Tesis Doctoral

Desarrollo de herramientas altamente escalables para la resolución eficiente de problemas de dispersión electromagnética y acústica

Miguel López Portugués

Director: Jesús A. López Fernández

Programa de Doctorado en Tecnologías de la Información y Comunicaciones en Redes Móviles

2017

Resumen

En los últimos años, la reducción del ruido está cobrando una considerable atención por parte de las autoridades competentes, debido al rechazo que genera la contaminación acústica en la población. Como consecuencia de lo anterior, el control de ruido en las aeronaves (especialmente durante despegues y aterrizajes) se ha convertido en un tema de especial interés, ya que los requisitos relativos al ruido ambiental son cada vez más restrictivos. No obstante, para explorar las posibilidades que se presentan en la fase de diseño de una aeronave, deben aplicarse técnicas de predicción acústica que permitan caracterizar el comportamiento de los modelos desde el punto de vista de la presión sonora dispersada. El análisis preciso de dicho problema puede requerir una gran cantidad de recursos computacionales, por lo que se debe recurrir a herramientas en las que se combine el uso de esquemas aceleradores junto con técnicas propias de la computación de altas prestaciones si se pretende incrementar el rango de aplicación de las técnicas de modelado y predicción.

Por su parte, el problema electromagnético inverso resulta de gran interés en múltiples aplicaciones, entre las que se incluyen la caracterización de antenas y la imagen electromagnética. En ambos casos, el coste computacional asociado a la resolución de dichos problemas puede suponer un cuello de botella, especialmente con problemas eléctricamente grandes, ya que el tiempo de cálculo necesario para obtener una solución puede no satisfacer los requisitos de tiempo real o cuasi-real de algunas aplicaciones como, por ejemplo, las relacionadas con la seguridad.

El objetivo de esta Tesis consiste en el desarrollo de un conjunto de herramientas software que permitan la resolución eficiente y precisa de

diferentes problemas dentro del ámbito de la dispersión acústica y electromagnética. En el problema de dispersión acústica, se persiguen varios objetivos: posibilitar el análisis de objetos de gran tamaño acústico, reducir los tiempos de resolución de los problemas analizados y mejorar la eficiencia energética de las herramientas desarrolladas. En el problema electromagnético inverso, el objetivo consiste en posibilitar análisis en tiempo real o cuasi-real para problemas de tamaño moderado. En ambos casos, se llevan a cabo modificaciones en la algoritmia inicial para su paralelización y se recurre, además, al uso de aceleradores hardware.

Abstract

In recent years, noise reduction is getting considerable attention from the competent authorities, given that people do not tolerate noise-related pollution. As a result, noise control in aircrafts (especially during take-offs and landings) has become a major topic, since environmental noise requirements become increasingly restrictive. Consequently, acoustic prediction techniques are applied to characterize the acoustic field scattered by different models, in order to explore the possibilities presented in the design process of a new aircraft. Nevertheless, the accurate analysis of the acoustic scattering problem may require a large amount of computational resources. Therefore, it is necessary to make use of tools that can exploit high performance computing, in addition to some accelerator schemes, in order to increase the applicability of modeling and prediction techniques.

The inverse electromagnetic problem can also be of great interest in many applications, for instance, antenna characterization and electromagnetic imaging. However, the computational cost for solving the beforementioned problems can be a bottleneck, especially with electrically large problems, since the calculation time that is needed to obtain a solution may not satisfy the real-time or quasi-real-time requirements of some applications, such as those related to security screening.

The objective of this Thesis is the development of software tools that allow accurate and efficient resolution of problems within the scope of acoustic and electromagnetic scattering. In relation to the acoustic scattering problem, several aims are considered: enabling the analysis of objects with large acoustic size, reducing the solution time of the analyzed problems and improving the energy-efficiency of the developed tools. Re-

garding the inverse electromagnetic problem, the goal is to enable real-time or quasi-real-time analysis for problems of moderate size. In both cases, several modifications are made to the initial algorithms in order to parallelize and exploit hardware-accelerators.

Lista de publicaciones

Publicaciones en revistas internacionales

[1] M. López-Portugués, J. A. López-Fernández, A. Rodríguez-Campa y J. Ranilla, “A GPGPU Solution of the FMM Near Interactions for Acoustic Scattering Problems”, Journal of Supercomputing, Vol. 58 Nº 3, (2011), 283–291.

[2] J. A. López-Fernández, M. López-Portugués, Y. Álvarez-López, C. García-González, D. Martínez y F. Las Heras Andrés, “Fast antenna characterization using the sources reconstruction method on graphics processors”, Progress In Electromagnetics Research, Vol. 126, (2012), 185–201.

[3] M. López-Portugués, J. A. López-Fernández, J. Menéndez-Canal, A. Rodríguez-Campa y J. Ranilla, “Acoustic scattering solver based on single level FMM for multi-GPU systems”, Journal of Parallel and Distributed Computing, Vol. 72 Nº 9, (2012), 1057–1064.

[4] M. López-Portugués, Y. Álvarez-López, J. A. López-Fernández, C. García-González, R. G. Ayestarán y F. Las Heras Andrés, “A multi-GPU sources reconstruction method for imaging applications”, Progress In Electromagnetics Research, Vol. 136, (2013), 703–724.

[5] M. López-Portugués, J. A. López-Fernández, J. Ranilla, R. G. Ayestarán y F. Las-Heras, “Parallelization of the FMM on distributed-memory GPGPU systems for acoustic-scattering prediction”, Journal of Supercomputing, Vol. 64 Nº 1, (2013), 17–27.

[6] M. López-Portugués, J. A. López-Fernández, N. Díaz-Gracia, R. G. Ayestarán y J. Ranilla, "Aircraft noise scattering prediction using different accelerator architectures", Journal of Supercomputing, Vol. 70 N° 2, (2014), 612–622.

[7] M. López-Portugués, J. A. López-Fernández, J. Ranilla, R. G. Ayestarán y F. Las-Heras, "Using heterogeneous computing for scattering prediction in scenarios with several source configurations", Journal of Supercomputing, Vol. 73 N° 1, (2017), 57–74.

[8] J. A. López-Fernández, M. López-Portugués y J. Ranilla, "Improving the FMM performance using optimal group size on heterogeneous system architectures", Journal of Supercomputing., Vol. 73 N° 1, (2017), 291–301.

Contribuciones a congresos

- Internacionales:

[1] J. Menéndez Canal, M. López Portugués, J. A. López Fernández, A. Rodríguez-Campa y J. Ranilla, "Computing the near interactions of the FMM for acoustic scattering using GPUs", Proceedings of the 10th International Conference on Computational and Mathematical Methods in Science and Engineering (ISBN: 978-84-613-5510-5), junio de 2010, Almería (España).

[2] M. López-Portugués, J. A. López-Fernández, J. Ranilla, R. G. Ayestarán y F. Las-Heras, "A parallel solver using the Fast Multipole Method for noise problems", Proceedings of the 11th International Conference on Computational and Mathematical Methods in Science and Engineering (ISBN: 978-84-614-6167-7), junio de 2011, Alicante (España).

[3] J A. López Fernández, M. López-Portugués, R. G. Ayestarán y F. Las-Heras, "A GPU Accelerated Scattering Solver Based on the

Single Level Fast Multipole Method", 8th Iberian Meeting on Computational Electromagnetics (VIII EIEC), noviembre de 2011, Sesimbra (Portugal).

[4] M. López-Portugués, J. A. López-Fernández, D. Marful-Díaz, R. G. Ayestarán y F. Las-Heras, "Computationally efficient algorithm for mesh refinement based on octrees and linked lists", Proceedings of the 13th International Conference on Computational and Mathematical Methods in Science and Engineering (ISBN: 978-84-616-2723-3), junio de 2013, Cabo de Gata (España).

[5] M. López-Portugués, J. A. López-Fernández, J. Ranilla y R. G. Ayestarán, "Aircraft noise scattering computation using GPUs", Proceedings of the 13th International Conference on Computational and Mathematical Methods in Science and Engineering (ISBN: 978-84-616-2723-3), junio de 2013, Cabo de Gata (España).

[6] M. López-Portugués, N. Díaz-Gracia, J. A. López-Fernández, J. Ranilla y R. G. Ayestarán, "Solving Noise Prediction Problems with Several Noise Source Configurations Using Multicore and Manycore Architectures", Proceedings of the 15th International Conference on Computational and Mathematical Methods in Science and Engineering (ISBN: 978-84-617-2230-3), julio de 2015, Rota (España).

[7] J. A. López-Fernández, M. López-Portugués, J. Ranilla, R. G. Ayestarán y F. Las-Heras, "How group size influences the efficiency of FMM", Proceedings of the 16th International Conference on Computational and Mathematical Methods in Science and Engineering (ISBN: 978-84-608-6082-2), julio de 2016, Rota (España).

- Nacionales:

[1] M. López Portugués, J. A. López Fernández, R. G. Ayestarán y F. Las-Heras, "Método de los multipolos rápidos (FMM) aplicado a dispersión acústica empleando aceleración hardware", Libro de actas del XXVI Simposium Nacional de la Unión Científica Internacional de Radio (ISBN: 978-84-933934-4-1), septiembre de 2011, Leganés (España).

[2] D. Marful Díaz, J. A. López Fernández, M. López Portugués, R. G. Ayestarán y F. Las-Heras, “Algoritmo eficiente para el refinamiento de mallas mediante el uso de octrees”, Libro de actas del XXVII Simposium Nacional de la Unión Científica Internacional de Radio (ISBN: 978-84-695-4327-6), septiembre de 2012, Elche (España).

[3] J. A. López Fernández, M. López Portugués, Y. Álvarez, C. García, R. G. Ayestarán y F. Las-Heras, “Aceleración del método de reconstrucción de fuentes usando procesadores gráficos”, Tipo de participación: Publicación y presentación oral. Libro de actas del XXVII Simposium Nacional de la Unión Científica Internacional de Radio (ISBN: 978-84-695-4327-6), septiembre de 2012, Elche (España).

[4] N. Díaz-Gracia, M. López-Portugués, J. A. López-Fernández, M. Alonso-González, R. Cortina, R. G. Ayestarán, F. Las-Heras y J. Ranilla, “Computación eficiente del ruido generado por aeronaves usando CUDA”, Libro de actas del XXIX Simposium Nacional de la Unión Científica Internacional de Radio (ISBN: 978-84-9048-264-3), septiembre de 2014, Valencia (España).

Abreviaturas

ABM	*Antena Bajo Medida*
ACA	*Adaptive Cross Approximation*
API	*Application Programming Interface*
ASCII	*American Standard Code for Information Interchange*
BEM	*Boundary Elements Method*
Bi-CGSTAB	*BiConjugate Gradient Stabilized*
BLAS	*Basic Linear Algebra Subprograms*
bMVP	*baseline Matrix-Vector Product*
CAO	*Compiler-Assisted Offload*
CBIE	*Conventional Boundary Integral Equation*
CG	*Conjugate Gradient*
CGNR	*Conjugate Gradient Normal equation Residual*
CPU	*Central Processing Unit*
CSI	*Contrast Source Inversion*
CUDA	*Compute Unified Device Architecture* (originariamente)
DDR	*Double Data Rate*
FBM	*Forward-Backward Method*
FFT	*Fast Fourier Transform*
FFTW	*Fastest Fourier Transform in the West*
FMM	*Fast Multipole Method*
FPGA	*Field Programmable Gate Array*

GCC	*GNU Compiler Collection*
GMRES	*Generalized Minimal Residual*
GNU	*GNU's Not Unix*
GPGPU	*General-Purpose computation on Graphics Processing Units*
GPU	*Graphics Processing Unit*
HBIE	*Hypersingular Boundary Integral Equation*
ICC	*Intel C/C++ Compiler*
ISA	*Instruction Set Architecture*
MATLAB	*MATrix LABoratory*
MIMD	*Multiple Instruction streams, Multiple Data streams*
MINRES	*Minimum Residual*
MKL	*Math Kernel Library*
MLACA	*Multi-Level Adaptive Cross Approximation*
MLFMA	*Multi-Level Fast Multipole Algorithm*
MLMDA	*Multi-Level Matrix Decomposition Algorithm*
mMVP	*multiple Matrix-Vector Product*
MoM	*Method of Moments*
MPI	*Message Passing Interface*
MR-CSI	*Multiplicative-Regularized Contrast Source Inversion*
MST	*Memory Saving Technique*
MVP	*Matrix-Vector Product*
PCI	*Peripheral Component Interconnect*
QDR	*Quad Data Rate*
RMS	*Root Mean Square*
RMSE	*Root Mean Square Error*
SIMD	*Single Instruction stream, Multiple Data streams*
SIMT	*Single-Instruction, Multiple-Thread*
SMP	*Symmetric Multi-Processor*

SRM	*Sources Reconstruction Method*
SVD	*Singular Value Decomposition*

Notación

j	unidad imaginaria
$\mathcal{I}m\{a\}$	parte imaginaria de la magnitud compleja a
$\cdot$	producto escalar
$\times$	producto vectorial
α	coeficiente de acoplo
β	tiempo necesario para enviar un elemento por un canal de comunicación
ϵ	error residual
ϵ_{rel}	error relativo
ϵ_{RMS}	error RMS
η	impedancia característica
κ	número de onda
λ	longitud de onda
f	frecuencia
F	factor computacional para modelar tiempos por operación
E	campo eléctrico
G	función de Green en espacio libre
J	densidad de corriente eléctrica
M	densidad de corriente magnética
$\mathcal{O}$	complejidad algorítmica (notación "O-grande")

P	presión acústica
S	aceleración computacional
T	tiempo
$\mathcal{T}$	operador de traslación

Índice general

1. Introducción **1**
1.1. Motivación 1
1.2. Antecedentes y estado de la cuestión 4
1.2.1. Problema directo de dispersión acústica 4
1.2.1.1. Métodos de resolución y esquemas aceleradores 6
1.2.2. Problema electromagnético inverso 9
1.2.2.1. Métodos de resolución y esquemas aceleradores 11
1.2.3. Paralelización y aceleración hardware 12
1.3. Arquitecturas y tecnologías utilizadas en la Tesis 14
1.3.1. Procesadores multinúcleo y sistemas de memoria compartida 15
1.3.2. Sistemas de memoria distribuida 17
1.3.3. Procesadores gráficos para computación de propósito general 19
1.3.4. Coprocesadores Xeon Phi 22
1.4. Contribuciones de la Tesis 24
1.5. Estructura de la memoria de la Tesis 26

2. FMM paralelo y heterogéneo aplicado a acústica **29**
2.1. Descripción del cálculo de la presión acústica 31
2.2. Algoritmo FMM aplicado al problema de dispersión acústica 34
2.2.1. Inicialización 37
2.2.2. Interacciones cercanas 40
2.2.3. Interacciones lejanas 41

2.2.3.1. Agregación 41
2.2.3.2. Traslación 42
2.2.3.3. Desagregación 43
2.2.4. Suma de interacciones 44
2.3. Paralelización para aceleradores gráficos 44
2.3.1. Decisiones de diseño de la implementación propia . . 45
2.3.1.1. Uso de *octrees* 45
2.3.1.2. Almacenamiento de matrices y otras estructuras de datos 45
2.3.1.3. Particionado 46
2.3.1.4. Comunicaciones 49
2.3.1.5. Balanceo de carga 50
2.3.2. Inicialización 52
2.3.2.1. Selección del tamaño de grupo 52
2.3.2.2. Reparto de la carga de trabajo 52
2.3.2.3. Distribución inicial de datos 53
2.3.2.4. Operador de traslación 54
2.3.3. Interacciones cercanas 57
2.3.4. Agregación 61
2.3.5. Traslación 66
2.3.6. Desagregación 70
2.4. Resultados 73
2.4.1. Presión acústica 73
2.4.2. Validación 76

3. FMM-FFT paralelo y heterogéneo aplicado a acústica 81
3.1. Algoritmo FMM-FFT aplicado al problema de dispersión acústica 83
3.1.1. Inicialización 84
3.1.2. Interacciones cercanas 84
3.1.3. Interacciones lejanas 85
3.1.3.1. Agregación 85
3.1.3.2. Traslación 86
3.1.3.3. Desagregación 87
3.2. Paralelización para sistemas convencionales y heterogéneos . 88
3.2.1. Decisiones de diseño de la implementación propia . . 88

3.2.1.1. Particionado 88
3.2.1.2. Balanceo de carga 91
3.2.2. Inicialización 93
3.2.2.1. Selección del tamaño de grupo 93
3.2.2.2. Reparto de la carga de trabajo 94
3.2.2.3. Distribución inicial de datos 95
3.2.2.4. Operador de traslación 95
3.2.3. Interacciones cercanas 98
3.2.4. Agregación 100
3.2.5. Traslación 103
3.2.6. Desagregación 108
3.3. Resultados 110
3.3.1. Validación 111

4. Cálculo de la presión total en el problema de acústica 115
4.1. Descripción del cálculo de la presión acústica 116
4.1.1. Cálculo de la presión total 117
4.1.2. Cálculo de la presión dispersada 118
4.2. Paralelización para sistemas convencionales y heterogéneos . 120
4.2.1. Decisiones de diseño de la implementación propia . . 120
4.2.2. Técnica basada en producto matriz-vector (bMVP) . 123
4.2.3. Técnica basada en múltiples productos matriz-vector (mMVP) 127
4.2.4. Técnicas basadas en productos matriciales 132
4.2.4.1. *SmallBlocks* 132
4.2.4.2. *BigBlocks* 137
4.2.5. Solución heterogénea para GPU y Xeon Phi 141
4.3. Resultados 145
4.3.1. Presión acústica 145
4.3.2. Validación 149

5. SRM paralelo y heterogéneo aplicado al problema inverso 151
5.1. SRM aplicado a la caracterización de antenas 153
5.1.1. Paralelización para aceleradores gráficos 156
5.2. SRM aplicado a problemas de *imaging* 159
5.2.1. Paralelización para aceleradores gráficos 161

5.3. Resultados 163
5.3.1. Caracterización de antenas 164
5.3.2. Imagen electromagnética 167

6. Resultados 173
6.1. FMM 175
6.1.1. Tiempo de ejecución y escalabilidad 175
6.1.2. Eficiencia energética 186
6.1.3. Problemas de gran tamaño acústico 189
6.2. FMM-FFT 193
6.2.1. Tiempo de ejecución y escalabilidad 194
6.2.2. Eficiencia energética 200
6.2.3. Problema de gran tamaño acústico 203
6.3. Comparativa FMM vs. FMM-FFT 205
6.3.1. Tiempo de ejecución y escalabilidad 205
6.3.2. Eficiencia energética 209
6.3.3. Problema de gran tamaño acústico 211
6.4. bMVP, mMVP, *SmallBlocks*, *BigBlocks* y solución heterogénea 212
6.5. SRM 219
6.5.1. Problemas de caracterización de antenas 219
6.5.2. Problema de imagen electromagnética 220

7. Conclusiones y líneas futuras 227
7.1. Conclusiones 227
7.2. Líneas futuras 230

A. Sistemas utilizados y parámetros de ejecución 231
A.1. UniOvi - CMI 232
A.2. Microway - Tesla MD SimCluster 234
A.3. TSC-UNIOVI - Mopar 236
A.4. IRPCG - Manycores 239
A.5. TSC-UNIOVI - SimLin02 243
A.6. TSC-UNIOVI - Barracuda 244
A.7. TSC-UNIOVI - Roadrunner 246

Bibliografía 249

Índice de figuras

1.1. Representación esquemática de un sistema de memoria compartida. . . . 16
1.2. Representación esquemática de un sistema de memoria distribuida. . . . 18
1.3. Representación esquemática de un sistema heterogéneo basado en CPU + GPU. . . . 20
1.4. Representación esquemática de un sistema heterogéneo basado en CPU + Xeon Phi. . . . 23

2.1. Representación esquemática de las etapas del FMM. . . . 36
2.2. *Quadtree* con ordenación Z. . . . 38
2.3. Grupos cercanos y grupos lejanos. . . . 40
2.4. Orden de ejecución de las etapas del FMM. . . . 47
2.5. Ejemplo del particionado diseñado para el FMM. . . . 48
2.6. Representación esquemática de las comunicaciones de tipo *scatter*. . . . 54
2.7. Representación esquemática de las comunicaciones de tipo *gather*. . . . 59
2.8. Representación de las comunicaciones de tipo envío-recepción. 65
2.9. Presión total sobre la superficie de una aeronave (motores debajo del ala). . . . 75
2.10. Presión total sobre la superficie de una aeronave (motores sobre el ala). . . . 77
2.11. Presión total sobre la superficie de una esfera $\oslash 4\,\mathrm{m}$. . . . 78

3.1. Orden de ejecución de las etapas del FMM-FFT. . . . 89

3.2. Ejemplo del particionado diseñado para el FMM-FFT en sistemas heterogéneos. 91

4.1. Paralelización del cálculo $p^s = K \cdot p$ usando CUDA. 127
4.2. Obtención de P^S usando múltiples productos matriz-vector. 128
4.3. Obtención de P^S usando un producto matricial por bloques (*SmallBlocks*). 133
4.4. Obtención de P^S usando un producto matricial por bloques (*BigBlocks*). 137
4.5. Obtención de P^S con una solución heterogénea. 143
4.6. Presión total en un plano situado debajo de la aeronave. . . 147
4.7. Vista comparativa de la presión total. 148

5.1. Estrategia para paralelizar el producto matriz-vector en el MST. 157
5.2. Corrientes equivalentes reconstruidas para una antena de una estación base. 165
5.3. Corrientes equivalentes reconstruidas para una antena de tipo hélice. 167
5.4. Distribución de las ondas planas incidentes. 168
5.5. Corrientes equivalentes reconstruidas y campo interno calculado en varios cortes. 169
5.6. Corrientes equivalentes reconstruidas, campo interno calculado en varios cortes e isosuperficies $-3\,\text{dB}$. 170

6.1. *Cluster* CMI. Aceleración vs. número de núcleos usando FMM. 179
6.2. Tesla MD SimCluster. Eficiencia para dos y cuatro nodos usando FMM. 182
6.3. *Cluster* Mopar. Eficiencia para dos nodos usando FMM. . . 185
6.4. Estimación del consumo de potencia del *cluster* CMI. . . . 186
6.5. *Cluster* CMI. Estimación del consumo de energía usando FMM. 187
6.6. *Cluster* Mopar. Consumo de energía usando FMM. 189
6.7. *Cluster* CMI. Aceleración vs. número de núcleos usando FMM-FFT. 196
6.8. *Cluster* Mopar. Eficiencia para dos nodos usando FMM-FFT. 199

6.9. *Cluster* CMI. Estimación del consumo de energía usando FMM-FFT. 200
6.10. *Cluster* Mopar. Consumo de energía usando FMM-FFT. . . 202
6.11. *Cluster* CMI. Comparativa de la aceleración usando FMM y FMM-FFT. 208
6.12. *Cluster* Mopar. Comparativa de la eficiencia para dos nodos usando FMM y FMM-FFT. 209
6.13. *Cluster* CMI. Comparativa de la estimación del consumo de energía usando FMM y FMM-FFT. 210
6.14. *Cluster* Mopar. Comparativa del consumo de energía usando FMM y FMM-FFT. 210
6.15. Sistema Manycores. Aceleración de las diferentes técnicas respecto de la implementación de referencia. 218
6.16. Comparativa de la eficiencia para dos GPU en un problema de *imaging*. 224

A.1. *Cluster* CMI de la Universidad de Oviedo. 232
A.2. Tesla MD SimCluster de Microway. 234
A.3. Componentes de la estación de trabajo Charger. 238
A.4. Componentes del sistema Manycores del IRPCG. 242
A.5. Componentes de la estación de trabajo Barracuda. 246

Índice de tablas

1.1. Relación entre software y hardware en CUDA. 21

2.1. Error del FMM para CPU + GPU en comparación con el FMM para CPU. 79

3.1. Error del FMM-FFT en comparación con el FMM para CPU. 112
3.2. Error del FMM-FFT en comparación con el FMM-FFT para CPU + GPU. 112
3.3. Error del FMM-FFT para CPU + GPU en comparación con el FMM-FFT para CPU. 113

4.1. Diferencias entre las soluciones proporcionadas por las técnicas implementadas. 149

5.1. Parámetros para la obtención de las corrientes equivalentes en una antena de una estación base usando el SRM-MST. . 164
5.2. Parámetros para la obtención de las corrientes equivalentes en una antena de tipo hélice usando el SRM-MST. 166

6.1. *Cluster* CMI. Airbus serie A3xx ($N = 1009392$, $f = 1\,\mathrm{kHz}$) usando FMM. 176
6.2. *Cluster* CMI. Esfera $\oslash 4\,\mathrm{m}$ ($N = 6001966$, $f = 11{,}5\,\mathrm{kHz}$) usando FMM. 178
6.3. Tesla MD SimCluster. Airbus serie A3xx ($N = 2132704$, $f = 1\,\mathrm{kHz}$) usando FMM. 180
6.4. Tesla MD SimCluster. Esfera $\oslash 4\,\mathrm{m}$ ($N = 6001966$, $f = 11{,}5\,\mathrm{kHz}$) usando FMM. 181

6.5. *Cluster* Mopar. Airbus serie A3xx ($N = 1009392$, $f = 1\,\text{kHz}$) usando FMM. 184
6.6. *Cluster* Mopar. Esfera $\oslash 4\,\text{m}$ ($N = 6001966$, $f = 11{,}5\,\text{kHz}$) usando FMM. 185
6.7. *Cluster* Mopar. Consumo de energía para Airbus serie A3xx ($N = 1009392$, $f = 1\,\text{kHz}$) usando FMM. 188
6.8. *Cluster* Mopar. Consumo de energía para esfera $\oslash 4\,\text{m}$ ($N = 6001966$, $f = 11{,}5\,\text{kHz}$) usando FMM. 188
6.9. Resolución de un problema acústico con cincuenta millones de incógnitas usando FMM en el *cluster* CMI. 190
6.10. Resolución de un problema acústico con cincuenta millones de incógnitas usando FMM en el *cluster* Mopar. 191
6.11. Resolución de un problema acústico con cien millones de incógnitas usando FMM en el *cluster* CMI. 192
6.12. Resolución de un problema acústico con cien millones de incógnitas usando FMM en el *cluster* Mopar. 193
6.13. *Cluster* CMI. Airbus serie A3xx ($N = 1009392$, $f = 1\,\text{kHz}$) usando FMM-FFT. 194
6.14. *Cluster* CMI. Esfera $\oslash 4\,\text{m}$ ($N = 6001966$, $f = 11{,}5\,\text{kHz}$) usando FMM-FFT. 195
6.15. *Cluster* Mopar. Airbus serie A3xx ($N = 1009392$, $f = 1\,\text{kHz}$) usando FMM-FFT. 197
6.16. *Cluster* Mopar. Esfera $\oslash 4\,\text{m}$ ($N = 6001966$, $f = 11{,}5\,\text{kHz}$) usando FMM-FFT. 198
6.17. *Cluster* Mopar. Consumo de energía para Airbus serie A3xx ($N = 1009392$, $f = 1\,\text{kHz}$) usando FMM-FFT. 201
6.18. *Cluster* Mopar. Consumo de energía para esfera $\oslash 4\,\text{m}$ ($N = 6001966$, $f = 11{,}5\,\text{kHz}$) usando FMM-FFT. 202
6.19. Resolución de un problema acústico con cincuenta millones de incógnitas usando FMM-FFT en el *cluster* CMI. 203
6.20. Resolución de un problema acústico con cincuenta millones de incógnitas usando FMM-FFT en el *cluster* Mopar. . . . 204
6.21. *Cluster* CMI. Comparativa del tiempo de ejecución con esfera $\oslash 4\,\text{m}$ ($N = 6001966$, $f = 11{,}5\,\text{kHz}$) usando FMM y FMM-FFT. 206

6.22. *Cluster* Mopar. Comparativa del tiempo de ejecución con esfera ⊘4 m ($N = 6001966$, $f = 11{,}5\,\text{kHz}$) usando FMM y FMM-FFT. 206
6.23. Comparativa de la resolución de un problema acústico con cincuenta millones de incógnitas en el *cluster* CMI. 212
6.24. Comparativa de la resolución de un problema acústico con cincuenta millones de incógnitas en el *cluster* Mopar. . . . 213
6.25. Cálculo de la presión total usando diferentes técnicas para CPU. 214
6.26. Cálculo de la presión total usando diferentes técnicas para coprocesadores Xeon Phi. 215
6.27. Cálculo de la presión total usando diferentes técnicas para GPU. 215
6.28. Cálculo de la presión total usando diferentes técnicas para sistemas heterogéneos basados en GPU + Xeon Phi. 216
6.29. Tiempo de ejecución y consumo de memoria en el sistema SimLin02 para el caso de una antena de una estación base. 220
6.30. Tiempo de ejecución y consumo de memoria en el sistema SimLin02 para el caso de una antena de tipo hélice. 221
6.31. Tiempo de ejecución para la obtención del campo interno en un problema de *imaging* usando el sistema Barracuda. . 222
6.32. Tiempo de ejecución para la obtención del campo interno en un problema de *imaging* usando el sistema Roadrunner. 223

A.1. Hardware del *cluster* CMI de la Universidad de Oviedo. . . 233
A.2. Software del *cluster* CMI de la Universidad de Oviedo. . . 233
A.3. Hardware del *cluster* Tesla MD SimCluster de Microway. . 235
A.4. Software del *cluster* Tesla MD SimCluster de Microway. . . 236
A.5. Hardware del *cluster* Mopar del grupo TSC-UNIOVI. . . . 237
A.6. Software del *cluster* Mopar del grupo TSC-UNIOVI. 239
A.7. Hardware del sistema Manycores del IRPCG. 240
A.8. Software del sistema Manycores del IRPCG. 241
A.9. Parámetros para los *kernels* CUDA correspondientes a diferentes técnicas desarrolladas. 241
A.10. Tamaño de bloque en los algoritmos basados en multiplicación matricial para CPU y Xeon Phi. 241

A.11. Hardware del sistema SimLin02 del grupo TSC-UNIOVI. . 243
A.12. Software del sistema SimLin02 del grupo TSC-UNIOVI. . . 244
A.13. Hardware del sistema Barracuda del grupo TSC-UNIOVI. . 245
A.14. Software del sistema Barracuda del grupo TSC-UNIOVI. . 245
A.15. Hardware del sistema Roadrunner del grupo TSC-UNIOVI. 247
A.16. Software del sistema Roadrunner del grupo TSC-UNIOVI. 247

Índice de algoritmos

2.1. Cálculo del operador de traslación usando MPI y OpenMP. 55
2.2. Interacciones cercanas usando MPI y CUDA. 57
2.3. Agregación usando MPI y CUDA. 61
2.4. Traslación usando MPI y OpenMP. 68
2.5. Desagregación usando MPI y CUDA. 71

3.1. Interacciones cercanas usando MPI y OpenMP. 98
3.2. Agregación usando MPI y OpenMP. 101
3.3. Traslación usando MPI y OpenMP. 104
3.4. Desagregación usando MPI y OpenMP. 109

4.1. Solución bMVP para CPU multinúcleo y coprocesadores tipo Xeon Phi usando OpenMP. 124
4.2. Solución bMVP para GPU usando CUDA. 125
4.3. Solución mMVP para CPU multinúcleo y coprocesadores tipo Xeon Phi usando OpenMP. 129
4.4. Solución mMVP para GPU usando CUDA. 131
4.5. Solución *SmallBlocks* para CPU multinúcleo y coprocesadores tipo Xeon Phi usando OpenMP y MKL. 134
4.6. Solución *BigBlocks* para CPU multinúcleo y coprocesadores tipo Xeon Phi usando OpenMP y MKL. 138
4.7. Solución *BigBlocks* para GPU usando CUDA (código ejecutado en el *host*). 140
4.8. Solución *BigBlocks* para GPU usando CUDA. Generación de los bloques (código ejecutado en GPU). 142
4.9. Solución *BigBlocks* para GPU usando CUDA. Almacenamiento de resultados parciales (código ejecutado en GPU). . 142

4.10. Solución heterogénea para GPU y Xeon Phi. 144

5.1. Solución heterogénea para múltiples GPU. 162

Capítulo 1

Introducción

Índice

1.1. Motivación **1**
1.2. Antecedentes y estado de la cuestión **4**
1.2.1. Problema directo de dispersión acústica 4
1.2.2. Problema electromagnético inverso 9
1.2.3. Paralelización y aceleración hardware 12
1.3. Arquitecturas y tecnologías utilizadas en la Tesis 14
1.3.1. Procesadores multinúcleo y sistemas de memoria compartida 15
1.3.2. Sistemas de memoria distribuida 17
1.3.3. Procesadores gráficos para computación de propósito general 19
1.3.4. Coprocesadores Xeon Phi 22
1.4. Contribuciones de la Tesis **24**
1.5. Estructura de la memoria de la Tesis **26**

1.1. Motivación

En los últimos años, la reducción del ruido está cobrando una considerable atención por parte de las autoridades competentes, debido al rechazo

que genera la contaminación acústica en la población. Como consecuencia de lo anterior, el control de ruido en las aeronaves (especialmente del ruido externo durante despegues y aterrizajes) se ha convertido en un tema de especial interés para los principales organismos y empresas del sector [1, 2, 3], ya que los requisitos relativos al ruido ambiental son cada vez más restrictivos.

No obstante, para poder explorar el amplio abanico de posibilidades que se presenta en la fase de diseño de una nueva aeronave, resulta indispensable aplicar técnicas de predicción acústica que permitan caracterizar el comportamiento de los distintos modelos desde el punto de vista de la presión sonora dispersada. De esta forma, es posible descubrir a priori qué cambios deben llevarse a cabo en un prototipo para alcanzar los requisitos de funcionamiento definidos, reduciendo el tiempo y los costes asociados a los procesos de fabricación. Por lo tanto, las herramientas de simulación para dispersión acústica en problemas de control de ruido tienen una aplicabilidad directa, ya que, siempre que permitan obtener resultados de forma rápida y precisa, posibilitarán una reducción de las complejas medidas sobre prototipos como las llevadas a cabo por las diferentes compañías y agencias del sector aeronáutico [4, 5].

Sin embargo, el análisis preciso de los problemas mencionados en el párrafo anterior puede requerir una gran cantidad de recursos computacionales (especialmente tiempo de cálculo y espacio en memoria), sobre todo para modelos realistas y rangos de frecuencias amplios. Por tanto, con el objetivo de incrementar el rango de aplicación de las técnicas de modelado y predicción, se debe recurrir al diseño de herramientas en las que se combine el uso de métodos de aceleración computacional (esquemas aceleradores) junto con técnicas propias de la computación de altas prestaciones.

Por su parte, el problema inverso de radiación y dispersión electromagnética es de gran interés en múltiples aplicaciones relacionadas con la evaluación no destructiva, entre las cuales se incluyen la caracterización de antenas y la imagen electromagnética.

Para caracterizar una antena y, en caso necesario, diagnosticar posibles problemas en su funcionamiento, resulta indispensable disponer del diagrama de radiación de la misma. Sin embargo, no siempre es posible medir

el diagrama de radiación de una antena en una cámara anecoica, debido a limitaciones de espacio (especialmente en antenas de tamaño considerable o para grandes longitudes de onda). Por ello, los métodos que permiten calcular el diagrama de radiación de la antena bajo medida a partir de transformaciones de campo cercano (típicamente medido en cámara anecoica) a campo lejano son especialmente interesantes [6, 7].

En lo referente a la imagen electromagnética o *imaging*, su aplicación en diversos ámbitos, en especial en el campo de la medicina y la seguridad, le confiere una relevancia notable. En concreto, los algoritmos para la reconstrucción del contorno de objetos metálicos a partir de medidas del campo dispersado en un dominio de observación que encierra al objeto bajo medida [8, 9] pueden aplicarse cuando se desea conocer la geometría de dichos objetos en situaciones en las que no son visibles por encontrarse ocultos.

No obstante, en ambos casos (caracterización de antenas e imagen electromagnética), el coste computacional asociado a la resolución de dichos problemas puede suponer un cuello de botella, especialmente con problemas eléctricamente grandes, ya que el tiempo de cálculo necesario para obtener una solución puede no satisfacer los requisitos de tiempo real o cuasi-real de algunas aplicaciones como, por ejemplo, las relacionadas con la seguridad (detección de armas ocultas, inspección de paquetes, etc.) [10, 11, 12].

Esta Tesis tiene como objetivo principal la generación de un conjunto de herramientas que permita la resolución eficiente y precisa de diferentes problemas dentro del ámbito de la dispersión electromagnética y acústica. Además, el trabajo desarrollado en los temas relacionados con el problema electromagnético inverso se enmarca dentro del proyecto "Técnicas de imaging mediante problema inverso de dispersión: nuevos algoritmos y técnicas de medida – iScat" (Ref.: TEC2011-24492), desarrollado por el grupo de investigación al que pertenece el doctorando (TSC-UNIOVI).

En los temas relacionados con la dispersión acústica, se toma como punto de partida la Tesis Doctoral del Dr. Jesús López Fernández [13], director de esta Tesis. Continuando con las líneas propuestas en su Tesis, se han marcado varios objetivos concretos: posibilitar el análisis de objetos de gran tamaño acústico, reducir los tiempos de resolución de los problemas

analizados (modificando la algoritmia y recurriendo al uso de aceleradores hardware) y, finalmente, mejorar la eficiencia energética de las herramientas desarrolladas. Todo ello enfocado a permitir la resolución de un amplio abanico de problemas sin necesidad de recurrir al uso de grandes sistemas computacionales.

Por su parte, para el desarrollo de las herramientas basadas en el problema electromagnético inverso, se parte del trabajo presentado por el Dr. Yuri Álvarez López en su Tesis [14] y del trabajo llevado a cabo posteriormente por el Dr. Cebrián García González en su Tesis [15]. En este caso, el objetivo perseguido consiste en posibilitar el análisis en tiempo real o cuasi-real para problemas de tamaño moderado, modificando la algoritmia inicial para su paralelización y recurriendo, además, al uso de aceleradores hardware.

1.2. Antecedentes y estado de la cuestión

En los siguientes apartados, se hace una breve revisión de los antecedentes y el estado actual tanto del problema directo de dispersión acústica como del problema electromagnético inverso. Asimismo, se comentan diversas técnicas de paralelización y aceleración hardware con aplicación en los diferentes problemas abordados en esta Tesis.

1.2.1. Problema directo de dispersión acústica

Los diferentes métodos numéricos para el análisis de problemas de dispersión acústica (o electromagnética) pueden clasificarse en dos grandes categorías:

- Métodos de alta frecuencia. Se pueden aplicar cuando la longitud de onda del campo incidente es notablemente menor que las dimensiones del objeto bajo análisis. Se basan, principalmente, en métodos estadísticos y aproximaciones asintóticas.

- Métodos de baja frecuencia. Se pueden utilizar con geometrías arbitrarias y sin limitaciones en la frecuencia de análisis proporcionando resultados rigurosos. Su mayor inconveniente viene dado por su alto coste computacional.

En el trabajo llevado a cabo en esta Tesis en el ámbito de la dispersión acústica, se emplea como método de análisis el método de los elementos de contorno (BEM) [16]. Su equivalente en el problema electromagnético, el método de los momentos (MoM) [17, 18, 19], se basa en los mismos principios que el BEM [13] y su uso también está ampliamente extendido.

Tanto el BEM como el MoM se encuentran dentro de los métodos denominados de baja frecuencia, los cuales permiten obtener resultados precisos con independencia del tamaño y de la forma del objeto dispersor, así como de la frecuencia del campo incidente con el que se ilumina dicho objeto. En el caso del BEM, se resuelve la ecuación de Helmholtz (problema acústico), mientras que en el MoM se resuelven las ecuaciones de Maxwell (problema electromagnético). En ambos casos, se recurre a las ecuaciones anteriormente mencionadas en su forma integral en combinación con las condiciones de contorno adecuadas. Los dos se catalogan, por tanto, como métodos integrales de baja frecuencia.

Ambos métodos permiten discretizar las ecuaciones de las que se parte, produciendo un sistema de ecuaciones lineales cuya solución puede obtenerse recurriendo a diferentes métodos o técnicas de resolución numérica. Puesto que el sistema de ecuaciones que se debe resolver presenta un tamaño proporcional al tamaño acústico (o eléctrico) del problema, su principal inconveniente radica en su elevado coste computacional. Si se nota como N el número de funciones base (incógnitas) en las que se discretiza el problema, la resolución directa de dicho problema tiene una complejidad $\mathcal{O}(N^3)$ en tiempo y $\mathcal{O}(N^2)$ en memoria, por lo que el análisis de problemas de gran tamaño (en longitudes de onda) puede resultar inviable.

1.2.1.1. Métodos de resolución y esquemas aceleradores

Los métodos de resolución que se pueden emplear en la resolución del sistema de ecuaciones que plantea el BEM/MoM pueden dividirse, fundamentalmente, en dos grandes categorías: los métodos de resolución directa y los métodos iterativos.

Los métodos de resolución directa se basan en la factorización o la inversión de la matriz del sistema (eliminación de Gauss, factorizaciones LU/LDLT, etc.), por lo que la obtención de la solución tiene un coste $\mathcal{O}(N^3)$ en tiempo y $\mathcal{O}(N^2)$ en memoria [20], limitando de forma notable su aplicación práctica [21].

Los métodos iterativos permiten reducir la complejidad temporal a $\mathcal{O}(N^2)$ por iteración. Si, además, se tiene en cuenta que el número de iteraciones necesarias para alcanzar una solución con la precisión exigida suele ser muy inferior a N, las reducciones en el tiempo de ejecución pueden ser considerables. Los métodos iterativos pueden dividirse [22], a su vez, en métodos de tipo estacionario y en métodos no estacionarios.

Los métodos estacionarios son cronológicamente anteriores a los no estacionarios y se caracterizan por generar el iterando en cada paso siguiendo la expresión $x^{(it)} = Bx^{(it-1)} + c$, en la que se suma un vector al resultado de multiplicar el iterando del paso previo por una matriz. Cabe mencionar que tanto el vector (c) como la matriz (B) usados en el cálculo del iterando no dependen de la iteración (it). Este tipo de métodos son más sencillos que los no estacionarios, pero no son tan eficaces, ya que su convergencia suele ser lenta y no siempre está garantizada [22]. Dentro de este tipo de métodos iterativos, se encuentran los métodos de Jacobi, Gauss-Seidel, y las diferentes variantes de Gauss-Seidel destinadas a acelerar la velocidad de convergencia [22]. Asimismo, dentro de esta categoría, merece la pena destacar el método *Forward-Backward* (FBM) [23], que ha mostrado una buena velocidad de convergencia para problemas bidimensionales, por ejemplo aplicado a superficies marinas para la simulación del comportamiento de las olas rompientes [24] o para realizar análisis de cobertura en superficies montañosas [25].

Por su parte, los métodos no estacionarios tienen un origen más re-

ciente, son más eficaces, y se caracterizan porque los datos utilizados en sus cálculos se modifican en cada iteración. Dentro de esta categoría, se encuentran los métodos de tipo Krylov [26], que se basan en la generación de una secuencia de vectores ortogonales, y el método de Chebyshev [22], que se basa en una secuencia de polinomios ortogonales. Los métodos de tipo Krylov son más comunes y, de entre ellos, merece la pena destacar el gradiente conjugado (CG) y el gradiente conjugado para la minimización de la norma del residuo (CGNR) [27], el gradiente bi-conjugado estabilizado (Bi-CGSTAB) [28], y el método del residuo mínimo generalizado (GMRES) [29]. En el caso del problema de dispersión acústica, el GMRES destaca especialmente, debido a su robustez y velocidad de convergencia en la resolución iterativa de este tipo de problemas [30].

Si bien los métodos iterativos mencionados en los párrafos anteriores logran una reducción importante en el coste temporal respecto de la resolución directa, dichos métodos deben llevar a cabo, al menos, un producto matriz-vector, con un coste $\mathcal{O}(N^2)$, por iteración. Por tanto, su uso también puede llegar a ser poco práctico en problemas realistas, donde el valor de N puede sobrepasar ampliamente el millón de incógnitas. Para superar esa limitación, puede recurrirse al uso de esquemas aceleradores aplicados a diferentes métodos iterativos.

Los esquemas aceleradores pueden dividirse en dos categorías principales: técnicas basadas en la compresión de matrices y técnicas basadas en la evaluación rápida del producto matriz-vector.

De entre las técnicas basadas en la compresión de matrices, merece la pena destacar la técnica basada en las matrices $\mathcal{H}$ [31, 32] y la posterior técnica *Adaptive Cross Approximation* (ACA) [33, 34], con aplicación en problemas electromagnéticos [35] y acústicos [21].

Por su parte, las técnicas basadas en la evaluación rápida del producto matriz-vector persiguen reducir el coste asociado a los productos matriz-vector presentes en la resolución iterativa del sistema de ecuaciones planteado por el BEM/MoM. De entre este tipo de esquemas aceleradores, cabe resaltar el algoritmo de descomposición matricial multinivel (MLMDA) [36, 37] y el método multipolar rápido o *Fast Multipole Method* (FMM) [38, 39, 40].

El FMM fue desarrollado inicialmente para evaluar de forma eficiente el potencial y el campo en sistemas con un gran número de partículas dentro del ámbito de la física [38]. Posteriormente, se ha utilizado para resolver diversos tipos de problemas, entre ellos los basados en BEM y MoM. En dichos casos, la solución a la ecuación de Helmholtz (acústica) o a las ecuaciones de Maxwell (electromagnetismo) se lleva a cabo realizando una expansión multipolar de la función de Green, con el objetivo de agrupar las contribuciones de aquellas fuentes que se encuentran suficientemente próximas entre sí para evaluarlas, sobre puntos de observación suficientemente lejanos, en un único paso. El FMM, considerado por algunos autores como uno de los diez mejores algoritmos del siglo XX [41, 42], permite reducir la complejidad temporal a $\mathcal{O}(N^{3/2})$ en su versión mononivel [40], $\mathcal{O}(N^{4/3}\log^{2/3} N)$ en la implementación que hace uso de la transformada rápida de Fourier (FMM-FFT) [43, 44, 45], y $\mathcal{O}(N \log N)$ en su versión multinivel o *Multilevel Fast Multipole Algorithm* (MLFMA/MLFMM) [46]. La aplicación del FMM para la resolución de problemas tridimensionales de dispersión acústica data de la primera década del siglo XXI [47] (mononivel) y [48] (multinivel), y continúa vigente en los últimos años [21, 49, 50].

La incorporación de esquemas aceleradores supone una mejora significativa en la complejidad computacional respecto de los métodos iterativos no acelerados, pero la resolución de problemas de dispersión de gran tamaño acústico (o eléctrico) sigue siendo costosa. De igual forma, el uso de esquemas aceleradores puede ser insuficiente en la resolución de problemas de tamaño moderado pero con requisitos de tiempo cuasi-real o real. Por ello, desde hace varios años, existe un interés manifiesto en el uso de técnicas de computación paralela y aceleración hardware, con el objetivo de reducir aún más los tiempos de ejecución necesarios para obtener una solución en los problemas mencionados. Dado el carácter transversal de la mayoría de técnicas relacionadas con la computación paralela y la aceleración hardware, en el apartado 1.2.3, se revisarán de forma conjunta diferentes estrategias de paralelización y aceleración hardware con aplicación en los dos problemas abordados en esta Tesis: el problema acústico y el problema electromagnético.

1.2.2. Problema electromagnético inverso

En la caracterización de antenas, los métodos para la transformación de campo cercano a campo lejano [51, 52, 53, 54] resultan especialmente útiles para obtener el diagrama de radiación de una antena a partir de medidas en campo cercano [55]. De esta forma, es posible eludir el problema de la limitación de espacio en las cámaras anecoicas, el cual puede hacer inviable la medida del diagrama de radiación para antenas de gran tamaño.

Las técnicas de transformación de campo cercano a campo lejano pueden dividirse en dos categorías: métodos basados en expansión en modos de onda y técnicas basadas en la obtención de una distribución de densidades de corriente (eléctrica y magnética) equivalentes.

En los métodos basados en la expansión en modos de onda, el campo radiado por la antena bajo medida se expande en modos de onda (planos, cilíndricos o esféricos) cuyos coeficientes se determinan a partir de las medidas realizadas en campo cercano [51, 52]. Estos métodos están limitados a superficies de adquisición canónicas, pero son muy eficientes en términos computacionales debido a que, las transformaciones de campo cercano a campo lejano que utilizan, se basan en la FFT [7].

Por su parte, el método de reconstrucción de fuentes (SRM) [53, 54] es una técnica que se basa en la obtención, a partir del campo medido, de la distribución de densidad de corriente equivalente (eléctrica y magnética) que radia el mismo campo que la antena bajo medida en cualquier punto situado en el exterior del domino fuente. El SRM no impone ninguna restricción en la geometría de los dominios de adquisición y de reconstrucción [56, 57], por lo que su ámbito de aplicación es más amplio. Sin embargo, el coste computacional asociado al cálculo de las densidades de corriente equivalentes puede ser elevado, especialmente para antenas de gran tamaño eléctrico [7].

En el ámbito de la reconstrucción de perfiles (estimación de la geometría) utilizando la información obtenida a partir del campo dispersado, los métodos existentes se basan en múltiples técnicas y algoritmos. Entre otros, cabe destacar:

- Los métodos de descomposición, en los que la solución del problema inverso se divide en dos partes: mejorar el mal condicionamiento y solventar la no linealidad del sistema [8].

- Los métodos basados en fuentes equivalentes [58, 59, 60, 61], en los que el uso de dichas fuentes equivalentes permite linealizar la ecuación integral utilizada para resolver el problema inverso. Dentro de esta categoría, se encuentra el SRM.

- Los algoritmos que utilizan técnicas de radar, los cuales se basan en la medida de la sección radar del objeto bajo análisis. En este caso, se usan técnicas inversas de apertura sintética radar para obtener imágenes de la distribución espacial de la reflectividad del objeto bajo estudio [62, 63].

- Los algoritmos basados en *Contrast Source Inversion* (CSI) [64, 65], en los que el problema inverso se resuelve mediante la minimización de una función de coste que viene dada por una combinación lineal de los errores en las denominadas ecuaciones de datos y objetivo. La ecuación de datos modela el campo eléctrico dispersado por el objeto bajo estudio en el dominio de observación, mientras que la ecuación objetivo modela el campo eléctrico total en el dominio en el que se calculan las fuentes de contraste.

- Los algoritmos evolutivos, que son métodos de optimización y búsqueda de soluciones que se inspiran en la evolución biológica presente en la naturaleza. Partiendo de un conjunto inicial de posibles soluciones al problema (población inicial), dichas soluciones pueden sufrir modificaciones (mutaciones) y también tienen la posibilidad de mezclarse entre sí (cruces), todo ello con el objetivo de evolucionar hacia una mejor solución al problema (las mejores soluciones de cada generación logran prevalecer a lo largo del tiempo). Dentro de esta categoría, destacan los algoritmos genéticos [66, 67, 68] y la optimización por enjambre de partículas [69].

En el caso del SRM, el contorno del objeto bajo estudio se puede estimar a partir de las corrientes equivalentes reconstruidas, teniendo en cuenta la

relación entre la ubicación de las partes metálicas del objeto analizado y la posición de los niveles máximos de la densidad de corriente eléctrica [9] o los valores del campo interno con mayor amplitud [70]. La aplicación del SRM para reconstrucción de perfiles es sencilla, pero suele requerir mayor cantidad de información relativa al campo dispersado (mayor número de ondas incidentes y/o frecuencias) que los métodos basados en algoritmos evolutivos [66, 67, 68, 69].

En esta Tesis, para el problema electromagnético inverso aplicado a la caracterización de antenas y a la reconstrucción de perfiles, se recurre al uso del SRM y se emplea el MoM [17, 18] para discretizar la ecuación integral del campo eléctrico.

1.2.2.1. Métodos de resolución y esquemas aceleradores

Si bien se pueden utilizar métodos de resolución directa como la descomposición en valores singulares (SVD) [71, 72], los métodos iterativos son los más comúnmente utilizados para resolver el sistema de ecuaciones planteado por el MoM en el caso del SRM.

De entre los métodos iterativos aplicados al problema inverso, destaca especialmente el CGNR [27] y la implementación del mismo presentada en [73], diseñada teniendo en cuenta su posible paralelización en sistemas de memoria compartida. Asimismo, algunos autores también recurren al uso del algoritmo iterativo GMRES para problemas basados en la obtención de densidades de corriente equivalentes [74].

Puesto que los métodos iterativos pueden tener un coste computacional demasiado alto (realizan al menos un producto matriz-vector por iteración), la incorporación de esquemas aceleradores, aplicados a los diferentes métodos iterativos o a soluciones basadas en métodos de resolución directa, puede ser de especial utilidad cuando se desea analizar objetos eléctricamente grandes o si se pretende obtener soluciones con un tiempo de respuesta muy reducido (posibilitando análisis en tiempo real o cuasi-real).

En el caso de los esquemas aceleradores basados en la compresión de matrices, merece la pena destacar la técnica *Adaptive Cross Approxima-*

tion (ACA), que puede ser aplicada a problemas electromagnéticos [35], entre ellos los basados en el SRM [75]. Asimismo, en [76], se presenta una implementación del ACA multinivel (MLACA) aplicada a problemas de radiación y dispersión electromagnética.

De igual forma, en el problema electromagnético inverso también se pueden aplicar técnicas basadas en la evaluación rápida de los productos matriz-vector presentes en la resolución iterativa del sistema de ecuaciones planteado por el MoM. Por ejemplo, en [77, 78] se muestra una implementación del FMM mononivel para el problema inverso que permite reducir los tiempos de ejecución en dos órdenes de magnitud respecto de la implementación no acelerada expuesta por los mismos autores en [9]. Por su parte, en [74] se presenta una implementación del FMM, en su versión multinivel, aplicado también al método de reconstrucción de fuentes.

Para finalizar, también merece la pena destacar la estrategia *Memory saving technique* (MST) aplicada al SRM [6], que permite reducir de forma notable el consumo de memoria a costa de aumentar el tiempo de ejecución, ya que la matriz de impedancias se recalcula en cada iteración, pero sin llegar a incrementar la complejidad temporal del algoritmo, que sigue siendo $\mathcal{O}(N^2)$ por iteración.

A continuación, se lleva a cabo una revisión de los logros obtenidos en los últimos años en la paralelización y aceleración de diferentes problemas relacionados con la dispersión electromagnética y acústica.

1.2.3. Paralelización y aceleración hardware

Dentro de la computación de altas prestaciones, la computación paralela [79] y el uso de aceleradores hardware (computación heterogénea) son los paradigmas más extendidos cuando se desea resolver problemas de gran tamaño [80].

En el caso del algoritmo FMM y sus diferentes variantes, las implementaciones paralelas para procesadores convencionales son las más extendidas. Para el problema electromagnético, destacan sobre el resto las implementaciones presentadas en [81], donde los autores utilizan el FMM-FFT binivel

para resolver un problema electromagnético con 500 millones de incógnitas, y en [82], donde los mismos autores resuelven un problema electromagnético con 1000 millones de incógnitas usando el MLFMA-FFT. Asimismo, merece la pena hacer referencia a la implementación del MLFMA presentada en [83], donde sus autores utilizan una implementación paralela que se apoya en técnicas *out-of-core* y unidades de disco de estado sólido para resolver problemas con más de 1000 millones de incógnitas. Para el problema acústico, cabe destacar los resultados mostrados en [21], donde los autores resuelven un problema con 14 millones de incógnitas (Airbus A321 a 5 kHz) usando una implementación del FMM multinivel en un total de 88 núcleos CPU.

Sin embargo, en los últimos años también se han presentado diferentes implementaciones del FMM (en sus diferentes variantes) capaces de aprovechar la potencia computacional de los procesadores gráficos. En [84], se presenta una implementación para GPU del MLFMA aplicado a la resolución de la ecuación de Helmholtz. Por su parte, en [85], los autores presentan una implementación del FMM mononivel para múltiples GPU aplicada al problema electromagnético, mientras que en [86] analizan su rendimiento tanto para procesadores gráficos como en FPGA (*Field Programmable Gate Array*). En [87], los mismos autores muestran una nueva implementación para múltiples GPU, esta vez basada en el algoritmo FMM-FFT, también aplicada al problema electromagnético.

En el caso específico de los problemas de imagen electromagnética, resulta de interés mencionar la implementación para GPU del FMM mononivel inverso presentada en [88]. Por otra parte, en [65] se lleva a cabo una implementación para GPU del *multiplicative-regularized contrast source inversion algorithm* (MR-CSI).

Asimismo, merece la pena destacar algunas implementaciones del FMM para procesadores gráficos aplicadas a otros problemas. En [89] se presenta una implementación del FMM multinivel en la que se usan cientos de GPU para resolver el problema de los dos cuerpos (interacción de partículas). Por su parte, en [90], los autores muestran una implementación del FMM multinivel (denominada ExaFMM) capaz de utilizar miles de GPU para aplicaciones de electrostática biomolecular y simulación de turbulencias.

Finalmente, como ejemplo de uso de los aceleradores de tipo Xeon Phi [91] en problemas similares a los abordados en esta Tesis, resulta interesante la implementación del BEM para la ecuación de Laplace llevada a cabo en [92].

1.3. Arquitecturas y tecnologías utilizadas en la Tesis

En este apartado, se realiza una descripción somera de las diferentes arquitecturas computacionales utilizadas en esta Tesis y de las tecnologías aplicadas para lograr un uso eficiente de las mismas, todo ello con el objetivo de introducir y clarificar la terminología utilizada en los siguientes capítulos.

Siguiendo la taxonomía de Flynn [93], los sistemas informáticos que se han utilizado pueden clasificarse en dos grupos:

1. Sistemas y dispositivos con una arquitectura de tipo MIMD (*Multiple Instruction streams, Multiple Data streams*). En este grupo, se enmarcarían las CPU (*Central Processing Unit*) multinúcleo, los sistemas de memoria compartida y distribuida, y los coprocesadores Xeon Phi [91].

2. Dispositivos con una arquitectura de tipo SIMD (*Single Instruction stream, Multiple Data streams*). Dentro de este grupo, se encontrarían las GPU (*Graphics Processing Unit*).

En la actualidad, los dispositivos mencionados en la lista anterior suelen pertenecer a ambas categorías a la vez, en mayor o menor medida. Por ejemplo, las CPU multinúcleo son dispositivos MIMD, pero cada núcleo puede disponer de unidades vectoriales de tipo SIMD. De igual forma, las GPU son dispositivos SIMD, pero pueden implementar *multithreading*, pasando a ser dispositivos de tipo SIMT (*Single-Instruction, Multiple-Thread*) que también pueden incorporar paralelismo a nivel de instrucción.

Con el objetivo de clarificar todos los términos que aparecen en los párrafos anteriores, en los siguientes apartados se describirá con más detalle los sistemas utilizados en esta Tesis y los modelos de programación escogidos para trabajar con cada uno de ellos.

1.3.1. Procesadores multinúcleo y sistemas de memoria compartida

A mediados de la primera década del siglo XXI, la evolución de las CPU sufrió un cambio radical. Se pasó de mejorar el rendimiento con aumentos de la frecuencia de reloj a optar por una estrategia centrada en la replicación de las unidades funcionales que componen una CPU [94], dando lugar a la generalización de los procesadores multinúcleo. Desde entonces, las CPU disponen cada vez de más núcleos, pero cada núcleo de forma individual no siempre ofrece un rendimiento significativamente mayor que uno de la generación anterior.

En términos generales, un sistema de memoria compartida es aquel en el que las unidades de procesamiento que lo componen trabajan con un único espacio de direcciones, compartiendo la memoria principal (ver figura 1.1). Dentro de este tipo de sistemas, los más sencillos son los procesadores multinúcleo. Asimismo, existen sistemas de memoria compartida más complejos, como los multiprocesadores simétricos (*Symmetric multiprocessor* o SMP), que se componen de varios procesadores (que a su vez pueden ser multinúcleo).

Teniendo en cuenta su planteamiento, si se quiere obtener un buen rendimiento en este tipo de sistemas, se deben utilizar todos los núcleos disponibles, logrando que trabajen de forma conjunta y coordinada. Por tanto, las herramientas que se diseñen para resolver problemas de forma eficiente utilizando estos sistemas deben aprovechar las técnicas de programación paralela disponibles.

De entre los diferentes lenguajes e interfaces de programación de aplicaciones (API, *Application Programming Interface*) adecuados para trabajar con sistemas de memoria compartida, en esta Tesis se ha escogido utilizar

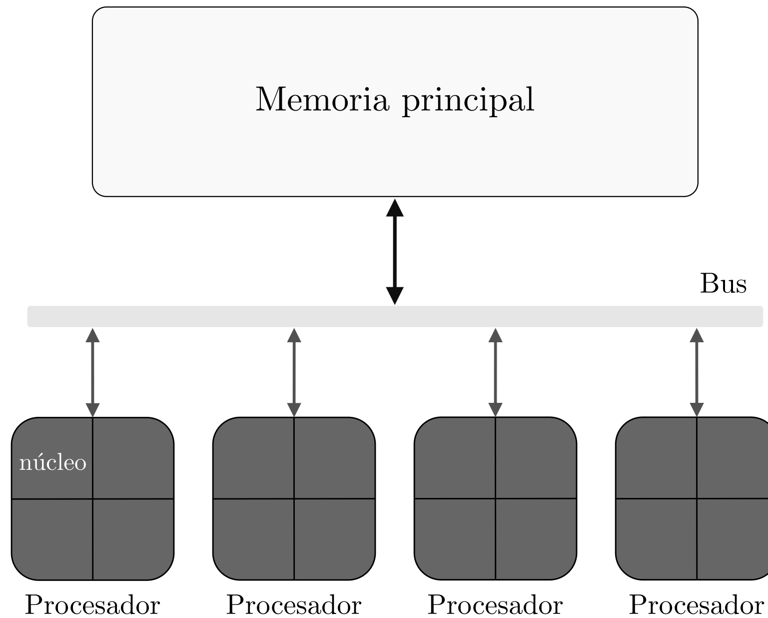


Figura 1.1: Representación esquemática de un sistema de memoria compartida en el que varios procesadores multinúcleo comparten la memoria principal.

el lenguaje C [95] junto con OpenMP [96] y, en algunas ocasiones, MPI (*Message Passing Interface*) [97, 98, 99].

En el caso de OpenMP, sus principales ventajas pueden resumirse en:

- Es un estándar para la programación paralela en sistemas de memoria compartida (la primera versión de esta API data de 1997).
- Permite una programación sencilla gracias al uso de directivas del tipo `#pragma omp`.
- Dispone de mecanismos de control sobre la planificación de los hilos generados (*scheduling*).
- Es independiente de la plataforma y compatible con los principales compiladores, lo que facilita su uso en diferentes entornos.

Además, en algunos casos, puede resultar conveniente combinar el uso de OpenMP y MPI, dando lugar a implementaciones híbridas MPI + OpenMP [100] que presentan ciertas ventajas:

- Se logra un mapeo [79] jerárquico que se asemeja más al hardware subyacente: cada proceso MPI se asigna a una CPU multinúcleo determinada y cada hilo OpenMP a un núcleo concreto.
- Se dispone de dos niveles de paralelismo diferentes (nivel de proceso y nivel de hilo) en los que poder repartir y equilibrar la carga de trabajo de forma más eficaz.
- Se logra una implementación que también es válida para sistemas de memoria distribuida.

1.3.2. Sistemas de memoria distribuida

Los sistemas de memoria distribuida son sistemas que constan de dos o más nodos independientes con capacidad para comunicarse entre sí mediante una red de interconexión. Cada nodo (compuesto por una o varias CPU) dispone de un espacio de memoria propio y puede intercambiar información con otros nodos mediante el paso de mensajes a través de la red que los interconecta.

En la actualidad, dentro del ámbito de la computación de altas prestaciones, es habitual que los nodos que componen un sistema de memoria distribuida sean de tipo SMP. Por tanto, los procesadores dentro de un nodo comparten la memoria principal, mientras que los procesadores de nodos diferentes sólo pueden compartir información haciendo uso de la red de interconexión. En la figura 1.2 se representa de forma esquemática un sistema de memoria distribuida compuesto por nodos de tipo SMP.

Las aplicaciones que se diseñen con el objetivo de aprovechar las capacidades de los sistemas de memoria distribuida deben hacer que los diferentes nodos trabajen de forma colaborativa y coordinada. Para lograrlo, se deben tener en cuenta tres aspectos fundamentales:

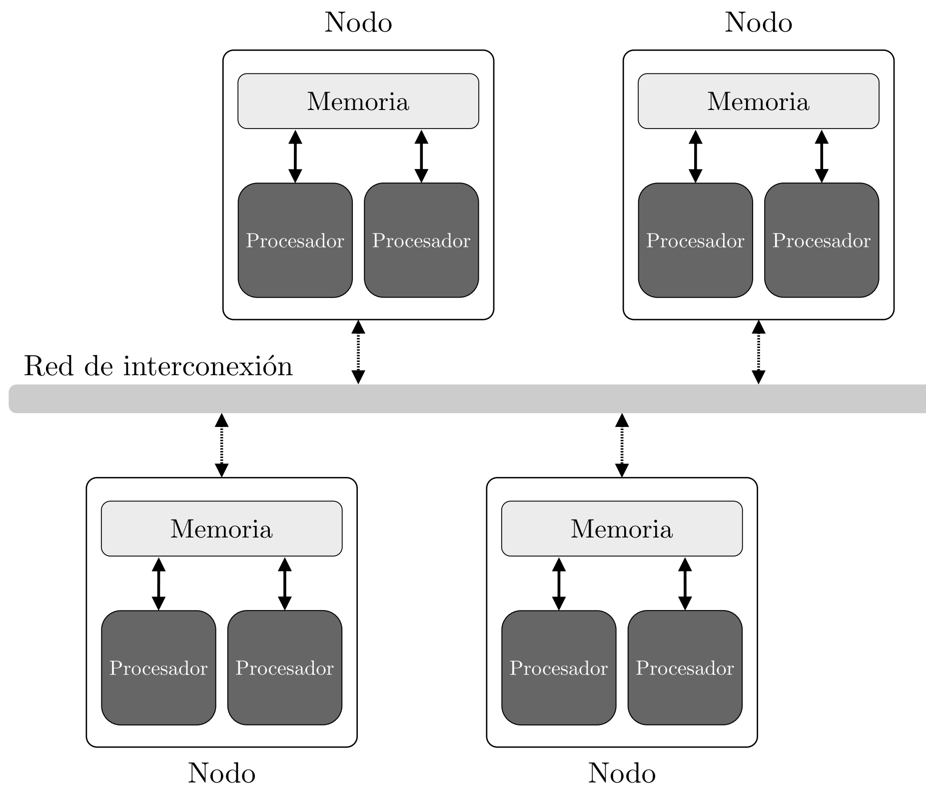


Figura 1.2: Representación esquemática de un sistema de memoria distribuida que consta de varios nodos compuestos por procesadores multinúcleo.

- La división del problema inicial en un conjunto equivalente de subproblemas más pequeños que se puedan resolver en paralelo. Esta fase se conoce como particionado.
- El reparto equitativo de las tareas a realizar por cada nodo, para evitar que haya nodos saturados mientras otros permanecen ociosos. A este objetivo se lo suele denominar balanceo (o equilibrado) de la carga de trabajo.
- El intercambio de información entre los distintos nodos, con el objetivo de minimizar la sobrecarga asociada al intercambio de mensajes.

Este aspecto se corresponde con las comunicaciones.

Asimismo, merece la pena añadir que, con el objetivo de llevar a cabo la paralelización de los diferentes algoritmos presentados en esta Tesis de una forma metodológica, se han usado como referencia las guías de diseño que se pueden encontrar en [79], donde se estudia el particionado, el balanceo de carga y las comunicaciones, entre otros aspectos.

De entre las diferentes API orientadas al desarrollo de aplicaciones para sistemas de memoria distribuida, la más extendida es MPI [97, 98, 99]. Para llevar a cabo la implementación para sistemas de memoria distribuida de las diferentes herramientas presentadas en esta Tesis, se han utilizado rutinas que se enmarcan dentro del estándar MPI-2 [98].

1.3.3. Procesadores gráficos para computación de propósito general

Los procesadores utilizados en la aceleración de gráficos 3D son un ejemplo de hardware altamente paralelo (también denominado *many-core*). Además, en la actualidad, los procesadores gráficos disponen de una gran capacidad para la computación de propósito general, lo que ha permitido extender la computación en GPU o GPGPU (*General-Purpose computation on Graphics Processing Units*) [101] a múltiples ámbitos de la ciencia y de la ingeniería [102]. Debido a su rendimiento, eficiencia energética y bajo coste [103, 104, 105, 106], las GPU se han convertido en los aceleradores hardware más utilizados en el ámbito de la computación de altas prestaciones [107, 108, 109, 110].

