UNIVERSIDAD
POLITECNICA
DE VALENCIA

Máster Universitario en Computación Paralela y Distribuida

Departamento de Sistemas Informáticos y Computación

Paralelización mediante procesadores gráficos (GPU) del cálculo de la dinámica de electrones en plasma confinado magnéticamente

TRABAJO FINAL DE MÁSTER

Autor: Tomás Navarro Cosme

Director: José Enrique Román Moltó

Septiembre de 2015

.

Abstract

CUDA parallelization of a program, originally written in Fortran with MPI, for the calculation of the dynamic of electrons, taking particular attention to the formation of runaway electrons in a mangentically confined plasma, under the effect of an electric field, using the Langevin's approximate calculation method. The final implementation will retain the most basic part source code written in Fortran, a language with a large base of use in the field of Physics, to show its compatibility with the newest NVIDIA's Graphical Processors Units via CUDA. The implementation is progressive, showing successive technical improvements that can be applied to such codes, analyzing the obtained speedups.

Resumen

Paralelización en CUDA de un programa, originalmente escrito en Fortran usando MPI, para el cálculo de la dinámica de los electrones, con especial atención a la aparición de electrones fugitivos (*runaway*), dentro de un plasma confinado magnéticamente y que se mueven por efecto de un campo eléctrico, utilizando el método de cálculo aproximado de Langevin. En la paralelización se mantendrá la parte básica de Fortran, lenguaje con una gran base de uso en el campo de la Física, para mostrar su compatibilidad de uso con los aceleradores gráficos de NVIDIA usando CUDA. La implementación es progresiva, mostrando las sucesivas técnicas y mejoras que pueden aplicarse a este tipo de códigos, analizando las aceleraciones de cálculo obtenidas.

Palabras clave: CUDA, Fortran, GPU, GPGPU, computación numérica, código científico, HPC, plasma, Langevin, electrones *runaway*

ÍNDICE ABREVIADO

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE LISTADOS DE CÓDIGO

CAPÍTULO 1

INTRODUCCIÓN

Históricamente, la ciencia ha evolucionado gracias a la constante creación, evolución y cambio de su cuerpo teórico, partiendo de la observación de la naturaleza y de la experimentación con pruebas diseñadas por el hombre a tal fin, sobre todo cuando la mera observación de la naturaleza (empirismo) no permitía alcanzar el grado de detalle deseado, o no con la suficiente frecuencia o aislamiento de otros fenómenos naturales del entorno inmedianto. Este cuerpo teórico se ha ido complicando conforme ha avanzado el conocimiento de la materia en cuestión, de tal forma que en prácticamente todas las ciencias, pero especialmente en la física de partículas, las ciencias de los materiales, la nanotecnología, la bioquímica, la metereología y la geología sísmica, el conjunto de ecuaciones que son aplicables a un determinado fenómeno para su explicación detallada es tan complicado que, o no existe una solución analítica del problema, o si existe, es tan complicada que se precisan de métodos numéricos para su tratamiento.

Hasta la llegada de la informática, las simulaciones de fenómenos naturales eran muy costosas pues requería de muchísimas horas de cálculo manual para resolver casos relativamente sencillos. Con los primeros pasos de la informática los problemas pudieron empezar a tratarse de forma aproximada mediante la realización de simulaciones por ordenador para lo cual nacieron lenguajes de programación especialmente diseñados para facilitar la transcripción de las ecuaciones y algoritmos matemáticos a código fuente. Tal es el caso de Fortran, que en un principio se denominó FORTRAN, como una abreviatura de *FORmula TRANslation*, nombre claramente indicativo de su intención original. Se trataban de lenguajes simples estructural y sintácticamente, pero potentes, por cuanto sus compiladores fueron optimizados pensando en extraer la máxima capacidad numérica de los ordenadores existentes. Además, se crearon gran cantidad de bibliotecas matemáticas y de cálculo matricial y algebraico, tales como BLAS, LINPACK, EISPACK y MINPACK, muy eficientes, todas ellas con interfaz de programación basada en Fortran. En esa época se realizó mucho código base científico, histórico desde el punto de vista actual, que se ha heredado a través de los tiempos gracias a que la evolución de la informática permitió aplicarles de una forma relativamente sencilla, la paralelización gracias a bibliotecas específicas al efecto. Ejemplos de estas bibliotecas es MPI, que permitió que los códigos se ejecutaran más rápidamente mediante la acción conjunta y coordinada de muchos nodos computacionales interconectados por una red de comunicación. A eso también ayudó que se actualizaran las bibliotecas numéricas originales a este nuevo paradigma de programación paralela, con paquetes muy utilizados como LAPACK, ARPACK, PETSc, ScaLAPACK, Lis o SLEPc.

Esa evolución ha permitido la simulación de procesos físicos, químicos, biológicos o naturales cada vez

más complejos, a mayor escala, y con mejor precisión. No obstante, la complejidad sigue siendo tan enorme, y los intereses tan grandes en ciertas áreas por su innegable impacto económico a efecto mundial (tal como el cambio climático, detección de terremotos, la fusión nuclear, el descifrado del ADN), que cada vez se han ido construyendo mayores supercomputadores para la ejecución de simulaciones mediante programas masivamente paralelizados.

Recientemente, a partir de mediados de la década de los 2000, se ha abierto paso un nuevo hardware, originalmente pensado para el procesamiento gráfico, que ha servido para revolucionar la computación de altas prestaciones. Es el caso de los procesadores gráficos (GPU, *Graphics Processing Unit*), que se caracterizan por ser masivamente paralelos en cuanto a su organización, aunque las unidades de cómputo que lo componen no son tan genéricas ni tan flexibles como en las CPU tradicionales. Aún así, muchas aplicaciones se benefician de importantes incrementos de rendimiento con el uso de las GPU, principalmente a través de CUDA, utilizando tarjetas basadas en procesadores gráficos de NVIDIA. En la actualidad de entre los primeros 10 supercomputadores que lideran la jerarquía del Top500, la mitad usan aceleradores para incrementar su rendimiento, 3 de ellos utilizando Intel Xeon Phi, y 2 basados en GPU de NVIDIA.

