealing candidates for mobile and embedded scenarios (their original target) as well as specific applications. Despite the fact that conventional general-purpose processors attained neither the performance of the GPUs nor the performance-per-watt rates of the low-power alternatives, they provide an interesting balance between these two extremes.

Future research will increase the scope of the study by adding problems not directly related to sparse linear algebra, e.g., the seven dwarfs [104], as well as more complex scientific applications.

Acknowledgments

The authors from Universitat Jaume I were supported by the CICYT project TIN2011-23283 and FEDER, and by the EU FET FP7 project 318793 “EXA2GREEN”.

We thank Francisco D. Igual, from *Universidad Complutense de Madrid*, for his help with the low-power architectures evaluated in this work.

References

[98] The Top500 list, 2014. `http://www.top500.org`.

[99] J. Aliaga, H. Anzt, M. Castillo, J. Fernández, G. Léon, J. Pérez, and E. S. Quintana-Ortí. Performance and energy analysis of the iterative solution of sparse linear systems on multicore and manycore architectures. In *Lecture Notes in Computer Science, 10th Int. Conf. on Parallel Processing and Applied Mathematics – PPAM 2013*, 2014.

[100] J. I. Aliaga, J. Pérez, E. S. Quintana-Ortí, and H. Anzt. Reformulated Conjugate Gradient for the energy-aware solution of linear systems on GPUs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 320–329, Oct 2013.

[101] H. Anzt, V. Heuveline, J. I. Aliaga, M. Castillo, J. C. Fernández, R. Mayo, and E. S. Quintana-Ortí. Analysis and optimization of power consumption in the iterative solution of sparse linear systems on multi-core and many-core platforms. In *Int. Green Computing Conf. Workshops (IGCC)*, pages 1–6, 2011.

[102] H. Anzt, S. Tomov, and J. Dongarra. Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs. Technical Report ut-eecs-14-727, University of Tennessee, March 2014.

[103] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, University of California at Berkeley, Electrical Engineering and Computer Sciences, 2006.

[104] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[105] S. Ashby et al. The opportunities and challenges of Exascale computing. Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, November 2010.

[106] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[107] Costas Bekas and Alessandro Curioni. A new energy aware performance metric. *Computer Science - Research and Development*, 25:187–195, 2010. 10.1007/s00450-010-0119-z.

[108] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corp., December 2008.

[109] A. Buluç et al. *Compressed Sparse Block Library*. Combinatorial Scientific Computing Lab at the University of California, Santa Barbara, 2014.

[110] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 233–244, New York, NY, USA, 2009. ACM.

[111] A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proc. IPDPS*, pages 721–733, 2011.

[112] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, Oct 1974.

[113] J. Dongarra et al. The international ExaScale software project roadmap. *Int. J. of High Performance Computing & Applications*, 25(1), 2011.

[114] J. Dongarra and M. A. Heroux. Toward a new metric for ranking high performance computing systems. Sandia Report SAND2013-4744, Sandia National Laboratories, June 2013.

[115] M. Duranton *et al*. The HiPEAC vision for advanced computing in horizon 2020, 2013.

[116] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer architecture*, ISCA'11, pages 365–376, 2011.

[117] S. H. Fuller and L. I. Millett. *The Future of Computing Performance: Game Over or Next Level?* National Research Council of the National Academies, 2011.

[118] The Green500 list, 2014. `http://www.green500.org`.

[119] N. J. Higham. *Accuracy and Stability of Numerical Algorithms: Second Edition*. Society for Industrial and Applied Mathematics, 2002.

[120] E. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, Feb. 2004.

[121] Intel. *Intel Math Kernel Library reference manual (release 10.3)*, 2014. Document Number: 630813-053US.

[122] P. Kogge et al. ExaScale computing study: Technology challenges in achieving ExaScale systems, 2008.

[123] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop. A unified sparse matrix data format for modern processors with wide SIMD units. *CoRR*, abs/1307.6209, 2013.

[124] A.N. Langville and C.D. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, 2009.

[125] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, Sept. 1979.

[126] MAGMA project home page. `http://icl.cs.utk.edu/magma/`.

[127] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, pages 111–125, Berlin, Heidelberg, 2010. Springer-Verlag.

[128] Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical Report Ottawa, Ontario, Canada, 1966.

[129] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[130] F. Vázquez, J. J. Fernández, and E. M. Garzón. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience*, 23(8):815–826, 2011.

[131] F. Vázquez, J. J. Fernández, and E. M. Garzón. Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach. *Parallel Computing*, 38(8):408 – 420, 2012.

[132] R. Vuduc. *Automatic performance tuning of sparse matrix kernels*. Ph.D. dissertation, Univ. California, Berkeley, January 2004.

[133] S. Williams, N. Bell, J. Choi, M. Garland, L. Oliker, and R. Vuduc. Sparse matrix vector multiplication on multicore and accelerator systems. In Jakub Kurzak, David A. Bader, and Jack Dongarra, editors, *Scientific Computing with Multicore Processors and Accelerators*. CRC Press, 2010.

CAPÍTULO 4

Reformulated Conjugate Gradient for the Energy-Aware Solution of Linear Systems on GPUs

J. I. Aliaga, H. Anzt, **J. Pérez**, *E. S. Quintana-Ortí.*
"Reformulated Conjugate Gradient for the energy-aware solution of linear systems on GPUs".
42nd International Conference on Parallel Processing – ICPP 2013, pp. 320-329. Lyon (Francia), 2013.
ISBN: 0190-3918/13.

Abstract

In this paper we introduce a redesign of the conjugate gradient method for the iterative solution of sparse linear systems on heterogeneous systems accelerated by graphics processing units (GPUs). Reshaping the GPU kernels induced by the classical formulation of the CG method into algorithm-specific routines results in a slight increase of performance. While these improvements could already justify the reformulation of the method, the actual motivation is the exploitation of power-saving techniques implicit in the hardware, like the processor C-states, which can be leveraged more efficiently in the new algorithm. Numerical experiments using data matrices from a popular sparse matrix collection show that the time overhead naturally associated with the application of these energy-aware techniques is no longer crucial to the overall runtime performance.

4.1 Introduction

In solving complex scientific problems using high performance computing (HPC) resources, the total energy demand is increasingly becoming a critical factor. While the high computational cost of the methods underlying these applications is often tackled by accessing powerful hardware, the consumed energy may pose economical and ecological limits to the simulations that can be conducted in HPC facilities [143, 145, 146]. To face the challenges of a more resource-friendly computing era, continued efforts from hardware manufacturers are resulting in the development of energy-efficient hardware components featuring fruitful power-saving mechanisms. However, the energy-aware optimization of numerical software

is still far behind [136], with one particular challenge being how to efficiently leverage the power-saving mechanisms currently available in the hardware without sacrificing application's performance.

HPC systems equipped with coprocessors have recently gained wide acceptance in order to accelerate computationally intensive parts of complex scientific applications [135], showing also an excellent performance-per-watt ratio [134]. For the solution of sparse linear systems using graphics processing units (GPUs), in [139, 140] we demonstrated how the energy footprint of the implementations can be reduced by applying dynamic voltage and frequency control (DVFS [149]) as well as preventing the CPU from entering polling loops via idle-wait. Similar conclusions were extracted in [137] for the heterogeneous CPU-GPU solution of dense linear systems via matrix factorizations. However, these solvers suffer from the overhead induced by the idle-wait technique, which blocks the CPU while computing on the GPU, resulting in a negative impact on runtime (because of the delay to reactivate the processor) and, consequently, affecting the energy balance.

In this paper we show that the reformulation of the numerical methods is an essential step to overcome this problem. In particular, we propose a redesign of the traditional algorithm for the conjugate gradient (CG) method that renders a GPU implementation with *merged* NVIDIA's CUDA kernels. The effect of this fusion is that the CPU remains blocked for longer periods of time, while the computation proceeds on the GPU, and thus the solver can better leverage the reduced power consumption of inactive C-states [148] via idle-wait, resulting in both competitive runtime and remarkable reduction of energy.

The rest of the paper is structured as follows. In Sections 4.2 and 4.3 we first introduce the CG method, an iterate procedure for solving symmetric positive definite (SPD) linear systems, and we then review a few details on the benchmark matrix collection and the hardware setup that will be used for the runtime and energy analysis. Section 4.4 offers a short introduction to two GPU implementations of the sparse matrix-vector product, using the popular compressed storage row (CSR) format [144], and describes our initial GPU code for the CG method built on top of this operation and NVIDIA's CUBLAS library [150]. While there exist alternative algorithms and storage layouts for the sparse matrix-vector product that, in general, render higher performance on the GPU [141, 155], we note here that our energy-saving techniques are orthogonal to the actual implementation of the matrix-vector product kernel. Indeed, we expect that a more efficient implementation of this operation will shift part of the relative cost towards other kernels of the CG method, so that by Amdahl's law, the gains could be even higher.

The main contribution of this paper follows in Section 4.4.2. There, we initially review the application of idle-wait that keeps the CPU from entering a power-hungry polling loop, trading off power dissipation for performance. To avoid this effect, we next reformulate the CG method, deriving several new algorithm-specific CUDA kernels that, in principle, should attain a more efficient hardware usage. The experiments in that section demonstrate that these new kernels render a significant reduction of the overall energy consumption by leveraging the power-saving techniques available while, simultaneously, retain the runtime performance of the polling state. Finally, in Section 4.5 we provide a few concluding remarks and discuss possible directions of further research.

4.2 Iterative Solution of Sparse Linear Systems

The CG method is one of the best known iterative algorithms for the numerical solution of SPD linear systems [153] that arise, e.g., when tackling partial differential equations numerically via finite element or finite difference methods [142]. Given the linear system $Ax = b$, where $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$, and $x \in \mathbb{R}^n$ is the sought-after solution, the algorithm generates a Krylov subspace for the pair (r,A), with the residual $r = Ax - b \in \mathbb{R}^n$. A detailed derivation of the CG method can be found in [154], including a broad convergence analysis. A mathematical description is given in Figure 4.1. There, the user-defined parameters ε

1: $x_0 := 0$
2: $r_0 := b - Ax_0$
3: $d_0 := r_0$
4: $\beta_0 := r_0^T r_0$
5: $k := 0$
6: **while** $(k < maxiter)$ & $(res > \varepsilon)$
7: $\quad z_k := Ad_k$ (SpMV)
8: $\quad \rho_k := \frac{\beta_k}{d_k^T z_k}$ (dot)
9: $\quad \gamma_k := \beta_k$
10: $\quad x_{k+1} := x_k + \rho_k d_k$ (saxpy)
11: $\quad r_{k+1} := r_k - \rho_k z_k$ (saxpy)
12: $\quad \beta_{k+1} := r_{k+1}^T r_{k+1}$ (dot)
13: $\quad \alpha_k := \frac{\beta_{k+1}}{\gamma_k}$
14: $\quad d_{k+1} := r_{k+1} + \alpha_k d_k$ (scal+saxpy)
15: $\quad res := \| r_{k+1} \|_2$
16: $\quad k := k + 1$
17: **end**

Figure 4.1: Algorithmic description of the CG method.

and *maxiter* set upper bounds, respectively, on the relative residual for the computed approximation to the solution x_k, and the maximum number of iterations.

Concerning the computational effort of the CG algorithm, in practical applications the cost of the iteration loop is dominated by the matrix-vector multiplication $z_k := Ad_k$ (line 7). Given a sparse matrix A with n_z nonzero entries, the cost of this operation is roughly $2n_z$ floating-point arithmetic operations (flops). Additionally, the loop body contains several vector operations that require $O(n)$ flops each.

4.3 Environment Setup

4.3.1 Test matrices

In our experiments with the CG method, the SPD matrices were chosen either from the University of Florida Matrix Collection (UFMC)[1], corresponding to finite element discretizations of several structural problems arising in mechanics, or derived from a finite difference discretization of the 3D Laplace problem. A few key parameters from these matrices are collected in Table 4.1 while sparsity plots are given in Figure 4.2. In all cases, the vector of independent terms b was initialized so that all the entries of the solution vector x were equal to 1, and the CG iteration was started with the initial guess $x_0 \equiv 0$.

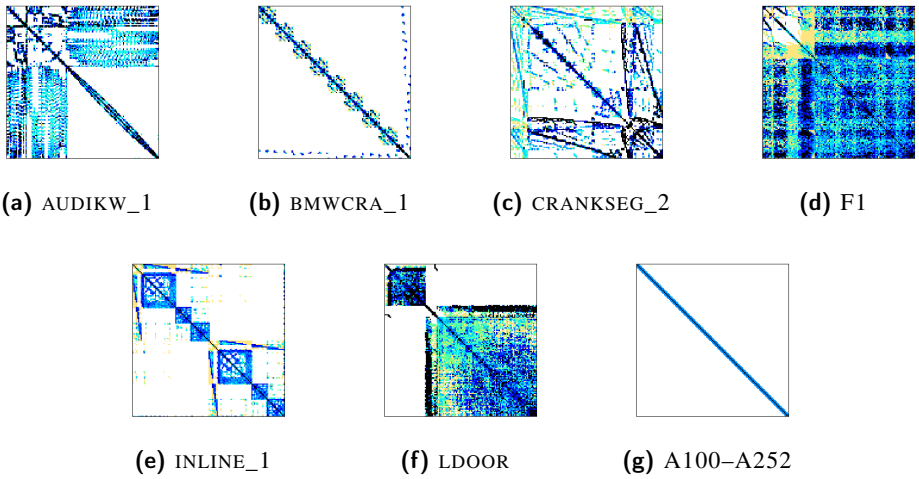
4.3.2 Hardware platform and arithmetic

The experimental evaluation was performed using IEEE-754 single precision (SP) arithmetic on a platform equipped with an Intel Core i7-3770K processor (4 cores at 3.5 GHz) and 16 GB of RAM, connected to an NVIDIA GeForce GTX480 GPU (1.4 GHz, 1.5 GB). The solvers employed CUDA 4.0 and, in case it was necessary, the legacy CUBLAS API (application programming interface).

[1] UFMC; see http://www.cise.ufl.edu/research/ sparse/matrices/

Source	Matrix	#nonzeros (n_z)	Size (n)	n_z/n
UFMC	AUDIKW_1	77,651,847	943,645	82.28
	BMWCRA1	10,641,602	148,770	71.53
	CRANKSEG_2	14,148,858	63,838	221.63
	F1	26,837,113	343,791	78.06
	INLINE_1	38,816,170	503,712	77.06
	LDOOR	42,493,817	952,203	44.62
Laplace	A100	6,940,000	1,000,000	6.94
	A126	13,907,370	2,000,376	6.94
	A159	27,986,067	4,019,679	6.94
	A200	55,760,000	8,000,000	6.94
	A252	111,640,032	16,003,001	6.94

Table 4.1: Description and properties of the test matrices.



(a) AUDIKW_1 **(b)** BMWCRA_1 **(c)** CRANKSEG_2 **(d)** F1

(e) INLINE_1 **(f)** LDOOR **(g)** A100–A252

Figure 4.2: Sparsity plots of the test matrices.

While the use of double precision (DP) arithmetic is in general mandatory for the solution of sparse linear systems of equations, in [140] we show how the use of mixed SP-DP, in combination with iterative refinement, leads to improved execution time and energy consumption when the target platform is a GPU accelerator. The computational key of the mixed precision implementation lies in the efficient SP solution of sparse linear systems, with the DP portion of the code having a negligible effect on both execution time and energy.

Power consumption was measured using an internal DC powermeter with a sampling frequency of 505 Hz. This device was directly attached to the 12V wires that connect the computer power supply unit with the mainboard and the GPU. A daemon application was ran on a separate tracing server, collecting power samples from the powermeter.

Unless otherwise stated, in all our experiments we executed the implementation of the CG method till $res < \varepsilon = 10^{-5}\|r\|_2$ or the number of iterations exceeded *maxiter*=1,000. In the results, we report average time (in milliseconds, ms) and energy (in Joules, J) per iteration of the solver.

4.4 CUBLAS Implementation of the CG method

In our work, we followed the ideas in [141] for the implementation of the sparse matrix-vector product (SPMV), using the popular and general-purpose CSR format. In particular, we implemented the original *scalar kernel* from [141] and a variant of the *vector kernel* that employs a 2D grid in order to operate on sparse matrices with more than 524,280 rows. Hereafter, we refer to these two kernels as SSPMV and VSPMV, respectively. On top of these kernels, we constructed two classical CUBLAS-based implementations of the CG method that simply invoke functions from the CUBLAS library [150] for the dot products, the vector scaling, and the saxpy operations (`cublasSdot`, `cublasSscal` and `cublasSaxpy`, respectively); see Figure 4.3. In the following, we will refer to these codes as the *CG-CUBLAS* implementations.

```
1  while( ( k < maxiter ) && ( res > epsilon ) ){
2    SSpMV <<<Gs,Bs>>> ( n, rowA, colA, valA, d, z ); // z := A * d
3    tmp = cublasSdot ( n, d, 1, z, 1 );              // tmp := d' * z
4    rho = beta / tmp;                                // rho := beta / tmp
5    gamma = beta;                                    // gamma := beta
6    cublasSaxpy (n,  rho, d, 1, x, 1 );              // x := x + rho * d
7    cublasSaxpy (n, -rho, z, 1, r, 1 );              // r := r - rho * z
8    beta = cublasSdot( n, r, 1, r, 1 );              // beta := r' * r
9    alpha = beta / gamma;                            // alpha := beta / gamma
10   cublasSscal (n, alpha, d, 1 );                   // d := alpha * d
11   cublasSaxpy (n, one, r, 1, d, 1 );               // d := d + r
12   res = sqrt( beta );                              // res := sqrt(beta)
13   k++;
14 } // end-while
```

Figure 4.3: Simplified fragment of the CG-CUBLAS implementation based on the scalar kernel SSPMV. The implementation based on the vector kernel only differs in that routine VSPMV (with the same parameters) is invoked instead of SSPMV.

In the first experiment we analyze the performance of the CG-CUBLAS implementations where either the scalar kernel or the vector kernel encoded in CUDA is used for SPMV. From the results in Table 4.2, we deduce that SSPMV offers higher efficiency for the Laplacian matrices while, for all other cases, switching to VSPMV results in significant performance improvements (compare the columns labelled as "CG"). In the table, we also report the runtime of the corresponding matrix-vector multiplication kernel (column "SPMV") and the ratio that the cost of this operation represents w.r.t. the total time (column "%"). We observe that, for the set of matrices chosen from UFMC, the average contribution of the matrix-vector product to the total runtime is roughly 94%. For the Laplacian matrices, the number of nonzero elements per row is significantly lower and, as could be expected, the SPMV is less dominant in the CG solver, accounting for only about 82% of the time on average. Note also that the dominance increases with the matrix size. This outcome agrees with the conclusions in [141] (for a variety of 14 test cases), which indicate that, in general, the vector kernel excels over the scalar kernel for matrices with a large number of nonzeros per row. Given these results, in the remainder of this paper, we will only utilize the optimal kernel (scalar or vector) for the sparse matrix-vector product depending on the test case.

4.4.1 Using CUDA blocking mode for energy efficiency

The execution of a CUDA typical program interleaves fragments of code that are executed by a CPU thread in the host, and CUDA kernels that, invoked from the CPU thread, off-load a certain part of the computation to the GPU. Some of the routines in CUBLAS are asynchronous so that, in principle, the CPU thread can continue in execution right after their invocation. However, due to the data dependencies between

Matrix	SSpMV			VSpMV		
	SpMV	CG	%	SpMV	CG	%
AUDIKW_1	135.63	135.98	99.7	8.83	9.22	95.5
BMWCRA1	17.97	18.07	99.4	1.24	1.34	91.7
CRANKSEG_2	24.82	24.87	99.8	1.16	1.22	94.7
F1	47.93	48.10	99.6	3.61	3.78	95.2
INLINE_1	63.88	64.21	99.4	4.65	4.92	94.1
LDOOR	78.72	79.15	99.4	7.26	7.72	93.6
A100	2.11	2.54	79.9	5.85	6.29	92.5
A126	4.33	5.14	81.2	11.70	12.52	92.9
A159	8.93	10.49	82.5	23.42	25.01	93.2
A200	17.51	20.56	82.6	46.56	49.62	93.4
A252	35.53	41.66	82.7	93.65	99.79	93.4

Table 4.2: Execution time of the two SpMV kernels and the CG-CUBLAS implementations built on top of them.

(almost any two) consecutive operations of the CG method, the CPU thread will be suspended till all the GPU threads complete the execution of each kernel. With the default polling operation mode of CUDA, the CPU thread will therefore enter a polling power-oblivious mode every time it has to wait for the termination of a kernel, which will happen continuously. To avoid this behavior, we can simply recur to the CUDA blocking operation mode, which suspends the CPU thread when waiting for the GPU to finish work.