El empleo de recursos hardware especializados para la resolución de problemas de propósito general da lugar a la computación heterogénea, que se puede definir como el uso de unidades computacionales (o dispositivos) de diferentes tipos o arquitecturas. La figura 1.3 representa un sistema heterogéneo compuesto por una CPU y una GPU que se encuentran interconectadas mediante un bus de tipo PCI Express. La parte relativa a la CPU y la memoria principal se denomina *host* o anfitrión, mientras que la parte correspondiente con la GPU se conoce como *device* o dispositi-

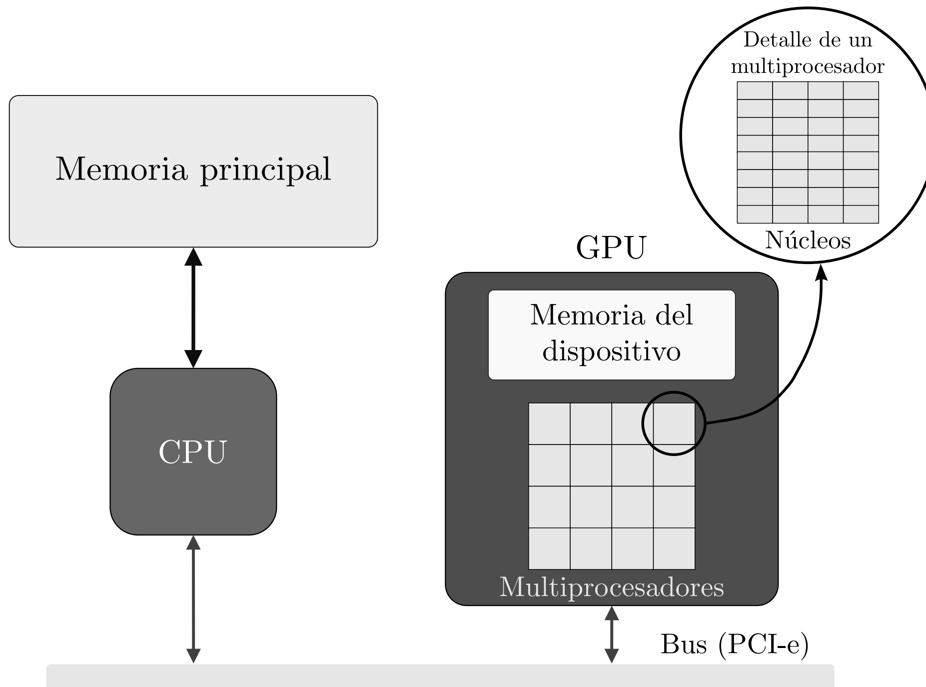


Figura 1.3: Representación esquemática de un sistema heterogéneo basado en CPU + GPU.

vo. Habitualmente, la memoria del anfitrión y la memoria del dispositivo son independientes, por lo que toda necesidad de coherencia entre ambas memorias debe ser gestionada de forma explícita (copiando datos entre el anfitrión y el dispositivo a través del bus que los conecta).

De entre las distintas tecnologías para la aceleración de aplicaciones de propósito general con GPU, en esta Tesis se ha decidido utilizar CUDA [111], que es una API específicamente diseñada para su uso con las GPU de NVIDIA que cuentan con una microarquitectura Tesla o superior.

A continuación, se aclaran ciertos conceptos propios del entorno seleccionado (API y hardware CUDA) [112]. En CUDA, una función que se invoca desde la CPU para ser ejecutada en la GPU se denomina *kernel*. Por tanto, una implementación heterogénea de un algoritmo para CPU +

GPU consta de uno o más *kernels* que permiten descargar ciertos cálculos sobre la GPU. Al igual que en el caso de las CPU, en CUDA, la entidad mínima encargada de ejecutar un *kernel* es el hilo o *thread*. Por su parte, los hilos se organizan en bloques, de tal forma que los hilos pertenecientes a un mismo bloque tienen la capacidad de compartir datos (usando una pequeña memoria compartida) y de sincronizar su ejecución. Por último, el conjunto de todos los bloques de hilos que componen un *kernel* se denomina *grid*. Además, cabe mencionar que los hilos pertenecientes a distintos bloques deben ser totalmente independientes, ya que no pueden sincronizar su ejecución (de forma sencilla). En resumen, un *kernel* consta de un *grid*, que a su vez se divide en uno o más bloques compuestos por uno o más hilos.

Software	$\Rightarrow$	**Hardware**
kernel	$\Rightarrow$	GPU
grid	$\Rightarrow$	varios multiprocesadores
bloque CUDA	$\Rightarrow$	un multiprocesador
hilo CUDA	$\Rightarrow$	núcleo

Tabla 1.1: Relación entre software y hardware en CUDA.

En la tabla 1.1, se muestra cómo las diferentes entidades dentro de la jerarquía software CUDA se asignan al hardware de la GPU (ver figura 1.3). Un *kernel* se ejecuta sobre una GPU y un *grid* se asigna a varios multiprocesadores (como mínimo uno y como máximo el número total de multiprocesadores). Por su parte, un bloque de hilos se asigna a un multiprocesador y, por último, cada hilo CUDA está asociado a un núcleo de la GPU.

La jerarquía de memoria de una GPU NVIDIA también se organiza en diferentes niveles, cada uno de ellos con sus características particulares. A continuación, se enumeran los niveles de memoria más importantes, ordenados de mayor a menor velocidad (y de menor a mayor tamaño):

- Registros. Es el nivel más rápido dentro de la memoria de la GPU. Son privados para cada hilo y es un recurso limitado, especialmente

si se tiene en cuenta el alto grado de paralelismo.

- Memoria compartida y caché L1. La memoria compartida permite compartir datos entre los diferentes hilos que componen un bloque, mientras que la caché L1 tiene como objetivo acelerar los accesos reiterados a posiciones de memoria pertenecientes a niveles más lentos. Ambos tipos de memoria son propios de cada multiprocesador y se implementan sobre el mismo chip. En las GPU de la serie Fermi y posteriores, la cantidad de memoria dedicada a cada rol (memoria compartida o caché L1) es configurable.
- Caché L2. De mayor tamaño que la caché L1 y única para toda la GPU, esta memoria también permite acelerar los accesos a posiciones de memoria pertenecientes a niveles más lentos.
- Memoria global del dispositivo. Todos los hilos tienen acceso de lectura/escritura a este nivel de memoria. Representa el nivel más lento dentro de la jerarquía de memoria de una GPU, pero también el de mayor capacidad. Puede equipararse a la memoria principal de una CPU.

Además, una GPU CUDA también cuenta con otros tipos de memoria más específicos, como la memoria y la caché para constantes, o la memoria y la caché para texturas [112].

1.3.4. Coprocesadores Xeon Phi

En el ámbito de los sistemas heterogéneos para la computación de altas prestaciones, los coprocesadores de tipo Xeon Phi [91] son otro ejemplo de dispositivo *many-core* cuyo uso se ha extendido en los últimos años [110, 113].

La figura 1.4 representa un sistema heterogéneo compuesto por una CPU y un coprocesador Xeon Phi interconectados mediante un bus PCI Express. Puesto que la memoria principal de la CPU y la memoria de coprocesador son independientes, el intercambio de información entre ambas debe realizarse copiando datos entre la CPU y el Xeon Phi a través del bus

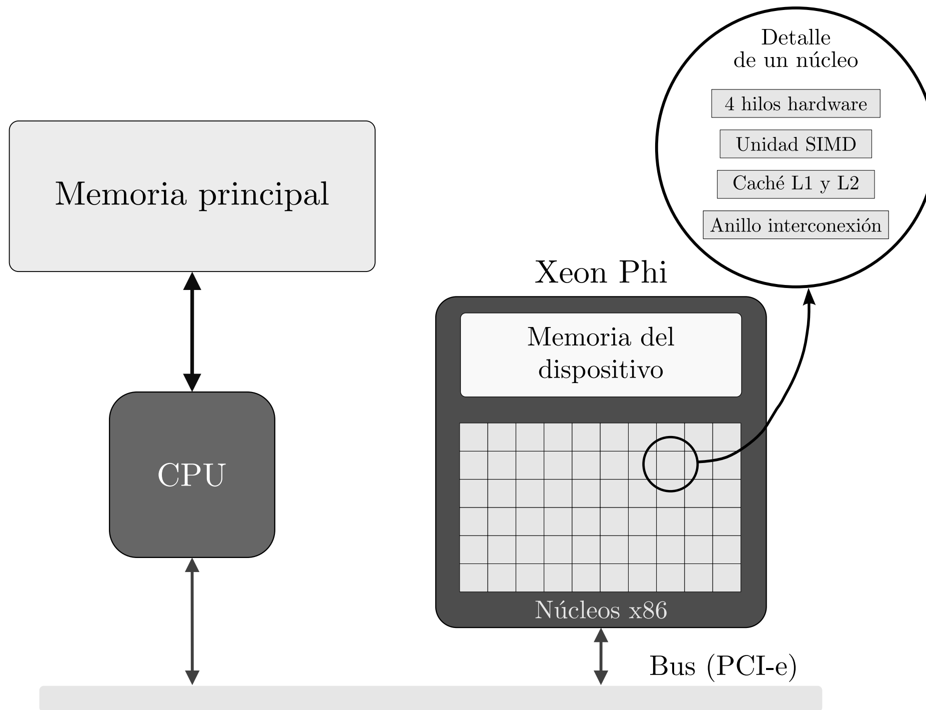


Figura 1.4: Representación esquemática de un sistema heterogéneo basado en CPU + Xeon Phi.

que los conecta, ya sea de forma explícita (mediante directivas incluidas por el programador) o implícita (de forma automática). Además, como se indica también en la figura 1.4, resulta interesante mencionar una de las ventajas de estos coprocesadores: su compatibilidad a nivel de ISA (*Instruction Set Architecture*) con los procesadores convencionales más habituales (x86-64).

Para desarrollar herramientas capaces de aprovechar los coprocesadores Xeon Phi, se pueden utilizar diferentes modelos o técnicas de programación [114], las cuales pueden englobarse en dos grandes grupos:

- El modelo *offload*, en el que el cauce de ejecución principal se lleva a cabo en la CPU y se descargan sobre el coprocesador ciertos cálculos. Este modelo es similar al modelo usado en CUDA con las GPU.

- El modelo nativo, en el que la aplicación se ejecuta de manera nativa tanto en la CPU como en el coprocesador. En este caso, tanto el procesador como el coprocesador pueden comunicarse entre sí de varias formas, siendo el uso de MPI la manera más habitual.

En esta Tesis, se ha utilizado un modelo de programación del primer grupo conocido como CAO (*Compiler-assisted Offload*). En este modelo, las zonas de código a ejecutar en el coprocesador se delimitan mediante el uso de *pragmas* o directivas (similares a las utilizadas en OpenMP), las cuales permiten también definir de manera explícita el intercambio de datos entre la CPU y el Xeon Phi. El uso de los *pragmas* específicos para Xeon Phi junto con las construcciones habituales de OpenMP para la ejecución paralela permite acelerar algoritmos de forma sencilla con un rendimiento similar al que se puede alcanzar con el resto de modelos [114].

1.4. Contribuciones de la Tesis

Puesto que esta Tesis se focaliza en el desarrollo de un conjunto de herramientas software que permitan la resolución eficiente de diversos problemas dentro del ámbito de la dispersión acústica y electromagnética, las contribuciones más destacables se centran en el análisis, diseño e implementación de algoritmos y técnicas para la paralelización aplicados a métodos ya presentes en la literatura científico-técnica.

En el ámbito de la dispersión acústica, destacan las implementaciones paralelas y heterogéneas, para sistemas de memoria distribuida basados en CPU + GPU, del FMM [115] y del FMM-FFT [116]. Dichas implementaciones permiten además la ejecución simultánea de la traslación y de las interacciones cercanas, gracias a la descomposición funcional ideada, en la cual la traslación se ejecuta en las CPU mientras que el cálculo de las interacciones cercanas se lleva a cabo en las GPU. Además, también se ha generado una herramienta paralela y heterogénea para el cálculo de la presión total en el problema de dispersión acústica que es capaz de ejecutarse sobre tres arquitecturas diferentes: CPU, GPU y Xeon Phi [117, 118].

El diseño de las herramientas mencionadas en el párrafo anterior se ha orientado hacia la escalabilidad, contemplando dos perspectivas diferentes: la escalabilidad fuerte (ley de Amdahl [119]) y la escalabilidad débil (ley de Gustafson [120]). La escalabilidad fuerte se centra en la reducción de los tiempos de ejecución para un problema determinado (tamaño fijo), mientras que la escalabilidad débil está relacionada con la posibilidad de resolver problemas de mayor tamaño cuando se dispone de sistemas con mayor paralelismo (tamaño escalado). En esta Tesis, gracias al aprovechamiento de las arquitecturas de tipo *many-core*, ha sido posible lograr una reducción de los tiempos de ejecución y se ha ampliado el abanico de problemas resolubles en tiempos razonables. De esta forma, la resolución de problemas de gran tamaño en estaciones de trabajo (o pequeños *clusters*) se convierte en una alternativa al uso de *clusters* convencionales, logrando además una mejora en la eficiencia energética de la solución aportada.

Por su parte, en el ámbito de la dispersión electromagnética, se ha llevado a cabo una implementación paralela y heterogénea, para sistemas de memoria compartida de tipo CPU + GPU, del SRM con dos aplicaciones diferentes: caracterización de antenas e imagen electromagnética. Para el caso de la caracterización de antenas, se ha desarrollado una implementación en la que se utiliza la GPU para acelerar el cálculo de los productos matriz-vector asociados a las iteraciones del método CGNR empleado [121]. Por su parte, la herramienta de *imaging* para reconstrucción de perfiles consiste en una implementación para múltiples procesadores gráficos en la que cada GPU se encarga de analizar una parte de los diferentes ángulos de incidencia utilizados para iluminar el objeto bajo medida [70]. Estas herramientas forman parte de los resultados generados dentro de uno de los paquetes de trabajo del proyecto "Técnicas de imaging mediante problema inverso de dispersión: nuevos algoritmos y técnicas de medida – iScat" (Ref.: TEC2011-24492).

Las herramientas para el problema electromagnético inverso descritas en el párrafo anterior se utilizan habitualmente para resolver problemas de menor peso computacional, en términos de tiempo de ejecución, que en el caso de las herramientas para dispersión acústica mencionadas anteriormente. Por tanto, en este caso, el objetivo principal ha consistido en lograr una gran escalabilidad fuerte que permita resolver los problemas habituales

(de tamaño moderado) en tiempo real o cuasi-real utilizando un hardware sencillo y asequible.

En resumen, se han desarrollado métodos y técnicas escalables y eficientes para acústica y electromagnetismo que, además, pueden aplicarse a otros problemas. En el caso del FMM y del FMM-FFT, las herramientas para acústica pueden adaptarse a otros ámbitos (p. ej., electromagnetismo) realizando modificaciones en la formulación pero conservando el esquema de paralelización planteado. En general, todas las técnicas diseñadas para abordar de forma eficiente el particionado, las comunicaciones y el balanceo de carga, pueden extrapolarse a otros problemas con una algoritmia similar.

1.5. Estructura de la memoria de la Tesis

La Tesis se divide en siete capítulos y un apéndice. Este primer capítulo contiene la motivación, junto con una revisión de los antecedentes y el estado actual de la cuestión, y una breve revisión de las tecnologías utilizadas. Además, en este capítulo se han incluido las contribuciones aportadas en esta Tesis.

En el capítulo 2 se analiza la algoritmia diseñada para implementar el método multipolar rápido, en su versión mono-nivel, para acústica. En primer lugar, se realiza una breve descripción del método. A continuación, se describen de forma detallada las estrategias utilizadas para implementar el FMM para sistemas de memoria distribuida con nodos heterogéneos (de tipo CPU + GPU). En la parte final del capítulo, se muestran resultados prácticos de la presión acústica sobre geometrías de interés y se realiza una validación de los resultados obtenidos.

A continuación, en el capítulo 3, se presentan dos versiones diferentes de una herramienta basada en el método FMM-FFT para acústica. Se comienza con una breve descripción del método, centrada en las diferencias respecto del FMM. Posteriormente, se analizan en detalle las técnicas usadas en el desarrollo de las dos implementaciones llevadas a cabo: una para sistemas de memoria distribuida convencionales (sólo CPU) y otra para sis-

temas de memoria distribuida heterogéneos (CPU + GPU). Para terminar, se lleva a cabo una validación de los resultados obtenidos con las diferentes versiones de la herramienta implementada.

El capítulo 4 se centra en el desarrollo de una herramienta, con múltiples variantes, para el cálculo de la presión total en planos situados en el exterior de un obstáculo sobre el que incide una onda acústica. Para comenzar, se realiza una breve descripción del método, que utiliza como punto de partida los resultados obtenidos con cualquiera de los métodos presentados en los dos capítulos anteriores. Seguidamente, se detallan todas las técnicas desarrolladas para implementar las diferentes variantes del algoritmo, con versiones para sistemas de memoria compartida basados en procesadores convencionales, y para sistemas heterogéneos del tipo CPU + Xeon Phi, CPU + GPU y CPU + GPU + Xeon Phi. Para finalizar el capítulo, se muestran varios resultados obtenidos con esta herramienta.

En el capítulo 5, se presentan dos herramientas con propósitos diferentes que abordan el problema inverso para electromagnetismo usando el método de reconstrucción de fuentes. Se comienza describiendo la primera de ellas, enfocada a la caracterización de antenas, y se detalla la implementación del SRM para procesadores gráficos. Posteriormente, se explica el funcionamiento de la segunda herramienta, con aplicación en problemas de imagen electromagnética para reconstrucción de perfiles e implementada para sistemas heterogéneos con múltiples GPU. Para concluir, se muestran varios ejemplos de aplicación de las herramientas desarrolladas.

En el capítulo 6 se recogen los resultados computacionales más relevantes del conjunto de herramientas desarrollado en esta Tesis. Se muestran resultados relativos a las métricas computacionales más habituales (tiempo de ejecución, consumo de memoria, aceleración y eficiencia), se estudia la ganancia de las soluciones heterogéneas frente a las implementaciones paralelas clásicas, y se analiza la eficiencia energética en el caso de las herramientas para acústica basadas en el FMM y el FMM-FFT. Este capítulo se apoya en el apéndice A, donde se detallan las características de los diferentes sistemas utilizados, así como los detalles de la ejecución de cada solución.

Por último, el capítulo 7 contiene las conclusiones de esta Tesis. Asi-

mismo, se presentan posibles líneas futuras que pueden suponer una continuación del trabajo llevado a cabo por el doctorando.

Capítulo 2

Algoritmo FMM paralelo y heterogéneo aplicado al problema de dispersión acústica

Índice

2.1. Descripción del cálculo de la presión acústica . 31

2.2. Algoritmo FMM aplicado al problema de dispersión acústica 34

2.2.1. Inicialización 37

2.2.2. Interacciones cercanas 40

2.2.3. Interacciones lejanas 41

2.2.4. Suma de interacciones 44

2.3. Paralelización para aceleradores gráficos 44

2.3.1. Decisiones de diseño de la implementación propia 45

2.3.2. Inicialización 52

2.3.3. Interacciones cercanas 57

2.3.4. Agregación 61

2.3.5. Traslación 66

2.3.6. Desagregación 70
2.4. Resultados 73
2.4.1. Presión acústica 73
2.4.2. Validación 76

A lo largo de este capítulo, se describe una herramienta diseñada para resolver problemas de dispersión acústica utilizando el algoritmo FMM y aprovechando la potencia de cálculo de las GPU actuales. Esta herramienta puede ser ejecutada sobre equipos con un único procesador gráfico, sobre sistemas con múltiples tarjetas gráficas, o sobre sistemas de memoria distribuida cuyos nodos respondan a una de las dos configuraciones anteriores.

Con esta herramienta se persiguen dos objetivos principales, reducir de forma notable el tiempo necesario para resolver un determinado problema (tiempo de respuesta) y mejorar sustancialmente la eficiencia energética, tomando como referencia una implementación paralela optimizada para procesadores convencionales. El primero de los objetivos debe permitir ampliar el rango de problemas abordables por estaciones de trabajo personales, sin necesidad de recurrir a un *cluster* de altas prestaciones. El segundo se centra en la reducción de la energía necesaria para resolver un determinado problema, atacando los dos factores presentes en el consumo energético: tiempo y potencia.

El lenguaje de programación elegido para desarrollar esta herramienta ha sido CUDA C, ya que permite aprovechar todo el potencial que ofrecen las tarjetas gráficas utilizadas a lo largo del desarrollo de la Tesis, basadas en procesadores gráficos NVIDIA de las series Fermi y Kepler.

En el apartado 2.1, se realiza una breve descripción de cómo se lleva a cabo el cálculo de la presión acústica en la superficie de un objeto dispersor.

A continuación, en el apartado 2.2, se describe de forma somera el algoritmo FMM y las diferentes etapas que lo componen, particularizando dicha descripción a la implementación propia del algoritmo llevada a cabo en esta Tesis. Asimismo, se analiza el coste temporal asociado a cada etapa.

Posteriormente, en el apartado 2.3, se analizan las estrategias computacionales que se han adoptado para desarrollar esta herramienta. De en-

tre las distintas decisiones de diseño, cabe mencionar el particionado [79] ideado, con una descomposición funcional que permite realizar de forma simultánea la traslación (en CPU) y el cálculo de las interacciones cercanas (en GPU), y una división del dominio de doble nivel (grano fino-grueso) que permite hacer un uso eficiente de los recursos, combinando MPI, OpenMP y CUDA para paralelizar la computación en CPU y GPU. Finalmente, merece la pena destacar el diseño del balanceo de carga, especialmente entre la CPU y la GPU, que se basa en un modelo que permite estimar a priori el tiempo por iteración en función de varios parámetros, haciendo posible un ajuste de dichos parámetros con el objetivo de minimizar el tiempo de ejecución.

Por último, en el apartado 2.4, se presentan varios resultados prácticos obtenidos con esta herramienta. Posteriormente, se muestran resultados de validación para determinar la corrección de la implementación. Al igual que en otros capítulos, los resultados computacionales correspondientes a este algoritmo se han recogido en el capítulo 6.

2.1. Descripción algorítmica del cálculo de la presión acústica en la superficie de un objeto dispersor

El problema estudiado consiste en predecir la presión acústica dispersada por un obstáculo sobre el que incide una onda acústica. El obstáculo, cuya superficie se nota como S, se encuentra iluminado por una onda incidente, P^i, cuyo valor es conocido (se considera que la fuente de ruido se encuentra correctamente caracterizada). Como consecuencia de la presencia de dicho obstáculo, se produce una onda dispersada, que nota como P^s. La presión total en la superficie del obstáculo (y en cualquier punto del espacio que lo rodea), P, puede expresarse como el resultado de la superposición del campo incidente y de la perturbación que representa el campo dispersado. Notando como $\mathbf{r}$ al vector de posición de un punto cualquiera

del espacio, se tiene que:

$$P(\mathbf{r}) \;=\; P^i(\mathbf{r}) \;+\; P^s(\mathbf{r}). \tag{2.1}$$

Para poder obtener la presión total, P, se recurre a un problema equivalente similar al definido por el teorema de equivalencia en dispersión electromagnética. El problema equivalente en dispersión acústica puede expresarse como la superposición de dos problemas, uno para el campo incidente y otro para el campo dispersado, en los que el obstáculo se sustituye por un medio con las mismas propiedades acústicas que el medio de propagación (véase el capítulo 2 de [13]).

De esta forma, se tienen dos ecuaciones, una para el campo incidente y otra para el campo dispersado, cuya suma, en combinación con las condiciones de contorno adecuadas, permite obtener la ecuación integral de contorno convencional (CBIE). La CBIE es una ecuación integro-diferencial que relaciona la presión incidente con los valores de la presión acústica total sobre S y de su derivada respecto de la dirección normal a S. Siguiendo el procedimiento citado anteriormente, en el que se formulan dos ecuaciones para el problema equivalente, una para el campo incidente y otra para el campo dispersado, puede obtenerse la siguiente ecuación para la CBIE [122]:

$$\frac{1}{2}P(\mathbf{r}) + \int_S \left[G(\mathbf{r},\mathbf{r}')\frac{\partial P(\mathbf{r}')}{\partial n(\mathbf{r}')} - P(\mathbf{r}')\frac{\partial G(\mathbf{r},\mathbf{r}')}{\partial n(\mathbf{r}')} \right] \mathrm{d}S(\mathbf{r}') \;=\; P^i(\mathbf{r}), \tag{2.2}$$

donde $P(\mathbf{r}')$ es la presión total debida a una fuente ficticia situada en el punto $\mathbf{r}' \in S$, $n(\mathbf{r}')$ es la dirección normal a S en el punto $\mathbf{r}'$, y $P^i(\mathbf{r})$ es la presión incidente en el punto de observación $\mathbf{r}$. Por su parte, $G(\mathbf{r},\mathbf{r}')$ es la función de Green de espacio libre, que se define como sigue:

$$G(\mathbf{r},\mathbf{r}') \;=\; \frac{e^{-j\kappa|\mathbf{r}-\mathbf{r}'|}}{4\pi|\mathbf{r}-\mathbf{r}'|}, \tag{2.3}$$

donde κ representa el número de onda.

En esta Tesis, la superficie del obstáculo se modela como una superficie perfectamente rígida, por lo que la derivada de la presión respecto de la

dirección normal se anula sobre S.

Para resolver el problema de la no unicidad (o de la frecuencia de resonancia) de la CBIE, se puede utilizar la solución inicialmente sugerida por Burton y Miller en [123]. Para ello, la CBIE se combina con su derivada respecto de la dirección normal al punto de observación, conocida como ecuación integral de contorno hipersingular (HBIE), de la siguiente forma:

$$\text{CBIE} \; - \; \alpha\text{HBIE} \quad | \quad \mathcal{I}m\{\alpha\} \; \neq \; 0. \tag{2.4}$$

A su vez, la formulación de Burton y Miller requiere de algún procedimiento de regularización, como el sugerido en [124], el cual se analiza en detalle en [13].

El método de los elementos de contorno (BEM) [16] permite discretizar ecuaciones como (2.2) y (2.4) dando lugar a un sistema de ecuaciones. Primero, la distribución de presión sobre el obstáculo (desconocida) se expande en una serie de N funciones base. Posteriormente, dichas funciones base se ponderan por un conjunto de coeficientes, produciendo un conjunto de N ecuaciones, cuya representación matricial es la siguiente [13]:

$$Kp \; = \; g, \tag{2.5}$$

donde K es la matriz de acoplos del sistema. Por su parte, g es el vector de excitación, que está relacionado con la presión acústica incidente, la cual es conocida. Finalmente, p es el vector solución que representa los coeficientes de la expansión a través de los cuales se puede obtener la presión desconocida.

En el caso de esta Tesis, se utilizan funciones base constantes sobre cada elemento en el que se discretiza la superficie del objeto dispersor. Además, se emplean funciones de ponderación de tipo delta de Dirac emplazadas en el centroide de cada elemento (generalmente triángulos). Puesto que el número de funciones de ponderación y de funciones base es el mismo, el sistema de ecuaciones resultante tiene tantas ecuaciones linealmente independientes como incógnitas, produciendo una única solución.

En un primer paso, la resolución del sistema planteado en (2.5), produce

la distribución de presión sobre S. Posteriormente, en un segundo paso, se puede obtener la presión dispersada en cualquier punto del espacio exterior a S partiendo del campo acústico obtenido en esta etapa.

2.2. Algoritmo FMM aplicado al problema de dispersión acústica

En la herramienta presentada en este capítulo para resolver el sistema de ecuaciones (2.5), el algoritmo FMM [40, 125] se utiliza como esquema acelerador del método iterativo GMRES [29]. Según el estudio realizado por Marburg y Schultz en [30], el GMRES ha demostrado ser un método muy robusto para la resolución de problemas de dispersión acústica. Además, dicho estudio también prueba que el GMRES es, en general, el método iterativo más eficiente para el tipo de problemas que se abordan en esta parte de la Tesis.

El GMRES es una extensión del método de residuo mínimo (*Minimum Residual* o MINRES) [126] que permite resolver sistemas de ecuaciones lineales no simétricos. Al igual que su antecesor, el GMRES produce una secuencia de vectores ortogonales que generan un subespacio de Krylov. Sin embargo, debido a la asimetría del sistema de ecuaciones lineales, es necesario almacenar todos los vectores calculados previamente para poder generar la solución final. Este requisito podría representar un problema, puesto que el espacio en memoria para almacenar los vectores crece de forma lineal con el número de iteraciones. Una forma de evitar dicho problema sería aplicar reinicios al algoritmo GMRES, pero en esta Tesis se ha optado por una solución más eficiente que consiste en almacenar estos vectores en memoria secundaria usando un fichero de tipo binario. De esta forma, no ha sido necesario aplicar reinicios en el algoritmo [29] para ninguno de los problemas analizados en este trabajo (aunque la posibilidad de reiniciar el algoritmo se encuentra implementada), evitando así la ralentización de la velocidad de convergencia [22].

La ventaja fundamental del GMRES frente a otros algoritmos de tipo Krylov reside en el hecho de que el cálculo de la norma del residuo se pue-

de llevar a cabo sin necesidad de formar el iterando de manera explícita. Así, la parte computacionalmente más costosa del GMRES se reduce a un único producto matriz-vector (*Matrix-Vector Product*o MVP) por iteración dedicado a obtener el iterando. Una vez que la norma del residuo es suficientemente pequeña, se forma el iterando como una combinación lineal de los vectores previamente almacenados y de la estimación inicial, y se finaliza la ejecución del algoritmo.

Por su parte, la implementación del FMM presentada en este capítulo (que parte de la implementación presentada en el capítulo 5 de [13]) consta de una fase de inicialización, o *setup* del algoritmo, a la que le sigue una fase iterativa cuyo objetivo es obtener un resultado que debe converger hacia la solución del problema, cumpliendo con unos requisitos de exactitud previamente fijados.

En cada una de las iteraciones que componen la fase iterativa, se lleva a cabo un producto matriz-vector, pero sin calcular de forma explícita la matriz de acoplos del sistema (o matriz de rigidez K). Para ello, el producto matriz-vector se sustituye por tres pasos alternativos, el cálculo de las interacciones lejanas, el cálculo de las interacciones cercanas y la suma de ambas interacciones.

Las interacciones cercanas, limitadas a aquellos elementos que pertenecen a grupos cercanos (en este caso, el mismo grupo o grupos adyacentes cuyos bordes comparten al menos un punto), no satisfacen las condiciones asociadas a las transformaciones descritas por los teoremas de adición [40]. Debido a esto, las interacciones cercanas deben ser calculadas evaluando directamente la parte correspondiente de la matriz del sistema.

Por contra, el cálculo de las interacciones lejanas se lleva a cabo de forma eficiente basándose en los teoremas de adición y en una descomposición en ondas planas [127]. Para ello, durante la fase de inicialización, el FMM comienza por dividir los N elementos (funciones base) en N_g grupos disjuntos. Posteriormente, se realiza el cálculo de las interacciones lejanas en tres etapas consecutivas, agregación, traslación y desagregación. En la agregación, el campo lejano producido por los elementos de cada grupo se representa por medio de una expansión de k_l componentes, cada una de las cuales expresa una onda plana propagándose en las diferentes direcciones

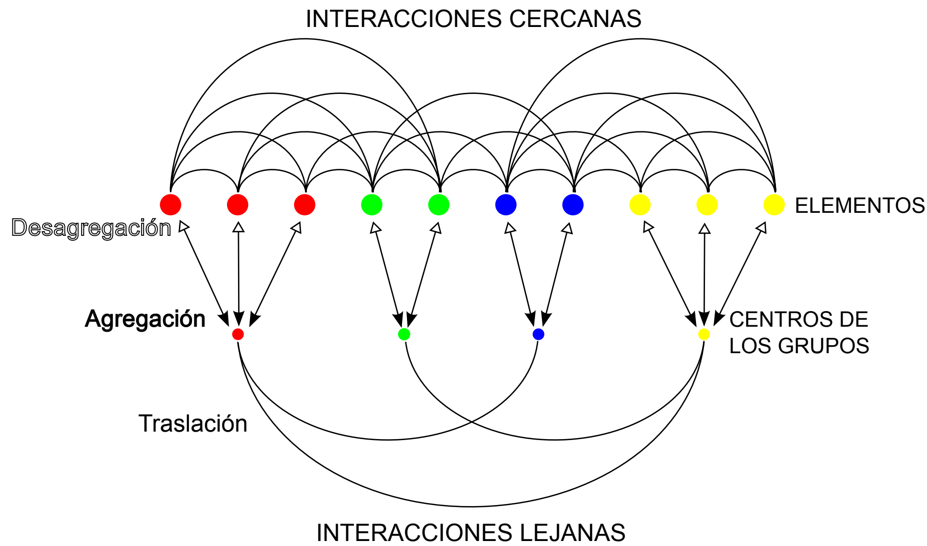


Figura 2.1: Representación esquemática de las distintas etapas del FMM. Interacciones cercanas frente a interacciones lejanas.

del espacio $\boldsymbol{\kappa}$ [128], con referencia de fase en el centro del grupo (véase la agregación en la figura 2.1). Al comienzo de la traslación, las k_l componentes del campo lejano se trasladan a todos los centros de los grupos que son lejanos (en este caso, grupos cuyos bordes no comparten ningún punto). Después, las contribuciones lejanas de todos los grupos se suman en el centro de cada grupo para cada una de las direcciones del espacio $\boldsymbol{\kappa}$ (véase la traslación en la figura 2.1). Finalmente, en la desagregación, se obtiene el campo lejano sobre cada elemento de cada grupo mediante la adición de las componentes del espacio $\boldsymbol{\kappa}$ trasladadas desde el centro del grupo al centro del propio elemento (véase la desagregación en la figura 2.1).

En la figura 2.1, los grupos adyacentes representan grupos cercanos entre sí. Por ejemplo, en el caso del grupo verde sus grupos cercanos serían el grupo rojo y el azul.

2.2.1. Inicialización

El primer paso que se lleva a cabo en el FMM es la inicialización, o *setup*, del propio algoritmo. Durante esta fase, se realizan varias operaciones que son fundamentales para la ejecución del algoritmo. De entre ellas, merece la pena destacar las siguientes:

- Lectura de los datos de entrada.
- Procesado de la geometría.
- Selección del tamaño de los grupos.
- Cálculo de la presión incidente.
- Cálculo del operador de traslación.

La ejecución del algoritmo comienza con la lectura y el procesado de los parámetros de configuración (frecuencia, fuentes de ruido, iteraciones, error residual, etc.) y de los ficheros de entrada que definen la geometría. Con el objetivo de mejorar la velocidad de entrada/salida, todos los ficheros con los que se trabaja son de tipo binario. Estos ficheros permiten obtener mayores tasas de transferencia y ocupan menos espacio en disco que sus equivalentes ASCII, aunque su manejo es más complejo. Sin embargo, para llevar a cabo el post-procesado de los resultados de una manera cómoda, los ficheros binarios de salida pueden convertirse de forma sencilla en ficheros de tipo ASCII utilizando el comando `hexdump` con las opciones adecuadas para cada caso.

Una vez leídos los datos de entrada, se prosigue con el procesado de la malla que describe la geometría bajo análisis. Además, a partir de la información relativa a las facetas que componen la malla y a los vértices que definen cada faceta, se lleva a cabo el cálculo del centro, el área y el vector normal unitario, para cada elemento. De esta forma, se genera la información necesaria para continuar con la ejecución del algoritmo.

A continuación, se lleva a cabo una de las operaciones más importantes dentro de la inicialización, la selección del tamaño de los grupos. Primero,

se comienza con una división espacial de los N elementos o funciones base en N_g grupos disjuntos. Para ello, se utiliza una estructura conocida como *octree*, junto con una ordenación de tipo Z y un entrelazado de los bits que definen las coordenadas de cada grupo [129]. Para ejemplificar de forma sencilla la división espacial llevada a cabo con este tipo de estructuras, en la figura 2.2 se utiliza el equivalente para dos dimensiones del *octree*, denominado *quadtree*. Gracias a estas estructuras, las tareas que involucran a los grupos (pertenencia de un elemento a un grupo, búsqueda del centro de un grupo, búsqueda de grupos cercanos, etc.) pueden realizarse de forma muy eficiente, usando únicamente operaciones a nivel de bit [129].

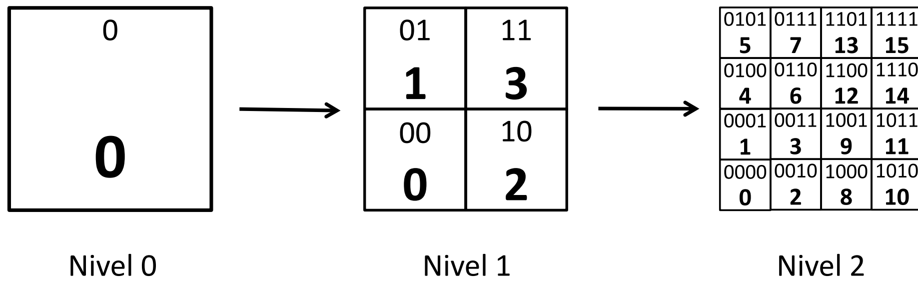


Figura 2.2: Varios niveles de un *quadtree* empleando ordenación Z.

Realizando un análisis asintótico del coste temporal del FMM, puede comprobarse que el tamaño de grupo ha de cumplir que $N_g \propto \sqrt{N}$ para lograr una complejidad $\mathcal{O}(N^{1{,}5})$ [40]. Sin embargo, esto no quiere decir que, para un problema concreto, el coste temporal sea óptimo para $N_g = \sqrt{N}$. Por ello, en el algoritmo FMM presentado en este capítulo, el tiempo por iteración se ha modelado teniendo en cuenta el número de operaciones y el tiempo por operación en cada paso, con el fin de estimar a priori el coste asociado a cualquier problema [13]. Así, en tiempo de ejecución y durante la fase de inicialización, se prueban diferentes divisiones espaciales que se obtienen con distintos tamaños para los grupos que componen el *octree*. Finalmente, de entre todos los tamaños probados en esta búsqueda, se selecciona aquel que lleva asociado un menor coste en términos de operaciones según el modelo empleado. De esta forma, con una sencilla búsqueda que tiene una complejidad temporal $\mathcal{O}(N)$, se puede obtener el

tamaño de grupo óptimo con el que minimizar los tiempos de ejecución. Una vez fijado el tamaño de grupo con el que se va a trabajar, el algoritmo ya está en disposición de calcular el orden de la expansión siguiendo las recomendaciones presentes en [125].

Por su parte, el cálculo de la presión incidente se lleva a cabo teniendo en cuenta que el algoritmo permite trabajar tanto con ondas planas como con fuentes puntuales. En el caso de trabajar con una onda plana, esta queda definida mediante un par de ángulos de incidencia (θ, ϕ). Por contra, si se utilizan fuentes puntuales, la posición de cada una de ellas se define mediante una tupla (x, y, z) que representa sus coordenadas cartesianas.

De entre las operaciones más importantes llevadas a cabo en la inicialización, la última de ellas es el cálculo del operador, o función de transferencia ($\mathcal{T}$), utilizado para llevar a cabo la traslación (puede encontrarse una descripción detallada del cálculo de $\mathcal{T}$ en el capítulo 4 de [13]). El operador de traslación debería calcularse para todas las combinaciones de cada dirección del espacio $\boldsymbol{\kappa}$ y de cada dirección espacial que une los centros de dos grupos lejanos. Sin embargo, la división espacial producida por el uso de *octrees* resulta en un entramado regular, lo que propicia que muchas de las direcciones que unen dos centros de grupos lejanos sean idénticas. De esta forma, si se descartan todas estas direcciones redundantes, puede lograrse un cálculo y un almacenamiento más eficientes del operador de traslación [130, 131]. Sin embargo, en el algoritmo implementado para esta Tesis, no se tienen en cuenta las simetrías que pudiesen presentar las direcciones espaciales que unen dos grupos lejanos. El hecho de tener en cuenta estas simetrías podría resultar interesante en implementaciones de tipo secuencial, debido a la reducción en el consumo de memoria y en el tiempo de cálculo del operador. Pero, para las implementaciones de tipo paralelo [13], supone un serio problema, ya que puede limitar el grado de paralelización debido a la dificultad para repartir la carga de trabajo usando las direcciones del espacio $\boldsymbol{\kappa}$.

2.2.2. Interacciones cercanas

En el FMM, el cálculo de las interacciones cercanas se limita a aquellas bases que pertenecen al mismo grupo o a grupos vecinos. En la implementación del FMM desarrollada en esta Tesis, se consideran grupos vecinos a aquellos grupos cuyos bordes comparten al menos un punto común (véase la figura 2.3). Puesto que estas interacciones no satisfacen los teoremas de adición [40], deben ser evaluadas directamente, lo que requiere el cálculo de la radiación producida por cada pareja de bases que pertenezcan al conjunto de grupos cercanos (véase el capítulo 4 de [13]).

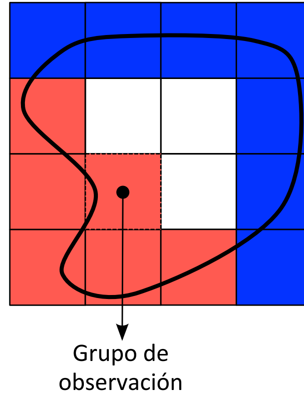


Figura 2.3: Grupos cercanos (en rojo) y grupos lejanos (en azul). Los grupos vacíos aparecen representados en blanco.

Una vez descrita esta etapa y teniendo en cuenta el perfil de esta Tesis, resulta conveniente analizar el coste computacional del cálculo de las interacciones cercanas para un problema arbitrario de tamaño N. Dicho coste se puede modelar utilizando las siguientes expresiones:

$$O_c = n_{pi} \cdot \sum_{i=0}^{N_g-1} n_i \cdot n_i^{(c)}, \tag{2.6a}$$

$$T_c \approx F_c \cdot O_c, \tag{2.6b}$$

donde n_{pi} es el número de puntos de integración de la cuadratura gaussiana

(en este caso, puede considerarse un factor constante) y N_g es el número de grupos no vacíos. Por su parte, n_i es el número de elementos del grupo i-ésimo y $n_i^{(c)}$ es el número de elementos cercanos al grupo i-ésimo. Para obtener el coste temporal (en segundos) de las interacciones cercanas, T_c, a partir del número de operaciones llevadas a cabo, O_c, se incorpora un factor ajustable, F_c, que permite modelar el tiempo por operación (en segundos por operación) para esta etapa. Este factor es dependiente del contexto (implementación, compilador, hardware, etc.) por lo que debe ser estimado y ajustado de forma experimental. En resumen, F_c aglutina el número de instrucciones básicas por operación y la velocidad con la que los procesadores empleados son capaces de ejecutarlas. Para ajustar este tipo de factor, basta con fijar un valor inicial arbitrario (dentro de unos límites razonables), resolver un problema a modo de prueba, y reajustar el valor del factor teniendo en cuenta la relación entre el tiempo real de ejecución y el tiempo estimado inicialmente.

2.2.3. Interacciones lejanas

Como se ha comentado anteriormente, en el FMM las interacciones lejanas se calculan de forma eficiente basándose en los teoremas de adición y en una descomposición de las ondas esféricas en ondas planas [127]. El cálculo de estas interacciones se lleva a cabo en tres etapas que deben realizarse de forma secuencial una tras otra, puesto que tienen una relación de precedencia de tipo final-comienzo. Estas etapas, analizadas a continuación, son la agregación, la traslación y la desagregación.

2.2.3.1. Agregación

En esta etapa del cálculo de las interacciones lejanas, se representa la radiación producida por las bases pertenecientes a cada grupo mediante una expansión multipolar de L multipolos situados en el centro del grupo (se ha mostrado una representación esquemática en la figura 2.1). Para un grupo fuente concreto, la radiación producida por cada función base que pertenece a dicho grupo se traslada al centro del grupo, donde se suma a la del resto de fuentes trasladadas de ese mismo grupo. De esta forma,

la agregación requiere calcular k_l componentes del espacio $\boldsymbol{\kappa}$ para cada elemento (véase el capítulo 4 de [13]).

De esta forma, el coste computacional para la agregación en un problema de tamaño N puede modelarse usando las expresiones que se muestran a continuación:

$$O_a = n_{pi} \cdot k_l \cdot N, \tag{2.7a}$$

$$T_a \approx F_a \cdot O_a, \tag{2.7b}$$

donde O_a representa el número de operaciones que se llevan a cabo en esta etapa, mientras que T_a representa el coste temporal. Para poder obtener el coste temporal a partir del número de operaciones, nuevamente es necesario incorporar un factor (en este caso, F_a) que permita modelar el tiempo por operación. Como se comentaba para el caso de las interacciones cercanas, este factor también es dependiente del contexto, por lo que debe ser estimado y ajustado de forma experimental.

2.2.3.2. Traslación

Esta etapa es fundamental en el algoritmo FMM, ya que permite que este esquema acelerador pueda lograr una reducción en la complejidad temporal frente a los algoritmos iterativos no acelerados (GMRES, CG, etc.). Esto se debe a que la traslación permite la diagonalización en el cálculo de las interacciones entre aquellas bases que son lejanas (la figura 2.1 muestra una representación esquemática de esta etapa).

En la traslación, desde el centro de cada grupo fuente (origen), la expansión multipolar producida en la agregación se traslada a todos los centros de aquellos grupos de observación (destino) que son lejanos. Concretamente, para cada dirección del espacio $\boldsymbol{\kappa}$ se acumulan en el centro de cada grupo de observación (destino) las contribuciones procedentes de todos los grupos lejanos. Así, al trasladar (y sumar) todas las contribuciones de un grupo en una sola vez y no base a base, se logra diagonalizar el cálculo de las interacciones lejanas (véase en más detalle en el capítulo 4 de [13]).

Tras analizar la traslación, merece la pena estudiar el coste computacional de la misma, para lo que se utilizan las siguientes expresiones:

$$O_t = k_l \cdot \sum_{i=0}^{N_g-1} g_i^{(l)}, \tag{2.8a}$$

$$T_t \approx F_t \cdot O_t, \tag{2.8b}$$

donde $g_i^{(l)}$ es el número de grupos lejanos al grupo i-ésimo. Por una parte, O_t representa el número de operaciones que se llevan a cabo en la traslación, mientras que T_t representa su coste temporal. Nuevamente, para obtener el coste temporal a partir del número de operaciones, se incorpora un nuevo factor, en este caso F_t, que permite modelar el tiempo por operación para esta etapa. Como este factor también es dependiente del contexto, debe ser ajustado usando datos empíricos.

2.2.3.3. Desagregación

En la última etapa del cálculo de las interacciones lejanas, la desagregación, se calcula el campo lejano sobre las bases de cada grupo mediante la expansión de las componentes del espacio $\boldsymbol{\kappa}$, las cuales son trasladadas desde el centro del grupo al propio elemento (puede verse una representación esquemática en la figura 2.1). De esta manera, se completa la diagonalización del cálculo de las interacciones lejanas (véase el capítulo 4 de [13]).

Tras definir esta etapa, resulta conveniente analizar el coste computacional de la misma, para lo que se utilizan las siguientes expresiones:

$$O_d = k_l \cdot N, \tag{2.9a}$$

$$T_d \approx F_d \cdot O_d, \tag{2.9b}$$

Siguiendo con la misma notación que en los puntos anteriores, O_d representa el número de operaciones que se llevan a cabo en esta etapa del algoritmo FMM, mientras que T_d representa su coste temporal expresado

en segundos. En este caso, F_d es el factor que se utiliza para modelar el tiempo por operación en la desagregación que, al igual que en el resto de casos, se debe estimar por medio de datos empíricos puesto que depende de la implementación, del compilador, del hardware, etc.

2.2.4. Suma de interacciones

Una vez que se dispone de los datos relativos a las interacciones lejanas y a las interacciones cercanas, ambas interacciones se suman elemento a elemento. De esta forma, se puede obtener el resultado del producto matriz-vector necesario por iteración sin necesidad de generar de forma explícita la matriz del sistema.

A continuación, se comprueba si la solución proporcionada por la iteración actual satisface la exactitud que se le ha exigido inicialmente al algoritmo. En caso afirmativo, la ejecución del algoritmo finaliza y la solución alcanzada en la iteración actual es devuelta como solución final. En caso contrario, se actualiza el iterando y se continúa con el proceso iterativo hasta que se alcance una solución satisfactoria, o hasta que se agote el número máximo de iteraciones permitidas, lo que primero suceda. Si se realiza el máximo de iteraciones permitidas y no se logra alcanzar una solución con un error residual por debajo del deseado, la solución devuelta por el algoritmo se corresponde con la solución obtenida en la última de las iteraciones (la mejor solución de las disponibles).

2.3. Paralelización heterogénea para aceleradores gráficos

Algunas particularidades de la implementación paralela y heterogénea del FMM que se muestra en esta Tesis son consecuencia de ciertas decisiones de diseño que, si bien no alteran el comportamiento global del algoritmo, sí que son relevantes desde el punto de vista computacional. Por ello, en el subapartado 2.3.1 se enumeran las decisiones de diseño más destacables. Posteriormente, en los subapartados 2.3.2 a 2.3.6, se analizan de forma

detallada las funciones o *kernels* desarrollados para abordar las distintas etapas del FMM, ya sea utilizando CPU multinúcleo convencionales (etapas en las que interviene el operador de traslación) o GPU (resto de etapas).

2.3.1. Decisiones de diseño de la implementación propia

2.3.1.1. Uso de *octrees*

En la implementación del FMM desarrollada en esta Tesis, se hace uso de las estructuras conocidas como *octrees* junto con su numeración jerárquica [129], con el objetivo de que las tareas que involucran a los grupos (p. ej. la búsqueda de grupos cercanos) se realicen de una forma muy eficiente usando operaciones a nivel de bit. Además, el modelado del tiempo por iteración, teniendo en cuenta el número de operaciones y el tiempo por operación, permite estimar a priori el coste, para un problema dado, usando cualquiera de las posibles divisiones en grupos. Así, el uso de los *octrees* en combinación con el modelado del tiempo por iteración permite obtener un tamaño de grupo óptimo con el que se minimiza el número de operaciones a realizar [13], lo que resulta equivalente a minimizar los tiempos de ejecución si se logra un reparto equilibrado de la carga de trabajo entre los diferentes procesos paralelos.

2.3.1.2. Almacenamiento de matrices y otras estructuras de datos

Los requisitos de almacenamiento en memoria también son de vital importancia para lograr una solución escalable que permita resolver problemas de gran tamaño. Por tanto, se ha tomado la decisión de no pre-calcular ninguna estructura cuyo coste espacial sea mayor que $\mathcal{O}(N)$, manteniendo en memoria únicamente estructuras de tamaño $\mathcal{O}(N)$, lo que permite aumentar de forma considerable el rango de problemas resolubles mediante este algoritmo.

Por otra parte, a la hora de manejar el operador de traslación, se evitan las direcciones redundantes para las distintas combinaciones grupo–grupo.

Sin embargo, debido a las estrategias que se llevan a cabo en la implementación paralela y heterogénea de este algoritmo, no se tienen en cuenta las posibles simetrías en el operador de traslación. Si bien esta decisión incrementa el consumo de memoria para una implementación secuencial, para una implementación paralela no debe suponer ningún problema, ya que el operador no se replica sino que se reparte entre los distintos procesos paralelos mediante una descomposición del dominio [79]. De no hacerlo así, esto es, descartando las simetrías, no sería viable un reparto del operador de traslación por direcciones del espacio $\boldsymbol{\kappa}$.

También merece la pena señalar que todas las matrices empleadas en la implementación de esta herramienta están almacenadas de forma lineal y por filas, algo habitual en el lenguaje C. Sirva como ejemplo una matriz A, de dimensiones $n \times m$, que se ha *linealizado* en un vector v, de tamaño $n \cdot m$. Si se quiere acceder al elemento ubicado en la columna j de la fila i-ésima, se tiene que:

$$A[i,j] \;=\; v[i \cdot m + j], \tag{2.10}$$

para lo que ha de tenerse en cuenta que se utilizan índices que varían entre 0 y $n-1$ para las filas, y entre 0 y $m-1$ para las columnas. De esta forma, el almacenamiento es más eficiente, ya que se evitan los punteros a filas, y el recorrido de los elementos dentro de un bucle se beneficia en mayor medida de la localidad espacial [132], puesto que los elementos están en posiciones de memoria consecutivas.

2.3.1.3. Particionado

Si se tiene en cuenta el enfoque metodológico presentado en [79], resulta indispensable realizar un estudio en relación al particionado, las comunicaciones y el balanceo de la carga de trabajo.

En lo relativo al particionado, las etapas del FMM se han rediseñado para adaptarse a la filosofía *Single-Instruction, Multiple-Thread* (SIMT) propuesta por CUDA [112]. Así, se ha decidido utilizar una estrategia de doble nivel, combinando una descomposición funcional, la cual permite simultanear la traslación y el cálculo de las interacciones cercanas (véase la figura 2.4), y una división del dominio.

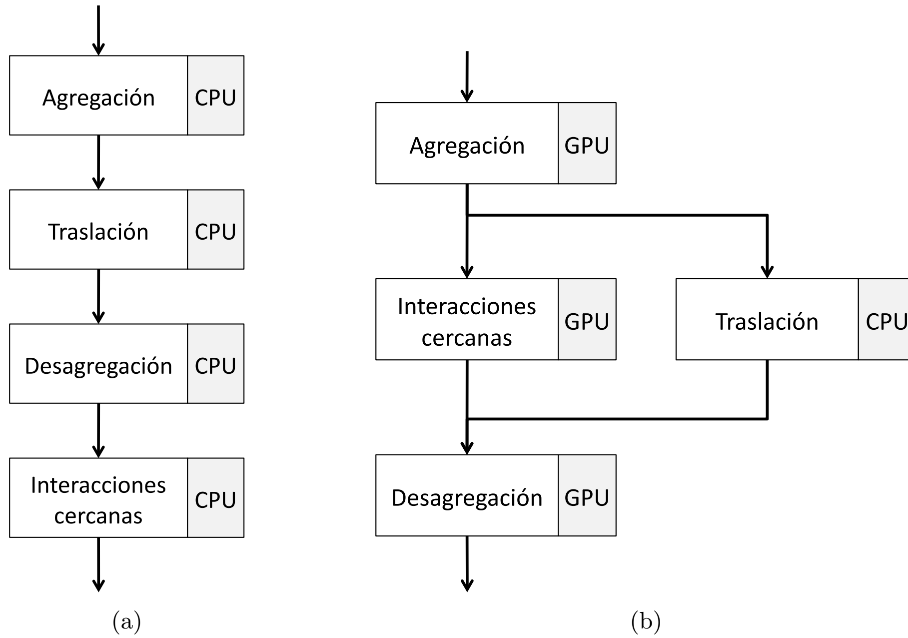


Figura 2.4: Orden de ejecución de las etapas del FMM. (a) Implementación típica para CPU: división del dominio. (b) Implementación ideada para CPU + GPU: descomposición funcional más división del dominio.

Por una parte, la descomposición funcional tiene como objetivo primordial el permitir un uso eficaz de todos los recursos computacionales disponibles. De esta forma, las etapas más intensivas en cálculo (interacciones cercanas, agregación y desagregación) se llevan a cabo en GPU (véase la figura 2.4) aprovechando el alto rendimiento de estos dispositivos en operaciones de coma flotante, mientras que la etapa más intensiva en memoria (traslación) se ejecuta sobre la CPU (véase la figura 2.4) para aprovechar su memoria principal y sus cachés de gran tamaño.

Por otra parte, la división del dominio que se propone aborda el problema desde dos perspectivas complementarias. Para las etapas en las que no interviene el operador de traslación (ejecutadas en GPU), la división se realiza a nivel de grupo (para cada GPU) y a nivel de elemento (para cada hilo CUDA), resultando en un nuevo particionado de doble nivel (grano grueso/fino). Para la traslación (en CPU), la división se hace por

direcciones del espacio $\boldsymbol{\kappa}$ (para cada proceso) y por grupos (para cada hilo OpenMP), produciendo nuevamente un particionado de doble nivel con grano grueso y fino.

La figura 2.5 muestra de forma esquemática la descomposición funcional llevada a cabo para la traslación y las interacciones cercanas, así como la división del dominio en dos niveles para dichas etapas.

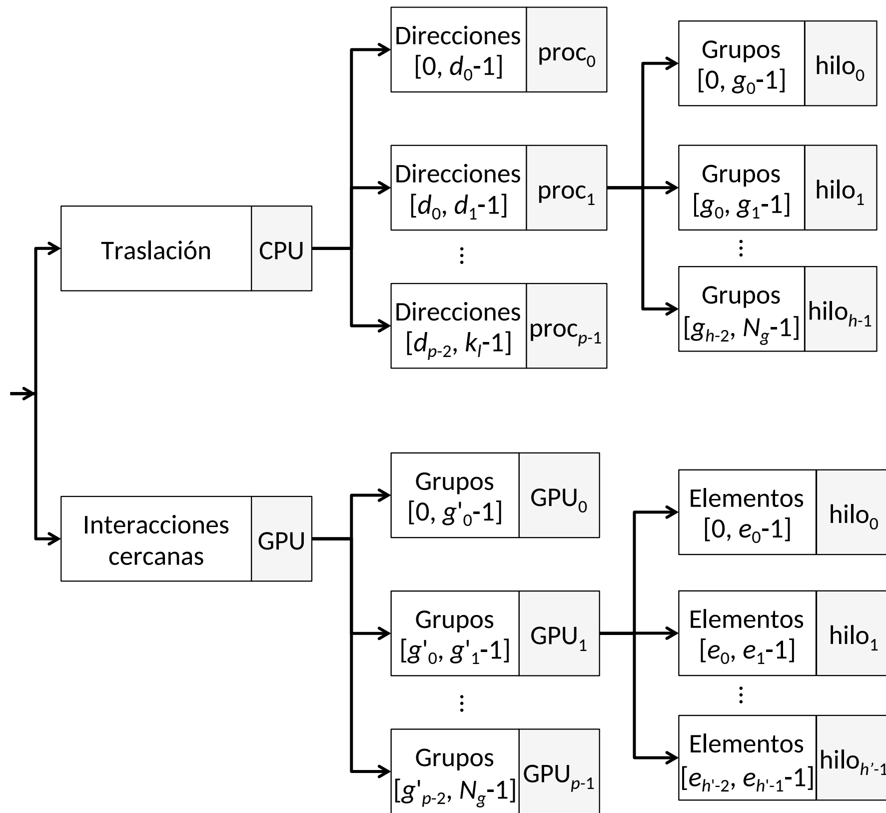


Figura 2.5: Particionado diseñado para el FMM. Descomposición funcional y división del dominio en varios niveles.

2.3.1.4. Comunicaciones

Las comunicaciones son especialmente importantes en aquellas herramientas diseñadas para su ejecución en sistemas de memoria distribuida, como en este caso. Con el objetivo de minimizar el tiempo de ejecución, se busca que los diferentes procesos, ubicados en los nodos que componen el sistema de memoria distribuida, intercambien los mensajes necesarios de una forma eficiente.

Por una parte, para llevar a cabo las comunicaciones de tipo global (aquellas que involucran a muchos procesos) en este algoritmo, se ha recurrido a las primitivas para comunicaciones colectivas que proporciona MPI [98], ya que son capaces de utilizar algoritmos óptimos de encaminamiento teniendo en cuenta la red de interconexión.

Por otra parte, en el caso de las comunicaciones de tipo local (involucran a pocos procesos) se ha desarrollado un esquema de comunicaciones específico, utilizando para ello primitivas para comunicaciones punto a punto sin bloqueo. Este esquema de comunicaciones se analiza en detalle en el subapartado 2.3.4. Además, es necesario tener en cuenta que, debido al uso combinado de MPI y OpenMP, las comunicaciones pueden desarrollarse de formas sustancialmente diferentes. Los tipos de comunicaciones más habituales en implementaciones híbridas MPI + OpenMP son los siguientes [133]:

- *Masteronly.* Con esta técnica, las comunicaciones de tipo MPI sólo tienen lugar fuera de las regiones paralelas definidas con directivas OpenMP. Por tanto, sólo un hilo por proceso se encarga de las comunicaciones. Este es el tipo de comunicaciones por defecto, por lo que todas las implementaciones de MPI deben admitirlo.
- De tipo embudo (*funneled*). En este caso, las comunicaciones MPI tienen lugar dentro de regiones paralelas definidas mediante OpenMP, pero sólo un hilo (el denominado hilo maestro) se encarga de dichas comunicaciones. Este tipo de comunicaciones se incorporó en el estándar MPI-2 [98] y sólo está permitido en aquellas implementaciones de MPI de tipo *thread-safe*.

- Por lotes (*serialized*). En este caso, las comunicaciones MPI también tienen lugar dentro de las regiones paralelas de OpenMP, pero sólo un hilo puede estar enviando mensajes en un momento dado. Este tipo de comunicaciones también se incorporó en el estándar MPI-2 [98] y sólo se admite en las implementaciones de MPI de tipo *thread-safe*.
- Múltiple. Con este modelo, todos los hilos de las regiones paralelas definidas mediante OpenMP pueden llevar a cabo comunicaciones sin restricciones usando MPI. Al igual que en los dos casos anteriores, este tipo de comunicaciones se incorporó en el estándar MPI-2 [98], pero sólo está permitido en aquellas implementaciones de MPI de tipo *full thread-safe*.

Por ello, es necesario decidir cómo se van a llevar a cabo las comunicaciones teniendo en cuenta las decisiones de diseño previas. Así, puesto que los modelos de comunicaciones en los que un único hilo lleva a cabo las comunicaciones resultan más eficientes para mensajes de tamaño superior a 72 KB [134] y que, además, la traslación y el cálculo de las interacciones cercanas pueden generar mensajes MPI de forma simultánea, la técnica más adecuada para los problemas que se pretende resolver con esta herramienta parece ser la comunicación por lotes o *serialized*.

2.3.1.5. Balanceo de carga

Para poder lograr un reparto equilibrado de la carga de trabajo asumida por las diversas CPU y GPU, resulta conveniente tener la capacidad de estimar a priori y de forma precisa el tiempo por iteración. Para ello, se utiliza la expresión que se muestra a continuación:

$$T_{it} \approx T_a + C_a + \text{máx}\{T_c, T_t\} + C_c + C_t + T_d + C_d, \qquad (2.11)$$

donde T_x representa el tiempo dedicado al cálculo y C_x representa el tiempo requerido por las comunicaciones, en ambos casos para cada etapa x (interacciones cercanas, agregación, traslación o desagregación).

Puede observarse que la ecuación (2.11) también refleja la ejecución concurrente de la traslación y del cálculo de las interacciones cercanas,

de forma que el tiempo empleado para llevar a cabo las dos etapas viene fijado por la etapa que consume un mayor tiempo ($\max\{T_c, T_t\}$). Como consecuencia de la aplicación de esta estrategia, para lograr un equilibrio de carga entre CPU y GPU, se debe seleccionar un tamaño adecuado para los grupos. Si el tamaño del grupo es grande, las etapas asignadas a las GPU tienen mucho peso, mientras que el peso de la traslación (asociada a las CPU) decrece. Por contra, si el tamaño de grupo es pequeño, la traslación aumenta su peso computacional, mientras que las GPU pierden carga de trabajo. Puesto que, para un tamaño de grupo dado, la agregación y la desagregación tienen un coste temporal mucho menor que las interacciones cercanas, un equilibrio adecuado entre el peso de la traslación y el peso de las interacciones cercanas produce un balanceo de carga equitativo entre las diversas CPU y GPU.

Además, merece la pena precisar que en la ecuación (2.11), por simplicidad, se asume el peor caso, que sería aquel en el que no hay ningún tipo de solapamiento entre el cálculo concurrente de la traslación y las interacciones cercanas, y las comunicaciones asociadas a dichas etapas.

Finalmente, para lograr un reparto que permita equilibrar la carga de trabajo de los distintos procesos paralelos, se ha diseñado una estrategia de balanceo de carga a varios niveles. Por una parte, se tiene un reparto estático del trabajo a nivel de proceso que se basa en el cálculo del tiempo de ejecución usando la expresión (2.11). Con este modelado de la computación y de las comunicaciones, se estima a priori y de forma precisa el tiempo por iteración, por lo que el balanceo de carga a nivel de proceso se puede llevar a cabo ya en la fase de inicialización. Por otra parte, para las rutinas ejecutadas en CPU, se dispone del balanceo de carga a nivel de hilo que actúa mediante las planificaciones dinámicas disponibles en OpenMP [96]. Y, por último, para los *kernels* ejecutados en GPU, la propia gestión de los *warps* que componen los bloques de hilos CUDA permite un balanceo de carga entre los diferentes multiprocesadores que componen una GPU [112, 111].

2.3.2. Inicialización

Dentro de la fase de inicialización del algoritmo, hay varias operaciones que se ven claramente influenciadas por las decisiones de diseño que se han tomado para llevar a cabo la paralelización heterogénea del FMM: la selección del tamaño de los grupos, el reparto de la carga de trabajo entre los diferentes procesos paralelos, la distribución inicial de datos, y el cálculo del operador de traslación.

2.3.2.1. Selección del tamaño de grupo

En la implementación heterogénea del FMM presentada en esta Tesis, el tamaño de grupo se selecciona utilizando el tiempo por iteración como criterio de decisión [13]. Para ello, se hace una búsqueda exhaustiva (pero acotada) hasta encontrar el tamaño de grupo que minimiza el coste de la expresión (2.11).

Como se comentó en el punto 2.3.1.5, si el tamaño de los grupos es grande, las etapas asignadas a las GPU (interacciones cercanas, agregación y desagregación) tienen mucho peso, mientras que el trabajo asociado a las CPU decrece. Sin embargo, si el tamaño de los grupos es pequeño, la traslación (asignada a las CPU) aumenta su peso computacional, mientras que las GPU pierden carga de trabajo.

Para llevar a cabo la selección del tamaño de grupo óptimo, se modelan los tiempos de cada etapa del FMM teniendo en cuenta tanto el tiempo invertido en la computación como el tiempo dedicado a las comunicaciones. El cálculo de cada uno de los sumandos de la expresión (2.11) se detallará más adelante, en los subapartados dedicados a estudiar de forma individualizada las interacciones cercanas, la agregación, la traslación y la desagregación.

2.3.2.2. Reparto de la carga de trabajo

Una vez definido el tamaño de los grupos, la inicialización continúa con el siguiente paso, el algoritmo de balanceo de carga. Para lograr una

carga de trabajo equilibrada entre los diferentes procesos paralelos, se debe tener en cuenta el particionado diseñado, analizado previamente en el punto 2.3.1.3.

El algoritmo ideado se encarga de repartir tanto las operaciones de cálculo asociadas a cada grupo (en el caso de interacciones cercanas, agregación y desagregación) como el trabajo vinculado a cada dirección del espacio $\boldsymbol{\kappa}$ (en el caso del cálculo del operador de traslación y de la traslación).

Para las etapas con un particionado por direcciones, el algoritmo reparte entre los diferentes procesos MPI direcciones consecutivas, utilizando para ello una división euclídea o división entera. Por tanto, si se tienen k_l direcciones del espacio $\boldsymbol{\kappa}$ y p procesos y, además, tenemos que $r = k_l \,\%\, p$, entonces el algoritmo resuelve que los primeros r procesos trabajan con $\lceil k_l/p \rceil$ direcciones y los $p-r$ procesos restantes trabajan con $\lfloor k_l/p \rfloor$ direcciones (donde $\lceil \cdot \rceil$ y $\lfloor \cdot \rfloor$ representan la función techo y suelo, respectivamente).

Para las etapas que se han diseñado siguiendo un particionado por grupos, el algoritmo reparte grupos consecutivos, teniendo en cuenta la carga de trabajo que representa cada grupo. Además, merece la pena destacar que las etapas se tratan de forma independiente, de forma que cada proceso puede trabajar con distintos grupos para el cálculo de las interacciones cercanas y para la agregación-desagregación, siempre con el objetivo de lograr una carga computacional lo más homogénea posible.

2.3.2.3. Distribución inicial de datos

Tras fijar el reparto de trabajo entre los diferentes procesos, tienen lugar las comunicaciones destinadas a la distribución inicial de los datos del problema a resolver.

Con los grupos para cada proceso ya definidos, el proceso raíz (habitualmente el proceso con rango 0) se encarga de distribuir los datos necesarios entre todos los procesos, principalmente datos relacionados con la geometría del objeto dispersor. Para ello, se utiliza una comunicación MPI de índole

global, con estructura de árbol, de tipo *scatter*. En la figura 2.6, se muestra de forma esquemática la estructura de este tipo de comunicaciones.

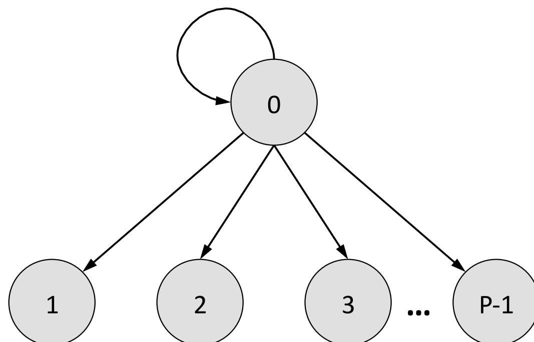


Figura 2.6: Representación esquemática de las comunicaciones de tipo *scatter*.

2.3.2.4. Operador de traslación

Dentro de la fase de inicialización del algoritmo, una de las tareas computacionalmente más costosas es el cálculo del operador de traslación, $\mathcal{T}$. Aunque esta rutina se invoca únicamente una vez en el ciclo de ejecución del algoritmo, resulta especialmente importante por su peso en el consumo total de memoria. La estrategia utilizada en el cálculo del operador de traslación usando MPI y OpenMP se muestra a continuación, en el algoritmo 2.1.

En el pseudocódigo 2.1, *ncd* es un contador que acumula el número de combinaciones diferentes de centros de grupos, entendidas como las direcciones espaciales que unen los centros de dos grupos lejanos. g' y g son los índices utilizados para identificar el grupo fuente y el grupo de observación, respectivamente. Por su parte, $CombCen$ es un array auxiliar que contiene las correspondencias entre las posiciones de los centros de los grupos y las posiciones de los datos dentro del operador de traslación. El uso de este array auxiliar es imprescindible puesto que, en la implementación del FMM de esta Tesis, el operador de traslación se almacena de forma compactada con el objetivo de evitar las combinaciones de centros redundantes [13].

Algoritmo 2.1 Cálculo del operador de traslación usando MPI y OpenMP.

Entrada: $MiRango$, h, $PrimeraDir$, $UltimaDir$, N_g, $DatosGeo_p$, L, $CombCen$ (inicializado a -1).

Salida: $\mathcal{T}_p$, $CombCen$ (actualizado).