Todo ello justifica el interés que existe por conseguir que el histórico código científico, principalmente escrito en Fortran y a veces ya paralelizado generalmente con MPI o OpenMP, se beneficie de esta nueva tecnología de paralelización basada en tarjetas gráficas y con la utilización de lenguajes como CUDA y OpenCL, así como de otros nuevos tales como OpenACC que prometen ser más sencillos que estos dos anteriores, y con mayor similitud con OpenMP, del que toma gran parte de su nomenclatura y conceptos.

El presente trabajo se trata de una prueba piloto para comprobar no sobre la viabilidad técnica de la transcripción de los programas en Fortran a CUDA, sino en analizar la forma en que incluirlo manteniendo la principal base de Fortran, y a la vez comprobar la aceleración obtenida con un caso de ejemplo de estudio. A partir de esta experiencia, se podrá valorar la dificultad del proceso, los requisitos necesarios, a nivel de programación general, y particularmente de C, CUDA y eventualmente otras bibliotecas paralelas, como OpenMP, y así servir de ayuda para el estudio de la viabilidad de la transcripción a gran escala de los antiguos códigos en Fortran (con o sin MPI) a utilizar el poder computacional de las nuevas tarjetas gráficas de NVIDIA.

El caso base elegido se trata de la simulación del comportamiento de un plasma confinado en un campo magnético, que es el caso base para poder estudiar los problemas que en estos fluidos pueden presentarse en instalaciones quasi-industriales para la obtención de energía eléctrica a partir de las reacción de fusión que se producen en dicho plasma cuando éste está formado por elementos ligeros, como el deuterio y el tritio. Ejemplo paradigmático actual de esas instalaciones es el *tokamak* denominado ITER (*International Thermonuclear Experimental Reactor*). Uno de los problemas técnicos con los que se enfrenta el trabajo, es el problema de la generación de ráfagas de electrones *runaway* (fugitivos), que son electrones que se aceleren en demasía por efecto del campo eléctrico perpendicular aplicado, y que escapan de su confinamiento magnético. Cuando estos electrones se generan en masa presentan dos problemas: las disrupciones, que abortan la reacción nuclear; y la generación de ráfagas, que provocan daños físicos importantes a los materiales que encapsulan el confinamiento magnético, con el consiguientemente debilitamiento estructural y posibles averías a las instalaciones primarias o secundarias del propio reactor.

CAPÍTULO 2

CONOCIMIENTOS PREVIOS

La idea directriz del presente trabajo ha sido la de mostrar las capacidades de cálculo numérico paralelo masivo que tienen las nuevas tarjetas gráficas, especialmente las de NVIDIA, y su idonedidad para la paralelización de códigos secuenciales o la reescritura de códigos paralelos existentes en el área científica, específicamente en la física de partículas elementales. A este respecto es importante conocer el estado en que se encuentran los conocimientos y desarrollos existentes en diferentes áreas, y que en el presente capítulo se repasarán.

En una primera aproximación al problema, debemos tener en cuenta las características de los códigos científicos actuales, los cuales, suelen llevar consigo una larga carga histórica de código heredado y convenientemente actualizado o modificado para su adaptación a los nuevos desarrollos teóricos del campo en cuestión. Por ello se repasarán las características de estos códigos, tales como el lenguaje de programación utilizado, su estructuración, muy ligada al desarrollo matemático de sus fundamentos teóricos, la utilización de determinados algoritmos numéricos, no siempre convenientemente implementados o escogidos para la tarea a realizar, al ser frecuentemente una mera transcripción a lenguaje de programación de las fórmulas matemáticas que se implementan, la utilización de tamaños especiales de variables, diferentes a los mucho más restringidos de C.

A continuación se expondrán las principales características del hardware gráfico que se va a utilizar para la realización de la computación numérica, concretamente de las tarjetas NVIDIA. Se verán sus características y se explicará en qué casos está recomendado su uso, y también en qué casos los resultados obtenidos no serán tan buenos. Para ello también será necesario presentar las diferentes familias de tarjetas NVIDIA, las cuales no sólo se diferencian por la cantidad de procesadores, su velocidad, y el tipo y cantidad de memoria dedicada interna que disponen, sino sobre todo es importante conocer sus diferentes niveles de capacidad computacional CUDA.

También se repasará el lenguaje de programación seleccionado para realizar la paralelización en GPU. En principio hay dos posibilidades principales, CUDA y OpenCL. Se expondrán ambos y se razonará porqué se ha escogido CUDA, así como las implicaciones que eso conlleva. Otro importante aspecto a tener en cuenta será que los códigos científicos suelen estar escritos en Fortran, por lo que la paralelización con tarjetas gráficas, tanto con OpenCL como con CUDA, que no están basados fuertemente en C, incluyendo la API de sus bibliotecas y *drivers* de bajo nivel.

Finalmente se expondrán otras tecnologías de paralelización, algunas de las cuales también han sido utilizadas e incluso incorporadas. Así por ejemplo tenemos MPI, una biblioteca de programación basada en el paradigma de memoria distribuida, mediante la utilización de paso de mensajes; y también OpenMP, que está especialmente pensado para memoria compartida. Últimamente han surgido otras posibilidades que intentan facilitar la labor de la paralelización abstrayéndose un poco más del hardware existente por debajo, de entre los cuales destacaremos el caso de OpenACC.

Por otra parte, la parte física del problema, por su especial relevancia y entidad, será tratada en el siguiente Capítulo 3 "Ecuaciones de Langevin para describir un plasma de partículas y la generación de electrones *runaway*", aunque sin entrar en excesivo detalle científico, sino más descriptiva de su significado, así como de las principales características a tener en cuenta a la hora de paralelizar el algoritmo de simulación.