Figure 4.4 summarizes the effect of this change for the CG-CUBLAS implementation. In particular, the figure reports the variations on time and CPU energy induced by the blocking mode over the default polling mode. In all cases there is a clear reduction of the energy dissipated by the CPU, as a suspended core can now be promoted by the hardware into an inactive power-friendly C-state, e.g., C1 (halt) or deeper [148]. In return, the negative impact on performance is evident, due to the cost of reactivating (promoting into the operating state C0) the suspended CPU thread. For AUDIKW_1, for example, the blocking mode introduces a penalty (i.e., increase) of 3.6% in time while energy consumption of the CPU is improved (i.e., reduced) by (-)26.6%. The source for the energy savings are elucidated in Figure 4.5, which shows a trace of the power rates for the CG-CUBLAS implementation operating in polling and blocking modes for this particular matrix.

In summary, the energy savings of the blocking mode for all cases cost a nonnegligible runtime performance decrease. We note that there is a connection between this outcome and the matrix size and number of nonzeros, as these determine the ratio between the matrix-vector products and the remaining operations in the CG solver (see Tables 4.1 and 4.2, respectively).

4.4.2 Merging CUDA kernels to regain performance

The general superior energy efficiency of the CUDA blocking mode comes along with a certain price from the point of view of performance. The challenge is thus to recover the performance of the polling mode while leveraging the power-saving mechanisms embedded in the hardware by operating in blocking mode. The key to this twofold goal lies in developing a new implementation of the CG method, that breaks down the rigid structure of the original code, so that several kernels can be gathered and merged together, with the result offering a higher computational intensity.

Specifically, consider the loop of the CG-CUBLAS implementation in Figure 4.3, which comprises a number of calls to CUBLAS kernels inside a loop that can be potentially executed from hundreds to thousands of times. We argue that this large number of kernel calls and the frequent data transfers between GPU and main memory are major sources of overhead. Reducing the number of calls by merging kernels

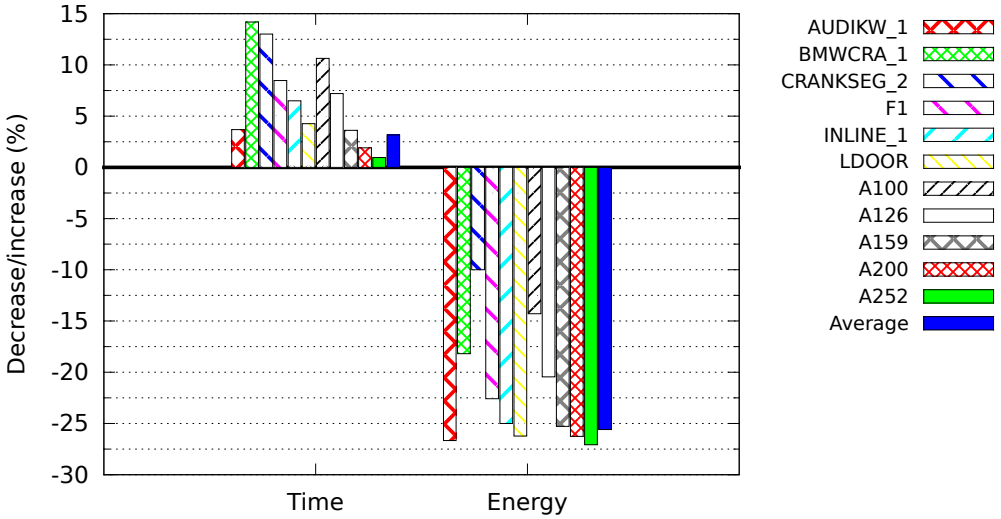


Figure 4.4: Variation of time and CPU energy of the CG-CUBLAS implementation when the blocking mode is set instead of the default polling mode.

and avoiding synchronizations due to data transfers are thus crucial to alleviate the time overhead, but can also render a lower number of activations of the CPU, yielding a more efficient usage of the C-states and, in consequence, an improved energy efficiency.

In the following two subsections we explain how to accommodate this into a merged variant of the CG method.

4.4.2.1 Avoiding synchronizations and data transfers

We started the elaboration of the merged solvers by developing a *CG-CUDA* implementation which differs from the straight-forward CG-CUBLAS implementation (Figure 4.3) in a few, but important details. First, the CG-CUDA code no longer employs CUBLAS kernels for the dot products, vector scaling and saxpy operations but, instead, refers to alternative *ad hoc* implementations. Second, in CG-CUDA all variables, except for `res`, are declared and kept in the GPU memory[2]. In principle, this is beneficial as it avoids the overhead associated with the transfers of data from the GPU memory to the main memory and the corresponding synchronizations. Furthermore, it also allows the kernel aggregation introduced in short.

One step further in this line is to check the convergence criterion every s iterations of the CG loop. While, on average, this technique incurs into $s/2$ additional iterations of the solver, it will also decrease the transfers from GPU to main memory (for the convergence residual, `res`), and the corresponding synchronizations associated with the convergence test by a factor of $(1-1/s)$. By setting s=10, for example, 90% of the tests are thus avoided.

4.4.2.2 Merged CUDA code

Current GPUs from NVIDIA feature a hardware design and programming model specially tailored for data-parallel applications [138], which also has some implications from the perspective of a power-aware execution. In particular, high performance GPUs exhibit a high number of cores, organized into several arrays called streaming multiprocessors, which can simultaneously execute the same operation (or kernel) on a large set of data, using thousands or even millions of GPU threads. The GPU arranges these threads

[2]We note that starting from CUDA 4.0, this is possible as well if one is willing to abandon the CUBLAS legacy API.

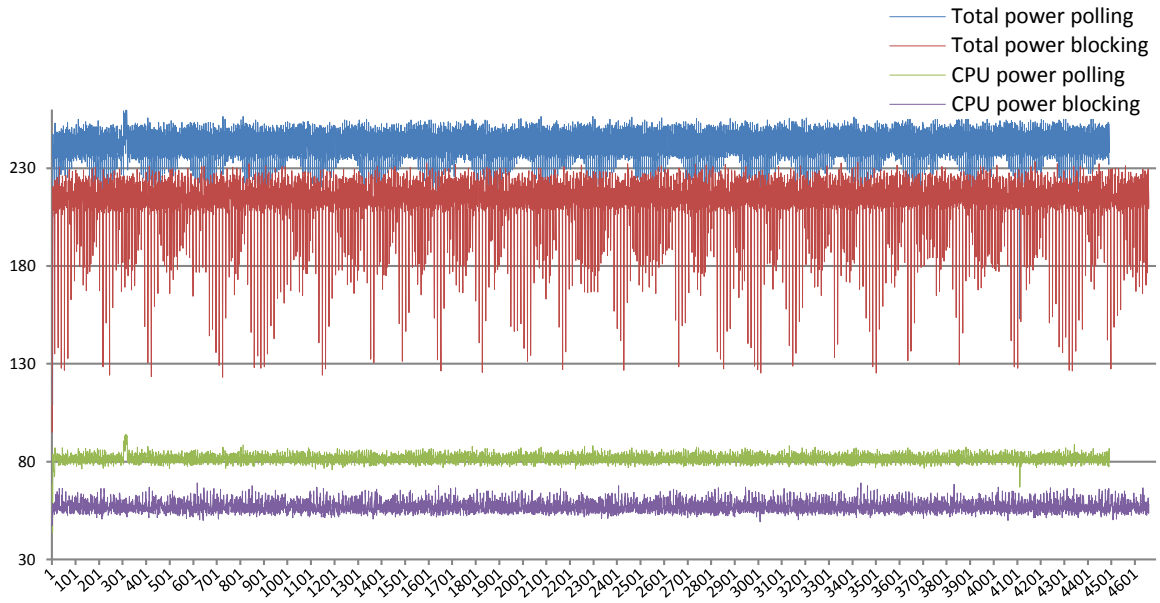


Figure 4.5: Power (in Watts) dissipated by the CG-CUBLAS implementation, using the polling and blocking modes, for the AUDIKW_1 matrix.

into a grid of several 1D, 2D or 3D thread blocks that are distributed among the multiprocessors [147]. Each multiprocessor then executes one or multiple thread blocks in SIMD fashion and, in turn, each core of a multiprocessor runs one or more threads within a block in SIMT mode [151]. While the threads within the same thread block execute the same instruction on different data and can communicate among themselves using barrier synchronization and shared memory, there is no synchronization between the thread blocks of the grid, except that they read/write the input/output data from/to the global memory [152].

```
while( ( k < maxiter ) && ( res > epsilon ) ){
  SSpMV <<<Gs,Bs>>> ( n, rowA, colA, valA, d, z );
  tmp = cublasSdot ( n, d, 1, z, 1 );
  rho = beta / tmp;
  gamma = beta;
  cublasSaxpy (n,  rho, d, 1, x, 1 );
  cublasSaxpy (n, -rho, z, 1, r, 1 );
  tmp = cublasSdot ( n, r, 1, r, 1 );
  beta = tmp;
  alpha = beta / gamma;
  cublasSscal (n, alpha, d, 1 );
  cublasSaxpy (n, one, r, 1, d, 1 );
  res = sqrt( beta );
  k++;
} // end-while
```

```
while( ( k < maxiter ) && ( res > epsilon ) ){
  scalar_fusion_1 <<<Gs, Bs, Ms>>> ( n, rowA, colA, valA,
                                     d, z, beta, rho, gamma, vtmp );
  fusion_2 ( Gs, Bs, Ms, beta, rho, vtmp );
  fusion_3 <<<Gs, Bs, Ms>>> ( n, rho, d, x, z, r, vtmp );
  fusion_4 ( Gs, Bs, Ms, vtmp, vtmp2 );
  fusion_5 <<<Gs, Bs>>> ( n, beta, gamma, alpha,
                                                  d, r, vtmp );
  cudaMemcopy( &res, beta, sizeof(float),
                                        cudaMemcpyDeviceToHost );
  res = sqrt( res );
  k ++;
} // end-while
```

Figure 4.6: Aggregation of kernels to transform the CG-CUBLAS implementation (using SSPMV) into the CG-merged implementation. The aggregation for the CG-CUBLAS implementation based on VSPMV is analogous.

```
1  __global__ void fusion_3(int n, float rho[],
2              float d[], float x[], float z[],
3                     float r[], float vtmp[]){
4
5    uint Idx = threadIdx.x;
6    uint i   = blockIdx.x * blockDim.x + Idx;
7    extern __shared__ float temp[];
8
9    if(i < n){
10     x[i] =  rho[0] * d[i] + x[i];
11     r[i] = -rho[0] * z[i] + r[i];
12   }
13   __syncthreads();
14
15   temp[Idx] = (i < n) ? r[i] * r[i] : 0;
16   __syncthreads();
17
18   if(Idx < 128){
19     temp[Idx] += temp[Idx+128];
20   }
21   __syncthreads();
22
23   if(Idx < 64){
24     temp[Idx] += temp[Idx+64];
25   }
26   __syncthreads();
27
28   if(Idx < 32){
29     volatile float *temp2 = temp;
30     temp2[Idx] += temp2[Idx+32];
31     temp2[Idx] += temp2[Idx+16];
32     temp2[Idx] += temp2[Idx+ 8];
33     temp2[Idx] += temp2[Idx+ 4];
34     temp2[Idx] += temp2[Idx+ 2];
35     temp2[Idx] += temp2[Idx+ 1];
36   }
37
38   if(Idx == 0)
39     vtmp[blockIdx.x] = temp[0];
40 }
```

Figure 4.7: `fusion_3` employed in the CG-merged code. Note that this kernel employs a block-size of 256.

These properties of NVIDIA CUDA's data-parallel programming model imply that GPU kernels must contain only operations that are independent or can be executed consecutively, favoring codes which consist of very few operations. Although this concept of simple kernels accommodates generic programming, it comes at the cost of a high number of kernel calls, with the aforementioned negative effect on the power dissipated by the CPU thread. The key aspect in our approach is thus to gather vector operations that can be executed independently or consecutively on a sub-vector without synchronizing with the other parts.

In particular, in Figure 4.6 we visualize how the individual kernels of the CG-CUBLAS code (left) are fused into our *CG-merged* implementation (right). To illustrate this process, consider e.g. how to merge the two saxpy operations followed by an inner product appearing in the CG-CUBLAS implementation, which basically become the invocation to kernel FUSION_3 and routine FUSION_4 in the merged variant. Obviously, the saxpy operations can be conducted independently on subsets of the vector, and therefore it is straight-forward to fuse them: see lines 10 and 11 of kernel FUSION_3 in Figure 4.7. For the inner product, merging is more involved. In this case, the component-wise multiplication can also be computed by parts

```
1  void fusion_4(int Gs, int Bs, int Ms,
2                float vtmp[], float vtmp2[]){
3
4    float *aux1 = vtmp, *aux2 = vtmp2;
5    int b = 1, Gs_next;
6    int Bs_2 = Bs/2, Ms_2 = Ms/2;
7
8    while(Gs > 1){
9      Gs_next = (unsigned int)
10     ceil((float) Gs / Bs);
11     if(Gs_next  > 1) Gs_next = Gs_next / 2;
12     fusion_4_1<<<Gs_next, Bs_2, Ms_2>>>
13                            (Gs, aux1, aux2);
14     Gs = Gs_next;
15     b = 1 - b;
16     if(b) {aux1 = vtmp; aux2 = vtmp2;}
17     else  {aux2 = vtmp; aux1 = vtmp2;}
18   }
19   if(b == 0)
20     cudaMemcpy(vtmp, aux1, sizeof(float),
21                   cudaMemcpyDeviceToDevice);
22 }
23
24 __global__ void fusion_4_1(int n,
25                float vtmp[], float vtmp2[]){
26
27   extern __shared__ float temp[];
28   unsigned int Idx = threadIdx.x;
29   unsigned int blockSize = blockDim.x;
30   unsigned int i = blockIdx.x
31                     * (blockSize * 2) + Idx;
32   unsigned int gridSize = blockSize
33                              * 2 * gridDim.x;
34
35   temp[Idx] = 0;
36   while (i < n) {
37     temp[Idx] += vtmp[i];
38     temp[Idx] += (i + blockSize < n)
39                   ? vtmp[i + blockSize] : 0;
40     i += gridSize;
41   }
42   __syncthreads();
43
44   if (Idx < 64){
45     temp[Idx] += temp[Idx + 64];
46   }
47   __syncthreads();
48
49   if (Idx < 32){
50     volatile float *temp2 = temp;
51     temp2[Idx] += temp2[Idx + 32];
52     temp2[Idx] += temp2[Idx + 16];
53     temp2[Idx] += temp2[Idx +  8];
54     temp2[Idx] += temp2[Idx +  4];
55     temp2[Idx] += temp2[Idx +  2];
56     temp2[Idx] += temp2[Idx +  1];
57   }
58   if (Idx == 0)
59     vtmp2[blockIdx.x] = temp[0];
60 }
```

Figure 4.8: Implementation of the kernel `fusion_4_1` and routine `fusion_4` employed in the CG-merged code. Note that `fusion_4_1` is based on a block-size of 128 threads.

and without strict order and, therefore it can become part of the previous kernel (see line 15 of FUSION_3). However, the reduction process requires careful synchronizations and, thus, results in an additional separate step. This procedure is accomplished in our case with routine FUSION_4, which in turn iteratively calls kernel FUSION_4_1 (see Figure 4.8), to perform a symmetric reduction using a thread block size of 256. The recursive reduction procedure is illustrated in Figure 4.9.

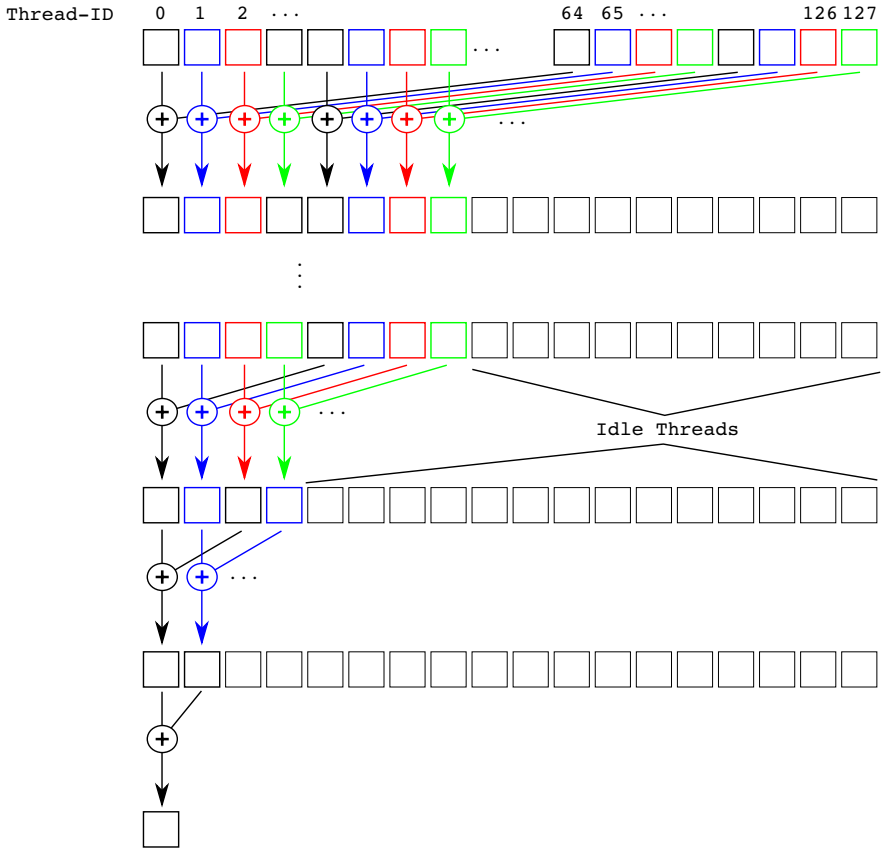


Figure 4.9: Symmetric reduction scheme with a thread block size of 128. For larger systems, the reduction has to be applied iteratively.

4.4.3 Performance and energy comparison

We next carry out an experimental analysis of the two implementations introduced so far, CG-CUBLAS and CG-merged, together with a third version, CG-merged-10, that checks the convergence criterion every s=10 iterations. For brevity, in some cases we drop the prefix "CG-" from the name of the implementations, replacing it by the operation mode (i.e., "polling-" or "blocking-") when it is convenient.

Table 4.3 collects the execution time, CPU energy and total energy measured for the three implementations, using both the polling and blocking modes, for the complete matrix collection.

At this point, it is worthwhile to recall the double objective of the "merged" implementations (i.e., CG-merge and CG-merge-10). In particular, these variants aimed at operating in blocking mode, to leverage the power-saving mechanisms of the hardware, while, at the same time, delivering performance close to that attained with the polling mode.

Let us consider the performance goal first. Figure 4.10 (top) reports the variation (increase or decrease) of the execution time experienced by the merged implementations, operating in polling and blocking modes,

Polling mode

		CUBLAS			CG-merge			CG-merge-10		
		Time	Cenergy	Tenergy	Time	Cenergy	Tenergy	Time	Cenergy	Tenergy
VSPMV	AUDIKW_1	9.23	0.75	2.22	9.15	0.74	2.20	9.14	0.74	2.20
	BMWCRA_1	1.34	0.11	0.31	1.32	0.11	0.30	1.31	0.10	0.30
	CRANKSEG_2	1.23	0.10	0.30	1.21	0.10	0.30	1.20	0.10	0.29
	F1	3.78	0.31	0.90	3.74	0.31	0.89	3.73	0.30	0.88
	INLINE_1	4.93	0.40	1.16	4.84	0.39	1.13	4.83	0.39	1.13
	LDOOR	7.73	0.61	1.79	7.58	0.60	1.75	7.57	0.60	1.76
SSPMV	A100	2.54	0.21	0.54	2.45	0.21	0.53	2.44	0.21	0.53
	A126	5.14	0.44	1.12	4.92	0.42	1.07	4.92	0.42	1.08
	A159	10.50	0.91	2.29	10.11	0.87	2.22	10.09	0.87	2.22
	A200	20.56	1.79	4.51	19.82	1.72	4.37	19.81	1.72	4.37
	A252	41.66	3.62	9.15	40.27	3.49	8.88	40.25	3.51	8.90
Average		9.87	0.84	2.20	9.58	0.81	2.14	9.57	0.81	2.15

Blocking mode

		CUBLAS			CG-merge			CG-merge-10		
		Time	Cenergy	Tenergy	Time	Cenergy	Tenergy	Time	Cenergy	Tenergy
VSPMV	AUDIKW_1	9.57	0.55	2.02	9.36	0.54	2.00	9.19	0.53	1.99
	BMWCRA_1	1.53	0.09	0.30	1.48	0.09	0.29	1.36	0.08	0.28
	CRANKSEG_2	1.39	0.09	0.29	1.34	0.08	0.28	1.24	0.08	0.27
	F1	4.10	0.24	0.83	3.96	0.23	0.82	3.79	0.22	0.80
	INLINE_1	5.25	0.30	1.07	5.05	0.29	1.04	4.88	0.28	1.02
	LDOOR	8.06	0.45	1.63	7.79	0.43	1.59	7.61	0.42	1.58
SSPMV	A100	2.81	0.18	0.52	2.60	0.17	0.50	2.48	0.16	0.48
	A126	5.51	0.35	1.04	5.16	0.32	0.99	4.98	0.31	0.97
	A159	10.88	0.68	2.08	10.34	0.65	2.01	10.17	0.64	1.98
	A200	20.95	1.32	4.05	20.08	1.26	3.93	19.90	1.24	3.90
	A252	42.06	2.64	8.17	40.51	2.53	7.93	40.32	2.52	7.90
Average		10.19	0.62	2.00	9.78	0.59	1.94	9.62	0.58	1.92

Table 4.3: Execution time, CPU energy (Cenergy) and total energy (Tenergy) consumption of the different CG implementations in polling (top) and (blocking) mode.

using the polling-CUBLAS implementation as reference. The left-hand side plot in the figure shows that there is some performance gain, below 5% in all cases, from the raw use of the merged versions in polling mode. The right-hand side plot exposes a more varied behavior when switching to the blocking mode. For example, the use of the blocking-merge implementation can have a negative impact on the execution time, in two of the cases close to 10%, and in many others below 5%; but it can also produce a positive effect, reducing the execution times by about 2% in some cases. When the blocking-merge implementation is slower than the polling-CUBLAS code, shifting to blocking-merge-10 compensates for this, turning the difference negligible. In all cases, the blocking-merge-10 implementation exhibits higher performance gains with respect to the reference than the blocking-merge code.