1: $ncd = 0$
2: $ndir = UltimaDir[MiRango] - PrimeraDir[MiRango] + 1$
3: {Para cada grupo de la geometría}
4: **para** $g = 0$ **hasta** $N_g - 1$ **hacer**
5: {Para cada grupo de la geometría}
6: **para** $g' = 0$ **hasta** $N_g - 1$ **hacer**
7: {Si la combinación de centros para los grupos g-g' es nueva}
8: **si** $IndCombCentros(g, g', DatosGeo_p, CombCen) == -1$ **entonces**
9: {Para cada orden de la expansión}
10: #pragma omp parallel for num_threads(h)
11: **para** $l = 0$ **hasta** L **hacer**
12: $b = ncd \cdot L + l$
13: $beta[b] = Hankel(l, DatosGeo_p, g, g') \cdot (2l + 1) \cdot j^l$
14: **fin para**
15: {Para las direcciones del espacio $\boldsymbol{\kappa}$ de cada proceso}
16: #pragma omp parallel for num_threads(h)
17: **para** $k = PrimeraDir[MiRango]$ **hasta** $UltimaDir[MiRango]$ **hacer**
18: $w_p = ncd \cdot ndir + k$
19: $\mathcal{T}_p[w_p] = CalculaPolLeg(L, beta, DatosGeo_p)$
20: **fin para**
21: $ncd = ncd + 1$
22: $ActualizarCombCentros(g, g', DatosGeo_p, CombCen, ncd)$
23: **fin si**
24: **fin para**
25: **fin para**

Previamente a la ejecución de esta rutina, $CombCen$ se inicializa con todas sus posiciones a un valor de -1. De esta forma, se puede detectar qué combinaciones de centros son nuevas, puesto que una combinación que ya ha sido tratada con anterioridad contiene un valor mayor o igual que 0 que representa una posición dentro de $\mathcal{T}_p$. Además, tras la ejecución de esta rutina, $CombCen$ también permitirá conocer si dos grupos son lejanos en-

tre sí, ya que las combinaciones de centros que se corresponden con grupos cercanos mantienen el valor de inicialización (-1). El array utilizado para almacenar el operador de traslación contiene el subíndice p en su nombre debido a que no alberga todo el conjunto de datos, sino que contiene únicamente la información parcial que necesita cada proceso. De igual manera, para acceder al array $\mathcal{T}_p$ se debe utilizar un índice apropiado (en este caso, w_p).

L se corresponde con el mayor orden de la expansión, mientras que el índice l permite iterar por cada orden de la expansión. Por su parte, la función $Hankel$ permite calcular la función de Hankel esférica de segunda especie y orden l, que se utiliza para obtener los valores almacenados en el array *beta*. En la función auxiliar $CalculaPolLeg$ se calcula el polinomio de Legendre y se multiplica por *beta* para obtener cada una de las combinaciones no redundantes del operador de traslación ($\mathcal{T}_p$). Finalmente, el array $CombCen$ se actualiza para que contenga la correspondencia entre la combinación de los centros de los grupos y la posición de los datos dentro del operador de traslación.

En esta rutina se ha utilizado un particionado de grano grueso basado en direcciones, presente en el bucle de la línea 17 del algoritmo 2.1, que itera por las distintas direcciones del espacio $\boldsymbol{\kappa}$. Este primer nivel del particionado es el resultado del algoritmo de balanceo de carga para la etapa de traslación. Además, hay un segundo nivel de particionado, reflejado en las líneas 10 y 16, que permite dividir de forma aún más fina, usando h hilos OpenMP, la carga de trabajo asociada a cada proceso MPI. La gran ventaja de utilizar un particionado por direcciones para la traslación y, por extensión, para el cálculo de $\mathcal{T}$, es que se evita cualquier tipo de replicación del operador de traslación. Cada proceso calcula y almacena únicamente la parte del operador que se corresponde con las direcciones que el algoritmo de balanceo de carga le ha asignado. De esta forma, el algoritmo logra una gran escalabilidad, permitiendo resolver problemas de mayor tamaño al aumentar los recursos computacionales, a diferencia de lo que ocurre usando otro tipo de particionado como, por ejemplo, el particionado por grupos, donde la replicación del operador de traslación es prácticamente total (el consumo de memoria por proceso no disminuye al aumentar el número de procesos).

Una vez que todos los procesos finalizan el cálculo del operador de traslación, no se lleva a cabo ningún tipo de comunicación. Puesto que cada proceso ya ha calculado y almacenado la parte del operador que debe utilizar en la traslación, no se necesita realizar ningún intercambio de datos.

2.3.3. Interacciones cercanas

El cálculo de las interacciones entre elementos cercanos es el paso más costoso del FMM desde el punto de vista computacional, con una alta intensidad computacional en relación al número de elementos tratados (bajo ratio bytes/flop). Para explicar las técnicas utilizadas en el cálculo de las interacciones cercanas usando MPI y CUDA, se muestra el pseudocódigo correspondiente al algoritmo 2.2.

Algoritmo 2.2 Interacciones cercanas usando MPI y CUDA.

Entrada: $MiRango$, $dimBloque$, $dimGrid$, $PrimerGrupo_c$, $UltimoGrupo_c$, $NumElementosCercanos$, $DatosCercanos_p$.

Salida: $Cercanas$ contiene las interacciones cercanas de cada elemento.

1: {**Cálculo**}
2: $miPrimerElemento = PrimerElemento(PrimerGrupo_c[MiRango]) + IdBloque \cdot dimBloque + IdHilo$
3: $miUltimoElemento = UltimoElemento(UltimoGrupo_c[MiRango])$
4: $salto = dimBloque \cdot dimGrid$
5: {Para cada elemento de la geometría a tratar por $MiRango$}
6: **para** $e = miPrimerElemento$ **hasta** $miUltimoElemento$ **paso** $salto$ **hacer**
7: $e_p = e - PrimerElemento(PrimerGrupo_c[MiRango])$
8: $g_p = IndiceGrupo(e) - PrimerGrupo_c[MiRango]$
9: $Cercanas_p[e_p] = ImpedanciaMutua(e_p, NumElementosCercanos[g], DatosCercanos_p[g_p])$
10: **fin para**
11: {**Comunicaciones**}
12: $Gather(Cercanas_p, Cercanas, 0)$

Al existir varios procesos paralelos (tantos como procesadores gráficos utilizados), no hay un único proceso encargado de iterar desde el primer hasta el último grupo, sino que los distintos procesos trabajan con un sub-

conjunto de grupos consecutivos. La variable $MiRango$ contiene el rango o identificador de cada proceso, el cual varía entre 0 y $p - 1$.

Por su parte, *dimBloque* representa el tamaño de los bloques CUDA (número de hilos por bloque), mientras que *dimGrid* representa el tamaño del *grid*, es decir, el número total de bloques. $IdBloque$ e $IdHilo$ son variables privadas para cada hilo que le permiten identificar su bloque y su número de hilo dentro del bloque, respectivamente. Con la variable *salto*, se define la longitud del paso entre iteraciones para el bucle de la línea 4, puesto que podría ser necesario realizar más de una iteración si el número de elementos a tratar por una GPU fuese mayor que el número total de hilos CUDA generados ($dimBloque \cdot dimGrid$). Por tanto, los cálculos se realizan elemento a elemento de forma cíclica, de tal forma que cada hilo activo trabaja con, al menos, un elemento (balanceo de carga a nivel de hilos CUDA). Este particionado de grano fino busca minimizar el número de hilos ociosos para obtener un alto rendimiento mediante el uso eficaz del paralelismo ofrecido por los procesadores gráficos.

Los arrays $PrimerGrupo_c$ y $UltimoGrupo_c$ definen el intervalo de grupos con los que debe trabajar cada GPU en la etapa del cálculo de las interacciones cercanas. El intervalo de grupos para cada proceso es el resultado de las decisiones tomadas por el algoritmo de balanceo de carga en la fase de inicialización (balanceo estático a nivel de proceso MPI). Por su parte, los arrays cuyo nombre contiene el subíndice p (p. ej. $DatosCercanos_p$) no contienen todo el conjunto de datos, sino que albergan únicamente la información parcial que necesita cada proceso, por lo que se accede a ellos usando los índices apropiados (p. ej. g_p). Dichos arrays se crean con la información recibida en las comunicaciones de tipo *scatter* (véase la figura 2.6) que se llevan a cabo en la fase de inicialización. La función auxiliar $ImpedanciaMutua$ (de tipo `__device__`) se encarga de realizar los cálculos necesarios para obtener las interacciones cercanas de cada elemento de forma individual, almacenando estas contribuciones en el array $Cercanas_p$. Además, conviene mencionar que los datos relativos a los elementos cercanos a cada grupo ($DatosCercanos_p$) se empaquetan previamente en la CPU y, después, se transfieren a la GPU inmediatamente antes de la ejecución del *kernel*. De esta forma, la GPU sólo necesita ocuparse de la parte intensiva en cálculo. A su vez, el array $Cercanas_p$ es transferido del dis-

positivo al anfitrión, una vez que la GPU finaliza con todos los cálculos, inmediatamente después de la ejecución del *kernel*.

Asimismo, es destacable la funcionalidad implementada que permite resolver problemas cuyo tamaño es superior a la memoria del dispositivo. En dichos casos, el bucle principal (línea 4 del algoritmo 2.2) se divide de tal forma que se trabaja con bloques de elementos contiguos hasta que se procesan todos los elementos. Esta ejecución *por lotes* de una simulación permite abordar un mayor número de problemas que de otra forma no cabrían en la memoria global del dispositivo.

Una vez que todos los procesos completan el cálculo de las contribuciones correspondientes a los elementos de sus grupos, comienza un período de comunicaciones. Esta comunicación tiene como objetivo que el proceso raíz reúna los datos calculados por los diferentes procesos, para así componer el array *Cercanas* que ha de contener las interacciones cercanas de todos los elementos. Para ello, se utiliza una comunicación de tipo global con estructura de árbol en la que la información fluye de las hojas hacia la raíz. Este esquema de comunicaciones, que se conoce como *gathering* o reunión, se muestra de forma esquemática en la figura 2.7.

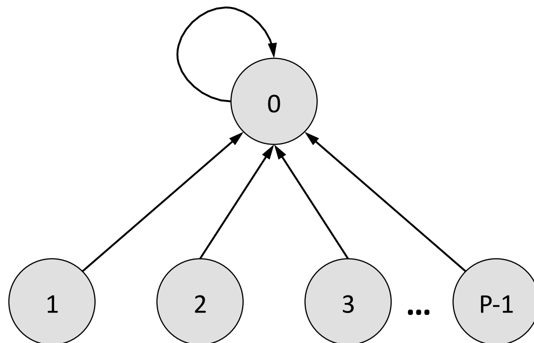


Figura 2.7: Representación esquemática de las comunicaciones de tipo *gather*.

Una vez analizado el pseudocódigo correspondiente, se pueden modelar las partes de la ecuación (2.11) relativas a las interacciones cercanas para

un problema de tamaño N que se resuelve usando p procesos:

$$\begin{aligned} T_c &\approx F_c \cdot n_{pi} \cdot \frac{\sum_{i=0}^{N_g-1} n_i \cdot n_i^{(c)}}{p} \\ &+ \beta' \cdot \frac{B_z \cdot N + \sum_{i=0}^{N_g-1} B_{geo} \cdot n_i^{(c)}}{p}, \end{aligned} \tag{2.12a}$$

$$C_c \approx \beta \cdot \frac{N}{p}, \tag{2.12b}$$

donde p es el número de procesos paralelos que, como ya se ha indicado, coincide con el número de procesadores gráficos (GPU). Por su parte, F_c es el factor ajustable que permite modelar el tiempo por operación para esta etapa. Asimismo, β' es el factor que permite modelar el tiempo necesario para enviar un byte por el bus de tipo *PCI Express* que conecta CPU y GPU. Además, se utilizan dos factores que permiten indicar el número de bytes por elemento enviado: B_z y B_{geo}. B_z indica el número de bytes que ocupa un elemento de tipo complejo (4 bytes en el caso de simple precisión). B_{geo} representa el número de bytes que se necesitan para identificar un elemento de la geometría, es decir, el número de bytes que ocupa la estructura encargada de almacenar las coordenadas del centro del elemento, las coordenadas de sus vértices, su vector normal, su área y, por simplificar la expresión, también se incluye la componente del iterando en dicho elemento. De esta manera, $\left(\sum_{i=0}^{N_g-1} B_{geo} \cdot n_i^{(c)}\right)/p$ representa el número de bytes enviados en sentido CPU-GPU (datos relacionados con la geometría), mientras que $(B_z \cdot N)/p$ representa el número de bytes enviados en sentido GPU-CPU (resultados parciales de las interacciones cercanas).

Al igual que sucede con las transferencias de datos por el bus *PCI Express* mencionadas en el párrafo anterior, para calcular el tiempo consumido por las comunicaciones de tipo MPI tampoco es suficiente con conocer el tamaño de los mensajes sino que también es necesario disponer de un valor que refleje la capacidad de la red de interconexión. Para ello, en la ecuación (2.12b) se utiliza el factor ajustable β que permite introducir el tiempo necesario para enviar un elemento. Este factor debe estimarse utilizando datos empíricos, ya que depende de la red de interconexión, del *middleware* (implementación de MPI), del sistema operativo, etc.

2.3.4. Agregación

La estrategia utilizada para llevar a cabo el cálculo de las contribuciones agregadas usando MPI y CUDA se muestra en el algoritmo 2.3. Con el objetivo de facilitar la comprensión del pseudocódigo, las variables comunes con el algoritmo 2.2 conservan el mismo nombre y representan la misma información.

Algoritmo 2.3 Agregación usando MPI y CUDA.

Entrada: $MiRango$, $dimBloque$, $dimGrid$, $PrimerGrupo_a$, $UltimoGrupo_a$, k_l, $NumElementos$, n_{pi}, $DatosGeo_p$, $Iterando_p$, $Agreg_p$ (inicializado a 0).

Salida: $AgregT_p$ contiene las contribuciones requeridas por el proceso $MiRango$ para la traslación.

```
1: {Cálculo}
2: miPrimeraDir = PrimerGrupo_a[MiRango] · k_l
       + IdBloque · dimBloque + IdHilo
3: miUltimaDir = UltimoGrupo_a[MiRango] · k_l − 1
4: salto = dimBloque · dimGrid
5: {Para cada dirección del espacio κ de los grupos asignados}
6: para a = miPrimeraDir hasta miUltimaDir paso salto hacer
7:    g = a / k_l
8:    k = a % k_l
9:    a_p = (g − PrimerGrupo_a[MiRango]) · k_l + k
10:   {Para cada elemento perteneciente al grupo actual}
11:   para i = 0 hasta NumElementos[g] − 1 hacer
12:      e_p = PrimerElemento(g) + i
            − PrimerElemento(PrimerGrupo_a[MiRango])
13:      {Para cada punto de integración}
14:      para pto = 0 hasta n_pi − 1 hacer
15:         DatosInt = CalculaPtosIntegracion(e_p, DatosGeo_p, pto)
16:         Agreg_p[a_p] = Agreg_p[a_p] + ContribAg(e_p, g, pto, k,
               DatosGeo_p, DatosInt, Iterando_p[e_p])
17:      fin para
18:   fin para
19: fin para
20: {Comunicaciones}
21: EnvioRecepcion(Agreg_p, AgregT_p)
```

Al igual que en el cálculo de las interacciones cercanas, los diferentes procesos MPI paralelos trabajan con un subconjunto de grupos consecutivos de forma que, entre todos los procesos, se trata el total de los N_g grupos. Los arrays $PrimerGrupo_a$ y $UltimoGrupo_a$, que son generados por el algoritmo de balanceo de carga en la fase de inicialización, definen el intervalo de grupos con los que debe trabajar cada proceso. En este punto, resulta necesario recalcar que el reparto de grupos para la agregación no tiene por qué coincidir con el reparto para el cálculo de las interacciones cercanas, ya que se busca el balanceo de carga óptimo de forma independiente para cada etapa del FMM.

En la línea 6 del algoritmo 2.3, puede verse el resultado del reparto de carga de trabajo a nivel de proceso (grano grueso) y a nivel de hilo CUDA (grano fino). Para este *kernel*, el particionado también es de grano fino, ya que se trabaja dirección a dirección de forma cíclica, lo que permite minimizar el número de hilos ociosos. Al iterar dirección a dirección, se necesita obtener el número de grupo y el número de dirección del espacio $\boldsymbol{\kappa}$ dentro de ese grupo. Para ello, basta con realizar una división entera con a como dividendo y k_l como divisor, obteniendo así el número de grupo (g). Por su parte, el resto (k) indica el número de dirección dentro del grupo. Es decir, el índice a permite representar la dirección k-ésima para el grupo g. A su vez, a_p permite representar la dirección k-ésima para el grupo relativo actual (contando a partir del primer grupo del reparto para el proceso $MiRango$), mientras que e_p indica el elemento relativo actual. n_{pi} representa el número de puntos de integración utilizados para la cuadratura gaussiana, y *pto* es el índice que marca el punto de integración actual. Por su parte, $DatosInt$ almacena la información relativa a los puntos de integración, la cual se obtiene mediante la rutina de tipo `__device__` $CalculaPtosIntegracion$. Por último, la función $ContribAg$ (también de tipo `__device__`) engloba los cálculos necesarios para obtener una nueva contribución, que se acumula en el array $Agreg_p$.

Tanto los datos relativos a la geometría de los grupos asignados como la parte del iterando correspondiente a cada proceso ($DatosGeo_p$ e $Iterando_p$), se transfieren a la GPU inmediatamente antes de la ejecución del *kernel*. Además, como se explicó anteriormente para el caso del *kernel* dedicado a las interacciones cercanas, en esta ocasión también se mantiene

la capacidad para resolver problemas cuyo tamaño es superior a la memoria del dispositivo, dividiendo el bucle principal (línea 6 del algoritmo 2.3) para trabajar con bloques de direcciones consecutivas. Una vez que la GPU finaliza con todos los cálculos relativos a la agregación, el array $Agreg_p$ es transferido del dispositivo al anfitrión para poder llevar a cabo las comunicaciones MPI previas a la ejecución de la traslación en la CPU.

El diseño de esta función presenta diferencias muy notables respecto de su equivalente para CPU [13]. En CPU, habitualmente, se trabaja grupo a grupo y elemento a elemento, mientras que en este caso (sistemas heterogéneos de memoria distribuida) se ha decidido trabajar dirección a dirección. De esta forma, se consigue un objetivo importante para lograr una implementación eficiente sobre procesadores gráficos, evitar las *reducciones* [135] innecesarias. Si, por el contrario, se hubiese optado por trabajar elemento a elemento, cada posición del array $Agreg_p$ sería actualizada por más de un hilo a la vez, lo que implicaría el uso de operaciones atómicas o de reducciones para garantizar la corrección de los resultados [112, 135]. El diseño presentado en esta Tesis resulta más eficiente, ya que está enfocado específicamente para la ejecución sobre GPU, evitando sincronizaciones innecesarias y el uso de memoria compartida como almacenamiento privado para cada hilo.

Una vez que se completa el cálculo de las agregaciones asignadas a cada proceso, comienza un período de comunicaciones en el que se busca satisfacer varios requisitos para lograr una solución escalable:

- Intercambiar los datos necesarios para comenzar con la etapa de traslación.
- Conseguir que ningún proceso almacene las contribuciones agregadas al completo.
- Minimizar la cantidad de información enviada por la red de interconexión.
- Permitir el solapamiento de las comunicaciones y la generación del nuevo array $AgregT_p$.

El primero de los requisitos hace referencia al cambio de particionado que se produce en la traslación. En la agregación, cada proceso trabaja con todas las direcciones del espacio $\boldsymbol{\kappa}$ para un subconjunto de grupos, mientras que en la traslación cada proceso debe trabajar con un subconjunto de direcciones para todos los grupos. Por tanto, cada proceso necesita intercambiar información con el resto de procesos para poder disponer de los datos necesarios para llevar a cabo su parte de la traslación.

El segundo requisito se debe al interés en mantener un perfil bajo en el consumo de memoria. Si cada proceso enviase a la vez su array $Agreg_p$ al resto de procesos entonces todos deberían almacenar un array con todas las contribuciones agregadas. Pero esto no es necesario (por lo explicado en el párrafo anterior) y además limitaría de manera importante la escalabilidad de la implementación. Por tanto, lo ideal es que los procesos envíen sus contribuciones de forma ordenada para que cada destinatario pueda recibir únicamente los datos necesarios evitando el uso de un búfer de recepción de gran tamaño.

El tercero de los requisitos busca reducir el tiempo dedicado a las comunicaciones minimizando los datos transferidos. Para ello, es necesario que cada proceso envíe al resto sólo los datos necesarios, por lo que la selección de la información a enviar la realiza el propio emisor. Esto implica que todos los procesos deben conocer qué direcciones trata el resto de procesos en la etapa de traslación. Pero no supone ningún problema, ya que todos los procesos tienen acceso a los arrays relativos al balanceo de carga y además su almacenamiento tiene un peso despreciable ($\mathcal{O}(p)$).

El último requisito tiene como objetivo lograr unas comunicaciones eficientes, de forma que los procesos no permanezcan ociosos durante el tiempo de transferencia de los datos por la red de comunicaciones. Para ello, el esquema de comunicaciones empleado debe utilizar rutinas MPI no bloqueantes.

Una vez analizados los requisitos para lograr una solución escalable en las comunicaciones llevadas a cabo tras el cálculo de las contribuciones agregadas, se opta por crear un esquema específico para este problema. Para explicar de forma sencilla cómo se llevan a cabo las comunicaciones en esta etapa, se muestra un ejemplo con 5 procesos en la figura 2.8.

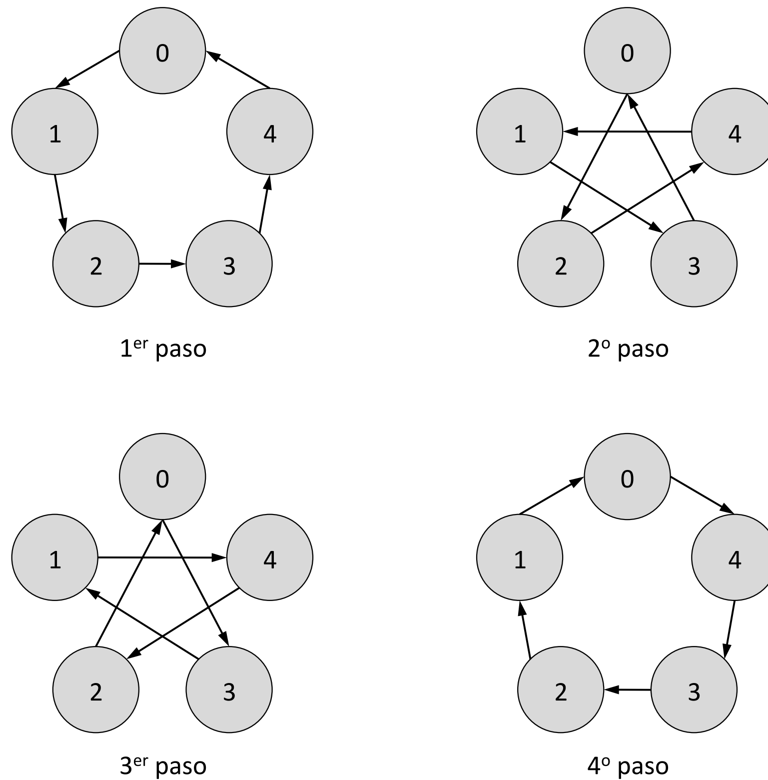


Figura 2.8: Representación de las comunicaciones de tipo envío-recepción (punto a punto) para un ejemplo con 5 procesos (comunicación en 4 pasos).

Como se muestra en la figura 2.8, el esquema de comunicaciones ideado se desarrolla en $p-1$ pasos. En el primer paso, cada proceso le envía la información estrictamente necesaria al proceso con el rango inmediatamente superior de forma cíclica (el siguiente al último es el primero). En el segundo paso, cada proceso le envía información al proceso cuyo rango es igual al suyo más dos, también en sentido cíclico. Y así sucesivamente hasta que, en el último paso, cada proceso le envía información al proceso con el rango inmediatamente inferior de forma cíclica (el anterior al primero es el último). Así, paso a paso, cada proceso va recibiendo únicamente la información relativa a sus direcciones del espacio $\boldsymbol{\kappa}$ (aquellas asignadas por el algoritmo de balanceo de carga) para todos los grupos. De esta forma,

se logra que cada proceso reciba sólo la información relacionada con las direcciones con las que va a trabajar en la etapa de traslación. Si bien el funcionamiento de este esquema de comunicaciones es muy similar al que proporciona MPI con la rutina `MPI_Alltoallw` [136], presenta dos ventajas interesantes desde el punto de vista de la programación: sencillez (ya que se evita el uso de tipos de datos derivados complicados) y capacidad de solapar las comunicaciones (no bloqueantes) con la gestión de los datos recibidos.

Una vez analizada esta etapa, se pueden modelar los sumandos de la ecuación (2.11) correspondientes con la agregación. Nuevamente, se tiene en cuenta tanto el número de operaciones asociadas a cada proceso como el peso de los mensajes intercambiados en las comunicaciones:

$$T_a \approx F_a \cdot n_{pi} \cdot \frac{k_l \cdot N}{p} + \beta' \cdot \frac{B_{geo} \cdot N + B_z \cdot k_l \cdot N_g}{p}, \tag{2.13a}$$

$$C_a \approx \beta \cdot (p-1) \cdot \frac{k_l \cdot N_g}{p^2}, \tag{2.13b}$$

donde F_a es el factor estimado empíricamente que permite modelar el tiempo por operación para esta etapa. De los bytes transferidos entre las CPU y sus GPU asociadas, $(B_{geo} \cdot N)/p$ se corresponden con los datos relativos a la geometría del objeto dispersor, en sentido CPU-GPU, y $(B_z \cdot k_l \cdot N_g)/p$ se corresponden con el array $Agreg_p$, en sentido GPU-CPU.

Por su parte, en la ecuación (2.13b) se muestra el tiempo dedicado a las comunicaciones de tipo MPI presentadas en la figura 2.8. Merece la pena destacar la escalabilidad del esquema de comunicaciones ideado, en el que se reduce el peso de las comunicaciones al incrementar el número de procesos (p).

2.3.5. Traslación

La etapa de traslación es el segundo paso más costoso desde el punto de vista computacional. Asimismo, es una rutina muy intensiva en memoria, que accede a la estructura de datos de mayor tamaño del algoritmo, el operador de traslación. Además, la lectura del operador de traslación no se

realiza de forma secuencial, ya que el operador se genera de forma compacta evitando redundancias, lo que provoca accesos a memoria poco regulares y numerosos fallos de caché.

Como ya se comentó en el apartado dedicado al particionado, esta etapa se ejecuta sólo en las CPU mientras las GPU trabajan en el cálculo de la interacciones cercanas. Por ello, resulta interesante explicar la técnica utilizada para llevar a cabo de forma concurrente el cálculo de las interacciones cercanas y la traslación mediante la descomposición funcional representada en la figura 2.4(b). Para simultanear el cálculo de las interacciones cercanas y el cálculo de la traslación, se utiliza la construcción `sections` de OpenMP. Se definen dos secciones paralelas, una para la traslación y otra para las interacciones cercanas, que son ejecutadas en paralelo usando un hilo diferente para cada una de ellas.

Sin embargo, las secciones OpenMP, por sí mismas, no son suficientes para ejecutar de forma concurrente y paralela la traslación, y el objetivo final no es trabajar con un único hilo. Por tanto, una vez que se lleva a cabo la descomposición funcional mencionada en el párrafo anterior, es necesario disponer de suficientes hilos para poder abordar la traslación usando todos los núcleos de las CPU empleadas. Para ello, se debe habilitar el uso del paralelismo anidado mediante la instrucción `omp_set_nested(1)`. De esta forma, cada proceso MPI puede llevar a cabo la traslación de forma paralela (como se muestra en el algoritmo 2.4) y, además, se dispone de un hilo independiente que controla el acelerador gráfico encargado del cálculo de las interacciones cercanas.

g' y g son índices que se utilizan para iterar por los grupos fuente (origen) y los grupos de observación (destino) de la traslación, respectivamente. $CombCen$ es un array auxiliar que contiene las correspondencias entre las posiciones de los centros de los grupos y las posiciones de los datos dentro del operador de traslación. Este array es necesario puesto que, como se ha apuntado anteriormente, el operador de traslación ($\mathcal{T}$) se almacena compactado para evitar combinaciones de centros redundantes. Además, $CombCen$ también permite detectar qué grupos son lejanos, ya que las combinaciones de centros que se corresponden con grupos cercanos se mantienen con el valor inicial (-1). En la rutina auxiliar $IndCombCentros$, y a partir de los

Algoritmo 2.4 Traslación usando MPI y OpenMP.

Entrada: $MiRango$, h, $PrimeraDir$, $UltimaDir$, N_g, $DatosGeo_p$, $CombCen$, $AgregT_p$, $\mathcal{T}_p$, $Tras_p$ (inicializado a 0).

Salida: $TrasD_p$ contiene las contribuciones requeridas por el proceso $MiRango$ para la desagregación.

```
1: {Cálculo}
2: ndir = UltimaDir[MiRango] − PrimeraDir[MiRango] + 1
3: {Para cada grupo de la geometría}
4: #pragma omp parallel for num_threads(h)
5: para g = 0 hasta N_g − 1 hacer
6:    {Para cada grupo de la geometría}
7:    para g' = 0 hasta N_g − 1 hacer
8:       cc = IndCombCentros(g, g', DatosGeo_p, CombCen)
9:       {Si g y g' son grupos lejanos}
10:      si cc ≥ 0 entonces
11:         {Para las direcciones del espacio κ de cada proceso}
12:         para k = 0 hasta ndir − 1 hacer
13:            t_p = g · ndir + k
14:            a_p = g' · ndir + k
15:            w_p = cc · ndir + k
16:            Tras_p[t_p] = Tras_p[t_p] + AgregT_p[a_p] · T_p[w_p]
17:         fin para
18:      fin si
19:   fin para
20: fin para
21: {Comunicaciones}
22: EnvioRecepcion(Tras_p, TrasD_p)
```

índices de los grupos fuente y de observación y de los datos de la geometría, se calcula el índice que marca la combinación de centros para g y g'. Con este índice se accede al array $CombCen$ para retornar la posición donde comienzan los datos necesarios dentro de $\mathcal{T}$, posición que se almacena en el índice cc. Por su parte, en el array $Tras_p$ (de tamaño $(N_g \cdot k_l)/p$) se acumulan las contribuciones trasladadas para cada grupo según las direcciones del espacio $\boldsymbol{\kappa}$ asignadas al proceso $MiRango$. Finalmente, los índices t_p, a_p, y w_p marcan la posición actual dentro de los arrays parciales correspondientes con las contribuciones trasladadas ($Tras_p$), las contribuciones

agregadas ($AgregT_p$), y el operador de traslación ($\mathcal{T}_p$), respectivamente.

En el bucle más interno del algoritmo 2.4, que comienza en la línea 12, puede observarse el particionado de grano grueso orientado a dividir la carga de trabajo entre los diferentes procesos MPI. En este caso, el particionado se produce a nivel de dirección del espacio $\boldsymbol{\kappa}$, por lo que cada proceso se encarga de un subconjunto de direcciones consecutivas para todos los grupos. De esta manera, entre todos los procesos se tratan las k_l direcciones totales. Los arrays $PrimeraDir$ y $UltimaDir$ son generados por el algoritmo de balanceo de carga en la fase de inicialización y definen el intervalo de direcciones que debe tratar cada proceso. Además, la variable auxiliar $ndir$ indica cuántas direcciones trata cada proceso con el objetivo de poder iterar de forma correcta por los distintos arrays parciales.

Por otra parte, en la línea 4 del algoritmo 2.4, se produce el particionado de grano fino orientado a repartir la carga de trabajo entre los diferentes hilos OpenMP. En este caso, el particionado se produce a nivel de grupo, de tal forma que cada uno de los hilos pertenecientes a un proceso dado trabaja con un subconjunto de grupos y con las direcciones del espacio $\boldsymbol{\kappa}$ asociadas al proceso al que pertenecen dichos hilos. Si bien el trabajo asociado a cada grupo es aproximadamente igual, ya que todos los grupos tienen un número muy similar de grupos lejanos, el uso de una planificación dirigida, `schedule(guided)`, o dinámica, `schedule(dynamic)`, permite corregir ligeros desajustes en la carga de trabajo asociada a cada hilo.

Una vez que todos los procesos completan las traslaciones asociadas a sus direcciones, comienza un nuevo período de comunicaciones similar al que se lleva a cabo tras la agregación. Por tanto, para lograr un intercambio de datos eficiente, se ha optado nuevamente por utilizar el esquema de comunicaciones creado para la agregación, esquematizado en la figura 2.8. De esta forma, paso a paso, cada proceso va recibiendo la información correspondiente a todas las direcciones para los grupos que necesita, que son aquellos que el algoritmo de balanceo de carga le ha asignado para la etapa de desagregación.

El análisis realizado de la computación y de las comunicaciones asociadas a esta etapa permite el modelado de los sumandos de la ecuación (2.11)

correspondientes con la traslación para un problema arbitrario que se resuelve utilizando p procesos y h hilos por proceso. Nuevamente, se tienen en cuenta tanto el número de operaciones asignadas a cada proceso como el peso de los mensajes intercambiados usando MPI:

$$T_t \approx F_t \cdot \frac{k_l}{p \cdot h} \cdot \sum_{i=0}^{N_g-1} g_i^{(l)}, \tag{2.14a}$$

$$C_t \approx \beta \cdot (p-1) \cdot \frac{k_l \cdot N_g}{p^2}, \tag{2.14b}$$

donde $g_i^{(l)}$ representa el número de grupos lejanos al grupo i-ésimo, mientras que F_t es el factor ajustable que permite modelar el tiempo por operación para la etapa de traslación. Como en las etapas anteriores, este factor se estima utilizando datos empíricos y depende del sistema empleado. Al igual que en el caso de la agregación, puede observarse cómo el esquema de comunicaciones utilizado (ver figura 2.8) es escalable, ya que al aumentar el número de procesos se reduce el peso de las comunicaciones.

2.3.6. Desagregación

Para explicar el procedimiento desarrollado para llevar a cabo la desagregación con múltiples procesadores gráficos usando MPI y CUDA, se muestra el pseudocódigo correspondiente con el algoritmo 2.5. Como se ha hecho en los subapartados anteriores, y para facilitar la comprensión del pseudocódigo, las variables comunes con los algoritmos ya presentados conservan el mismo nombre y representan la misma información.

Al igual que en el caso de la agregación y del cálculo de las interacciones cercanas, los diferentes procesos paralelos trabajan con un subconjunto de grupos consecutivos de forma que, entre todos, se trata el total de los N_g grupos. Los arrays $PrimerGrupo_a$ y $UltimoGrupo_a$, generados por el algoritmo de balanceo de carga en la fase de inicialización, definen el intervalo de grupos con los que debe trabajar cada proceso MPI. Merece la pena destacar que estos arrays coinciden con los usados para la agregación,

Algoritmo 2.5 Desagregación usando MPI y CUDA.

Entrada: $MiRango$, $dimBloque$, $dimGrid$, $PrimerGrupo_a$, $UltimoGrupo_a$, k_l, $DatosGeo_p$, $TrasD_p$, $Iterando_p$, $Lejanas_p$ (inicializado a 0).

Salida: $Lejanas$ contiene las interacciones lejanas de cada elemento.

1: {**Cálculo**}
2: $miPrimerElemento = PrimerElemento(PrimerGrupo_a[MiRango]) + IdBloque \cdot dimBloque + IdHilo$
3: $miUltimoElemento = UltimoElemento(UltimoGrupo_a[MiRango])$
4: $salto = dimBloque \cdot dimGrid$
5: {Para cada elemento de la geometría a tratar por $MiRango$}
6: **para** $e = miPrimerElemento$ **hasta** $miUltimoElemento$ **paso** $salto$ **hacer**
7: $g = IndiceGrupo(e)$
8: {Para cada dirección del espacio $\boldsymbol{\kappa}$}
9: **para** $k = 0$ **hasta** $k_l - 1$ **hacer**
10: $e_p = e - PrimerElemento(PrimerGrupo_a[MiRango])$
11: $t_p = (g - PrimerGrupo_a[MiRango]) \cdot k_l + k$
12: $Lejanas_p[e_p] = Lejanas_p[e_p] + ContribDis(e_p, g, k, DatosGeo_p, TrasD_p[t_p], Iterando_p[e_p])$
13: **fin para**
14: **fin para**
15: {**Comunicaciones**}
16: $Gather(Lejanas_p, Lejanas, 0)$

puesto que el coste computacional de la agregación es proporcional al de la desagregación, como se desprende de las ecuaciones (2.7a) y (2.9a).

La línea 6 del algoritmo 2.5 muestra el resultado del reparto de la carga de trabajo a nivel de proceso MPI (grano grueso) y a nivel de hilo (grano fino). El reparto por grupos puede considerarse como de grano grueso, mientras que el reparto a nivel de elemento asociado a cada hilo CUDA puede considerarse de grano fino. Al igual que en los *kernels* mostrados anteriormente, para resolver aquellos problemas cuyo tamaño es superior a la memoria del dispositivo, el bucle principal puede dividirse para trabajar en varios pasos con bloques de elementos contiguos hasta que se procesan todos los elementos.

La función de tipo `__device__` *ContribDis* engloba los cálculos necesarios para obtener una nueva contribución lejana. Cada una de las interacciones lejanas calculadas por el proceso $MiRango$ se acumula en el array $Lejanas_p$, donde cada posición se corresponde con un elemento de la geometría analizada. En el caso de la desagregación, los datos parciales relativos a la geometría ($DatosGeo_p$) y al iterando ($Iternado_p$) ya se encuentran en la memoria del dispositivo, puesto que se transfieren inmediatamente antes de la ejecución del *kernel* que se encarga de la agregación. Sin embargo, el array $TrasD_p$ sí que debe ser transferido desde el anfitrión al dispositivo antes de poder llevar a cabo la desagregación. Posteriormente, una vez que la GPU finaliza con los cálculos relativos a la desagregación, el array $Lejanas_p$ es transferido del dispositivo al anfitrión.

En cuanto todos los procesos completan el cálculo de las contribuciones lejanas correspondientes a los elementos de sus grupos, comienza un nuevo período de comunicaciones. Esta comunicación tiene como objetivo que el proceso raíz reúna toda la información calculada por los procesos. De esta forma, a partir de la información contenida en los arrays $Lejanas_p$ de cada proceso, el proceso raíz compone el array $Lejanas$ para disponer de las interacciones lejanas sobre todos los elementos de la geometría. Para llevar cabo esta comunicación, se utiliza una rutina MPI de tipo *gather* que sigue un esquema como el mostrado en la figura 2.7.

Tras el período de comunicaciones, y una vez que el proceso raíz ya dispone de todas las contribuciones lejanas, la suma elemento a elemento del array $Lejanas$ y el array $Cercanas$ (obtenido tras la ejecución del algoritmo 2.2) va a permitir obtener el vector solución para la iteración actual.

Después de analizar cómo se lleva a cabo esta etapa del FMM, ya se pueden modelar los sumandos de la ecuación (2.11) correspondientes con la desagregación para un problema con N elementos o funciones base y que se resuelve utilizando p GPU. Al igual que en las etapas anteriores, se tiene en cuenta tanto el número de operaciones asociadas a cada proceso como

el peso de los mensajes intercambiados en las comunicaciones:

$$T_d \approx F_d \cdot \frac{k_l \cdot N}{p} + \beta' \cdot \frac{B_z \cdot k_l \cdot N_g + B_z \cdot N}{p}, \tag{2.15a}$$

$$C_d \approx \beta \cdot \frac{N}{p}, \tag{2.15b}$$

donde F_d es el factor incorporado para modelar el tiempo por operación para esta etapa y que, como en los casos anteriores, se estima de forma empírica. De los bytes transferidos entre cada CPU y su GPU asociada, $(B_z \cdot k_l \cdot N_g)/p$ se corresponden con el array $TrasD_p$, en sentido CPU-GPU, y $(B_z \cdot N)/p$ se corresponden con las contribuciones lejanas almacenadas en cada array $Lejanas_p$, en sentido GPU-CPU. Por su parte, en la ecuación (2.15b) se muestra el tiempo dedicado a las comunicaciones de tipo MPI presentadas en la figura 2.7. Las comunicaciones empleadas en esta etapa también cumplen el objetivo de ser escalables, puesto que al aumentar el número de procesos se reduce el número de elementos enviados por cada proceso.

2.4. Resultados

En este apartado, se muestran varios resultados experimentales obtenidos con la implementación paralela y heterogénea para sistemas de memoria distribuida basados en CPU + GPU. Por su parte, los datos de tiempo de ejecución, consumo de memoria, aceleración y eficiencia asociados a los problemas analizados en este capítulo se muestran en los puntos 6.1 y 6.3.

2.4.1. Presión acústica

Los resultados de presión acústica que se muestran a continuación se corresponden con el análisis de dos objetos dispersores diferentes:

- Una aeronave (Airbus A3xx a escala 1:1), en la cual, a su vez, se analizan dos configuraciones diferentes:

- Configuración típica con los motores bajo el ala.
- Configuración de bajo ruido con los motores sobre el ala (ver más ejemplos de configuraciones de bajo ruido en [137, 138]).

- Una esfera con un diámetro de 4 m.

del ruido generado por una aeronave considerando dos configuraciones diferentes para las fuentes de ruido (motores de la aeronave). La primera de ellas es una configuración típica con los motores bajo el ala, mientras que en la segunda los motores están situados sobre el ala (configuración de bajo ruido [137, 138]).

La distribución de la presión sobre la superficie de la aeronave se ha obtenido a una frecuencia de 1 kHz. Para ello, la aeronave se ha modelado utilizando una malla con $\sim$6 elementos por longitud de onda lineal, lo que representa un total de 1009392 facetas triangulares. Puesto que cada faceta se corresponde con una incógnita, se tiene que $N = 1009392$. El ruido producido por los dos motores situados en el ala izquierda se ha modelado utilizando dos fuentes puntuales, colocadas debajo o encima de dicha ala, dependiendo de la configuración analizada en cada caso. Con el objetivo de lograr unos resultados suficientemente precisos, se ha fijado como condición de parada del algoritmo iterativo un error residual $\epsilon \leq 10^{-2}$, el cual se alcanza tras 89 iteraciones en la configuración con los motores bajo el ala y tras 92 iteraciones en la configuración de bajo ruido (motores sobre el ala).

Por su parte, en el caso de la esfera, se ha obtenido la distribución de presión acústica sobre su superficie cuando dicha esfera se ilumina con una onda plana de 11,5 kHz que se propaga en la dirección $+z$. En esta ocasión, el objeto se ha modelado utilizando una malla con $\sim$10,2 elementos por longitud de onda lineal, lo que se traduce en un total de 6001966 facetas triangulares ($N = 6001966$). Para este problema, se ha impuesto un error residual $\epsilon \leq 10^{-3}$, lo que supone un total de 14 iteraciones del algoritmo.

En la figura 2.9, se muestra la presión acústica sobre la superficie de la aeronave en su configuración típica (motores bajo el ala). La vista desde la parte inferior izquierda del avión permite observar la concordancia entre los valores máximos de la presión acústica y la posición de los motores bajo

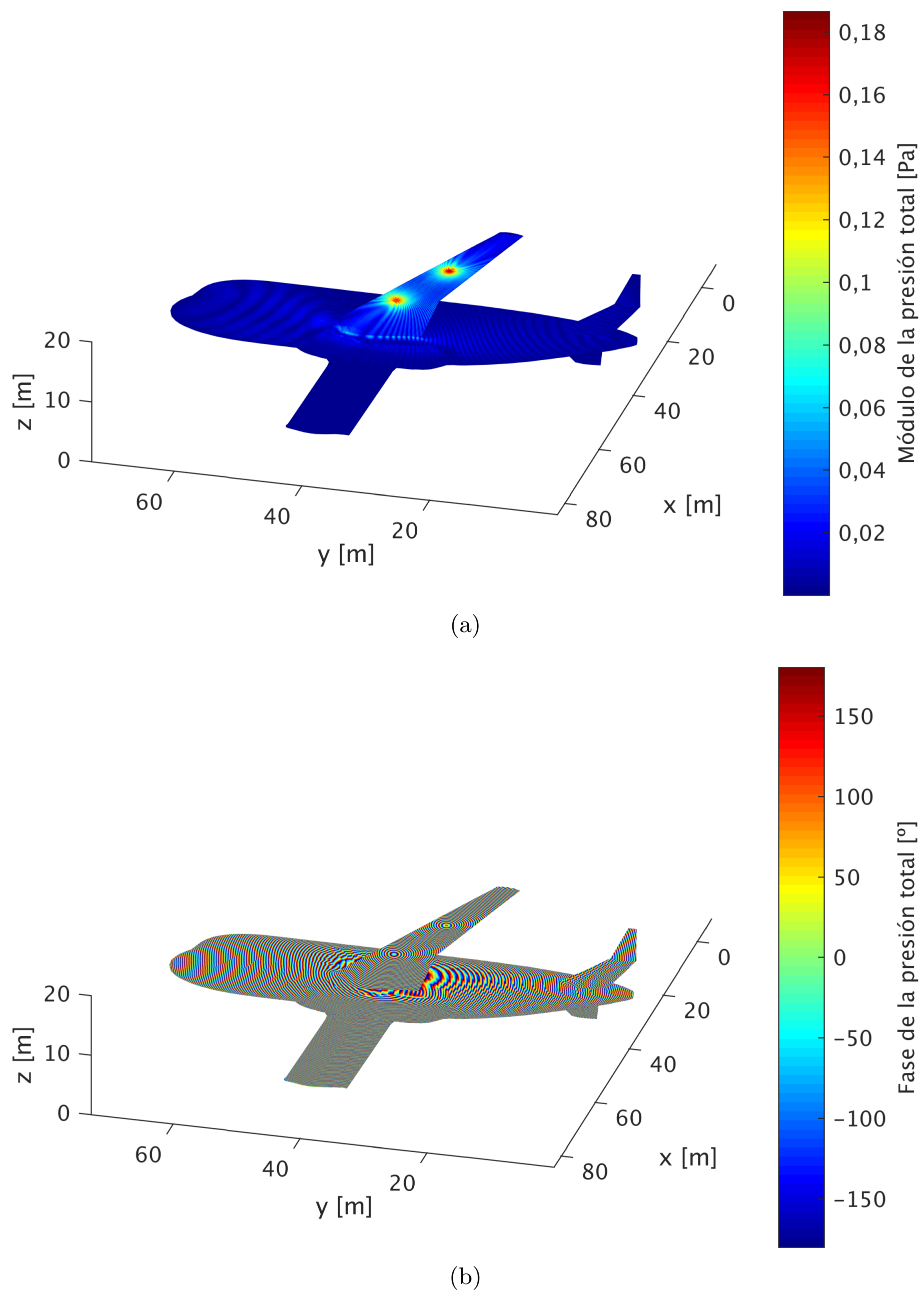


Figura 2.9: Presión acústica total sobre la superficie de una aeronave Airbus serie A3xx. Dos motores debajo del ala izquierda. Frecuencia de las fuentes de ruido: 1 kHz. (a) Módulo. (b) Fase.

el ala (situados en las coordenadas $x_1 = 23{,}5\,\mathrm{m}$, $y_1 = 41{,}5\,\mathrm{m}$, $z_1 = 2{,}5\,\mathrm{m}$, y $x_2 = 13\,\mathrm{m}$, $y_2 = 36\,\mathrm{m}$, $z_2 = 3{,}35\,\mathrm{m}$).

Por su parte, en la figura 2.10, se muestra la presión acústica sobre la superficie de la misma aeronave, usando una configuración de bajo ruido (motores sobre el ala). La vista desde la parte superior izquierda del avión permite apreciar que en esta ocasión también existe una correspondencia entre los valores máximos de la presión acústica y la nueva ubicación de los motores (situados, en este caso, en las coordenadas $x_1 = 23{,}5\,\mathrm{m}$, $y_1 = 41{,}5\,\mathrm{m}$, $z_1 = 5{,}25\,\mathrm{m}$, y $x_2 = 13\,\mathrm{m}$, $y_2 = 36\,\mathrm{m}$, $z_2 = 5{,}7\,\mathrm{m}$).

Finalmente, en la figura 2.11, se muestra la presión acústica sobre la superficie de la esfera cuando esta se ilumina con una onda plana de 11,5 kHz que se propaga en la dirección $+z$. En este caso, la vista utilizada permite observar la parte inferior de la esfera, donde se localizan los valores máximos de presión acústica y una menor variación en su fase.

2.4.2. Validación

Con el objetivo de comprobar la corrección de los resultados obtenidos y así validar la implementación paralela y heterogénea del FMM para sistemas basados en CPU + GPU, se ha recurrido a la comparación con la implementación para CPU del FMM presentada en [13].

La exactitud de la implementación presentada en este capítulo se ha evaluado teniendo en cuenta distintas exigencias con respecto al error residual (inferior a 10^{-2} en el caso de la aeronave e inferior a 10^{-3} en el caso de la esfera). Dicho error residual se define de la siguiente manera:

$$\epsilon = \frac{\|Kp - g\|_2}{\|g\|_2}, \tag{2.16}$$

donde $\|\cdot\|_2$ es la norma euclídea (norma-2), K es la matriz del sistema (matriz de rigidez), g es el vector de excitación (relacionado con la presión incidente), y p es la solución numérica correspondiente al sistema $Kp = g$.

Para cuantificar las diferencias entre los resultados de la implementación paralela y heterogénea del FMM y la implementación de referencia

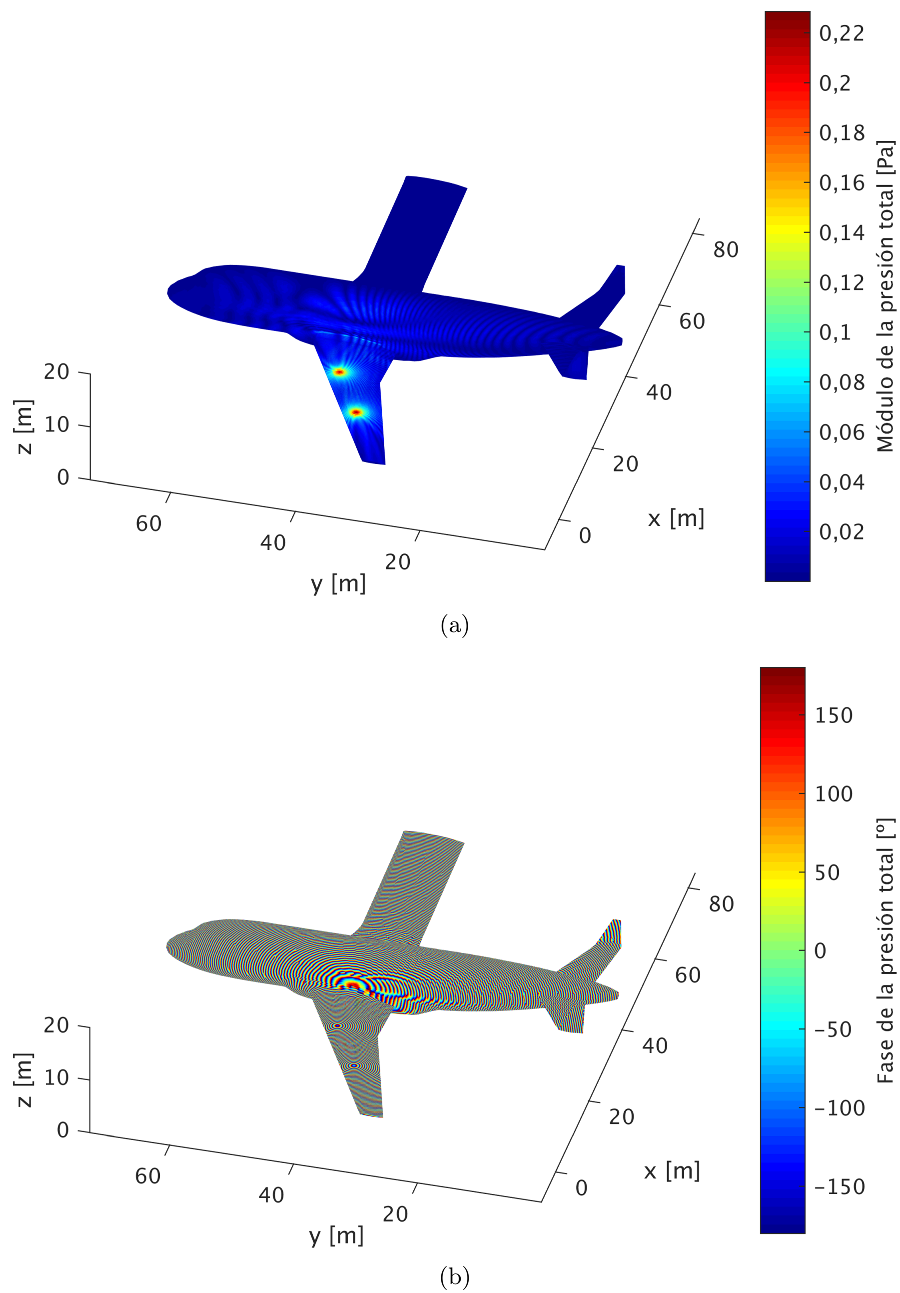


Figura 2.10: Presión acústica total sobre la superficie de una aeronave Airbus serie A3xx. Dos motores sobre el ala izquierda. Frecuencia de las fuentes de ruido: 1 kHz. (a) Módulo. (b) Fase.

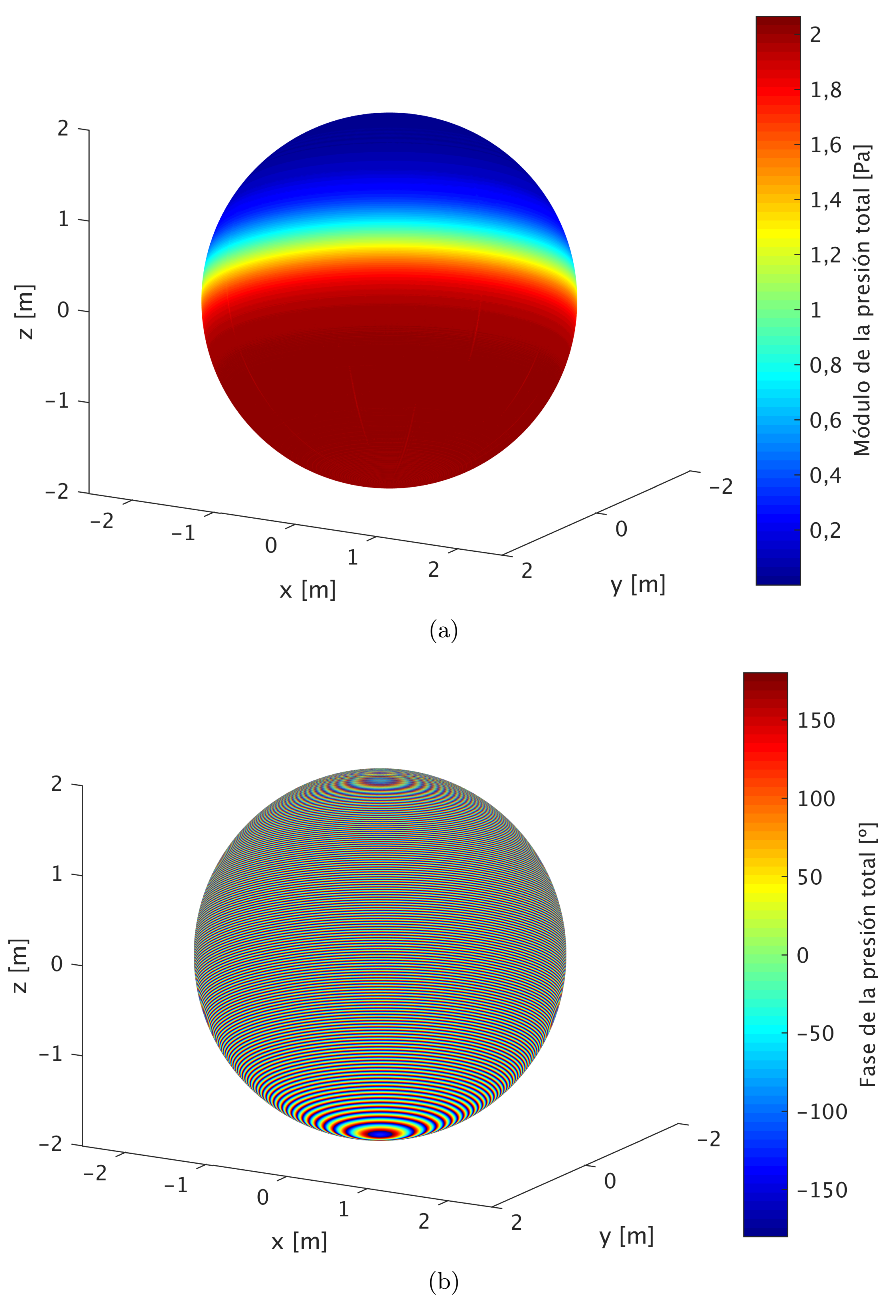


Figura 2.11: Presión acústica total sobre la superficie de una esfera $\oslash 4\,\mathrm{m}$ iluminada por una onda plana de 11,5 kHz que se propaga en la dirección $+z$. (a) Módulo. (b) Fase.

(implementación paralela para procesadores convencionales), se ha utilizado tanto el error relativo (ϵ_{rel}) como la raíz cuadrada del error cuadrático medio (ϵ_{RMS}), definidos de la siguiente manera:

$$\epsilon_{rel} = \frac{\|\hat{p} - p\|_2}{\|p\|_2}, \tag{2.17a}$$

$$\epsilon_{\mathrm{RMS}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} |\hat{p}_i - p_i|^2}, \tag{2.17b}$$

donde N es el número de elementos, p es la solución tomada como referencia (en este caso, obtenida con la implementación para CPU del FMM), y $\hat{p}$ es la solución que se desea comparar (en esta ocasión, obtenida con la implementación para CPU + GPU del FMM).

		Iteraciones	ϵ	ϵ_{rel}	ϵ_{RMS}
A3xx	CPU	89	$9{,}7 \cdot 10^{-3}$	–	–
	CPU + GPU	89	$9{,}7 \cdot 10^{-3}$	$7{,}9 \cdot 10^{-4}$	$1{,}2 \cdot 10^{-5}$
Esfera	CPU	14	$7{,}7 \cdot 10^{-4}$	–	–
	CPU + GPU	14	$7{,}7 \cdot 10^{-4}$	$4{,}7 \cdot 10^{-4}$	$6{,}6 \cdot 10^{-4}$

Tabla 2.1: Error de la implementación para CPU + GPU del FMM en comparación con la implementación del FMM para procesadores convencionales. Airbus serie A3xx ($N = 1009392$, $f = 1\,\mathrm{kHz}$) y esfera $\oslash 4\,\mathrm{m}$ ($N = 6001966$, $f = 11{,}5\,\mathrm{kHz}$).

En la tabla 2.1, se muestra la comparativa de los resultados obtenidos con la implementación de FMM para CPU + GPU frente a los resultados obtenidos usando usando la implementación de referencia. En ambos casos (aeronave y esfera), la diferencia entre la solución heterogénea y la solución de referencia es menor que el error residual impuesto. Si la tolerancia exigida requiriese errores residuales de 10^{-4} o inferiores, la precisión de la aritmética que se ha utilizado (simple precisión) podría no ser suficiente. En dichos casos, sería conveniente utilizar aritmética de doble precisión (números reales de 64 bits y complejos de 128 bits).

Capítulo 3

Algoritmo FMM-FFT paralelo y heterogéneo aplicado al problema de dispersión acústica

Índice

3.1. Algoritmo FMM-FFT aplicado al problema de dispersión acústica 83

3.1.1. Inicialización 84

3.1.2. Interacciones cercanas 84

3.1.3. Interacciones lejanas 85

3.2. Paralelización para sistemas convencionales y heterogéneos 88

3.2.1. Decisiones de diseño de la implementación propia 88

3.2.2. Inicialización 93

3.2.3. Interacciones cercanas 98

3.2.4. Agregación 100

3.2.5. Traslación 103

3.2.6. Desagregación 108

3.3. Resultados 110
3.3.1. Validación 111

En este capítulo, se describe una nueva herramienta que se basa en el algoritmo FMM-FFT y que se aplica a la resolución de problemas de dispersión acústica. Por una parte, se ha desarrollado una versión dirigida a CPU convencionales, preparada tanto para sistemas de memoria compartida como para sistemas de memoria distribuida. Además, al igual que sucedía con la herramienta presentada en el capítulo 2, también se ha desarrollado una versión que puede ejecutarse en equipos con un único procesador gráfico, en sistemas con múltiples tarjetas gráficas, o en sistemas de memoria distribuida cuyos nodos satisfagan una de las dos configuraciones anteriores.

Con esta herramienta se persigue un objetivo principal, reducir el tiempo de ejecución tomando como referencia la implementación del FMM presentada en el capítulo 2. El lenguaje elegido para llevar a cabo la programación de esta herramienta ha sido CUDA C para la implementación orientada a sistemas heterogéneos con tarjetas gráficas, y C para la implementación dirigida a sistemas con procesadores convencionales.

En el apartado 3.1, se describen brevemente las diferentes etapas que componen el algoritmo FMM-FFT, particularizando para el caso de la implementación propia del algoritmo que se ha llevado a cabo en esta Tesis.

Después, en el apartado 3.2, se analizan en detalle las estrategias adoptadas en el desarrollo de las diferentes versiones de esta herramienta. Merece la pena resaltar el particionado ideado, tanto en la versión para CPU, basado en una división del dominio, como en la versión para CPU + GPU, basado en una división del dominio combinada con una descomposición funcional [79]. También resulta destacable el uso simultáneo de diferentes paradigmas de programación paralela, combinando MPI y OpenMP en la versión para CPU, y recurriendo al uso de MPI, OpenMP y CUDA en la versión para CPU + GPU.

Finalmente, en el apartado 3.3, se muestran los resultados de validación utilizados para determinar la corrección de las diferentes implementaciones

del algoritmo que se han presentado en este capítulo. Por su parte, los resultados computacionales se muestran en el capítulo dedicado a tal fin, el capítulo 6.

3.1. Algoritmo FMM-FFT aplicado al problema de dispersión acústica

Al igual que sucedía en el caso del FMM presentado en el capítulo anterior, el FMM-FFT [44, 45] se utiliza como esquema acelerador del método iterativo empleado, el GMRES [29], elegido por ser un método eficiente y muy robusto para la resolución de problemas de dispersión acústica [30].

En lo relativo al GMRES, en este caso tampoco ha sido necesario aplicar reinicios en el algoritmo [29] para los problemas analizados, aunque la posibilidad de reiniciar el algoritmo se encuentre implementada, evitando así ralentizaciones en la velocidad de convergencia [22].

Por su parte, la implementación desarrollada del FMM-FFT conserva la misma estructura que el FMM, con una fase de inicialización (*setup* del algoritmo) seguida de una fase iterativa cuyo objetivo es converger hacia la solución del problema, cumpliendo con unos requisitos de exactitud establecidos previamente. En cada una de las iteraciones, se lleva a cabo un producto matriz-vector sin calcular de forma explícita la matriz del sistema (K). Para ello, el producto matriz-vector se sustituye por tres pasos alternativos, el cálculo de las interacciones lejanas, el cálculo de las interacciones cercanas y la suma de ambas interacciones. En el caso del FMM-FFT, sólo el cálculo de las interacciones lejanas difiere con respecto del FMM, como se mostrará en los siguientes puntos.

Las interacciones cercanas, aquellas limitadas a elementos pertenecientes a grupos cercanos entre sí, no satisfacen las condiciones asociadas a las transformaciones descritas por los teoremas de adición [40], por lo que se calculan evaluando directamente la parte correspondiente de la matriz del sistema. Por su parte, el cálculo de las interacciones lejanas puede realizarse de forma eficiente basándose en los teoremas de adición y en una descomposición en ondas planas [127]. Al igual que en el caso del FMM, durante

la fase de inicialización del algoritmo los N elementos (funciones base) se dividen en N_g grupos disjuntos. Posteriormente, ya en la fase iterativa, el cálculo de las interacciones lejanas se lleva a cabo en tres etapas consecutivas: agregación, traslación y desagregación. La representación esquemática de estos pasos puede verse en la figura 2.1.

La diferencia fundamental entre el FMM y el FMM-FFT se centra en la forma en la que se lleva a cabo la traslación. En el FMM-FFT, la traslación se lleva a cabo sustituyendo una convolución tridimensional en el dominio real por un producto en el dominio transformado.

3.1.1. Inicialización

Al igual que en el FMM, el primer paso del FMM-FFT consiste en la inicialización o *setup* del algoritmo. Durante esta fase, las operaciones más importantes de entre las que se realizan son las siguientes: i) lectura de los datos de entrada, ii) procesado de la geometría, iii) selección del tamaño de los grupos, iv) cálculo de la presión incidente, v) cálculo del operador de traslación. La descripción de las operaciones i) a iv) puede verse en el punto 2.2.1.

La última de las tareas llevadas a cabo en la inicialización es el cálculo del operador de traslación ($\mathcal{T}$). Esta tarea resulta de especial importancia en el FMM-FFT ya que, dependiendo de cómo se aborde la resolución del problema, el cálculo y posterior almacenamiento de $\mathcal{T}$ podría diferir notablemente respecto del enfoque tomado para el FMM. En esta Tesis, para el caso del FMM-FFT, el operador de traslación se calcula evitando las direcciones espaciales redundantes (ver punto 2.3.2.4) y se almacena únicamente su versión en el dominio real (ver punto 3.2.2.4 para más detalles). Ambas decisiones permiten lograr unos consumos de memoria moderados sin penalizar de forma notable los tiempos de ejecución.

3.1.2. Interacciones cercanas

Las interacciones cercanas no satisfacen los teoremas de adición [40] y debido a ello deben ser evaluadas directamente, lo que requiere el cálculo de

la radiación producida por cada pareja de bases que pertenezcan al conjunto de grupos cercanos (véase el capítulo 4 de [13]). Por tanto, el cálculo de las interacciones cercanas se limita a aquellas bases que pertenecen al mismo grupo o a grupos vecinos. Al igual que en el caso del FMM, en esta Tesis se consideran grupos cercanos o vecinos a aquellos grupos cuyos bordes comparten al menos un punto común, como se mostraba en la figura 2.3).

El coste computacional de la etapa del FMM-FFT dedicada al cálculo de las interacciones cercanas puede modelarse usando las expresiones (2.6a) y (2.6b), puesto que el número de operaciones no varía con respecto al FMM. A partir del número de operaciones, O_c, se puede obtener el coste temporal de las interacciones cercanas, T_c, incorporando un factor ajustable, F_c, que permita modelar el tiempo por operación para esta etapa. Este factor permite agrupar en un único valor el número de instrucciones básicas por operación y la velocidad con la que los procesadores empleados son capaces de ejecutarlas. Puesto que F_c depende del contexto (implementación, compilador, hardware, etc.), debe ser estimado y ajustado de forma experimental.

3.1.3. Interacciones lejanas

En el FMM-FFT las interacciones lejanas se calculan de forma eficiente basándose en los teoremas de adición y en una descomposición de las ondas esféricas en ondas planas [127]. El cálculo de dichas interacciones se lleva a cabo en tres etapas consecutivas: agregación, traslación y desagregación. Estas etapas se describen brevemente a continuación.

3.1.3.1. Agregación

En la agregación, se representa la radiación producida por las bases pertenecientes a cada grupo mediante una expansión multipolar de L multipolos que se sitúan en el centro del grupo. En la figura 2.1, se muestra una representación esquemática de esta etapa. Para un grupo fuente dado, la radiación producida por cada función base perteneciente a dicho grupo se traslada al centro del mismo, donde se suma a la del resto de fuentes

trasladadas de ese mismo grupo. La agregación conlleva el cálculo de k_l componentes del espacio $\boldsymbol{\kappa}$ para cada elemento o función base (véase en más detalle en el capítulo 4 de [13]).

El coste computacional para la agregación puede modelarse usando las expresiones (2.7a) y (2.7b), ya que esta etapa tampoco sufre modificaciones en el FMM-FFT. Para obtener el coste temporal a partir del número de operaciones, nuevamente se incorpora un factor que permite modelar el tiempo por operación. Como en el caso de las interacciones cercanas, este factor debe ser estimado y ajustado de forma experimental.

3.1.3.2. Traslación

En el caso del FMM-FFT, esta etapa es de suma importancia, ya que en ella se centra el principal cambio con respecto al FMM. En la traslación, la expansión multipolar producida en la agregación se traslada desde el centro de cada grupo fuente (origen) a todos los centros de aquellos grupos de observación (destino) que son lejanos. A continuación, las contribuciones trasladadas se acumulan en el centro de cada grupo de observación para cada una de las direcciones del espacio $\boldsymbol{\kappa}$ (véase el capítulo 4 de [13]). Así, al trasladar todas las contribuciones de un grupo en un único paso y no base a base, se logra diagonalizar el cálculo de las interacciones lejanas. La figura 2.1 muestra una representación esquemática de esta etapa.

La diferencia entre el FMM-FFT y el FMM se basa en la forma en la que se lleva a cabo esta etapa. En el caso del FMM-FFT, la traslación realizada en el FMM (que puede verse como una convolución [44]) se sustituye por una sucesión de k_l FFT directas, productos punto a punto (en el dominio transformado) y FFT inversas, que permiten obtener las contribuciones trasladadas para todos los grupos y para todas las direcciones del espacio $\boldsymbol{\kappa}$ de forma eficiente. En los puntos 3.2.2.4 y 3.2.5 se mostrará una descripción más detallada de las particularidades del cálculo del operador de traslación y de la propia traslación, respectivamente, para el caso concreto del FMM-FFT paralelo y heterogéneo aplicado a problemas de dispersión acústica.