2.1 La simulación computacional en la Física

En el ámbito de la física de particulas es muy habitual el uso de la simulación computacional. Esta utilización no se realiza para su estudio directo, sino como método de comprobación de las de hipótesis y nuevos desarrollos teóricos. En este campo de la física de partículas elementales, el método de observación está limitado por la máxima precisión alcanzable, determinada por el principio de indeterminación de Heisenberg, y ese límite físico acota también la extracción de conocimiento a partir de la experimentación. A esto se une la complejidad de las actuales teorías, que deben compaginar tanto fenómenos bien conocidos y teóricamente bien desarrollados, como la gravitación, la dinámica y el electromagnetismo, con los efectos cuánticos y la aplicación de la teoría de la relatividad, ya que es habitual trabajar con partículas que se mueven a altas velocidades (esto es, son de alta energía). Por todo ello, cuando se trabaja a nivel de partículas tales como electrones, neutrones, protones y partículas alfa, las ecuaciones teóricas no pueden resolverse analíticamente. El procedimiento habitual es adaptarlas al caso particular en estudio, realizando las oportunas simplificaciones e ndo que no afecten al resultado a obtener, y finalmente, con el conjunto de fórmulas simplificadas, y por lo tanto aproximadas, realizar intensos cálculos de simulación para comprobar si se pueden reproducir los resultados obtenidos experimental o empíricamente. En caso de que así sea, ese conjunto de fórmulas aproximadas puede someterse a un nuevo proceso de simulación que se extienda a casos más complejos que difícil o muy onerósamente pueden contrastarse experimentalmente. De esta forma se obtiene información valiosa que permite acelerar el proceso tecnológico y de ingeniería para la construcción de nuevas instalaciones de experimentación.

Este es el caso de la física de plasma, en donde partículas elementales de los átomos más ligeros se hallan confinadas por el efecto de fuertes campos electromagnéticos a altas temperaturas, inonizadas, libres entre sí, y produciéndose choques ocasionales de alta energía, que, en condiciones de tiempo, temperatura y densidad suficiente, pueden provocar el inicio del proceso de la fusión nuclear.

Desde el punto de vista informático, los procesos de simulación se caracterizan por ser computacionalmente intensos, dado que se debe tratar un número lo más grande posible de partículas, y porque generalmente interesa la dinámica del sistema, esto es, cómo evoluciona desde un determinado estado inicial. Esto implica generalmente que se debe calcular el estado inicial, después suponer un incremento de tiempo pequeño, para intentar linealizar lo más posible la dinámica del sistema, esto es, que los errores por las aproximaciones realizadas no se propaguen en exceso, pero no tan pequeño como para suponer una inmensa cantidad de incrementos "diferenciales" del tiempo, y con ello, una gran cantidad de nuevos estados a calcular hasta alcanzar el estado final buscado (generalmente, ver si alcanza un estado estacionario, o analizar y comprender cómo se comporta si no. Por lo tanto, las ecuaciones son complejas, con muchos cálculos por cada estado, y muchos estados a computar. Esto puede complicarse en función de la complejidad del desarrollo teórico utilizado. De esta forma, interesa obtener unas ecuaciones que calculen el estado

de cada partícula de forma independiente al comportamiento individual de cada una del resto, puesto que si existe interdependencia entre las partículas, el orden del número de operaciones a computar por cada estado pasa de ser de $\mathcal{O}(N)$, siendo N el número de partículas, a orden $\mathcal{O}(N \cdot m)$, siendo m el número de particulas vecinas con las que interactúa, o incluso $\mathcal{O}(N^2)$ si todas interactúan con todas en cada estado.

Esta alta necesidad de poder de cálculo implica que, si se utiliza un único procesador, el tiempo de cálculo sea excesivamente elevado, lo que conlleva problemas por: incremento del retraso entre diferentes simulaciones, con la consiguiente paralización de la validación de los desarrollos teóricos aproximativos realizados; mayor riesgo de finalización abrupta de la simulación, tanto por fenómenos externos (cortes de luz) como internos (sobrecalentamiento del nodo, error de software,...). La solución que se ha venido aplicando en los últimos 20 años ha sido la paralelización de los procesos, bien sea mediante OpenMP (en sistemas multiprocesadores de memoria compartida), o mediante MPI (paso de mensajes entre nodos con memoria distribuida). Todo ello es posible utilizando el lenguaje Fortran, que tiene su principal nicho de uso dentro del terreno de cómputo científico, puesto que: está pensado para transcribir las fórmulas matemáticas; permite la programación estructurada, generalmente idónea para implementar los habituales algoritmos de cálculo científico; permite un uso muy simplificado de las operaciones vectoriales y matriciales, muy habituales en ciencia; está orientado al cómputo, con compiladores muy optimizados al efecto; dispone de librerías específicas de cálculo numérico, algebraico y científico muy optimizadas, robustas y contrastadas; tiene una muy buena implementación y soporte en los dos históricamente principales paradigmas de programación paralela, OpenMP y MPI, dado que sus estándares se desarrollan e implementan teniendo muy en cuenta las particularidades y ventajas de Fortran. Este último aspecto es fundamental para continuar la herencia de muchos programas y bibliotecas de cálculo matemático y científico desde los años 60 hasta la actualidad, lo cual tiene aspectos muy positivos (programas contrastados, eficientes y robustos), pero que sutilmente esconde algunos inconvenientes que conviene tener en cuenta y analizar, como se reveló duante la fase de transcripción y adaptación al nuevo paradigma SIMD de CUDA, tal como se verá en este trabajo.

2.2 Características de los códigos científicos

Por código científico entenderemos en este trabajo aquel programa o biblioteca que implemente directamente un determinado algoritmo de computación científica. Este algoritmo de computación científica será generalmente un conjunto de ecuaciones interdependientes que permiten resolver un determinado problema físico. Generalmente, este conjunto de fórmulas proviene del desarrollo teórico en una determinada área científica, y más específicamente, nos centraremos en aquellas que sirven para la simulación de sistemas físicos.