In summary, these experiments demonstrate that the performance goal is fulfilled, so that we can utilize the CG-merge-10 implementation in blocking mode instead of CG-CUBLAS in polling mode at either no expense or even a slight improvement from the performance perspective.

Consider the energy-related goal next. Figure 4.10 (middle and bottom) illustrates the CPU and total energy consumptions of the merged implementations, compared against those of the CG-CUBLAS code. While the merged variants executed in polling mode (left-hand side plots) render some appealing savings, the clear winners are the same variants operating in blocking mode (right-hand side plots), which yield a

reduction of the CPU energy draft of about 25–30%. Taking also the GPU power draft into account, the total energy savings are still around 10–12% for all cases except the smallest matrices. For those, we recall that the usage of idle-wait is almost counter-productive for the CUBLAS implementation, as the decreased average power draft comes along with significant runtime increase almost leveling the energy balance (see Figure 4.4). Comparing the respective values in Table 4.3 we observe that this is different for the merge-10 implementation. In this case, the negligible runtime increase, of less than 2%, induced by switching to blocking mode is easily compensated by a reduction of the average CPU power draft of almost 30% (specifically, from an average 80.00 Watts in polling mode to 56.90 Watts in blocking mode).

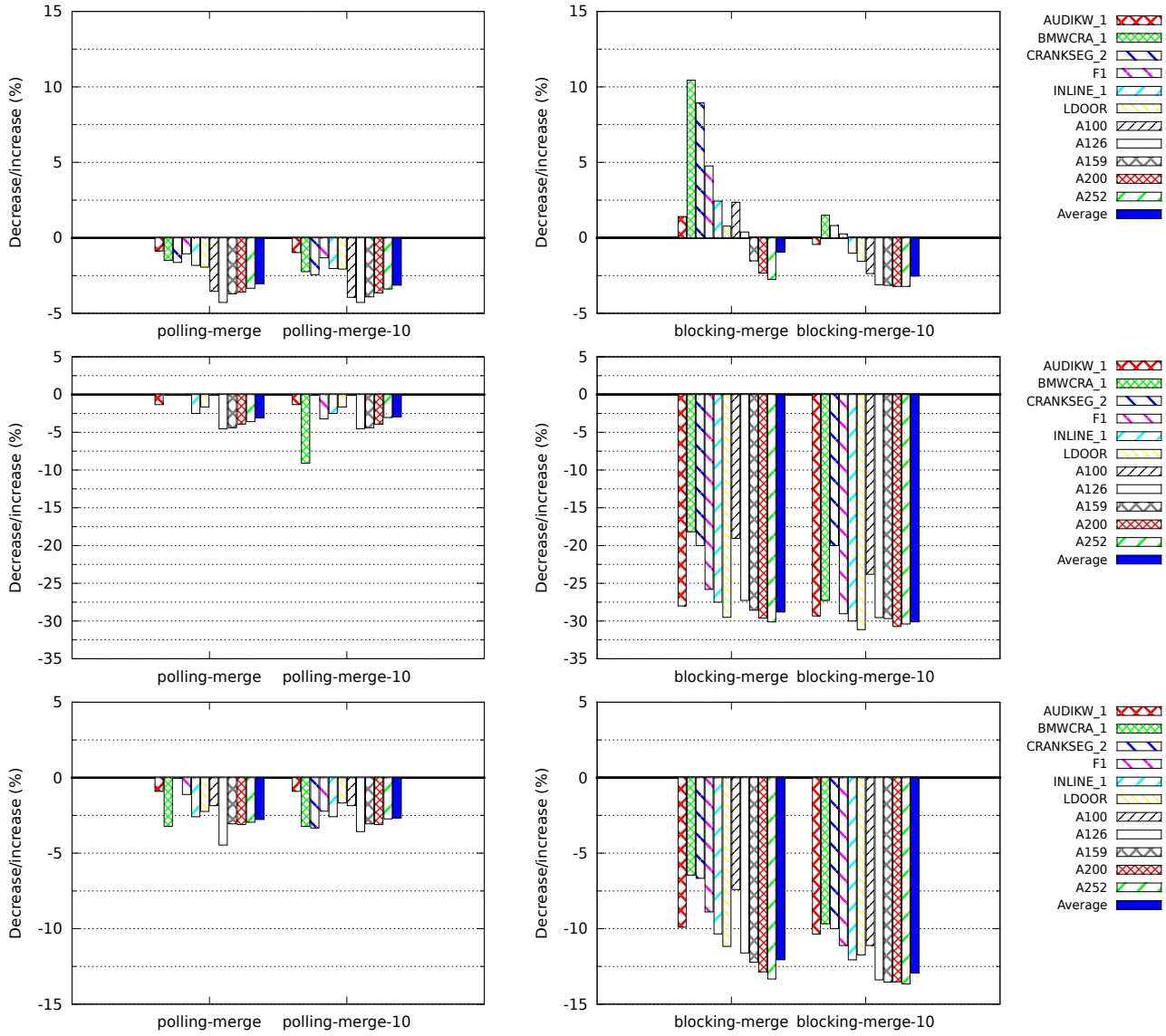


Figure 4.10: Comparison of time (top), CPU energy (middle) and total energy (bottom) of the merged implementations, in polling (left) and blocking (right) modes, using the polling-CUBLAS code as reference.

In summary, from the total values for runtime and energy provided in Figure 4.11 we can deduce that the reformulated conjugate gradient meets the twofold goal, rendering a significant reduction of the total energy consumption while retaining the runtime performance.

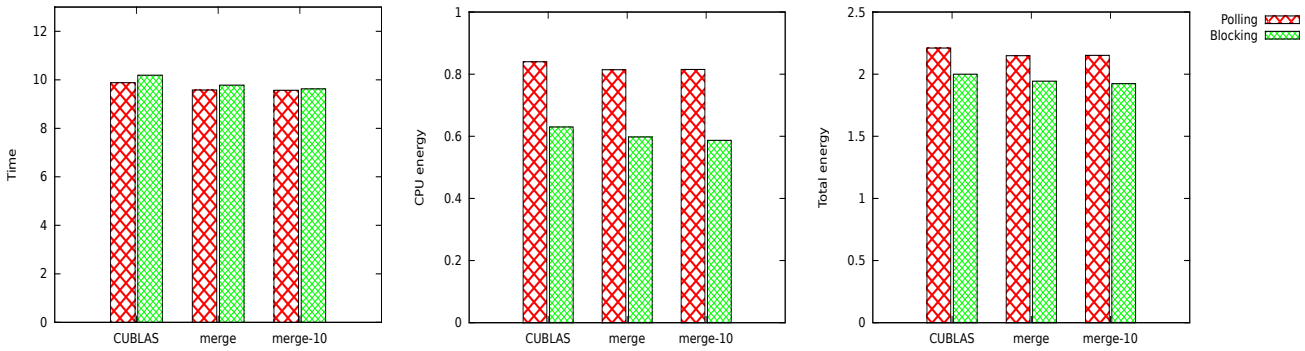


Figure 4.11: Execution time, CPU energy and total energy, averaged for all matrices in the benchmark collection.

4.5 Conclusions

In this paper we have proposed a new formulation of the CG iterative method that advocates for the use of algorithm-specific CUDA kernels when solving sparse linear systems on GPUs. These specialized kernels evolve from merging multiple, typically separated numerical operations, which reduces the number of synchronizations and data transfers, and in turn yields a more efficient hardware utilization. Experimental results with a varied benchmark of linear systems from the University of Florida Matrix Collection and the 3D Laplace problem reveal average CPU energy savings of about 25–30% as well as improvements around 2.5% on runtime.

Future research will focus on detecting repetitive patterns, which may enable auto-rearranging of operations and auto-merging of kernels, possibly with the aid of compiler technology.

Acknowledgments

This research were supported by project TIN2011-23283 of the *Ministerio de Ciencia e Innovación* and FEDER, and the EU Project FP7 318793 “EXA2GREEN”.

We thank Francisco D. Igual for his comments on an earlier version of this manuscript.

Acknowledgments

References

[134] The Green 500 list, 2013. `http://www.green500.org/`.

[135] The Top 500 list, 2013. `http://www.top500.org/`.

[136] S. Albers. Energy-efficient algorithms. *Commun. ACM*, 53:86–96, May 2010.

[137] P. Alonso, M. F. Dolz, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Reducing energy consumption of dense linear algebra operations on hybrid CPU-GPU platforms. In *Proc. 10th IEEE Int. Symp. on Parallel and Distributed Processing with Applications– ISPA 2012*, pages 56–62, 2012.

[138] H. Anzt. *Asynchronous and Multiprecision Linear Solvers - Scalable and Fault-Tolerant Numerics for Energy Efficient High Performance Computing* . PhD thesis, Karlsruhe Institute of Technology, Institute for Applied and Numerical Mathematics, Nov. 2012.

[139] H. Anzt, M. Castillo, J. C. Fernández, V. Heuveline, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Optimization of power consumption in the iterative solution of sparse linear systems on graphics processors. *Computer Science - Research and Development*, pages 1–9, 2011.

[140] H. Anzt, V. Heuveline, J.I. Aliaga, M. Castillo, J.C. Fernández, R. Mayo, and E.S. Quintana-Ortí. Analysis and optimization of power consumption in the iterative solution of sparse linear systems on multi-core and many-core platforms. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–6, july 2011.

[141] 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.

[142] D. Braess. *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*, volume 3. Cambridge University Press, 2007.

[143] J. Dongarra *et al*. The international ExaScale software project roadmap. *Int. J. of High Performance Computing & Applications*, 25(1), 2011.

[144] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1986.

[145] M. Duranton et al. The HiPEAC vision, 2010. Available from `http://www.hipeac.net/roadmap`.

[146] W. Feng, X. Feng, and R. Ge. Green supercomputing comes of age. *IT Professional*, 10(1):17 –23, jan.-feb. 2008.

[147] D. Göddeke. *Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters*. PhD thesis, Technische Universität Dortmund, Fakultät für Mathematik, May 2010.

[148] HP Corp., Intel Corp., Microsoft Corp., Phoenix Tech. Ltd., and Toshiba Corp. Advanced configuration and power interface specification, revision 5.0, 2011.

[149] C. Hsu and W. Feng. A feasibility analysis of power awareness in commodity-based high-performance clusters. In *Cluster Computing, 2005. IEEE Int.*, pages 1 –10, 2005.

[150] NVIDIA Corporation. *NVIDIA CUDA CUBLAS Library Programming Guide*, 1.0 edition, June 2007.

[151] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2.3.1 edition, August 2009.

[152] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware, 2007.

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

[154] J. R. Shewchuk. An introduction to the Conjugate Gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA, 1994.

[155] F. Vázquez, J. J. Fernández, and E. M. Garzón. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience*, 23(8):815–826, 2011.

4.6 Fe de erratas

La Figura 4.7 de la página 93 muestra el código que implementa la fusión entre dos AXPY y la primera parte de una operación DOT, en la que la sincronización que aparece en la línea 13 es innecesaria y por lo tanto se debe eliminar. La razón es que las sincronizaciones, que aparecen en las operaciones de reducción, aseguran que los datos cargados en memoria compartida son correctos, por lo tanto sólo tienen sentido cuando se produce alguna carga de datos sobre memoria compartida. Puesto que la primera carga se produce en la línea 15, la sincronización de la linea 13 no es necesaria. La Figura 4.12 muestra el código antes y después de esta corrección.

```
1  __global__ void fusion_3(int n, float rho[],
2              float d[], float x[], float z[],
3                     float r[], float vtmp[]){
4
5    uint Idx = threadIdx.x;
6    uint i   = blockIdx.x * blockDim.x + Idx;
7    extern __shared__ float temp[];
8
9    if(i < n){
10     x[i] =  rho[0] * d[i] + x[i];
11     r[i] = -rho[0] * z[i] + r[i];
12   }
13   __syncthreads();
14
15   temp[Idx] = (i < n) ? r[i] * r[i] : 0;
16   __syncthreads();
17
18   if(Idx < 128){
19     temp[Idx] += temp[Idx+128];
20   }
21   __syncthreads();
22
23   if(Idx < 64){
24     temp[Idx] += temp[Idx+64];
25   }
26   __syncthreads();
27
28   if(Idx < 32){
29     volatile float *temp2 = temp;
30     temp2[Idx] += temp2[Idx+32];
31     temp2[Idx] += temp2[Idx+16];
32     temp2[Idx] += temp2[Idx+ 8];
33     temp2[Idx] += temp2[Idx+ 4];
34     temp2[Idx] += temp2[Idx+ 2];
35     temp2[Idx] += temp2[Idx+ 1];
36   }
37
38   if(Idx == 0)
39     vtmp[blockIdx.x] = temp[0];
40 }
```

```
1  __global__ void fusion_3(int n, float rho[],
2              float d[], float x[], float z[],
3                     float r[], float vtmp[]){
4
5    uint Idx = threadIdx.x;
6    uint i   = blockIdx.x * blockDim.x + Idx;
7    extern __shared__ float temp[];
8
9    if(i < n){
10     x[i] =  rho[0] * d[i] + x[i];
11     r[i] = -rho[0] * z[i] + r[i];
12   }
13
14
15   temp[Idx] = (i < n) ? r[i] * r[i] : 0;
16   __syncthreads();
17
18   if(Idx < 128){
19     temp[Idx] += temp[Idx+128];
20   }
21   __syncthreads();
22
23   if(Idx < 64){
24     temp[Idx] += temp[Idx+64];
25   }
26   __syncthreads();
27
28   if(Idx < 32){
29     volatile float *temp2 = temp;
30     temp2[Idx] += temp2[Idx+32];
31     temp2[Idx] += temp2[Idx+16];
32     temp2[Idx] += temp2[Idx+ 8];
33     temp2[Idx] += temp2[Idx+ 4];
34     temp2[Idx] += temp2[Idx+ 2];
35     temp2[Idx] += temp2[Idx+ 1];
36   }
37
38   if(Idx == 0)
39     vtmp[blockIdx.x] = temp[0];
40 }
```

Figure 4.12: En la izquierda se muestra el código de la Figura 4.7, y en la derecha su código corregido, en el que se ha eliminado la línea 13.

CAPÍTULO 5

Systematic Fusion of CUDA Kernels for Iterative Sparse Linear System Solvers

J. I. Aliaga, **J. Pérez**, *E. S. Quintana-Ortí.*
"Systematic fusion of CUDA kernels for iterative sparse linear system solvers".
Lecture Notes in Computer Science 9233, Euro-Par 2015, pp. 675-686, Viena (Austria), 2015.
ISBN: 978-3-662-48095-3.

Abstract

We introduce a systematic analysis in order to fuse CUDA kernels arising in efficient iterative methods for the solution of sparse linear systems. Our procedure characterizes the input and output vectors of these methods, combining this information together with a dependency analysis, in order to decide which kernels to merge. The experiments on a recent NVIDIA "Kepler" GPU report significant gains, especially in energy consumption, for the fused implementations derived from the application of the methodology to three of the most popular Krylov subspace solvers with/without preconditioning.

5.1 Introduction

The solution of sparse linear systems [166] is an ubiquitous problem in ranking and search methodologies for the web, boundary value problems and finite element models for partial differential equations, economic modeling, and information retrieval, among others. The interest of these applications has given rise to a very large number of sophisticated sparse matrix storage layouts, libraries and algorithms for general-purpose processors (CPUs); see, e.g., [156, 161, 162, 169]. NVIDIA also supports the solution of sparse linear systems on graphics processors (GPUs), via the libraries CUBLAS and cuSPARSE, which respectively contain (CUDA) GPU kernels operating on vectors and sparse matrices.

Despite the importance of energy consumption [163, 165], few analyses of sparse linear algebra operations focus on this metric [157]. One particular source of energy inefficiency during the execution of an iterative solver [166] on a heterogeneous CPU-GPU server is that, when implemented via calls to the GPU kernels in CUBLAS/cuSPARSE, the CPU thread in control of the GPU repeatedly invokes fine-grain

CUDA kernels of low cost and, therefore, short duration. Even if the solver avoids most data transfers between (the memories of) CPU and GPU, this continuous stream of kernel calls often prevents the CPU from entering an energy-efficient C-state. In [158] we introduced the *fusion of GPU kernels* as a means to avoid this power-hungry scenario, for the particular case of the conjugate gradient (CG) method [166]. The results in that work report significant energy gains combined with a slight improvement in performance on a platform equipped with an Intel i7-3770K plus an NVIDIA "Fermi" GTX480 board. In this paper we make the following major contributions:

- We evolve [158] into a systematic analysis of the fusion of GPU kernels arising in a representative collection of sparse linear solvers: CG, BiCG and BiCGStab [166], including Jacobi-based preconditioned versions of these.
- We include three alternative implementations (scalar CSR, 2-D vector CSR and ELL [160]) for the sparse matrix-vector multiplication (SPMV), with different properties/characterization which impact the possibilities of merging the corresponding solvers.
- We experimentally demonstrate the benefits of kernel fusion in a platform comprising an Intel Core i3770K plus an NVIDIA "Kepler" K20c GPU.

The rest of the paper is structured as follows. In Section 5.2 we briefly review related work on the fusion of GPU kernels. In Section 5.3 we present the iterative solvers targeted in our work, identifying the mathematical operations that are implemented as CUDA kernels. Furthermore, we provide a systematic characterization of these GPU kernels, defining the properties that allow the fusion of two (or more) kernels. Finally, in Sections 5.4 and 5.5 we respectively evaluate the new merged iterative solvers and discuss the conclusions from this work.

5.2 Related Work

Kernel fusion has received considerable attention in the past as an optimization technique via, e.g., increased memory locality, lower overhead by eliminating multiple calls to kernels, and richer space for compiler optimizations. For brevity, we next discuss a few efforts that specifically target fusion of GPU kernels.

In [164] the authors analyze how to fuse several types of CUDA kernels (map, reduce, and combinations of these) corresponding to BLAS-1 and *dense* BLAS-2 operations. Our work specifically targets iterative solvers for sparse linear systems, and leads us to consider a richer set of operations, different from those in [164]. Furthermore, we break the implementation of reduction kernels into two stages so that one of them, which concentrates most of the computational work, can still be fused.

In [168] the authors study the fusion of CUDA kernels with the purpose of improving their power-energy efficiency by accommodating a higher and better balanced utilization of the GPU cores. Three classes of fusions are identified in their paper: "inner thread", "inner thread block", and "inter thread block", and their effects are *simulated* using two general benchmarks. Our fusions correspond to the first class as, for the type of operations arising in sparse linear algebra, this option yields a fair balance of the workload. Our approach differs in that we focus on the type of kernel fusions arising in sparse linear algebra, we provide a precise characterization of the kernels arising in this domain, and we offer experimental performance and energy results.

In [167] the authors propose the fusion of CUDA kernels arising in iterative sparse linear systems to improve performance, but only consider merging kernels that provide the same functionality and have no dependencies among them. The authors of [159] apply the techniques described in [158] to the iterative solution of sparse linear systems via BiCGStab. None of these works provides a systematic characterization of the GPU kernels and the conditions that allow their fusion.