Tras analizar esta etapa, resulta especialmente interesante estudiar el nuevo coste computacional de la misma debido al cambio sustancial res-

pecto del FMM. Para ello, se recurre a las siguientes expresiones:

$$O_t = k_l \cdot n_{fft} \cdot Q \cdot \log(Q), \quad (3.1a)$$

$$T_t \approx F_t \cdot O_t, \quad (3.1b)$$

donde O_t refleja el número de operaciones que se llevan a cabo en la traslación, con un total de $k_l \cdot n_{fft}$ FFT, donde cada una de ellas tiene una complejidad computacional $\mathcal{O}(Q \cdot \log(Q))$. El número de FFT a realizar por cada dirección del espacio euclídeo, n_{fft}, depende de la estrategia utilizada (más detalles en los puntos 3.2.2.4 y 3.2.5), mientras que Q representa el número total de grupos, incluyendo grupos vacíos y grupos no vacíos. T_t define el coste temporal de la traslación, que nuevamente se deriva a partir del número de operaciones y de un factor, en este caso F_t, que permite modelar el tiempo por operación para esta etapa. Puesto que F_t es dependiente del contexto, se debe ajustar usando datos empíricos obtenidos en ejecuciones de prueba.

3.1.3.3. Desagregación

La desagregación es la última etapa dentro del cálculo de las interacciones lejanas. En esta etapa, se calcula el campo lejano sobre las bases de cada grupo mediante la expansión de las componentes del espacio $\boldsymbol{\kappa}$, que son trasladadas desde el centro del grupo al propio elemento, completando así la diagonalización del cálculo de las interacciones lejanas (véase el capítulo 4 de [13]). En la figura 2.1 del capítulo dedicado al FMM se muestra una representación esquemática.

El coste computacional para la desagregación se puede modelar utilizando las expresiones (2.9a) y (2.9b), puesto que esta etapa es idéntica en el caso del FMM y del FMM-FFT. Al igual que en el resto de etapas, el factor utilizado para modelar el tiempo por operación se debe estimar forma empírica.

3.2. Paralelización para sistemas convencionales y heterogéneos

Tanto la versión paralela para procesadores convencionales como la versión paralela y heterogénea orientada a GPU del algoritmo FMM-FFT que se presentan en este capítulo implementan varias técnicas ideadas para obtener una herramienta eficiente desde el punto de vista computacional. En el subapartado 3.2.1, se enumeran las decisiones de diseño más destacables. Posteriormente, en los subapartados 3.2.2 a 3.2.6, se analizan de forma detallada las rutinas desarrolladas para implementar las distintas etapas del FMM-FFT, ya sea utilizando CPU convencionales o procesadores gráficos.

3.2.1. Decisiones de diseño de la implementación propia

Para llevar a cabo la implementación de las diferentes versiones del FMM-FFT, se ha proseguido con la aplicación de aquellas técnicas empleadas en la implementación del FMM cuyo uso conserve las ventajas originales, por ejemplo: el uso de *octrees*, la minimización de los requisitos de almacenamiento en memoria, el particionado a varios niveles para aumentar el grado de paralelismo del algoritmo, etc. Asimismo, se han incorporado técnicas específicas para el caso del FMM-FFT, como se muestra en los siguientes apartados.

3.2.1.1. Particionado

En lo relativo al particionado, para el FMM-FFT se han utilizado dos diseños diferentes, uno para la implementación orientada a procesadores convencionales y otra para la implementación orientada a aceleradores gráficos (ver figura 3.1). En la versión orientada a CPU, el particionado utilizado se basa únicamente en una división del dominio. Por su parte, en la versión orientada a sistemas basados en CPU + GPU se utiliza nuevamente una estrategia de doble nivel, basada en una descomposición funcional y una división del dominio, de la misma forma que en el caso del FMM presentado en el capítulo anterior.

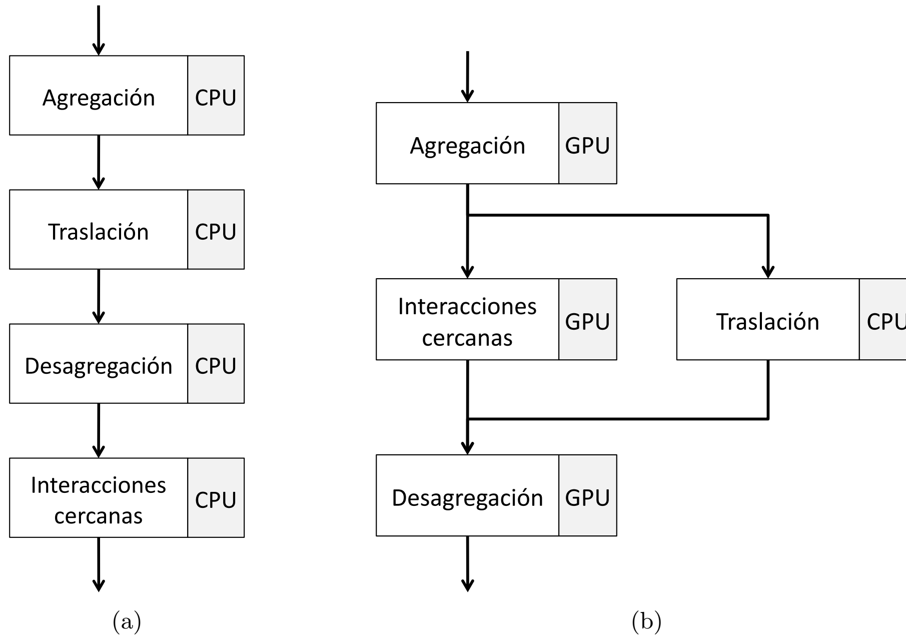


Figura 3.1: Orden de ejecución de las etapas del FMM-FFT. (a) Implementación para procesadores convencionales: sólo división del dominio. (b) Implementación para sistemas basados en CPU + GPU: descomposición funcional más división del dominio.

En el caso de la versión para procesadores convencionales, debido a que la traslación y el cálculo de las interacciones cercanas comparten recursos hardware (las CPU), no tiene sentido simultanear estas dos etapas computacionalmente costosas. Por contra, en la versión orientada a aceleradores gráficos, sí que se pueden llevar a cabo de forma concurrente la traslación (en CPU) y el cálculo de las interacciones cercanas (en GPU), permitiendo así un uso eficaz de los recursos computacionales disponibles. La decisión de ejecutar en CPU la traslación se debe a que es la etapa más intensiva en memoria, por lo que puede beneficiarse del mayor tamaño de la memoria principal y las cachés asociadas a las CPU.

La división del dominio que se propone en este caso también aborda el problema de dos formas diferentes dependiendo de la arquitectura a la que esté dirigida la implementación. A continuación, se muestra de forma

esquemática cómo se divide el dominio de trabajo en las diferentes etapas para cada una de las implementaciones.

- FMM-FFT para procesadores convencionales:
  - Traslación
    - Asignación a nivel de proceso $\mapsto$ por bloques de direcciones consecutivas
    - Asignación a nivel de hilo $\mapsto$ por direcciones
  - Resto de etapas (no interviene el operador de traslación)
    - Asignación a nivel de proceso $\mapsto$ por bloques de grupos consecutivos
    - Asignación a nivel de hilo $\mapsto$ por grupos
- FMM-FFT para sistemas basados en CPU + GPU:
  - Traslación
    - Asignación a nivel de proceso $\mapsto$ por bloques de direcciones consecutivas
    - Asignación a nivel de hilo OpenMP $\mapsto$ por direcciones
  - Resto de etapas (no interviene el operador de traslación)
    - Asignación a nivel de proceso (GPU) $\mapsto$ por bloques de grupos consecutivos
    - Asignación a nivel de hilo CUDA $\mapsto$ por elementos

Como se puede observar, en todos los casos se logra producir un particionado de doble nivel, de grano grueso y de grano fino.

Por su parte, la figura 3.2 muestra, también de forma esquemática, la descomposición funcional que permite la ejecución simultánea de la traslación y las interacciones cercanas, y la división del dominio en dos niveles para las etapas mencionadas anteriormente, en la implementación paralela y heterogénea del FMM-FFT.

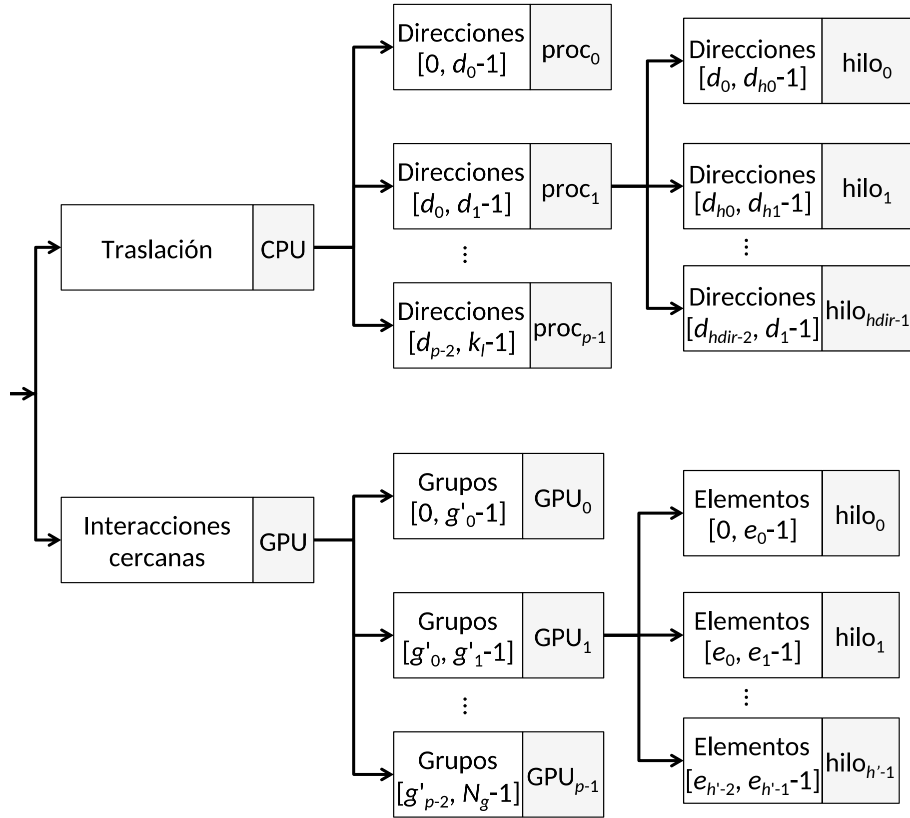


Figura 3.2: Particionado diseñado para el FMM-FFT en sistemas basados en CPU + GPU. Descomposición funcional y división del dominio en varios niveles.

3.2.1.2. Balanceo de carga

Un reparto equilibrado de la carga de trabajo entre los diferentes procesos paralelos permite reducir el tiempo de ejecución, pero requiere disponer de la capacidad de estimar, a priori y de forma precisa, el tiempo por iteración. En el caso del FMM-FFT, debido a las diferencias que presentan las versiones implementadas, se deben utilizar dos expresiones distintas para modelar el tiempo por iteración. En el caso de la implementación orientada a sistemas heterogéneos basados en CPU + GPU, la expresión para mode-

lar el tiempo por iteración se muestra en la ecuación (3.2a) y coincide con la presentada en la ecuación (2.11) del capítulo dedicado al FMM. Por su parte, en el caso de la implementación para sistemas basados en CPU convencionales, la expresión utilizada difiere de la anterior debido a la ausencia de descomposición funcional (la traslación y el cálculo de las interacciones cercanas no se ejecutan de forma simultánea). En dicho caso, se utiliza la expresión correspondiente a la ecuación (3.2b).

$$T_{it}^{het} \approx T_a + C_a + \text{máx}\{T_c, T_t\} + C_c + C_t + T_d + C_d, \tag{3.2a}$$

$$T_{it}^{cpu} \approx T_a + C_a + T_t + C_t + T_d + C_d + T_c + C_c. \tag{3.2b}$$

Como se puede observar, en ambas expresiones, (3.2a) y (3.2b), se tiene en cuenta tanto el tiempo de cálculo, T, como el tiempo consumido por las comunicaciones (intercambio de datos entre procesos), C.

Una vez modelado el tiempo por iteración, y siguiendo con el objetivo de lograr un reparto equilibrado de la carga de trabajo, se ha vuelto a optar por el diseño de una estrategia de balanceo de carga a varios niveles. En primer lugar, se lleva a cabo un reparto estático del trabajo a nivel de proceso basado en el cálculo del tiempo de ejecución usando las expresiones (3.2a) y (3.2b). Puesto que el modelado de la computación y de las comunicaciones permite estimar a priori y de forma precisa el tiempo por iteración, el balanceo de carga a nivel de proceso ya se puede llevar a cabo durante la fase de inicialización del algoritmo.

Además, para todas las etapas ejecutadas en CPU, también se dispone de un balanceo de carga a nivel de hilo, el cual aprovecha los planificadores dinámicos de OpenMP [96]. Finalmente, para aquellas etapas ejecutadas en GPU, la gestión de los bloques de hilos CUDA y los *warps* [112, 111] que los componen permite equilibrar la carga de trabajo entre los diferentes multiprocesadores existentes en una GPU.

3.2.2. Inicialización

Durante la fase de inicialización del algoritmo, se llevan a cabo diversas operaciones cuya ejecución no vuelve a repetirse a lo largo del algoritmo, pero que tienen una influencia muy notable en la fase iterativa del FMM-FFT. Tanto la selección del tamaño de los grupos, como el reparto de la carga de trabajo, la distribución de datos o el cálculo del operador de traslación se deben adaptar al diseño ideado para el FMM-FFT. En los siguientes apartados, se detallan los aspectos más importantes de las tareas abordadas en la inicialización.

3.2.2.1. Selección del tamaño de grupo

Al igual que en la implementación del FMM presentada en el capítulo anterior, en el caso del FMM-FFT el tamaño del grupo también se selecciona buscando minimizar el tiempo de ejecución. Utilizando la expresión (3.2a) o (3.2b), dependiendo de la versión del FMM-FFT de la que se trate en cada caso, se hace una búsqueda para encontrar el tamaño de grupo que produzca un menor tiempo por iteración.

En este punto, merece la pena comentar que, en el caso de la implementación para sistemas basados en CPU convencionales, parece evidente que el tamaño óptimo de los grupos va a ser menor (en términos de longitud de onda) que el que se escogería en el caso del FMM. Puesto que en el FMM-FFT la aceleración lograda se centra en la etapa de la traslación, esto se va a traducir en un mayor número de grupos más pequeños (más traslaciones) y, a su vez, en un menor número de direcciones del espacio $\boldsymbol{\kappa}$.

Sin embargo, en la implementación para sistemas heterogéneos, la reducción del tamaño de los grupos respecto del tamaño que se seleccionaría en el caso del FMM no siempre es la mejor solución [116]. Si el tamaño de los grupos es pequeño, la traslación pasa a tener un mayor peso computacional, logrando un mejor aprovechamiento de las FFT, pero el cálculo de las interacciones cercanas pierde peso, lo que puede llegar a ser contraproducente al ser una etapa cuyo cálculo está notablemente acelerado por el uso de GPU. Si, por contra, se selecciona un tamaño de grupo grande,

el cálculo de las interacciones cercanas pasa a tener un peso adecuado a la aceleración de dicha etapa, pero la traslación mediante FFT pierde peso, resultando en un peor aprovechamiento de las FFT, puesto que deben realizarse más FFT (debido al aumento en el número de direcciones del espacio $\boldsymbol{\kappa}$) y cada FFT es de menor tamaño (debido a la reducción en el número de grupos).

Para llevar a cabo la selección del tamaño de grupo, el tiempo consumido por cada etapa del FMM-FFT se modela teniendo en cuenta tanto la computación como las comunicaciones. El cálculo de cada uno de los sumandos de las expresiones (3.2a) y (3.2b) se analizará más adelante, en los puntos 3.2.3 a 3.2.6, donde se estudiarán de forma individualizada las interacciones cercanas, la agregación, la traslación y la desagregación, tanto para la implementación orientada a procesadores convencionales como para la implementación orientada a sistemas heterogéneos.

3.2.2.2. Reparto de la carga de trabajo

Tras fijar el tamaño de los grupos, la siguiente tarea llevada a cabo en la inicialización es el reparto y balanceo de la carga de trabajo. Para conseguir equilibrar la carga de trabajo entre los diferentes procesos paralelos, se debe tener en cuenta el particionado, analizado previamente en el punto 3.2.1.1.

El algoritmo ideado para el caso del FMM puede aplicarse igualmente en este caso ya que, de nuevo, el reparto se basa en la división del trabajo asociado a cada grupo (interacciones cercanas, agregación y desagregación) o en la división el trabajo vinculado a cada dirección del espacio $\boldsymbol{\kappa}$ (cálculo del operador de traslación y traslación).

Para las etapas con un particionado por direcciones, el algoritmo reparte entre los diferentes procesos MPI bloques consecutivos de direcciones consecutivas, utilizando para ello una división euclídea o división entera. En un problema con k_l direcciones y p procesos, se resolverá que los primeros r procesos ($r = k_l \,\%\, p$) trabajan con $\lceil k_l/p \rceil$ direcciones y los $p-r$ procesos restantes trabajan con $\lfloor k_l/p \rfloor$ direcciones ($\lceil \cdot \rceil$ y $\lfloor \cdot \rfloor$ representan la función techo y suelo, respectivamente). Para las etapas con un particionado por grupos, el algoritmo reparte bloques de grupos consecutivos, teniendo en

cuenta la carga de trabajo asociada a cada grupo. Nuevamente, las etapas se tratan de forma independiente, de tal forma que cada proceso puede trabajar con distintos grupos para el cálculo de las interacciones cercanas y para la agregación-desagregación, con el objetivo de alcanzar una carga computacional lo más homogénea posible.

3.2.2.3. Distribución inicial de datos

Una vez que se reparte la carga de trabajo entre los diferentes procesos, tienen lugar las comunicaciones en las que se lleva a cabo la distribución inicial de los datos del problema a resolver.

El proceso raíz, que conoce el rango de grupos asignado a cada proceso, se encarga de distribuir los datos necesarios entre todos los procesos, principalmente datos relacionados con la geometría. Para ello, se utiliza una comunicación MPI de tipo *scatter* que sigue el esquema mostrado anteriormente en la figura 2.6.

3.2.2.4. Operador de traslación

Dentro de las tareas llevadas a cabo en la fase de inicialización del algoritmo, el cálculo del operador de traslación puede resultar computacionalmente costoso, especialmente para problemas de gran tamaño. La rutina encargada de esta tarea resulta especialmente importante en el FMM-FFT puesto que, dependiendo de las decisiones de diseño tomadas, el almacenamiento del operador $\mathcal{T}$ podría diferir notablemente respecto del mismo caso para el FMM.

En esta Tesis y para el caso del FMM, el operador de traslación se calcula y almacena evitando las direcciones espaciales redundantes (ver punto 2.3.2.4). En el caso del FMM-FFT, el problema del almacenamiento de $\mathcal{T}$ puede atacarse desde dos perspectivas diferentes: primando el tiempo de ejecución o primando el almacenamiento en memoria.

Si se busca reducir el tiempo de ejecución, el procedimiento para calcular y almacenar el operador de traslación durante la fase de inicialización podría seguir el siguiente esquema:

- Calcular el operador $\mathcal{T}$ sin compactar, incluyendo grupos vacíos y direcciones espaciales redundantes.
- Aplicarle una FFT a la versión completa de $\mathcal{T}$ para obtener la versión transformada del mismo.
- Eliminar la versión en el dominio real de $\mathcal{T}$.
- Almacenar únicamente la versión transformada del operador, $\mathcal{F}(\mathcal{T})$.

Sin embargo, si lo que prima es minimizar los requisitos de almacenamiento en memoria, el procedimiento para calcular y almacenar el operador de traslación podría basarse en los siguientes pasos:

- Durante la fase de inicialización:
  - Calcular y almacenar el operador $\mathcal{T}$ de forma compacta, evitando grupos vacíos, cercanos, y direcciones espaciales redundantes.
- Durante la fase iterativa, en la etapa de traslación y para cada dirección del espacio $\boldsymbol{\kappa}$:
  - Desplegar la parte de $\mathcal{T}$ que se corresponde con la dirección del espacio $\boldsymbol{\kappa}$ a tratar (pasando de una versión compactada a una completa).
  - Aplicar una FFT para obtener la versión transformada de la parte de $\mathcal{T}$ con la que se está trabajando.
  - Llevar a cabo la traslación para la dirección actual (ver punto 3.2.5).
  - Eliminar la versión de $\mathcal{T}$ en el dominio espectral.

Cada una de las estrategias mostradas en los párrafos anteriores presenta pros y contras, por lo que resulta conveniente analizarlos antes de

decantarse por una de ellas. En la estrategia en la que se prima el tiempo de ejecución, el número de transformadas a realizar en la traslación (en cada iteración), una FFT y una IFFT por dirección del espacio $\boldsymbol{\kappa}$, es menor que en el caso de la estrategia en la que se prima el almacenamiento en memoria, donde se realizan dos FFT y una IFFT por dirección del espacio $\boldsymbol{\kappa}$. Sin embargo, en la estrategia en la que se prima el tiempo los requisitos de almacenamiento para $\mathcal{T}$ pueden llegar a ser mucho mayores que en la estrategia en la que se prima la memoria. En el primer caso, se mantiene en memoria una copia del operador en el dominio espectral en su forma desplegada (no se pueden aprovechar las direcciones espaciales redundantes). Por contra, en el segundo caso, sólo es necesario mantener en memoria el operador en el dominio real, en su forma compactada, y la parte del operador en el dominio espectral en su forma desplegada correspondiente a una única dirección del espacio $\boldsymbol{\kappa}$ (aquella sobre la que se está operando actualmente).

En esta Tesis, tras realizar un análisis experimental para problemas de diferente tamaño resueltos utilizando un número variable de procesos, se ha optado por la estrategia que prima la minimización de los requisitos de almacenamiento en memoria. En el punto dedicado a la etapa de la traslación, 3.2.5, se explicará de forma detallada las implicaciones de utilizar esta estrategia en dicha etapa.

Tanto en el caso de la implementación para procesadores convencionales como en el caso de la implementación heterogénea para procesadores gráficos, la rutina encargada de esta tarea es la misma, puesto que se ejecuta sólo en las CPU. Por ello, y puesto que en esta Tesis se ha decidido no mantener en memoria la versión transformada de $\mathcal{T}$, la estrategia seguida para calcular el operador de traslación se corresponde con la mostrada en el algoritmo 2.1 dentro del capítulo dedicado al FMM.

En este caso, al finalizar el cálculo de $\mathcal{T}$ tampoco se lleva a cabo ningún tipo de comunicación. Cada proceso calcula y almacena la parte del operador que va a utilizar en la traslación, por lo que no se necesita realizar ningún intercambio de datos.

3.2.3. Interacciones cercanas

El cálculo de las interacciones entre elementos cercanos es, junto con la traslación, el paso más costoso del FMM-FFT desde el punto de vista computacional. Puesto que la versión paralela y heterogénea es idéntica a la presentada en el capítulo 2 y detallada en el algoritmo 2.2, en este apartado se explican únicamente las técnicas utilizadas para la versión orientada a procesadores convencionales, basadas en el uso de MPI + OpenMP. Para ello, se muestra el pseudocódigo correspondiente al algoritmo 3.1.

Algoritmo 3.1 Interacciones cercanas usando MPI y OpenMP.

Entrada: $MiRango$, h, $PrimerGrupo_c$, $UltimoGrupo_c$, $NumElementos$, $NumElementosCercanos$, $DatosGeo_p$, $Iterando_p$.
Salida: $Cercanas$ contiene las interacciones cercanas de cada elemento.
1: {**Cálculo**}
2: {Para los grupos a tratar por el proceso $MiRango$}
3: #pragma omp parallel for num_threads(h)
4: **para** $g = PrimerGrupo_c[MiRango]$ **hasta** $UltimoGrupo_c[MiRango]$ **hacer**
5: $g_p = g - PrimerGrupo_c[MiRango]$
6: $DatosCercanos = EmpaquetaDatos(g_p, DatosGeo_p, Iterando_p)$
7: {Para cada elemento del grupo actual}
8: **para** $i = 0$ **hasta** $NumElementos[g] - 1$ **hacer**
9: $e = PrimerElemento(g) + i$
10: $e_p = e - PrimerElemento(PrimerGrupo_c[MiRango])$
11: $Cercanas_p[e_p] = ImpedanciaMutua(e_p, NumElementosCercanos[g], DatosCercanos)$
12: **fin para**
13: **fin para**
14: {**Comunicaciones**}
15: $Gather(Cercanas_p, Cercanas, 0)$

Con el objetivo de facilitar la comprensión de los diferentes pseudocódigos y mantener la coherencia con el capítulo anterior, las variables y rutinas comunes con los algoritmos presentados previamente conservan el mismo nombre y representan la misma información. Por su parte, i indica la posición relativa de un elemento dentro de su grupo, e es el índice de la posición absoluta del elemento actual, y e_p marca la posición relativa (contando

desde el primer elemento asignado a un proceso dado) del elemento actual. Los arrays $NumElementos$, y $NumElementosCercanos$ tienen N_g posiciones en las que se almacenan el número de elementos y el número de elementos cercanos de cada grupo, respectivamente. La rutina auxiliar $EmpaquetaDatos$ permite encapsular en la estructura $DatosCercanos$ todos los datos necesarios para calcular las interacciones cercanas relativas al grupo actual, utilizando información relativa a la geometría ($DatosGeo_p$) y el array que contiene la parte asignada del iterando actual ($Iterando_p$).

En las líneas 3 y 4 del algoritmo 3.1, se muestra el doble nivel utilizado para dividir el trabajo computacional en esta etapa del FMM-FFT. En este caso, se realiza un reparto de la carga a nivel de grupo tanto para la división de grano más grueso (a nivel de procesos MPI) como para la división de grano más fino (a nivel de hilos OpenMP). Nuevamente, al existir múltiples procesos paralelos, no hay un único proceso encargado de iterar desde el primer hasta el último grupo, sino que los distintos procesos trabajan con un subconjunto de grupos consecutivos. A su vez, los hilos asociados a cada proceso se reparten de forma dinámica los grupos con los que deben trabajar, usando para ello la directiva de OpenMP `schedule(dynamic)`.

Una vez que todos los procesos completan el cálculo de las contribuciones cercanas correspondientes a los elementos de sus grupos, comienza un período de comunicaciones que tiene como objetivo que el proceso raíz reúna los datos calculados por los diferentes procesos, para así componer el array $Cercanas$ que debe contener las interacciones cercanas de todos los elementos. Para ello, se utiliza una comunicación global de tipo *gather* como la mostrada en el caso del FMM en la figura 2.7.

Tras analizar los pseudocódigos correspondientes a esta etapa, ya se pueden modelar aquellas partes de las ecuaciones (3.2a) y (3.2b) relativas a las interacciones cercanas. Nuevamente, conviene recalcar que la estimación del tiempo dedicado a cada etapa se realiza antes de comenzar con la fase iterativa, durante la inicialización del algoritmo.

Por una parte, en la implementación heterogénea para GPU tendríamos:

$$
\begin{aligned}
T_c &\approx F_c \cdot n_{pi} \cdot \frac{\sum_{i=0}^{N_g-1} n_i \cdot n_i^{(c)}}{p} \\
&+ \beta' \cdot \frac{B_z \cdot N + \sum_{i=0}^{N_g-1} B_{geo} \cdot n_i^{(c)}}{p}, \qquad (3.3a)
\end{aligned}
$$

$$
C_c \approx \beta \cdot \frac{N}{p}, \qquad (3.3b)
$$

es decir, exactamente el mismo modelo que en el caso del FMM, analizado en el capítulo anterior en el punto 2.3.3.

Por otra parte, en la implementación paralela para sistemas basados únicamente en CPU, tendríamos:

$$
T_c \approx F_c \cdot n_{pi} \cdot \frac{\sum_{i=0}^{N_g-1} n_i \cdot n_i^{(c)}}{p \cdot h}, \qquad (3.4a)
$$

$$
C_c \approx \beta \cdot \frac{N}{p}, \qquad (3.4b)
$$

donde se tiene en cuenta tanto el número de procesos (p) como el número de hilos por proceso (h). En este caso, la carga de trabajo se divide entre el total de $p \cdot h$ hilos y ya no hay transferencias a través del bus *PCI Express*. Para calcular el tiempo consumido por las comunicaciones de tipo MPI, se tienen en cuenta tanto el tamaño de los mensajes como el factor ajustable β que refleja la capacidad de la red de interconexión. Las comunicaciones modeladas en la ecuación (3.4b) (y en la ecuación (3.3b)) se corresponden con una rutina MPI de tipo *gather* idéntica a la esquematizada para el caso del FMM en la figura 2.7.

3.2.4. Agregación

En este caso, la versión paralela y heterogénea (implementación para GPU) también es idéntica a la presentada en el capítulo 2, detallada en el algoritmo 2.3. Por ello, en este apartado se explican únicamente las técnicas utilizadas para la versión orientada a procesadores convencionales, cuya

implementación está basada en el uso combinado de MPI y OpenMP. A continuación, en el algoritmo 3.2, se muestra el pseudocódigo correspondiente a la implementación para CPU. Nuevamente, las variables comunes con los algoritmos presentados con anterioridad conservan el mismo nombre y representan la misma información.

Algoritmo 3.2 Agregación usando MPI y OpenMP.

Entrada: $MiRango$, h, $PrimerGrupo_a$, $UltimoGrupo_a$, k_l, $NumElementos$, n_{pi}, $DatosGeo_p$, $Iterando_p$, $Agreg_p$ (inicializado a 0).

Salida: $AgregT_p$ contiene las contribuciones requeridas por el proceso $MiRango$ para la traslación.

1: {**Cálculo**}
2: {Para los grupos a tratar por el proceso $MiRango$}
3: #pragma omp parallel for num_threads(h)
4: **para** $g = PrimerGrupo_a[MiRango]$ **hasta** $UltimoGrupo_a[MiRango]$ **hacer**
5:   {Para cada elemento perteneciente al grupo actual}
6:   **para** $i = 0$ **hasta** $NumElementos[g] - 1$ **hacer**
7:     $e_p = PrimerElemento(g) + i - PrimerElemento(PrimerGrupo_a[MiRango])$
8:     {Para cada punto de integración}
9:     **para** $pto = 0$ **hasta** $n_{pi} - 1$ **hacer**
10:       $DatosInt = CalculaPtosIntegracion(e_p, DatosGeo_p, pto)$
11:       {Para cada dirección del espacio $\boldsymbol{\kappa}$}
12:       **para** $k = 0$ **hasta** $k_l - 1$ **hacer**
13:         $a_p = (g - PrimerGrupo_a[MiRango]) \cdot k_l + k$
14:         $Agreg_p[a_p] = Agreg_p[a_p] + ContribAg(e_p, g, pto, k, DatosGeo_p, DatosInt, Iterando_p[e_p])$
15:       **fin para**
16:     **fin para**
17:   **fin para**
18: **fin para**
19: {**Comunicaciones**}
20: $EnvioRecepcion(Agreg_p, AgregT_p)$

En esta etapa del FMM-FFT, los diferentes procesos paralelos también trabajan con un subconjunto de grupos consecutivos, de tal forma que,

entre todos los procesos, llevan a cabo los cálculos asociados al total de grupos (N_g). Los arrays $PrimerGrupo_a$ y $UltimoGrupo_a$, que definen el intervalo de grupos con los que debe trabajar cada proceso, también son generados en este caso por el algoritmo de balanceo de carga en la fase de inicialización. Al igual que sucedía en el caso del FMM, el reparto de grupos para la agregación no tiene por qué coincidir con el reparto obtenido para el caso de las interacciones cercanas.

En las líneas 3 y 4 puede observarse el reparto de la carga de trabajo a doble nivel. La línea 4 es el resultado del reparto a nivel de proceso (grano grueso), mientras que la línea 3 define el reparto a nivel de hilo (grano más fino). Con objeto de lograr un reparto de la carga de trabajo lo más equilibrado posible, en este caso los hilos OpenMP también toman de forma dinámica los grupos asignados a cada proceso.

Las comunicaciones que tienen lugar tras el cálculo de las contribuciones agregadas siguen el mismo diseño específico presentado en el capítulo anterior y representado de forma esquemática en la figura 2.8. Con esta técnica, en $p-1$ pasos, todos los procesos disponen de la información relativa a sus direcciones del espacio $\boldsymbol{\kappa}$ para todos los grupos. Asimismo, conviene enfatizar que cada proceso recibe sólo los datos para aquellas direcciones con las que va a trabajar en la etapa de traslación.

Una vez analizados los pseudocódigos correspondientes a esta etapa, ya se pueden modelar aquellas partes de las ecuaciones (3.2a) y (3.2b) relativas a la agregación, teniendo en cuenta tanto el número de operaciones asociadas a cada proceso como el peso de los mensajes intercambiados en las comunicaciones.

En el caso de la implementación heterogénea para GPU tendríamos:

$$T_a \approx F_a \cdot n_{pi} \cdot \frac{k_l \cdot N}{p} + \beta' \cdot \frac{B_{geo} \cdot N + B_z \cdot k_l \cdot N_g}{p}, \tag{3.5a}$$

$$C_a \approx \beta \cdot (p-1) \cdot \frac{k_l \cdot N_g}{p^2}, \tag{3.5b}$$

es decir, el mismo modelo que para el caso de la agregación en la implementación paralela y heterogénea del FMM, analizado en el punto 2.3.4 del

capítulo anterior.

Por su parte, en el caso de la implementación paralela para sistemas basados únicamente en procesadores convencionales, tendríamos:

$$T_a \approx F_a \cdot n_{pi} \cdot k_l \cdot \frac{N}{p \cdot h}, \tag{3.6a}$$

$$C_a \approx \beta \cdot (p-1) \cdot \frac{k_l \cdot N_g}{p^2}, \tag{3.6b}$$

donde se tiene en cuenta tanto el número de procesos MPI (p) como el número de hilos OpenMP por proceso (h) en la división de la carga de trabajo. Nuevamente, se utiliza un factor obtenido empíricamente (F_a) para modelar el tiempo por operación y así estimar el valor del tiempo de ejecución a partir del número de operaciones. Por su parte, como se puede observar, las comunicaciones son idénticas en ambas implementaciones.

3.2.5. Traslación

La etapa de traslación es especialmente importante en el algoritmo FMM-FFT, puesto que en esta etapa se centra la gran diferencia con el algoritmo FMM del que deriva. Es un paso costoso desde el punto de vista computacional y además muy intensivo en memoria.

Debido a las decisiones tomadas al diseñar el particionado, esta etapa se ejecuta sólo en las CPU (ver algoritmo 3.3), incluso en la versión orientada a procesadores gráficos. En dicha versión, al mismo tiempo que las GPU trabajan en el cálculo de la interacciones cercanas, los procesadores convencionales llevan a cabo la traslación. Nuevamente, se ha recurrido al uso de la construcción `sections` de OpenMP para definir dos secciones paralelas, una para la traslación y otra para las interacciones cercanas. Sin embargo, dichas secciones no sirven para ejecutar de forma concurrente y paralela la propia traslación. Por tanto, además de la descomposición funcional resultante del uso de secciones paralelas, se debe disponer de suficientes hilos para abordar la traslación usando todos los núcleos disponibles en las CPU empleadas. Para ello, se habilita el uso del paralelismo anidado mediante la instrucción `omp_set_nested(1)`. Así, cada proceso MPI puede ejecutar la

traslación de forma paralela y, además, se dispone de un hilo independiente que controla el acelerador gráfico encargado del cálculo de las interacciones cercanas. Asimismo, cabe mencionar que en la versión orientada a procesadores convencionales no es necesario recurrir a esta solución, ya que la traslación y el cálculo de las interacciones cercanas no se ejecutan de forma concurrente debido a que utilizan los mismos recursos hardware.

Algoritmo 3.3 Traslación usando MPI y OpenMP.

Entrada: $MiRango$, h, $PrimeraDir$, $UltimaDir$, N_g, Q, $DatosGeo_p$, $CombCen$, $AgregT_p$, $\mathcal{T}_p$, $Tras_p$ (inicializado a 0).

Salida: $TrasD_p$ contiene las contribuciones requeridas por el proceso $MiRango$ para la desagregación.

1: {**Cálculo**}
2: $ndir = UltimaDir[MiRango] - PrimeraDir[MiRango] + 1$
3: fftw_plan_with_nthreads(h_{fft})
4: {Para las direcciones del espacio $\boldsymbol{\kappa}$ asignadas al proceso $MiRango$}
5: #pragma omp parallel num_threads(h_{dir})
6: **para** $k = 0$ **hasta** $ndir - 1$ **hacer**
7: $\quad \mathcal{T}_p^+ = Extiende_{\mathcal{T}}(\mathcal{T}_p, k, DatosGeo_p, CombCen)$
8: $\quad AgregT_p^+ = Extiende_A(AgregT_p, k, DatosGeo_p, N_g)$
9: $\quad$ fftw_execute_dft($directa, \mathcal{T}_p^+, \mathcal{T}_p^+$)
10: $\quad$ fftw_execute_dft($directa, AgregT_p^+, AgregT_p^+$)
11: $\quad$ #pragma omp parallel for num_threads(h_{fft})
12: $\quad$ **para** $i = 0$ **hasta** $NumMuestras - 1$ **hacer**
13: $\quad\quad Tras_p^+[i] = AgregT_p^+[i] \cdot \mathcal{T}_p^+[i]$
14: $\quad$ **fin para**
15: $\quad$ fftw_execute_dft($inversa, Tras_p^+, Tras_p^+$)
16: $\quad$ **para** $g = 0$ **hasta** $N_g - 1$ **hacer**
17: $\quad\quad t_p = g \cdot ndir + k$
18: $\quad\quad Tras_p[t_p] = Extrae(Tras_p^+, g, DatosGeo_p)$
19: $\quad$ **fin para**
20: **fin para**
21: {**Comunicaciones**}
22: $EnvioRecepcion(Tras_p, TrasD_p)$

En el pseudocódigo 3.3, $CombCen$ es el array auxiliar que contiene las correspondencias entre las posiciones de los centros de los N_g grupos no vacíos y las posiciones de los datos dentro del operador de traslación, en

el dominio real y en su forma compactada, de cada proceso ($\mathcal{T}_p$). Por su parte, la rutina auxiliar $Extiende_{\mathcal{T}}$ permite obtener, a partir del operador de traslación en su forma compactada ($\mathcal{T}_p$), el operador de traslación en su forma desplegada para la dirección actual k ($\mathcal{T}_p^+$). De forma similar, $Extiende_A$ permite obtener, a partir de las contribuciones agregadas ($AgregT_p$), un array extendido ($AgregT_p^+$) que contiene las contribuciones agregadas para la dirección actual teniendo en cuenta todos los grupos y que se rellena además con ceros hasta igualar el tamaño de $\mathcal{T}_p^+$. Conviene aclarar que sólo aquellas posiciones de $AgregT_p^+$ que se correspondan con grupos no vacíos contendrán información relevante, mientras que el resto se mantendrán con un valor nulo. Finalmente, la rutina *Extrae* permite obtener las contribuciones trasladadas para los N_g grupos no vacíos y la dirección actual k a partir del array $Tras_p^+$, el cual contiene información espuria debida a los grupos vacíos y al *zero padding*.

En el bucle más externo del algoritmo 3.3, que comienza en la línea 6, se llevan a cabo de manera simultánea dos de los niveles del particionado de la carga de trabajo para esta etapa del FMM-FFT. El primer nivel, de grano más grueso, permite dividir la carga de trabajo entre los diferentes procesos MPI. En este caso, el particionado se produce a nivel de dirección de forma que cada proceso paralelo se encarga de un subconjunto de direcciones del espacio $\boldsymbol{\kappa}$ consecutivas. De esta manera, entre todos los procesos se tratan las k_l direcciones. Los arrays *PrimeraDir* y *UltimaDir* son generados por el algoritmo de balanceo de carga en la fase de inicialización y definen el intervalo de direcciones que debe tratar cada proceso. Por su parte, el segundo nivel de particionado se encuentra definido en las líneas 5 y 6 del algoritmo 3.3. En esta ocasión, el particionado también se produce a nivel de dirección, de forma que cada uno de los h_{dir} hilos OpenMP se encarga de un subconjunto de las *ndir* direcciones que debe tratar su proceso padre.

Además de los dos niveles de particionado detallados en el párrafo anterior, también se ha ideado un tercer nivel de particionado, de grano más fino, que permite aumentar el nivel de paralelismo disponible en esta etapa. Este tercer nivel, que comprende desde la línea 9 hasta la línea 15, permite paralelizar las FFT directas e inversa, así como el producto punto a punto llevado a cabo en las líneas 12 a 14. En este punto, resulta necesario destacar que el número de hilos utilizados en los dos niveles más finos (h_{dir} y

h_{fft}) es tal que $h_{dir} \cdot h_{fft} = h$, cumpliendo así con el uso de h hilos por proceso para la traslación.

En el caso de la implementación propia de la traslación para el FMM-FFT, reflejada en el algoritmo 3.3, se recurre al uso de la FFT y la FFT inversa implementadas en la librería FFTW [139]. Se ha escogido dicha librería por ser libre, gratuita, multi-plataforma, multi-hilo, y permitir obtener un rendimiento a la altura de las librerías comerciales de uso habitual. Resulta especialmente interesante el hecho de que la librería FFTW sea multi-hilo, ya que esto permite disponer de un tercer nivel de paralelismo, para la ejecución de FFT directas e inversa, como ya se ha mencionado en el párrafo anterior. Con la instrucción utilizada en la línea 3, se configura la librería para que ejecute todas las FFT (tanto las dos directas como la inversa) de forma paralela y utilizando h_{fft} hilos. Asimismo, las tres FFT empleadas en esta etapa se han configurado para que sean de tipo *in-place*, de forma que el resultado se almacena en el propio array de entrada, minimizando el uso de memoria. Finalmente, también merece la pena destacar que el número de muestras utilizadas para llevar a cabo la FFT ($NumMuestras$ en el algoritmo 3.3), se selecciona de forma que $NumMuestras \geq 8Q$, pero buscando siempre el menor número de muestras que sea múltiplo de 2, 3, 5, o 7, con el objetivo de lograr un rendimiento óptimo del algoritmo empleado por la librería FFTW.

Como se comentó en el apartado dedicado a la estrategia para calcular el operador de traslación (ver punto 3.2.2.4), en la implementación propia del FMM-FFT presentada en este capítulo y entre las dos alternativas estudiadas, se ha optado por un diseño en el que no se almacena la versión del operador de traslación en el domino espectral. Esto resulta en una implementación ligeramente más lenta, puesto que se realizan dos FFT y una IFFT en lugar de una única FFT y una IFFT, pero que tiene unos requisitos de memoria notablemente menores que su alternativa. Aún así, merece la pena mencionar un caso límite en el que la alternativa basada en el almacenamiento del operador de traslación en su versión espectral puede resultar interesante: cuando el número de procesos MPI es tal que la cantidad de direcciones del espacio $\boldsymbol{\kappa}$ asignadas a cada proceso es próxima a uno ($k_l/p \approx 1$), el almacenamiento del operador de traslación en su forma desplegada ya no supone un sobrecoste puesto que, incluso en la estrategia

que prima la memoria, se debe contar con el operador desplegado, al menos, para una dirección.

Tras la parte dedicada al cálculo, una vez que todos los procesos completan las traslaciones asociadas a sus direcciones, comienza un nuevo período de comunicaciones que sigue una estructura idéntica al que se lleva a cabo tras la agregación. Por tanto, se utiliza nuevamente el esquema de comunicaciones ejemplificado en la figura 2.8 del capítulo dedicado al FMM. En resumen, tras $p-1$ pasos, los procesos logran disponer de la información correspondiente a todas las direcciones para los grupos que necesitan (aquellos asignados por el algoritmo de balanceo de carga), todo ello minimizando las transferencias de datos entre procesos.

Una vez analizada esta etapa de forma detallada, puede llevarse a cabo el modelado de los sumandos de las ecuaciones (3.2a) y (3.2b) correspondientes con la traslación. Al igual que en el resto de etapas, se tienen en cuenta tanto el número de operaciones asignadas a cada proceso como el peso de los mensajes MPI intercambiados:

$$T_t \approx F_t \cdot \frac{k_l}{p \cdot h} \cdot Q \cdot \log(Q), \tag{3.7a}$$

$$C_t \approx \beta \cdot (p-1) \cdot \frac{k_l \cdot N_g}{p^2}, \tag{3.7b}$$

donde Q representa el número total de grupos (vacíos y no vacíos) y F_t es el factor ajustable utilizado para modelar el tiempo por operación en esta etapa. Como en el resto de etapas, este factor se estima utilizando datos empíricos y depende del sistema (hardware, software, middleware, etc.) empleado. Por simplicidad, en este caso, F_t también encapsula el número de FFT por dirección (ver punto 3.1.3.2). Al igual que en el caso de la agregación, puede comprobarse que el esquema de comunicaciones utilizado (ver figura 2.8) es escalable, de forma que al aumentar el número de procesos se reduce el peso de las comunicaciones. Puesto que esta etapa se ejecuta sólo en CPU tanto en la implementación para procesadores convencionales como en la implementación orientada a procesadores gráficos, las expresiones (3.7a) y (3.7b) son perfectamente válidas para ambas versiones.

3.2.6. Desagregación

Puesto que la principal diferencia entre los algoritmos FMM y FMM-FFT se encuentra en la traslación y en el cálculo del propio operador de traslación, nuevamente la implementación para GPU es idéntica a la presentada en el punto 2.3.6 del capítulo anterior. Teniendo esto en cuenta, este subapartado se centra en la explicación de las técnicas utilizadas en la versión para procesadores convencionales, cuya implementación está basada en el uso de MPI y OpenMP.

Para comenzar con el análisis de la desagregación en la implementación propia del FMM-FFT para CPU, se muestra el pseudocódigo correspondiente al algoritmo 3.4. Como en el caso de los subapartados anteriores, las variables comunes con algoritmos ya presentados tienen el mismo nombre y representan la misma información.

Al igual que en la agregación y en el cálculo de las interacciones cercanas, los diferentes procesos trabajan con un subconjunto de grupos consecutivos de forma que, entre todos ellos, se trata el conjunto de los N_g grupos no vacíos. Los arrays $PrimerGrupo_a$ y $UltimoGrupo_a$, creados en la fase de inicialización por el algoritmo de balanceo de carga, definen el intervalo de grupos con los que debe trabajar cada proceso MPI. Como se indicaba en el capítulo anterior, la asignación de grupos para la agregación y para la desagregación es la misma, puesto que el coste computacional de la agregación es proporcional al de la desagregación (ver ecuaciones (2.7a) y (2.9a)).

Las líneas 3 y 4 del algoritmo 3.4 definen el reparto de la carga de trabajo a nivel de proceso (grano grueso) y a nivel de hilo (grano más fino), respectivamente. Puesto que el trabajo asociado a cada grupo varía según el número de elementos del mismo, para lograr un reparto de la carga lo más equilibrado posible, en este caso los hilos también toman los grupos asignados a su proceso padre de forma dinámica, usando la planificación `schedule(dynamic)` para la paralelización con OpenMP.

Una vez que todos los procesos finalizan con el cálculo de las contribuciones lejanas correspondientes a los elementos de sus grupos asignados y almacenadas en los arrays $Lejanas_p$, comienza un nuevo período de comu-

Algoritmo 3.4 Desagregación usando MPI y OpenMP.

Entrada: $MiRango$, h, $PrimerGrupo_a$, $UltimoGrupo_a$, k_l, $NumElementos$, $DatosGeo_p$, $TrasD_p$, $Iterando_p$, $Lejanas_p$ (inicializado a 0).

Salida: $Lejanas$ contiene las interacciones lejanas de cada elemento.

1: {**Cálculo**}
2: {Para los grupos a tratar por el proceso $MiRango$}
3: #pragma omp parallel for num_threads(h)
4: **para** $g = PrimerGrupo_a[MiRango]$ **hasta** $UltimoGrupo_a[MiRango]$ **hacer**
5: {Para cada elemento perteneciente al grupo actual}
6: **para** $i = 0$ **hasta** $NumElementos[g] - 1$ **hacer**
7: $e_p = PrimerElemento(g) + i - PrimerElemento(PrimerGrupo_a[MiRango])$
8: {Para cada dirección del espacio $\boldsymbol{\kappa}$}
9: **para** $k = 0$ **hasta** $k_l - 1$ **hacer**
10: $t_p = (g - PrimerGrupo_a[MiRango]) \cdot k_l + k$
11: $Lejanas_p[e_p] = Lejanas_p[e_p] + ContribDis(e_p, g, k, DatosGeo_p, TrasD_p[t_p], Iterando_p[e_p])$
12: **fin para**
13: **fin para**
14: **fin para**
15: {**Comunicaciones**}
16: $Gather(Lejanas_p, Lejanas, 0)$

nicaciones. El objetivo de esta comunicación es que el proceso raíz reúna toda la información calculada por los procesos, de forma que, a partir de la información contenida en los arrays parciales $Lejanas_p$, pueda componerse el array completo $Lejanas$ que pasará a contener las interacciones lejanas sobre todos los elementos de la geometría. Esta comunicación se lleva a cabo utilizando una rutina MPI de tipo *gather*, idéntica a la utilizada en el caso de las interacciones cercanas y que sigue un esquema como el mostrado en la figura 2.7 del capítulo previo.

Después del período de comunicaciones, cuando el proceso raíz ya dispone de todas las contribuciones lejanas, se lleva a cabo la suma elemento a elemento del array $Lejanas$ y el array $Cercanas$, y se obtiene, de esta forma, el vector solución para la iteración actual.

Una vez analizados los algoritmos relativos a esta etapa, ya se pueden modelar las partes de las ecuaciones (3.2a) y (3.2b) correspondientes a la desagregación. Nuevamente, se tienen en cuenta tanto el número de operaciones como el peso de las comunicaciones.

En el caso de la implementación heterogénea para GPU tendríamos:

$$T_d \approx F_d \cdot \frac{k_l \cdot N}{p} + \beta' \cdot \frac{B_{geo} \cdot N + B_z \cdot k_l \cdot N_g}{p}, \tag{3.8a}$$

$$C_d \approx \beta \cdot \frac{N}{p}, \tag{3.8b}$$

que coincide con el modelo para la desagregación en la implementación paralela y heterogénea del FMM, analizado en el capítulo anterior dentro del punto 2.3.6, ya que esta fase es idéntica en el FMM y en el FMM-FFT.

Por su parte, en el caso de la implementación paralela sólo para CPU, tendríamos:

$$T_d \approx F_d \cdot k_l \cdot \frac{N}{p \cdot h}, \tag{3.9a}$$

$$C_d \approx \beta \cdot \frac{N}{p}, \tag{3.9b}$$

donde puede apreciarse cómo se tiene en consideración tanto el número de procesos (p) como el número de hilos por proceso (h) para la división de esta etapa del problema. Como sucede en las etapas anteriores, también se recurre al uso un factor obtenido empíricamente (en este caso, F_d). Dicho factor permite modelar el tiempo por operación con objeto de poder estimar el tiempo de ejecución a partir del número de operaciones. Asimismo, como se puede observar en las ecuaciones (3.8b) y (3.9b), las comunicaciones son idénticas en ambas implementaciones (sólo CPU y CPU + GPU).

3.3. Resultados

En esta sección, se validan los resultados obtenidos con las diferentes implementaciones del FMM-FFT presentadas en este capítulo. Los proble-

mas utilizados son los mismos que para el caso del FMM (ver punto 2.4), por lo que, en esta ocasión, se omiten los resultados relativos a la representación de la presión acústica sobre los objetos analizados. Nuevamente, los datos de tiempo de ejecución, consumo de memoria, aceleración y eficiencia asociados a los problemas analizados se muestran en el capítulo 6 (ver puntos 6.2 y 6.3).

3.3.1. Validación

Para comprobar la corrección de los resultados obtenidos con las implementaciones del FMM-FFT presentadas en este capítulo, se ha llevado a cabo una comparación entre dichas implementaciones y sus equivalentes para el caso del FMM. Asimismo, se ha comparado la implementación paralela y heterogénea del FMM-FFT con la implementación del mismo algoritmo para procesadores convencionales.

La exactitud de las implementaciones del FMM-FFT presentadas en este capítulo se ha evaluado teniendo en cuenta diferentes valores objetivo para el error residual: $\epsilon \leq 10^{-2}$ en el caso de la aeronave y $\epsilon \leq 10^{-3}$ para la esfera (ver definición de ϵ en la ecuación (2.16)).

Para medir de forma cuantitativa las diferencias existentes entre los resultados obtenidos con las implementaciones del FMM-FFT (sólo CPU y CPU + GPU) y las implementaciones de referencia (basadas en el FMM), se ha utilizado tanto el error relativo (ϵ_{rel}) como la raíz cuadrada del error cuadrático medio (ϵ_{RMS}), definidos en las ecuaciones (2.17a) y (2.17b), respectivamente.

En la tabla 3.1, se muestra una comparación de los resultados ofrecidos por la implementación del FMM-FFT para sistemas basados en procesadores convencionales y los resultados obtenidos con la implementación del FMM para el mismo tipo de sistemas (utilizada como referencia). En ambos casos (aeronave y esfera), la diferencia entre la solución obtenida con el FMM-FFT y la solución de referencia es menor que el error residual fijado.

Por su parte, en la tabla 3.2, se muestra una comparativa de los resultados proporcionados por la implementación heterogénea del FMM-FFT y los

		Iteraciones	ϵ	ϵ_{rel}	ϵ_{RMS}
A3xx	FMM	89	$9{,}7 \cdot 10^{-3}$	–	–
	FMM-FFT	89	$9{,}7 \cdot 10^{-3}$	$6{,}0 \cdot 10^{-4}$	$9{,}1 \cdot 10^{-6}$
Esfera	FMM	14	$7{,}7 \cdot 10^{-4}$	–	–
	FMM-FFT	14	$7{,}7 \cdot 10^{-4}$	$4{,}2 \cdot 10^{-4}$	$6{,}0 \cdot 10^{-4}$

Tabla 3.1: Error de la implementación del FMM-FFT en comparación con la implementación del FMM para procesadores convencionales. Airbus serie A3xx ($N = 1009392$, $f = 1\,\text{kHz}$) y esfera ⌀4 m ($N = 6001966$, $f = 11{,}5\,\text{kHz}$).

resultados obtenidos con la implementación heterogénea del FMM para el mismo tipo de sistemas (implementación de referencia). Nuevamente, para ambos problemas la diferencia entre la solución ofrecida por el FMM-FFT y la solución de referencia es menor que el error residual fijado.

Finalmente, en la tabla 3.3, se muestra una comparación de los resultados obtenidos con las dos implementaciones del FMM-FFT presentadas en este capítulo: para sistemas basados en procesadores convencionales (utilizada como referencia) y para sistemas heterogéneos basados en

		Iteraciones	ϵ	ϵ_{rel}	ϵ_{RMS}
A3xx	FMM	89	$9{,}7 \cdot 10^{-3}$	–	–
	FMM-FFT	89	$9{,}7 \cdot 10^{-3}$	$7{,}0 \cdot 10^{-4}$	$1{,}1 \cdot 10^{-5}$
Esfera	FMM	14	$7{,}7 \cdot 10^{-4}$	–	–
	FMM-FFT	14	$7{,}7 \cdot 10^{-4}$	$4{,}1 \cdot 10^{-4}$	$5{,}8 \cdot 10^{-4}$

Tabla 3.2: Error de la implementación del FMM-FFT en comparación con la implementación del FMM-FFT para sistemas heterogéneos del tipo CPU + GPU. Airbus serie A3xx ($N = 1009392$, $f = 1\,\text{kHz}$) y esfera ⌀4 m ($N = 6001966$, $f = 11{,}5\,\text{kHz}$).

		Iteraciones	ϵ	ϵ_{rel}	ϵ_{RMS}
A3xx	CPU	89	$9{,}7 \cdot 10^{-3}$	–	–
	CPU + GPU	89	$9{,}7 \cdot 10^{-3}$	$5{,}8 \cdot 10^{-4}$	$8{,}7 \cdot 10^{-6}$
Esfera	CPU	14	$7{,}7 \cdot 10^{-4}$	–	–
	CPU + GPU	14	$7{,}7 \cdot 10^{-4}$	$3{,}6 \cdot 10^{-4}$	$5{,}2 \cdot 10^{-4}$

Tabla 3.3: Error de la implementación para CPU + GPU del FMM-FFT en comparación con la implementación del FMM-FFT para procesadores convencionales. Airbus serie A3xx ($N = 1009392$, $f = 1\,\mathrm{kHz}$) y esfera $\oslash 4\,\mathrm{m}$ ($N = 6001966$, $f = 11{,}5\,\mathrm{kHz}$).

CPU + GPU. En ambos problemas se observa que la diferencia entre ambas soluciones es menor que el error residual marcado como objetivo.

Capítulo 4

Cálculo de la presión total en el problema de dispersión acústica

Índice

4.1. Descripción del cálculo de la presión acústica . 116

4.1.1. Cálculo de la presión total 117

4.1.2. Cálculo de la presión dispersada 118

4.2. Paralelización para sistemas convencionales y heterogéneos 120

4.2.1. Decisiones de diseño de la implementación propia 120

4.2.2. Técnica basada en producto matriz-vector (bMVP) 123

4.2.3. Técnica basada en múltiples productos matriz-vector (mMVP) 127

4.2.4. Técnicas basadas en productos matriciales 132

4.2.5. Solución heterogénea para GPU y Xeon Phi . . . 141

4.3. Resultados 145

4.3.1. Presión acústica 145

4.3.2. Validación 149

A lo largo de este capítulo, se describe una herramienta diseñada para obtener la presión total en planos situados en el exterior de un obstáculo sobre el que incide una onda acústica. Esta herramienta cuenta con múltiples variantes y puede ejecutarse en diferentes tipos de sistemas: sistemas de memoria compartida basados en procesadores convencionales, y sistemas heterogéneos en los que se disponga de procesadores gráficos y/o coprocesadores de tipo Xeon Phi.

Con esta herramienta se pretende reducir el tiempo necesario para resolver un determinado problema, para lo que se ha recurrido a la paralelización y a la aceleración hardware. Asimismo, también se busca posibilitar la resolución de varios problemas de manera simultánea, siempre que se den las condiciones necesarias.

Los lenguajes de programación utilizados han sido C, en las implementaciones para procesadores convencionales y aceleradores Xeon Phi, y CUDA C, para las implementaciones orientadas a procesadores gráficos NVIDIA.

En el apartado 4.1, se realiza una descripción de cómo se lleva a cabo el cálculo de la presión acústica en el exterior de un objeto dispersor. Posteriormente, en el punto 4.2, se presentan las técnicas y estrategias desarrolladas para implementar las diferentes variantes del algoritmo, basadas en el producto matriz-vector o en la multiplicación matricial por bloques. Para finalizar, en el apartado 4.3, se muestran varios resultados obtenidos con esta herramienta aplicada al ámbito del control de ruido en aeronaves.

4.1. Descripción algorítmica del cálculo de la presión acústica en el exterior de un objeto dispersor

El problema abordado en este capítulo consiste en la predicción de la presión acústica en cualquier punto del espacio que rodea a un obstáculo (objeto dispersor) sobre el cual incide una onda acústica. Dicho proble-

ma, se puede caracterizar mediante uno equivalente resoluble empleando la versión integral de la ecuación de Helmholtz y las condiciones de contorno adecuadas [122]. Desde un punto de vista algorítmico, la resolución de este problema que se lleva a cabo en esta Tesis consta de dos etapas bien diferenciadas:

1. Cálculo, sobre la superficie del obstáculo (notada como S), de la presión acústica y de su derivada respecto de la dirección normal a S.

2. Partiendo del campo acústico (presión y su derivada respecto de la dirección normal) obtenido en la etapa anterior, cálculo de la presión acústica en cualquier punto del espacio que rodea al obstáculo.

El campo acústico sobre S se aproxima mediante un desarrollo en serie de N funciones base. El objetivo del primero de los pasos mencionados anteriormente consiste en obtener los N coeficientes asociados a dicho desarrollo. En esta Tesis, se han considerado funciones base constantes. Asimismo, la superficie del obstáculo se modela como una superficie perfectamente rígida, por lo que la derivada de la presión respecto de la dirección normal se anula sobre S. En el segundo paso, a partir de los N coeficientes que describen la presión (y su derivada) sobre S, se calcula la presión en un conjunto de M puntos situados en el espacio que rodea al obstáculo (puntos de observación).

La resolución del primer paso, que implica resolver un sistema lineal de N ecuaciones con N incógnitas (con N proporcional al tamaño acústico, es decir en longitudes de onda, del obstáculo), se aborda en los capítulos dedicados al FMM (capítulo 2) y al FMM-FFT (capítulo 3). La resolución del segundo paso, en la cual se centra este capítulo, requiere llevar a cabo un producto matriz-vector cuyo coste es $\mathcal{O}(NM)$, siendo M del mismo orden (o incluso mayor) que N.

4.1.1. Cálculo de la presión total

La presión total, P, en la superficie del obstáculo (S) y en cualquier punto del espacio que lo rodea puede expresarse como el resultado de la

superposición del campo incidente (campo que proviene la fuente real), P^i, y de la perturbación que representa el campo dispersado, P^s. De esta forma, notando como $\mathbf{r}$ al vector de posición de un punto cualquiera del espacio exterior, se tiene que:

$$P(\mathbf{r}) \;=\; P^i(\mathbf{r}) \;+\; P^s(\mathbf{r}). \tag{4.1}$$

El valor del campo incidente se considera conocido (se supone que la fuente de ruido está correctamente caracterizada), por lo que el problema fundamental consiste en obtener el valor del campo dispersado, P^s, en aquellos puntos del espacio que se consideren de interés. El conjunto de puntos, $\mathbf{r}$, sobre los que se obtendrá la presión total, constituye el dominio de observación.

4.1.2. Cálculo de la presión dispersada

Se parte de que los valores de presión P son conocidos para cualquier punto $\mathbf{r}'$ situado en S (resultado del primer paso del problema). Así, siguiendo el procedimiento descrito en [122], la presión dispersada en cualquier punto $\mathbf{r}_j$ externo a S ($P^s(\mathbf{r}_j)$) puede calcularse usando la siguiente ecuación:

$$P^s(\mathbf{r}_j) \;=\; \int_S P(\mathbf{r}') \frac{\partial G(\mathbf{r}_j, \mathbf{r}')}{\partial n(\mathbf{r}')} \mathrm{d}S(\mathbf{r}'), \tag{4.2}$$

con $j \in \{0, 1, \ldots, M-1\}$ y donde $G(\mathbf{r}_j, \mathbf{r}')$ representa la función de Green en espacio libre, que viene dada por:

$$G(\mathbf{r}_j, \mathbf{r}') \;=\; \frac{e^{-j\kappa|\mathbf{r}_j - \mathbf{r}'|}}{4\pi\,|\mathbf{r}_j - \mathbf{r}'|}\;, \tag{4.3}$$

donde κ representa el número de onda.

Por su parte, la presión acústica evaluada sobre cualquier punto $\mathbf{r}' \in S$ se puede aproximar mediante una expansión en serie de N funciones base, de la forma:

$$P(\mathbf{r}') \;\approx\; \sum_{i=0}^{N-1} p_i f_i(\mathbf{r}'), \quad \mathbf{r}' \in S, \tag{4.4}$$

donde p_i es el i-ésimo coeficiente de la expansión y $f_i(\mathbf{r}')$ representa la función base i-ésima evaluada en $\mathbf{r}' \in S$.

Considerando la expansión dada en la ecuación (4.4), la presión dispersada también puede formularse de la siguiente manera:

$$P^s(\mathbf{r}_j) \approx \sum_{i=0}^{N-1} P_i^s(\mathbf{r}_j), \tag{4.5}$$

donde $P_i^s(\mathbf{r}_j)$ es el campo dispersado por la función base $f_i(\mathbf{r})$ sobre el punto $\mathbf{r}_j$.

Si además, como sucede en el caso de esta Tesis, se utilizan funciones base constantes [122], $P_i^s(\mathbf{r}_j)$ puede expresarse de la siguiente manera:

$$P_i^s(\mathbf{r}_j) = \int_{S_i} p_i \frac{\partial G(\mathbf{r}_j, \mathbf{r}')}{\partial n(\mathbf{r}')} \mathrm{d}S(\mathbf{r}'), \tag{4.6}$$

donde S_i representa el área de la faceta de S correspondiente a la i-ésima función base de las N facetas en las que se discretiza la superficie del obstáculo (dominio fuente).

De esta manera, partiendo de la expresión (4.6), el campo dispersado P^s en los puntos pertenecientes al dominio de observación puede representarse mediante el siguiente producto matriz-vector:

$$p^s = K \cdot p, \tag{4.7}$$

donde los elementos del vector p^s son de la forma $p_j^s = P^s(\mathbf{r}_j)$ (ver ecuación (4.5)), los elementos de la matriz K vienen dados por la expresión $\int_{S_i} \frac{\partial G(\mathbf{r}_j, \mathbf{r}')}{\partial n(\mathbf{r}')} \mathrm{d}S(\mathbf{r}')$, y los elementos del vector p se corresponden con los coeficientes p_i (ver ecuaciones (4.4) y (4.6)).

La matriz K depende únicamente del dominio fuente ($\mathbf{r}' \in S_i$, con $i \in \{0, 1, \ldots, N-1\}$), del dominio de observación ($\mathbf{r}_j$ con $j \in \{0, 1, \ldots, M-1\}$), y de la frecuencia acústica. Por tanto, si todo ello se mantiene constante, se tiene que la matriz K no varía aunque se modifique el número y la posición de las fuentes de ruido. Teniendo en cuenta lo anterior, la ecuación (4.7) puede transformarse en un producto matricial como el mostrado

a continuación:

$$P^s = K \cdot P, \tag{4.8}$$

donde las diferentes columnas de P y P^s contienen los datos correspondientes a cada una de las N_c configuraciones diferentes de las fuentes reales de ruido que se desean analizar de forma simultánea.

4.2. Paralelización para sistemas convencionales y heterogéneos

Los diferentes algoritmos que se presentan en este capítulo complementan a los presentados en capítulos anteriores con el objetivo de obtener una solución completa para el problema de dispersión acústica. Las soluciones diseñadas permiten alcanzar una alta eficiencia computacional y están preparadas para trabajar con diferentes microarquitecturas: procesadores convencionales (CPU), procesadores gráficos (GPU), y coprocesadores del tipo Xeon Phi.

En el subapartado 4.2.1, se mencionan las decisiones de diseño más importantes. Posteriormente, en los subapartados comprendidos entre los puntos 4.2.2 y 4.2.5, se presentan y analizan de forma detallada los diferentes algoritmos diseñados, que implementan diversas técnicas para las diferentes arquitecturas utilizadas.

4.2.1. Decisiones de diseño de la implementación propia

La primera de las decisiones que se han tomado a la hora de diseñar las soluciones propuestas en este capítulo tiene relación con el tipo de sistemas a los que van dirigidas dichas soluciones. Los algoritmos presentados están enfocados a sistemas de memoria compartida, en concreto a sistemas basados en un único nodo con una o varias CPU y cualquier combinación de GPU (compatible con CUDA) y aceleradores de tipo Xeon Phi. Si bien en esta Tesis no se ha implementado una solución orientada a sistemas de memoria distribuida para resolver este paso del problema de dispersión

acústica, todos los algoritmos presentados podrían extenderse de forma sencilla para su uso en dicho tipo de sistemas realizando las siguientes modificaciones:

- Aplicar una división del dominio sobre el problema original de forma que cada proceso paralelo se encargase del cálculo de la presión dispersada, P^s, en varios puntos de observación (división por bloques de filas).

- Distribuir la matriz P (vector en el caso $N_c = 1$) entre todos los procesos usando una comunicación de tipo *broadcast*, en la que el proceso raíz enviase una copia completa de P al resto de procesos.

- Permitir que cada proceso calcule únicamente las filas de P^s que le corresponden. Gracias al particionado utilizado, el cálculo de K no se replicaría en los diferentes procesos, lo que resultaría en una solución escalable.

- Recopilar la solución completa usando una comunicación de tipo *gather*, de forma que cada proceso enviase su contribución al proceso raíz para que fuera capaz de componer la matriz P^s (vector en el caso $N_c = 1$) completa.

Por su parte, la gestión del cálculo y del almacenamiento de la matriz K supone una de las decisiones de más peso en las diferentes soluciones presentadas. Si se tiene en cuenta que K es una matriz de dimensiones $N \times M$ y que tanto N como M pueden ser del orden de millones incluso para problemas de tamaño moderado, el cálculo y almacenamiento de dicha matriz completa en memoria principal puede resultar inviable. Para solventar este problema, los elementos de K se calculan bajo demanda y sobre la marcha. De esta forma, sólo los elementos que estén en uso en un momento determinado estarán ocupando espacio en memoria principal, reduciendo así de forma drástica los requisitos de memoria de la solución. Además, se sigue la máxima de no recalcular elementos de K, ya que el cálculo de un elemento, que sigue la expresión $K_{j,i} = \int_{S_i} \frac{\partial G(\mathbf{r}_j,\mathbf{r}')}{\partial n(\mathbf{r}')} \mathrm{d}S(\mathbf{r}')$, es una tarea costosa.

En lo relativo a las comunicaciones, como se deriva de lo explicado en el primer párrafo, las soluciones presentadas no cuentan con diferentes procesos paralelos que deban comunicarse entre sí por paso de mensajes. Por tanto, las únicas comunicaciones que se contemplan en este capítulo son aquellas que se llevan a cabo entre el *host* (CPU) y los diferentes aceleradores hardware (GPU y Xeon Phi).