Estos problemas tienen la característica común de que son lo suficientemente complejos como para que el conjunto de fórmulas derivadas estrictamente de la teoría sean demasiado complejas para su resolución analítica. Esta imposibilidad de resolución analítica impide que se pueda analizar el comportamiento de estos sistemas en diferentes situaciones prácticas. Al objeto de poder estudiarlos, se hace necesario realizar determinadas suposiciones que permiten aligerar la complejidad del conjunto de ecuaciones, mediante la inclusión de aproximaciones razonables para el caso concreto en estudio. Tras estas aproximaciones, es muy frecuente que se obtenga otro conjunto de ecuaciones más simple, pero aún así irresolubles o de difícil interpretación. Para resolverlas se recurren a métodos numéricos mediante la implementación de programas informáticos específicos para ese problema. Estos códigos científicos se ejecutan en múltiples instancias, puesto que es frecuente que estas funciones tengan términos estocásticos, esto es, que para capturar la variabilidad de las condiciones externas o de la complejidad interna innata del sistema, y ante la imposibilidad o gran dificultad de expresión teórica estricita, en el cuerpo teórico se introducen términos

de variabilidad aleatoria (denominados términos estocásticos), que permiten dotar al sistema de un cierto comportamiento y adaptación pseudoaleatoria necesario. Estas múltiples ejecuciones se denominan simulaciones, y permiten extraer nuevo conocimiento del sistema físico en estudio del análisis de sus resultados.

Históricamente, estos códigos científicos han sido implementados por los propios científicos teóricos encargados de desarrollar las fórmulas básicas y sus derivaciones aproximadas, y presentan las siguientes características:

- Lenguaje de programación: preferencia por lenguajes como Fortran o entornos como MatLab, sobre todo en comparación con C, con sintaxis más sintéticas en las instrucciones de cálculo matemático, especialmente al permitir trabajar con vectores y matrices sin necesidad de generar bucles que los recorran.
- Estructuración: pensada para la implementación rápida de algoritmos matemáticos. Los módulos permiten definir bibliotecas de funciones, y la evitación de bucles explícitos para operaciones matriciales permite una mayor claridad del código, permitiendo al programador a centrarse en el algoritmo y no en el código.
- Estructura: Los códigos cientíticos son principalmente de computación numérica. Por lo tanto suelen estar claramente estructurados en entrada o lectura de datos, preparación del cálculo, ejecución del cálculo (parte computacionalmente más intensa) y recopilación de resultados para mostrar o guardar. Por este hecho utilizan bibliotecas para la entrada y salida de datos (tal como silo, con formatos especiales como HDF5, y precisan de un lenguaje y bibliotecas muy completas y eficientes computacionalmente. Está fuertemente basado en biblitecas que resuelven los problemas computacionales más generales.
- Independiencia entre cálculo y visualización: En relación con lo anterior, el código científico está pensado para ser revisado y reescrito, y por ello se hace independiente de procesos posteriores como el de la visualización. Los programas generan ficheros en formatos estandarizados para su posterior tratamiento por otros códigos, o por paquetes de visualización de datos, como VisIt, o el propio MatLab
- Legibilidad: Pensados para ir actualizándolos conforme evolucione la teoría que genera el conjunto de ecuaciones a resolver. Por lo tanto, interesa que el lenguaje de programación evite artefactos propios del lenguaje que enmascare o dificulte la expresión y comprensión del algoritmo implementado.
- Fuerte influencia de los algoritmos: lo cual puede ser un problema. Muchas veces, la transcripción directa de la fórmula a código, hace que se incurra en la utilización de algoritmos computacionalmente ineficientes cuando son implementados por un científico no experto y que no considere el coste computacional de su implementación.
- Necesidad de tipos específicos de variables: ejemplo claro es la necesidad de trabajar con números complejos, y que la bibliotecas matemáticas y computacionales también lo hagan. También el reconocimiento de números especiales (i, e) y del correcto tratamiento de valores extremos (valores mínimos y máximos predefinidos, generación y trabajo con variables que hayan excedido el intervalo definido (NaN, *Not a Number*), para lo cual es muy interesante que se sigan el estándar IEEE 754 (*Institute of Electrical and Electronics Engineers Standard for Floating-Point Arithmetic*).

2.3 GPGPU

GPGPU [5], *General-Purpose computing on Graphics Processing Units*, consiste en la realización de cómputos de propósito general utilizando para ello procesadores gráficos, GPU, en lugar de los habituales procesadores genéricos, CPU.

Debido a la economía de escala y al exitoso mercado de tarjetas gráficas impulsado por el sector de los juegos en el PC y consolas, los procesadores gráficos que se utilizan en esas tarjetas gráficas han evolucionado desde ser unos coprocesadores que simplemente representaban caracteres o gráficos de mapa de bits, a ser complejos procesadores capaces de realizar no tan simples cálculos matemáticos y vectoriales sobre cada píxel para poder dotar al conjunto de la imagen de un mucho mayor realismo, a la vez que proporcionar un alto refresco en la imagen para suavizar las transiciones entre cada instantánea [11].

De esta forma, el poder computacional de las GPU se ha multiplicado mucho más en estos últimos años que el de los procesadores de propósito general, CPU. Una tarjeta gráfica de alta gama puede tener perfectamente más de 2800 procesadores individuales todos ellos trabajando de forma (casi) independiente en paralelo, y con capacidad de comunicarse a través de rápidas memorias DDR5, incluso de varios GB de capacidad, ubicadas en la misma tarjeta [9]. No obstante, su conjunto de instrucciones es mucho más específico, orientado a la realización de cálculos gráficos, y se resiente de la falta de instrucciones optimizadas muy comunes en la programación estándar no gráfica (saltos y comparación, continua necesidad de trasvasar gran cantidad de información de la CPU a la GPU y viceversa, por ejemplo). Tampoco tienen la universal flexibilidad de las CPU pues tiene diferentes zonas de memoria, cada una con usos específicos y no intercambiables, o la coordinación entre procesadores dentro de un hilo de ejecución. Otra característica que penalizaba mucho las anteriores tarjetas gráficas era la inexistencia de instrucciones específicas para el flujo de control, toda vez que desde el punto de vista meramente gráfico, los programas siempre han sido preferentemente secuenciales. En la actualidad, los procesadores gráficos ya incorporan instrucciones para el control de flujo, pero las ramificaciones (`if-then-else`) son muy costosas, sobre todo si en cada hilo el resultado es diferente, lo que provoca graves penalizaciones en tiempo, para conseguir una adecuada sincronización en la ejecución del programa, por lo que conviene tenerlo muy en cuenta para obtener buenos resultados [6]. Estas restricciones se han suavizado con la llegada de las nuevas arquitecturas Kepler y Maxwell, pero aún así son aspectos de funcionamiento que conviene seguir teniendo en consideración para la optimización de la ejecución del código en las GPU.