$A \rightarrow M$	Compute Jacobi preconditioner
Initialize $r_0, r_0^*, p_0, p_0^*, x_0, \sigma_0, \tau_0; j := 0$	
while $(\tau_j > \tau_{\max})$	Loop for iterative solver
1. $v_j := Ap_j$	1. SPMV
2. $\alpha_j := \sigma_j/(v_j, p_j^*)$	2. DOT
3. $x_{j+1} := x_j + \alpha_j p_j$	3. AXPY
4. $r_{j+1} := r_j - \alpha_j v_j$	4. AXPY
5. $z_j := M^{-1} r_{j+1}$	5. JPRED (Jacobi preconditioner)
6. $v_j^* := A^T p_j^*$	6. SPMV
7. $r_{j+1}^* := r_j^* - \alpha_j v_j^*$	7. AXPY
8. $z_j^* := M^{-1} r_{j+1}^*$	8. JPRED (Jacobi preconditioner)
9. $\zeta_j := (z_j, r_{j+1}^*)$	9. DOT
10. $\beta_j := \zeta_j/\sigma_j$	10. Scalar op
11. $\sigma_j = \zeta_j$	11. Scalar op
12. $p_{j+1} := z_j + \beta_j p_j$	12. XPAY (AXPY-like)
13. $p_{j+1}^* := z_j^* + \beta_j p_j^*$	13. XPAY (AXPY-like)
14. $\tau_{j+1} := \| r_{j+1} \|_2$	14. Vector 2-norm (DOT + sqrt)
$j := j+1$	
endwhile	

Figure 5.1: Algorithmic formulation of the preconditioned BiCG method.

5.3 Systematic Kernel Fusion for Sparse Iterative Solvers

5.3.1 Overview of Iterative Solvers for Sparse Linear Systems

Given a linear system $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is sparse, $b \in \mathbb{R}^n$ contains the independent terms, and $x \in \mathbb{R}^n$ is the sought-after solution, iterative projection methods based on Krylov subspaces, in combination with an appropriate preconditioner, often outperform the most efficient direct solvers available today in terms of memory consumption and execution time [166].

Concerning the computational effort of iterative Krylov subspace methods, in practical applications the cost of the iteration loop is dominated by one or two SPMV involving A. Given a sparse matrix A with n_z nonzero entries, in general the cost of the SPMV is roughly $2n_z$ floating-point arithmetic operations (flops). Additionally, the loop body contains several vector operations that require $O(n)$ flops each.

Figure 5.1 offers an algorithmic description of the preconditioned BiCG method. In general, we use Greek letters for scalars, lowercase for vectors and uppercase for matrices. There, the user-defined parameter $\tau_{\max}$ sets an upper bound on the relative residual for the computed approximation to the solution x_j, and (z_1, z_2) denotes the inner product (DOT) of vectors z_1, z_2. The method involves two SPMV as well as several BLAS-1 (vector) operations per iteration (AXPY, XPAY and DOT). The application of the Jacobi preconditioner matrix M requires an element-wise product of two vectors.

The preconditioned BiCG method in Figure 5.1 contains all the GPU kernels that appear also in the preconditioned CG and BiCGStab. In the following section we characterize these kernels from the point of view of the type of access they perform to the data/results, we employ the preconditioned BiCG in order to present the systematic fusion of GPU kernels, and we generalize these principles to other variants of BiCG as well as other solvers.

5.3.2 Characterization of GPU kernels for sparse iterative solvers

A GPU kernel K performs a *mapped* access to a vector v if each thread of K accesses one of the elements of v, independently of other threads, and the global access is coalesced. We note that this property can be

Operation		Input vector(s)		Output vector
		x	y	y
AXPY	$y := \alpha x + y$	mapped	mapped	mapped
XPAY	$y := \alpha y + x$	mapped	mapped	mapped
DOT	$\alpha := x^T y = (x,y)$	mapped	mapped	unmapped
JPRED	$y := M^{-1}x$	–	mapped	mapped
SPMV scalar CSR	$y := Ax$	unmapped	–	mapped
SPMV vector CSR	$y := Ax$	unmapped	–	unmapped
SPMV ELL	$y := Ax$	unmapped	–	mapped

Table 5.1: Types of access to the vector inputs/output of the GPU kernels.

applied separately to the kernel input and output vectors. For the specific kernels identified in the sparse iterative solvers, we can then characterize their access types as shown in Table 5.1. For SPMV, we consider three well-known kernels/implementations [160]: *scalar* CSR, *vector* CSR and ELL.

5.3.3 Fusion of GPU kernels

We first discuss two factors that may impact the performance that can be attained by merging two GPU kernels:

Grid dimensionality (1D, 2D or 3D). For kernels that operate on vectors, this parameter has little impact on the performance. Therefore, for simplicity, a practical approach is to enforce the same dimensionality for both kernels by, e.g., setting that to the highest one of the two kernels.

Grid dimensions (number of threads per block and number of blocks). The approach here is, for simplicity, to enforce the same grid dimensions for both kernels, and to set the dimensions to the largest values employed by any of the two kernels. However, this must be done with care, as this parameter may have a real effect on the performance of the kernels.

Fusing kernels is targeted to improve performance and/or energy consumption, but obviously should produce the results of a non-fused execution. Let us elaborate now the properties that two GPU kernels, namely K_1 and K_2, must exhibit in order to participate in a fusion:

- In case K_1 and K_2 do not share any data (i.e., are independent), they can always be merged.
- Consider that K_1 produces a result or output vector v that is also an input for K_2, denoted hereafter as $K_1 \xrightarrow{v} K_2$. (That is, there exists a read-after-write or RAW data dependency between K_1 and K_2, dictated by the type and order of shared access to vector v.) For the type of (dependent) kernels arising in the sparse iterative solvers, the fusion is possible if K_1/K_2 perform a mapped access to the output/input vector v. This guarantees that *(i)* both kernels apply the same mapping of threads to the vector elements shared (exchanged) via registers; *(ii)* both kernels apply the same mapping of thread blocks to the vector elements shared (exchanged) via shared memory; and *(iii)* a global barrier is not necessary between the two kernels.

From the characterization in Table 5.1, we easily derive that AXPY, XPAY and JPRED can be always merged with any other dependent kernel (one or more of them) of the same sort (i.e., AXPY, XPAY and JPRED). Also, the scalar CSR and ELL versions of SPMV can be merged with any kernel of these three types that consumes the vector resulting from the product, i.e., SPMV (scalar CSR, ELL)$\xrightarrow{y} K_2 \in$

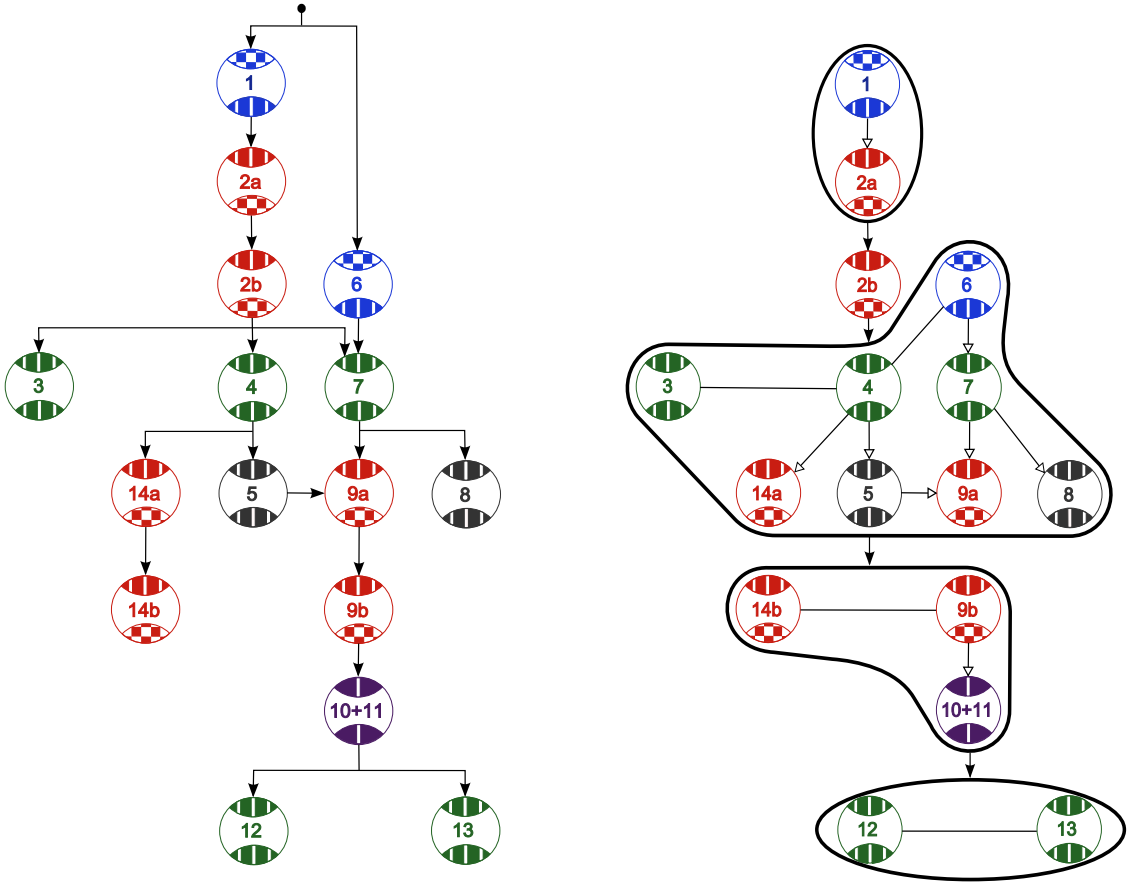


Figure 5.2: Dependencies between GPU kernels and fusions (left and right, respectively) for the preconditioned BiCG solver with SPMV based on the scalar CSR or ELL format.

{AXPY, XPAY, JPRED} can be merged; but $K_1 \in$ {AXPY, XPAY, JPRED} $\xrightarrow{y}$ SPMV cannot for any version of the sparse matrix-vector product.

The reduction kernel DOT is a special case that needs a tailored implementation so that it can be efficiently merged in $K_1 \xrightarrow{y}$ DOT. Concretely, in [158] we divided this kernel into two stages, say DOT_{ini} and DOT_{fin}, with the first one being implemented as a GPU kernel which performs the costly element-wise products and subsequent reduction within a thread block, producing a partial result in the form of a temporary vector with one entry per block. This is followed by routine DOT_{fin}, which completes the operation by repeatedly reducing the contents of this vector into a single scalar via a sequence of calls to GPU kernels. The important aspect to note at this point is that, because the reduction proceeds within blocks, this initial stage of the reduction performs a mapped read of the input vectors, and therefore can be efficiently merged in the sequence $K_1 \in$ {AXPY, XPAY, JPRED, SPMV} $\xrightarrow{y}$ DOT_{ini}. Routine DOT_{fin} is in practice implemented as a sequence of GPU kernels with mapped/unmapped input/output; see [158]. In consequence, this collection of kernels cannot be merged into a single one themselves, and $\text{DOT}_{\text{fin}} \xrightarrow{y} K_2$ cannot be fused.

5.3.4 Fusions in BiCG

We next apply the previous fusion principles to the preconditioned BiCG with SPMV based on the scalar CSR or ELL format, and we summarize the results for the (2-D) vector CSR format and the non-preconditioned version.

The left-hand side graph in Figure 5.2 identifies the dependencies (using arrows/edges) between operations of the preconditioner BiCG, with the nodes and their numeric labels identifying the operations within the loop body of the solver; see Figure 5.1. (For simplicity, we do not include the operations before the loop body or the dependencies between different iterations.) As argued earlier, the DOT operations (2, 9 and 14) are partitioned into two stages (a or b, corresponding respectively to kernel DOT_{ini} and routine DOT_{fin}) in order to facilitate the fusion of the first part, if possible, with a previous kernel. The node colors distinguish between the four different operation types: SPMV, DOT, AXPY/XPAY and JPRED. The patterns on top and bottom of each node specify, respectively, the type of mapping for the input and output vector(s) of each operation. Concretely, the parallel lines correspond to a mapped operator and the chessboard pattern an unmapped one. Operations 10 and 11 are special cases as they only receive/produce (input/output) one scalar and are merged into a single node.

The right-hand side graph in Figure 5.2 illustrates one specific fusion of kernels among the several possibilities dictated by the kernel dependencies and the mappings of the input/output vectors. The fusions are encircled by thick lines and designate four macro-kernels: {1-2a}, {3-4-5-6-7-8-9a-14a}, {9b-10-11-14b}, {12-13}; plus a single-node (macro-)kernel: {2b}. The arrowless lines connect groups of independent kernels (e.g., 3 and 4). For simplicity, we do not include all the connections within a group. The arrows identify dependencies inside macro-kernels (e.g., from 4 to 5) and between them (e.g., from {1-2a} to {2b}).

Our fused version of the preconditioned BiCG, when SPMV employs the alternative vector CSR format (with unmapped input and output for SPMV), differs from that in Figure 5.2 in that the two matrix-vector operations (kernels 1 and 6) are merged together; in addition, due to the unmapped output of kernel 1, kernel 2a becomes a single-node macro-kernel. The resulting macro-kernels are therefore: {1-6}, {2a}, {2b}, {3-4-5-7-8-9a-14a}, {9b-10-11-14b} and {12-13}. Also, for all variants of the BiCG solver (based on scalar CSR, vector CSR and ELL SPMV), the fusion graphs of their non-preconditioned counterparts simply differ in that kernels 5 and 8, corresponding to the application of the preconditioner, are not present.

These particular fusions were chosen following the fusion principles exposed in this section and some general performance guidelines:

- The fusions can be decided by performing a systematic analysis of each kernel, starting e.g. at 1, 2, etc., with those labeled with a higher number, taking into account the dependencies and the type of input/output (mapped or unmapped). In general, the strategy is to reduce as much as possible the total number of macro-kernels, in order to avoid the associated performance and energy overheads. For the preconditioned BiCG, the right-hand side graph in Figure 5.2 presents the minimum number of macro-kernels due to the restrictions imposed by the unmapped output vectors of the three DOT operations (2a/b, 9a/b and 14a/b). We note that 10+11 could have been instead merged with {12-13} but we selected the first option for performance reasons.

- The dependencies between operations within the same macro-kernel specify a partial order for their execution. In principle, independent kernels are merged by integrating their instructions into a single code one after another. As an exception, for performance reasons, when the initial or final stages of two independent DOT operations are merged together into a single macro-kernel (e.g., 9a with 14a; and also 9b with 14b), their instructions are interleaved in the code. (Interleaving of multiple DOT operations was proposed in [159].)

- Alternatively, 6 can be merged with {1-2a}, but this option was discarded because, for the scalar CSR and ELL implementations of SPMV, the result attained lower performance.

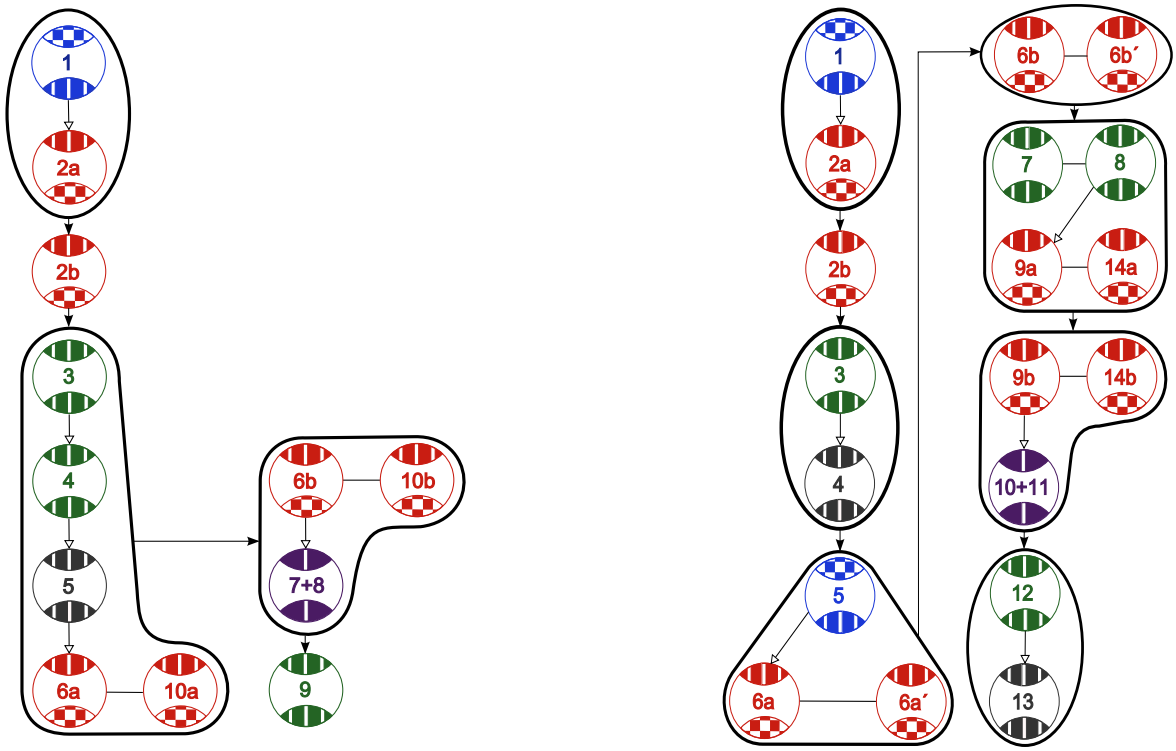


Figure 5.3: Fusions of GPU kernels in the preconditioned CG and BiCGStab solvers (left and right, respectively) with SPMV based on the scalar CSR or ELL format. The colors of the nodes match those employed for the preconditioned BiCG solver, and identify the same four types of operations: SPMV, DOT, AXPY/XPAY and JPRED.

5.3.5 Fusions in CG and BiCGStab

Figure 5.3 presents the fusion graphs for the preconditioned versions of CG and BiCGStab[1] when SPMV is based on the scalar CSR or ELL format. For the CG solver, the only difference when SPMV employs the vector CSR format is that kernels `1` and `2a` become two separate single-node macro-kernels. The same applies to the two SPMV in BiCGStab, i.e. kernels `1` and `5`, which become an isolated macro-kernel each. As in the BiCG solver, the non-preconditioned versions of CG and BiCGStab differ in that the nodes corresponding to the preconditioner application (`5` for the former and `4`, `13` in the latter) disappear.

The graphs in Figure 5.3 contain the minimum number of macro-kernels. Due to stricter dependencies of CG and BiCGStab compared with BiCG, the number of alternative fusions in the former two is reduced to instead joining `7+8` with `9` in CG, and `10+11` with `12-13` in BiCGStab.

In summary, the study of this collection of cases (three solvers, with and without preconditioner, and three different implementations of SPMV) exposes that, for the type of operations involved in these iterative solvers, the two stages of the DOT operations act as barriers (or synchronization points), enforcing a particular fusion/division of the macro-kernels.

[1]For BiCGStab, nodes `7` and `12` of the graph actually embed two dependent operations of type AXPY/XPAY each. For brevity, they are represented with a single node each.

Matrix	n_z	n	n_z/n
BMWCRA1_1	10,641,602	148,770	71.53
CRANKSEG_2	14,148,858	63,838	221.63
F1	26,837,113	343,791	78.06
INLINE_1	38,816,170	503,712	77.06
LDOOR	42,493,817	952,203	44.62
AUDIKW_1	77,651,847	943,645	82.28
FEM_3DTH2	147,900	3,489,300	23.59

Matrix	n_z	n	n_z/n
A100	6,940,000	1,000,000	6.94
A126	13,907,370	2,000,376	6.94
A159	27,986,067	4,019,679	6.94
A200	55,760,000	8,000,000	6.94
A252	111,640,032	16,003,001	6.94

Table 5.2: Description and properties of the test matrices from the UFMC (left) and the 3D Laplace problem (right). In the matrix names, FEM_3DTH2 corresponds to the "FEM 3D nonlinear thermal problem".

5.4 Experimental Evaluation

In this section we evaluate the performance and energy gains of the merged solvers, comparing them with non-fused counterparts. For this purpose, we employ several sparse matrices from the University of Florida Matrix Collection (UFMC)[2] and a difference discretization of the 3D Laplace problem; see Table 5.2. The coefficient matrix A for AUDIKW_1 and `inline_1` is too large to be stored in the ELL format and, therefore, these particular combinations of test/format are excluded from the evaluation. Moreover, A is unsymmetric for FEM_3DTH2 and, thus, cannot be tackled via the CG solver. For all cases, the solution vector was chosen to have all entries equal 1, and the independent vector was set to $b = Ax$. The iterative solvers were initialized with the starting guess $x_0 = 0$. All experiments were done using IEEE single precision (SP) arithmetic. While the use of double precision (DP) arithmetic is in general mandatory for the solution of sparse linear systems, the use of mixed SP-DP in combination with iterative refinement leads to improved execution time and energy consumption when the target platform is a GPU accelerator.

The target architecture is a Linux server (CentOS release 6.2 with kernel 2.6.32) equipped with a single Intel Core i7-3770K CPU (3.5 GHz, four cores) and 16 Gbytes of DDR3 RAM, connected via a PCI-e 2.0 bus to an NVIDIA "Kepler" K20c GPU (compute capability 3.5, 706 MHz, 2,496 CUDA cores) with 5 GB of DDR5 RAM integrated into the accelerator board. Power was collected using a *National Instruments* (NI) Data Acquisition System, composed of the NI9205 module and the NIcDAQ-9178 chassis, and plugged to the lines that connect the output of the power supply unit with motherboard and GPU.