Para implementar la versión para Xeon Phi de los diferentes algoritmos presentados se ha utilizado el modelo de programación conocido como CAO (*Compiler-assisted Offload*) [114]. En dicho modelo, las transferencias de datos se pueden controlar de manera explícita mediante el uso de directivas o *pragmas*, evitando el uso de memoria compartida entre la CPU y el propio Xeon Phi. El uso de *pragmas* específicos para Xeon Phi junto con construcciones de OpenMP permite paralelizar algoritmos de forma sencilla obteniendo, además, un rendimiento comparable al del resto de modelos disponibles para esta arquitectura [114].

Otra de las decisiones de diseño importantes, que afecta especialmente a la implementación heterogénea (ver apartado 4.2.5), tiene relación con el trabajo asignado a las CPU cuando también se emplean aceleradores hardware. En la implementación heterogénea presentada en este capítulo, una GPU y un Xeon Phi se encargan de llevar a cabo los cálculos más pesados, mientras que las CPU se utilizan únicamente para gestionar los aceleradores. Esta decisión se debe a que, en comparación con una GPU o un Xeon Phi, las CPU aportan poco rendimiento extra [118]. Además, en caso de utilizarse también las CPU para tareas de cálculo, habría que limitar el número de hilos para evitar una sobrecarga de las CPU como consecuencia de realizar dos tareas diferentes de forma simultánea (cálculo y gestión de los aceleradores), lo que minimizaría aún más la aportación de las CPU a las tareas de cálculo.

También en el caso de la implementación heterogénea, se ha decidido tener en cuenta el rendimiento de los diferentes aceleradores con el objetivo de equilibrar la carga de trabajo entre ellos de forma que, a mayor rendimiento de un dispositivo, mayor carga de trabajo se le asigna. Considerando que se dispone de d dispositivos aceleradores y notando al tiempo de ejecución asociado al dispositivo acelerador i-ésimo como T_i y su carga

de trabajo asignada, en tanto por uno, como W_i, se puede escribir:

$$W_i = \frac{(T_i)^{-1}}{\sum_{j=0}^{d-1} (T_j)^{-1}}. \quad (4.9)$$

Con el fin de obtener W_i, se recurre a la ejecución de un problema de tamaño reducido, que servirá de referencia, calculando el rendimiento individual de cada acelerador como el inverso de su tiempo de ejecución asociado, $(T_j)^{-1}$. En el apartado 4.2.5, se particularizará la expresión (4.9) para el caso de dos dispositivos aceleradores, concretamente un Xeon Phi y una GPU Tesla.

Finalmente, y al igual que en las implementaciones del FMM y del FMM-FFT presentadas anteriormente en esta Tesis, las matrices empleadas en las diferentes versiones de esta herramienta también están almacenadas en memoria de forma unidimensional. En este caso, la decisión de almacenar por filas o por columnas vendrá condicionada por las librerías a utilizar (MKL [140] o cuBLAS [141]), trabajando siempre según las recomendaciones de los diseñadores de las mismas para obtener un mayor rendimiento.

4.2.2. Técnica basada en producto matriz-vector (bMVP)

La primera de las técnicas presentadas en este capítulo se basa en el cálculo de un único producto matriz-vector que permite obtener la presión dispersada para un problema determinado (dominio fuente, dominio de observación y frecuencia de análisis permanecen invariantes) con una única configuración de las fuentes de ruido (número y posición de las mismas).

Esta técnica, que se ha denominado bMVP (*baseline* MVP), se utiliza como punto de partida en la implementación y como referencia comparativa para las siguientes técnicas, basadas en múltiples productos matriz-vector o en productos matriciales.

En el algoritmo 4.1, se muestra cómo se ha llevado a cabo la implementación de esta técnica para procesadores convencionales y para aceleradores

Algoritmo 4.1 Solución basada en un único producto matriz-vector (bMVP) para CPU multinúcleo y coprocesadores tipo Xeon Phi usando OpenMP.

Entrada: M, N, p, $\mathbf{r}$, $\mathbf{r}'$, $n(\mathbf{r}')$, S.
Salida: p^s contiene la presión dispersada en cada punto de observación.
1: {Directiva específica para Xeon Phi}
2: #pragma offload target(mic) in(p: length(N)) \
   in($\mathbf{r}'$,$n(\mathbf{r}')$,S: length(N)) in($\mathbf{r}$: length(M)) out(p^s: length(M))
3: #pragma omp parallel for
4: {Para cada punto de observación}
5: **para** $j = 0$ **hasta** $M-1$ **hacer**
6: $\quad p^s_j = 0$ {inicialización del vector p^s}
7: $\quad$ {Para cada elemento fuente}
8: $\quad$ **para** $i = 0$ **hasta** $N-1$ **hacer**
9: $\quad\quad k = \int_{S_i} \frac{\partial G(\mathbf{r}_j,\mathbf{r}')}{\partial n(\mathbf{r}')} \mathrm{d}S(\mathbf{r}')$ {cálculo de un único elemento de K}
10: $\quad\quad p^s_j = p^s_j + k \cdot p_i$ {MVP}
11: $\quad$ **fin para**
12: **fin para**

Xeon Phi. En ambos casos, se ha recurrido al uso de OpenMP para paralelizar los cálculos asociados al problema, utilizando una directiva del tipo `parallel for` reflejada en la línea 3. De esta forma, cada hilo se encarga de calcular una posición del vector solución, p^s, de forma cíclica hasta obtener la presión dispersada en los M puntos que componen el dominio de observación. Además, como se observa en la línea 9, la matriz K no se almacena de forma completa en ningún momento, sino que sus elementos se calculan una única vez, bajo demanda, cuando son necesarios.

La implementación para los coprocesadores Xeon Phi se basa en la implementación para CPU, a la que se le añade una directiva específica, `#pragma offload target(mic)` (ver algoritmo 4.1, línea 2), que permite descargar a la CPU de los cálculos, los cuales pasan a ser asignados al acelerador. Para ello, además, se indican los arrays que deben ser transferidos desde la memoria de la CPU hacia la memoria del acelerador, `in(...)`, y el array donde se almacenarán los resultados que deberá ser transferido de la memoria del acelerador a la memoria de la CPU al finalizar la ejecución de los cálculos descargados, `out(...)`. En relación a la transferencia de da-

tos entre la CPU y el acelerador, cabe mencionar que dichas transferencias tienen un coste $\mathcal{O}(\text{máx}\{N, M\})$, como se puede deducir del pseudocódigo, mientras que la computación asociada a este problema es $\mathcal{O}(NM)$, por lo que la sobrecarga debida a la copia de datos es despreciable para los problemas que se abordan en esta Tesis, donde N y M pueden ser, ambos, del orden de millones.

Sin embargo, la implementación para GPU no parte de la implementación para CPU o Xeon Phi presentada en los párrafos anteriores. La división del problema original llevada a cabo en las implementaciones orientadas a dispositivos de tipo `x86` no es suficiente para aprovechar todo el potencial de las GPU actuales, que disponen de miles de núcleos y que son capaces de manejar millones de hilos de forma simultánea. Por tanto, el enfoque de la implementación para procesadores gráficos debe permitir una división del problema original en un mayor número de subproblemas de menor tamaño, adaptándose a la filosofía de paralelismo de grano fino presente en CUDA.

Algoritmo 4.2 Solución basada en un único producto matriz-vector (bMVP) para GPU usando CUDA.

Entrada: $dimBloque$, $dimGrid$, M, N, p, $\mathbf{r}$, $\mathbf{r}'$, $n(\mathbf{r}')$, S.
Salida: p^s contiene la presión dispersada en cada punto de observación.
1: {Para cada punto de observación}
2: **para** $j = IdBloque$ **hasta** $M-1$ **paso** $dimGrid$ **hacer**
3: $p^s_{j,IdHilo} = 0$ {inicialización del bloque de memoria compartida}
4: {Para cada elemento fuente}
5: **para** $i = IdHilo$ **hasta** $N-1$ **paso** $dimBloque$ **hacer**
6: $k = \int_{S_i} \frac{\partial G(\mathbf{r}_j, \mathbf{r}')}{\partial n(\mathbf{r}')} \mathrm{d}S(\mathbf{r}')$ {cálculo de un único elemento de K}
7: $p^s_{j,IdHilo} = p^s_{j,IdHilo} + k \cdot p_i$ {MVP}
8: **fin para**
9: {Suma por reducción en memoria compartida}
10: $p^s_j = \sum_{i=0}^{dimBloque-1} p^s_{j,i}$
11: **fin para**

El pseudocódigo mostrado en el algoritmo 4.2 sintetiza los detalles de la implementación de esta técnica para procesadores gráficos usando CUDA. Las filas de la matriz K (puntos de observación) se asignan a los diferentes bloques CUDA, mientras que las columnas de dicha matriz (elementos

fuente) se asignan a los hilos pertenecientes a un mismo bloque. Puesto que el número de filas (M) y el número de columnas (N) pueden ser mucho mayores que el número de bloques ($dimGrid$) y el número de hilos por bloque ($dimBloque$) respectivamente, tanto la asignación de filas a bloques como la de columnas a hilos se llevan a cabo de forma cíclica (líneas 2 y 5 del algoritmo 4.2). El proceso que se sigue en esta versión del algoritmo puede dividirse conceptualmente en tres partes bien diferenciadas que se repiten de forma iterativa hasta obtener el resultado completo del producto matriz-vector:

1. Cada hilo calcula el elemento de la matriz K que necesita en un momento dado (k) sobre la marcha. Un hilo determinado sólo debe tratar los elementos de K que le corresponden, dependiendo del bloque al que pertenezca y de su posición dentro de dicho bloque. Además, el almacenamiento de un elemento k es temporal (sólo se almacena mientras sea necesario) y se utiliza para ello un registro (la memoria más rápida en una GPU). Esta parte se corresponde con la línea 6 del algoritmo 4.2.

2. Cada hilo multiplica el elemento k previamente calculado por el elemento de p correspondiente y acumula dicho producto en un bloque de memoria compartida. Esta parte se corresponde con la línea 7 del pseudocódigo 4.2.

3. Todos los hilos de un mismo bloque colaboran para llevar a cabo una suma por reducción paralelizada, obteniendo así una posición del array en el que se almacena el resultado del producto matriz-vector (p^s). Esta última parte se corresponde con la línea 10 del algoritmo 4.2.

La figura 4.1 representa gráficamente cómo se divide el trabajo descrito en los puntos anteriores entre los diferentes bloques e hilos CUDA. Puede verse que todos los hilos de un mismo bloque deben cooperar para obtener un valor del vector p^s. Por simplicidad en la representación esquemática, en la figura 4.1 se asume que N es múltiplo del número de hilos por bloque ($dimBloque$) y que M es múltiplo del número de bloques ($dimGrid$).

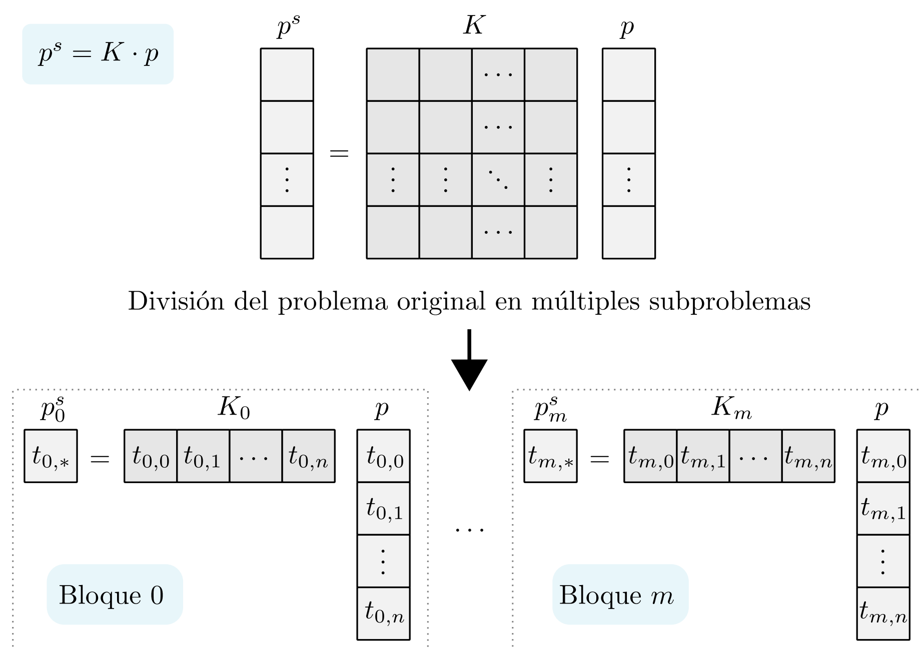


Figura 4.1: Paralelización del cálculo $p^s = K \cdot p$ usando CUDA. Por simplicidad y compacidad, $n = dimBloque - 1$ y $m = dimGrid - 1$. $t_{i,j}$ representa el hilo j-ésimo dentro del bloque i.

4.2.3. Técnica basada en múltiples productos matriz-vector (mMVP)

Como se ha comentado anteriormente en este capítulo, la matriz K que relaciona los elementos fuente y los puntos de observación se calcula siguiendo la expresión $K_{j,i} = \int_{S_i} \frac{\partial G(\mathbf{r}_j, \mathbf{r}')}{\partial n(\mathbf{r}')} \mathrm{d}S(\mathbf{r}')$, para el elemento i-ésimo y el punto de observación j-ésimo. Por tanto, si el dominio fuente, el dominio de observación, y la frecuencia de las fuentes de ruido permanecen fijos, se tiene que la matriz K no sufre variaciones, independientemente del número y de la posición de las fuentes de ruido.

Si se aprovecha lo explicado en el párrafo anterior, la expresión $p^s = K \cdot p$ puede transformarse en un producto matricial de la forma $P^s = K \cdot P$, donde las diferentes columnas de P y P^s almacenan los datos correspondientes

a cada una de las configuraciones para las fuentes de ruido. La figura 4.2 muestra una representación de cómo es el proceso para obtener una matriz P^s a partir del producto de K y P, llevado a cabo como un conjunto de productos matriz-vector.

Figura 4.2: Obtención de la matriz P^S a partir de K y P usando múltiples productos matriz-vector (técnica mMVP). El problema representado en la figura consta de N elementos fuente, M puntos de observación y N_c configuraciones diferentes de las fuentes de ruido (debe tenerse en cuenta que $N \gg N_c$ y $M \gg N_c$).

El algoritmo 4.3 permite analizar los detalles de la implementación de la técnica mMVP para el caso de procesadores convencionales y para el caso de coprocesadores de tipo Xeon Phi. Al igual que en el caso de la técnica bMVP mostrada en el apartado anterior, la paralelización se ha llevado a cabo usando OpenMP, tanto para el código orientado a CPU como para el código ejecutable sobre aceleradores Xeon Phi.

En la línea 3 del algoritmo 4.3 puede verse la directiva `parallel for` de OpenMP que se utiliza para dividir el problema inicial en múltiples subproblemas resolubles en paralelo. La división realizada permite que cada hilo se encargue de calcular una fila de la matriz P^s, repitiéndose este proceso de forma cíclica hasta que se obtiene la presión dispersada en los M puntos en los que se divide el dominio de observación. Por tanto, en cada

Algoritmo 4.3 Solución basada en múltiples productos matriz-vector (mMVP) para CPU multinúcleo y coprocesadores tipo Xeon Phi usando OpenMP.

Entrada: M, N, N_c, P, $\mathbf{r}$, $\mathbf{r}'$, $n(\mathbf{r}')$, S.

Salida: P^s contiene la presión dispersada en cada punto de observación para todas las configuraciones de las fuentes de ruido.

1: {Directiva específica para Xeon Phi}
2: #pragma offload target(mic) in(P: length($N \cdot N_c$)) \ in($\mathbf{r}'$,$n(\mathbf{r}')$,S: length(N)) in($\mathbf{r}$: length(M)) out(P^s: length($M \cdot N_c$))
3: #pragma omp parallel for
4: {Para cada punto de observación}
5: **para** $j = 0$ **hasta** $M-1$ **hacer**
6:    {Para cada configuración de las fuentes de ruido}
7:    **para** $l = 0$ **hasta** N_c-1 **hacer**
8:       $p_l^s = 0$ {inicialización de un vector fila de P^s}
9:    **fin para**
10:    {Para cada elemento fuente}
11:    **para** $i = 0$ **hasta** $N-1$ **hacer**
12:       $k = \int_{S_i} \frac{\partial G(\mathbf{r}_j,\mathbf{r}')}{\partial n(\mathbf{r}')} \mathrm{d}S(\mathbf{r}')$ {cálculo de un único elemento de K}
13:       {Para cada configuración de las fuentes de ruido}
14:       **para** $l = 0$ **hasta** N_c-1 **hacer**
15:          $p_l^s = p_l^s + k \cdot P_{i,l}$ {MVP}
16:       **fin para**
17:    **fin para**
18:    {Para cada configuración de las fuentes de ruido}
19:    **para** $l = 0$ **hasta** N_c-1 **hacer**
20:       $P_{j,l}^s = p_l^s$ {almacenamiento de una fila de P^s}
21:    **fin para**
22: **fin para**

una de las iteraciones mencionadas, un hilo determinado calcula la presión dispersada sobre un único punto de observación para las N_c configuraciones de las fuentes de ruido. En la técnica mMVP también se logra evitar el almacenamiento de la matriz K de forma completa, ya que sus elementos se calculan únicamente una vez y sobre la marcha, justo en el momento en el que son necesarios, aprovechando cada elemento $K_{j,i}$ para los N_c problemas que se resuelven de forma simultánea (ver líneas 10 a 17 del algoritmo 4.3).

La implementación orientada a dispositivos Xeon Phi sigue la misma filosofía que la versión para CPU. Puesto que se utiliza el modelo de programación CAO, simplemente es necesario incorporar una directiva específica que permita indicarle al compilador que la ejecución del algoritmo se desea *descargar* sobre el coprocesador (ver algoritmo 4.3, línea 2). Dicha directiva permite, además, indicar las porciones de memoria que deben ser transferidas de la CPU al acelerador antes de comenzar la ejecución (parámetros de tipo `in`) y aquellas que deben ser transferidas de la memoria del Xeon Phi a la CPU al finalizar la ejecución pues contienen los resultados (parámetro de tipo `out`). En la técnica mMVP, puesto que se trabaja con N_c configuraciones de las fuentes de ruido de forma simultánea, P y P^s pasan a ser matrices linealizadas por filas y almacenadas en arrays de tamaño $N \cdot N_c$ y $M \cdot N_c$, respectivamente. Por tanto, el coste de las transferencias de datos es $\mathcal{O}(\text{máx}\{N \cdot N_c,\ M \cdot N_c\})$, mientras que los cálculos asociados a este problema tienen un coste $\mathcal{O}(NMN_c)$ (además, debe recordarse que $N \gg N_c$ y $M \gg N_c$). De nuevo, la relación cálculo/transferencia de datos sigue siendo muy favorable, resultando en una baja sobrecarga para los problemas analizados en esta Tesis (ver resultados en el apartado 6.4).

Por su parte, la implementación para GPU presenta diferencias notables respecto de la implementación para CPU y Xeon Phi presentada en los párrafos anteriores. Puesto que en el modelo de paralelismo de CUDA, conocido como SIMT (*Single Instruction, Multiple Threads*) [112], se trabaja con un grano muy fino y teniendo en cuenta, además, que la memoria asociada a un procesador gráfico es muy diferente a la que se encuentra en un procesador convencional, la técnica mMVP se debe rediseñar si se quieren aprovechar las capacidades de las GPU.

Como se muestra en el algoritmo 4.4, en la solución mMVP para GPU, cada una de las filas de la matriz K (puntos de observación) se asigna a un bloque CUDA, mientras que las diferentes columnas para una fila dada (elementos fuente) se asignan a los hilos que pertenecen a un mismo bloque. La asignación de filas a bloques y la asignación de columnas a hilos son cíclicas, lo que permite abordar sin problemas todos aquellos casos en los que $M > \mathit{dimGrid}$ o $N > \mathit{dimBloque}$ (líneas 2 y 7 del algoritmo 4.4). Por tanto, la división del problema inicial en relación al cálculo de los elementos de K es idéntica a la presentada en la figura 4.2 para el caso

Algoritmo 4.4 Solución basada en múltiples productos matriz-vector (mMVP) para GPU usando CUDA.

Entrada: $dimBloque$, $dimGrid$, M, N, N_c, P, $\mathbf{r}$, $\mathbf{r}'$, $n(\mathbf{r}')$, S.
Salida: P^s contiene la presión dispersada en cada punto de observación para todas las configuraciones de las fuentes de ruido.

1: {Para cada punto de observación}
2: **para** $j = IdBloque$ **hasta** $M-1$ **paso** $dimGrid$ **hacer**
3:   **para** $l = 0$ **hasta** N_c-1 **hacer**
4:     $p^s_{l,IdHilo} = 0$ {inicialización del bloque de memoria compartida}
5:   **fin para**
6:   {Para cada elemento fuente}
7:   **para** $i = IdHilo$ **hasta** $N-1$ **paso** $dimBloque$ **hacer**
8:     $k = \int_{S_i} \frac{\partial G(\mathbf{r}_j,\mathbf{r}')}{\partial n(\mathbf{r}')} \mathrm{d}S(\mathbf{r}')$ {cálculo de un único elemento de K}
9:     {Para cada configuración de las fuentes de ruido}
10:     **para** $l = 0$ **hasta** N_c-1 **hacer**
11:       $p^s_{l,IdHilo} = p^s_{l,IdHilo} + k \cdot P_{i,l}$ {MVP}
12:     **fin para**
13:   **fin para**
14:   {Para cada configuración de las fuentes de ruido}
15:   **para** $l = 0$ **hasta** N_c-1 **hacer**
16:     $P^s_{j,l} = \sum_{i=0}^{dimBloque-1} p^s_{l,i}$ {suma por reducción en memoria compartida}
17:   **fin para**
18: **fin para**

de la técnica bMVP. Sin embargo, debido a que se tratan N_c problemas de forma simultánea, el resto de operaciones difieren respecto de la técnica basada en un único producto matriz-vector. En la técnica mMVP para GPU, los pasos a repetir hasta obtener el resultado final son los siguientes:

1. Cada hilo calcula sobre la marcha el elemento de la matriz K que necesita en un determinado momento. El almacenamiento de un elemento k es temporal y se utiliza para ello un registro. Este paso se corresponde con la línea 8 del algoritmo 4.4 y es idéntico al primer paso de la técnica bMVP.

2. Cada hilo multiplica su elemento k por los N_c elementos de la fila i-ésima de P y acumula los resultados parciales en un bloque de

memoria compartida. Este paso se corresponde con las líneas 10 a 12 del pseudocódigo 4.4.

3. Todos los hilos de un mismo bloque colaboran para llevar a cabo N_c sumas por reducción, lo que permite obtener los N_c elementos de la fila j-ésima de la matriz P^s (un elemento por cada una de las N_c configuraciones de las fuentes de ruido). Este último paso se corresponde con las líneas 15 a 17 del algoritmo 4.4.

4.2.4. Técnicas basadas en productos matriciales

Una vez analizadas las técnicas basadas en el producto matriz-vector (bMVP y mMVP), a continuación se estudian dos técnicas diferentes basadas en la multiplicación matricial. Por su propio diseño, las técnicas basadas en productos matriciales sólo cobran sentido en aquellos casos en los que se desea analizar varias configuraciones diferentes para las fuentes de ruido, es decir, cuando se cumple que $N_c > 1$.

Las dos técnicas presentadas en este apartado se basan en la multiplicación matricial por bloques, con el objetivo de aprovechar el principio de localidad [132] para lograr unos accesos a memoria más eficientes. Dichas técnicas son el resultado de aplicar dos enfoques completamente diferentes a la hora de abordar el problema: cada hilo resuelve de forma independiente una pequeña parte del problema (*SmallBlocks*) o todos los hilos trabajan de forma conjunta para resolver cada uno de los pasos en los que se puede dividir el problema original (*BigBlocks*).

4.2.4.1. *SmallBlocks*

La primera de las técnicas basadas en el producto matricial se ha denominado *SmallBlocks* y está orientada específicamente a procesadores convencionales y a aceleradores Xeon Phi. Su nombre se debe al tamaño reducido de los bloques en los que se divide la carga de trabajo asignada a cada hilo (bloques de dimensiones no superiores a 128×128).

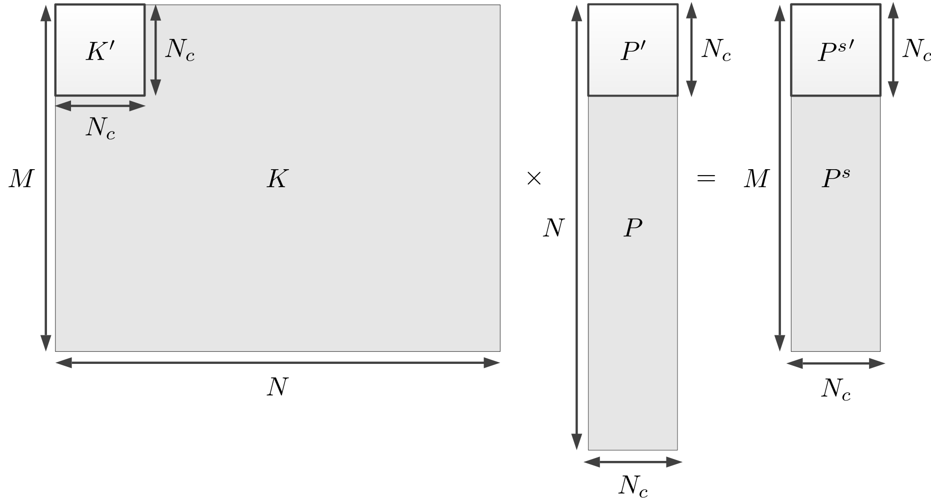


Figura 4.3: Obtención de la matriz P^S a partir de K y P mediante un producto matricial por bloques (técnica *SmallBlocks*). El problema consta de N elementos fuente, M puntos de observación y N_c configuraciones para las fuentes de ruido. Los bloques utilizados en el ejemplo son de tamaño $N_c \times N_c$.

La figura 4.3 representa de forma esquemática cómo se ha rediseñado el problema original para convertirlo en una multiplicación matricial por bloques. En la figura, por simplicidad, los bloques utilizados son de tamaño $N_c \times N_c$, aunque su tamaño puede ser diferente. Si el tamaño de bloque escogido es mayor que $N_c \times N_c$, los bloques P' deben completarse con ceros. De igual forma, si N, M o N_c no son múltiplos del tamaño de bloque escogido, los bloques de los extremos deben completarse con ceros.

Puesto que no es viable almacenar la matriz K de forma completa en memoria principal, ya que N y M pueden ser del orden de millones, dicha matriz debe calcularse por partes sobre la marcha. Esta particularidad elimina la posibilidad de obtener el resultado de $K \cdot P$ de forma directa (en una única operación) recurriendo, por ejemplo, al uso de una librería matemática que implemente rutinas BLAS [142] (*Basic Linear Algebra Subprograms*) de nivel 3 (operaciones entre matrices). Por tanto, para poder usar librerías que permitan llevar a cabo el producto matricial de forma

eficiente, es necesario generar bloques de un tamaño conveniente (K' y P' en la figura 4.3). De esta forma, en varios pasos, se van obteniendo cada uno de los bloques $P^{s\prime}$ que componen la matriz resultado P^s (ver figura 4.3).

Algoritmo 4.5 Solución *SmallBlocks* para CPU multinúcleo y coprocesadores tipo Xeon Phi usando OpenMP y MKL.

Entrada: bs, M, N, N_c, P, $\mathbf{r}$, $\mathbf{r}'$, $n(\mathbf{r}')$, S.

Salida: P^s contiene la presión dispersada en cada punto de observación para todas las configuraciones de las fuentes de ruido.

1: {Directiva específica para Xeon Phi}
2: #pragma offload target(mic) in(P: length($N \cdot N_c$)) \
in($\mathbf{r}'$,$n(\mathbf{r}')$,S: length(N)) in($\mathbf{r}$: length(M)) out(P^s: length($M \cdot N_c$))
3: #pragma omp parallel for
4: **para** $j = 0$ **hasta** $M-1$ **paso** bs **hacer**
5:   **para** $i = 0$ **hasta** $N-1$ **paso** bs **hacer**
6:     **para** $jj = 0$ **hasta** $bs - 1$ **hacer**
7:       **para** $ii = 0$ **hasta** $bs - 1$ **hacer**
8:         $K'_{jj,ii} = \int_{S_{i+ii}} \frac{\partial G(\mathbf{r}_{j+jj},\mathbf{r}')}{\partial n(\mathbf{r}')} \mathrm{d}S(\mathbf{r}')$ {cálculo de un bloque K'}
9:       **fin para**
10:     **fin para**
11:     **para** $l = 0$ **hasta** N_c-1 **paso** bs **hacer**
12:       **para** $ii = 0$ **hasta** $bs - 1$ **hacer**
13:         **para** $ll = 0$ **hasta** $bs - 1$ **hacer**
14:           $P'_{ii,ll} = P_{i+ii,l+ll}$ {generación de un bloque P'}
15:         **fin para**
16:       **fin para**
17:       {Multiplicación matricial usando BLAS}
18:       cblas_cgemm($\ldots, K', bs, P', bs, \ldots, P^{s\prime}, bs$)
19:       **para** $jj = 0$ **hasta** $bs - 1$ **hacer**
20:         **para** $ll = 0$ **hasta** $bs - 1$ **hacer**
21:           {Almacenamiento de resultados}
22:           $P^s_{j+jj,l+ll} = P^s_{j+jj,l+ll} + P^{s\prime}_{jj,ll}$
23:         **fin para**
24:       **fin para**
25:     **fin para**
26:   **fin para**
27: **fin para**

En el algoritmo 4.5 se muestran los detalles de la implementación de

la técnica *SmallBlocks* para CPU y coprocesadores Xeon Phi. En esta implementación se combina el uso de OpenMP y de una rutina de multiplicación matricial de la librería MKL de Intel [140]. La versión para Xeon Phi sólo presenta un añadido respecto de la versión para CPU, la directiva `#pragma offload target(mic)` (línea 2 del algoritmo 4.5). Los cuatro pasos en los que se divide esta técnica, y que se repiten hasta obtener el resultado final, son los siguientes:

1. Cada hilo crea su propio bloque K', de tamaño $bs \times bs$, calculando los elementos sobre la marcha. Este primer paso queda reflejado en las líneas 6 a 10 del algoritmo 4.5.

2. Cada hilo genera el bloque P' que se corresponde con el bloque K' que acaba de crear. Este paso se muestra en las líneas 12 a 16 del pseudocódigo 4.5.

3. Cada hilo multiplica sus bloques K' y P' de forma eficiente gracias al uso de la rutina `cblas_cgemm` de Intel MKL, y almacena este resultado parcial en el bloque $P^{s\prime}$. Este paso se lleva a cabo en la línea 18 del algoritmo 4.5.

4. Cada hilo acumula el resultado parcial que acaba de obtener, $P^{s\prime}$, en las posiciones adecuadas de la matriz que va a contener los resultados finales, P^s. Este último paso se corresponde con las líneas 19 a 24 del algoritmo 4.5.

Asimismo, el algoritmo 4.5 dispone de un bucle (lineas 11 a 25) que permite tratar correctamente aquellos casos en los que el tamaño de los bloques ($bs \times bs$) es menor que $N_c \times N_c$.

Esta técnica se ha diseñado teniendo en cuenta que el cálculo de los elementos de K es computacionalmente costoso, por lo que cada bloque K' sólo se genera una única vez a lo largo de la ejecución del algoritmo. Por su parte, los bloques P' se generan bajo demanda tantas veces como sea necesario, pero esto no supone ninguna sobrecarga debido a que la matriz P se encuentra previamente calculada por ser uno de los datos de entrada del algoritmo.

En la técnica *SmallBlocks*, cada hilo se encarga de obtener un bloque de P^s realizando los pasos descritos anteriormente. Es decir, cada hilo resuelve de forma independiente una pequeña parte del problema. Esta forma de dividir la carga de trabajo permite usar hilos cuyo ciclo de vida es el mismo que el del propio algoritmo: se crean justo al comienzo del algoritmo (línea 3 del algoritmo 4.5) y se destruyen con la finalización del mismo (línea 27 del pseudocódigo 4.5). Por ello, esta solución presenta una baja sobrecarga relacionada con la gestión de los hilos.

Por último, resulta conveniente mencionar que se ha analizado la aplicación de esta técnica para procesadores gráficos usando CUDA. Sin embargo, los inconvenientes que se enumeran a continuación hacen que la solución *SmallBlocks* no sea adecuada para su uso en GPU:

- Los requisitos relativos a la memoria compartida y al número de registros son muy elevados, lo que afecta negativamente al grado de paralelismo que se puede alcanzar. Para lograr una alta ocupación en una GPU, el grado de paralelismo debe aumentar de forma notable con respecto a las implementaciones para CPU y Xeon Phi, ya que la cantidad de núcleos disponibles es mucho mayor. En los diferentes algoritmos presentados en este capítulo, el número de hilos CUDA puede llegar a ser tres órdenes de magnitud superior al número de hilos OpenMP.

- Para poder utilizar una rutina cuBLAS [141] dentro de un *kernel*, la GPU debe disponer de capacidad para implementar *paralelismo dinámico* [112], ya que implicaría lanzar un nuevo *kernel* desde un *kernel* que se encuentra en ejecución. El paralelismo dinámico introduce cierta sobrecarga y, además, sólo está disponible en dispositivos con arquitectura 3.5 o superior.

- Si no se puede utilizar una librería como cuBLAS para realizar el producto matricial de los diferentes bloques K' y P' por lo mencionado en el punto anterior, la alternativa pasaría por utilizar un producto matricial de diseño propio, lo que reduciría el rendimiento en comparación con las librerías optimizadas a tal efecto.

4.2.4.2. *BigBlocks*

La segunda de las soluciones basadas en el producto matricial por bloques se ha denominado *BigBlocks*. Su nombre se debe al uso de bloques de grandes dimensiones en comparación con los bloques usados en la técnica *SmallBlocks*. En el caso de la técnica *BigBlocks*, debido a su diseño, ha sido viable generar versiones para todos los dispositivos utilizados: CPU, coprocesadores Xeon Phi y GPU.

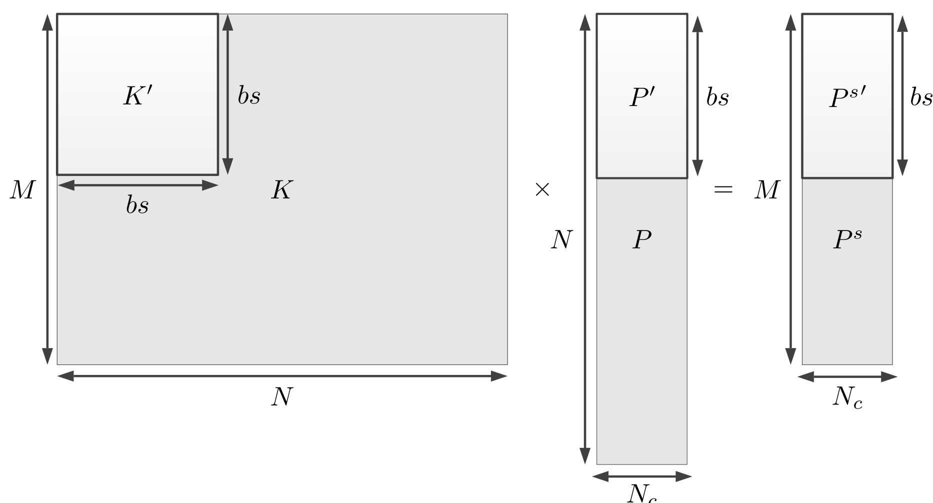


Figura 4.4: Obtención de la matriz P^S a partir de K y P usando un producto matricial por bloques (técnica *BigBlocks*). El ejemplo muestra un problema con N elementos fuente, M puntos de observación y N_c configuraciones para las fuentes de ruido.

En la figura 4.4 se muestra de forma esquemática la división realizada sobre el problema inicial para llevar a cabo una multiplicación matricial por bloques. En este caso, los bloques utilizados son mucho mayores que en la solución *SmallBlocks*, llegando a utilizarse valores para bs hasta tres órdenes de magnitud superiores. También merece la pena destacar que los bloques K' son cuadrados (de dimensión $bs \times bs$), mientras que los bloques P' y $P^{s'}$ son claramente rectangulares (de tamaño $bs \times N_c$ con $bs \gg N_c$). Teniendo en cuenta todo lo anterior, el proceso consiste en generar los

diferentes bloques K' y P' que multiplicados (usando una rutina eficiente para productos matriciales) van a permitir obtener los bloques $P^{s\prime}$ en los que se ha dividido la matriz P^s (ver figura 4.4).

Algoritmo 4.6 Solución *BigBlocks* para CPU multinúcleo y coprocesadores tipo Xeon Phi usando OpenMP y MKL.

Entrada: bs, M, N, N_c, P, $\mathbf{r}$, $\mathbf{r}'$, $n(\mathbf{r}')$, S.
Salida: P^s contiene la presión dispersada en cada punto de observación para todas las configuraciones de las fuentes de ruido.

1: {Directiva específica para Xeon Phi}
2: #pragma offload target(mic) in(P: length($N \cdot N_c$)) \
in($\mathbf{r}'$,$n(\mathbf{r}')$,S: length(N)) in($\mathbf{r}$: length(M)) out(P^s: length($M \cdot N_c$))
3: **para** $j = 0$ **hasta** $M-1$ **paso** bs **hacer**
4:   **para** $i = 0$ **hasta** $N-1$ **paso** bs **hacer**
5:     #pragma omp parallel for
6:     **para** $jj = 0$ **hasta** $bs - 1$ **hacer**
7:       **para** $ii = 0$ **hasta** $bs - 1$ **hacer**
8:         $K'_{jj,ii} = \int_{S_{i+ii}} \frac{\partial G(\mathbf{r}_{j+jj},\mathbf{r}')}{\partial n(\mathbf{r}')} \mathrm{d}S(\mathbf{r}')$ {cálculo de un bloque K'}
9:       **fin para**
10:     **fin para**
11:     #pragma omp parallel for
12:     **para** $ii = 0$ **hasta** $bs - 1$ **hacer**
13:       **para** $ll = 0$ **hasta** $N_c - 1$ **hacer**
14:         $P'_{ii,ll} = P_{i+ii,ll}$ {generación de un bloque P'}
15:       **fin para**
16:     **fin para**
17:     {Multiplicación matricial usando BLAS}
18:     cblas_cgemm($\ldots, K', bs, P', bs, \ldots, P^{s\prime}, bs$)
19:     #pragma omp parallel for
20:     **para** $jj = 0$ **hasta** $bs - 1$ **hacer**
21:       **para** $ll = 0$ **hasta** $N_c - 1$ **hacer**
22:         $P^s_{j+jj,ll} = P^s_{j+jj,ll} + P^{s\prime}_{jj,ll}$ {almacenamiento de resultados}
23:       **fin para**
24:     **fin para**
25:   **fin para**
26: **fin para**

El pseudocódigo mostrado en el algoritmo 4.6 muestra cómo se ha implementado la técnica *BigBlocks* para procesadores convencionales y co-

procesadores Xeon Phi. Nuevamente la paralelización se basa en el uso de OpenMP junto con una rutina de multiplicación matricial de la librería MKL de Intel. Como en los casos anteriores, la implementación para Xeon Phi se obtiene añadiendo la directiva `#pragma offload target(mic)` junto con sus parámetros a la versión para CPU (línea 2 del algoritmo 4.6). Los diferentes pasos en los que puede dividirse esta implementación, y que se repiten de forma cíclica hasta obtener el resultado final, se describen a continuación:

1. Todos los hilos generados trabajan de forma conjunta para crear un gran bloque K', calculando sobre la marcha los diferentes elementos $K'_{jj,ii}$. Este primer paso se corresponde con las líneas 5 a 10 del algoritmo 4.6.

2. Todos los hilos colaboran para generar el bloque P' que se corresponde con el bloque K' anteriormente creado. Este paso se lleva a cabo en las líneas 11 a 16 del pseudocódigo 4.6.

3. Se realiza una única llamada a la rutina `cblas_cgemm` (en su implementación multihilo) de Intel MKL para multiplicar los bloques K' y P' de forma eficiente. El resultado parcial se almacena en el bloque $P^{s\prime}$ creado a tal efecto. Este paso queda reflejado en la línea 18 del algoritmo 4.6.

4. Todos los hilos trabajan en paralelo para acumular el resultado parcial que se acaba de obtener, $P^{s\prime}$, en las posiciones adecuadas de la matriz P^s. Este paso final se aborda en las líneas 19 a 24 del algoritmo 4.6.

Al igual que sucede en la técnica *SmallBlocks*, puede comprobarse que en este caso también se evita recalcular elementos de K, ya que cada bloque K' sólo se genera una única vez.

En la solución de tipo *BigBlocks*, todos los hilos trabajan de forma conjunta en los diferentes pasos llevados a cabo para obtener cada uno de los bloques $P^{s\prime}$. Esta forma de dividir y paralelizar el problema permite trabajar con bloques muy grandes, lo que mejora drásticamente el rendimiento obtenido con la rutina para la multiplicación matricial. Sin embargo, esta

técnica tiene un inconveniente: presenta una clara sobrecarga relacionada con la creación y destrucción de los hilos OpenMP. Como puede observarse en las líneas 5, 11, 18 y 19 del algoritmo 4.6, los hilos se crean (e igualmente se destruyen) tantas veces como se ejecuten los bucles presentes en las líneas 3 y 4. Por este motivo, es de suma importancia que los bloques generados sean de grandes dimensiones, lo que va a permitir resolver el problema con un número reducido de pasos.

Algoritmo 4.7 Solución *BigBlocks* para GPU usando CUDA (código ejecutado en el *host*).

Entrada: bs, M, N, N_c, P, $\mathbf{r}$, $\mathbf{r}'$, $n(\mathbf{r}')$, S.
Salida: P^s contiene la presión dispersada en cada punto de observación para todas las configuraciones de las fuentes de ruido.

1: {Transferencia inicial de datos de la CPU a la GPU}
2: cudaMemcpy($P, \mathbf{r}', n(\mathbf{r}'), S, \mathbf{r}, \dots$)
3: {Operaciones para calcular P^s por bloques}
4: **para** $j = 0$ **hasta** $M-1$ **paso** bs **hacer**
5:   **para** $i = 0$ **hasta** $N-1$ **paso** bs **hacer**
6:     {Generación de los bloques K' y P'}
7:     GeneraBloques($P, \mathbf{r}', n(\mathbf{r}'), S, \mathbf{r}, K', P', \dots$)
8:     {Multiplicación matricial usando cuBLAS}
9:     cublasCgemm($\dots, K', bs, P', bs, \dots, P^{s\prime}, bs$)
10:     {Almacenamiento de resultados parciales}
11:     AlmacenaResultados($P^s, P^{s\prime}, \dots$)
12:   **fin para**
13: **fin para**
14: {Transferencia de resultados de la GPU a la CPU}
15: cudaMemcpy($P^s, \dots$)

Por su parte, la implementación de la técnica *BigBlocks* para GPU usando CUDA presenta ciertas diferencias respecto de la implementación para CPU y Xeon Phi. Los algoritmos 4.7, 4.8 y 4.9 muestran las diferentes partes en las que se divide el problema cuando se usa CUDA. Si bien los pasos a realizar son básicamente los mismos que en la implementación basada en OpenMP, merece la pena describirlos con el fin de apreciar los cambios de diseño para adaptarse a las diferentes arquitecturas:

1. Inicialmente, se transfieren todos los datos necesarios desde la me-

moria del *host* (CPU) hacia la memoria del *dispositivo* (GPU). Este primer paso se realiza una única vez, al comienzo de la ejecución del algoritmo, y se corresponde con la línea 2 del algoritmo 4.7.

2. Posteriormente, puesto que N y M pueden ser mayores que bs, se repiten los siguientes pasos de forma iterativa hasta alcanzar la solución final:

   a) Se lanza un *kernel* CUDA en el que se calcula un bloque K' y se genera su bloque P' correspondiente. Al realizar las dos tareas en una única rutina se logra reducir la sobrecarga asociada al lanzamiento de un *kernel*. Este paso se refleja en la línea 7 del pseudocódigo 4.7 (lanzamiento del *kernel* desde la CPU) y se detalla en el algoritmo 4.8.

   b) Se realiza una llamada a la rutina de cuBLAS `cublasCgemm` para multiplicar los bloques K' y P' de forma eficiente, almacenando el resultado parcial en el bloque $P^{s\prime}$ que reside en la memoria de la GPU. Este paso se lleva a cabo en la línea 9 del algoritmo 4.7.

   c) Se lanza un nuevo *kernel* que acumula el resultado parcial obtenido, $P^{s\prime}$, en las posiciones adecuadas de la matriz P^s, todo ello en la memoria de la GPU. Este paso se muestra en la línea 11 del algoritmo 4.7 y se detalla en el algoritmo 4.9.

3. Finalmente, se recupera la matriz solución, P^s, transfiriéndola desde la memoria del *dispositivo* hasta la memoria del *host*. Este último paso sólo se realiza una vez, al final del algoritmo, y se muestra en la línea 15 del algoritmo 4.7.

Puesto que se usa una rutina de cuBLAS [141] y dicha librería usa almacenamiento por columnas, los bloques K', P' y $P^{s\prime}$ se han adecuado a este planteamiento y se almacenan por columnas.

4.2.5. Solución heterogénea para GPU y Xeon Phi

La última de las variantes diseñadas es una solución heterogénea, ideada para utilizar de forma conjunta dos tipos de aceleradores diferentes: una GPU y un Xeon Phi.

Algoritmo 4.8 Solución *BigBlocks* para GPU usando CUDA. Generación de los bloques (código ejecutado en GPU).

1: **para** $ii = IdBloque$ **hasta** $bs - 1$ **paso** $dimGrid$ **hacer**
2: **para** $jj = IdHilo$ **hasta** $bs - 1$ **paso** $dimBloque$ **hacer**
3: $K'_{ii,jj} = \int_{S_{i+ii}} \frac{\partial G(\mathbf{r}_{j+jj},\mathbf{r}')}{\partial n(\mathbf{r}')} \mathrm{d}S(\mathbf{r}')$ {cálculo de un bloque K'}
4: **fin para**
5: **fin para**
6: **para** $ll = IdBloque$ **hasta** $N_c - 1$ **paso** $dimGrid$ **hacer**
7: **para** $ii = IdHilo$ **hasta** $bs - 1$ **paso** $dimBloque$ **hacer**
8: $P'_{ll,ii} = P_{i+ii,ll}$ {generación de un bloque P'}
9: **fin para**
10: **fin para**

Algoritmo 4.9 Solución *BigBlocks* para GPU usando CUDA. Almacenamiento de resultados parciales (código ejecutado en GPU).

1: **para** $jj = IdBloque$ **hasta** $bs - 1$ **paso** $dimGrid$ **hacer**
2: **para** $ll = IdHilo$ **hasta** $N_c - 1$ **paso** $dimBloque$ **hacer**
3: $P^s_{j+jj,ll} = P^s_{j+jj,ll} + P^{s\prime}_{ll,jj}$ {acumulación de resultados parciales}
4: **fin para**
5: **fin para**

Esta solución se basa en la paralelización mediante división del dominio, de forma que el cálculo de P^s se reparte entre los diferentes aceleradores utilizados. Como se muestra en la figura 4.5, el problema original se divide por filas (de P^s), asignando las primeras al Xeon Phi y las últimas a la GPU. El reparto por filas permite que no haya ningún tipo de replicación en los cálculos de los elementos de K, logrando así una solución eficiente y escalable. Por su parte, P se debe copiar de forma completa en la memoria de cada uno de los aceleradores, pero esto no supone ningún problema ya que P tiene una dimensión $N \times N_c$ (con $N \gg N_c$ y $M \gg N_c$).

En el algoritmo 4.10 se muestra la forma en la que se implementa la solución heterogénea para GPU y Xeon Phi. Se recurre al uso de OpenMP para generar dos hilos (línea 3), de los cuales el primero de ellos se ocupa del Xeon Phi, mientras que el segundo maneja la GPU. Dichos hilos se encargan del intercambio de datos con el acelerador (transferencias en sentido ascendente y descendente) y de lanzar las rutinas o *kernels* necesarios

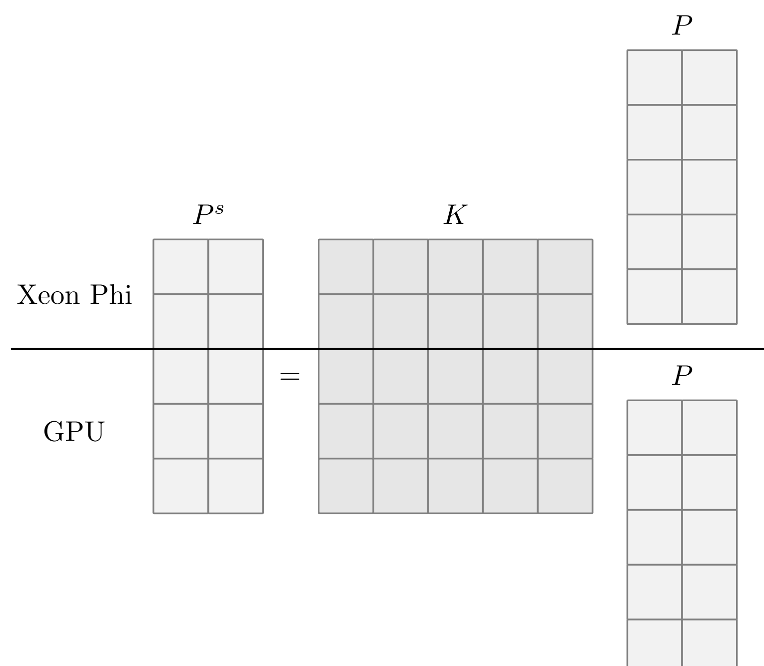


Figura 4.5: Obtención de la matriz P^S a partir de K y P con una solución heterogénea. El coprocesador Xeon Phi calcula las M' primeras filas de P^s, mientras que la GPU calcula las $M - M'$ últimas filas de P^s. Nótese que ambos aceleradores necesitan una copia completa de P.

(líneas 6 y 9 del algoritmo 4.10). La técnica usada por cada acelerador para resolver su parte del problema es aquella que logra un mayor rendimiento (menor tiempo de ejecución) para el valor de N_c que se esté utilizando. Para el caso de $N_c = 1$, la técnica bMVP es siempre la más adecuada por estar ante un único producto matriz-vector. En los casos en los que $N_c > 1$, como se mostrará en el capítulo dedicado a los resultados computacionales (capítulo 6), la técnica *BigBlocks* logra el mejor rendimiento en la mayoría de las ocasiones.

La línea 2 del algoritmo 4.10 es el resultado del reparto de la carga de trabajo (filas de P^s) en función del rendimiento de cada acelerador. La carga de trabajo, en tanto por uno, asociada a cada uno de los aceleradores (W_{phi} y W_{gpu}) puede calcularse mediante la expresión (4.9), que simplifi-

Algoritmo 4.10 Solución heterogénea para GPU y Xeon Phi.

1: {Carga de trabajo proporcional al rendimiento de cada acelerador}
2: $M' = M \cdot W_{\text{phi}}$
3: #pragma omp parallel num_threads(2)
4: **si** omp_get_thread_num() = 0 **entonces**
5: {El primer hilo se encarga del Xeon Phi}
6: PresDispersada_phi(0, $M' - 1$, ...)
7: **si no**
8: {El segundo hilo controla la GPU}
9: PresDispersada_gpu(M', $M - 1$, ...)
10: **fin si**

cada para este caso particular puede expresarse de la siguiente manera:

$$W_{\text{phi}} = \frac{T_{\text{gpu}}}{T_{\text{phi}} + T_{\text{gpu}}}, \tag{4.10}$$

$$W_{\text{gpu}} = 1 - W_{\text{phi}}, \tag{4.11}$$

para lo que se utiliza como referencia de tiempos un problema de tamaño moderado. De esta forma, con el objetivo de conseguir el mayor rendimiento para las posibles combinaciones de diferentes GPU y Xeon Phi, cuanto menor es el tiempo asociado a un acelerador (mayor rendimiento) más carga de trabajo se le asigna. Cabe mencionar que, una vez que se modela el rendimiento de los aceleradores, no es necesario recalcular la carga de trabajo que se les asigna aunque se resuelvan problemas de diferente tamaño, siempre que los aceleradores empleados sean idénticos y que el número de configuraciones para las fuentes de ruido, N_c, no varíe en gran medida.

Por último, merece la pena mencionar otra posibilidad barajada a la hora de implementar una solución heterogénea: un reparto de la carga de trabajo utilizando una *descomposición funcional* [79]. Puesto que las dos tareas más costosas desde el punto de vista computacional son el cálculo de los elementos de K y el propio producto matricial $K \cdot P$ (para los casos con $N_c = 1$, producto matriz-vector $K \cdot p$), podría plantearse una descomposición funcional en la que un dispositivo fuese calculando bloques de la matriz K mientras que otro dispositivo realiza el producto matricial de los bloques previamente calculados, todo ello de forma simultánea.

Sin embargo, el planteamiento descrito en el párrafo anterior se ha descartado debido al peso que pasan a tener las comunicaciones (transferencias entre aceleradores o entre la CPU y los aceleradores). Con este planteamiento, la matriz K se transfiere completamente (bloque a bloque) entre dos dispositivos, el que calcula sus bloques y el que lleva a cabo el producto entre bloques. Dicha transferencia tiene un coste $\mathcal{O}(N \cdot M)$, mientras que el coste de las transferencias de la alternativa implementada es $\mathcal{O}(\text{máx}\{N \cdot N_c, M \cdot N_c\})$. Por tanto, en la práctica, puesto que $N \gg N_c$ y $M \gg N_c$, el hecho de transferir la matriz K, de dimensión $N \times M$, requiere un tiempo muy superior al de transferir P, que es $N \times N_c$, y P^s, que es $M \times N_c$.

4.3. Resultados

En este apartado, se muestran varios resultados experimentales obtenidos con la herramienta presentada en este capítulo. Por su parte, los datos de tiempo de ejecución y aceleración asociados a estos problemas se muestran en el punto 6.4 del capítulo que recoge los resultados computacionales.

4.3.1. Presión acústica

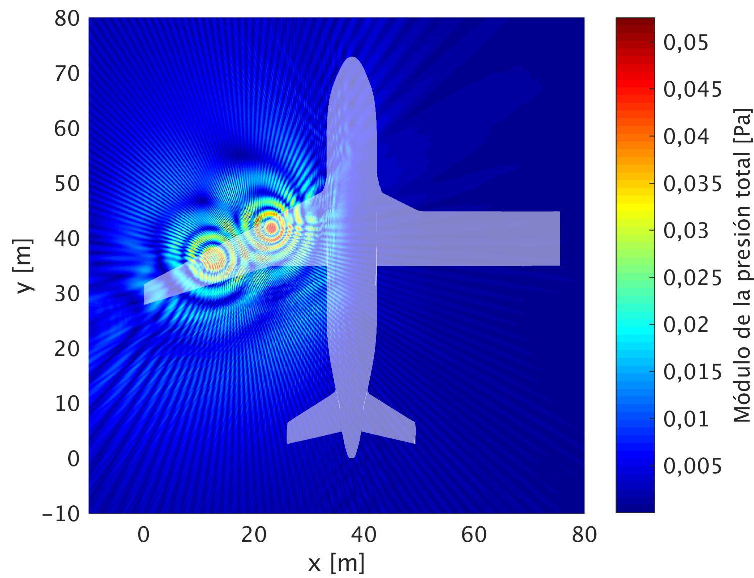
Los resultados de presión acústica mostrados a continuación se corresponden con el análisis del ruido generado por una aeronave considerando dos configuraciones diferentes para las fuentes de ruido (motores de la aeronave). La primera de ellas es una configuración típica con los motores bajo el ala, mientras que en la segunda los motores están situados sobre el ala (configuración de bajo ruido [137, 138]).

Previamente, para cada configuración de las fuentes de ruido, se obtiene la distribución de la presión sobre la superficie de la aeronave (Airbus serie A3xx) a una frecuencia de 1 kHz, usando la implementación del FMM mostrada en el capítulo 2. Para llevar a cabo dichos análisis, la aeronave se modela utilizando una malla con ∼6 elementos por longitud de onda lineal, lo que se traduce en un total de 1009392 facetas triangulares ($N =$

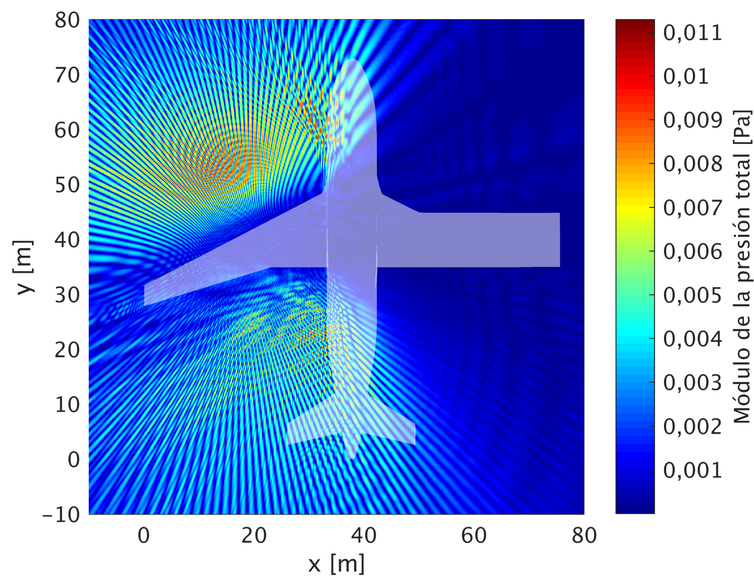
1009392). A continuación, partiendo de los resultados obtenidos con las simulaciones previamente mencionadas, se calcula la presión total para las dos configuraciones de las fuentes de ruido ($N_c = 2$) en un plano situado bajo la aeronave ($z = 0$) utilizando la herramienta presentada en este capítulo. La región del plano estudiada tiene unas dimensiones de 90,015 m $\times$ 90,015 m, con una resolución de 4 elementos por longitud de onda lineal, lo que da como resultado 1123600 puntos de observación ($M = 1123600$).

En la figura 4.6, se muestra el módulo de la presión total en un plano situado debajo de la aeronave. Para poder determinar de forma sencilla la ubicación de los puntos de observación utilizados en el análisis, en la figura también se muestra la aeronave (de forma translúcida) en su posición real. En la figura 4.6(b), puede apreciarse con claridad el efecto de apantallamiento producido por el ala izquierda del avión cuando se sitúan los motores sobre la misma.

Por su parte, en la figura 4.7 se representan los mismos resultados que en la figura anterior, pero se mantiene fija la escala de la barra de colores. De esta forma, se puede valorar con mayor facilidad la reducción en los niveles de presión acústica lograda con la configuración de bajo ruido (motores sobre el ala) respecto de la configuración típica (motores bajo el ala).



(a)



(b)

Figura 4.6: Módulo de la presión acústica total en un plano situado en $z = 0\,\mathrm{m}$ (debajo de la aeronave). Vista cenital. Frecuencia de las fuentes de ruido: 1 kHz. (a) Motores bajo el ala (configuración típica). (b) Motores sobre el ala (configuración de bajo ruido).

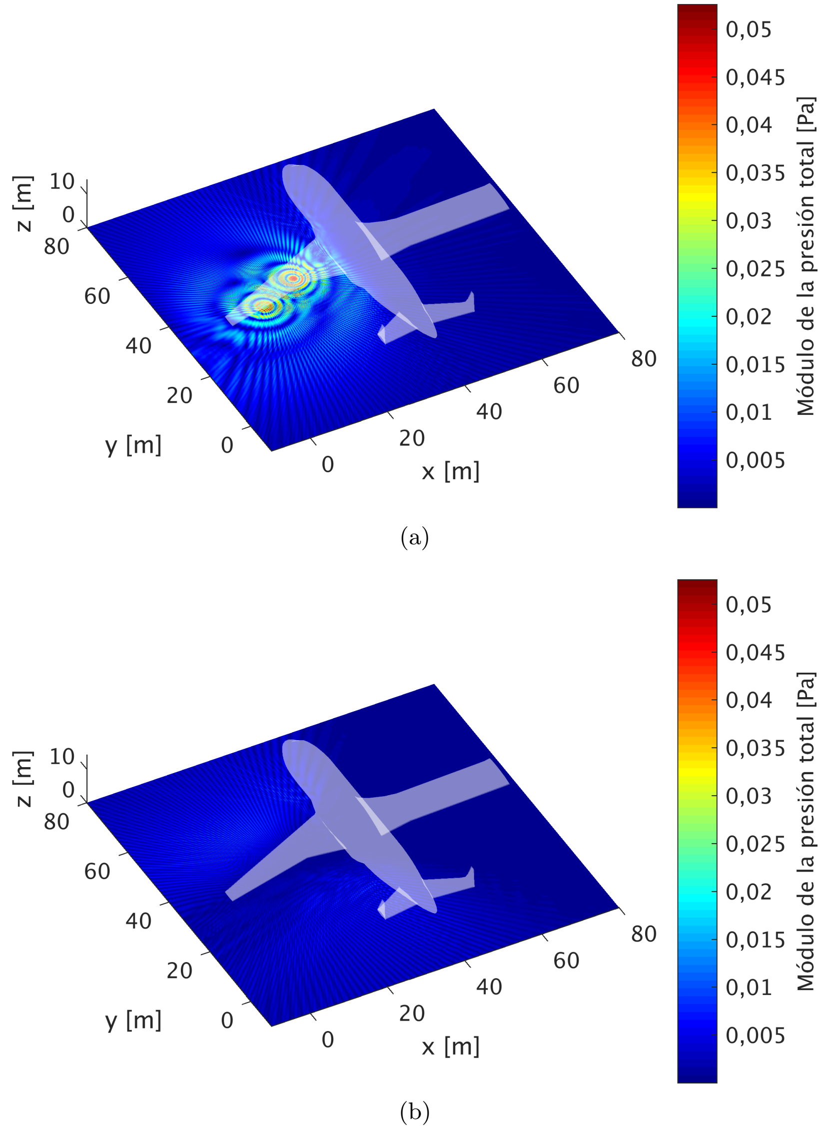


Figura 4.7: Vista comparativa del módulo de la presión acústica total en un plano situado en $z = 0\,\mathrm{m}$ (debajo de la aeronave). Frecuencia de las fuentes de ruido: $1\,\mathrm{kHz}$. (a) Motores bajo el ala. (b) Motores sobre el ala.

4.3.2. Validación

Con el objetivo de comprobar la corrección de las técnicas presentadas en este capítulo, se han comparado los resultados proporcionados por todas ellas para el problema analizado en el punto anterior.

Para medir las diferencias existentes entre los resultados obtenidos, se ha utilizado tanto el error relativo (ϵ_{rel}) como la raíz cuadrada del error cuadrático medio (ϵ_{RMS}), definidos previamente en las ecuaciones (2.17a) y (2.17b).

		bMVP	mMVP	*SmallBlocks*	*BigBlocks*
CPU	ϵ_{rel}	–	$1{,}7 \cdot 10^{-5}$	$6{,}7 \cdot 10^{-6}$	$2{,}7 \cdot 10^{-4}$
	ϵ_{RMS}	–	$1{,}0 \cdot 10^{-7}$	$4{,}0 \cdot 10^{-8}$	$1{,}6 \cdot 10^{-6}$
Xeon Phi	ϵ_{rel}	$7{,}7 \cdot 10^{-6}$	$7{,}7 \cdot 10^{-6}$	$1{,}5 \cdot 10^{-5}$	$2{,}7 \cdot 10^{-4}$
	ϵ_{RMS}	$4{,}6 \cdot 10^{-8}$	$4{,}6 \cdot 10^{-8}$	$8{,}7 \cdot 10^{-8}$	$1{,}6 \cdot 10^{-6}$
GPU	ϵ_{rel}	$6{,}1 \cdot 10^{-6}$	$6{,}1 \cdot 10^{-6}$	–	$2{,}6 \cdot 10^{-5}$
	ϵ_{RMS}	$3{,}6 \cdot 10^{-8}$	$3{,}6 \cdot 10^{-8}$	–	$1{,}6 \cdot 10^{-7}$

Tabla 4.1: Medida de las diferencias entre las soluciones proporcionadas por todas las técnicas implementadas en comparación con la implementación bMVP para CPU. $N = 1009392$, $M = 1123600$, $N_c = 2$.

En la tabla 4.1, se muestran las diferencias entre los resultados obtenidos con las técnicas implementadas. Los resultados proporcionados por la implementación más simple en términos computacionales (bMVP para CPU) se han utilizado como solución de referencia. En todos los casos, las diferencias son menores o iguales que $2{,}7 \cdot 10^{-4}$.

Capítulo 5

Implementación paralela y heterogénea del método de reconstrucción de fuentes aplicado al problema electromagnético inverso

Índice

5.1. SRM aplicado a la caracterización de antenas . 153
5.1.1. Paralelización para aceleradores gráficos 156
5.2. SRM aplicado a problemas de *imaging* 159
5.2.1. Paralelización para aceleradores gráficos 161
5.3. Resultados 163
5.3.1. Caracterización de antenas 164
5.3.2. Imagen electromagnética 167

En este capítulo, se describen las herramientas paralelas y heterogéneas, basadas en el método de reconstrucción de fuentes, que se han desarrollado

para abordar dos aplicaciones diferentes del problema inverso en electromagnetismo: caracterización de antenas e imagen electromagnética.

El objetivo principal de las herramientas desarrolladas consiste en reducir los tiempos de ejecución que se obtienen con las herramientas de partida que se toman como referencia (implementaciones paralelas para procesadores convencionales). De tal forma que, mediante una notable aceleración computacional sobre las implementaciones de partida, puedan obtenerse resultados en tiempo real o cuasi-real para los problemas de tamaño habitual.

En el apartado 5.1, se describe de forma somera el método de reconstrucción de fuentes (SRM) y se detalla la paralelización para procesadores gráficos de la herramienta para caracterización de antenas. Además, se muestra la relación entre la técnica utilizada para implementar el SRM en GPU y una de las técnicas utilizadas en el capítulo 4 para el problema de dispersión acústica.

Posteriormente, en el apartado 5.2, se explica el método de reconstrucción de perfiles basado en el SRM y se detalla la estrategia diseñada para su paralelización en múltiples GPU, basada en un particionado de la carga de trabajo a doble nivel (grano grueso-grano fino).

El lenguaje de programación utilizado para desarrollar las herramientas presentadas en este capítulo es CUDA C, ya que permite aprovechar todas las características de los procesadores gráficos utilizados (NVIDIA series Fermi y Kepler).

Para finalizar el capítulo, en el apartado 5.3, se muestran varios ejemplos de aplicación para las herramientas desarrolladas y se validan los resultados comparándolos con los obtenidos con las implementaciones de referencia. Por su parte, los datos relativos a tiempo de ejecución, aceleración y eficiencia se muestran en el capítulo 6.

5.1. SRM aplicado a la caracterización de antenas

Para llevar a cabo la caracterización y, en su caso, el diagnóstico de una antena, resulta indispensable conocer su diagrama de radiación. Puesto que no siempre es posible medir el diagrama de radiación de una antena en una cámara anecoica, debido a la limitación del espacio que separa la antena bajo medida (ABM) y la sonda, los métodos para calcular el diagrama de radiación de la ABM por medio de transformaciones de campo cercano (típicamente medido en cámara anecoica) a campo lejano [6, 51, 52, 53, 54, 57] resultan de especial utilidad en dichos casos.

El SRM permite obtener a partir del campo medido (habitualmente en la región de campo cercano) una distribución de la densidad de corriente equivalente, tanto magnética como eléctrica, que radia el mismo campo que la ABM en el exterior del dominio fuente [6, 14, 53, 54, 57]. De esta forma, las densidades de corriente equivalentes se reconstruyen a partir del campo medido en el dominio de adquisición y, posteriormente, se puede determinar el campo en cualquier punto a partir de las densidades de corriente referidas. Sin embargo, el mayor inconveniente del SRM reside en su alto coste computacional, especialmente en aquellos casos en los que se desea caracterizar antenas de gran tamaño eléctrico [7].

La distribución de corrientes equivalentes se determina a partir del campo medido mediante la resolución de una ecuación integral, la cual relaciona el campo electromagnético y las fuentes mediante el potencial vectorial [143]. Dicha ecuación puede utilizarse cuando el problema original se plantea según los términos del principio de equivalencia electromagnética [6, 14, 143, 121].