Aún así, a mediados de los 2000 empezaron los experimentos de realizar cómputos matemáticos pesados usando órdenes puramente gráficas, principalmente basadas en el estándar gráfico OpenGL. A pesar de la dificultad de programación ofrecida por esta vía, sí que se comprobó empíricamente la gran potencialidad que tenían las GPU para el cómputo numérico, y fruto de ello, NVIDIA lanzó en 2007 su lenguaje de programación propietario denominado CUDA C. Desde entonces prácticamente NVIDIA ha ido proporcionando la capacidad de ejecución de programas CUDA a la gran mayoría de sus tarjetas gráficas. Esta característica está especificada mediante la denominada "capacidad de computación CUDA" (en inglés, *CUDA capabilities*), que consiste en un número que permite conocer qué características y qué comportamientos tiene la tarjeta gráfica para la ejecución de código CUDA, sobre todo teniendo en cuenta que el lenguaje y las prestaciones ha evolucionado mucho desde 2007 hasta la fecha. Además, la numeración también tiene cierta correlación con las diferentes arquitecturas hardware que han aparecido: Tesla, Fermi, Kepler y la más reciente Maxwell.

Posteriormente, Apple, liderando un conjunto de otras compañías (AMD, IBM, Intel y NVIDIA), impulsó el lanzamiento de otro lenguaje para GPGPU, denominado OpenCL [13], que al contrario que CUDA es de especificaciones abiertas. Actualmente el encargado de la especificación es el Grupo Khronos [8], quien también es el responsable de la especificación de OpenGL [14].

Por ello, mientras CUDA únicamente está implementado en los procesadores gráficos de NVIDIA por ser un lenguaje propietario y totalmente cerrado, OpenCL puede utilizarse no sólo en los procesadores gráficos de diferentes fabricantes, como ATI [2], Intel [7] y la propia NVIDIA [12], sino que incluso hay CPUs que son capaces de ejecutar código de OpenCL (IBM Power 77X e IBM BladeCenters, Intel Ion y la tercera generación de Intel Cores [7] y los recientes procesadores de AMD, incluidos los AMD Fusion [1]). Samsung también dispone de una implementación de OpenCL que puede ejecutarse sobre procesadores Cell BE, ARM y sobre DSP [15].

A pesar del mayor número de fabricantes que proporcionan hardware compatible con OpenCL, CUDA es un lenguaje con una mayor evolución, y su compilador está mucho más optimizado para el hardware al que va destinado, dado que está específicamente diseñado para utilizar todo el potencial de los procesadores gráficos de NVIDIA. OpenCL es mucho más genérico, e incluso, como ya se ha comentado, puede ser ejecutado por CPU, por lo que tiene un mayor grado de abstracción sobre el hardware y por tanto, no permite emplear instrucciones o recursos específicos de ciertos procesadores gráficos. En la Tabla 2.1 se compara el desarrollo de cada uno de estos lenguajes.

CUDA Versión del *Toolkit*	Fecha	OpenCL Especificación
1.0	julio 2007	
1.1	noviembre 2007	
	junio 2008	1.0
2.0	agosto 2008	
2.1	diciembre 2008	
2.2	mayo 2009	
2.3	julio 2009	
3.0	marzo 2010	
3.1	mayo 2010	
	junio 2010	1.1
3.2	noviembre 2010	
4.0	mayo 2011	
	noviembre 2011	1.2
4.1	enero 2012	
4.2	abril 2012	
5.0	octubre 2012	
5.5	julio 2013	
	noviembre 2013	2.0
6.0	abril 2014	
6.5	agosto 2014	
	enero 2015	2.1 provisional
7.0	marzo 2015	
7.5	septiembre 2015	

Tabla 2.1: Comparación de la evolución cronológica de CUDA y OpenCL

No obstante, es necesario recalcar que la computación a través de GPU puede en algún momento sobrevalorarse. Existen campos en los que las actuales CPU son tan capaces como las más potentes GPU, y es por ello que siempre es necesario valorar primero si el problema a abordar, o cada una de las partes en que se pueda dividir, debe serlo a través de computación genérica o de computación basada en GPU. Así,

en campos tales como las matrices dispersas, los cálculos en que en cada procesador pueden divergir en cuanto a su flujo de control, o en los que existe una gran dependencia entre datos en diferentes ubicaciones espaciales, la computación genérica puede ser más eficiente, tal como se señala en [16].

2.4 CUDA

CUDA son las siglas de *Compute Unified Device Architecture*. Ha sido propuesta, implementada y desarrollada exclusivamente y de forma cerrada por NVIDIA. Se trata de una arquitectura hardware para procesadores gráficos que permite que éstos puedan realizar cómputos de carácter general (no gráficos) de una forma sencilla y directa, sin necesidad de invocar directamente las primitivas gráficas originales basadas en OpenGL o DirectX. Todo ello sin disminuir su propia capacidad intrínseca de ejecución de tareas gráficas basadas en cualquiera de estos estándares gráficos.

2.4.1 Lenguaje CUDA C

Este concepto arquitectónico es aprovechado en la práctica mediante la definición de un nuevo lenguaje que es capaz de abstraer la complejidad del multiparalelismo de los actuales procesadores gráficos manteniendo una estructura y nomenclatura muy similar al lenguaje C, universalmente utilizado para la programación genérica usando CPUs. Este nuevo lenguaje se denomina CUDA C, y en lo sucesivo, en este trabajo, se abreviará simplemente por CUDA, el mismo nombre que la arquitectura.