In total, we evaluated CG, BiCG and BiCGStab, with and without preconditioning, using three different implementations of SPMV (scalar CSR, vector CSR and ELL), and five different versions of each solver:

- CUBLASL is a plain version of the solver implemented via calls CUBLAS kernels from the legacy programming interface of this library, combined with *ad-hoc* implementations of SPMV. In this version, one or more scalars may be transferred between the main memory and the GPU memory address space each time a kernel is invoked and/or its execution is completed.

- CUBLASN is an evolved version of the previous implementation that, whenever possible, maintains the scalars in the GPU memory (via the new interface of CUBLAS), in order to avoid unnecessary communication/synchronization between CPU and GPU.

- CUDA replaces the CUBLAS (vector) kernels in the previous version by our *ad-hoc* implementations, including the two-stage DOT.

- MERGE applies the fusions described in Section 5.3.

[2] `http://www.cise.ufl.edu/research/sparse/matrices/`

- MERGE_10 applies the fusions as well and, in addition, only checks the convergence every 10 iterations of the solver, thus reducing the amount of synchronizations between CPU and GPU due to the evaluation of this test.

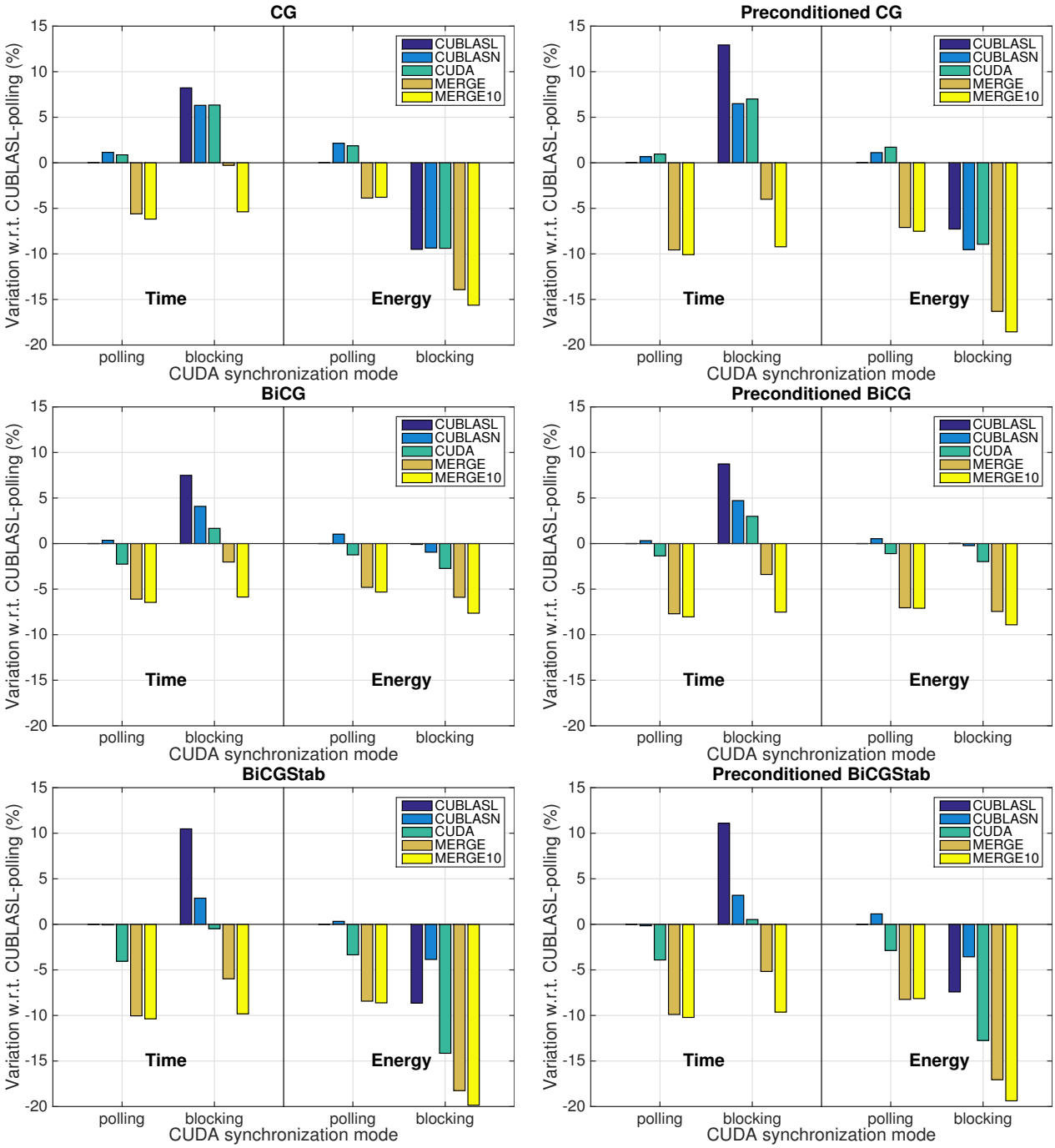


Figure 5.4: Execution time and energy consumption for CG, BiCG and BiCGStab solvers (top, middle and bottom, resp.) without and with preconditioner (left and right, resp.).

In review, there are 3 solvers, 2 preconditioning modes, 3 implementations of SPMV, and 5 versions of the solver; i.e., 90 combinations. Furthermore, we execute these configurations under the polling and blocking CUDA synchronization modes, and evaluate them for 12 test matrices, collecting the time and energy for

each scenario. In order to reduce the number of results to show, we report the variations in time/energy of the different implementations with respect to CUBLASL (as a percentage). In addition, we summarize the results for the 12 test cases into a single average value, giving the same weight to all of matrix tests. Also, we consider only the ELL implementation of SPMV for the UFMC cases and the scalar CSR variant for the Laplace problems since our experiments showed that these are the best options from the point of view of performance.

With these considerations, Figure 5.4 reports the time and energy variations for three solvers (CG, BiCG, BiCGStab) and four different versions of each (CUBLASN, CUDA, MERGE, MERGE_10), executed under two different synchronization modes (polling and blocking).

The first aspect to note is that all plots in Figure 5.4 reflect the same qualitative trend, independently of the specific solver and whether or not the preconditioner is present. Let us consider, e.g., the top-left plot (CG solver without preconditioner) and perform the comparison against the baseline case: CUBLASL executed in polling mode. The two non-fused versions CUBLASN and CUDA only experience a slight increase in both time and energy (around 2–3%) when operating under the polling mode. For the alternative blocking mode, these versions present an appealing reduction of the energy consumption (close to 10%), but unfortunately this comes at the cost of a more visible performance penalty (a time increase around 6–7%). The desired combination (reduction in both time and energy) is attained by the merged versions (MERGE and MERGE_10). Both algorithms report a decrease of execution time superior to 5%, except for MERGE executed in blocking mode, for which the variation of time is negligible. The best combination is clearly MERGE_10, which combines this reduction of time with a remarkable decrease of energy consumption, superior to 15%.

The best option is in general to employ MERGE_10 executed in blocking mode. Compared with the baseline case, the reduction in time for all solvers and preconditioning modes is between 5.1% and 10.2%, while from the energy perspective the savings vary between 4.0% and 20.0%. Comparing MERGE_10 with the same implementation executed in polling mode, the blocking mode basically matches its performance (around the same execution time) while producing higher energy gains, especially for CG and BiCGStab.

5.5 Concluding Remarks

We have introduced and applied a systematic methodology to derive fused versions some of the most popular iterative solvers (with and without preconditoning) for sparse linear systems. An analysis of the type of access that the threads in charge of a kernel's execution perform on the kernel inputs and outputs, together with the observation of the data dependencies between kernels determines which candidates can be fused. For performance and energy efficiency reasons, the general goal is to minimize the number of macro-kernels that resuls from the application of the fusions. From this point of view, e.g. we obtain reductions from 10→5, 13→5 and 14→8 for the preconditioned versions of CG, BiCG and BiCGStab, respectively. The gains are experimentally demonstrated on a recent CPU-GPU architecture, consisting of an Intel "Sandy-Bridge" multicore processor and an NVIDIA "Kepler" GPU. Compared with plain versions of the solvers based on CUBLAS and *ad-hoc* implementations of SPMV, the fused versions attain remarkable energy savings when executed in blocking mode. Furthermore, in general they match the performance of an execution of the same versions when executed in the performance-active but power-hungrier polling mode.

Acknowledgements

This research was supported by projects EU FP7 318793 (Exa2Green) and TIN2011-23283 of the *Ministerio de Economía y Competitividad* and EU FEDER. We thank Hartwig Anzt from the University of Tennessee for his comments.

References

[156] CSB library, 2014. http://gauss.cs.ucsb.edu/~aydin/csb/html/.

[157] J. I. Aliaga, H. Anzt, M. Castillo, J. C. Fernández, G. León, J. Pérez, and E. S. Quintana-Ortí.

[158] J. I. Aliaga, J. Pérez, E. S. Quintana-Ortí, and H. Anzt. Reformulated Conjugate Gradient for the Energy-Aware Solution of Linear Systems on GPUs. In *42nd Int. Conference on Parallel Processing (ICPP)*, pages 320–329, 2013.

[159] H. Anzt, W. Sawyer, S. Tomov, P. Luszczek, I. Yamazaki, and J. Dongarra. Optimizing Krylov subspace solvers on graphics processing units. In *2014 IEEE Int. Parallel Distributed Processing Symp. Workshops (IPDPSW)*, pages 941–949, 2014.

[160] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corp., December 2008.

[161] A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *IEEE Int. Parallel & Distributed Processing Symposium*, (IPDPS), pages 721–733, 2011.

[162] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, volume 45, pages 115–126, 2010.

[163] M. Duranton et al. HiPEAC vision 2015. High performance and embedded architecture and compilation, 2015. http://www.hipeac.net/vision.

[164] J. Filipovic, M. Madzin, J. Fousek, and L. Matyska. Optimizing CUDA code by kernel fusion—application on BLAS. *Computing Research Repository (CoRR)*, abs/1305.1183, 2013. http://arxiv.org/abs/1305.1183.

[165] S. H. Fuller and L. I. Millett. *The Future of Computing Performance: Game Over or Next Level?* National Research Council of the National Academies, 2011.

[166] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[167] S. Tabik, G. Ortega, and E.M. Garzón. Performance evaluation of kernel fusion BLAS routines on the GPU: iterative solvers as case study. *The Journal of Supercomputing*, 70(2):577–587, 2014.

[168] G. Wang, Y. Lin, and W. Yi. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *Green Computing and Communications (GreenCom)*, pages 344–350, 2010.

[169] S. Williams, N. Bell, J. Choi, M. Garland, L. Oliker, and R. Vuduc. Sparse matrix vector multiplication on multicore and accelerator systems. In Jakub Kurzak, David A. Bader, and Jack Dongarra, editors, *Scientific Computing with Multicore Processors and Accelerators*. CRC Press, 2010.

CAPÍTULO 6

Harnessing CUDA Dynamic Parallelism in the Solution of Sparse Linear Systems

J. I. Aliaga, D. Davidović, **J. Pérez**, *E. S. Quintana-Ortí.*
"Harnessing CUDA dynamic parallelism in the solution of sparse linear systems".
Parallel Computing – ParCo2015, (aceptado y pdte. de publicación). Edinburgo (Reino Unido), 2015.

Abstract

We leverage CUDA dynamic parallelism to reduce execution time while significantly reducing energy consumption of the Conjugate Gradient (CG) method for the iterative solution of sparse linear systems on graphics processing units (GPUs). Our new implementation of this solver is launched from the CPU in the form of a single "parent" CUDA kernel, which invokes other "child" CUDA kernels. The CPU can then continue with other work while the execution of the solver proceeds asynchronously on the GPU, or block until the execution is completed. Our experiments on a server equipped with an Intel Core i7-3770K CPU and an NVIDIA "Kepler" K20c GPU illustrate the benefits of the new CG solver.

Introduction

The discretization of partial differential equations (PDEs) often leads to large-scale linear systems of the form $Ax = b$, where the coefficient matrix $A \in \mathbb{R}^{n \times n}$ is *sparse*, $b \in \mathbb{R}^n$ contains the independent terms, and $x \in \mathbb{R}^n$ is the sought-after solution. For many problems (especially those associated with 3-D models), the size and complexity of these systems have turned iterative projection methods, based on Krylov subspaces, into a highly competitive approach compared with direct methods [177].

The Conjugate Gradient (CG) method is one of the most efficient Krylov subspace-based algorithms for the solution of sparse linear systems when the coefficient matrix is symmetric positive definite (s.p.d.) [177]. Furthermore, the structure and numerical kernels arising in this iterative solver are representative of a wide variety of efficient solvers for other specialized types of sparse linear systems.

When the target platform is a heterogeneous server consisting of a multicore processor plus a graphics processing unit (GPU), a conventional implementation of the CG method completely relies on the GPU for the computations, and leaves the general-purpose multicore processor (CPU) in charge of controlling the GPU only. The reason is that this type of iterative solvers is composed of fine-grain kernels, which exhibit

a low ratio between computation and data accesses (in general, O(1)). In this scenario, communicating data via a slow PCI-e bus mostly blurs the benefits of a CPU-GPU collaboration. In addition, in [170] we demonstrated the negative effect of CPU-GPU synchronization when the body of the iterative loop in the CG solver is implemented via calls to the GPU kernels, e.g. in CUBLAS/cuSPARSE. The cause is that, in such implementation, the CPU thread in control of the GPU repeatedly invokes fine-grain CUDA kernels of low cost and short duration, resulting in continuous stream of kernel calls that prevents the CPU from entering an energy-efficient C-state.

Our solution to alleviate these performance and energy overheads in [170] was to *fuse* (i.e. *merge*) CUDA kernels in order to decrease their number, thus reducing the volume of CPU-GPU synchronizations. The main contribution of this paper lies in the investigation of *dynamic parallelism* (DP) [172], as a complementary/alternative technique to achieve the same effect with a more reduced programming effort. Concretely, this work provides a practical demonstration of the benefits of DP on a solver like the CG method, representative of many other sparse linear system solvers as well as, in general, fine-grain computations. Our experimental evaluation of this algorithm on a platform equipped with an Intel Xeon processor and an NVIDIA "Kepler" GPU reports savings in both execution time and energy consumption, respectively of 3.65% and 14.23% on average, for a collection of problems.

DP is a recent technology introduced recently in the CUDA programming model, and is available for NVIDIA devices with compute capability 3.5 or higher. With DP, a child CUDA kernel can be called from within a parent CUDA kernel and then optionally synchronized on the completion of that child CUDA kernel. Some research on DP pursue the implementation of clustering and graph algorithms on GPUs [173, 174, 178], and a more complete analysis of unstructured applications on GPUs appears in [179]. This technology is also included as a compiler technique [180] to handle nested parallelism in GPU applications. DP is also used to avoid deadlocks in intra-GPU synchronization, reducing the energy consumption of the system [176].

The rest of the paper is structured as follows. In section 6.1 we briefly review the CG method and the fusion-based approach proposed in our earlier work. In section 6.2 we present the changes required to a standard implementation of the CG solver in order to efficiently exploit DP. In section 6.3 we evaluate the dynamic(-parallel) implementation of the new solver, and in section 6.4 we summarize the insights gained from our study.

6.1 Fusions in the CG method

6.1.1 Overview

Figure 6.1 offers an algorithmic description of the CG solver. Concerning the computational effort of the method, in practice the cost of the iteration loop is dominated by the sparse matrix-vector multiplication (SpMV) involving A. In particular, given a sparse matrix A with n_z nonzero entries, the cost of the SpMV in O1 is roughly $2n_z$ floating-point arithmetic operations (flops), while the vector operations in the loop body (O2, O3, O4, O5 and O8) require $O(n)$ flops each.

The dependencies between the operations in the body of the iterative loop of the CG method dictate a partial order for their execution. Specifically, at the $(j+1)$-th iteration,

$$\ldots \rightarrow \mathrm{O8} \rightarrow \overbrace{\mathrm{O1} \rightarrow \mathrm{O2} \rightarrow \mathrm{O4} \rightarrow \mathrm{O5} \rightarrow \mathrm{O6} \rightarrow \mathrm{O7} \rightarrow \mathrm{O9}}^{(j+1)\text{-th iteration}} \rightarrow \mathrm{O1} \rightarrow \ldots$$

must be computed in that order, but O3 and O8 can be computed any time once O2 and O6 are respectively available.

Initialize $r_0, p_0, x_0, \sigma_0, \tau_0; j := 0$	
while $(\tau_j > \tau_{\max})$	Loop for iterative CG solver
$v_j := Ap_j$	O1. SPMV
$\alpha_j := \sigma_j / p_j^T v_j$	O2. DOT
$x_{j+1} := x_j + \alpha_j p_j$	O3. AXPY
$r_{j+1} := r_j - \alpha_j v_j$	O4. AXPY
$\zeta_j := r_{j+1}^T r_{j+1}$	O5. DOT product
$\beta_j := \zeta_j / \sigma_j$	O6. Scalar op
$\sigma_{j+1} := \zeta_j$	O7. Scalar op
$p_{j+1} := z_j + \beta_j p_j$	O8. XPAY (AXPY-like)
$\tau_{j+1} := \| r_{j+1} \|_2 = \sqrt{\zeta_j}$	O9. Vector 2-norm (in practice, sqrt)
$j := j + 1$	
endwhile	

Figure 6.1: Algorithmic formulation of the CG method. In general, we use Greek letters for scalars, lowercase for vectors and uppercase for matrices. Here, $\tau_{\max}$ is an upper bound on the relative residual for the computed approximation to the solution.

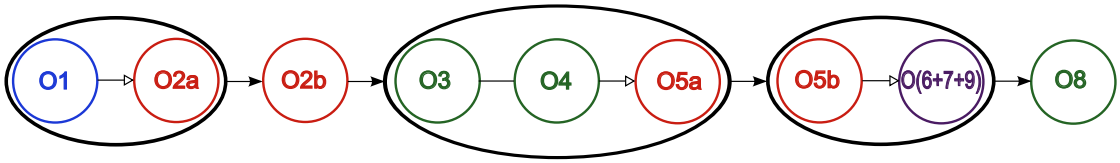


Figure 6.2: Fusions for the CG solver with SPMV based on the scalar CSR or ELL format.

6.1.2 Merging kernels in CG

In [170], we exploited that two CUDA kernels related by a RAW (read-after-write) dependency [175], dictated by a vector v that is an output of/input to the first/second kernel, can be merged if *(i)* both kernels apply the same mapping of threads to the elements of v shared (exchanged) via registers; *(ii)* both kernels apply the same mapping of thread blocks to the vector elements shared (exchanged) via shared memory; and *(iii)* a global barrier is not necessary between the two kernels.

In addition, in [170] we developed a tailored implementation of the kernel DOT that consisted of two stages, say $\text{DOT}^{\text{M}}_{\text{ini}}$ and $\text{DOT}^{\text{M}}_{\text{fin}}$, so that the first stage can be efficiently merged with a prior dependent kernel. In particular, the first stage was implemented as a GPU kernel which performs the costly element-wise products and subsequent reduction within a thread block, producing a partial result in the form of a temporary vector with one entry per block. This was followed by a *routine* $\text{DOT}^{\text{M}}_{\text{fin}}$ which completed the operation by repeatedly reducing the contents of this vector into a single scalar via a sequence of calls to GPU kernels; see [170] for details.

Figure 6.2 illustrates the fusions that were derived in our previous work for the CG solver when SPMV is based on the CSR scalar or ELL formats [171]. The node colors distinguish between three different operation types: SPMV, DOT and AXPY-like (AXPY/XPAY). As argued earlier, each DOT operation (O2 and O5) is divided into two stages (`a` or `b`, corresponding respectively to kernel $\text{DOT}^{\text{M}}_{\text{ini}}$ and routine $\text{DOT}^{\text{M}}_{\text{fin}}$) in order to facilitate the fusion of the first one with a previous kernel. The fusions are encircled by thick lines and designate three macro-kernels: {O1-O2a}, {O3-O4-O5a}, {O5b-O6-O7-O9}; plus two single-node (macro-)kernels: {O2b} and {O8}. The arrowless lines connect independent kernels (e.g., O3 and O4) and the arrows identify dependencies inside macro-kernels (e.g., from O1 to O2a) and between them (e.g., from {O1-O2a} to {O2b}). When SPMV employs the CSR vector format [171] the fusion graph differs from that in Figure 6.2 in that O1 and O2a cannot be merged.

This specialized formulation of the CG solver merged multiple numerical operations, reducing the number of synchronizations and data transfers, in turn yielding a more efficient hardware utilization. The experimental evaluation with a varied benchmark of linear systems from the University of Florida Matrix Collection and the 3D Laplace problem, using a server equipped with an Intel Core i7-3770K processor and an NVIDIA GeForce GTX480 GPU, revealed remarkable CPU energy savings and minor improvements on runtime, with respect to a plain implementation of the CG method based on the use of CUBLAS kernels and the CUDA polling synchronization mode.