En el problema equivalente considerado, el medio es homogéneo, por lo que es posible expresar el campo eléctrico total en cualquier punto de observación $\mathbf{r}$ situado en el exterior del dominio fuente como la suma de las contribuciones debidas a las distribuciones de densidad de corriente

eléctrica y magnética:

$$\vec{E}(\mathbf{r}) = \vec{E}_{J_{eq}}(\mathbf{r}) + \vec{E}_{M_{eq}}(\mathbf{r}), \tag{5.1a}$$

$$\vec{E}_{J_{eq}}(\mathbf{r}) = -\frac{j\eta}{4\pi\kappa}\int_{S'}\left\{\left(\kappa^2+\nabla^2\right)\left[\vec{J}_{eq}(\mathbf{r}')\,G(\mathbf{r},\mathbf{r}')\right]\right\}\mathrm{d}S', \tag{5.1b}$$

$$\vec{E}_{M_{eq}}(\mathbf{r}) = -\frac{1}{4\pi}\nabla\times\int_{S'}\left[\vec{M}_{eq}(\mathbf{r}')\,G(\mathbf{r},\mathbf{r}')\right]\mathrm{d}S', \tag{5.1c}$$

donde η es la impedancia característica del medio, κ es el número de onda, y $\mathbf{r}'$ son los puntos pertenecientes a la superficie S', en la que se ubican las fuentes equivalentes. $\vec{J}_{eq}(\mathbf{r}')$ y $\vec{M}_{eq}(\mathbf{r}')$ representan las densidades de corriente eléctrica y magnética equivalentes, respectivamente, definidas sobre la superficie de reconstrucción S'. Por su parte, $G(\mathbf{r},\mathbf{r}')$, que es la función de Green en espacio libre en el punto $\mathbf{r}$ (perteneciente al dominio de observación) para la fuente equivalente situada en el punto $\mathbf{r}'$, se define de la siguiente manera:

$$G(\mathbf{r},\mathbf{r}') = \frac{e^{-j\kappa|\mathbf{r}-\mathbf{r}'|}}{4\pi|\mathbf{r}-\mathbf{r}'|}\,. \tag{5.2}$$

La ecuación integral (5.1a) se discretiza, en este caso usando el método de los momentos con funciones base de tipo pulso (ver [6, 14]), dando lugar a un sistema de M ecuaciones (proporcionales al número de puntos de adquisición) y N incógnitas (proporcionales al número de funciones base que expanden las corrientes equivalentes), como el mostrado a continuación:

$$\begin{bmatrix} E_x \\ E_y \\ E_z \end{bmatrix} = \begin{bmatrix} Z_{E_x,J_x} & Z_{E_x,J_y} & Z_{E_x,J_z} & Z_{E_x,M_x} & Z_{E_x,M_y} & Z_{E_x,M_z} \\ Z_{E_y,J_x} & Z_{E_y,J_y} & Z_{E_y,J_z} & Z_{E_y,M_x} & Z_{E_y,M_y} & Z_{E_y,M_z} \\ Z_{E_z,J_x} & Z_{E_z,J_y} & Z_{E_z,J_z} & Z_{E_z,M_x} & Z_{E_z,M_y} & Z_{E_z,M_z} \end{bmatrix} \begin{bmatrix} J_x \\ J_y \\ J_z \\ M_x \\ M_y \\ M_z \end{bmatrix}, \tag{5.3}$$

donde E_x, E_y, E_z son las componentes cartesianas del campo eléctrico en el dominio de adquisición. Asimismo, J_x, J_y, J_z y M_x, M_y, M_z son las componentes cartesianas de la densidad de corriente eléctrica y magnética, respectivamente, en el dominio de reconstrucción. Los elementos de la

matriz de impedancias (Z) relacionan cada una de las componentes del campo eléctrico con cada una de las componentes de la densidad de corriente eléctrica y magnética. A modo de ejemplo, el término de la matriz Z que relaciona la componente x del campo eléctrico con la componente y de la densidad de corriente magnética equivalente se calcula de la siguiente manera:

$$Z_{E_x,M_y}(m,n) = \frac{-1}{4\pi}(z_m - z'_n)\left[\frac{1 + j\kappa|\mathbf{r}_m - \mathbf{r}'_n|}{|\mathbf{r}_m - \mathbf{r}'_n|^3}\right]e^{-j\kappa|\mathbf{r}_m - \mathbf{r}'_n|}\Delta S'_n\,, \quad (5.4)$$

siendo m un índice relacionado con el dominio de adquisición del campo eléctrico y n un índice relacionado con el domino de reconstrucción de las densidades de corriente equivalentes. Por su parte, $\Delta S'_n$ representa una de las facetas del dominio de reconstrucción.

En la práctica, es habitual considerar un dominio de adquisición esférico (utilizado en cámaras anecoicas de rango esférico del tipo *roll* sobre acimut) y un sistema de coordenadas local para cada faceta del dominio de reconstrucción. En dicho caso, el sistema de ecuaciones (5.3) puede transformarse, mediante las proyecciones adecuadas (ver [6, 14]), en un sistema como el siguiente:

$$\begin{bmatrix} E_\theta \\ E_\varphi \end{bmatrix} = \begin{bmatrix} Z_{\theta,J_u} & Z_{\theta,J_v} \\ Z_{\varphi,J_u} & Z_{\varphi,J_v} \end{bmatrix} \begin{bmatrix} J_u \\ J_v \end{bmatrix} + \begin{bmatrix} Z_{\theta,M_u} & Z_{\theta,M_v} \\ Z_{\varphi,M_u} & Z_{\varphi,M_v} \end{bmatrix} \begin{bmatrix} M_u \\ M_v \end{bmatrix}\,, \quad (5.5)$$

en el cual el campo eléctrico se expresa según sus componentes tangenciales θ y φ. Además, se tiene un sistema de coordenadas local para cada faceta del dominio de reconstrucción del tipo (u, v, n), siendo $\hat{u}$ y $\hat{v}$ dos vectores unitarios ortogonales y tangenciales a la superficie de la faceta, y $\hat{n}$ el vector normal unitario.

A su vez, la ecuación (5.5) también se puede expresar de forma compacta como se muestra a continuación:

$$[E] \;=\; [Z_J][J] + [Z_M][M] \;=\; [Z][I]\,. \quad (5.6)$$

Para resolver el sistema de ecuaciones planteado, se suele recurrir al uso de métodos iterativos. En este caso, se ha utilizado el método del gradiente

conjugado para la minimización de la norma del residuo (CGNR) [73], el cual realiza varios productos matriz-vector por iteración. La función de coste empleada por el CGNR busca minimizar la diferencia entre el campo medido y el campo radiado por las corrientes equivalentes obtenidas en cada iteración. Si, además, se normaliza el sistema de ecuaciones como se muestra en [6], el condicionamiento de la matriz mejora, aumentando la velocidad de convergencia del algoritmo.

El uso de un esquema iterativo presenta una ventaja especialmente interesante: permite el cálculo de la matriz de impedancias sobre la marcha. Esta ventaja se aprovecha en la técnica de ahorro de memoria (MST) para el SRM, presentada en [6] en su versión secuencial para procesadores convencionales. La técnica MST elude el almacenamiento completo de la matriz de impedancias, calculando cada fila justo en el momento necesario para llevar a cabo el producto matriz-vector. Esta técnica permite reducir de forma drástica los requisitos de memoria, con un coste espacial que pasa de $\mathcal{O}(MN)$ a $\mathcal{O}(\text{máx}\{M, N\})$. Sin embargo, la necesidad de recalcular la matriz Z en cada iteración incrementa de forma notable los tiempos de ejecución.

5.1.1. Paralelización heterogénea para aceleradores gráficos

Para determinar si puede ser beneficioso (en términos computacionales) rediseñar la técnica MST [6] para su uso en procesadores gráficos, resulta conveniente analizar el coste espacial y temporal de dicha técnica.

La principal ventaja de la técnica MST reside en la reducción del coste espacial de $\mathcal{O}(MN)$ a $\mathcal{O}(\text{máx}\{M, N\})$, ya que la matriz de impedancias no se almacena de forma completa en ningún momento. Si se tiene en cuenta que la cantidad de memoria disponible en una tarjeta gráfica suele ser reducida en comparación con la memoria de todo el sistema, el MST parece adecuado para su ejecución en GPU en lo relativo a los requisitos de memoria.

Por su parte, los productos matriz-vector que se llevan a cabo en las iteraciones del algoritmo CGNR tienen una complejidad temporal $\mathcal{O}(MN)$. Además, puesto que se evita el precálculo y almacenamiento de la matriz de

impedancias, sus elementos deben calcularse de nuevo en todas las iteraciones. Si bien esto último no modifica la complejidad temporal del algoritmo, afecta de forma negativa a los tiempos de ejecución [6].

De este modo, se tiene un algoritmo con bajos requisitos de memoria y con una intensidad computacional alta, lo que representa una situación idónea para el uso de procesadores gráficos, siempre que se cumpla otro requisito adicional: el algoritmo utilizado debe ser altamente paralelizable.

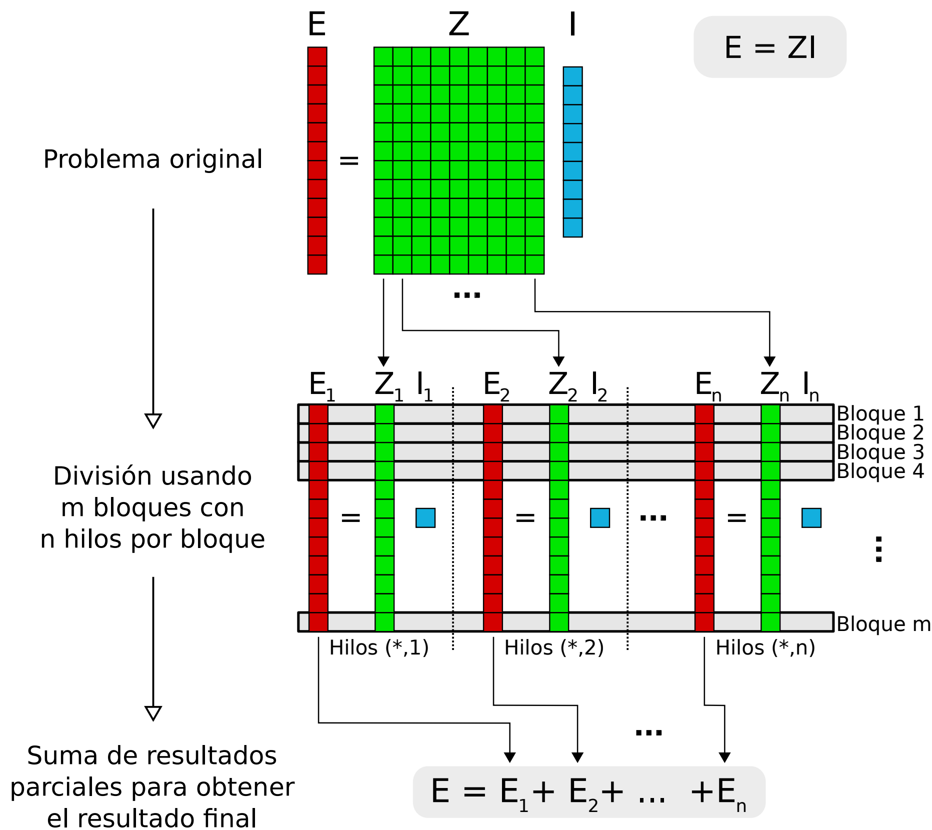


Figura 5.1: Estrategia utilizada para paralelizar el producto matriz-vector de la técnica MST. Por simplicidad, se asume que N es múltiplo del número de hilos por bloque (n) y que M es múltiplo del número de bloques (m).

La estrategia diseñada para paralelizar los productos matriz-vector consiste en dividir el problema original en múltiples subproblemas que pueden

resolverse de forma simultánea, como se muestra en la figura 5.1.

Las filas de la matriz se asignan de forma cíclica a los m bloques de hilos CUDA [112]. Por su parte, los elementos que componen una fila se asignan a los n hilos pertenecientes a un mismo bloque. Los elementos de la matriz Z nunca se almacenan de forma permanente, sino que se calculan bajo demanda, justo en el momento necesario para llevar a cabo el producto matriz-vector, conservando el bajo consumo de memoria de la implementación original (para CPU) de la técnica MST.

La estrategia utilizada en este caso, que sigue la misma filosofía de la técnica bMVP aplicada al cálculo de la presión total en el problema de dispersión acústica (ver punto 4.2.2), consta de tres pasos que se repiten hasta obtener el resultado del producto matriz-vector:

1. Cada hilo calcula el elemento de la matriz de impedancias que necesita en ese instante sobre la marcha. Un hilo determinado sólo calcula los elementos de Z que le corresponden, dependiendo de su número de bloque (fila) y de su posición dentro de dicho bloque (columna).

2. Cada hilo multiplica el elemento de Z que acaba de calcular por el elemento correspondiente del vector de densidades de corriente y acumula dicho producto en un bloque de memoria compartida de la GPU.

3. Todos los hilos pertenecientes a un mismo bloque realizan una suma por reducción en paralelo, obteniendo así un valor del campo eléctrico.

En la figura 5.1, se representa cómo los hilos de un mismo bloque deben cooperar para obtener un valor del vector E.

Para finalizar, merece la pena indicar que la implementación para GPU de la técnica MST aplicada al método de reconstrucción de fuentes se divide en un total de diez rutinas o *kernels* diferentes: cinco para operar con la densidad de corriente eléctrica y otros cinco que se encargan de la densidad de corriente magnética.

5.2. SRM aplicado a problemas de imagen electromagnética

El problema de la imagen electromagnética suscita gran interés debido a su amplio abanico de aplicaciones. Sin embargo, su utilidad puede verse mermada en problemas de gran tamaño eléctrico o cuando los tiempos de cálculo asociados a la resolución del problema puedan incumplir los requisitos de tiempo real o cuasi-real de algunas aplicaciones como, por ejemplo, las relacionadas con la seguridad [10, 11, 12]. En este último caso, no resultaría viable soportar amplios y molestos tiempos de espera para escanear a personas (o paquetes) en busca de objetos peligrosos ocultos.

La herramienta para la reconstrucción de perfiles que se presenta en este capítulo está basada en el SRM y su funcionamiento puede dividirse en tres pasos:

1. Problema directo. Se mide el campo dispersado por el objeto bajo análisis en el dominio de adquisición.

2. Problema inverso. Se define una superficie de reconstrucción, de dimensiones adecuadas, que debe encerrar al objeto bajo medida. A partir de las medidas del campo dispersado, se obtiene la densidad de corriente equivalente, tanto magnética como eléctrica, reconstruida en dicha superficie utilizando el SRM.

3. Cálculo del campo interno. A partir de las densidades de corriente reconstruidas en el paso anterior, se calcula el campo en el interior del dominio de reconstrucción.

Puesto que se desea conocer el valor del campo en el interior del dominio de reconstrucción, las densidades de corriente equivalentes reconstruidas en el segundo paso se obtienen sin forzar la condición de campo interno nulo (no se utiliza el principio de equivalencia de Love).

En el primer paso, se mide el campo dispersado por el objeto, para lo que se pueden utilizar múltiples ondas planas incidentes, con el objetivo de aumentar la información disponible acerca del objeto dispersor.

En el segundo paso, para cada una de las ondas planas incidentes utilizadas, se reconstruyen las densidades de corriente equivalentes en la superficie del dominio de reconstrucción usando el SRM (ver punto 5.1). El problema inverso que se plantea en este paso supone resolver un sistema de M ecuaciones (con $M/2$ puntos de adquisición de las componentes E_θ y E_φ) y N incógnitas (con $N/4$ puntos en los que se reconstruyen las componentes u y v de la densidad de corriente eléctrica y magnética). Dicho sistema de ecuaciones puede resolverse de forma iterativa con el CGNR, utilizando una función de coste que permita minimizar la diferencia entre el campo dispersado que se ha medido y el campo radiado por las corrientes equivalentes reconstruidas, como se muestra en la expresión (5.7):

$$f(I) = \frac{1}{2}\left\| \begin{bmatrix} E_\theta \\ E_\varphi \end{bmatrix} - \begin{bmatrix} Z_{\theta,J_u} & Z_{\theta,J_v} & Z_{\theta,M_u} & Z_{\theta,M_v} \\ Z_{\varphi,J_u} & Z_{\varphi,J_v} & Z_{\varphi,M_u} & Z_{\varphi,M_v} \end{bmatrix} \begin{bmatrix} J_u \\ J_v \\ M_u \\ M_v \end{bmatrix} \right\|^2 , \quad (5.7)$$

donde se asume un dominio de adquisición esférico (con el campo expresado según sus componentes tangenciales θ y φ) y un sistema de coordenadas local para cada faceta del dominio de reconstrucción (con la densidad de corriente expresada en sus componentes u y v). Dicha función de coste se puede expresar de forma más compacta como sigue:

$$f(I) = \frac{1}{2} \left\| [E] - [Z][I] \right\|^2 . \quad (5.8)$$

En el último paso, una vez que se han reconstruido las corrientes equivalentes para una incidencia determinada, puede calcularse el campo producido por dicha onda incidente en el interior del dominio de reconstrucción de la siguiente manera:

$$\begin{bmatrix} E_{i,x} \\ E_{i,y} \\ E_{i,z} \end{bmatrix} = \begin{bmatrix} Z_{E_{i,x};J_u} & Z_{E_{i,x};J_v} & Z_{E_{i,x};M_u} & Z_{E_{i,x};M_v} \\ Z_{E_{i,y};J_u} & Z_{E_{i,y};J_v} & Z_{E_{i,y};M_u} & Z_{E_{i,y};M_v} \\ Z_{E_{i,z};J_u} & Z_{E_{i,z};J_v} & Z_{E_{i,z};M_u} & Z_{E_{i,z};M_v} \end{bmatrix} \begin{bmatrix} J_u \\ J_v \\ M_u \\ M_v \end{bmatrix} , \quad (5.9)$$

donde el campo interno se expresa en base a un sistema de coordenadas

cartesiano.

Este paso permite obtener, mediante el producto matriz-vector mostrado en la expresión (5.9), M_i valores para el campo interno (con $M_i/3$ puntos en los que se calculan las componentes x, y, z del campo interno).

La información relativa a la geometría del objeto bajo medida se deduce a partir del campo interno calculado en el último paso, de tal forma que los valores del campo con mayor amplitud se corresponden con la ubicación de las partes metálicas del objeto analizado. La amplitud normalizada del campo interno se obtiene calculando el valor cuadrático medio para sus componentes x, y, z, como se muestra a continuación:

$$|E_i| = \sqrt{\frac{|E_{i,x}|^2 + |E_{i,y}|^2 + |E_{i,z}|^2}{3}} \tag{5.10}$$

Finalmente, puesto que se contempla la posibilidad de utilizar múltiples ondas planas incidentes, las amplitudes obtenidas para cada una de las L incidencias utilizadas se combinan teniendo en cuenta su valor cuadrático medio de la siguiente manera:

$$|E_i| = \sqrt{\frac{1}{L}\sum_{l=1}^{L} |E_i|_l^2} \tag{5.11}$$

5.2.1. Paralelización heterogénea para aceleradores gráficos

El cálculo del campo interno para cada uno de los ángulos de incidencia utilizados puede dividirse algorítmicamente en dos pasos:

1. Reconstrucción de las densidades de corriente equivalentes utilizando el SRM.
2. Cálculo del campo interno a partir de las corrientes equivalentes reconstruidas.

El primero de los pasos coincide fundamentalmente con el problema analizado en la sección dedicada a la caracterización de antenas usando

el SRM y se resuelve utilizando la misma algoritmia (ver punto 5.1). Por su parte, el segundo paso únicamente requiere un producto matriz-vector adicional.

Por tanto, el coste del problema viene determinado por dos tareas: los productos matriz vector asociados a las iteraciones del algoritmo CGNR del primer paso, con una complejidad temporal $\mathcal{O}(MN)$, y el producto matriz-vector que permite calcular el campo interno, con una complejidad temporal $\mathcal{O}(M_iN)$.

En esta ocasión, se ha diseñado un particionado de dos niveles que permite aumentar el grado de paralelismo de la solución propuesta, con el objetivo de poder utilizar múltiples GPU simultáneamente para la resolución de un mismo problema.

Algoritmo 5.1 Solución heterogénea para múltiples GPU.

Entrada: g, L, $[E]$ para cada ángulo de incidencia.
Salida: $|E_i|$ contiene la combinación de los valores del campo interno calculados para todos los ángulos de incidencia.

1: #pragma omp parallel for num_threads(g) schedule(dynamic,1)
2: {Las GPU trabajan con g ángulos de incidencia de forma simultánea}
3: **para** $l = 1$ **hasta** L **hacer**
4: {Obtener las densidades de corriente equivalentes con el SRM}
5: Minimizar $f(I) = \frac{1}{2} \left\| [E] - [Z][I] \right\|^2$
6: {Calcular el campo interno a partir de las corrientes equivalentes reconstruidas}
7: $[E_i] = [Z_i][I]$
8: **fin para**
9: {Combinar los resultados de los L ángulos de incidencia}
10: $|E_i| = \sqrt{\frac{1}{L} \sum_{l=1}^{L} |E_i|_l^2}$

El algoritmo 5.1 muestra de forma simplificada la estrategia diseñada para lograr una paralelización para múltiples GPU (paralelización de grano grueso). Como se puede observar en la línea 1, se recurre al uso de una directiva OpenMP [96] del tipo `parallel for` en la que se generan g hilos con el objetivo de repartir los L ángulos de incidencia entre los g procesadores gráficos disponibles. De este modo, se logra simultanear el análisis de g ángulos de incidencia. Además, se tiene en cuenta la posibilidad de

que las GPU utilizadas sean diferentes entre sí, por lo que se utiliza una planificación dinámica para equilibrar automáticamente la carga de trabajo de los diferentes procesadores gráficos (ver línea 1 del algoritmo 5.1). Como resultado, se logra que una GPU más rápida analice más ángulos de incidencia que una más lenta, para compensar la diferencia de rendimiento.

A su vez, los diferentes cálculos para cada ángulo de incidencia se llevan a cabo usando CUDA [111] (paralelización de grano fino). Para ello, se paralelizan los distintos productos matriz-vector de la forma explicada en el punto 5.1.1 (ver figura 5.1).

Esta herramienta multi-GPU, diseñada para su aplicación en problemas de imagen electromagnética, consta de doce *kernels* diferentes, divididos en seis para los cálculos en los que se utiliza la densidad de corriente eléctrica y otros seis para la densidad de corriente magnética. Diez de ellos se utilizan en el paso en el que se reconstruyen las corrientes equivalentes utilizando el SRM (con la técnica MST). Los dos restantes se utilizan en el paso en el que se calcula el campo interno a partir de las corrientes equivalentes reconstruidas.

Por último, merece la pena señalar que la estrategia diseñada también puede aplicarse en aquellos casos en los que se lleva a cabo un análisis utilizando múltiples frecuencias con el objetivo de mejorar la exactitud de la reconstrucción de los objetos bajo medida. En dichos casos, las diferentes frecuencias y los distintos ángulos de incidencia se reparten entre todas las GPU utilizadas, de forma similar a la mostrada en las líneas 1–3 del algoritmo 5.1.

5.3. Resultados

En este apartado, se muestran los resultados experimentales obtenidos con las herramientas presentadas en este capítulo para varios ejemplos de aplicación. Por su parte, los resultados de tiempo de ejecución, aceleración y eficiencia asociados a dichos problemas se muestran en el punto 6.5 del capítulo dedicado a los resultados computacionales.

5.3.1. Caracterización de antenas

En el caso de la herramienta para caracterización de antenas basada en el SRM, se ha recurrido a dos ejemplos diferentes: una antena de una estación base y una antena de tipo hélice.

Tamaño de la ABM (D_0)	1,6 m (9,6 λ)
Frecuencia	1,8 GHz
Distancia ABM–sonda	5 m (30 λ) Campo cercano
Dominio de adquisición	21901 muestras $\Delta\theta = 1^\circ,\ \Delta\varphi = 3^\circ$
Número de ecuaciones (M)	43802
Dominio de reconstrucción	1910 facetas $\sim$6,7 facetas/λ
Número de incógnitas (N)	7640
Convergencia CGNR (RMSE)	3,4 %

Tabla 5.1: Parámetros para la obtención de las densidades de corriente equivalentes en una antena de una estación base usando la técnica MST para el método de reconstrucción de fuentes.

En el primero de ellos, se calculan las densidades de corriente equivalentes en el radomo que cubre a la antena bajo medida: una antena de tipo *array* de una estación base. Los parámetros relativos a este primer ejemplo se muestran en la tabla 5.1. Como se puede observar en dicha tabla, se tiene un sistema con 43802 ecuaciones, a partir de las 21901 medidas de las componentes E_θ y E_φ obtenidas en una cámara anecoica de rango esférico, y 7640 incógnitas (1910 para cada componente, u y v, de la densidad de corriente eléctrica y magnética).

La región de campo lejano (R_{FF}) para una antena puede definirse, de

forma general, como sigue [52, 144]:

$$R_{FF} \;=\; 2\,\frac{(D_0)^2}{\lambda}, \tag{5.12}$$

siendo D_0 la mayor de las dimensiones lineales (o el diámetro) de la ABM. Por tanto, en este caso, se tiene que las medidas se han realizado en campo cercano ($R_{FF} = 31\,\mathrm{m}$).

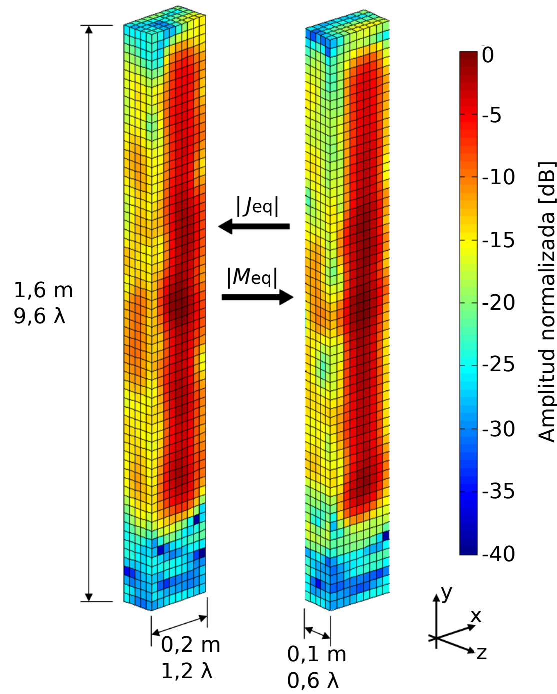


Figura 5.2: Densidades de corriente (eléctrica y magnética) equivalentes reconstruidas en el radomo que cubre a la antena de una estación base. Amplitud normalizada en dB.

En la figura 5.2, se muestra la amplitud normalizada de las densidades de corriente equivalentes reconstruidas en la superficie del radomo que cubre a la antena analizada.

El segundo ejemplo consiste en el cálculo de las corrientes equivalentes sobre una antena de tipo hélice. En la tabla 5.2, se muestran los paráme-

Tamaño de la ABM (D_0)	0,2 m (3 λ)
Frecuencia	4,5 GHz
Distancia ABM–sonda	4,85 m (72,75 λ) Campo lejano
Dominio de adquisición	21901 muestras $\Delta\theta = 1°,\ \Delta\varphi = 3°$
Número de ecuaciones (M)	43802
Dominio de reconstrucción	20640 facetas $\sim$20 facetas/λ
Número de incógnitas (N)	82560
Convergencia CGNR (RMSE)	3,9 %

Tabla 5.2: Parámetros para la obtención de las densidades de corriente equivalentes en una antena de tipo hélice usando la técnica MST para el método de reconstrucción de fuentes.

tros que definen este problema. En este caso, se tiene un sistema con 43802 ecuaciones (21901 medidas para E_θ y E_φ) y 82560 incógnitas (20640 para cada una de las componentes de la densidad de corriente eléctrica y magnética). Nuevamente, las medidas del campo eléctrico se han obtenido en una cámara anecoica de rango esférico pero, en esta ocasión, en la región de campo lejano ($R_{FF} = 1{,}2\,\mathrm{m}$).

La figura 5.3 muestra la amplitud normalizada de las densidades de corriente eléctrica equivalentes, reconstruidas en la superficie de la antena de tipo hélice que se analiza.

Finalmente, merece la pena señalar que el error RMS entre la solución de referencia (implementación para CPU) y la solución heterogénea para GPU es menor que $2{,}1 \cdot 10^{-4}$ en los casos analizados.

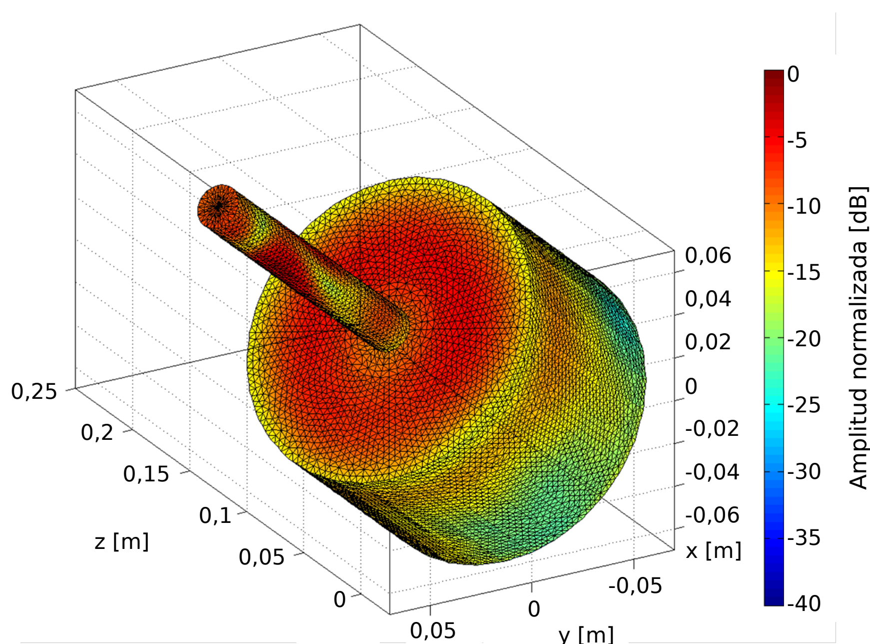


Figura 5.3: Densidad de corriente eléctrica equivalente reconstruida en una antena de tipo hélice. Amplitud normalizada en dB.

5.3.2. Imagen electromagnética

En el caso de la herramienta para imagen electromagnética, se ha utilizado, como ejemplo de aplicación, la resolución de un problema de reconstrucción de perfiles con dos objetos metálicos.

El primero de los objetos bajo medida es un cilindro con un diámetro de $0{,}5\,\lambda$ y una altura de $2\,\lambda$ (a lo largo del eje z), que se encuentra centrado en la posición $x = -0{,}75\,\lambda$, $y = 0{,}25\,\lambda$, $z = 0\,\lambda$. El segundo objeto bajo medida es un prisma rectangular de dimensiones $0{,}5\,\lambda \times 2\,\lambda \times 1\,\lambda$ (en x, y y z, respectivamente), que se encuentra centrado en la posición $x = 1\,\lambda$, $y = 0\,\lambda$, $z = -0{,}5\,\lambda$.

La resolución del problema se divide en tres pasos diferentes: el problema directo, el problema inverso y el cálculo del campo interno a partir de

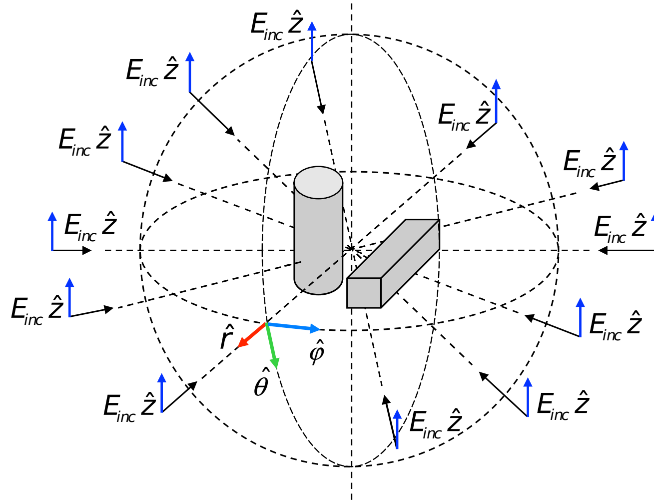


Figura 5.4: Distribución de las ondas planas incidentes.

las corrientes equivalentes reconstruidas.

En el primer paso, se resuelve el problema directo usando un software comercial que implementa el método de los momentos (Altair FEKO). Como se muestra en la figura 5.4, los objetos bajo medida se iluminan utilizando doce ondas planas incidentes con polarización según el eje z, situadas en el plano $z = 0$ y equiespaciadas desde $\varphi = 0°$ hasta $330°$ (en pasos de $30°$). Para cada onda incidente, se obtiene el campo dispersado por los objetos en una esfera de radio $10\,\lambda$ con una resolución tal que $\Delta\theta = 3°$ y $\Delta\varphi = 6°$, lo que resulta en un total de 3600 puntos para el dominio de adquisición.

En el segundo paso, para cada uno de los doce problemas directos planteados, se resuelve un problema inverso en el que se reconstruyen las densidades de corriente equivalentes en una caja de tamaño $Lx = Ly = Lz = 4\,\lambda$ que encierra el dominio bajo estudio. La superficie de la caja se discretiza utilizando facetas cuadradas de $\sim$0,235 λ de lado, lo que resulta en un total de 1734 facetas. Por tanto, cada problema inverso abordado con la herramienta presentada en este capítulo requiere resolver un sistema de ecuaciones lineales con $M = 7200$ ecuaciones (3600 puntos de adquisi-

ción para cada componente tangencial del campo eléctrico) y $N = 6936$ incógnitas (1734 para cada una de las componentes u y v de la densidad de corriente eléctrica y magnética). Para resolver cada sistema de ecuaciones, el CGNR itera hasta que el error RMS entre el campo dispersado y el campo producido por las corrientes equivalentes se sitúa por debajo del 2 % (en este ejemplo, entre 11 y 12 iteraciones).

En el último paso, a partir de las densidades de corriente equivalentes reconstruidas, se calcula el campo interno en un total de 19683 puntos del interior de la caja que encierra los objetos bajo medida (27 puntos por cada dimensión x, y, z con $\Delta x = \Delta y = \Delta z \approx 0{,}15\,\lambda$). Por tanto, se tiene que $M_i = 59049$, que se corresponden con los 19683 puntos en los que se calculan cada una de las tres componentes (x, y, z) del campo interno. Finalmente, una vez que se dispone de todos los resultados, se combina el campo interno calculado para las doce incidencias utilizadas como se muestra en el algoritmo 5.1.

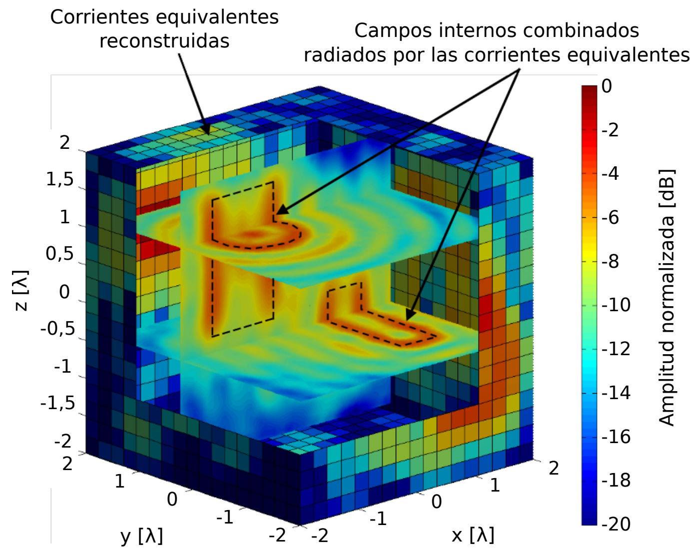


Figura 5.5: Densidad de corriente eléctrica equivalente reconstruida en la caja que encierra a los objetos bajo medida y campo eléctrico interno calculado en varios cortes. La línea discontinua de color negro marca el contorno de los objetos bajo medida.

En la figura 5.5, se muestra la amplitud del campo eléctrico interno para diferentes cortes teniendo en cuenta todas las incidencias utilizadas, cuyos resultados se combinan siguiendo las expresiones (5.10) y (5.11). Asimismo, se representa la densidad de corriente eléctrica equivalente reconstruida en la caja que encierra el dominio bajo estudio. Se aprecia que los valores de máxima amplitud para el campo eléctrico interno se ajustan al perfil de los objetos bajo medida.

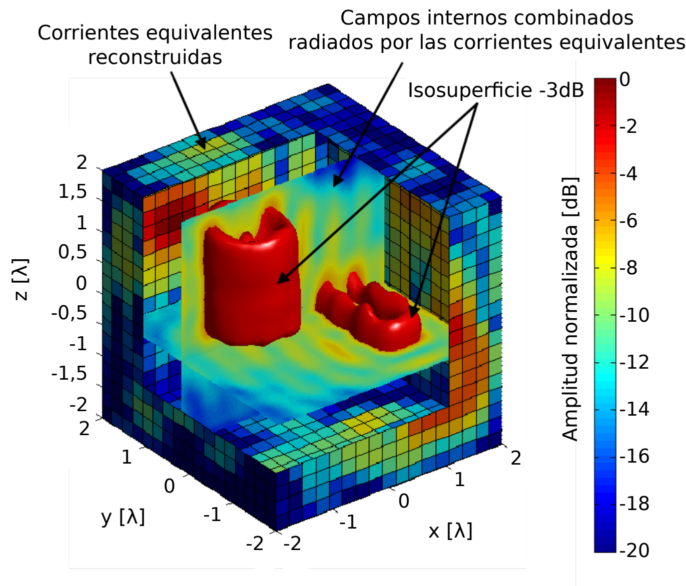


Figura 5.6: Densidad de corriente eléctrica equivalente reconstruida en la caja que encierra a los objetos bajo medida, campo eléctrico interno calculado en varios cortes y superficies de nivel $-3\,\mathrm{dB}$ para la amplitud del campo normalizada.

Por su parte, en la figura 5.6 se representan las superficies de nivel $-3\,\mathrm{dB}$ para la amplitud normalizada del campo eléctrico interno, la cual permite obtener información de la geometría de los objetos bajo medida. Dichas isosuperfices se ajustan al contorno de los objetos analizados, por lo que es posible conocer su forma y dimensiones aproximadas. En este ejemplo, la superficie superior e inferior de los objetos no se puede reconstruir fielmente debido a que no se dispone de información suficiente (dichas par-

tes no se iluminan directamente con ninguna de las ondas planas incidentes utilizadas).

Por último, resulta conveniente mencionar que el error RMS entre la solución de referencia (implementación para CPU) y la solución para múltiples GPU es menor que $3{,}6 \cdot 10^{-4}$ en las diferentes microarquitecturas utilizadas para resolver este problema (Fermi y Kepler).

Capítulo 6

Resultados

Índice

6.1. FMM 175
6.1.1. Tiempo de ejecución y escalabilidad 175
6.1.2. Eficiencia energética 186
6.1.3. Problemas de gran tamaño acústico 189
6.2. FMM-FFT 193
6.2.1. Tiempo de ejecución y escalabilidad 194
6.2.2. Eficiencia energética 200
6.2.3. Problema de gran tamaño acústico 203
6.3. Comparativa FMM vs. FMM-FFT 205
6.3.1. Tiempo de ejecución y escalabilidad 205
6.3.2. Eficiencia energética 209
6.3.3. Problema de gran tamaño acústico 211
6.4. bMVP, mMVP, *SmallBlocks*, *BigBlocks* y solución heterogénea 212
6.5. SRM 219
6.5.1. Problemas de caracterización de antenas 219
6.5.2. Problema de imagen electromagnética 220

A lo largo de este capítulo, se muestran los resultados computacionales más relevantes para las herramientas desarrolladas en esta Tesis. Se utilizan las métricas computacionales más comunes (tiempo de ejecución, consumo de memoria, aceleración o *speedup*, y eficiencia paralela), se comparan las soluciones paralelas heterogéneas con las implementaciones paralelas para procesadores convencionales, y también se analiza en detalle la eficiencia energética de las soluciones basadas en el FMM y el FMM-FFT. Además, se ha decidido presentar resultados obtenidos en sistemas muy diversos, cuyas características se detallan en el apéndice A, con el objetivo de mostrar la portabilidad de las soluciones desarrolladas a diferentes plataformas y de constatar su rendimiento en todas ellas.

El apartado 6.1 está dedicado a la implementación del algoritmo FMM para acústica y en él se muestran resultados relativos al tiempo de ejecución, la escalabilidad y la eficiencia energética, comparando la implementación heterogénea para CPU + GPU con la implementación paralela para procesadores convencionales. Además, se muestra la resolución de varios problemas de gran tamaño acústico en dos sistemas de memoria distribuida diferentes: un *cluster* convencional con múltiples nodos y un pequeño *cluster* de dos estaciones de trabajo con GPU.

Posteriormente, en el apartado 6.2, se lleva a cabo un estudio de los resultados obtenidos con la implementación del algoritmo FMM-FFT para acústica, similar al realizado previamente para el FMM.

En el apartado 6.3, se seleccionan los resultados más significativos de los dos apartados anteriores para realizar una comparación de las implementaciones del FMM y FMM-FFT, analizando las ventajas y los inconvenientes de cada algoritmo.

El apartado 6.4 se centra en la herramienta desarrollada para calcular la presión total en el problema de acústica, mostrando resultados de rendimiento para todas las técnicas presentadas en el capítulo 4: bMVP, mMVP, *SmallBlocks*, *BigBlocks*, y solución heterogénea para GPU y Xeon Phi.

Finalmente, en el apartado 6.5, se muestran resultados computacionales de las herramientas basadas en el método de reconstrucción de fuentes,

tanto para problemas de caracterización de antenas como para problemas de imagen electromagnética o *imaging*.

6.1. FMM

Los resultados que se muestran en esta sección se han obtenido usando tres sistemas diferentes: el *cluster* CMI de la Universidad de Oviedo (ver apéndice A.1), el Tesla MD SimCluster de Microway (ver A.2), y el *cluster* Mopar del grupo de investigación TSC-UNIOVI (ver A.3). Asimismo, también merece la pena señalar que, de forma general, todas las implementaciones analizadas en este apartado utilizan aritmética de simple precisión (reales de 32 bits y complejos de 64 bits).

6.1.1. Tiempo de ejecución y escalabilidad

Para llevar a cabo tanto el estudio del rendimiento computacional como de la eficiencia energética de la implementación del FMM para problemas de dispersión acústica, se han utilizado dos problemas diferentes, ambos de tamaño moderado.

En el primero de ellos se analiza un modelo de una aeronave de tamaño real (Airbus serie A3xx) con el objetivo de obtener la distribución de la presión acústica sobre su superficie a una frecuencia de 1 kHz. Para ello, la aeronave se discretiza utilizando una malla compuesta por 1009392 facetas triangulares (cada faceta se corresponde con una incógnita, por lo que $N = 1009392$), con aproximadamente 6 elementos por longitud de onda lineal. El ruido producido por dos de sus motores se ha modelado utilizando dos fuentes puntuales situadas debajo del ala izquierda (ver apartado 2.4). Además, se ha fijado como condición de parada del algoritmo iterativo un error residual $\epsilon \leq 10^{-2}$, el cual se alcanza tras un total de 89 iteraciones del FMM.

En el segundo problema se analiza una esfera con un diámetro de cuatro metros para obtener la distribución de presión acústica sobre su superficie cuando se ilumina con una onda plana a una frecuencia de 11,5 kHz (ver

apartado 2.4). Para ello, se genera una malla con 6001966 facetas triangulares ($N = 6001966$), dando como resultado aproximadamente 10 elementos por longitud de onda lineal. En este caso, se ha impuesto un error residual $\epsilon \leq 10^{-3}$ como condición de parada, lo que supone un total de 14 iteraciones del FMM.

Cluster CMI de UniOvi (A.1)	T_{MVP} [s]	T_{Total} [s]	S_{MVP}	S_{Total}	Tamaño grupos [λ]
1 nodo (8 hilos CPU)	192,91	17521,7	8,0	8,0	2,32
4 nodos (32 hilos CPU)	49,98	4553,9	30,9	30,8	2,32
8 nodos (64 hilos CPU)	26,03	2379,7	59,3	58,9	2,32
12 nodos (96 hilos CPU)	17,99	1650,5	85,8	84,9	2,32
16 nodos (128 hilos CPU)	13,94	1283,8	110,7	109,2	2,32
20 nodos (160 hilos CPU)	11,56	1072,4	133,5	130,7	2,11
24 nodos (192 hilos CPU)	9,98	923,9	154,6	151,7	2,32
28 nodos (224 hilos CPU)	8,80	817,0	175,3	171,6	2,32

Tabla 6.1: *Cluster* CMI de UniOvi (ver apéndice A.1). Tiempo MVP, tiempo total, S_{MVP}, S_{Total}, y tamaño de los grupos. Airbus serie A3xx ($N = 1009392$, $f = 1\,\text{kHz}$) usando FMM.

En la tabla 6.1, se muestran el tiempo por iteración (notado como T_{MVP}), el tiempo de ejecución total (T_{Total}) y el tamaño de los grupos para el problema de la aeronave variando el número de nodos del *cluster* CMI utilizados. El tiempo por iteración mostrado es el promedio de todas las iteraciones, mientras que el tiempo total representa el tiempo hasta obtener la solución (inicialización o *setup* del algoritmo más fase iterativa).

Además, la tabla 6.1 muestra dos valores de aceleración paralela o *speedup*, teniendo en cuenta el tiempo por iteración (S_{MVP}) y el tiempo total (S_{Total}), que se definen de la siguiente manera:

$$S_{\mathrm{MVP}}(p \cdot h) \approx \frac{8 \cdot T_{\mathrm{MVP}}(8)}{T_{\mathrm{MVP}}(p \cdot h)}, \tag{6.1a}$$

$$S_{\mathrm{Total}}(p \cdot h) \approx \frac{8 \cdot T_{\mathrm{Total}}(8)}{T_{\mathrm{Total}}(p \cdot h)}, \tag{6.1b}$$

donde p representa el número de procesos MPI utilizados y h es el número de hilos OpenMP por proceso. Puesto que el tiempo tomado como referencia para calcular la aceleración es el de la medida con un único nodo, en la cual se tiene que $p = 2$ (tantos procesos como procesadores) y $h = 4$ (mismos hilos por proceso que núcleos por procesador), dicho tiempo se multiplica por 8 (es decir, $p \cdot h$) para obtener una aproximación del tiempo de ejecución secuencial.

De los resultados mostrados en la tabla 6.1, merece la pena destacar los valores de aceleración obtenidos cuando se usa el *cluster* CMI al completo: por encima de 171 para un sistema con 224 núcleos con un problema de tamaño moderado (un millón de incógnitas). Dicha aceleración permite resolver en menos de catorce minutos un problema que requiere casi cinco horas en un único nodo.

La tabla 6.2 refleja el tiempo por iteración, el tiempo de ejecución total y el tamaño de los grupos para el problema de la esfera dependiendo del número de nodos del *cluster* CMI utilizados. Debido a que el tiempo tomado como referencia para calcular la aceleración es el de la medida con dos nodos (16 hilos en total), los valores para la aceleración paralela teniendo en cuenta el tiempo por iteración y el tiempo total que se muestran en la tabla 6.2 se definen, en este caso, como sigue:

$$S_{\mathrm{MVP}}(p \cdot h) \approx \frac{16 \cdot T_{\mathrm{MVP}}(16)}{T_{\mathrm{MVP}}(p \cdot h)}, \tag{6.2a}$$

$$S_{\mathrm{Total}}(p \cdot h) \approx \frac{16 \cdot T_{\mathrm{Total}}(16)}{T_{\mathrm{Total}}(p \cdot h)}. \tag{6.2b}$$

Cluster CMI de UniOvi (A.1)	T_{MVP} [s]	T_{Total} [s]	S_{MVP}	S_{Total}	Tamaño grupos [λ]
2 nodos (16 hilos CPU)	820,13	12935,5	16,0	16,0	2,01
4 nodos (32 hilos CPU)	419,98	6664,3	31,2	31,1	2,01
8 nodos (64 hilos CPU)	216,54	3481,2	60,6	59,5	2,01
12 nodos (96 hilos CPU)	150,16	2439,1	87,4	84,9	2,01
16 nodos (128 hilos CPU)	115,64	1896,4	113,5	109,1	2,01
20 nodos (160 hilos CPU)	95,00	1570,9	138,1	131,8	2,01
24 nodos (192 hilos CPU)	82,58	1376,4	158,9	150,4	2,01
28 nodos (224 hilos CPU)	72,26	1213,4	181,6	170,6	2,01

Tabla 6.2: *Cluster* CMI de UniOvi. Tiempo MVP, tiempo total, S_{MVP}, S_{Total}, y tamaño de los grupos. Esfera $\oslash 4\,\text{m}$ ($N = 6001966$, $f = 11{,}5\,\text{kHz}$) usando FMM.

Al igual que en el problema anterior, resulta especialmente destacable la aceleración que se logra al usar el *cluster* CMI al completo, en este caso por encima de 170 para un sistema con 224 núcleos con un problema de seis millones de incógnitas. La aceleración obtenida permite resolver en veinte minutos un problema que requiere más de tres horas y media usando dos nodos.

En la figura 6.1, se muestra de forma gráfica la aceleración teniendo en cuenta el tiempo por iteración (S_{MVP}) y el tiempo total (S_{Total}) para los dos problemas analizados usando el *cluster* CMI. El mejor resultado de escalabilidad se obtiene en el caso de la esfera con seis millones de incógnitas, en el que la aceleración lograda en el tiempo por iteración alcanza el valor

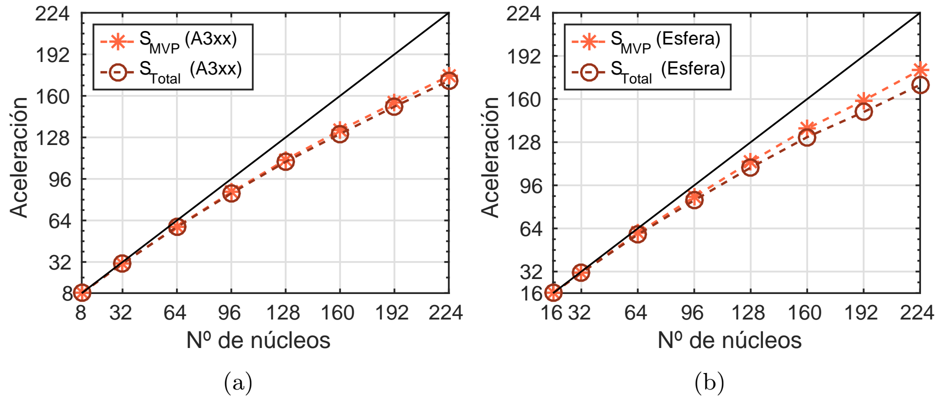


Figura 6.1: *Cluster* CMI de UniOvi (A.1). Aceleración en relación al número de núcleos teniendo en cuenta el tiempo por iteración (S_{MVP}) y el tiempo total (S_{Total}) usando la implementación paralela del FMM para CPU. (a) Airbus serie A3xx ($N = 1009392$, $f = 1\,\mathrm{kHz}$). (b) Esfera $\varnothing 4\,\mathrm{m}$ ($N = 6001966$, $f = 11{,}5\,\mathrm{kHz}$).

de 181,6.

Si bien la implementación para sistemas de memoria distribuida basados en procesadores convencionales [13, 145] es previa al trabajo desarrollado en esta Tesis, resulta conveniente mostrar el rendimiento que es capaz de ofrecer para tener un punto de referencia en las diferentes comparaciones con la implementación para sistemas heterogéneos basados en CPU + GPU.

La primera de las pruebas de la implementación paralela y heterogénea del FMM para CPU + GPU se ha llevado a cabo utilizando el Tesla MD SimCluster (A.2), cedido por la empresa estadounidense Microway. En esta ocasión, la aeronave analizada a 1 kHz se ha discretizado utilizando una malla más densa, con aproximadamente 9 elementos por longitud de onda lineal, compuesta por 2132704 facetas triangulares. La condición de parada ($\epsilon \leq 10^{-2}$) se alcanza, en este caso, tras 158 iteraciones del FMM. El problema de la esfera mantiene todos sus parámetros inalterados.

En la tabla 6.3, se muestran el tiempo por iteración, el tiempo total, el consumo de memoria y el tamaño de los grupos para el problema de la aeronave variando el número de nodos utilizados. En este caso, al tratarse de un

Tesla MD SimCluster de Microway (A.2)	T_{MVP} [s]	T_{Total} [s]	Memoria [GB]	Tamaño grupos [λ]
1 nodo (24 hilos CPU)	167,0	26740,7	7,5	1,40
2 nodos (48 hilos CPU)	86,3	13829,8	8,0	1,40
4 nodos (96 hilos CPU)	44,2	7095,9	9,4	1,40
1 nodo (24 hilos CPU + 2 GPU)	16,6	2704,1	2,9	3,26
2 nodos (48 hilos CPU + 4 GPU)	8,5	1400,3	3,4	3,26
4 nodos (96 hilos CPU + 8 GPU)	4,2	706,7	4,5	3,26

Tabla 6.3: Tesla MD SimCluster de Microway (ver apéndice A.2). Tiempo MVP, tiempo total, consumo de memoria, y tamaño de los grupos. Airbus serie A3xx ($N = 2132704$, $f = 1\,\text{kHz}$) usando FMM.

sistema heterogéneo, se han realizado medidas tanto de la implementación de referencia (FMM paralelo para sistemas con procesadores convencionales) como de la implementación heterogénea para CPU + GPU.

De los resultados presentados en la tabla 6.3, merece la pena destacar la aceleración obtenida al usar la implementación heterogénea en lugar de la implementación de referencia, ya que permite reducir los tiempos en un orden de magnitud. Otra ventaja adicional de la implementación para CPU + GPU reside en su menor consumo de memoria. Esto último se debe a la selección de grupos de mayor tamaño (que producen un operador de traslación de menor peso) para aumentar la carga de trabajo asociada a las etapas aceleradas (agregación, desagregación y cálculo de las interacciones cercanas) [116].

Por su parte, la tabla 6.4 muestra el tiempo por iteración, el tiempo total, el consumo de memoria y el tamaño de los grupos para el problema de

Tesla MD SimCluster de Microway (A.2)	T_{MVP} [s]	T_{Total} [s]	Memoria [GB]	Tamaño grupos [λ]
1 nodo (24 hilos CPU)	741,2	11879,6	18,8	1,59
2 nodos (48 hilos CPU)	376,2	6054,0	19,3	1,59
4 nodos (96 hilos CPU)	195,1	3134,6	18,9	1,69
1 nodo (24 hilos CPU + 2 GPU)	66,0	1177,1	6,4	3,80
2 nodos (48 hilos CPU + 4 GPU)	33,3	617,5	6,9	3,80
4 nodos (96 hilos CPU + 8 GPU)	17,9	343,7	8,0	3,80

Tabla 6.4: Tesla MD SimCluster de Microway. Tiempo MVP, tiempo total, consumo de memoria, y tamaño de los grupos. Esfera $\oslash 4\,\text{m}$ ($N = 6001966$, $f = 11{,}5\,\text{kHz}$) usando FMM.

la esfera, usando tanto la implementación de referencia (sólo CPU) como la implementación heterogénea (CPU + GPU). De nuevo, merece la pena incidir en la aceleración obtenida al usar la implementación heterogénea, que permite reducir los tiempos en un orden de magnitud respecto a la implementación de referencia. Asimismo, en este problema también se aprecia una reducción en el consumo de memoria en la implementación para CPU + GPU.

En la figura 6.2, se muestra de forma gráfica la eficiencia paralela teniendo en cuenta el tiempo por iteración (E_{MVP}) y el tiempo total (E_{Total}) usando las dos implementaciones del FMM (CPU y CPU + GPU) para los dos problemas analizados en el Tesla MD SimCluster.

Las expresiones utilizadas para definir la eficiencia paralela son las si-

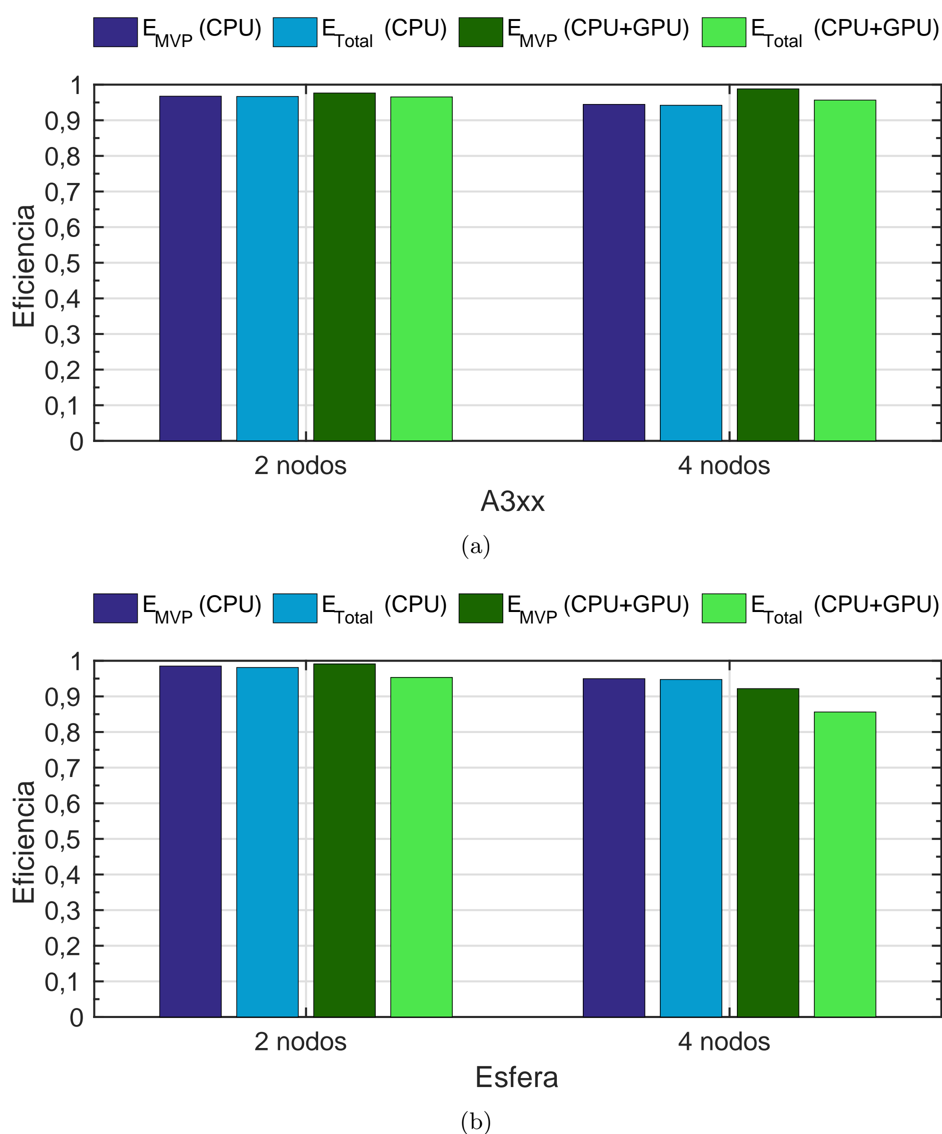


Figura 6.2: Tesla MD SimCluster de Microway (A.2). Eficiencia paralela para dos y cuatro nodos teniendo en cuenta el tiempo por iteración (E_{MVP}) y el tiempo total (E_{Total}) usando las implementaciones paralelas del FMM para CPU y CPU + GPU. (a) Airbus serie A3xx ($N = 2132704$, $f = 1\,\mathrm{kHz}$). (b) Esfera $\oslash 4\,\mathrm{m}$ ($N = 6001966$, $f = 11{,}5\,\mathrm{kHz}$).

guientes:

$$E_{\mathrm{MVP}}(n) = \frac{T_{\mathrm{MVP}}(1)}{n \cdot T_{\mathrm{MVP}}(n)}, \tag{6.3a}$$

$$E_{\mathrm{Total}}(n) = \frac{T_{\mathrm{Total}}(1)}{n \cdot T_{\mathrm{Total}}(n)}, \tag{6.3b}$$

donde n representa el número de nodos utilizados. El tiempo de la medida con un único nodo se usa como referencia para calcular la eficiencia.

Como se puede observar en la figura 6.2, en todos los casos la eficiencia está por encima del 85 %. Asimismo, merece la pena destacar la eficiencia de la fase iterativa (E_{MVP}), que llega a alcanzar el 97,7 % para el problema de la aeronave resuelto con la implementación heterogénea usando cuatro nodos.

Otra de las pruebas de la implementación del FMM para CPU + GPU se ha llevado a cabo utilizando el *cluster* Mopar del grupo TSC-UNIOVI (A.3), un sistema formado únicamente por dos estaciones de trabajo en las que se han instalado tarjetas gráficas de la serie Kepler. En este caso, los problemas analizados son idénticos a los resueltos en el *cluster* CMI de UniOvi: una aeronave y una esfera cuyos parámetros se han detallado al comienzo de este subapartado (ver 6.1.1).

La tabla 6.5 muestra el tiempo por iteración, el tiempo total, el consumo de memoria y el tamaño de los grupos para el problema de la aeronave. En este caso también se han realizado medidas tanto de la implementación de referencia (FMM para CPU) como de la implementación heterogénea para CPU + GPU.

De entre los resultados que se muestran en la tabla 6.5, merece la pena señalar la aceleración obtenida al usar la implementación heterogénea, la cual permite reducir los tiempos de simulación en más de 17,5 veces en comparación con la implementación para CPU. Además, como ya se ha visto en los casos anteriores, la implementación para CPU + GPU también permite reducir de forma notable los requisitos de memoria.

En la tabla 6.6, se muestran los datos relativos al tiempo de ejecución y

Cluster Mopar de TSC-UNIOVI (A.3)	T_{MVP} [s]	T_{Total} [s]	Memoria [GB]	Tamaño grupos [λ]
1 nodo (8 hilos CPU)	141,14	12888,4	6,8	1,40
2 nodos (16 hilos CPU)	73,54	6726,4	6,9	1,40
1 nodo (8 hilos CPU + 2 GPU)	7,27	681,4	1,9	4,55
2 nodos (16 hilos CPU + 4 GPU)	4,07	383,7	2,0	4,55

Tabla 6.5: *Cluster* Mopar de TSC-UNIOVI (ver apéndice A.3). Tiempo MVP, tiempo total, consumo de memoria, y tamaño de los grupos. Airbus serie A3xx ($N = 1009392$, $f = 1\,\text{kHz}$) usando FMM.

al consumo de memoria para el problema de la esfera usando las dos implementaciones desarrolladas del FMM para acústica (CPU y CPU + GPU). En esta ocasión, la implementación heterogénea permite dividir el tiempo de ejecución entre 21,2 y el consumo de memoria entre 4, tomando como referencia la implementación para CPU.

En la figura 6.3, se representa la eficiencia paralela teniendo en cuenta el tiempo por iteración y el tiempo total para los dos problemas analizados en el *cluster* Mopar. Puede apreciarse que la eficiencia siempre está por encima del 88 %. De igual manera, resulta interesante señalar la eficiencia de la fase iterativa (E_{MVP}), que supera el 95 % para el problema de la esfera resuelto con la implementación heterogénea en dos nodos.

Para finalizar con el análisis del rendimiento computacional de la implementación heterogénea del FMM, resulta especialmente interesante comparar los resultados mostrados en las tablas 6.1 y 6.2 con los resultados de las tablas 6.5 y 6.6. En dichas tablas, puede observarse cómo la implementación para CPU + GPU ejecutada en un único nodo con dos GPU es capaz de mejorar los tiempos obtenidos con la implementación para CPU ejecutada en un *cluster* compuesto por 28 nodos.

Cluster Mopar de TSC-UNIOVI (A.3)	T_{MVP} [s]	T_{Total} [s]	Memoria [GB]	Tamaño grupos [λ]
1 nodo (8 hilos CPU)	1336,12	21339,2	24,3	1,30
2 nodos (16 hilos CPU)	700,00	11234,7	24,9	1,30
1 nodo (8 hilos CPU + 2 GPU)	57,84	999,4	6,2	4,00
2 nodos (16 hilos CPU + 4 GPU)	30,13	530,2	6,3	4,00

Tabla 6.6: *Cluster* Mopar de TSC-UNIOVI. Tiempo MVP, tiempo total, consumo de memoria, y tamaño de los grupos. Esfera $\oslash 4\,\text{m}$ ($N = 6001966$, $f = 11{,}5\,\text{kHz}$) usando FMM.

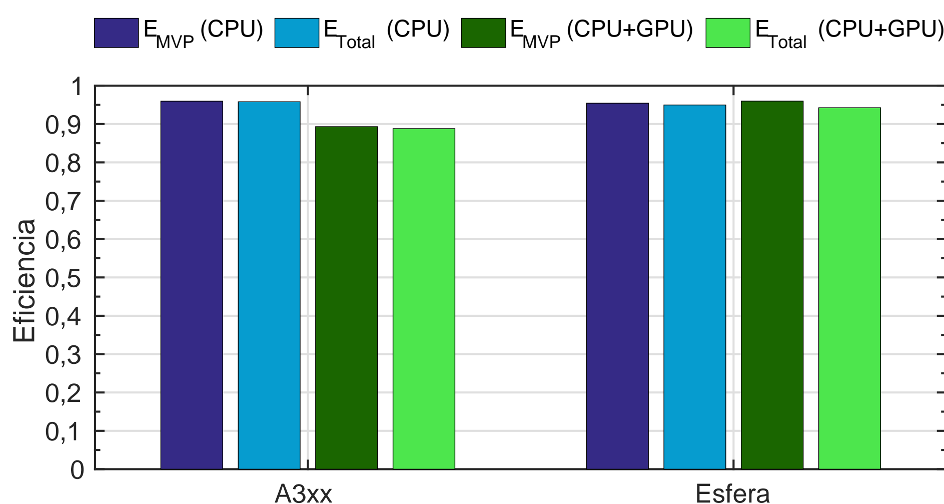


Figura 6.3: *Cluster* Mopar de TSC-UNIOVI. Eficiencia paralela para dos nodos teniendo en cuenta el tiempo por iteración y el tiempo total usando las implementaciones paralelas del FMM para CPU y CPU + GPU. Airbus serie A3xx ($N = 1009392$, $f = 1\,\text{kHz}$). Esfera $\oslash 4\,\text{m}$ ($N = 6001966$, $f = 11{,}5\,\text{kHz}$).

6.1.2. Eficiencia energética

Una vez analizados los datos relativos al tiempo de ejecución y la escalabilidad de la implementación para CPU + GPU del FMM, resulta conveniente analizar la eficiencia energética [103, 104, 105] de la solución presentada en esta Tesis.

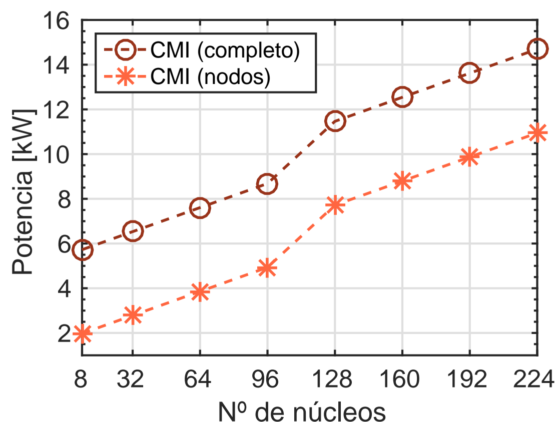


Figura 6.4: Estimación del consumo de potencia del *cluster* CMI de la Universidad de Oviedo (A.1).

Para comenzar, y como referencia, se utiliza nuevamente la implementación del FMM para CPU. En la figura 6.4, se muestra el consumo de potencia estimado del *cluster* CMI cuando se encuentra con carga de trabajo, para lo que se ha utilizado la herramienta web ofrecida por el fabricante del *cluster* [146]. En dicha figura, se detalla tanto el consumo del *cluster* completo (nodos de cálculo, electrónica de red, sistema de almacenamiento, etc.) como el consumo del *cluster* teniendo únicamente en cuenta los nodos de cálculo. Cabe mencionar que para los cálculos de la energía consumida sólo se tiene en cuenta la potencia requerida por los nodos de cálculo.

La figura 6.5 muestra una estimación del consumo de energía para los problemas analizados cuando se utiliza la implementación del FMM para CPU en el *cluster* CMI. Para calcular el consumo de energía se utilizan las medidas de tiempo de ejecución (tablas 6.1 y 6.2) y la estimación del consumo de potencia media en condiciones de carga (figura 6.4).

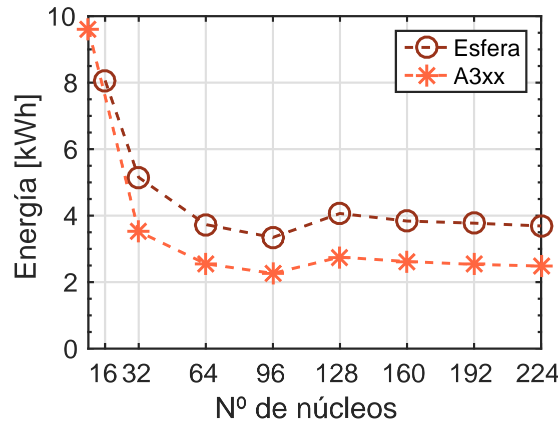


Figura 6.5: *Cluster* CMI de la Universidad de Oviedo. Estimación del consumo de energía usando la implementación paralela del FMM para CPU. Airbus serie A3xx ($N = 1009392$, $f = 1\,\mathrm{kHz}$). Esfera $\oslash 4\,\mathrm{m}$ ($N = 6001966$, $f = 11{,}5\,\mathrm{kHz}$).

En el caso de las pruebas realizadas en el *cluster* Mopar del grupo TSC-UNIOVI, todos los datos mostrados se basan en medidas realizadas directamente sobre dicho *cluster*. Por una parte, se dispone de las medidas de tiempo de ejecución (tablas 6.5 y 6.6). Por otra parte, los datos relativos a la energía (y la potencia máxima) se obtienen usando un medidor de energía Brennenstuhl PM231 [147].

Las tablas 6.7 y 6.8 muestran los resultados de consumo energético para los dos problemas analizados (aeronave y esfera, respectivamente) usando las implementaciones del FMM para CPU y CPU + GPU en el *cluster* Mopar. Si bien la potencia máxima es notablemente más elevada cuando se usan las CPU y las GPU de manera simultánea, la enorme reducción en el tiempo de ejecución convierte a la solución para CPU + GPU en la más eficiente también desde el punto de vista energético.

Por su parte, en la figura 6.6 se muestra gráficamente el consumo de energía para los dos problemas analizados en el *cluster* Mopar. Puede verse cómo la energía necesaria para resolver ambos problemas usando la implementación heterogénea se reduce prácticamente en un orden de magnitud

Cluster Mopar de TSC-UNIOVI (A.3)	Tiempo [s]	Energía [kWh]	Potencia máx. [W]
1 nodo (8 hilos CPU)	12888,4	0,75	229
2 nodos (16 hilos CPU)	6726,4	0,70	434
1 nodo (8 hilos CPU + 2 GPU)	681,4	0,08	578
2 nodos (16 hilos CPU + 4 GPU)	383,7	0,09	1128

Tabla 6.7: *Cluster* Mopar de TSC-UNIOVI. Tiempo total, consumo de energía y potencia máxima. Airbus serie A3xx ($N = 1009392$, $f = 1\,\text{kHz}$) usando FMM.