Se trata de un lenguaje de programación de especificaciones cerradas, que desarrolla en exclusiva NVIDIA, que permite la utilización de sus procesadores gráficos para computación genérica, no exclusivamente gráfica. Los programas realizados con CUDA solamente puede ejecutarse en aquellos procesadores gráficos que tengan «capacidad CUDA» (CUDA *capability*).

Es importante diferenciar entre la versión del *Toolkit*, que representa el API de programación en CUDA con una sintaxis basada en el lenguaje C, y la capacidad de computación CUDA. Esta última se trata de una característica del hardware, esto es, de la GPU, y representa qué subconjunto de instrucciones es capaz de ejecutar directamente cada procesador gráfico. Existe una evidente interrelación, por cuanto en cada versión del *Toolkit* se menciona expresamente qué capacidad CUDA es necesaria para cada una de las funciones de librería que se proporciona, al objeto de utilizar optimamente el lenguaje en función del hardware que se vaya a utilizar. En [3] se listan todas las GPU que tienen capacidad de computación CUDA indicando expresamente su nivel de capacidad, que actualmente llega hasta la 5.2 para las GPU GeForce más avanzadas (serie GTX 970 y 980).

2.4.2 Tipos de código en CUDA C

La programación en CUDA se caracteriza por contener dos tipos bien diferenciados de bloques de código: el código del anfitrión (*host code*) y el código del dispositivo (*device code*).

- Anfitrión (*host*): Por anfitrión se entiende el entorno hardware de ejecución genérico proporcionado por las CPU convencionales. Su código es puro C/C++ y no precisa nada de los procesadores gráficos para su ejecución, pero obviamente, no se aprovecha el potencial de ellos.
- Dispositivo (*device*): Por dispositivo se entiende el entorno hardware de ejecución propio de los procesadores gráficos, GPU. Su código, como ya se ha comentado, tiene una estructura y sintaxis muy similar al C, pero extendido para permitir programar fácilmente las nuevas capacidades que posee la arquitectura CUDA.

CUDA proporciona un compilador propio, denominado `nvcc`, que está especializado en la compilación de código de dispositivo, pero que también compila la parte del anfitrión y efectúa el enlace entre ambos códigos en un único programa ejecutable.

El código del dispositivo se caracteriza por estar basado en *kernels*, que tienen la apariencia de funciones estándar C, pero con una nomenclatura de llamada especial, y que instruyen al procesador genérico de la CPU a enviar el código y los datos necesarios a la GPU, donde dicho *kernel* efectivamente se ejecuta. Los resultados de esos *kernels* pueden devolverse directamente a la CPU, para lo cual es necesario hacer una nueva transferencia de datos, en este caso entre la GPU y la CPU, o incluso mantenerse dentro de la memoria de la GPU para reutilizar en posteriores ejecuciones de nuevos *kernels*.

Consiguientemente, la CPU siempre es la que lleva el control de ejecución del programa global, pero va delegando en la GPU aquellas partes del programa que permitan realmente obtener una gran ventaja de la paralelización masiva que ofrecen las GPU.

2.4.3 Compilación de aplicaciones CUDA

La compilación de las aplicaciones CUDA se realiza fundamentalmente con el *driver* compilador `nvcc`. `nvcc` puede realizar muchas funciones, siguiendo caminos de compilación diferentes según las necesidades, tal como se muestra en la Figura 2.1. Una de las principales consiste en separar el código del anfitrión (*host*, que está en C) del dispositivo (*device*, para el que se usan extensiones específicas CUDA C). Una vez separados los códigos, lanza los compiladores específicos de cada tipo de código por separado para después ensamblarlos en un único ejectuable, generalmente. La parte del anfitrión se delega en un compilador de uso genérico, como pueden ser `gcc` o `icc`.

La compilación del código de la tarjeta gráfica es mucho más crítica, y sólo puede realizarse con las herramientas específicas proporcionadas por NVIDIA, dado el carácter cerrado de la especificación hardware de sus tarjetas, a pesar de ser pública la especificación del código intermedio, formato denominado PTX (*Parallel Thread eXecution*), para el cual NVIDIA provee el compilador `ptxas`. `ptxas` puede ser invocado directamente por `nvcc`, pero también puede hacerse en tiempo real por la propia aplicación, realizando una compilación JIT (*Just In Time*) del código intermedio a código máquina específico de la arquitectura y capacidades CUDA de la tarjeta concreta en la que se va a ejectura el *kernel*.

La compilación del código puede realizarse de tres formas básicas, hacia tres formatos diferentes, a saber:

- `.cubin`: El formato `cubin` es microcódigo nativo específico de una tarjeta concreta NVIDIA con capacidad CUDA. Por lo tanto, es directamente ejecutable por ella. Los códigos binarios `cubin` en general no pueden intercambiarse entre diferentes arquitecturas o diferentes capacidades CUDA (CUDA *capabilities*.
- `.ptx`: Es la representación intermedia del código a ejecutar en la GPU. Éste puede ser compilado a su vez y convertirse en código binario `cubin`, arriba descrito. El formato `ptx` es neutro, en el sentido de que es independiente de la arquitectura y de las capacidades computacionales de la tarjeta NVIDIA en la que finalmente se ejecutará. Está diseñado para ser resistente a cambios futuros («*future-proof*») por innovaciones tecnológicas hardware y software. Puede ser compilado *offline* por el compilador *driver* `nvcc`, o compilado automáticamente *online* JIT según se necesite.
- `.fatbin`: Es un formato especial que contiene en su seno diferentes compilaciones o precompilaciones, bien sea por contener tanto código `cubin` como código `ptx`, bien por tener código dirigido a diferentes arquitecturas o con diferentes capacidades CUDA, así como todo ello a la vez.