6.2 Exploiting DP to Enhance CG

In principle, kernel fusion and DP are orthogonal techniques that can be applied independently or in combination. As described next, our *dynamic version* of CG integrates specialized implementations of DOT, AXPY and XPAY, which are more efficient than their "fusible" counterparts previously developed for the *merge version*.

6.2.1 Two-stage dynamic DOT

Our previous implementation of DOT divided this operation into two stages, $\text{DOT}^{\text{M}}_{\text{ini}}$ and $\text{DOT}^{\text{M}}_{\text{fin}}$, with the former one implemented as a CUDA kernel and the second being a routine that consists of a loop which invoked a CUDA kernel per iteration. The problem with this approach is that, if integrated with DP, $\text{DOT}^{\text{M}}_{\text{fin}}$ involves a sequence of nested calls to CUDA kernels from inside the GPU and, in practice, incurs a large overhead. In order to avoid this negative effect, we redesigned DOT as a two-stage procedure, $\text{DOT}^{\text{D}}_{\text{ini}}$ and $\text{DOT}^{\text{D}}_{\text{fin}}$, but with each stage implemented as a single CUDA kernel. One main difference between $\text{DOT}^{\text{D}}_{\text{ini}}$ and $\text{DOT}^{\text{M}}_{\text{ini}}$ is that the former cannot be merged with other kernels. However, we note that the "dynamic" version of the first stage is intended to be used in combination with DP and, therefore, reducing the number of kernels is no longer a strong urge (though it may still be convenient).

Figures 6.3 and 6.4 illustrate the implementation of the GPU kernels $\text{DOT}^{\text{D}}_{\text{ini}}$ and $\text{DOT}^{\text{D}}_{\text{fin}}$, respectively. There, `n` specifies the length of the vectors, `x` and `y` are the vectors involved in the reduction. The first kernel is invoked as

```
DOT_D_ini <<NumBlk, BlkSize, sizeof(float)*BlkSize>> (n, x, y, valpha);
```

and reduces the two vectors into `NumBlk`=256 partial results, stored upon completion in `valpha`. We note that, for performance reasons, this kernel spawns `GrdSize` = `NumBlk` · `BlkSize`= 256 · 192 threads, and each thread, processes two entries (one at `threadId` and a second at `threadId+BlkSize`) per iteration and per chunk of 2·`GrdSize` elements of the vectors, yielding a coalesced access to their entries; see Figure 6.5. The subsequent invocation to kernel

```
DOT_D_fin <<NumBlk2, BlkSize2, sizeof(float)*BlkSize2>> (valpha);
```

then produces the sought-after scalar result into the first component of this vector. The grid and block dimensions `NumBlk` = 256, `BlkSize` = 192, `NumBlk2` = 1, `BlkSize2` = `NumBlk` = 256 passed for the kernel launches were experimentally determined, except for `BlkSize` which was set to the number of CUDA cores per SMX (streaming multiprocessor).

6.2.2 Two-stage dynamic AXPY/XPAY

For performance reasons, the AXPY and XPAY have been also reorganized in the dynamic version of CG so that each CUDA thread operates with a pair of elements of each vectors. Figure 6.6 offers the code for

```
__global__ void DOT_D_ini(int n, float *x, float *y, float *valpha) {
  extern __shared__ float vtmp[];

  // Each thread loads two elements from each chunk
  // from global to shared memory
  unsigned int   tid      = threadIdx.x;
  unsigned int   NumBlk   = gridDim.x;    // = 256
  unsigned int   BlkSize  = blockDim.x;   // = 192
  unsigned int   Chunk    = 2 * NumBlk * BlkSize;
  unsigned int   i        = blockIdx.x * (2 * BlkSize) + tid;
  volatile float *vtmp2   = vtmp;

  // Reduce from n to NumBlk * BlkSize elements. Each thread
  // operates with two elements of each chunk
  vtmp[tid] = 0;
  while (i < n) {
    vtmp[tid] += x[i] * y[i];
    vtmp[tid] += (i+BlkSize < n) ? (x[i+BlkSize] * y[i+BlkSize]): 0;
    i         += Chunk;
  }
  __syncthreads();

  // Reduce from BlkSize=192 elements to 96, 48, 24, 12, 6, 3 and 1
  if (tid < 96) { vtmp[tid] += vtmp[tid + 96]; } __syncthreads();
  if (tid < 48) { vtmp[tid] += vtmp[tid + 48]; } __syncthreads();
  if (tid < 24) {
    vtmp2[tid] += vtmp2[tid + 24]; vtmp2[tid] += vtmp2[tid + 12];
    vtmp2[tid] += vtmp2[tid + 6 ]; vtmp2[tid] += vtmp2[tid + 3 ];
  }

  // Write result for this block to global mem
  if (tid == 0) valpha[blockIdx.x] = vtmp[0] + vtmp[1] + vtmp[2];
}
```

Figure 6.3: Implementation of kernel $\text{DOT}^{\text{D}}_{\text{ini}}$.

the former, with the second operation being implemented in an analogous manner. There, `n` specifies the length of the vectors, `x` and `y` are the vectors involved in the operation, and `alpha` is the scalar. The call to this kernel is done as

```
AXPY_D <<NumBlk3, BlkSize3/2>> (n, alpha, x, y);
```

with `NumBlk3`=$\lceil$`n/BlkSize3`$\rceil$ and `BlkSize3` = 256. The same values are also used for the dynamic implementation of XPAY.

Finally, our dynamic CG solver merges O3 and O4 into a single macro-kernel, creating another macro-kernel with the scalar operations (O6, O7 and O9) and the second stage of the last DOT (O5b).

6.3 Experimental Evaluation

We next expose the performance and energy of the dynamic CG solver compared with our previous implementations [170]. For this purpose, we employ several sparse matrices from the University of Florida Matrix Collection (UFMC)[1] and a difference discretization of the 3D Laplace problem; see Table 6.1. For all cases, the solution vector was chosen to have all entries equal 1, and the independent vector was set to $b = Ax$. The iterative solvers were initialized with the starting guess $x_0 = 0$. All experiments were done using

[1] `http://www.cise.ufl.edu/research/sparse/matrices/`

```
__global__ void DOT_D_fin(float *valpha) {
  extern __shared__ float vtmp[];

  // Each thread loads one element from global to shared mem
  unsigned int   tid = threadIdx.x;
  volatile float *vtmp2 = vtmp;

  vtmp[tid] = valpha[tid]; __syncthreads();

  // Reduce from 256 elements to 128, 64, 32, 16, 8, 2 and 1
  if (tid < 128) { vtmp[tid] += vtmp[tid + 128]; } __syncthreads();
  if (tid < 64)  { vtmp[tid] += vtmp[tid + 64 ]; } __syncthreads();
  if (tid < 32)  {
    vtmp2[tid] += vtmp2[tid + 32]; vtmp2[tid] += vtmp2[tid + 16];
    vtmp2[tid] += vtmp2[tid + 8 ]; vtmp2[tid] += vtmp2[tid + 4 ];
    vtmp2[tid] += vtmp2[tid + 2 ]; vtmp2[tid] += vtmp2[tid + 1 ];
  }

  // Write result for this block to global mem
  if (tid == 0) valpha[blockIdx.x] = *vtmp;
}
```

Figure 6.4: Implementation of kernel $\text{DOT}^{\text{D}}_{\text{fin}}$.

Matrix	Acronym	n_z	n	n_z/n
BMWCRA1_1	bmw	10,641,602	148,770	71.53
CRANKSEG_2	crank	14,148,858	63,838	221.63
F1	F1	26,837,113	343,791	78.06
INLINE_1	inline	38,816,170	503,712	77.06
LDOOR	ldoor	42,493,817	952,203	44.62
AUDIKW_1	audi	77,651,847	943,645	82.28
A252	A252	111,640,032	16,003,001	6.94

Table 6.1: Description and properties of the test matrices from the UFMC and the 3D Laplace problem.

IEEE single precision arithmetic. While the use of double precision arithmetic is in general mandatory for the solution of sparse linear systems, the use of mixed single/double-precision in combination with iterative refinement leads to improved execution time and energy consumption when the target platform is a GPU accelerator.

The target architecture is a Linux server (CentOS release 6.2 with kernel 2.6.32 with CUDA v5.5.0) equipped with a single Intel Core i7-3770K CPU (3.5 GHz, four cores) and 16 Gbytes of DDR3 RAM, connected via a PCI-e 2.0 bus to an NVIDIA "Kepler" K20c GPU (compute capability 3.5, 706 MHz, 2,496 CUDA cores) with 5 GB of DDR5 RAM integrated into the accelerator board. Power was collected using a *National Instruments* data acquisition system, composed of the NI9205 module and the NIcDAQ-9178 chassis, and plugged to the lines that connect the power supply unit with motherboard and GPU.

Our experimental evaluation included four implementations of the CG solver:

- CUBLASL is a plain version that relies on CUBLAS kernels from the legacy programming interface of this library, combined with *ad-hoc* implementations of SPMV. In this version, one or more scalars may be transferred between the main memory and the GPU memory address space each time a kernel is invoked and/or its execution is completed.

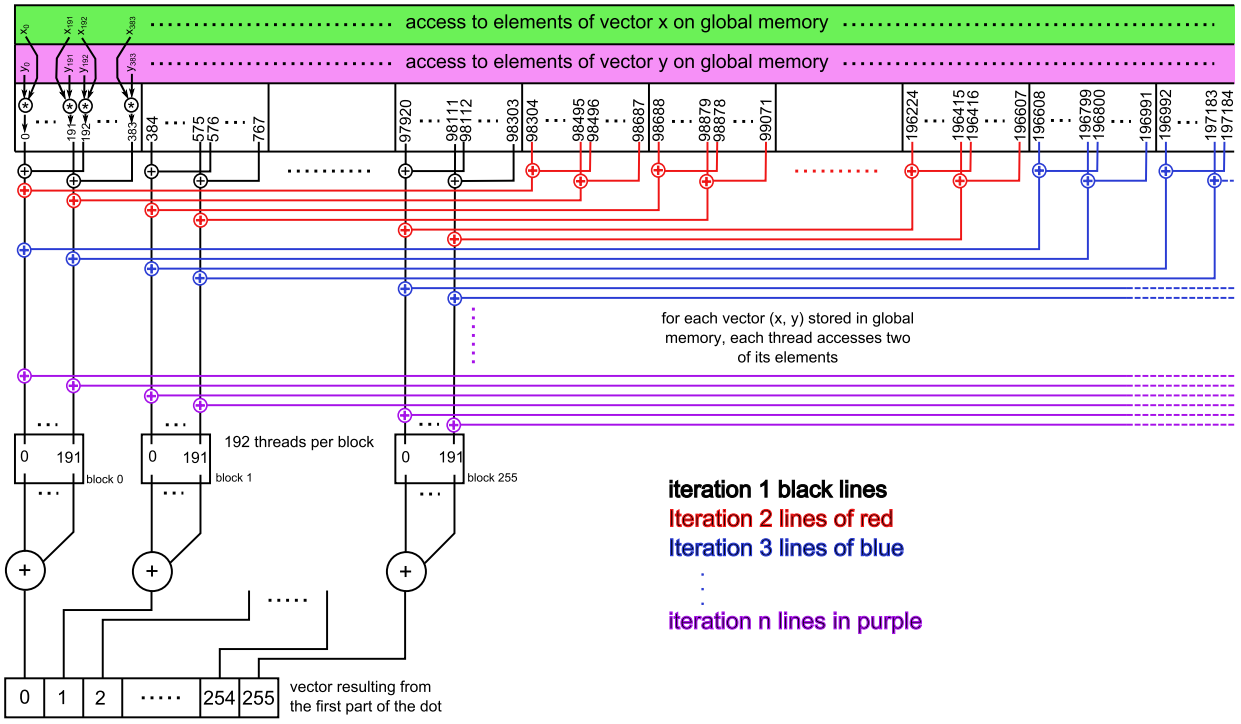


Figure 6.5: Implementation of $\text{DOT}^{\text{D}}_{\text{ini}}$.

- CUDA replaces the CUBLAS (vector) kernels in the previous version by our *ad-hoc* implementations.
- MERGE applies the fusions described in Section 6.1, including the two-stage $\text{DOT}^{\text{M}}_{\text{ini}}$ +$\text{DOT}^{\text{M}}_{\text{fin}}$.
- DYNAMIC exploits DP including the two-stage dynamic implementations of DOT, AXPY and XPAY introduced in Section 6.2 and merging O3 and O4 into a single macro-kernel.

Furthermore, we execute these configurations under the CUDA polling and blocking synchronization modes. We evaluated three different implementations of SPMV, scalar CSR, vector CSR and ELL [171], but only report results for the second one (vector CSR) which was experimentally determined to be the best option for most of the matrix cases.

Figure 6.7 reports the time and energy variations of the CUDA, MERGE and DYNAMIC versions of the CG solver with respect to CUBLASL executed in the CUDA polling synchronization mode (*baseline case*) for each matrix case. Table 6.2 summarizes these results via minimum, maximum and average numbers (with the former two corresponding to the largest and smallest in absolute value).

Let us analyze first the CUBLASL implementation executed in blocking mode. Compared with the same routine executed in polling mode (i.e, the baseline case), we observe an appealing reduction of the energy consumption, -10.85% on average, though it comes at the cost of a noticeable increase in the execution time, around 7.15% on average. The CUDA version of the solver executed in polling mode incurs a very small overhead in execution time with respect to the baseline (0.21% on average) and a slightly larger one in energy (1.79% on average). Moreover, when executed in blocking mode, the CUBLASL and CUDA implementations offer a close behavior, with a considerable increase in execution time that neutralizes the benefits of the notable reduction in energy. The MERGE variant of the solver combines the speed of a polling execution with the energy efficiency of a blocking one. In particular, this variant executed in blocking mode

```
1  __global__ void AXPY_D(int n, float *alpha, float *x, float *y) {
2    unsigned int NumBlk  = gridDim.x;
3    unsigned int BlkSize = blockDim.x;
4    unsigned int i       = blockIdx.x * (2 * BlkSize) + threadIdx.x;
5    unsigned int Chunk   = 2 * NumBlk * BlkSize;
6
7    while (i < n) {
8      y[i] +=  *alfa * x[i];
9      if (i + BlkSize < n) y[i + BlkSize] += *alfa * x[i + BlkSize];
10     i += Chunk;
11   }
12 }
```

Figure 6.6: Implementation of kernel AXPY^{D}.

CUDA mode	Implementation	Time			Energy		
		Min	Max	Avg.	Min	Max	Avg.
Polling	CUBLASL	0.00	0.00	0.00	0.00	0.00	0.00
	CUDA	0.08	0.41	0.21	0.23	7.94	1.79
	MERGE	-0.89	-3.07	-1.71	-1.42	5.03	0.62
	DYNAMIC	-1.54	-4.76	-3.65	-1.17	-3.32	-2.58
Blocking	CUBLASL	0.62	12.88	7.15	-3.30	-13.48	-10.85
	CUDA	0.78	9.39	4.74	-4.45	-12.62	-10.70
	MERGE	-0.59	-1.70	-1.06	-8.31	-13.96	-12.47
	DYNAMIC	-1.54	-4.71	-3.65	-13.73	-14.50	-14.23

Table 6.2: Minimum, maximum and average variations (in %) of execution time and energy consumption for CG with respect to the baseline.

slightly reduces the average execution time (by -1.06%) while extracting much of the energy advantages of this mode (reduction of -12.47% on average). Finally, the DYNAMIC implementation outperforms all other variants, including MERGE, obtaining the largest reduction in both time and energy (respectively, -3.65% and -14.23% on average). In addition, the DYNAMIC version of the solver does not require the complex reorganization of the code entailed by the MERGE case.

6.4 Concluding Remarks

We have presented a CUDA implementation of the CG method for the solution of sparse linear systems that exploits DP as means to produce a more energy efficient solution. With this new implementation, the CPU invokes a single "parent" CUDA kernel in order to launch the CG solver on the GPU, and can then proceed asynchronously to perform other work or be simply put to sleep via the CUDA blocking synchronization mode and the CPU C-states. The GPU is then in charge of executing the complete solver, with the parent kernel calling other "child" CUDA kernels to perform specific parts of the job. In order to improve efficiency, we have redesigned the implementation of the key vector operations arising in CG, concretely the dot product and axpy-like operations, into two-stage CUDA kernels. This is particular important for the former operation in order to avoid that the exploitation of DP results in a hierarchy of "nested" invocations to other CUDA kernels (i.e., a multilevel structure of parents and children).

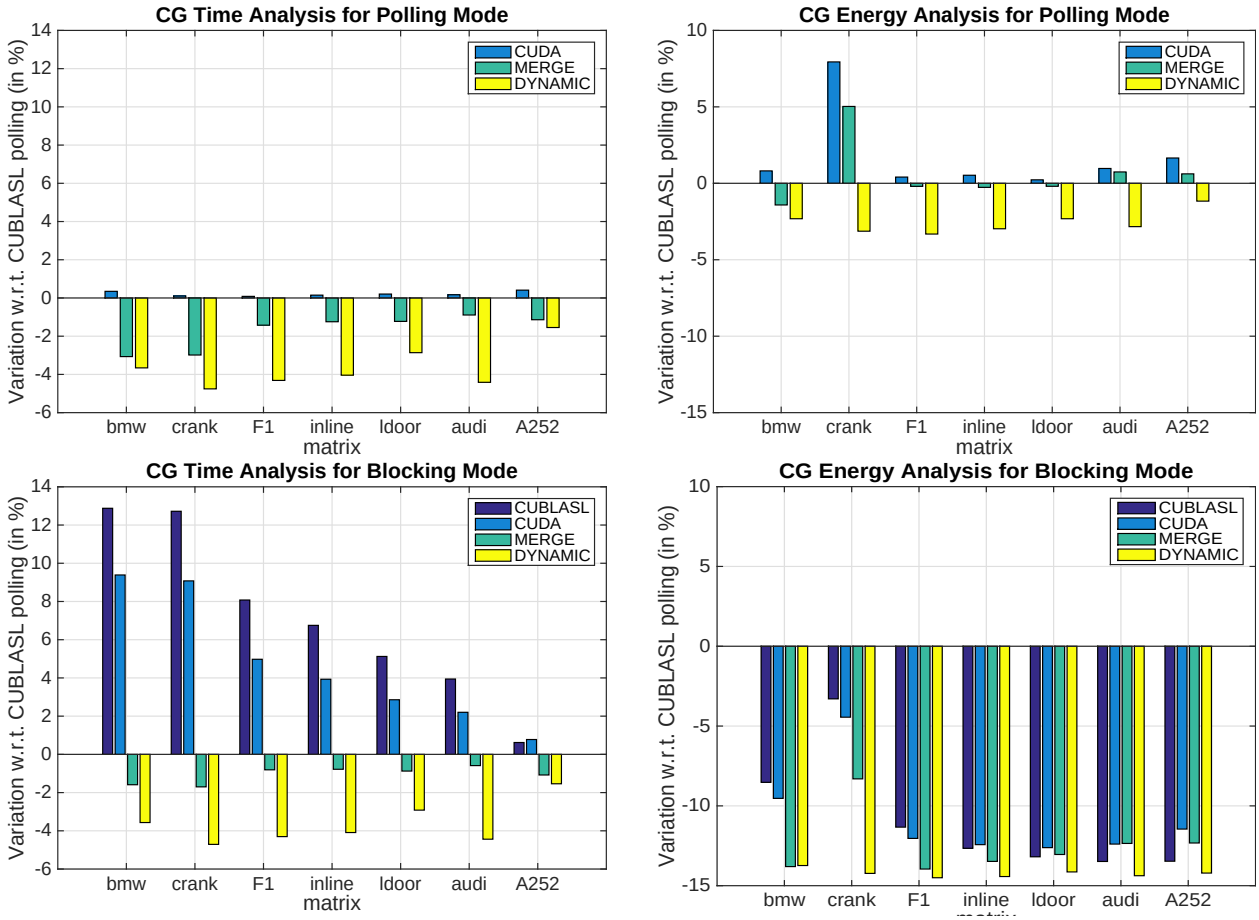


Figure 6.7: Variations (in %) of execution time and energy consumption of the CG solver (left and right, respectively) with respect to the baseline.

The experimentation on a platform with a recent Intel Core i7-3770K CPU and an NVIDIA "Kepler" K20c GPU reports the superiority of the dynamic CG solver, which outperforms our previous fusion-based implementation, in both execution time and energy consumption. From the programming point of view, the dynamic version also presents the important advantage of being more modular, as it does not require the major reorganization of CG, via kernel fusions, that were entailed by the fusion-based implementation.

We have applied similar techniques to iterative solvers based on BiCG and BiCGStab, as well as variants of these and CG that include a simple, Jacobi-based preconditioner, with similar benefits on performance and energy efficiency. The gains that can be obtained with DP heavily depend on the granularity of the CUDA kernels and, as the complexity of the preconditioner grows, we can expect that the positive impact of DP decreases.

Acknowledgments

This research has been supported by EU under projects EU FP7 318793 (Exa2Green), EU FEDER, the COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS), and the project TIN2011-23283 of the *Ministerio de Economía y Competitividad*.