Cluster Mopar de TSC-UNIOVI (A.3)	Tiempo [s]	Energía [kWh]	Potencia máx. [W]
1 nodo (8 hilos CPU)	21339,2	1,20	228
2 nodos (16 hilos CPU)	11234,7	1,20	429
1 nodo (8 hilos CPU + 2 GPU)	999,4	0,13	582
2 nodos (16 hilos CPU + 4 GPU)	530,2	0,13	1134

Tabla 6.8: *Cluster* Mopar de TSC-UNIOVI. Tiempo total, consumo de energía y potencia máxima. Esfera $\oslash 4\,\text{m}$ ($N = 6001966$, $f = 11{,}5\,\text{kHz}$) usando FMM.

en comparación con la implementación para CPU.

Teniendo en cuenta los datos mostrados en las tablas 6.7 y 6.8, y en la figura 6.6, puede comprobarse que, incluso ante eventuales limitaciones en la potencia máxima disponible, resulta energéticamente más eficiente

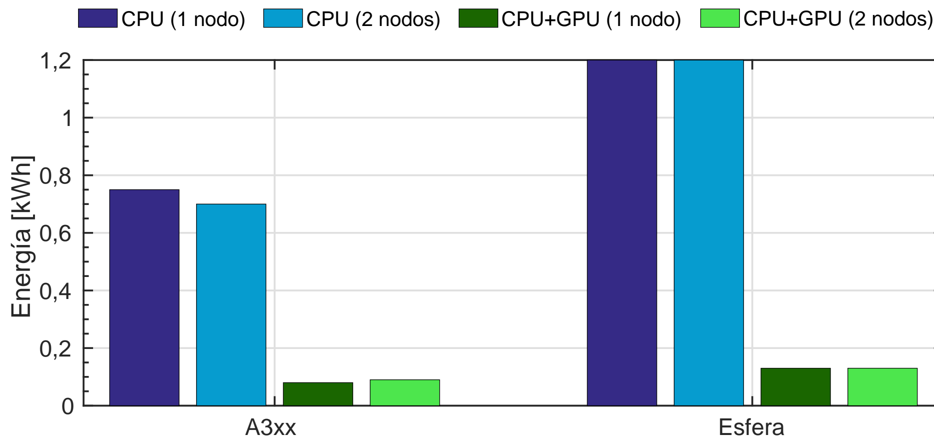


Figura 6.6: *Cluster* Mopar de TSC-UNIOVI. Consumo de energía usando las implementaciones paralelas del FMM para CPU y CPU + GPU. Airbus serie A3xx ($N = 1009392$, $f = 1\,\text{kHz}$). Esfera $\oslash 4\,\text{m}$ ($N = 6001966$, $f = 11{,}5\,\text{kHz}$).

recurrir a la solución heterogénea, aunque se reduzca el número de nodos a utilizar.

Por último, si se comparan los resultados mostrados en las figuras 6.5 y 6.6, puede apreciarse la magnitud de la mejora en la eficiencia energética que supone la implementación para CPU + GPU, ejecutada en el *cluster* Mopar, frente a la implementación para CPU, ejecutada en el *cluster* CMI.

6.1.3. Problemas de gran tamaño acústico

Para comprobar cómo la implementación paralela y heterogénea del FMM presentada en esta Tesis es capaz de aumentar el rango de problemas que se pueden resolver en estaciones de trabajo y pequeños sistemas de memoria distribuida, en este subapartado se muestran los datos relativos a la resolución de dos problemas de gran tamaño acústico. Nuevamente, se utiliza como referencia la implementación paralela para CPU ejecutada en el *cluster* CMI y se compara con la solución heterogénea para CPU + GPU ejecutada en el *cluster* Mopar.

El primero de los problemas consiste en la obtención de la distribución de la presión acústica en la superficie de una esfera con un diámetro de 20 m, sobre la que incide una onda plana con una frecuencia de 5 kHz. Para ello, se ha generado una malla compuesta por aproximadamente cincuenta millones de facetas triangulares ($N = 50003414$) con $\sim$13,6 elementos por longitud de onda lineal.

Diámetro de la esfera	20 m (294,12 λ)
Frecuencia	5 kHz
Número de incógnitas	50003414
Tamaño de los grupos	2,22 λ
Grupos no vacíos	81864
Direcciones del espacio $\boldsymbol{\kappa}$	1378
Error residual objetivo	$\epsilon \leq 10^{-2}$
Número de iteraciones	9
Tiempo por iteración	20 min 33 s
Tiempo total	3 h 59 min
Máximo uso de memoria por nodo	5,03 GB
Consumo total de memoria	125,99 GB

Tabla 6.9: Datos de la resolución de un problema acústico con cincuenta millones de incógnitas usando la implementación para memoria distribuida del FMM en el *cluster* CMI (A.1).

Las tablas 6.9 y 6.10 resumen los datos más importantes de la resolución de este primer problema en los *clusters* CMI y Mopar, respectivamente. En el caso la implementación para CPU ejecutada en el *cluster* CMI, la resolución de este problema requiere prácticamente cuatro horas y un total de 126 GB de memoria. Por su parte, cuando se usa el *cluster* Mopar junto con la implementación para CPU + GPU, el mismo problema puede resolverse en poco más de una hora y media, y usando 50 GB de memoria. Por tanto,

Diámetro de la esfera	20 m (294,12 λ)
Frecuencia	5 kHz
Número de incógnitas	50003414
Tamaño de los grupos	4,6 λ
Grupos no vacíos	18848
Direcciones del espacio $\boldsymbol{\kappa}$	4851
Error residual objetivo	$\epsilon \leq 10^{-2}$
Número de iteraciones	9
Tiempo por iteración	8 min
Tiempo total	1 h 38 min
Máximo uso de memoria por nodo	25,3 GB
Consumo total de memoria	50,3 GB

Tabla 6.10: Datos de la resolución de un problema acústico con cincuenta millones de incógnitas usando la implementación paralela y heterogénea del FMM en el *cluster* Mopar (A.3).

el tiempo total se divide entre 2,44, mientras que el tiempo por iteración se reduce 2,56 veces. Los requisitos de memoria, a su vez, se dividen entre 2,5.

En el segundo de los problemas, se busca conocer la distribución de la presión acústica sobre la superficie de una esfera con un diámetro de 20 m que se ilumina con una onda plana de 7 kHz. Dicha esfera se discretiza utilizando una malla compuesta por aproximadamente cien millones de facetas triangulares (N = 100001174), con ∼13,7 elementos por longitud de onda lineal.

En las tablas 6.11 y 6.12 se muestran los datos más significativos de la resolución de este problema en el *cluster* CMI (CPU) y Mopar (CPU + GPU). La solución para procesadores convencionales ejecutada en el *cluster*

Diámetro de la esfera	20 m (411,76 λ)
Frecuencia	7 kHz
Número de incógnitas	100001174
Tamaño de los grupos	2,52 λ
Grupos no vacíos	124176
Direcciones del espacio $\boldsymbol{\kappa}$	1711
Error residual objetivo	$\epsilon \leq 10^{-2}$
Número de iteraciones	10
Tiempo por iteración	55 min 8 s
Tiempo total	11 h 32 min
Máximo uso de memoria por nodo	10,9 GB
Consumo total de memoria	282,5 GB

Tabla 6.11: Datos de la resolución de un problema acústico con cien millones de incógnitas usando la implementación para memoria distribuida del FMM en el *cluster* CMI (A.1).

CMI requiere un total de once horas y media, y 283 GB de memoria. La implementación heterogénea ejecutada en el *cluster* Mopar es capaz de resolver el mismo problema en menos de cinco horas usando sólo 109 GB de memoria. En este caso, el tiempo total y el tiempo por iteración también se reducen 2,44 y 2,56 veces, respectivamente, mientras que los requisitos de memoria se dividen entre 2,6.

Además, si se tienen en cuenta las enormes diferencias en el hardware del *cluster* CMI (28 nodos, ver A.1) y del *cluster* Mopar (2 nodos, ver A.3), las mejoras en tiempo de ejecución y consumo de memoria en la resolución de ambos problemas son aún más notables.

Con los datos mostrados en las tablas 6.9, 6.10, 6.11 y 6.12, puede comprobarse cómo la implementación heterogénea del FMM para CPU + GPU

Diámetro de la esfera	20 m (411,76 λ)
Frecuencia	7 kHz
Número de incógnitas	100001174
Tamaño de los grupos	5,3 λ
Grupos no vacíos	27872
Direcciones del espacio $\boldsymbol{\kappa}$	6105
Error residual objetivo	$\epsilon \leq 10^{-2}$
Número de iteraciones	10
Tiempo por iteración	21 min 31 s
Tiempo total	4 h 44 min
Máximo uso de memoria por nodo	54,6 GB
Consumo total de memoria	109 GB

Tabla 6.12: Datos de la resolución de un problema acústico con cien millones de incógnitas usando la implementación paralela y heterogénea del FMM en el *cluster* Mopar (A.3).

presentada en esta Tesis permite resolver en simples estaciones de trabajo problemas que con una implementación paralela convencional para CPU requieren el uso de sistemas de memoria distribuida con un mayor tamaño, coste, consumo energético y complejidad.

6.2. FMM-FFT

Los resultados presentados en esta sección se han obtenido usando dos sistemas diferentes: el *cluster* CMI de la Universidad de Oviedo y el *cluster* Mopar de TSC-UNIOVI. Al igual que en el caso del FMM, todas las implementaciones del FMM-FFT que se han desarrollado utilizan aritmética de simple precisión.

6.2.1. Tiempo de ejecución y escalabilidad

Con el objetivo de generar una serie de resultados comparables con los obtenidos utilizando el FMM, se ha decidido analizar los mismos problemas (aeronave con un millón de incógnitas y esfera con seis millones de incógnitas) que en el caso del FMM, cuyos parámetros se detallan al comienzo del punto 6.1.1. Asimismo, cabe mencionar que en ambos casos el número de iteraciones necesarias para alcanzar el error residual objetivo es el mismo (89 iteraciones para el caso de la aeronave y 14 para el caso de la esfera).

Cluster CMI de UniOvi (A.1)	T_{MVP} [s]	T_{Total} [s]	S_{MVP}	S_{Total}	Tamaño grupos [λ]
1 nodo (8 hilos CPU)	137,56	12552,6	8,0	8,0	1,69
4 nodos (32 hilos CPU)	32,85	3029,7	33,5	33,1	1,69
8 nodos (64 hilos CPU)	17,02	1587,5	64,7	63,3	1,69
12 nodos (96 hilos CPU)	11,69	1101,9	94,1	91,1	1,69
16 nodos (128 hilos CPU)	9,07	865,9	121,3	116,0	1,59
20 nodos (160 hilos CPU)	7,32	703,1	150,3	142,8	1,69
24 nodos (192 hilos CPU)	6,25	604,8	176,1	166,0	1,69
28 nodos (224 hilos CPU)	5,59	548,2	196,8	183,2	1,59

Tabla 6.13: *Cluster* CMI de UniOvi. Tiempo MVP, tiempo total, S_{MVP}, S_{Total}, y tamaño de los grupos. Airbus serie A3xx ($N = 1009392$, $f = 1\,\text{kHz}$) usando FMM-FFT.

La tabla 6.13 muestra el tiempo por iteración, el tiempo total y el tamaño de los grupos para el problema de la aeronave utilizando diferente

número de nodos del *cluster* CMI. Además, se presentan los valores de aceleración teniendo en cuenta el tiempo por iteración y el tiempo total, calculados utilizando las expresiones (6.1a) y (6.1b), respectivamente.

Merece la pena destacar la aceleración obtenida cuando se usa el *cluster* CMI al completo, que se sitúa por encima de 183 para un sistema con 224 núcleos. Usando la implementación paralela del FMM-FFT para CPU con todos los nodos del *cluster*, puede resolverse en nueve minutos un problema que requiere tres horas y media en un único nodo.

Cluster CMI de UniOvi (A.1)	T_{MVP} [s]	T_{Total} [s]	S_{MVP}	S_{Total}	Tamaño grupos [λ]
2 nodos (16 hilos CPU)	260,38	4414,3	16,0	16,0	1,17
4 nodos (32 hilos CPU)	131,45	2390,0	31,7	29,6	1,17
8 nodos (64 hilos CPU)	67,16	1379,3	62,0	51,2	1,17
12 nodos (96 hilos CPU)	45,55	1035,6	91,5	68,2	1,17
16 nodos (128 hilos CPU)	35,05	866,8	118,9	81,5	1,17
20 nodos (160 hilos CPU)	28,46	757,8	146,4	93,2	1,17
24 nodos (192 hilos CPU)	23,48	675,4	177,4	104,6	1,17
28 nodos (224 hilos CPU)	20,81	630,5	200,2	112,0	1,17

Tabla 6.14: *Cluster* CMI de UniOvi. Tiempo MVP, tiempo total, S_{MVP}, S_{Total}, y tamaño de los grupos. Esfera $\oslash 4\,\mathrm{m}$ ($N = 6001966$, $f = 11{,}5\,\mathrm{kHz}$) usando FMM-FFT.

Por su parte, en la tabla 6.14 se muestra el tiempo por iteración, el tiempo de ejecución total y el tamaño de los grupos para el problema de la esfera. En este caso, la aceleración paralela teniendo en cuenta el tiempo

por iteración y el tiempo total se calculan utilizando las expresiones (6.2a) y (6.2b), respectivamente (en este problema el tiempo tomado como referencia para calcular la aceleración es el de la medida con dos nodos).

La aceleración que se logra en este problema al usar el *cluster* CMI completo es de 112, lo que permite resolver en diez minutos y medio un problema que requiere casi una hora y cuarto usando dos nodos.

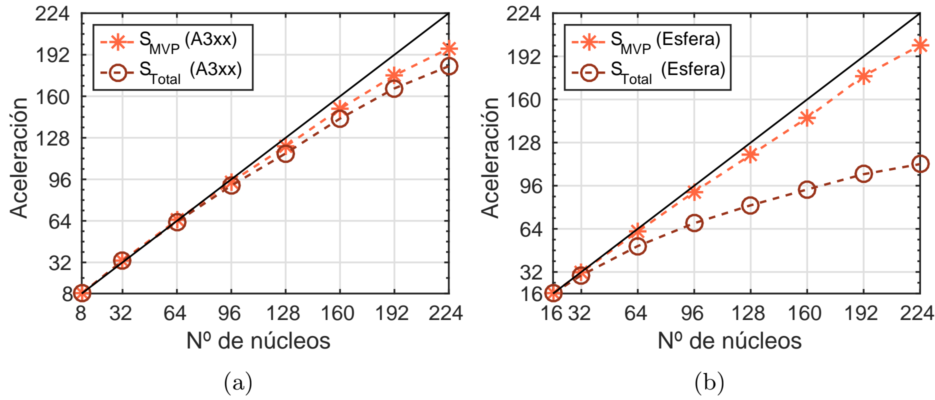


Figura 6.7: *Cluster* CMI de UniOvi. Aceleración en relación al número de núcleos teniendo en cuenta el tiempo por iteración (S_{MVP}) y el tiempo total (S_{Total}) usando la implementación paralela del FMM-FFT para CPU. (a) Airbus serie A3xx ($N = 1009392$, $f = 1\,\mathrm{kHz}$). (b) Esfera $\oslash 4\,\mathrm{m}$ ($N = 6001966$, $f = 11{,}5\,\mathrm{kHz}$).

En la figura 6.7, se representan gráficamente los valores de aceleración teniendo en cuenta el tiempo por iteración y el tiempo total para los dos problemas analizados, usando el *cluster* CMI con la implementación paralela del FMM-FFT.

El caso de la esfera con seis millones de incógnitas (figura 6.7(b)) ejemplifica un caso especialmente curioso: gran escalabilidad teniendo en cuenta el tiempo por iteración ($S_{\mathrm{MVP}}(224) = 200$) y una escalabilidad pobre teniendo en cuenta el tiempo total ($S_{\mathrm{Total}}(224) = 112$). Esta diferencia en la escalabilidad se debe a que, en el problema de la esfera, la fase de inicialización tiene un gran peso comparada con la fase iterativa. Debido al reducido tamaño de los grupos (se optimiza el tiempo por iteración [116]), en la inicialización se debe calcular un operador de traslación de gran ta-

maño, mientras que la fase iterativa se limita a sólo catorce iteraciones. Por tanto, en este caso, la peor escalabilidad de la inicialización (en concreto del cálculo del operador de traslación) limita la escalabilidad de la solución global.

Por su parte, las pruebas de la implementación heterogénea del FMM-FFT para CPU + GPU se han llevado a cabo en el *cluster* Mopar de TSC-UNIOVI, utilizando los mismos problemas que se han resuelto en el *cluster* CMI de UniOvi con la implementación paralela del FMM-FFT para CPU.

Cluster Mopar de TSC-UNIOVI (A.3)	T_{MVP} [s]	T_{Total} [s]	Memoria [GB]	Tamaño grupos [λ]
1 nodo (8 hilos CPU)	106,38	9744,3	10,3	1,20
2 nodos (16 hilos CPU)	55,46	5098,6	10,8	1,20
1 nodo (8 hilos CPU + 2 GPU)	8,73	815,7	1,7	5,06
2 nodos (16 hilos CPU + 4 GPU)	4,84	454,3	1,8	5,06

Tabla 6.15: *Cluster* Mopar de TSC-UNIOVI. Tiempo MVP, tiempo total, consumo de memoria, y tamaño de los grupos. Airbus serie A3xx ($N = 1009392$, $f = 1\,\mathrm{kHz}$) usando FMM-FFT.

La tabla 6.15 muestra el tiempo por iteración, el tiempo total, el consumo de memoria y el tamaño de los grupos para el problema de la aeronave. Como en el caso del FMM, se han realizado medidas tanto de la implementación de referencia (CPU) como de la implementación heterogénea (CPU + GPU).

Como se puede observar en la tabla 6.15, la aceleración obtenida al usar la implementación heterogénea permite dividir entre once los tiempos de ejecución, en comparación con la implementación paralela para CPU. De igual forma, la implementación para CPU + GPU permite reducir el uso de memoria a una sexta parte.

Cluster Mopar de TSC-UNIOVI (A.3)	T_{MVP} [s]	T_{Total} [s]	Memoria [GB]	Tamaño grupos [λ]
1 nodo (8 hilos CPU)	485,34	8381,9	62,8	0,72
2 nodos (16 hilos CPU)	232,15	4974,9	89,4	0,63
1 nodo (8 hilos CPU + 2 GPU)	27,76	521,4	9,5	2,71
2 nodos (16 hilos CPU + 4 GPU)	15,60	301,6	9,7	2,71

Tabla 6.16: *Cluster* Mopar de TSC-UNIOVI. Tiempo MVP, tiempo total, consumo de memoria, y tamaño de los grupos. Esfera $\oslash 4\,\mathrm{m}$ ($N = 6001966$, $f = 11{,}5\,\mathrm{kHz}$) usando FMM-FFT.

En la tabla 6.16, se presentan los datos de tiempo de ejecución, consumo de memoria y tamaño de los grupos para el problema de la esfera usando las implementaciones del FMM-FFT para CPU y CPU + GPU en el *cluster* Mopar. En este caso, la implementación heterogénea permite dividir el tiempo de ejecución entre 16 y el consumo de memoria entre 9, usando como referencia la implementación para CPU.

De los resultados que se muestran en la tabla 6.16, cabe mencionar que, en la medida para un nodo usando la implementación para CPU, no ha sido posible utilizar el tamaño de grupo óptimo ($0{,}63\,\lambda$) debido a limitaciones de memoria (64 GB). Por tanto, se ha forzado el uso de un tamaño de grupo de $0{,}72\,\lambda$ que, según las estimaciones del propio algoritmo para la selección del tamaño de grupo óptimo [116], debería producir un tiempo por iteración tan sólo un 1 % superior.

En la figura 6.8, se muestra gráficamente la eficiencia paralela teniendo en cuenta el tiempo por iteración y el tiempo total para los dos problemas analizados en el *cluster* Mopar (los cálculos se han realizado siguiendo las expresiones (6.3a) y (6.3b)).

En el caso de la implementación del FMM-FFT para CPU, y debido al

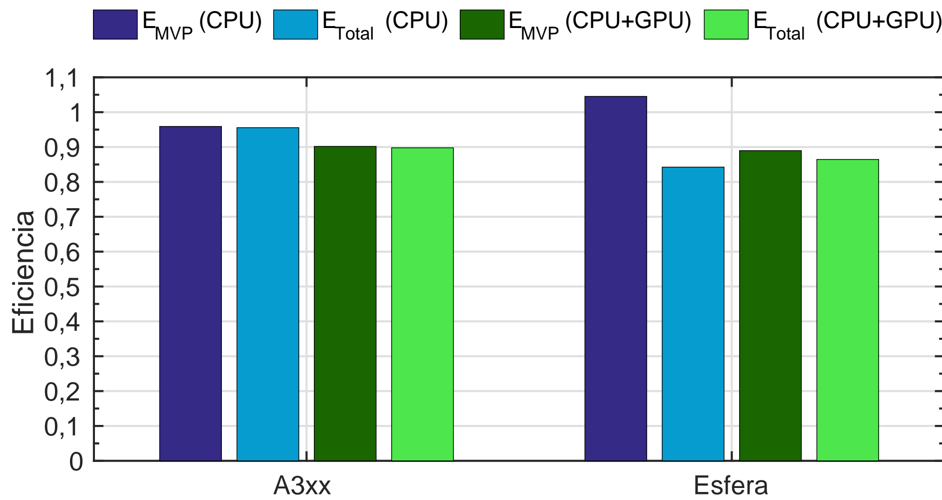


Figura 6.8: *Cluster* Mopar de TSC-UNIOVI. Eficiencia paralela para dos nodos teniendo en cuenta el tiempo por iteración (E_{MVP}) y el tiempo total (E_{Total}) usando las implementaciones paralelas del FMM-FFT para CPU y CPU + GPU. Airbus serie A3xx ($N = 1009392$, $f = 1\,\mathrm{kHz}$). Esfera $\oslash 4\,\mathrm{m}$ ($N = 6001966$, $f = 11{,}5\,\mathrm{kHz}$).

cambio en el tamaño de los grupos para la medida de referencia (un nodo), en la figura 6.8 puede apreciarse que se produce superescalabilidad cuando se tiene en cuenta el tiempo por iteración ($E_{\mathrm{MVP}}(2) = 1{,}045$ para CPU). Asimismo, merece la pena resaltar que la eficiencia es superior al 84 % en todos los casos.

Por último, resulta especialmente interesante estudiar el tiempo que requiere la resolución de los dos problemas analizados (aeronave y esfera) con las diferentes implementaciones del FMM-FFT. Si se comparan los resultados de la aeronave mostrados en la tabla 6.13 con los resultados de la tabla 6.15, puede observarse que la implementación para CPU + GPU, ejecutada en el *cluster* Mopar (dos nodos), es capaz de mejorar los tiempos obtenidos con la implementación para CPU, ejecutada en el *cluster* CMI (28 nodos). Por su parte, si se comparan las tablas 6.14 y 6.16, correspondientes a la esfera, puede comprobarse que la implementación para CPU + GPU ejecutada en un único nodo del *cluster* Mopar mejora el tiempo logrado

con la implementación para CPU usando el *cluster* CMI completo.

6.2.2. Eficiencia energética

Tras analizar los resultados relativos al tiempo de ejecución y a la escalabilidad de las diferentes implementaciones del FMM-FFT, a continuación se estudia la eficiencia energética de ambas soluciones (para CPU y para CPU + GPU).

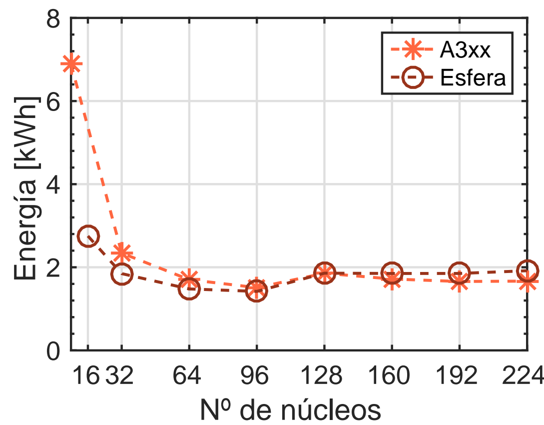


Figura 6.9: *Cluster* CMI de la Universidad de Oviedo. Estimación del consumo de energía usando la implementación paralela del FMM-FFT para CPU. Airbus serie A3xx ($N = 1009392$, $f = 1\,\text{kHz}$). Esfera $\oslash 4\,\text{m}$ ($N = 6001966$, $f = 11{,}5\,\text{kHz}$).

Partiendo de los datos de potencia del *cluster* CMI mostrados en la figura 6.4 y de las medidas de tiempo de ejecución del FMM-FFT para CPU de las tablas 6.13 y 6.14, en la figura 6.9 se muestra una estimación de la energía necesaria para resolver los dos problemas analizados usando el FMM-FFT en el *cluster* CMI.

Por su parte, en las tablas 6.17 y 6.18, se muestran los resultados de consumo energético para los dos problemas analizados usando las implementaciones del FMM-FFT para CPU y CPU + GPU en el *cluster* Mopar. De nuevo, merece la pena resaltar que tanto los datos de energía como

Cluster Mopar de TSC-UNIOVI (A.3)	Tiempo [s]	Energía [kWh]	Potencia máx. [W]
1 nodo (8 hilos CPU)	9744,3	0,55	222
2 nodos (16 hilos CPU)	5098,6	0,55	418
1 nodo (8 hilos CPU + 2 GPU)	815,7	0,10	563
2 nodos (16 hilos CPU + 4 GPU)	454,3	0,11	1092

Tabla 6.17: *Cluster* Mopar de TSC-UNIOVI. Tiempo total, consumo de energía y potencia máxima. Airbus serie A3xx ($N = 1009392$, $f = 1\,\mathrm{kHz}$) usando FMM-FFT.

de potencia máxima se han medido directamente, utilizando un sistema Brennenstuhl PM231 [147]. En ambos problemas, la potencia máxima es mucho más elevada cuando se usa la solución heterogénea, en comparación con la implementación para CPU. Sin embargo, la reducción en el tiempo de ejecución hace que la implementación del FMM-FFT para CPU + GPU sea más eficiente teniendo en cuenta la energía consumida.

En la figura 6.10 se muestra gráficamente el consumo de energía para los dos problemas analizados en el *cluster* Mopar usando el FMM-FFT. La energía necesaria para resolver ambos problemas usando la implementación heterogénea se reduce entre 5 y 7,5 veces en comparación con la implementación para CPU.

Finalmente, si se comparan las figuras 6.9 y 6.10, puede observarse la drástica disminución del consumo energético proporcionada por la implementación del FMM-FFT para CPU + GPU, ejecutada en el *cluster* Mopar, frente a la implementación del FMM-FFT para CPU, ejecutada en el *cluster* CMI.

Cluster Mopar de TSC-UNIOVI (A.3)	Tiempo [s]	Energía [kWh]	Potencia máx. [W]
1 nodo (8 hilos CPU)	8381,9	0,45	221
2 nodos (16 hilos CPU)	4974,9	0,50	418
1 nodo (8 hilos CPU + 2 GPU)	521,4	0,06	564
2 nodos (16 hilos CPU + 4 GPU)	301,6	0,07	1104

Tabla 6.18: *Cluster* Mopar de TSC-UNIOVI. Tiempo total, consumo de energía y potencia máxima. Esfera ⌀4 m ($N = 6001966$, $f = 11{,}5$ kHz) usando FMM-FFT.

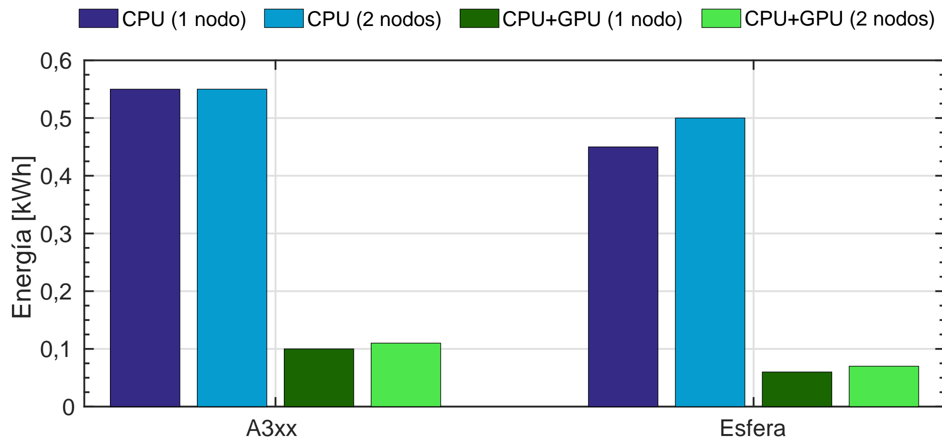


Figura 6.10: *Cluster* Mopar de TSC-UNIOVI. Consumo de energía usando las implementaciones paralelas del FMM-FFT para CPU y CPU + GPU. Airbus serie A3xx ($N = 1009392$, $f = 1$ kHz). Esfera ⌀4 m ($N = 6001966$, $f = 11{,}5$ kHz).

6.2.3. Problema de gran tamaño acústico

Con el objetivo de demostrar las capacidades de las implementaciones del FMM-FFT presentadas en esta Tesis, en este punto se muestran los datos relativos a la resolución de un problema de gran tamaño acústico. La implementación del FMM-FFT para CPU se prueba sobre el *cluster* CMI, mientras que la solución heterogénea se ejecuta en el *cluster* Mopar.

En este caso, el problema resuelto utilizando el FMM-FFT se corresponde con el problema de cincuenta millones de incógnitas previamente utilizado en el apartado dedicado al FMM (ver punto 6.1.3).

Diámetro de la esfera	20 m (294,12 λ)
Frecuencia	5 kHz
Número de incógnitas	50003414
Tamaño de los grupos	1,4 λ
Grupos no vacíos	203584
Grupos totales	9528128
Direcciones del espacio $\boldsymbol{\kappa}$	666
Error residual objetivo	$\epsilon \leq 10^{-2}$
Número de iteraciones	9
Tiempo por iteración	5 min 31 s
Tiempo total	1 h 54 min
Máximo uso de memoria por nodo	20,2 GB
Consumo total de memoria	560,3 GB

Tabla 6.19: Datos de la resolución de un problema acústico con cincuenta millones de incógnitas usando la implementación para memoria distribuida del FMM-FFT en el *cluster* CMI (A.1).

En las tablas 6.19 y 6.20, se resumen los datos más relevantes de la

Diámetro de la esfera	20 m (294,12 λ)
Frecuencia	5 kHz
Número de incógnitas	50003414
Tamaño de los grupos	2,58 λ
Grupos no vacíos	60332
Grupos totales	1481544
Direcciones del espacio $\boldsymbol{\kappa}$	1830
Error residual objetivo	$\epsilon \leq 10^{-2}$
Número de iteraciones	9
Tiempo por iteración	3 min 5 s
Tiempo total	44 min 8 s
Máximo uso de memoria por nodo	50 GB
Consumo total de memoria	99,8 GB

Tabla 6.20: Datos de la resolución de un problema acústico con cincuenta millones de incógnitas usando la implementación paralela y heterogénea del FMM-FFT en el *cluster* Mopar (A.3).

resolución del problema con cincuenta millones de incógnitas usando los *clusters* CMI y Mopar, respectivamente.

Se debe señalar que, en el caso de la implementación del FMM-FFT para CPU ejecutada en el *cluster* CMI (tabla 6.19), debido a limitaciones de memoria y a un excesivo peso del cálculo del operador de traslación durante la inicialización, no resulta conveniente utilizar el tamaño de grupo que minimiza el tiempo por iteración (0,91 λ). En su lugar, se ha forzado el uso de un tamaño de grupo de 1,4 λ que, según las estimaciones del algoritmo encargado de la selección del tamaño de los grupos [116], produce un tiempo por iteración un 35 % superior, pero reduce notablemente el tiempo de *setup* y el consumo de memoria.

En el caso la implementación del FMM-FFT para procesadores convencionales ejecutada en el *cluster* CMI, la resolución de este problema requiere prácticamente dos horas y un total de 560 GB de memoria. Por su parte, con la implementación del FMM-FFT para CPU + GPU y utilizando el *cluster* Mopar, el mismo problema puede resolverse en 44 minutos, y usando 100 GB de memoria.

En comparación, el tiempo por iteración se divide entre 1,79, mientras que el tiempo total se reduce 2,58 veces. Por su parte, los requisitos de memoria se dividen entre 5,6.

6.3. Comparativa FMM vs. FMM-FFT

Una vez analizados los resultados obtenidos con las diferentes implementaciones del FMM y del FMM-FFT en los dos aparatados anteriores, parece conveniente seleccionar una muestra de dichos resultados para examinar las ventajas y los inconvenientes de cada algoritmo.

Los datos seleccionados y presentados en esta sección se han obtenido usando el *cluster* CMI de la Universidad de Oviedo y el *cluster* Mopar del grupo TSC-UNIOVI, puesto que son los sistemas en los que se han analizado las diferentes soluciones basadas en el FMM y el FMM-FFT.

6.3.1. Tiempo de ejecución y escalabilidad

Para llevar a cabo las comparativas de rendimiento computacional y de eficiencia energética, se ha utilizado el problema de mayor tamaño de entre los dos analizados en los apartados anteriores: la esfera con seis millones de incógnitas cuyos parámetros se detallan al comienzo del punto 6.1.1.

En la tabla 6.21, se muestra una comparativa del tiempo de ejecución (por iteración y total) para la resolución del problema de la esfera en el *cluster* CMI, utilizando las implementaciones paralelas para procesadores convencionales del FMM y del FMM-FFT.

	FMM		FMM-FFT	
Cluster CMI de UniOvi (A.1)	T_{MVP} [s]	T_{Total} [s]	T_{MVP} [s]	T_{Total} [s]
16 hilos CPU	820,13	12935,5	260,38	4414,3
32 hilos CPU	419,98	6664,3	131,45	2390,0
64 hilos CPU	216,54	3481,2	67,16	1379,3
96 hilos CPU	150,16	2439,1	45,55	1035,6
128 hilos CPU	115,64	1896,4	35,05	866,8
160 hilos CPU	95,00	1570,9	28,46	757,8
192 hilos CPU	82,58	1376,4	23,48	675,4
224 hilos CPU	72,26	1213,4	20,81	630,5

Tabla 6.21: *Cluster* CMI de UniOvi. Tiempo MVP y tiempo total usando FMM y FMM-FFT. Esfera $\oslash 4\,\text{m}$ ($N = 6001966$, $f = 11{,}5\,\text{kHz}$).

En este caso, la disminución en el tiempo de ejecución que se logra con el FMM-FFT es notable, teniendo en cuenta que se trata de un problema de tamaño moderado. Si se considera el tiempo total, la reducción de tiempos se encuentra entre 2 y 3, dependiendo del número de nodos empleados. Por su parte, si se tiene en cuenta el tiempo por iteración, la ganancia es aún mayor, con unos tiempos que llegan a dividirse entre 3,5 cuando se usa el *cluster* completo.

	FMM		FMM-FFT	
Cluster Mopar de TSC-UNIOVI (A.3)	T_{MVP} [s]	T_{Total} [s]	T_{MVP} [s]	T_{Total} [s]
8 hilos CPU	1336,12	21339,2	485,34	8381,9
16 hilos CPU	700,00	11234,7	232,15	4974,9
8 hilos CPU + 2 GPU	57,84	999,4	27,76	521,4
16 hilos CPU + 4 GPU	30,13	530,2	15,60	301,6

Tabla 6.22: *Cluster* Mopar de TSC-UNIOVI. Tiempo MVP y tiempo total usando FMM y FMM-FFT. Esfera $\oslash 4\,\text{m}$ ($N = 6001966$, $f = 11{,}5\,\text{kHz}$).

La tabla 6.22 muestra la comparativa de los tiempos de ejecución para la resolución del problema de la esfera en el *cluster* Mopar, utilizando todas las implementaciones desarrolladas: FMM para CPU y CPU + GPU, y FMM-FFT para CPU y CPU + GPU.

En esta ocasión, conviene analizar por separado las implementaciones para procesadores convencionales (dos primeras filas de la tabla 6.22) y las implementaciones heterogéneas (dos últimas filas de la tabla 6.22).

En el caso de las implementaciones para CPU ejecutadas en el *cluster* Mopar, el FMM-FFT permite reducir a menos de la mitad el tiempo total, mientras que el tiempo por iteración se reduce a aproximadamente un tercio. Por su parte, cuando se utilizan las implementaciones heterogéneas, el FMM-FFT no llega a dividir el tiempo total entre dos en comparación con el FMM, mientras que si se tiene en cuenta el tiempo por iteración, los tiempos se reducen a prácticamente la mitad.

Si se comparan las tablas a partir de las cuales se obtienen los datos mostrados en la tabla 6.22 (tablas 6.6 y 6.16), puede verse que la mejora en los tiempos de ejecución del FMM-FFT respecto del FMM es mayor cuando se utilizan grupos con un tamaño reducido, ya que se aumenta el peso de la traslación, que es la etapa acelerada en el FMM-FFT. Por contra, los requisitos de memoria del FMM-FFT pueden llegar a multiplicar por 3,6 los del FMM cuando se utilizan grupos con un tamaño muy reducido.

En la figura 6.11, se representa gráficamente la aceleración, considerando el tiempo por iteración y el tiempo total, en la resolución del problema de la esfera con seis millones de incógnitas usando las implementaciones del FMM y del FMM-FFT para procesadores convencionales en el *cluster* CMI.

Teniendo en cuenta el tiempo por iteración, se aprecia que el FMM-FFT presenta una mayor escalabilidad en comparación con el FMM. Por contra, si se tiene en cuenta el tiempo total, la escalabilidad del FMM-FFT se ve penalizada en los casos en los que la fase de inicialización tiene demasiado peso en comparación con la fase iterativa (ver punto 6.2.1).

Por último, la figura 6.12 muestra la eficiencia paralela (calculada si-

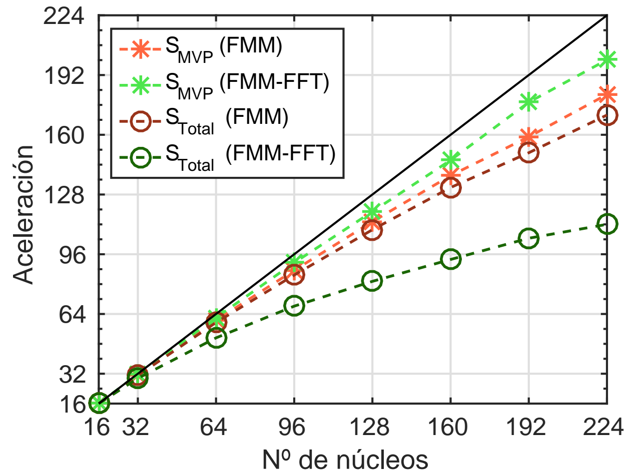


Figura 6.11: *Cluster* CMI de UniOvi. Comparativa de la aceleración en relación al número de núcleos teniendo en cuenta el tiempo por iteración (S_{MVP}) y el tiempo total (S_{Total}) usando las implementaciones paralelas del FMM y del FMM-FFT para CPU. Esfera ⊘4 m ($N = 6001966$, $f = 11{,}5$ kHz).

guiendo las expresiones (6.3a) y (6.3b)) considerando el tiempo por iteración y el tiempo total para el problema de la esfera resuelto en el *cluster* Mopar.

El comportamiento en el caso de las implementaciones para CPU es similar al visto en el caso del *cluster* CMI: el FMM-FFT presenta una mejor escalabilidad cuando se tiene en cuenta el tiempo por iteración, pero su escalabilidad puede verse penalizada cuando la fase de inicialización tiene demasiado peso en comparación con la fase iterativa. En el caso de las implementaciones heterogéneas, el FMM logra una mayor eficiencia paralela que el FMM-FFT. En todos los casos, la eficiencia siempre se mantiene por encima del 84 %.

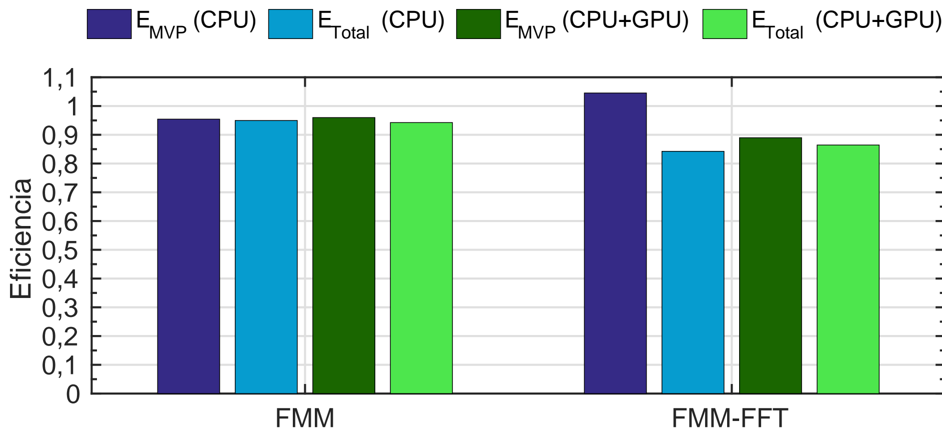


Figura 6.12: *Cluster* Mopar de TSC-UNIOVI. Comparativa de la eficiencia paralela para dos nodos teniendo en cuenta el tiempo por iteración (E_{MVP}) y el tiempo total (E_{Total}) usando las implementaciones paralelas del FMM y del FMM-FFT para CPU y CPU + GPU. Esfera ⌀4 m ($N = 6001966$, $f = 11{,}5\,\mathrm{kHz}$).

6.3.2. Eficiencia energética

En lo relativo a la eficiencia energética de las soluciones basadas en el FMM y en el FMM-FFT, los resultados esperados son claros: si la potencia requerida se mantiene constante (ver medidas en las tablas 6.8 y 6.18) y el tiempo de ejecución se reduce de forma notable, las implementaciones basadas en el FMM-FFT deben arrojar mejores resultados de consumo energético.

La figura 6.13 permite comparar la estimación del consumo de energía al usar la implementación del FMM y del FMM-FFT para CPU en el *cluster* CMI. Para calcular los consumos de energía mostrados, se utilizan las medidas de tiempo de ejecución (tablas 6.2 y 6.14) y la estimación de la potencia requerida (figura 6.4). Gracias a la reducción en los tiempos de ejecución, la implementación para CPU del FMM-FFT es claramente más eficiente desde el punto de vista energético.

Por su parte, en la figura 6.14 se muestra una comparación del consumo

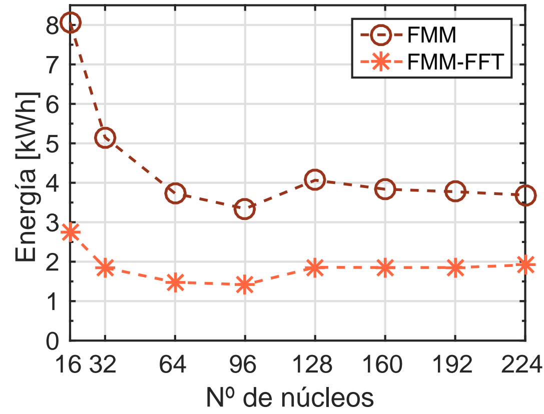


Figura 6.13: *Cluster* CMI de la Universidad de Oviedo. Comparativa de la estimación del consumo de energía usando la implementación paralela del FMM y del FMM-FFT para CPU. Esfera ⌀4 m ($N = 6001966$, $f = 11{,}5$ kHz).

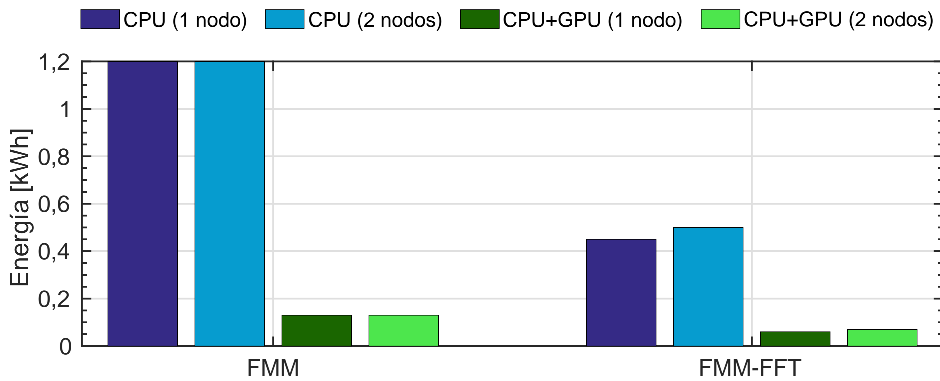


Figura 6.14: *Cluster* Mopar de TSC-UNIOVI. Comparativa del consumo de energía usando las implementaciones paralelas del FMM y del FMM-FFT para CPU y CPU + GPU. Esfera ⌀4 m ($N = 6001966$, $f = 11{,}5$ kHz).

de energía de las diferentes implementaciones del FMM y del FMM-FFT en el *cluster* Mopar. Nuevamente, se tiene que las implementaciones del FMM-FFT son más eficientes desde el punto de vista energético que sus equivalentes para el FMM.

Para finalizar, resulta interesante analizar la figura 6.14 desde una perspectiva diferente: considerando que se parte de una situación inicial en la que sólo se dispone de una implementación paralela del FMM para CPU. En dicho caso, puede resultar más conveniente decantarse por desarrollar una implementación heterogénea del FMM que llevar a cabo una implementación para CPU del FMM-FFT.

6.3.3. Problema de gran tamaño acústico

Para comparar las implementaciones del FMM y del FMM-FFT cuando se resuelven problemas con un gran tamaño acústico, en este punto se muestran los datos relativos a la resolución del problema de cincuenta millones de incógnitas utilizado previamente en los apartados 6.1.3 y 6.2.3.

Las implementaciones para CPU del FMM y del FMM-FFT se han ejecutado en el *cluster* CMI, mientras que las implementaciones heterogéneas para CPU + GPU del FMM y del FMM-FFT se han ejecutado en el *cluster* Mopar.

La tabla 6.23 resume los datos más relevantes de la resolución del problema con las implementaciones paralelas para CPU del FMM y del FMM-FFT en el *cluster* CMI. Las mejoras en los tiempos de ejecución al utilizar el FMM-FFT son notables: el tiempo por iteración se divide entre 3,7 y el tiempo total se reduce a la mitad. Sin embargo, en este caso, el consumo de memoria del FMM-FFT es claramente mayor, llegando a multiplicarse por 4,4 respecto del consumo del FMM.

Por su parte, en la tabla 6.24 se muestran los datos más importantes de la resolución del problema con cincuenta millones de incógnitas usando las implementaciones heterogéneas del FMM y del FMM-FFT en el *cluster* Mopar.

	FMM	FMM-FFT
Diámetro de la esfera	20 m (294,12 λ)	
Frecuencia	5 kHz	
Número de incógnitas	50003414	
Error residual objetivo	$\epsilon \leq 10^{-2}$	
Tamaño de los grupos	2,22 λ	1,4 λ
Número de iteraciones	9	
Tiempo por iteración	20 min 33 s	5 min 31 s
Tiempo total	3 h 59 min	1 h 54 min
Máximo uso de memoria por nodo	5,03 GB	20,2 GB
Consumo total de memoria	125,99 GB	560,3 GB

Tabla 6.23: Comparativa de la resolución de un problema acústico con cincuenta millones de incógnitas usando las implementaciones para memoria distribuida del FMM y del FMM-FFT en el *cluster* CMI (A.1).

Al igual que en el caso de la implementación para CPU, las mejoras en los tiempos de ejecución proporcionadas por el FMM-FFT para CPU + GPU son destacables: el tiempo por iteración se divide entre 2,6 y el tiempo total se reduce 2,2 veces. Por contra, el consumo de memoria del FMM-FFT prácticamente duplica el del FMM.

6.4. bMVP, mMVP, *SmallBlocks*, *BigBlocks*, y solución heterogénea

Los resultados que se muestran en esta sección se han obtenido usando el sistema Manycores del grupo de Recuperación de Información y Computación Paralela (IRPCG) de la Universidad de Oviedo, cuyas características

	FMM	FMM-FFT
Diámetro de la esfera	20 m (294,12 λ)	
Frecuencia	5 kHz	
Número de incógnitas	50003414	
Error residual objetivo	$\epsilon \leq 10^{-2}$	
Tamaño de los grupos	4,6 λ	2,58 λ
Número de iteraciones	9	
Tiempo por iteración	8 min	3 min 5 s
Tiempo total	1 h 38 min	44 min 8 s
Máximo uso de memoria por nodo	25,3 GB	50 GB
Consumo total de memoria	50,3 GB	99,8 GB

Tabla 6.24: Comparativa de la resolución de un problema acústico con cincuenta millones de incógnitas usando las implementaciones paralelas y heterogéneas del FMM y del FMM-FFT en el *cluster* Mopar (A.3).

se detallan en el anexo A.4, dentro del apéndice dedicado a la descripción del hardware y del software de los sistemas utilizados. También merece la pena señalar que todos los algoritmos analizados en este apartado utilizan aritmética de simple precisión, con reales de 32 bits y complejos de 64 bits.

El análisis del rendimiento de las diferentes técnicas desarrolladas para llevar a cabo el cálculo de la presión total en el problema de dispersión acústica se realiza recurriendo a un conjunto de problemas en los que se analiza la presión acústica generada por una aeronave, considerando diferentes configuraciones para las fuentes de ruido (motores de la aeronave).

Primero, se analiza un modelo de una aeronave a tamaño real (Airbus serie A3xx) con el objetivo de obtener la distribución de la presión sobre su superficie a una frecuencia de 1 kHz. Para ello, la aeronave se modela utilizando una malla con aproximadamente 6 elementos por longitud de

onda lineal, resultando en un total de 1009392 facetas triangulares. Los problemas correspondientes a cada configuración de las fuentes de ruido se resuelven usando la implementación del FMM mostrada en el capítulo 2.

Posteriormente, a partir de los resultados obtenidos con las simulaciones descritas en el párrafo anterior, se calcula la presión total en un plano situado bajo el avión (ver punto 4.3.1) usando las diferentes técnicas presentadas en el capítulo 4. Dicho plano tiene unas dimensiones de 90,015 m × 90,015 m y se utiliza una resolución de 4 elementos por longitud de onda lineal, lo que se traduce en 1123600 puntos de observación.

Por tanto, en los problemas que se resuelven en este apartado, se tienen $N = 1009392$ elementos fuente y $M = 1123600$ puntos de observación. El número de configuraciones para las fuentes de ruido, N_c, se varía entre 1 y 128.

$N = 1009392, \ M = 1123600$

N_c	bMVP [s]	mMVP [s]	*SmallBlocks* [s]	*BigBlocks* [s]
1	**722**	766	1636	1195
16	11554	3483	**1574**	1644
64	46216	10166	3070	**2469**
128	92431	20619	4119	**3641**

Tabla 6.25: Tiempo de ejecución para el cálculo de la presión total usando diferentes técnicas implementadas para CPU en un conjunto de problemas con $N = 1009392$, $M = 1123600$ y $N_c \in [1, 128]$.

En la tabla 6.25, se muestran los tiempos de ejecución de las diferentes técnicas desarrolladas para calcular la presión total (ver apartado 4.2), ejecutadas en procesadores convencionales. El mejor resultado para cada valor de N_c se marca en letra negrita. Cuando se resuelve un problema con un número elevado de configuraciones para las fuentes de ruido ($N_c \geq 16$), las técnicas basadas en la multiplicación matricial por bloques (*SmallBlocks* y *BigBlocks*) permiten obtener los menores tiempos de cálculo.

La tabla 6.26 presenta los tiempos de ejecución de las diferentes técnicas

$N = 1009392,\ M = 1123600$

N_c	bMVP [s]	mMVP [s]	*SmallBlocks* [s]	*BigBlocks* [s]
1	**185**	192	1178	781
16	2967	1663	1432	**1016**
64	11868	5199	2636	**1291**
128	23735	7013	5443	**1407**

Tabla 6.26: Tiempo de ejecución para el cálculo de la presión total usando diferentes técnicas implementadas para coprocesadores Xeon Phi en un conjunto de problemas con $N = 1009392$, $M = 1123600$ y $N_c \in [1, 128]$.

cuando se utiliza un coprocesador Intel Xeon Phi [91]. Nuevamente, para problemas con $N_c \geq 16$ las técnicas basadas en la multiplicación matricial por bloques permiten obtener mejores tiempos. En dichos casos, la técnica *BigBlocks* produce el menor tiempo de ejecución.

$N = 1009392,\ M = 1123600$

N_c	bMVP [s]	mMVP [s]	*BigBlocks* [s]
1	**137**	156	311
16	2184	695	**302**
64	8738	2758	**506**
128	17475	5520	**886**

Tabla 6.27: Tiempo de ejecución para el cálculo de la presión total usando diferentes técnicas implementadas para GPU en un conjunto de problemas con $N = 1009392$, $M = 1123600$ y $N_c \in [1, 128]$.

Por su parte, la tabla 6.27 muestra los tiempos de ejecución de las técnicas implementadas para GPU. En este caso, la técnica *SmallBlocks* se ha descartado para el caso de procesadores gráficos (ver justificación en el punto 4.2.4.1). Al igual que en los casos anteriores, cuando se tienen problemas con $N_c \geq 16$ la técnica *BigBlocks* logra tiempos de ejecución menores en comparación con las técnicas basadas en el producto matriz-

vector.

$N = 1009392,\ M = 1123600$

N_c	Solución heterogénea (bMVP) [s]	Solución heterogénea (mejor algoritmo) [s]
1	**88,0**	**88,0**
16	1408,3	**221,7**
64	5633,3	**352,9**
128	11266,6	**557,5**

Tabla 6.28: Tiempo de ejecución para el cálculo de la presión total usando diferentes técnicas implementadas para sistemas heterogéneos basados en GPU + Xeon Phi en un conjunto de problemas con $N = 1009392$, $M = 1123600$ y $N_c \in [1, 128]$.

Finalmente, en la tabla 6.28, se presentan los resultados obtenidos utilizando de forma conjunta una GPU y un coprocesador Xeon Phi. En la columna central, se muestran los resultados usando la implementación heterogénea de la técnica bMVP para GPU + Xeon Phi (ver punto 4.2.5). En la columna de la derecha, se muestran los resultados usando la implementación heterogénea que escoge la mejor técnica (menor tiempo de ejecución) para el valor de N_c analizado (en este caso, bMVP para $N_c = 1$ y *BigBlocks* para el resto de valores de N_c).

Asimismo, merece la pena destacar que, cuando se resuelve un determinado problema con un número reducido de configuraciones diferentes para las fuentes de ruido (valores de N_c muy próximos a 1), el problema a resolver ($P^s = K \cdot P$) se convierte prácticamente en un producto matriz-vector. En dichos casos, las técnicas basadas en el producto matriz-vector pueden llegar a ofrecer un rendimiento superior a las técnicas basadas en el producto matricial.

Como caso límite de lo comentado en el párrafo anterior, se puede tomar como ejemplo las medidas para $N_c = 1$ de las tablas 6.25, 6.26 y 6.27, donde las implementaciones bMVP y mMVP (basadas en el producto matriz-

vector) ofrecen un rendimiento muy superior a las técnicas *SmallBlocks* y *BigBlocks* (basadas en el producto matricial por bloques).

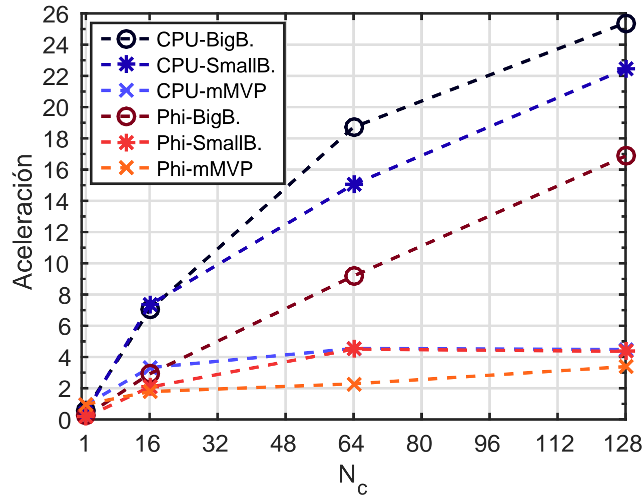
La figura 6.15 muestra la aceleración lograda con cada una de las soluciones desarrolladas en comparación con el tiempo de ejecución de la técnica bMVP (usada como referencia). Cada algoritmo se compara con la implementación del bMVP para la misma arquitectura siguiendo la expresión:

$$\mathrm{Acel}_x(N_c) \;=\; \frac{T_{\mathrm{bMVP}}(N_c)}{T_x(N_c)} \;=\; \frac{N_c \cdot T_{\mathrm{bMVP}}(1)}{T_x(N_c)}, \tag{6.4}$$

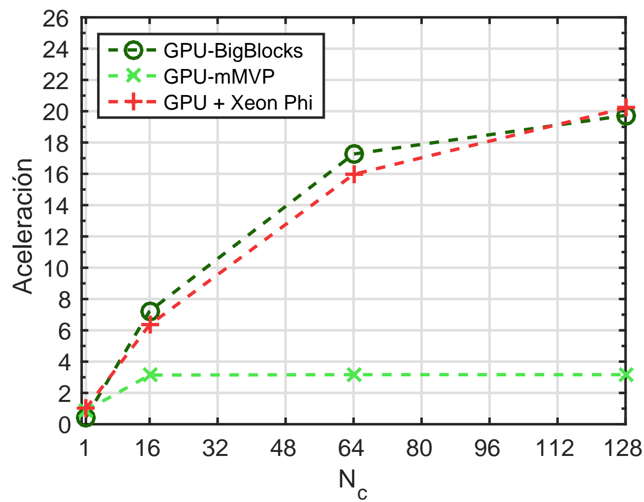
donde $T_{\mathrm{bMVP}}(N_c)$ es el tiempo de ejecución para N_c configuraciones de las fuentes de ruido usando la implementación bMVP (ver punto 4.2.2) y $T_x(N_c)$ es el tiempo de ejecución del mismo problema usando la técnica x, con $x \in \{$mMVP, *SmallBlocks*, *BigBlocks*$\}$.

En todos los casos, la mayor aceleración se obtiene al usar la técnica *BigBlocks*:

- En CPU, $\mathrm{Acel}_{BigBlocks}(128) = 25{,}4$
- Para Xeon Phi, $\mathrm{Acel}_{BigBlocks}(128) = 16{,}9$
- En GPU, $\mathrm{Acel}_{BigBlocks}(128) = 19{,}7$
- Con GPU + Xeon Phi, $\mathrm{Acel}_{BigBlocks}(128) = 20{,}2$



(a)



(b)

Figura 6.15: Sistema Manycores del IRPCG (A.4). Aceleración de las diferentes técnicas respecto de la implementación de referencia (bMVP). (a) Implementaciones para CPU y Xeon Phi. (b) Implementaciones para GPU y GPU + Xeon Phi.

6.5. SRM

En este apartado, se muestran los resultados computacionales más relevantes obtenidos con las herramientas basadas en el método de reconstrucción de fuentes (ver capítulo 5).

6.5.1. Problemas de caracterización de antenas

Los resultados que se muestran en este punto se han obtenido usando el sistema SimLin02 del grupo de investigación TSC-UNIOVI (ver A.5). Al igual que en el caso de las herramientas desarrolladas para acústica, la herramienta para caracterización de antenas analizada en este apartado utiliza aritmética de simple precisión (reales de 32 bits y complejos de 64 bits).

Para analizar el rendimiento computacional de la solución propuesta, se utilizan dos problemas diferentes (ver punto 5.3.1). En el primero de ellos, se calculan las corrientes equivalentes en una antena de una estación base, mientras que en el segundo se analiza una antena de tipo hélice. En ambos casos, el CGNR itera hasta que el error RMS entre el campo medido y el campo radiado por las corrientes equivalentes calculadas se encuentra por debajo del 5 %.

En la tabla 6.29, se muestran los parámetros más importantes de la ejecución del primer problema (caracterización de una antena de una estación base). Puede observarse cómo la implementación del MST para CPU + GPU desarrollada logra dividir entre 133 el tiempo de ejecución de la implementación del MST para CPU, lo que permite obtener una solución en tiempo cuasi-real (2 segundos), manteniendo un consumo de memoria muy bajo.

Como referencia, la resolución del mismo problema en CPU sin aplicar la técnica MST (precalculando y almacenando la matriz de impedancias) requiere 77 segundos y 5100 MB de memoria. Por tanto, la solución heterogénea del MST resulta 38,5 veces más rápida con unos requisitos de memoria tres órdenes de magnitud inferiores.

	MST (CPU)	MST (CPU + GPU)
Tamaño de la ABM (D_0)	1,6 m (9,6 λ)	
Frecuencia	1,8 GHz	
Número de incógnitas	7640	
Número de ecuaciones	43802	
Tiempo por iteración CGNR	26,6 s	0,2 s
Número de iteraciones	10	
Tiempo total	266 s	2 s
Convergencia (RMSE)	3,4 %	
Uso de memoria	5 MB	4 MB

Tabla 6.29: Comparativa del tiempo de ejecución y la memoria necesarios para la obtención de las corrientes equivalentes en una antena de una estación base usando las diferentes implementaciones del MST en el sistema SimLin02 (A.5).

Por su parte, la tabla 6.30 presenta los datos relativos a la ejecución del segundo problema (caracterización de una antena de tipo hélice). En esta ocasión, la implementación del MST para CPU + GPU permite dividir entre 76 el tiempo de ejecución que se obtiene con la implementación del MST para CPU.

Asimismo, merece la pena destacar que, en este caso, la resolución del mismo problema en CPU sin aplicar la técnica MST requeriría aproximadamente 55 GB de memoria, lo que hace inviable su ejecución en el sistema SimLin02.

6.5.2. Problema de imagen electromagnética

Los resultados presentados en este punto se han obtenido usando dos sistemas diferentes del grupo TSC-UNIOVI: el sistema Barracuda (ver A.6) y el sistema Roadrunner (ver A.7). En ambos sistemas, se compara la im-

	MST (CPU)	MST (CPU + GPU)
Tamaño de la ABM (D_0)	0,2 m (3 λ)	
Frecuencia	4,5 GHz	
Número de incógnitas	82560	
Número de ecuaciones	43802	
Tiempo por iteración CGNR	220 s	2,9 s
Número de iteraciones	9	
Tiempo total	1980 s	26 s
Convergencia (RMSE)	3,9 %	
Uso de memoria	10 MB	8 MB

Tabla 6.30: Comparativa del tiempo de ejecución y la memoria necesarios para la obtención de las corrientes equivalentes en una antena de tipo hélice usando las diferentes implementaciones del MST en el sistema SimLin02.

plementación heterogénea del SRM para *imaging* presentada en esta Tesis con una implementación para CPU utilizada como referencia. Cabe destacar que la implementación de referencia es una implementación paralela para sistemas de memoria compartida que combina el uso de C (en el SRM) y MATLAB (en el cálculo del campo interno).

Con el objetivo de ilustrar el rendimiento de la herramienta desarrollada, se recurre a la resolución de un problema de reconstrucción de perfiles en el que se tienen dos objetos metálicos diferentes: un cilindro y un prisma rectangular (ver punto 5.3.2).

La solución de dicho problema conlleva la resolución de doce problemas inversos (cálculo de las corrientes equivalentes), uno por cada una de las doce ondas planas incidentes que se utilizan para iluminar los objetos bajo medida. A su vez, cada uno de los problemas inversos, que se resuelven usando el SRM, está compuesto por 6936 incógnitas y 7200 ecuaciones. En este caso, el CGNR itera hasta que el error RMS entre el campo dispersado y el campo radiado por las corrientes equivalentes reconstruidas se encuen-

tra por debajo del 2 % (para este problema, entre 11 y 12 iteraciones). Posteriormente, a partir de dichas corrientes equivalentes reconstruidas, se calcula cada una de las tres componentes (x, y, z) del campo interno en un total de 19683 puntos.

Sistema Barracuda de TSC-UNIOVI (A.6)			
CPU	SRM	Tiempo por incidencia	11,4 s
		Tiempo total	137 s
	Campo interno	Tiempo por incidencia	17,8 s
		Tiempo total	214 s
CPU + 1 GPU	SRM	Tiempo por incidencia	0,318 s
		Tiempo total	3,82 s
	Campo interno	Tiempo por incidencia	0,027 s
		Tiempo total	0,32 s
CPU + 2 GPU	SRM	Tiempo por incidencia	0,17 s
		Tiempo total	2,04 s
	Campo interno	Tiempo por incidencia	0,014 s
		Tiempo total	0,162 s

Tabla 6.31: Comparativa del tiempo de ejecución de la solución de referencia (CPU) frente a la implementación heterogénea (CPU + GPU) para la obtención del campo interno en un problema de *imaging* usando el sistema Barracuda.

La tabla 6.31 muestra los datos relativos al tiempo de ejecución del problema analizado cuando se usa el sistema Barracuda (sistema con tarjetas gráficas de la serie Fermi). Como se puede observar, la implementación heterogénea para CPU + GPU permite obtener resultados en tiempo cuasireal: 4,14 segundos usando una GPU y 2,2 segundos usando las dos tarjetas gráficas disponibles.

Si se compara la solución heterogénea con la solución paralela para CPU, que requiere 351 segundos para resolver el mismo problema, puede comprobarse que la implementación para CPU + GPU permite dividir el

tiempo de ejecución entre 159.

Sistema Roadrunner de TSC-UNIOVI (A.7)			
CPU	SRM	Tiempo por incidencia Tiempo total	6,3 s 76 s
	Campo interno	Tiempo por incidencia Tiempo total	14,2 s 170 s
CPU + 1 GPU	SRM	Tiempo por incidencia Tiempo total	0,13 s 1,56 s
	Campo interno	Tiempo por incidencia Tiempo total	0,011 s 0,126 s
CPU + 2 GPU	SRM	Tiempo por incidencia Tiempo total	0,072 s 0,86 s
	Campo interno	Tiempo por incidencia Tiempo total	0,0055 s 0,065 s

Tabla 6.32: Comparativa del tiempo de ejecución de la solución de referencia (CPU) frente a la implementación heterogénea (CPU + GPU) para la obtención del campo interno en un problema de *imaging* usando el sistema Roadrunner.

Por su parte, en la tabla 6.32, se presentan los datos de tiempo de ejecución en el sistema Roadrunner (sistema con tarjetas gráficas de la serie Kepler). En este caso, la implementación heterogénea también proporciona resultados en tiempo cuasi-real: 1,69 segundos cuando se usa una GPU y 0,93 segundos si se usan dos.

Puesto que la implementación paralela para CPU requiere 246 segundos, cuando se usa el sistema Roadrunner, la implementación para CPU + GPU permite resolver este problema 265 veces más rápido.

Para finalizar, la figura 6.16 muestra la eficiencia paralela de la implementación heterogénea en los diferentes sistemas utilizados. En este caso,

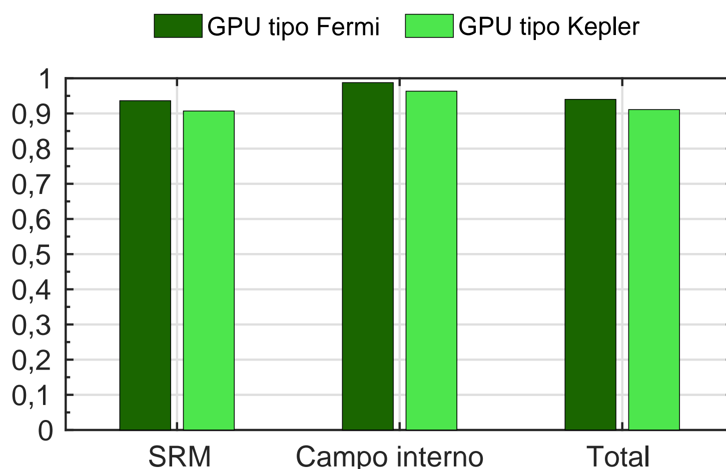


Figura 6.16: Comparativa de la eficiencia paralela para dos GPU en un problema de *imaging* teniendo en cuenta el tiempo de ejecución del SRM, el tiempo para calcular el campo interno y el tiempo total (suma de los anteriores), usando la implementación heterogénea para CPU + GPU. Sistema Barracuda con dos GPU de tipo Fermi. Sistema Roadrunner con dos GPU de tipo Kepler.

la eficiencia paralela, $E(g)$, se define de la siguiente manera:

$$E_{\mathrm{SRM}}(g) = \frac{T_{\mathrm{SRM}}(1)}{g \cdot T_{\mathrm{SRM}}(g)}, \tag{6.5a}$$

$$E_{\mathrm{Ei}}(g) = \frac{T_{\mathrm{Ei}}(1)}{g \cdot T_{\mathrm{Ei}}(g)}, \tag{6.5b}$$

$$E_{\mathrm{Total}}(g) = \frac{T_{\mathrm{SRM}}(1) + T_{\mathrm{Ei}}(1)}{g \cdot [T_{\mathrm{SRM}}(g) + T_{\mathrm{Ei}}(g)]}, \tag{6.5c}$$

donde g representa el número de GPU utilizadas, $T(1)$ es el tiempo de ejecución utilizado como referencia (resolución usando una única GPU), y $T(g)$ es el tiempo de ejecución cuando se usan de forma simultánea g procesadores gráficos. T_{SRM} es el tiempo de ejecución del SRM (obtención de las corrientes equivalentes) y T_{Ei} es el tiempo dedicado a obtener el campo interno (a partir de las corrientes equivalentes reconstruidas).