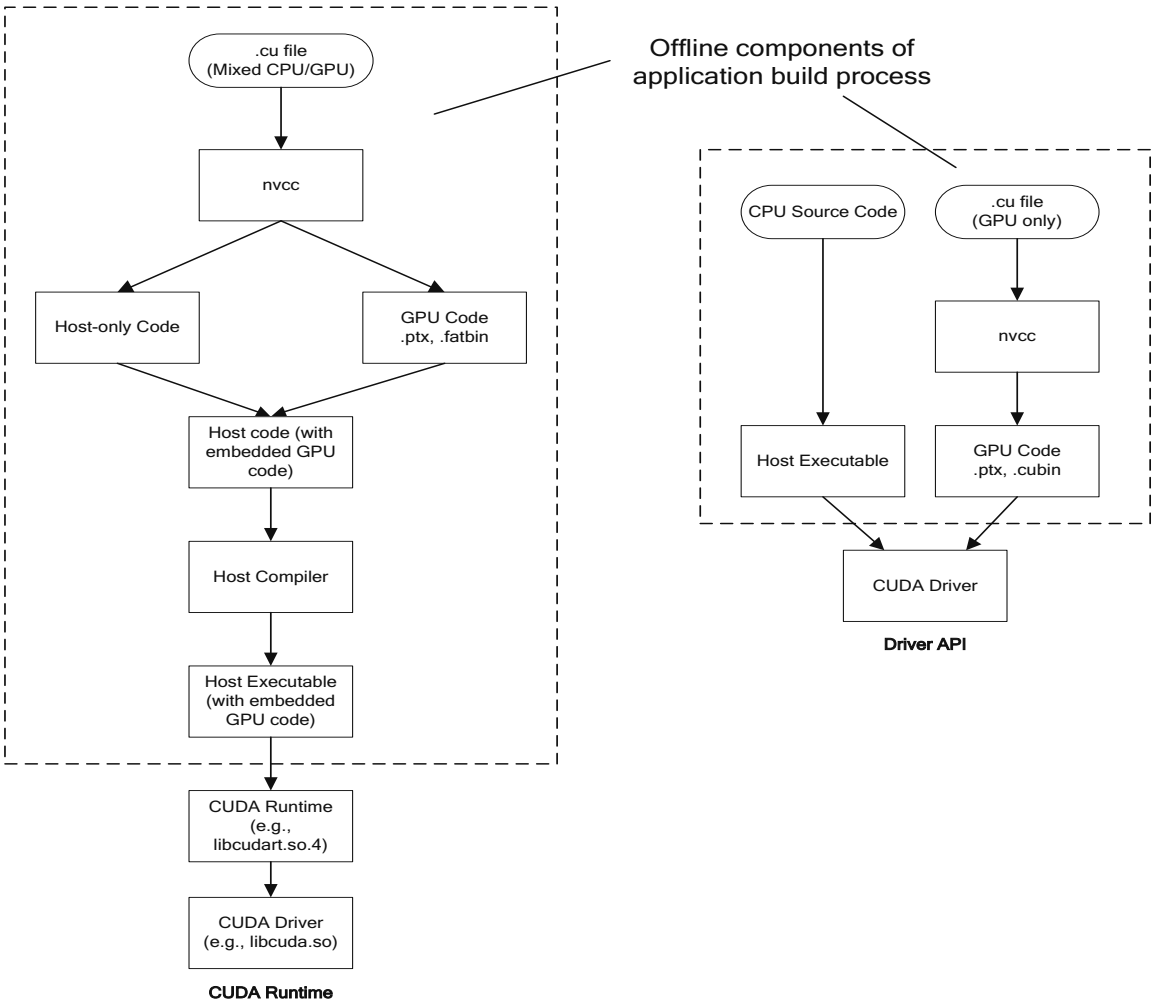


Figura 2.1: Diagrama de los flujos de compilación más habituales de `nvcc`

Es evidente que los `cubin` son más rápidos en cargarse en la GPU, puesto que se envían directamente en cuanto debe ejecutarse dicho *kernel*. Por su parte, los `ptx` son más portables, pues pueden compilarse a cualquier arquitectura o capacidad igual o superior a su formato, pero incurren en la considerable sobrecarga de la compilación en línea justo en el momento en que debe lanzarse a la GPU. Los `fatbin` son ficheros más voluminosos, pero con mayor compatibilidad, según lo que se incluya, sobre todo si se incluyen `ptx` de diferentes arquitecturas y capacidades.

Nótese (y esto es crucial para el presente trabajo) que la compilación específica del `ptx` se realiza justo en el momento en que se va a lanzar el *kernel*, no durante la carga en memoria de la aplicación al inicio de su ejecución. De igual forma que la selección del `cubin` concreto, en el caso de que ya estén incluidos varios y no sea necesario compilar.

Es interesante indicar que los *kernels* CUDA pueden estar incrustados en el propio fichero ejecutable ELF (o EXE, en Windows), o también pueden cargarse durante la ejecución desde otros ficheros, en cualquiera de los tres formatos arriba indicados. De hecho, cuando el *kernel* está incluido en el programa principal, está en formato cadena (*string literals*), y lo que se hace es leerlos y pasarlos a la GPU mediante unas instrucciones específicas previas, que se encargan también de lanzar su ejecución dentro de la tarjeta gráfica.

Por último, la correcta selección del tipo de tarjeta gráfica para el cual se quiere compilar (directamente a código binario `.cubin` u obtener código intermedio `PTX`) es esencial. Esta selección se verá de vital importancia a la hora de la migración, dado que en todo caso, la aplicación que se lanza deberá, como requisito básico, poder ejecutarse tanto en la GPU inicial como en la GPU a la que se migra.

CAPÍTULO 3

ECUACIONES DE LANGEVIN PARA DESCRIBIR UN PLASMA DE PARTÍCULAS Y LA GENERACIÓN DE ELECTRONES *runaway*

En este capítulo resumen brevemente la base física nuleótica que soporta el conjunto de ecuaciones que son resueltas numérica a través del código original de Langevin realizado en Fortran, y que permiten la simulación del comportamiento de los electrones de un plasma a alta temperatura confinado magnéticamente, y eventualmente, bajo la acción de un campo eléctrico perpendicular.