References

[170] J. I. Aliaga, J. Pérez, E. S. Quintana-Ortí, and H. Anzt. Reformulated Conjugate Gradient for the Energy-Aware Solution of Linear Systems on GPUs. In *42nd Int. Conference on Parallel Processing (ICPP)*, pages 320–329, 2013.

[171] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corp., December 2008.

[172] NVIDIA Coorporation. Dynamic parallelism in CUDA. `http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf`, February 2015.

[173] J. DiMarco and M. Taufer. Performance impact of dynamic parallelism on different clustering algorithms. volume 8752, pages 87520E–87520E–8, 2013.

[174] J. Dong, F. Wang, and B. Yuan. Accelerating BIRCH for clustering large scale streaming data using CUDA dynamic parallelism. In *IDEAL-2013*, volume 8206 of *Lecture Notes in Computer Science*, pages 409–416. Springer Berlin Heidelberg, 2013.

[175] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Pub., San Francisco, 2003.

[176] L. Oden, B. Klenk, and H. Froning. Energy-efficient stencil computations on distributed gpus using dynamic parallelism and gpu-controlled communication. In *Energy Efficient Supercomputing Workshop (E2SC), 2014*, pages 31–40, 2014.

[177] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003.

[178] F. Wang, J. Dong, and B. Yuan. Graph-based substructure pattern mining using cuda dynamic parallelism. In *IDEAL-2013*, volume 8206 of *Lecture Notes in Computer Science*, pages 342–349. Springer Berlin Heidelberg, 2013.

[179] J. Wang and S. Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured gpu applications. In *2014 IEEE International Symposium on Workload Characterization*, 2014.

[180] Y. Yang, C. Li, and H. Zhou. Cuda-np: Realizing nested thread-level parallelism in gpgpu applications. *Journal of Computer Science and Technology*, 30(1):3–19, 2015.

CAPÍTULO 7

Conclusiones y líneas abiertas de investigación

En este capítulo se discuten las conclusiones y aportaciones más relevantes de los trabajos realizados a lo largo de esta Tesis Doctoral, junto con una serie de líneas abiertas de investigación.

Seguidamente se presenta la estructura de este capítulo. En la Sección 7.1 se analiza la validez de diferentes arquitecturas paralelas para la resolución de sistemas lineales dispersos, donde se constata la idoneidad de las GPUs para el cálculo de una versión básica del CG. El Capítulo 3 extiende este trabajo, en el que se incorporan diferentes versiones optimizadas del CG, centradas en la optimización de la operación SpMV, e incorporando técnicas de *"fusionado de kernels CUDA"* sólo en el caso de las GPUs. En la Sección 7.2 se discute la necesidad de reducir el consumo energético de las GPUs, cambiando su modo de ejecución de *polling* a *blocking*. La brusca bajada de prestaciones llevó a desarrollar metodologías de programación alternativas, lo que condujo al desarrollo de *"técnicas de fusionado de kernels CUDA"* (Capítulo 4). Las mejoras obtenidas en rendimiento/energía, alentó el desarrollo de una *"metodología de fusionado de kernels"* de un modo sistemático y eficiente (Capítulo 5). En la Sección 7.3 se argumenta la necesidad de ampliar los trabajos realizados de *"fusionado de kernels CUDA"* para incorporar *"paralelismo dinámico"*, posibilitando la ejecución desacoplada de resolutores en la GPU (Capítulo 6). Finalmente, la Sección 7.4 incluye una lista de publicaciones derivadas del trabajo realizado, mientras que en la sección 7.5 se detallan las líneas abiertas de investigación que definirán el trabajo futuro.

7.1 Análisis de arquitecturas paralelas

Actualmente, la computación de altas prestaciones (HPC, *High Performance Computing*) supone un alto coste energético que produce un fuerte impacto tanto en el aspecto económico como social y medioambiental. Una de las razones es que la producción eléctrica requiere un elevado consumo de recursos fósiles [186], tendencia que puede alargarse más allá del año 2040 [187]. Por lo que respecta al diseño de sistemas de cómputo de HPC, el principal objetivo siempre ha sido incrementar la cantidad de operaciones de coma flotante por segundo (FLOPS, *Floating-point Operations per Second*), como se puede comprobar en la lista TOP500 [188]. En esta carrera, no se tiene en cuenta ni la energía consumida para alcanzar ese rendimiento, ni los sofisticados sistemas de refrigeración que requieren (por cada megavatio de cómputo se consumen 0.7 megavatios de refrigeración), que todavía aumentan más el consumo energético. Afortunadamente, esta tendencia está cambiando y ahora no sólo se considera el rendimiento sino también su consumo, utilizando una métrica de rendimiento por Vatio (GFLOPS/W). Prueba de ello es que, desde 2007, la lista Green500 [189]

clasifica a los supercomputadores por su eficiencia energética. Todos estos hechos, propiciaron la idea de que ya no se puede considerar la construcción de supercomputadores de alto rendimiento a cualquier precio, sino que además, se debe considerar su eficiencia energética, cuyo objetivo fundamental es minimizar el impacto medioambiental y asumir unos costes equilibrados. Esta nueva perspectiva en la computación dio lugar al nacimiento del concepto de computación verde o ecológica (*Green Computing*) [190].

En el mundo real, existen infinidad de aspectos (sistemas naturales y problemas) que pueden ser modelados mediante Ecuaciones en Derivadas Parciales (EDPs), que sirven de apoyo en la realización de predicciones de la evolución de los mismos, ayudando a realizar descubrimientos científicos y mejoras en proyectos de ingeniería. La reformulación del problema en forma débil o variacional, su discretización en elementos finitos y la proyección del problema sobre el espacio de elementos finitos, lo convierte en un problema de álgebra lineal, en la mayoría de las ocasiones definido por matrices dispersas de gran dimensión. Debido al tamaño de estos sistemas y a las restricciones energéticas imperantes para resolverlos, es conveniente hacer uso de la simulación por computador consciente de la energía. El incremento de los problemas a estudio y el imparable aumento de sus tamaños, ha llevado a que la resolución de sistemas dispersos ocupe cada vez más tiempo en los servidores HPC. La mayor influencia de los sistemas dispersos justificó el desarrollo del nuevo *benchmark* HPCG (*High Performance Conjugate Gradient Benchmark*) como complemento o incluso sustituto de LINPACK (*High-Performance Linpack*), que es utilizado actualmente para clasificar supercomputadoras de la lista TOP500. Mientras que LINPACK resuelve sistemas lineales densos, HPCG resuelve sistemas lineales dispersos de gran dimensión, apoyándose en el algoritmo del gradiente conjugado precondicionado.

El diseño del *benchmark* HPCG y su impacto en la medición de la capacidad de sistemas HPC, fue el motivo principal para realizar un estudio completo sobre el comportamiento rendimiento/energía de una implementación básica del CG utilizando aritmética de simple precisión (SP), ejecutada sobre una muestra representativa de arquitecturas *manycore* y *multicore*. Aunque el uso de la aritmética de doble precisión (DP) es obligatoria para la solución de sistemas de ecuaciones lineales dispersos, en [191] se muestra cómo el uso de precisión mixta SP-DP, combinada con un refinamiento iterativo, mejora el tiempo de ejecución y el consumo de energía. Es por ello que el estudio de las arquitecturas incluido en [PPAM:2013] [181] utiliza aritmética de SP. Las arquitecturas objeto del estudio fueron las siguientes:

- Procesadores multinúcleo de propósito general (Intel Xeon E5504 y E5-2620, AMD Opteron 6128 y 6276).
- Procesador de señal digital multinúcleo de baja potencia (Texas Instruments C6678).
- Procesadores multinúcleo de baja potencia (ARM Cortex A9, Intel Atom D510).
- GPUs con diferentes capacidades de cálculo (NVIDIA Quadro M1000, Tesla C2050, y Kepler K20).

En los resultados obtenidos, se observó que la tasa de (GFLOPS/W) alcanzada por las arquitecturas *manycore* (GPUs) puede ser equiparable a la alcanzada por dispositivos de baja potencia, tales como el procesador Intel Atom, o el A9 de ARM, pero quedan muy por debajo de la eficiencia energética alcanzada por el DSP de Texas Instruments. Mientras que las GPU tradicionalmente alcanzan una alta eficiencia energética con un rendimiento excepcional, las arquitecturas de baja potencia proporcionan una disipación de potencia inferior, pero utilizando menos núcleos y memorias más pequeñas. Esto reduce la idoneidad de estos dispositivos de baja potencia para la computación de propósito general, y los hace atractivos para su uso en aparatos móviles y sistemas embebidos (su objetivo original), así como para aplicaciones específicas. Los procesadores de propósito general convencionales no alcanzaron el rendimiento ni las tasas de eficiencia energética de las GPUs, pero en cambio fueron las únicas arquitecturas capaces de procesar todos los test de prueba, debido a su mayor capacidad de memoria. En resumen, los resultados incluidos en [PPAM:2013] [181] permitieron

concluir que las GPUs son las arquitecturas que presentan un mejor rendimiento (GFLOPS), mientras que a nivel energético (GFLOPS/W), presentan mejor resultado que los procesadores de propósito general, pero quedan lejos de los dispositivos de bajo consumo.

El Capítulo 3 presenta una extensión del estudio realizado en [PPAM:2013] [181], en el que se incorporan versiones optimizadas del CG, principalmente centradas en la operación SpMV sobre las mismas arquitecturas. En concreto, para las arquitecturas multinúcleo se utilizaron los formatos CSR, BCSR y CSB [192, 193, 194, 195, 196], mientras que para las GPUs se utilizaron los formatos ELLPACK, ELLR_T y SELL-P [197, 198, 199, 200, 201], y también se incluyó el "*fusionado de kernels CUDA*". Además, en el estudio se utilizó aritmética de DP, aunque como complemento final, también se comprobó el uso de SP para la GPU (Kepler) y un procesador de propósito general (Intel Bridge). Tambien se incorporaron al estudio dos procesadores de baja potencia, Exynos5 Octa (ARM Cortex A15) e Intel Atom S1260, sustituyendo éste último al procesador Intel Atom D510 utilizado en el trabajo anterior.

A diferencia del [PPAM:2013] [181], los resultados obtenidos en este estudio mostraron que el ratio de (GFLOPS/W) alcanzado por las arquitecturas *manycore* (GPUs) puede ser equiparable al de los dispositivos *multicore* de baja potencia (Intel Atom, el A9 de ARM o un DSP de Texas Instrument), siendo las arquitecturas *manycore* solo superadas de forma contundente por el procesador ARM Cortex A15. Por su parte, los procesadores convencionales de propósito general no alcanzaron los rendimientos de las GPUs ni las bajas tasas de consumo de rendimiento por Vatio de los dispositivos *multicore* de baja potencia, aunque sí proporcionaron un equilibrio muy interesante entre estos dos extremos. En resumen, los nuevos resultados mostraron que las GPUs seguían siendo las arquitecturas que mayor rendimiento mostraban, pero se mejoraron los resultados de consumo energético incluidos en el anterior estudio, debido al uso del "*fusionado de kernels CUDA*". Este trabajo concluye que las GPUs son indiscutiblemente arquitecturas idóneas para HPC de acuerdo con la tendencia actual de *Green computing* [190], por lo que su uso permite el desarrollo de una computación consciente de energía muy equilibrada en cuanto a rendimiento y eficiencia energética.

7.2 Fusionado de *kernels* CUDA

El análisis de las arquitecturas realizado y presentado en [PPAM:2013] [181] mostró el alto rendimiento que las GPUs podían alcanzar en la resolución del CG, pero dejó al descubierto la necesidad de mejorar su eficiencia energética. La primera idea fue configurar la GPU para que se sincronizase con la CPU en modo *blocking* en lugar de utilizar el modo *polling*, lo que produjo una drástica bajada del consumo energético, pero también una pérdida nada despreciable de rendimiento. Esta bajada de prestaciones aconsejó la búsqueda de alternativas, entre las que destaca el "*fusionado de kernels CUDA*", cuya **primera aproximación** (Apartado 7.2.1) fue incluida en [ICPP:2013] [183] (Capítulo 4). Debido a las mejoras alcanzadas tanto en rendimiento como en eficiencia energética, se estudiaron patrones de fusión entre distintos tipos de *kernels* que dieron lugar al desarrollo de una **metodología de fusionado de *kernels*** (Apartado 7.2.2), que fusiona *kernels* de forma sistemática y eficiente, y que dio lugar a un trabajo que se presentó en [EPAR:2015] [184] (Capítulo 5).

7.2.1 Primera aproximación

La primera solución para resolver la pérdida de rendimiento cuando la GPU está configurada en modo *blocking* fue acelerar los *kernels*, centrándose en los más costosos (SpMV) y los más complejos (DOT). Los trabajos sobre la primera operación se centraron en la implementación de las versiones "scalar kernel" (SSpMV) y "vector kernel" (VSpMV) incluidas en [202], para el caso de matrices almacenadas en formato CSR. Este artículo sólo presentaba códigos para *grids* 1D, lo que en algún caso puede limitar el tamaño de las matrices, por lo que fue necesario desarrollar códigos para *grids* 2D, especialmente útiles

para VSpMV [202]. Por lo que respecta a la implementación de DOT, los trabajos se centraron en la identificación de las dos fases en las que se descompone, el producto componente a componente y la posterior reducción, la segunda de las cuales requiere la ejecución de una secuencia de *kernels* para los que debe ajustarse el tamaño y el número de bloques para maximizar las prestaciones [181]. Estas mejoras no consiguieron resolver el problema, por lo que se optó por analizar con mayor detalle las diferencias de ambos modos de sincronización entre la CPU y la GPU.

En el modo *polling*, la CPU consulta de manera continua el estado de la GPU para saber cuándo ha finalizado, reduciendo al mínimo el tiempo de respuesta a costa del aumento del consumo energético. Por su parte, en el modo *blocking*, la CPU pasa a un estado de bajo consumo energético (estados C del procesador) hasta que la GPU le indica que ha finalizado, minimizando el consumo energético pero aumentando el tiempo de respuesta relacionado con el tránsito a un estado activo. Bajo esta perspectiva, la solución debería permitir desarrollar algoritmos que tuvieran la eficiencia energética del modo *blocking* con el rendimiento del modo *polling*. La clave fue disminuir el número de *kernels* que la CPU tiene que lanzar sobre la GPU para resolver un determinado problema, mediante técnicas de "*fusionado de kernels CUDA*". De este modo, se eliminan tiempos de lanzamiento de *kernels*, y tiempos de transferencia de información entre los espacios de memoria de la CPU y la GPU. Además, si la GPU está sincronizada en modo *blocking*, la reducción del número de *kernels* también elimina tiempos de espera de recuperación de la CPU desde el cambio de un estado de bajo consumo energético hasta otro activo, que permite que el coste de ejecución del modo *blocking* sea muy parecido al del modo *polling*.

El punto de partida de la investigación fue el desarrollo de una versión básica del CG utilizando rutinas de CUBLAS, salvo en las operaciones de tipo producto matriz-vector para las que se utilizaron los *kernels* SSpMV y VSpMV. Su implementación se desarrolló sobre el interfaz de programación de aplicaciones (API, *Application Programming Interface*) de CUBLAS "*Legacy cuBLAS API*", que obligó a que los escalares residieran en la memoria principal de la CPU, mientras que los vectores y la matriz que modeliza el problema se almacenaron en la memoria global de la GPU. Esta versión del CG se utilizó como referencia para comprobar todas las mejoras conseguidas a lo largo de los trabajos desarrollados en esta Tesis Doctoral. Con posterioridad, se desarrolló la versión CUDA del CG, en la que se diseñó un *kernel* CUDA específico para cada una de las operaciones del algoritmo (AXPY, SCAL, DOT, ...), en donde, cuando era posible, cada hebra de un *grid* sólo maneja un elemento de cada vector y su acceso se realiza de forma coalescente. En esta versión del CG, tanto los escalares como los vectores y la matriz que modeliza el problema residían en la memoria global de la GPU. Además, se introdujeron nuevas mejoras en DOT, entre las que destaca que en la fase de reducción se redujeron a la mitad el número de bloques y el número de hebras activas por bloque, sin disminuir el tamaño del segmento de vector que cada bloque del *grid* maneja como dato de entrada. De este modo se consigue un patrón de acceso coalescente a los mismos pero distinto al anterior, en donde cada hebra maneja dos datos. El análisis experimental, no incluido en ninguno de los artículos, mostró una leve reducción de las prestaciones en todos los casos, pero permitió concluir que VSpMV es la mejor opción en la mayoría de los casos, excepto para aquellos en el que el número de elementos de nulos de las filas es menor que el tamaño de *warp*, para los que SSpMV obtiene mejor rendimiento. Aunque el diseño de los algoritmos empeoró levemente el rendimiento, se buscaba la posibilidad de realizar fusiones, en una versión posterior del CG, con lo que se esperaba compensar esta bajada de rendimiento o incluso mejorarlo.

El desarrollo de *kernels* CUDA específicos permitió comprobar que era posible fundir dos *kernels* en uno único en aquellas operaciones sin dependencias, y en aquellas con dependencias cuya implementación realizara un acceso coalescente a los datos (vector/es) que comparten, lo que fue el fundamento para desarrollar la versión MERGE. En este proceso, la primera tarea fue caracterizar el acceso a los operandos de los *kernels* que permitiera localizar las fusiones que se pudieran realizar en el CG. Así, se detectó que el acceso a los datos de entrada de SSpMV y VSpMV es no coalescente, mientras que sobre el dato de salida de SSpMV sí se realiza un acceso coalescente. La razón era que en SSpMV cada hebra obtiene un resultado en la misma posición del vector de salida que la posición del vector/es de entrada donde toma los datos, a

diferencia de VSpMV, que obtiene un resultado por cada *warp*. Respecto del DOT, todos sus *kernels* tienen un acceso coalescente sobre los datos de entrada, pero los *kernels* que implementan la reducción no lo tienen sobre su dato de salida. Por su parte, el resto de operaciones sobre vectores tienen un acceso coalescente sobre los datos de entrada y salida, mientras que para las operaciones sobre escalares hay que controlar los riesgos RAW. Fruto de ese análisis se obtuvieron los esquemas de fusiones detallados en la Tabla 7.1.

	CG con SSpMV	CG con VSpMV
Fusión 1	SSpMV + DOT_1 + 2 SCAL	VSpMV + SCAL
Fusión 2	DOT_2 + SCAL	DOT_1 + SCAL
Fusión 3	2 AXPY + DOT_1	DPT_2 + SCAL
Fusión 4	DOT_2	2 AXPY + DOT_1
Fusión 5	2 SCAL + 2 AXPY	DOT_2
Fusión 6		2 SCAL + 2 AXPY

Tabla 7.1: En la izquierda de la tabla se muestran las fusiones efectuadas en el CG al utilizar el algoritmo SSpMV, mientras que en la derecha se muestran las efectuadas al utilizar el algoritmo VSpMV. Ambos esquemas se corresponden con los del trabajo presentado en [ICPP:2013] [183] (Capítulo 4).

Una propiedad clave que posibilitó el fusionado de *kernels*, y que además proporcionó un incremento en el rendimiento, fue el particular diseño de la operación DOT. Esta operación fue dividida en dos partes bien diferenciadas. La primera parte, DOT_1, incluye el producto componente a componente y la primera reducción. Esta primera reducción es coalescente con respecto a sus datos de entrada, pero no respecto a su salida. La segunda parte, DOT_2, es una función que incluye un bucle en el que, en cada iteración, se ejecuta un *kernel* de reducción. Puesto que cada *kernel* no puede ser fundido con un *kernel* anterior, ni uno posterior, cada uno se diseñó de acuerdo a la mejora descrita con anterioridad, de modo que cada hebra maneja más de un elemento. Este esquema de funcionamiento hace que el resultado de la operación se pueda producir al final de cualquiera de estas dos partes, en función del tamaño del vector, aunque siempre será la hebra 0 del bloque 0 la encargada de escribir el resultado. Esta es la razón por lo que hay operaciones escalares que se repiten tras DOT_1 y DOT_2, siendo siempre realizadas por la hebra 0 correspondiente.

Otra mejora fue reducir el número de transferencias de información entre la CPU y la GPU, evitando el envío del valor que controla el fin del algoritmo en cada iteración del bucle. Tras diferentes pruebas se comprobó que verificar el fin del CG cada 10 iteraciones era un valor de compromiso suficiente para obtener buenos resultados en rendimiento y eficiencia energética, razón por la que esta variante se nombró como MERGE_10.

La evaluación de todas estas variantes permitió valorar las prestaciones de las diferentes versiones, tanto en modo *polling* como en modo *blocking*, utilizando como referencia la ejecución en modo *polling* de la versión CUBLAS. Con las pruebas realizadas, el equilibrio máximo entre rendimiento y consumo energético se alcanzó con la versión MERGE_10 en modo *blocking*, que mostró una mejora media de rendimiento de un 2.53 %, mientras que la eficiencia energética se incrementó entre rango del 25-30 %.