En este problema, el valor de $E_{\text{Total}}(2)$ es del 94 % cuando se usa el sistema Barracuda y del 91 % cuando se usa el sistema Roadrunner. De igual forma, merece la pena destacar los valores obtenidos para $E_{\text{Ei}}(2)$, que alcanza el 99 % y el 96 % para Barracuda y Roadrunner, respectivamente. Como se puede observar, la solución desarrollada presenta una buena escalabilidad, teniendo en cuenta que el problema que se resuelve en este apartado es un problema con un tamaño reducido.

Capítulo 7

Conclusiones y líneas futuras

Índice

7.1. Conclusiones 227
7.2. Líneas futuras 230

7.1. Conclusiones

El control del ruido generado por las aeronaves es un tema de especial importancia para los organismos y las empresas del sector aeronáutico, que buscan reducir la contaminación acústica producida por dichas aeronaves durante los despegues y los aterrizajes con el objetivo de minimizar las molestias ocasionadas a la población. Partiendo del trabajo desarrollado en la Tesis del Dr. Jesús López Fernández [13], se ha generado un conjunto de herramientas que permiten simular de forma precisa y eficiente el campo acústico dispersado por objetos de gran tamaño (en términos de longitud de onda) como pueden ser las aeronaves.

En el capítulo 2, se presenta una implementación paralela y heterogénea del FMM para sistemas de memoria distribuida con nodos del tipo CPU + GPU. Por su parte, en el capítulo 3, se describen las dos versiones del método FMM-FFT generadas: una para sistemas de memoria distribuida

convencionales (sólo CPU) y otra para sistemas de memoria distribuida heterogéneos (CPU + GPU). Finalmente, en el capítulo 4, se presenta una herramienta para calcular la presión total en diferentes planos situados en el espacio que rodea al objeto dispersor, y que consta de múltiples versiones: para sistemas de memoria compartida basados en procesadores convencionales, y para sistemas heterogéneos del tipo CPU + Xeon Phi, CPU + GPU y CPU + GPU + Xeon Phi.

Las implementaciones paralelas y heterogéneas del FMM y del FMM-FFT permiten acelerar los pasos computacionalmente más costosos, posibilitando además la ejecución simultánea de la traslación, que se ejecuta en las CPU, y del cálculo de las interacciones cercanas, que se lleva a cabo en las GPU. La herramienta para CPU + GPU basada en el FMM que se ha desarrollado permite dividir los tiempos de ejecución entre 21 y los requisitos de memoria entre 4, en comparación con la implementación paralela para CPU, cuando se resuelve un problema de tamaño moderado con 6 millones de incógnitas. Por su parte, la implementación heterogénea para CPU + GPU basada en el FMM-FFT permite dividir los tiempos de ejecución entre 16 y el consumo de memoria entre 9 en comparación con la implementación del mismo algoritmo para CPU, usando un problema idéntico al del caso anterior. Las mejoras obtenidas, tanto en tiempo de ejecución como en consumo de memoria, han permitido resolver problemas de gran tamaño acústico con 50 y 100 millones de incógnitas sin recurrir al uso de grandes sistemas, utilizando únicamente la potencia computacional de dos estaciones de trabajo con un total de cuatro GPU.

Por su parte, la herramienta presentada en el capítulo 4, aprovecha las ventajas del producto matricial por bloques para calcular de forma eficiente la presión total en un plano situado en el exterior del objeto dispersor, obteniendo de forma simultánea y en un único paso los resultados para diferentes configuraciones de las fuentes de ruido. Además, la herramienta también logra una aceleración en el tiempo de ejecución gracias al uso (independiente o combinado) de aceleradores hardware como las GPU o los coprocesadores de tipo Xeon Phi.

El problema inverso de dispersión electromagnética es de gran interés en diversas aplicaciones en las que se deben aplicar técnicas de evaluación

no invasiva, entre las que se encuentran la caracterización de antenas y la imagen electromagnética. Tomando como base el trabajo desarrollado por los doctores Yuri Álvarez López y Cebrián García González en sus respectivas Tesis [14, 15], se han desarrollado herramientas basadas en el método de reconstrucción de fuentes con el objetivo de posibilitar análisis en tiempo real o cuasi-real para los problemas más habituales (de tamaño moderado).

En el capítulo 5, se presenta una implementación paralela y heterogénea del SRM para sistemas de memoria compartida de tipo CPU + GPU con dos aplicaciones diferentes: caracterización de antenas y reconstrucción de perfiles. Para el primer caso, se ha desarrollado una herramienta en la que se aprovecha la potencia computacional de la GPU para acelerar el cálculo de los productos matriz-vector asociados a la resolución iterativa del SRM. La aceleración del tiempo de ejecución en casi dos órdenes de magnitud respecto de la implementación paralela para CPU permite obtener resultados en pocos segundos. Por su parte, la herramienta para reconstrucción de perfiles consiste en una implementación para múltiples procesadores gráficos en la que cada GPU se encarga de analizar parte de los ángulos de incidencia que se utilizan para iluminar el objeto bajo medida. En este caso, la reducción de los tiempos de ejecución alcanza (y supera en algún paso) los dos órdenes de magnitud, permitiendo obtener los resultados en menos de un segundo para un problema cuya resolución requiere cuatro minutos utilizando una CPU multinúcleo.

Las herramientas de simulación desarrolladas para el problema de dispersión acústica tienen una aplicabilidad directa en problemas de control de ruido, ya que facilitan la obtención de resultados de forma rápida y precisa. De esta forma, es posible reducir las medidas sobre prototipos para descubrir las modificaciones que deben realizarse en un modelo con el objetivo de alcanzar los requisitos de funcionamiento predefinidos.

Por su parte, las herramientas desarrolladas para el problema inverso de dispersión electromagnética, generadas dentro del proyecto “iScat” (Ref.: TEC2011-24492), logran satisfacer los requisitos iniciales de proporcionar resultados en tiempo real o cuasi-real. En el caso de los problemas de reconstrucción de perfiles, para ciertas aplicaciones como las relaciona-

das con la seguridad (detección de armas ocultas, inspección de paquetes, etc.), disponer de resultados en tiempo real utilizando un hardware sencillo y asequible (como una estación de trabajo con GPU) resulta especialmente interesante.

7.2. Líneas futuras

Como continuación del trabajo llevado a cabo en esta Tesis, se presentan diversas posibilidades. Las más directas, por su inmediatez, son las relacionadas con las modificaciones sobre los algoritmos desarrollados. Otras implican el uso de algoritmos diferentes o cambios en la formulación.

En el caso del problema acústico, tomando como punto de partida las implementaciones para CPU y CPU + GPU del FMM y del FMM-FFT, sería sencillo generar nuevas versiones de dichos algoritmos orientadas a su ejecución en coprocesadores Xeon Phi. Asimismo, como ya se planteaba en el punto 4.2.1, se podrían modificar las técnicas bMVP, mMVP, *SmallBlocks* y *BigBlocks* para su uso en sistemas de memoria distribuida, permitiendo de esa manera resolver problemas de mayor tamaño. Por último, sería interesante explorar el uso de otros algoritmos en arquitecturas heterogéneas, como las diferentes variantes del FMM multinivel.

Por su parte, en el ámbito del problema inverso para electromagnetismo, ya se ha comenzado a desarrollar una nueva variante del SRM-MST para múltiples GPU presentado en [70] que busca explotar las ventajas del producto matriz-matriz para calcular en un único paso las corrientes equivalentes obtenidas a partir de todas las ondas incidentes utilizadas con una misma frecuencia.

Finalmente, otra posibilidad pasaría por la aplicación de las técnicas desarrolladas en otros problemas. Por ejemplo, en el caso del FMM y del FMM-FFT, las herramientas para acústica podrían adaptarse para su uso en otros ámbitos realizando modificaciones en la formulación pero conservando las estrategias de paralelización presentadas en esta Tesis.

Apéndice A

Sistemas utilizados y parámetros de ejecución

Índice

A.1. UniOvi - CMI **232**

A.2. Microway - Tesla MD SimCluster **234**

A.3. TSC-UNIOVI - Mopar **236**

A.4. IRPCG - Manycores **239**

A.5. TSC-UNIOVI - SimLin02 . . . . **243**

A.6. TSC-UNIOVI - Barracuda **244**

A.7. TSC-UNIOVI - Roadrunner **246**

En este apéndice, se describen los diferentes sistemas (estaciones de trabajo, servidores y *clusters*) utilizados para obtener los resultados mostrados en el capítulo 6.

Se detallan las características del hardware y del software disponible en cada sistema, y se muestran los parámetros de ejecución más importantes para las soluciones analizadas en cada sistema.

A.1. UniOvi - CMI

El *cluster* CMI de la Universidad de Oviedo es un sistema de memoria distribuida con nodos de tipo SMP en formato *blade*. En la figura A.1, puede verse, de forma simplificada, la composición del *cluster* CMI.

Figura A.1: Composición del *cluster* CMI de la Universidad de Oviedo.[1]

[1] Imágenes extraídas de `https://www.hpe.com/us/en/integrated-systems/bladesystem.html` y `http://www.mellanox.com/related-docs/voltaire_ib_switch_systems/Grid-Switch-ISR-9024-WEB-091409.pdf`

En cada nodo		
CPU	Modelo	AMD Opteron 2356
	Procesadores	2
	Núcleos / procesador	4
	Frecuencia	2,3 GHz
Memoria principal	Tamaño	32 GB
	Tipo	DDR2-667
	Interfaz	64-bit
Red	Tipo	InfiniBand 4× DDR
	Capacidad	16–20 Gb/s (Troncal con ratio 2:1)
Total (28 nodos)		
CPU	Procesadores	56
	Núcleos	224
Memoria principal	Tamaño	896 GB

Tabla A.1: Características hardware del *cluster* CMI de la Universidad de Oviedo.

La tabla A.1 resume de forma precisa las características del hardware que compone el *cluster* CMI de la Universidad de Oviedo.

Sistema operativo	Scientific Linux 5.9
Compilador para C	Intel icc 12.0
Librería para FFT	FFTW 3.3.4
Librería para paso de mensajes	HP-MPI 2.3

Tabla A.2: Características software del *cluster* CMI de la Universidad de Oviedo.

Por su parte, en la tabla A.2 se detalla el software del *cluster* CMI que se ha utilizado para obtener los resultados presentados en esta Tesis.

En este sistema, se han ejecutado las implementaciones paralelas del FMM y del FMM-FFT para CPU (ver puntos 6.1, 6.2 y 6.3). En las medidas llevadas a cabo en CMI, se lanzan tantos procesos MPI [98] como procesadores y tantos hilos OpenMP [96] por proceso como núcleos por procesador, con el objetivo de lograr una buena correspondencia entre el software y el hardware subyacente.

A.2. Microway - Tesla MD SimCluster

El Tesla MD SimCluster de Microway es un sistema de memoria distribuida con nodos de tipo SMP que cuenta, además, con tarjetas gráficas para cálculo de propósito general. En la figura A.2, se muestra una imagen de este sistema dentro de un pequeño *rack*.

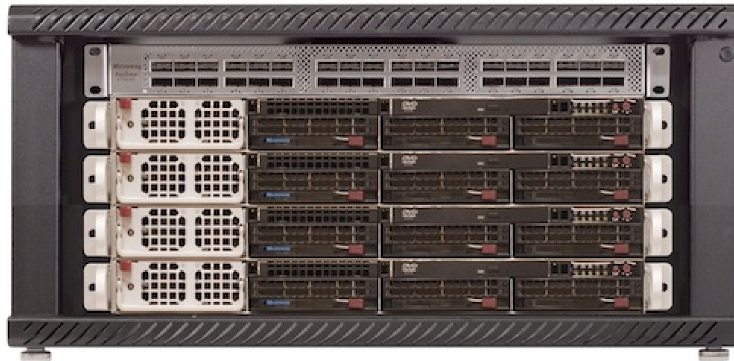


Figura A.2: Tesla MD SimCluster de Microway.[2]

La tabla A.3 muestra las características del hardware que compone el Tesla MD SimCluster, con especial detalle en el caso de los componentes que forman cada nodo.

En la tabla A.4, se enumera el software que se ha utilizado para compilar y ejecutar el software analizado en este sistema.

[2] Imagen extraída de `https://www.microway.com/gpu-test-drive/`

En cada nodo		
CPU	Modelo	Intel Xeon X5675
	Procesadores	2
	Núcleos / procesador	6
	Hilos / núcleo	2
	Frecuencia base	3,06 GHz
Memoria principal	Tamaño	48 GB
	Tipo	DDR3-1333
	Interfaz	64-bit
GPU	Microarquitectura	Fermi (2.0)
	Modelo	NVIDIA Tesla M2090
	Tarjetas	2
	Núcleos / tarjeta	512
	Frecuencia	1,3 GHz
Memoria GPU	Tamaño	6 GB / tarjeta
	Tipo	GDDR5 a 1,85 GHz
	Interfaz	384-bit
Red	Tipo	InfiniBand 4× QDR
	Capacidad	32–40 Gb/s
Total (4 nodos)		
CPU	Procesadores	8
	Núcleos	48
	Hilos	96
Memoria principal	Tamaño	192 GB
GPU	Tarjetas	8
	Núcleos	4096
Memoria GPU	Tamaño	48 GB

Tabla A.3: Características hardware del *cluster* Tesla MD SimCluster de Microway.

Sistema operativo	CentOS 6.0
Compilador para C	GNU GCC 4.4.6
Entorno de desarrollo CUDA	NVIDIA CUDA 4.0
Librería para paso de mensajes	MVAPICH2 1.6

Tabla A.4: Características software del *cluster* Tesla MD SimCluster de Microway.

En este sistema, propiedad de la empresa estadounidense Microway, sólo se han ejecutado las implementaciones paralelas del FMM para CPU y para CPU + GPU (ver punto 6.1), ya que su disponibilidad en el tiempo estaba limitada.

En las medidas del FMM para CPU realizadas en el Tesla MD SimCluster, se lanzan tantos procesos MPI como procesadores y doce hilos OpenMP por proceso.

Por su parte, en las medidas del FMM para CPU + GPU realizadas en este sistema, el número de procesos MPI utilizados se ajusta al número de procesadores gráficos (que, en este caso, coincide con el número de procesadores convencionales).

A.3. TSC-UNIOVI - Mopar

El *cluster* Mopar del grupo de investigación TSC-UNIOVI es un pequeño sistema de memoria distribuida, con sólo dos nodos, que dispone de tarjetas gráficas con capacidades de cálculo de propósito general. En la figura A.3, se muestran los componentes de una de las estaciones de trabajo que forman este sistema.

En la tabla A.5, se detallan las características del hardware presente en el *cluster* Mopar, incidiendo especialmente en las diferentes GPU instaladas en este sistema.

	En cada nodo	
CPU	Modelo	Intel Core i7-3820
	Procesadores	1
	Núcleos / procesador	4
	Hilos / núcleo	2
	Frecuencia base	3,6 GHz
Memoria principal	Tamaño	64 GB
	Tipo	DDR3-1600
	Interfaz	64-bit
GPU	Microarquitectura	Kepler (3.0)
	Modelo	NVIDIA GeForce GTX 680 + GTX 770
	Tarjetas	1 + 1
	Núcleos / tarjeta	1536
	Frecuencia	1,14–1,19 GHz
Memoria GPU	Tamaño	2 GB / tarjeta
	Tipo	GDDR5 a 3,0–3,5 GHz
	Interfaz	256-bit
Red	Tipo	Ethernet 1000BASE-T
	Capacidad	1 Gb/s
	Total (2 nodos)	
CPU	Procesadores	2
	Núcleos	8
	Hilos	16
Memoria principal	Tamaño	128 GB
GPU	Tarjetas	4
	Núcleos	6144
Memoria GPU	Tamaño	8 GB

Tabla A.5: Características hardware del *cluster* Mopar del grupo TSC-UNIOVI.

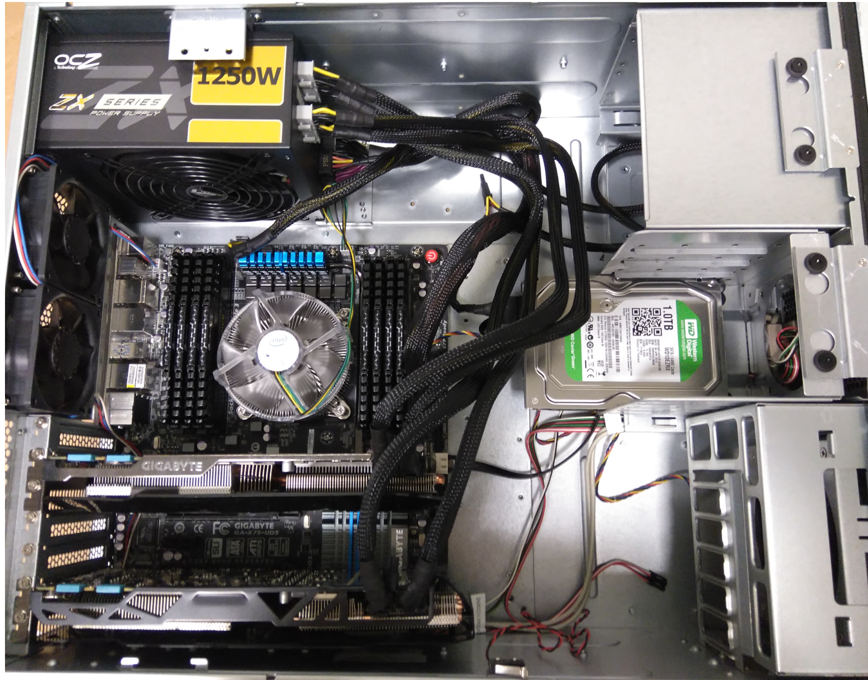


Figura A.3: Detalle de los componentes de la estación de trabajo Charger. Sistema heterogéneo con una CPU multi-núcleo y dos GPU perteneciente al *cluster* Mopar de TSC-UNIOVI.

Finalmente, la tabla A.6 muestra el software del *cluster* Mopar utilizado para compilar y ejecutar las diferentes soluciones analizadas en este sistema.

El *cluster* Mopar se ha utilizado para analizar el rendimiento de las implementaciones del FMM y del FMM-FFT para CPU y para CPU + GPU (ver puntos 6.1, 6.2 y 6.3).

En el caso de las medidas del FMM y del FMM-FFT para CPU realizadas en el *cluster* Mopar, se lanzan tantos procesos MPI como procesadores (uno por nodo) y ocho hilos OpenMP por proceso.

Para las medidas del FMM y del FMM-FFT para CPU + GPU, el número de procesos MPI utilizados se fija al número de procesadores gráficos y el número de hilos OpenMP se ajusta evitando saturar las CPU.

Sistema operativo	CentOS 7.2
Compilador para C	Intel icc 12.1.5
Entorno de desarrollo CUDA	NVIDIA CUDA 7.5
Librería para FFT	FFTW 3.3.3
Librería para paso de mensajes	MPICH2 1.5

Tabla A.6: Características software del *cluster* Mopar del grupo TSC-UNIOVI.

A.4. IRPCG - Manycores

El servidor Manycores del grupo de Recuperación de Información y Computación Paralela (IRPCG) es un sistema de memoria compartida de tipo SMP. Este sistema dispone, además, de una tarjeta gráfica para cálculo de propósito general y de un coprocesador Xeon Phi.

La tabla A.7 muestra el hardware que compone el sistema Manycores, detallando las características de los diferentes aceleradores de los que dispone. Por su parte, la tabla A.8 muestra el software que se ha utilizado para compilar y ejecutar la herramienta analizada en este sistema.

El sistema Manycores del IRPCG se ha utilizado para analizar el rendimiento de las diferentes técnicas, presentadas en el capítulo 4, que se han desarrollado para llevar a cabo el cálculo de la presión total en el problema de dispersión acústica (ver punto 6.4).

La tabla A.9 muestra los parámetros de ejecución para procesadores gráficos de tipo Kepler de los *kernels* correspondientes a las diferentes técnicas implementadas: bMVP, mMVP y *BigBlocks*. Nuevamente, merece la pena señalar que la técnica *SmallBlocks* no se ha implementado en este caso, puesto que no resulta adecuada para GPU (ver punto 4.2.4.1). Además, en el caso de la técnica mMVP, se ha comprobado que la ocupación de la GPU [148] decae en aquellos casos en los que se trabaja de forma simultánea con más de cuatro configuraciones para las fuentes de ruido, como consecuencia del aumento en los requisitos de memoria compartida.

Nodo único		
CPU	Modelo	Intel Xeon E5-2650
	Procesadores	2
	Núcleos / procesador	8
	Hilos / núcleo	2
	Frecuencia base	2,0 GHz
Memoria principal	Tamaño	64 GB
	Tipo	DDR3-1333
	Interfaz	64-bit
GPU	Microarquitectura	Kepler (3.5)
	Modelo	NVIDIA Tesla K20c
	Tarjetas	1
	Núcleos / tarjeta	2496
	Frecuencia	706 MHz
Memoria GPU	Tamaño	5 GB
	Tipo	GDDR5 a 2,6 GHz
	Interfaz	320-bit
MIC	Microarquitectura	Knights Corner
	Modelo	Intel Xeon Phi 5110P
	Tarjetas	1
	Núcleos / tarjeta	60
	Hilos / núcleo	4
	Frecuencia	1,053 GHz
Memoria MIC	Tamaño	8 GB
	Tipo	GDDR5 a 2,5 GHz
	Interfaz	512-bit

Tabla A.7: Características hardware del sistema Manycores del grupo de Recuperación de Información y Computación Paralela (IRPCG).

Sistema operativo	Scientific Linux 6.x
Compilador para C	Intel icc 14
Entorno de desarrollo CUDA	NVIDIA CUDA 6.0

Tabla A.8: Características software del sistema Manycores del grupo de Recuperación de Información y Computación Paralela (IRPCG).

	bMVP	mMVP	*BigBlocks*
Registros por hilo	28	36	30
Uso de memoria compartida por cada bloque [kB]	1	$\text{mín}\{4, N_c\}$	0
Bloques × hilos / bloque	8192×128	8192×128	8192×256

Tabla A.9: Principales parámetros de ejecución en GPU de tipo Kepler para los *kernels* correspondientes a las diferentes técnicas desarrolladas.

Por tanto, cuando se tiene que $N_c > 4$, el problema original se subdivide en varios *kernels* que se encargan de tratar, como máximo, cuatro configuraciones de cada vez. Esta particularidad queda reflejada en la entrada de la tabla A.9 correspondiente al uso de memoria compartida para el caso de mMVP.

	SmallBlocks	*BigBlocks*
bs (CPU)	$\text{máx}\{16, N_c\}$	10000
bs (Xeon Phi)	$N_c \leq 32: \ 32$ $N_c > 32: \ 64$	10000

Tabla A.10: Tamaño de bloque (*bs*) en los algoritmos basados en multiplicación matricial por bloques para CPU y para coprocesadores Xeon Phi.

Por su parte, en la tabla A.10, se muestra el tamaño de bloque utilizado (notado como *bs*) en las técnicas basadas en multiplicación matricial por bloques para CPU y Xeon Phi. Cuando se utiliza la técnica *SmallBlocks* y

se tiene un tamaño de bloque tal que $bs > N_c$, los bloques se rellenan con ceros de forma adecuada.

En el caso de las medidas en las que se usan únicamente las CPU, se utiliza un único proceso y 16 hilos OpenMP (el *Hyper-Threading* [149] se encuentra desactivado). Para las medidas en las que se utiliza la GPU y/o el coprocesador Xeon Phi, las CPU sólo se encargan de gestionar dichos aceleradores.

Para finalizar con la descripción del servidor Manycores, en la figura A.4 se muestran los componentes principales de este sistema.

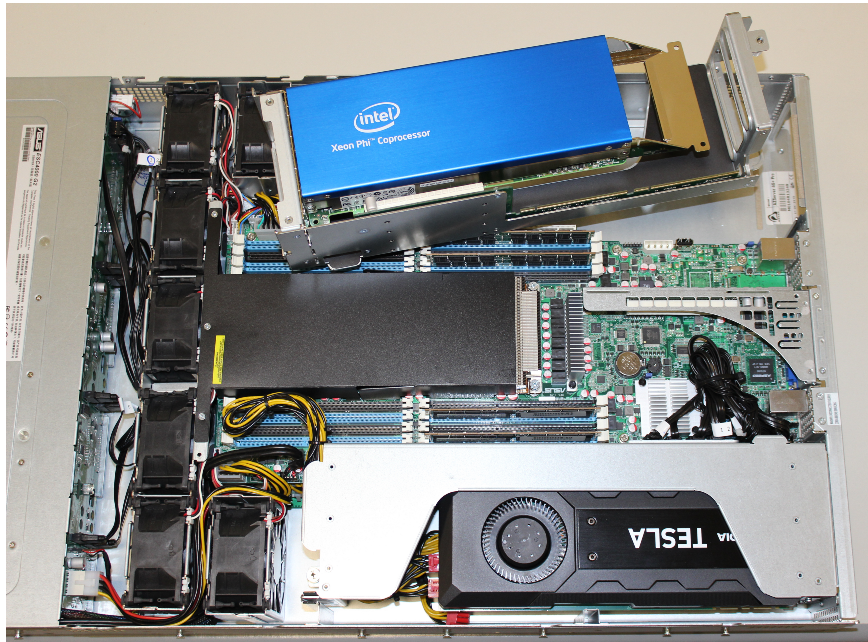


Figura A.4: Detalle de los componentes del sistema Manycores del grupo de Recuperación de Información y Computación Paralela (IRPCG). Sistema heterogéneo con dos CPU multi-núcleo, una GPU de la serie Kepler y un coprocesador Xeon Phi.

A.5. TSC-UNIOVI - SimLin02

El servidor SimLin02 del grupo TSC-UNIOVI es un sistema de memoria compartida de tipo SMP que también cuenta con una tarjeta gráfica para cálculo de propósito general.

Nodo único		
CPU	Modelo	AMD Opteron 265
	Procesadores	2
	Núcleos / procesador	2
	Frecuencia	1,8 GHz
Memoria principal	Tamaño	16 GB
	Tipo	DDR-400
	Interfaz	64-bit
GPU	Microarquitectura	Fermi (2.1)
	Modelo	NVIDIA GeForce GTX 460
	Tarjetas	1
	Núcleos / tarjeta	336
	Frecuencia	715 MHz
Memoria GPU	Tamaño	1 GB
	Tipo	GDDR5 a 1,8 GHz
	Interfaz	256-bit

Tabla A.11: Características hardware del sistema SimLin02 del grupo TSC-UNIOVI.

En la tabla A.11, se presentan las características del hardware del servidor SimLin02. Por su parte, en la tabla A.12 se muestra el software utilizado para compilar y ejecutar las herramientas analizadas en este sistema.

El sistema SimLin02 se ha utilizado para analizar el rendimiento de las diferentes implementaciones de la herramienta para la caracterización de antenas basada en el SRM (ver punto 6.5.1).

En las medidas en las que se usan únicamente las CPU, se utiliza un

Sistema operativo	CentOS 5.5
Compilador para C	Intel icc 11.1
Entorno de desarrollo CUDA	NVIDIA CUDA 4.1

Tabla A.12: Características software del sistema SimLin02 del grupo TSC-UNIOVI.

único proceso y 4 hilos OpenMP. Por su parte, en las medidas en las que se utiliza la implementación para CPU + GPU, se hace un uso combinado de las CPU (usando 4 hilos) y de la GPU de la que dispone el sistema.

A.6. TSC-UNIOVI - Barracuda

La estación de trabajo Barracuda del grupo de investigación TSC-UNIOVI es un sistema de memoria compartida que dispone, además, de dos tarjetas gráficas de la serie Fermi.

La tabla A.13 resume las características del hardware del que dispone la estación de trabajo Barracuda. Por su parte, en la tabla A.14 se muestra el software utilizado para compilar y ejecutar las herramientas que se han analizado en este sistema.

Esta estación de trabajo se ha utilizado en el análisis del rendimiento computacional de las diferentes implementaciones de la herramienta para la reconstrucción de perfiles basada en el SRM (ver punto 6.5.2).

En las medidas en las que se usan únicamente las CPU, se utiliza un único proceso y 4 hilos OpenMP (tantos como núcleos disponibles).

Por su parte, en las medidas en las que se utiliza la implementación para CPU + GPU, se lanzan tantos hilos OpenMP como GPU a utilizar (hasta un máximo de dos). Dichos hilos se encargan de las gestiones relacionadas con la ejecución de los diferentes *kernels*.

Finalmente, en la figura A.5, pueden verse los componentes que integran esta estación de trabajo.

Nodo único		
CPU	Modelo	Intel Core i5-2500T
	Procesadores	1
	Núcleos / procesador	4
	Frecuencia	2,3 GHz
Memoria principal	Tamaño	16 GB
	Tipo	DDR3-1333
	Interfaz	64-bit
GPU	Microarquitectura	Fermi (2.1)
	Modelo	NVIDIA GeForce GTX 460 + GTX 560
	Tarjetas	1 + 1
	Núcleos / tarjeta	336
	Frecuencia	715–810 MHz
Memoria GPU	Tamaño	1 GB / tarjeta
	Tipo	GDDR5 a 1,8–2,0 GHz
	Interfaz	256-bit

Tabla A.13: Características hardware del sistema Barracuda del grupo TSC-UNIOVI.

Sistema operativo	CentOS 6.x
Compilador para C	Intel icc 11.1
Entorno de desarrollo CUDA	NVIDIA CUDA 4.2
Entorno de desarrollo MATLAB	MathWorks MATLAB 2012a

Tabla A.14: Características software del sistema Barracuda del grupo TSC-UNIOVI.

Figura A.5: Detalle de los componentes de la estación de trabajo Barracuda. Sistema heterogéneo con una CPU multi-núcleo y dos GPU de la serie Fermi.

A.7. TSC-UNIOVI - Roadrunner

La estación de trabajo Roadrunner del grupo TSC-UNIOVI es un sistema de memoria compartida que cuenta, asimismo, con dos tarjetas gráficas de la serie Kepler.

En la tabla A.15, se muestra el hardware que compone la estación de trabajo Roadrunner. Por su parte, la tabla A.16 detalla el software utilizado para compilar y ejecutar las herramientas cuyo rendimiento se ha analizado en este sistema.

Al igual que el sistema Barracuda, esta estación de trabajo se ha utilizado en el análisis del rendimiento de la herramienta para la reconstrucción de perfiles basada en el SRM (ver punto 6.5.2).

Nodo único		
CPU	Modelo	Intel Core i7-3820
	Procesadores	1
	Núcleos / procesador	4
	Hilos / núcleo	2
	Frecuencia base	3,6 GHz
Memoria principal	Tamaño	64 GB
	Tipo	DDR3-1600
	Interfaz	64-bit
GPU	Microarquitectura	Kepler (3.0)
	Modelo	NVIDIA GeForce GTX 680
	Tarjetas	2
	Núcleos / tarjeta	1536
	Frecuencia	1,14 GHz
Memoria GPU	Tamaño	2 GB / tarjeta
	Tipo	GDDR5 a 3,0 GHz
	Interfaz	256-bit

Tabla A.15: Características hardware del sistema Roadrunner del grupo TSC-UNIOVI.

Sistema operativo	CentOS 6.x
Compilador para C	Intel icc 11.1
Entorno de desarrollo CUDA	NVIDIA CUDA 4.2
Entorno de desarrollo MATLAB	MathWorks MATLAB 2012a

Tabla A.16: Características software del sistema Roadrunner del grupo TSC-UNIOVI.

En las medidas en las que se usan únicamente las CPU, se utiliza un único proceso y 8 hilos OpenMP ya que, en este caso, se hace uso del *Hyper-Threading* [149].

En el caso de las medidas en las que se utiliza la implementación para CPU + GPU, al igual que en la estación de trabajo Barracuda, se lanzan tantos hilos OpenMP como GPU a utilizar, cuya misión es encargase de todas las tareas relacionadas con la ejecución de los diferentes *kernels*.

Bibliografía

[1] Advisory Council for Aviation Research and innovation in Europe, 2008 addendum to the strategic research agenda (2008).
URL http://ec.europa.eu/research/transport/publications/items/advisory_council_for_aeronautics_research_in_europe_en.htm (Citado en la página 2)

[2] Advisory Council for Aviation Research and innovation in Europe, Flightpath 2050 - Europe's vision for aviation (2011). doi:10.2777/50266.
URL http://www.acare4europe.com/documents/latest-acare-documents/acare-flightpath-2050 (Citado en la página 2)

[3] Clean Sky, European research programme aimed at reducing CO_2, gas emissions and noise levels produced by aircraft (2008).
URL http://www.cleansky.eu/ (Citado en la página 2)

[4] Boeing, Boeing and General Electric validate 747-8 quiet technologies (2006).
URL https://web.archive.org/web/20110823055345/http://www.boeing.com/news/releases/2006/q3/060920a_pr.html (Citado en la página 2)

[5] Japan Aerospace Exploration Agency (JAXA), Development of technology for investigating noise sources on passenger aircraft (2011).
URL http://www.aero.jaxa.jp/eng/publication/magazine/apgnews/2011_no20/apn2011no20_03.html (Citado en la página 2)

[6] Y. A. López, F. Las-Heras, M. R. Pino, Reconstruction of equivalent currents distribution over arbitrary three-dimensional surfaces based on integral equation algorithms, IEEE Transactions on Antennas and Propagation 55 (12) (2007) 3460–3468. doi:10.1109/TAP.2007.910316. (Citado en las páginas 3, 12, 153, 154, 155, 156 y 157)

[7] Y. Álvarez, F. Las-Heras, M. R. Pino, On the comparison between the spherical wave expansion and the sources reconstruction method, IEEE Transactions on Antennas and Propagation 56 (10) (2008) 3337–3341. doi:10.1109/TAP.2008.929519. (Citado en las páginas 3, 9 y 153)

[8] M. Çayören, I. Akduman, A. Yapar, L. Crocco, A new algorithm for the shape reconstruction of perfectly conducting objects, Inverse Problems 23 (3) (2007) 1087–1100. doi:10.1088/0266-5611/23/3/015. (Citado en las páginas 3 y 10)

[9] Y. Álvarez-López, A. Domínguez-Casas, C. García-González, F. Las-Heras, Geometry reconstruction of metallic bodies using the sources reconstruction method, IEEE Antennas and Wireless Propagation Letters 9 (2010) 1197–1200. doi:10.1109/LAWP.2010.2098385. (Citado en las páginas 3, 11 y 12)

[10] K. B. Cooper, R. J. Dengler, N. Llombart, B. Thomas, G. Chattopadhyay, P. H. Siegel, THz imaging radar for standoff personnel screening, IEEE Transactions on Terahertz Science and Technology 1 (1) (2011) 169–182. doi:10.1109/TTHZ.2011.2159556. (Citado en las páginas 3 y 159)

[11] Y. Álvarez, B. González-Valdés, J. A. Martínez, F. Las-Heras, C. M. Rappaport, 3D whole body imaging for detecting explosive-related threats, IEEE Transactions on Antennas and Propagation 60 (9) (2012) 4453–4458. doi:10.1109/TAP.2012.2207068. (Citado en las páginas 3 y 159)

[12] J. A. Martínez-Lorenzo, F. Quivira, C. M. Rappaport, SAR imaging of suicide bombers wearing concealed explosive threats, Progress In Electromagnetics Research 125 (2012) 255–272. doi:10.2528/PIER11120518. (Citado en las páginas 3 y 159)

[13] J. A. López Fernández, Development of Efficient Techniques for the Solution of Acoustic and Electromagnetic Scattering Problems, Universidad de Vigo (Tesis Doctoral), Vigo, España, 2009. (Citado en las páginas 3, 5, 32, 33, 35, 38, 39, 40, 42, 43, 45, 52, 54, 63, 76, 85, 86, 87, 179 y 227)

[14] Y. Álvarez López, Método de reconstrucción de fuentes para el diagnóstico y caracterización de sistemas radiantes, Universidad de Oviedo (Tesis Doctoral), Gijón, España, 2009. (Citado en las páginas 4, 153, 154, 155 y 229)

[15] C. García González, Métodos electromagnéticos de onda completa aplicados al problema inverso de radiación y dispersión, Universidad de Oviedo (Tesis Doctoral), Gijón, España, 2016. (Citado en las páginas 4 y 229)

[16] L. H. Chen, D. G. Schweikert, Sound radiation from an arbitrary body, The Journal of the Acoustical Society of America 35 (10) (1963) 1626–1632. doi:10.1121/1.1918770. (Citado en las páginas 5 y 33)

[17] R. F. Harrington, Matrix methods for field problems, Proceedings of the IEEE 55 (2) (1967) 136–149. doi:10.1109/PROC.1967.5433. (Citado en las páginas 5 y 11)

[18] R. F. Harrington, Origin and development of the method of moments for field computation, IEEE Antennas and Propagation Magazine 32 (3) (1990) 31–35. doi:10.1109/74.80522. (Citado en las páginas 5 y 11)

[19] R. F. Harrington, Field Computation by Moment Methods, Wiley-IEEE Press, Nueva York, Estados Unidos de América, 1993. doi:10.1109/9780470544631. (Citado en la página 5)

[20] H. A. van der Vorst, Iterative Krylov Methods for Large Linear Systems, Cambridge University Press, Cambridge, Reino Unido, 2003. (Citado en la página 6)

[21] N. Balin, G. Sylvand, J. Robert, Fast methods applied to BEM solvers for acoustic propagation problems, en: 22nd AIAA/CEAS Aeroacous-

tics Conference, American Institute of Aeronautics and Astronautics, 2016. doi:10.2514/6.2016-2712. (Citado en las páginas 6, 7, 8 y 13)

[22] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. V. der Vorst, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, segunda edición, Society for Industrial and Applied Mathematics, Filadelfia, Estados Unidos de América, 1994. (Citado en las páginas 6, 7, 34 y 83)

[23] D. Holliday, L. L. DeRaad, G. J. St.-Cyr, Forward-backward: a new method for computing low-grazing angle scattering, IEEE Transactions on Antennas and Propagation 44 (5) (1996) 722–729. doi:10.1109/8.496263. (Citado en la página 6)

[24] J. C. West, J. M. Sturm, On iterative approaches for electromagnetic rough-surface scattering problems, IEEE Transactions on Antennas and Propagation 47 (8) (1999) 1281–1288. doi:10.1109/8.791944. (Citado en la página 6)

[25] J. A. López, M. R. Pino, F. Obelleiro, J. L. Rodríguez, Application of the spectral acceleration forward-backward method to coverage analysis over terrain profiles, Journal of Electromagnetic Waves and Applications 15 (8) (2001) 1049–1074. doi:10.1163/156939301X00427. (Citado en la página 6)

[26] H. A. van der Vorst, Krylov subspace iteration, Computing in Science and Engineering 2 (1) (2000) 32–37. doi:10.1109/5992.814655. (Citado en la página 7)

[27] M. R. Hestenes, E. Stiefel, Methods of conjugate gradients for solving linear systems, Journal of Research of the National Bureau of Standards 49 (6) (1952) 409–436. (Citado en las páginas 7 y 11)

[28] H. A. van der Vorst, Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems, SIAM Journal of Scientific and Statistical Computing 13 (2) (1992) 631–644. doi:10.1137/0913035. (Citado en la página 7)

[29] Y. Saad, M. H. Schultz, GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM Journal of Scientific and Statistical Computing 7 (3) (1986) 856–869. doi:10.1137/0907058. (Citado en las páginas 7, 34 y 83)

[30] S. Marburg, S. Schneider, Performance of iterative solvers for acoustic problems. Part I. Solvers and effect of diagonal preconditioning, Engineering Analysis with Boundary Elements 27 (7) (2003) 727–750. doi:10.1016/S0955-7997(03)00025-0. (Citado en las páginas 7, 34 y 83)

[31] W. Hackbusch, A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: Introduction to $\mathcal{H}$-matrices, Computing 62 (2) (1999) 89–108. doi:10.1007/s006070050015. (Citado en la página 7)

[32] W. Hackbusch, B. N. Khoromskij, A sparse $\mathcal{H}$-matrix arithmetic. Part II: Application to multi-dimensional problems, Computing 64 (1) (2000) 21–47. doi:10.1007/PL00021408. (Citado en la página 7)

[33] M. Bebendorf, Approximation of boundary element matrices, Numerische Mathematik 86 (4) (2000) 565–589. doi:10.1007/PL00005410. (Citado en la página 7)

[34] M. Bebendorf, S. Rjasanow, Adaptive low-rank approximation of collocation matrices, Computing 70 (1) (2003) 1–24. doi:10.1007/s00607-002-1469-6. (Citado en la página 7)

[35] K. Zhao, M. N. Vouvakis, J.-F. Lee, The adaptive cross approximation algorithm for accelerated method of moments computations of EMC problems, IEEE Transactions on Electromagnetic Compatibility 47 (4) (2005) 763–773. doi:10.1109/TEMC.2005.857898. (Citado en las páginas 7 y 12)

[36] E. Michielssen, A. Boag, A multilevel matrix decomposition algorithm for analyzing scattering from large structures, IEEE Transactions on Antennas and Propagation 44 (8) (1996) 1086–1093. doi:10.1109/8.511816. (Citado en la página 7)

[37] J. M. Rius, J. Parrón, E. Úbeda, J. R. Mosig, Multilevel matrix decomposition algorithm for analysis of electrically large electromagnetic problems in 3-D, Microwave and Optical Technology Letters 22 (3) (1999) 177–182. doi:10.1002/(SICI)1098-2760(19990805)22:3<177::AID-MOP8>3.0.CO;2-2. (Citado en la página 7)

[38] L. Greengard, V. Rokhlin, A fast algorithm for particle simulations, Journal of Computational Physics 73 (2) (1987) 325–348. doi:10.1016/0021-9991(87)90140-9. (Citado en las páginas 7 y 8)

[39] V. Rokhlin, Rapid solution of integral equations of scattering theory in two dimensions, Journal of Computational Physics 86 (2) (1990) 424–439. doi:10.1016/0021-9991(90)90107-C. (Citado en la página 7)

[40] V. Rokhlin, Diagonal forms of translation operators for the helmholtz equation in three dimensions, Applied and Computational Harmonic Analysis 1 (1) (1993) 82–93. doi:10.1006/acha.1993.1006. (Citado en las páginas 7, 8, 34, 35, 38, 40, 83 y 84)

[41] J. Dongarra, F. Sullivan, Guest Editors' Introduction: The Top 10 Algorithms, Computing in Science & Engineering 2 (1) (2000) 22–23. doi:10.1109/MCISE.2000.814652. (Citado en la página 8)

[42] J. Board, K. Schulten, The Fast Multipole Algorithm, Computing in Science & Engineering 2 (1) (2000) 76–79. doi:10.1109/5992.814662. (Citado en la página 8)

[43] L. Greengard, V. Rokhlin, On the efficient implementation of the fast multiple algorithm, Research Report-602, Department of Computer Science, Yale University, 1988. (Citado en la página 8)

[44] R. L. Wagner, J. Song, W. C. Chew, Monte carlo simulation of electromagnetic scattering from two-dimensional random rough surfaces, IEEE Transactions on Antennas and Propagation 45 (2) (1997) 235–245. doi:10.1109/8.560342. (Citado en las páginas 8, 83 y 86)

[45] C. Waltz, K. Sertel, M. A. Carr, B. C. Usner, J. L. Volakis, Massively parallel fast multipole method solutions of large electromagnetic scattering problems, IEEE Transactions on Antennas and Propagation

55 (6) (2007) 1810–1816. doi:10.1109/TAP.2007.898511. (Citado en las páginas 8 y 83)

[46] J. Song, W. Chew, Multilevel fast-multipole algorithm for solving combined field integral equations of electromagnetic scattering, Microwave and Optical Technology Letters 10 (1) (1995) 14–19. doi:10.1002/mop.4650100107. (Citado en la página 8)

[47] J. A. López, F. Obelleiro, H. J. Rice, Application of an accelerated generalized minimum residual iteration method in 2d and 3d acoustic scattering problems, en: 9th AIAA/CEAS Aeroacoustics Conference and Exhibit, American Institute of Aeronautics and Astronautics, 2003. doi:10.2514/6.2003-3118. (Citado en la página 8)

[48] S. Schneider, Application of fast methods for acoustic scattering and radiation problems, Journal of Computational Acoustics 11 (3) (2003) 387–401. doi:10.1142/S0218396X03002012. (Citado en la página 8)

[49] H. Wu, Y. Liu, W. Jiang, A fast multipole boundary element method for 3D multi-domain acoustic scattering problems based on the Burton-Miller formulation, Engineering Analysis with Boundary Elements 36 (5) (2012) 779–788. doi:10.1016/j.enganabound.2011.11.018. (Citado en la página 8)

[50] W. R. Wolf, S. K. Lele, Fast Multipole Burton-Miller Boundary Element Method for Two and Three-Dimensional Acoustic Scattering, Journal of Aerospace Technology and Management 4 (2) (2012) 145–161. doi:10.5028/jatm.2012.04021812. (Citado en la página 8)

[51] A. Yaghjian, An overview of near-field antenna measurements, IEEE Transactions on Antennas and Propagation 34 (1) (1986) 30–45. doi:10.1109/TAP.1986.1143727. (Citado en las páginas 9 y 153)

[52] J. E. Hansen, Spherical Near-field Antenna Measurements, Institution of Engineering and Technology, Londres, Reino Unido, 1988. doi:10.1049/PBEW026E. (Citado en las páginas 9, 153 y 165)

[53] S. Ponnapalli, Near-field to far-field transformation utilizing the conjugate gradient method, Progress In Electromagnetics Research 5 (1991) 391–422. (Citado en las páginas 9 y 153)

[54] P. Petre, T. K. Sarkar, Planar near-field to far-field transformation using an equivalent magnetic current approach, IEEE Transactions on Antennas and Propagation 40 (11) (1992) 1348–1356. `doi:10.1109/8.202712`. (Citado en las páginas 9 y 153)

[55] F. Las-Heras, M. R. Pino, S. Loredo, Y. Álvarez, T. K. Sarkar, Evaluating near-field radiation patterns of commercial antennas, IEEE Transactions on Antennas and Propagation 54 (8) (2006) 2198–2207. `doi:10.1109/TAP.2006.879190`. (Citado en la página 9)

[56] K. Persson, M. Gustafsson, Reconstruction of equivalent currents using a near-field data transformation - with radome application, Progress In Electromagnetics Research 54 (2005) 179–198. `doi:10.2528/PIER04111602`. (Citado en la página 9)

[57] J. L. A. Quijano, G. Vecchi, Improved-accuracy source reconstruction on arbitrary 3-D surfaces, IEEE Antennas and Wireless Propagation Letters 8 (2009) 1046–1049. `doi:10.1109/LAWP.2009.2031988`. (Citado en las páginas 9 y 153)

[58] A. J. Devaney, G. C. Sherman, Nonuniqueness in inverse source and scattering problems, IEEE Transactions on Antennas and Propagation 30 (5) (1982) 1034–1037. `doi:10.1109/TAP.1982.1142902`. (Citado en la página 10)

[59] S. Caorsi, G. L. Gragnani, M. Pastorino, Two-dimensional microwave imaging by a numerical inverse scattering solution, IEEE Transactions on Microwave Theory and Techniques 38 (8) (1990) 981–989. `doi:10.1109/22.57321`. (Citado en la página 10)

[60] Y. Qin, I. Ciric, Inverse scattering solution with current modeling and Tikhonov regularization, en: Proceedings of IEEE Antennas and Propagation Society International Symposium, IEEE, 1993, págs. 492–495 vol.1. `doi:10.1109/APS.1993.385300`. (Citado en la página 10)

[61] C.-Y. Lin, Y.-W. Kiang, Inverse scattering for conductors by the equivalent source method, IEEE Transactions on Antennas and Propagation 44 (3) (1996) 310–316. `doi:10.1109/8.486298`. (Citado en la página 10)

[62] A. Broquetas, J. Palau, L. Jofre, A. Cardama, Spherical wave near-field imaging and radar cross-section measurement, IEEE Transactions on Antennas and Propagation 46 (5) (1998) 730–735. doi:10.1109/8.668918. (Citado en la página 10)

[63] B. González-Valdés, Y. Álvarez-López, J. A. Martínez-Lorenzo, F. Las-Heras, C. M. Rappaport, SAR processing for profile reconstruction and characterization of dielectric objects on the human body surface, Progress In Electromagnetics Research 138 (2013) 269–282. doi:10.2528/PIER13020607. (Citado en la página 10)

[64] P. M. van den Berg, R. E. Kleinman, A contrast source inversion method, Inverse Problems 13 (6) (1997) 1607–1620. (Citado en la página 10)

[65] M. Li, X. Y. Wang, A. Abubakar, Accelerating nonlinear inversion algorithms on GPU platform for electromagnetic data, en: 2016 International Symposium on Antennas and Propagation (ISAP), IEEE, 2016, págs. 574–575. (Citado en las páginas 10 y 13)

[66] S. Caorsi, A. Massa, M. Pastorino, M. Donelli, Improved microwave imaging procedure for nondestructive evaluations of two-dimensional structures, IEEE Transactions on Antennas and Propagation 52 (6) (2004) 1386–1397. doi:10.1109/TAP.2004.830254. (Citado en las páginas 10 y 11)

[67] A. Massa, D. Franceschini, G. Franceschini, M. Pastorino, M. Raffetto, M. Donelli, Parallel GA-based approach for microwave imaging applications, IEEE Transactions on Antennas and Propagation 53 (10) (2005) 3118–3127. doi:10.1109/TAP.2005.856311. (Citado en las páginas 10 y 11)

[68] M. Benedetti, M. Donelli, A. Massa, Multicrack detection in two-dimensional structures by means of GA-based strategies, IEEE Transactions on Antennas and Propagation 55 (1) (2007) 205–215. doi:10.1109/TAP.2006.888399. (Citado en las páginas 10 y 11)

[69] M. Donelli, A. Massa, Computational approach based on a particle swarm optimizer for microwave imaging of two-dimensional dielectric

scatterers, IEEE Transactions on Microwave Theory and Techniques 53 (5) (2005) 1761–1776. doi:10.1109/TMTT.2005.847068. (Citado en las páginas 10 y 11)

[70] M. López-Portugués, Y. Álvarez-López, J. A. López-Fernández, C. García-González, R. G. Ayestarán, F. Las-Heras, A multi-GPU sources reconstruction method for imaging applications, Progress In Electromagnetics Research 136 (2013) 703–724. doi:10.2528/PIER12122104. (Citado en las páginas 11, 25 y 230)

[71] A. Taaghol, T. K. Sarkar, Near-field to near/far-field transformation for arbitrary near-field geometry, utilizing an equivalent magnetic current, IEEE Transactions on Electromagnetic Compatibility 38 (3) (1996) 536–542. doi:10.1109/15.536088. (Citado en la página 11)

[72] T. K. Sarkar, A. Taaghol, Near-field to near/far-field transformation for arbitrary near-field geometry utilizing an equivalent electric current and MoM, IEEE Transactions on Antennas and Propagation 47 (3) (1999) 566–573. doi:10.1109/8.768793. (Citado en la página 11)

[73] H.-C. Wang, K. Hwang, Multicoloring of grid-structured PDE solvers on shared-memory multiprocessors, IEEE Transactions on Parallel and Distributed Systems 6 (11) (1995) 1195–1205. doi:10.1109/71.476191. (Citado en las páginas 11 y 156)

[74] T. F. Eibert, C. H. Schmidt, Multilevel fast multipole accelerated inverse equivalent current method employing Rao-Wilton-Glisson discretization of electric and magnetic surface currents, IEEE Transactions on Antennas and Propagation 57 (4) (2009) 1178–1185. doi:10.1109/TAP.2009.2015828. (Citado en las páginas 11 y 12)

[75] Y. A. López, F. Las-Heras, M. R. Pino, Application of the adaptive cross approximation algorithm to the sources reconstruction method, en: 2009 3rd European Conference on Antennas and Propagation, IEEE, 2009, págs. 761–765. (Citado en la página 12)

[76] J. M. Tamayo, A. Heldring, J. M. Rius, Application of multilevel adaptive cross approximation (MLACA) to electromagnetic scatte-

ring and radiation problems, en: 2009 International Conference on Electromagnetics in Advanced Applications, IEEE, 2009, págs. 178–181. doi:10.1109/ICEAA.2009.5297536. (Citado en la página 12)

[77] Y. Álvarez, F. Las-Heras, M. R. Pino, J. A. López, Acceleration of the sources reconstruction method via the fast multipole method, en: 2008 IEEE Antennas and Propagation Society International Symposium, IEEE, 2008, págs. 1–4. doi:10.1109/APS.2008.4619598. (Citado en la página 12)

[78] Y. Álvarez, J. A. Martínez, F. Las-Heras, C. M. Rappaport, An inverse fast multipole method for geometry reconstruction using scattered field information, IEEE Transactions on Antennas and Propagation 60 (7) (2012) 3351–3360. doi:10.1109/TAP.2012.2196950. (Citado en la página 12)

[79] I. Foster, Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering, primera edición, Addison Wesley, Boston, Estados Unidos de América, 1995. (Citado en las páginas 12, 17, 19, 31, 46, 82 y 144)

[80] S. Mittal, J. S. Vetter, A survey of CPU-GPU heterogeneous computing techniques, ACM Computing Surveys 47 (4) (2015) 69:1–69:35. doi:10.1145/2788396. (Citado en la página 12)

[81] J. M. Taboada, L. Landesa, J. M. Bértolo, F. Obelleiro, J. L. Rodríguez, J. C. Mouriño, A. Gómez, High scalability multipole method for the analysis of hundreds of millions of unknowns, en: 2009 3rd European Conference on Antennas and Propagation, IEEE, 2009, págs. 2753–2756. (Citado en la página 12)

[82] J. M. Taboada, M. G. Araújo, F. O. Basteiro, J. L. Rodríguez, L. Landesa, MLFMA-FFT parallel algorithm for the solution of extremely large problems in electromagnetics, Proceedings of the IEEE 101 (2) (2013) 350–363. doi:10.1109/JPROC.2012.2194269. (Citado en la página 13)

[83] M. Hidayetoğlu, L. Gürel, Full-wave and approximate solutions of large electromagnetic scattering problems, en: 2015 IEEE Interna-

tional Symposium on Antennas and Propagation USNC/URSI National Radio Science Meeting, IEEE, 2015, págs. 566–567. doi:10.1109/APS.2015.7304669. (Citado en la página 13)

[84] M. Cwikla, J. Aronsson, V. Okhmatovski, Low-frequency MLFMA on graphics processors, IEEE Antennas and Wireless Propagation Letters 9 (2010) 8–11. doi:10.1109/LAWP.2010.2040571. (Citado en la página 13)

[85] Q. M. Nguyen, V. Dang, O. Kilic, E. El-Araby, Parallelizing Fast Multipole Method for large-scale electromagnetic problems using GPU clusters, IEEE Antennas and Wireless Propagation Letters 12 (2013) 868–871. doi:10.1109/LAWP.2013.2271743. (Citado en la página 13)

[86] V. Dang, Q. Nguyen, O. Kilic, Fast Multipole Method for large-scale electromagnetic scattering problems on GPU cluster and FPGA-accelerated platforms, Applied Computational Electromagnetics Society Journal 28 (12) (2013) 1187–1198. (Citado en la página 13)

[87] V. Dang, Q. M. Nguyen, O. Kilic, GPU cluster implementation of FMM-FFT for large-scale electromagnetic problems, IEEE Antennas and Wireless Propagation Letters 13 (2014) 1259–1262. doi:10.1109/LAWP.2014.2332972. (Citado en la página 13)

[88] L. E. Tirado, G. Ghazi, J. Á. Martínez-Lorenzo, C. M. Rappaport, Y. Álvarez, F. Las-Heras, A GPU implementation of the inverse fast multipole method for multi-bistatic imaging applications, en: 2014 IEEE Antennas and Propagation Society International Symposium (APSURSI), IEEE, 2014, págs. 874–875. doi:10.1109/APS.2014.6904765. (Citado en la página 13)

[89] I. Lashuk, A. Chandramowlishwaran, H. Langston, T. A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, G. Biros, A massively parallel adaptive fast-multipole method on heterogeneous architectures, en: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, IEEE, 2009, págs. 1–12. doi:10.1145/1654059.1654118. (Citado en la página 13)

[90] R. Yokota, L. Barba, Hierarchical n-body simulations with autotuning for heterogeneous systems, Computing in Science Engineering 14 (3) (2012) 30–39. doi:10.1109/MCSE.2012.1. (Citado en la página 13)

[91] Intel Corporation, Intel Xeon Phi coprocessor architecture overview (2013).
URL https://software.intel.com/sites/default/files/Intel%C2%AE_Xeon_Phi%E2%84%A2_Coprocessor_Architecture_Overview.pdf (Citado en las páginas 14, 22 y 215)

[92] M. Merta, J. Zapletal, J. Jaros, Many core acceleration of the boundary element method, en: High Performance Computing in Science and Engineering: Second International Conference, HPCSE 2015, Springer International Publishing, 2016, págs. 116–125. doi:10.1007/978-3-319-40361-8_8. (Citado en la página 14)

[93] M. J. Flynn, Some computer organizations and their effectiveness, IEEE Transactions on Computers C-21 (9) (1972) 948–960. doi:10.1109/TC.1972.5009071. (Citado en la página 14)

[94] P. Gepner, M. F. Kowalik, Multi-core processors: New way to achieve high system performance, en: Proceedings of the 5-th International Symposium on Parallel Computing in Electrical Engineering (PARELEC 2006), IEEE, 2006, págs. 9–13. doi:10.1109/PARELEC.2006.54. (Citado en la página 15)

[95] B. W. Kernighan, D. M. Ritchie, The C Programming Language, primera edición, Prentice Hall, Nueva Jersey, Estados Unidos de América, 1978. (Citado en la página 16)

[96] The OpenMP ARB, The OpenMP API specification for parallel programming (2015).
URL http://www.openmp.org/ (Citado en las páginas 16, 51, 92, 162 y 234)

[97] Message Passing Interface Forum, MPI: A message-passing interface standard, version 1.3 (2008).

URL `http://www.mpi-forum.org/docs/docs.html` (Citado en las páginas 16 y 19)

[98] Message Passing Interface Forum, MPI: A message-passing interface standard, version 2.2 (2009).
URL `http://www.mpi-forum.org/docs/docs.html` (Citado en las páginas 16, 19, 49, 50 y 234)

[99] Message Passing Interface Forum, MPI: A message-passing interface standard, version 3.1 (2015).
URL `http://www.mpi-forum.org/docs/docs.html` (Citado en las páginas 16 y 19)

[100] R. Rabenseifner, G. Hager, G. Jost, Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes, en: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, IEEE, 2009, págs. 427–436. `doi:10.1109/PDP.2009.43`. (Citado en la página 17)

[101] GPGPU.org, General-Purpose computation on Graphics Processing Units (2015).
URL `http://gpgpu.org` (Citado en la página 19)

[102] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, J. C. Phillips, GPU computing, Proceedings of the IEEE 96 (5) (2008) 879–899. `doi:10.1109/JPROC.2008.917757`. (Citado en la página 19)

[103] TOP500.org, About the Green500 (2016).
URL `https://www.top500.org/green500/about/` (Citado en las páginas 19 y 186)

[104] S. Huang, S. Xiao, W. Feng, On the energy efficiency of graphics processing units for scientific computing, en: 2009 IEEE International Symposium on Parallel Distributed Processing, IEEE, 2009, págs. 1–8. `doi:10.1109/IPDPS.2009.5160980`. (Citado en las páginas 19 y 186)

[105] T. R. W. Scogland, H. Lin, W. Feng, A first look at integrated GPUs for green high-performance computing, Computer Science - Research and Development 25 (3) (2010) 125–134. `doi:10.1007/s00450-010-0128-y`. (Citado en las páginas 19 y 186)

[106] M. López-Portugués, J. A. López-Fernández, A. Rodríguez-Campa, J. Ranilla, A GPGPU solution of the FMM near interactions for acoustic scattering problems, Journal of Supercomputing 58 (3) (2011) 283–291. `doi:10.1007/s11227-011-0584-6`. (Citado en la página 19)

[107] TOP500.org, Accelerator/CP family / NVIDIA Fermi (2016).
URL `https://www.top500.org/statistics/details/accelfam/1` (Citado en la página 19)

[108] TOP500.org, Accelerator/CP family / NVIDIA Kepler (2016).
URL `https://www.top500.org/statistics/details/accelfam/2` (Citado en la página 19)

[109] TOP500.org, Accelerator/CP family / NVIDIA Pascal (2016).
URL `https://www.top500.org/statistics/details/accelfam/10` (Citado en la página 19)

[110] TOP500.org, Green500 list - November 2016 (2016).
URL `https://www.top500.org/green500/lists/2016/11/` (Citado en las páginas 19 y 22)

[111] NVIDIA Corporation, CUDA (2016).
URL `https://developer.nvidia.com/cuda-zone` (Citado en las páginas 20, 51, 92 y 163)

[112] NVIDIA Corporation, CUDA C programming guide (2016).
URL `http://docs.nvidia.com/cuda/cuda-c-programming-guide/` (Citado en las páginas 20, 22, 46, 51, 63, 92, 130, 136 y 158)

[113] TOP500.org, Accelerator/CP family / Intel Xeon Phi (2016).
URL `https://www.top500.org/statistics/details/accelfam/7` (Citado en la página 22)

[114] J. Jeffers, J. Reinders, Intel Xeon Phi Coprocessor High Performance Programming, Morgan Kaufmann Publishers Inc., San Francisco, Estados Unidos de América, 2013. (Citado en las páginas 23, 24 y 122)

[115] M. López-Portugués, J. A. López-Fernández, J. Ranilla, R. G. Ayestarán, F. Las-Heras, Parallelization of the FMM on distributed-memory GPGPU, Journal of Supercomputing 64 (1) (2013) 17–27. doi:10.1007/s11227-012-0786-6. (Citado en la página 24)

[116] J. A. López-Fernández, M. López-Portugués, J. Ranilla, Improving the FMM performance using optimal group size on heterogeneous system architectures, Journal of Supercomputing 73 (1) (2017) 291–301. doi:10.1007/s11227-016-1860-2. (Citado en las páginas 24, 93, 180, 196, 198 y 204)

[117] M. López-Portugués, J. A. López-Fernández, N. Díaz-Gracia, R. G. Ayestarán, J. Ranilla, Aircraft noise scattering prediction using different accelerator architectures, Journal of Supercomputing 70 (2) (2014) 612–622. doi:10.1007/s11227-014-1107-z. (Citado en la página 24)

[118] M. López-Portugués, J. A. López-Fernández, J. Ranilla, R. G. Ayestarán, F. Las-Heras, Using heterogeneous computing for scattering prediction in scenarios with several source configurations, Journal of Supercomputing 73 (1) (2017) 57–74. doi:10.1007/s11227-015-1618-2. (Citado en las páginas 24 y 122)

[119] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, en: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, ACM, 1967, págs. 483–485. doi:10.1145/1465482.1465560. (Citado en la página 25)

[120] J. L. Gustafson, Reevaluating Amdahl's law, Communications of the ACM 31 (5) (1988) 532–533. doi:10.1145/42411.42415. (Citado en la página 25)

[121] J. A. López-Fernández, M. López-Portugués, Y. Álvarez-López, C. García-González, D. Martínez, F. Las-Heras, Fast antenna characterization using the sources reconstruction method on graphics processors, Progress In Electromagnetics Research 126 (2012) 185–201. doi:10.2528/PIER11121408. (Citado en las páginas 25 y 153)

[122] T. W. Wu, Boundary Element Acoustics: Fundamentals and Computer Codes, WIT Press, Southampton, Reino Unido, 2000. (Citado en las páginas 32, 117, 118 y 119)

[123] A. J. Burton, G. F. Miller, The application of integral equation methods to the numerical solution of some exterior boundary-value problems, Proceedings of the Royal Society of London 323 (1553) (1971) 201–210. `doi:10.1098/rspa.1971.0097`. (Citado en la página 33)

[124] W. L. Meyer, W. A. Bell, B. T. Zinn, M. P. Stallybrass, Boundary integral solutions of three dimensional acoustic radiation problems, Journal of Sound and Vibration 59 (2) (1978) 245–262. `doi:10.1016/0022-460X(78)90504-7`. (Citado en la página 33)

[125] R. Coifman, V. Rokhlin, S. Wandzura, The fast multipole method for the wave equation: a pedestrian prescription, IEEE Antennas and Propagation Magazine 35 (3) (1993) 7–12. `doi:10.1109/74.250128`. (Citado en las páginas 34 y 39)

[126] C. C. Paige, M. A. Saunders, Solution of sparse indefinite systems of linear equations, SIAM Journal on Numerical Analysis 12 (4) (1975) 617–629. `doi:10.1137/0712047`. (Citado en la página 34)

[127] M. Abramowitz, I. A. Stegun, Handbook of Mathematical Functions: with Formulas, Graphs, and Mathematical Tables, décima edición, Dover Publications, Mineola, Estados Unidos de América, 1972. (Citado en las páginas 35, 41, 83 y 85)

[128] T. F. Eibert, A diagonalized multilevel fast multipole method with spherical harmonics expansion of the $\boldsymbol{\kappa}$-space integrals, IEEE Transactions on Antennas and Propagation 53 (2) (2005) 814–817. `doi:10.1109/TAP.2004.841310`. (Citado en la página 36)

[129] N. A. Gumerov, R. Duraiswami, E. A. Borovikov, Data structures, optimal choice of parameters, and complexity results for generalized multilevel fast multipole methods in d dimensions (2003).
URL `http://drum.lib.umd.edu/handle/1903/1270` (Citado en las páginas 38 y 45)

[130] M. Nilsson, Iterative Solution of Maxwell's Equations in Frequency Domain, Uppsala Universitet (Tesis Doctoral), Uppsala, Suecia, 2002. (Citado en la página 39)

[131] G. Sylvand, La Méthode Multipôle Rapide en Electromagnétisme: Performances, Parallélisation, Applications, École Nationale des Ponts et Chaussées (Tesis Doctoral), París, Francia, 2002. (Citado en la página 39)

[132] P. J. Denning, The locality principle, Communications of the ACM 48 (7) (2005) 19–24. `doi:10.1145/1070838.1070856`. (Citado en las páginas 46 y 132)

[133] R. Rabenseifner, Hybrid parallel programming on HPC platforms, en: Proceedings of the Fifth European Workshop on OpenMP (EWOMP '03), The Community of OpenMP Users, Researchers, Tool Developers and Providers (cOMPunity), 2003, págs. 185–194. (Citado en la página 49)

[134] A. Rane, D. Stanzione, Experiences in tuning performance of hybrid MPI/OpenMP applications on quad-core systems, en: Proceedings of the 10th LCI International Conference on High-Performance Clustered Computing, Linux Clusters Institute (LCI), 2009, págs. 1–11. (Citado en la página 50)

[135] M. Harris, Optimizing parallel reduction in CUDA (2007). URL `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf` (Citado en la página 63)

[136] Message Passing Interface Forum, All-to-all scatter/gather (2009). URL `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node101.htm` (Citado en la página 66)

[137] H. Batard, Aircraft noise reduction: AIRBUS industrial needs in terms of new materials for nacelle liners (2003). URL `https://web.archive.org/web/20120723021702/http://www.onera.fr/congres/jso2003mat-bruit/pdfs/AIRBUS-HBatard.pdf` (Citado en las páginas 74 y 145)

[138] L. Leylekian, M. Lebrun, P. Lempereur, An overview of aircraft noise reduction technologies, AerospaceLab 7 (2014) 1–15. doi:10.12762/2014.AL07-01. (Citado en las páginas 74 y 145)

[139] M. Frigo, S. G. Johnson, The design and implementation of FFTW3, Proceedings of the IEEE 93 (2) (2005) 216–231, special issue on "Program Generation, Optimization, and Platform Adaptation". doi:10.1109/JPROC.2004.840301. (Citado en la página 106)

[140] Intel Corporation, Intel Math Kernel Library documentation (2014). URL http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation (Citado en las páginas 123 y 135)

[141] NVIDIA Corporation, cuBLAS library (2014).
URL http://docs.nvidia.com/cuda/cublas/ (Citado en las páginas 123, 136 y 141)

[142] Netlib Repository at UTK and ORNL, BLAS (Basic Linear Algebra Subprograms) (2015).
URL http://www.netlib.org/blas/ (Citado en la página 133)

[143] C. A. Balanis, Advanced Engineering Electromagnetics, John Wiley & Sons, Nueva York, Estados Unidos de América, 1989. (Citado en la página 153)

[144] IEEE standard for definitions of terms for antennas, IEEE Std 145-2013 (Revision of IEEE Std 145-1993) (2014) 1–50. doi:10.1109/IEEESTD.2014.6758443. (Citado en la página 165)

[145] J. A. López-Fernández, M. L. Portugués, J. M. Taboada, H. J. Rice, F. Obelleiro, HP-FASS: a hybrid parallel fast acoustic scattering solver, International Journal of Computer Mathematics 88 (9) (2011) 1960–1968. doi:10.1080/00207160.2010.521239. (Citado en la página 179)

[146] Hewlett Packard Enterprise, HPE power advisor 8.1.2 (2017).
URL https://paonline56.itcs.hpe.com/ (Citado en la página 186)

[147] RS Amidata S.A., Medidor de energía Brennenstuhl serie PM231, monofásico, precisión: $\pm 1\,\%$ (2017).

URL http://es.rs-online.com/web/p/medidores-digitales-de-potencia/7975456/ (Citado en las páginas 187 y 201)

[148] NVIDIA Corporation, CUDA C best practices guide (2016).
URL http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/ (Citado en la página 239)

[149] Intel Corporation, Intel Hyper-Threading Technology (Intel HT Technology) (2015).
URL http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html (Citado en las páginas 242 y 248)