7.2.2 Metodología de fusionado de *kernels*

Los buenos resultados obtenidos en el trabajo presentado en [ICPP:2013] [183] (Capítulo 4) alentaron la búsqueda de patrones y criterios de fusionado de *kernels* que pudiesen llevar al desarrollo de un modelo sistemático, con el fin de aplicar estos conocimientos como asistencia tecnológica de compiladores o ayuda para programadores CUDA. Este fue el origen del desarrollo de una metodología para automatizar el *"fusionado de kernels CUDA"*, trabajo que se presentó en [EPAR:2015] [184] (Capítulo 5), y que constituye

la principal aportación de esta Tesis Doctoral. La clave de esta metodología es la caracterización del acceso a los datos de entrada/salida de los *kernels*, que junto con el grafo de dependencias del algoritmo asociado, representado mediante un DAG (Grafo Acíclico Dirigido), permite decidir qué *kernels* deben fusionarse. El posterior análisis de las diferentes alternativas determina el esquema de fusiones que permite maximizar el rendimiento y reducir el consumo energético.

La caracterización de los *kernels* tiene en cuenta dos aspectos fundamentales: en primer lugar, la configuración dimensional del *grid* asociado, y, en segundo lugar, el análisis del acceso que sus hebras realizan a los datos (vector/es) de entrada/salida. Respecto a esta última premisa, los *kernels* se pueden clasificar según las siguiente reglas:

- Un *kernel* $\boldsymbol{K}$ hace un acceso *mapped* sobre un vector $\boldsymbol{v}$, si $\boldsymbol{K}$ tiene definida una hebra distinta por cada elemento de $\boldsymbol{v}$ y además su acceso es coalescente.
- Un *kernel* $\boldsymbol{K}$ hace un acceso *unmapped* si su acceso no es *mapped*.

Criterio 1. Los *kernels* $\boldsymbol{K}_1$ y $\boldsymbol{K}_2$ serán fundibles si coinciden en la dimensionalidad tanto de *grid* como de bloque (1D, 2D ó 3D), y además cumplen con alguna de las siguientes condiciones:

- No comparten datos.
- Realizan un acceso *mapped* sobre los datos compartidos, que son de salida para el primero y de entrada para el segundo.

Una mención especial merece la fusión de *kernels* que sólo realizan operaciones escalares con *kernels* que implementan operaciones vectoriales, cuyo manejo incorrecto puede generar riesgos de datos del tipo lectura después de escritura (RAW, *Read after Write*). Las conclusiones varían ligeramente a lo expuesto en [ICPP:2013] [183] (Capítulo 4), por lo que el esquema de fusiones del CG en éste, es ligeramente distinto al del trabajo presentado en [EPAR:2015] [184] (Capítulo 5). La conclusión final fue que el lugar más apropiado para efectuar las fusiones de las operaciones escalares, cuando ésta es posible, es la etapa final de una operación de reducción, puesto que en ese punto se eliminan los riesgos de datos RAW que algunas operaciones escalares pueden causar si las ejecuta de forma independiente cada hebra de un *grid*. Por otra parte, cualquier secuencia de *kernels* escalares son fundibles entre sí, puesto que un *kernel* de este tipo se implementa definiendo un *grid* compuesto por un bloque de una hebra, y en ningún caso se pueden dar riesgos RAW.

Criterio 2. Sean $\{ \boldsymbol{K}_S, \boldsymbol{K}_V \}$ dos *kernels*, en donde $\boldsymbol{K}_S$ implementa una secuencia de operaciones escalares y $\boldsymbol{K}_V$ implementa operaciones vectoriales. Guardando el orden de su secuencia de aparición en el código origen $\{ \boldsymbol{K}_S, \boldsymbol{K}_V \}$ o $\{ \boldsymbol{K}_V, \boldsymbol{K}_S \}$, respectivamente la fusión $\{ \boldsymbol{K}_S + \boldsymbol{K}_V \}$ o $\{ \boldsymbol{K}_V + \boldsymbol{K}_S \}$ puede llevarse a cabo de forma directa, salvo que $\boldsymbol{K}_S$ incluya alguna variable que se utiliza como argumento de una operación y que luego actualiza (origina riesgos RAW). En este caso las fusiones se puede llevar a cabo si $\boldsymbol{K}_V$ incluye una operación de reducción, en la que las operaciones escalares las realizará la hebra 0 al final de tal operación, y sucede lo siguiente:

- La fusión $\{ \boldsymbol{K}_S + \boldsymbol{K}_V \}$ es posible si en la secuencia $\{ \boldsymbol{K}_S, \boldsymbol{K}_V \}$ no existen dependencias.
- La fusión $\{ \boldsymbol{K}_V + \boldsymbol{K}_S \}$ es posible con independencia de que en la secuencia $\{ \boldsymbol{K}_V, \boldsymbol{K}_S \}$ existan dependencias.

En el proceso de fusionado puede aparecer un *kernel* $\boldsymbol{K}_M$ que incluya tanto operaciones vectoriales como escalares, seguido de otro *kernel* que puede implementar operaciones o bien vectoriales $\boldsymbol{K}_V$ o escalares $\boldsymbol{K}_S$. En estas condiciones:

- La fusión { $\boldsymbol{K_M + K_V}$ } es posible si cumplen el **criterio 1**.
- La fusión { $\boldsymbol{K_M + K_S}$ } es posible si cumplen el **criterio 2**, pero aplicado sobre el conjunto del total de las operaciones escalares que sumen ambos *kernels*.

Tras caracterizar los *kernels* del algoritmo, se debe iniciar el análisis del DAG que permitirá localizar el mejor patrón de fusionado. Existen una serie de reglas, cuya validez ha sido comprobada empíricamente, que permiten obtener óptimas prestaciones:

- Si es posible, se reorganiza el DAG del algoritmo para que las operaciones escalares puedan fundirse con las operaciones que incluyan una reducción.
- Si es posible, se reorganiza el DAG para que el primer *kernel* realice un acceso *unmapped* sobre sus datos de entrada, posibilitando más oportunidades de realizar fusiones entre los *kernels* restantes.
- En general, se debe evitar fusionar *kernels* que implementen cualquier producto SpMV, a excepción de VSpMV, cuya fusión sí obtiene buenas prestaciones.

La metodología utiliza una estrategia ascendente, en la que a partir de *kernels* sencillos, se van construyendo *kernels* más complejos, llamados *macrokernels*. Este proceso se inicia en el primer nodo del DAG, buscando los *kernels* que pueden ser fundidos con él, y finaliza cuando ya no es posible realizar ningún fusionado. Dentro de cada *macrokernel* debe quedar claro cuál es la dependencia de las diferentes operaciones para asegurar que la posterior implementación se realiza correctamente.

El Capítulo 5 incluye la aplicación de esta metodología sobre 6 métodos numéricos (CG, PCG, BiCG, PBiCG, BiCGSTAB, PBiCGSTAB), utilizando 3 diferentes implementaciones de SpMV (SSpMV, VSpMV, ELLSpMV), y comparando las prestaciones de 5 implementaciones de cada algoritmo (CUBLASL, CUBLASN, CUDA, MERGE, MERGE_10) sobre los dos modos de funcionamiento de la GPU (*blocking* y *polling*). La Tabla 7.2 muestra el número de *macrokernels* obtenido para cada algoritmo cuando se utiliza el producto SSpMV.

(SSpMV)	Número de *kernels*	Número de *macrokernels*
CG	10	5
PCG	13	5
BICG	15	5
PBICG	17	5
BICGstab	18	8
PBICGstab	20	8

Tabla 7.2: Para cada método iterativo, la tabla muestra el número de *kernels* que lo compone, y el número de *macrokernels* que la aplicación del *"método de fusionado de kernels CUDA"* ha obtenido. Estos resultados corresponden al trabajo presentado en [EPAR:2015] [184] (Capítulo 5).

La evaluación consideró cada algoritmo por separado, aunque en todos los casos se utilizó como referencia la versión CUBLASL_*polling* del algoritmo correspondiente. Los resultados obtenidos sobre 12 matrices de prueba mostraron que la version MERGE_10_*blocking* es la que obtuvo mejores prestaciones en todos los casos, alcanzando un incremento de rendimiento entre el 5.1 % y el 10.2 %, y una mejora energética entre el 4.0 % y el 20 % respecto del valor de referencia. Sin embargo, cuando se utilizó como referencia la versión MERGE_10_*polling*, no se observó ninguna diferencia en el rendimiento, pero en cambio el ahorro energético alcanzó el 10 %.

7.3 Paralelismo dinámico

El *"Paralelismo Dinámico"* se incorpora en la versión 5.0 del API de CUDA y se puede utilizar a partir de la familia Kepler de GPUs, permitiendo que un proceso (hebra) de la GPU pueda iniciar nuevos subprocesos (mallas de hebras) sin intervención de la CPU. Su uso en el marco de esta Tesis Doctoral permitió que un *kernel-padre*, configurado mediante un *grid* de un bloque con una única hebra, lance secuencialmente los *kernels-hijo* que implementan cada una de las operaciones de un algoritmo, y además controle la finalización de los mismos. De este modo, durante la ejecución del algoritmo, la GPU queda completamente desacoplada de la CPU, que bien realiza tareas en paralelo a la GPU, o bien entra en un estado de bajo consumo energético (C-estados del procesador). Bajo estas perspectivas, la explotación de esta característica podría mejorar el ahorro energético ya obtenido en los trabajos anteriores, por lo que se desarrolló una versión del CG que incorporó el *"Paralelismo Dinámico"*, y que cuyos resultados se presentaron en [PARC:2015] [185] (Capítulo 6).

La incorporación del *"Paralelismo Dinámico"* al código del CG implementado en el Capítulo 5 y presentado en [EPAR:2015] [184] permitió constatar que las fusiones proporcionaban las mismas ventajas, aumentando la eficiencia energética sin sacrificar rendimiento, pero los códigos eran más lentos que sin el uso de esta nueva característica. Se comprobó empíricamente que el origen de la bajada de prestaciones era la ejecución de los *kernels* asociados a la operación DOT y, más concretamente, el número de niveles en el árbol de llamadas a *kernels*, que en el caso de DOT_2 era igual o mayor a 3, por lo que se desarrolló una nueva versión del DOT que eliminara este problema.

La nueva DOT se formó únicamente con dos *kernels*, DOT_D_ini y DOT_D_fini, con lo cual la profundidad de lanzamientos se limitó a dos. Esta nueva versión era más rápida que la DOT presentada en el Capítulo 5, pero el acceso que el *kernel* DOT_D_ini realiza a sus datos de entrada no se puede clasificar como *mapped*, lo que obligó a rediseñar algunas fusiones. Aún así, este patrón de acceso tiene un especial interés, ya que es coalescente pero definiendo un *grid* con la mitad de bloques y la mitad de hebras por bloque, razón por la que a su caracterización sobre sus datos de entrada se la denominó como *half-mapped*. Este acceso pudo ser extendido a otras operaciones sencillas sobre vectores, constatándose que era posible fundir *kernels* con este tipo de acceso, por lo que fue necesario modificar/enriquecer la *"metodología de fusionado de kernels"* para que incorporase esta nueva opción.

Así pues, se pudo concluir que, al utilizar *"Paralelismo Dinámico"* es más importante limitar la profundidad de las llamadas a los *kernels* antes que aplicar fusionado. En cambio, si no se utiliza *"Paralelismo Dinámico"*, el fusionado es la tarea prioritaria. Los resultados de la combinación del *"Paralelísmo Dinámico"* y de la *"metodología de fusionado de kernels"* se presentaron en [PARC:2015] [185] (Capítulo 6), en donde se mostró que no hay diferencia en el rendimiento de las versiones optimizadas en modo *polling* y *blocking*, pero sí es muy grande el impacto en la eficiencia energética. Las pruebas realizadas utilizando como referencia la versión de CUBLASL en modo *polling*, mostraron ganancias muy significativas de rendimiento y eficiencia energética, en torno a un 3.65 % y un 14.23 % respectivamente.

7.4 Difusión de los resultados del trabajo

Los resultados del trabajo desarrollado en esta Tesis Doctoral han sido presentados en ámbitos de difusión científica de prestigio.

Artículo de revista

- J. Aliaga, H. Anzt, M. Castillo, J.C. Fernández, G. Léon, J. Pérez, and E. S. Quintana-Ortí. Unveiling the performance-energy trade-off in iterative linear system solvers for multithreaded processors. In Concurrency and Computation: Practice & Experience, Vol. 27(4), pp. 885-904, 2015.

Actas de conferencias internacionales

- J. Aliaga, H. Anzt, M. Castillo, J. Fernández, G. Léon, J. Pérez, and E. S. Quintana-Ortí. Performance and energy analysis of the iterative solution of sparse linear systems on multicore and manycore architectures. In Lecture Notes in Computer Science, 10th Int. Conf. on Parallel Processing and Applied Mathematics – PPAM 2013, 2014.

- J. I. Aliaga, J. Pérez, E. S. Quintana-Ortí, and H. Anzt. Reformulated Conjugate Gradient for the energy-aware solution of linear systems on GPUs. In Parallel Processing (ICPP), 2013 42nd International Conference on, pages 320–329, Oct 2013.

- J. Aliaga, J. Pérez, and E. S. Quintana-Ortí. Systematic fusion of CUDA kernels for iterative sparse linear system solvers. In Lecture Notes in Computer Science 9233, Euro-Par 2015, pp. 675-686, Viena (Austria), 2015.

- J. Aliaga, D. Davidović , J. Pérez, and E. S. Quintana-Ortí. Harnessing CUDA dynamic parallelism in the solution of sparse linear systems. In Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, pages 217–226, Edinburgh, Scotland, (UK), 1-4 September 2015.

7.5 Trabajo futuro

La Tesis Doctoral cubre con el objetivo general fijado en el inicio de la misma: el diseño, desarrollo y evaluación de nuevos resolutores de sistemas de ecuaciones lineales para GPUs, que mejoren la eficiencia energética de este tipo de cálculos sin sacrificar su rendimiento computacional. En esta misma línea de trabajo, pueden identificarse una serie de problemas no resueltos hasta la fecha, que podrían mejorar los desarrollos realizados en esta Tesis Doctoral en la misma dirección, y que constituyen líneas abiertas de investigación:

- El fusionado de alguna de las versiones de SpMV con otras operaciones fue una de las claves para mejorar las prestaciones de los algoritmos obtenidos por la *"metodología de fusionado de kernels"*. El acceso *half-mapped* no ha sido utilizado en ninguna de las versiones del SpMV, y su uso podría tener un impacto importante en las prestaciones de los algoritmos, ya que permitiría que los *kernels* asociados se pudieran fusionar con otros *kernels*. Así pues, una línea de investigación viable es el desarrollo de versiones *half-mapped* de SpMV para analizar el impacto del fusionado sobre los *kernels* resultado, y que también se pudiera combinar opcionalmente con Paralelismo Dinámico.

- El manejo de la jerarquía de memoria no fue considerada en la definición de la metodología de fusionado, aún cuando sí se utiliza de modo implícito al crear múltiples fusiones de DOT en los resolutores, sacando provecho de la reutilización de la memoria compartida mediante el uso de aritmética de punteros. También es posible almacenar en un registro, o en memoria compartida, algún dato que sólo es utilizado por una hebra del *macrokernel*, o incluso utilizar la memoria de textura para almacenar constantes y/o valores variables precalculados, como por ejemplo la incorporación de nuevas estrategias de optimización para acelerar los cálculos de los índices calculados en los *kernels*. Una posibilidad es el uso de memoria de textura y/o constantes, con la ventaja de que ambas disponen de cachés muy efectivas.

- Desde Kepler se incorporaron a los aceleradores gráficos nuevas características muy potentes y versátiles, como por ejemplo *Shuffle Instruction*, mejoras en las operaciones atómicas y gestión de unidades

de Textura, etc. ..., que pueden tener un fuerte impacto en el aumento del rendimiento de los algoritmos implementados. Una vía de mejora es el rediseño de las operaciones implementadas y adaptarlas a cada nueva característica de las distintas familias de GPUs que se van presentando, incluyendo además un software de selección en tiempo de ejecución.

- Las GPUs actuales incluyen varias colas de trabajo, cada una de ellas puede ser utilizada en paralelo si se definen diferentes *streams*. Como mejora al *"fusionado de kernels"*, y con independencia de la inclusión de *"Paralelismo Dinámico"*, la incorporación del uso de las diferentes colas de trabajo puede acelerar la ejecución de aquellos *macrokernels* que tuviesen en su composición *kernels* independientes entre sí.

- También es de gran interés el desarrollo de herramientas que apliquen de modo automático la metodología de fusionado sobre un código. En entornos de programación CUDA con una sintaxis rígida y bien determinada (CUBLAS, cuSPARSE, etc...), se podría sustituir las operaciones, por algoritmos optimizados y bien tipificados (*mapped, half-mapped*). Esto sería la base para desarrollar alguna herramienta transcompilador (*source-to-source compiler*), que aplicara de forma autónoma el *"modelo sistemático de fusionado de kernels CUDA"* desarrollado en esta Tesis Doctoral. Por el contrario, una herramienta de *"fusionado de kernels CUDA"* en un entorno cuya sintaxis sea libre, requiere de un proceso más complejo, aunque no imposible, dando lugar al desarrollo de un software de caracterización de los *kernels* que componen un código CUDA. Quizás está última línea de investigación sea de mayor envergadura, y con un objetivo más ambicioso, pero promete proporcionar herramientas generales para optimizar el consumo energético sin sacrificar rendimiento allá donde se requiera el uso de GPUs.

Bibliografía

[181] J. Aliaga, H. Anzt, M. Castillo, J. Fernández, G. Léon, J. Pérez, and E. S. Quintana-Ortí. Performance and energy analysis of the iterative solution of sparse linear systems on multicore and manycore architectures. In *Lecture Notes in Computer Science, 10th Int. Conf. on Parallel Processing and Applied Mathematics – PPAM 2013*, 2014.

[182] J. Aliaga, H. Anzt, M. Castillo, J.C. Fernández, G. Léon, J. Pérez, and E. S. Quintana-Ortí. Unveiling the performance-energy trade-off in iterative linear system solvers for multithreaded processors. In *Concurrency and Computation: Practice & Experience, Vol. 27(4), pp. 885-904*, 2015.

[183] J. I. Aliaga, J. Pérez, E. S. Quintana-Ortí, and H. Anzt. Reformulated Conjugate Gradient for the energy-aware solution of linear systems on GPUs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 320–329, Oct 2013.

[184] J. Aliaga, J. Pérez, and E. S. Quintana-Ortí. Systematic fusion of CUDA kernels for iterative sparse linear system solvers. In *Lecture Notes in Computer Science 9233, Euro-Par 2015, pp. 675-686, Viena (Austria)*, 2015.

[185] J. Aliaga, D. Davidović, J. Pérez, and E. S. Quintana-Ortí. Harnessing CUDA dynamic parallelism in the solution of sparse linear systems. In *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, pages 217–226, Edinburgh, Scotland, (UK)*, 1-4 September 2015.

[186] IEA International Energy Agency. IEA statistics. Monthly electricity statistics. January 2016. http://www.iea.org/statistics/.

[187] EIA U.S. Energy Information Administration. Annual Energy Outlook 2015 with projections to 2040. http://www.eia.gov/.

[188] The Top500 list, 2014. http://www.top500.org.

[189] The Green500 list, 2014. http://www.green500.org.

[190] Mathew, Ajoy P. Green Computing, 2008.

[191] H. Anzt, V. Heuveline, J. I. Aliaga, M. Castillo, J. C. Fernández, R. Mayo, and E. S. Quintana-Ortí. Analysis and optimization of power consumption in the iterative solution of sparse linear systems on multi-core and many-core platforms. In *Int. Green Computing Conf. Workshops (IGCC)*, pages 1–6, 2011.

[192] A. Buluç et al. *Compressed Sparse Block Library*. Combinatorial Scientific Computing Lab at the University of California, Santa Barbara, 2014.

[193] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 233–244, New York, NY, USA, 2009. ACM.

[194] Intel. *Intel Math Kernel Library reference manual (release 10.3)*, 2014. Document Number: 630813-053US.

[195] E. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, Feb. 2004.

[196] R. Vuduc. *Automatic performance tuning of sparse matrix kernels*. Ph.D. dissertation, Univ. California, Berkeley, January 2004.

[197] H. Anzt, S. Tomov, and J. Dongarra. Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs. Technical Report ut-eecs-14-727, University of Tennessee, March 2014.

[198] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop. A unified sparse matrix data format for modern processors with wide SIMD units. *CoRR*, abs/1307.6209, 2013.

[199] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, pages 111–125, Berlin, Heidelberg, 2010. Springer-Verlag.

[200] F. Vázquez, J. J. Fernández, and E. M. Garzón. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience*, 23(8):815–826, 2011.

[201] F. Vázquez, J. J. Fernández, and E. M. Garzón. Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach. *Parallel Computing*, 38(8):408 – 420, 2012.

[202] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corp., December 2008.