Primeramente se introducirá el concepto de plasma, y su importancia económica por el impacto que podría llegar a suponer la obtención de una reacción de fusión nuclear con balance positivo de energía, que resultaría en una fuente relativamente limpia de energía a partir de elementos muy comunes en nuestro entorno, tales como el Helio, el Deuterio, el Tritio y el Litio.

Posteriormente, se repesará muy por encima, la base teórica de la física de partículas que explica el comportamiento de los plasmas, haciendo especial mención a una forma de abordarlas denominada ecuaciones de Langevin. Este método de Langevin permite, mediante unas aproximaciones razonables, obtener unas ecuaciones relativamente sencillas, sobre las que es posible aplicar métodos numéricos y con ello conseguir la simulación del comportamiento del plasma. Este trabajo se centrará específicamente en la simulación de un tipo de partículas muy iteresantes, los electrones fugitivos (runaway).

3.1 El plasma y su aplicación práctica: reacción controlada de fusión

El plasma es un estado de la materia en el cual ésta está sometida a altas temperaturas (generalmente millones de grados Kelvin), a resultas de lo cual, sus partículas están totalmente ionizadas. En este estado su comportamiento es similar al de un gas, por la gran libertad de movimiento de sus partículas. Esa gran temperatura y gran libertad, implica una alta energía cinética, esto es, unas altas velocidades dentro del medio. Las condiciones necesarias para la generación y mantenimiento de la materia en estado de plasma se dan comúnmente en la naturaleza, pero no en nuestro inmediato entorno, sino en el interior de las estrellas, por ejemplo.

El plasma, en las condiciones normales de la Tierra, no puede subsistir apenas unos milisegundos dado que el altísimo gradiente de energía, y por tanto de energía, existente entre él y su alrededor, hace que rápidamente se tienda a enfriar, y con ello perder las condiciones necesarias para el plasma. Sin embargo, sí que puede crearse y confinarse en un espacio determinado mediante la acción de un campo magnético lo suficientemente grande como para crear unas líneas de fuerza cerradas, generalmente de forma helicoidal o toroidales, que limitan la capacidad de las partículas de escapar de él.

No obstante, aún bajo confinamiento magnético, a veces reforzado por campos eléctricos verticales, el comportamiento errático de las partículas dada la probabilidad de choque entre ellas, con cambios en el momento y la energía cinética, permite que eventualmente alguna partícula, generalmente las menos pesadas, como los electrones, se escapen. Este fenómeno por el cual un electrón se escapa es más frecuente cuanto a más temperatura, más concentración de materia y más tiempo esté confinado el plasma. A estos electrones se les denomina electrones fugitivos, o en terminología inglesa muy extendida, *runaway electrons*, o directamente electrones *runaway*.

El interés de los plasmas, a parte del teórico de investigación sobre la física de partículas y el entendimiento último sobre el comportamiento de las partículas en situaciones extremas, es un interés eminentemente económico y estratégico: la obtención de una fuente de energía virtualmente inagotable y relativamente limpia en su producción. Esto es así porque los plasmas, cuando están confinados durante el sufiente tiempo, a la alta temperatura adecuada, con la densidad de materia necesaria y se trata de iones de bajo peso atómico, tieden a chocar violentamente entre sí con mucha frecuencia de tal forma que espontáneamente se genera una reacción nuclear de fusión.

En esta reacción nuclear, dos átomos ligeros (generalmente algunos isótopos especialmente reactivos de hidrógeno, helio o litio), se unen formando un átomo más pesado, que tiene menor masa total que la suma de la de los iones origianles. Esta pérdida de materia, implica, que por la equivalencia entre materia y energía dada por la conocida fómula de Einstein, $e = m \cdot c^2$, se libere una cantidad equivalente de energía, que para nuestras escalas ordinarias es mucha.

Si fuera posible controlar esta reacción, la energía obtenida sería muy grande, mientras que los subproductos de este tipo de reacciones nucleares tienen unos tiempos de vida medio muy inferiores a los generados en una reacción nuclear de fisión. Otra ventaja es que mientras la fisión precisa de minerales muy escasos en la naturales (uranio, plutonio, torio), los elementos idóneos para la fusión nuclear son muy abundantes en la naturaleza (deuterio -agua de mar-, tritio -aparece junto al litio, componente principal de la corteza terrestre-, helio -en la propia atmósfera terrestre).

Existe una condición física, formulada por Lawson, que establece la condición mínima que debe tener un plasma para que la reacción de fusión sea espontánea y si se mantenga en el tiempo. Esta condición mínima se denomina *criterio de Lawson* y se suele expresar de la siguiente manera:

$$n \cdot R \cdot \tau \geqslant 5 \times 10^{21} m^{-3} \cdot k \cdot e \cdot V \cdot s$$

siendo el producto $n \cdot T\dot{\tau}$ conocido como *producto triple de fusión*, en donde n es la densidad de partículas, T la temperatura del plasma y τ el tiempo de confinamiento.

3.2 Confinamiento artificial del plasma

Si se analiza el criterio de Lawson, existen diferentes formas de conseguir la condición de ignición de un plasma. Puede ser que exista una alta densidad de partículas, en cuyo caso será necesario o una temperatura menor global del conjunto, o será necesario un tiempo de confinamiento menor. Por otro lado, si el

plasma tiene una baja densidad, será necesario o mantener su confinamiento durante mucho tiempo, o incrementar mucho su temperatura.

En las estrellas, por ejemplo, la fuerza gravitacional es tan grande que consigue que la densidad en su interior, n, sea muy elevada. La temperatura, T, dado que está en constante reacción nuclear de fusión, es muy elevada. Esto hace que su tiempo de confinamiento, τ, alcance valores de varios miles de millones de años.

En la búsque del confinamiento artificial del plasma se ha comprobado experimentalmente, que una de las opciones más prometedoras es la del denominado esquema *tokamak* (ver Figura 3.1, propuesto ya a mediados de la década de 1950 por los científicos rusos. El confinamiento se consigue mediante potentes campos magnéticos ayudados por un campo eléctrico en sentido vertical, pasando a través del eje del toro formado por el plasma confinado.

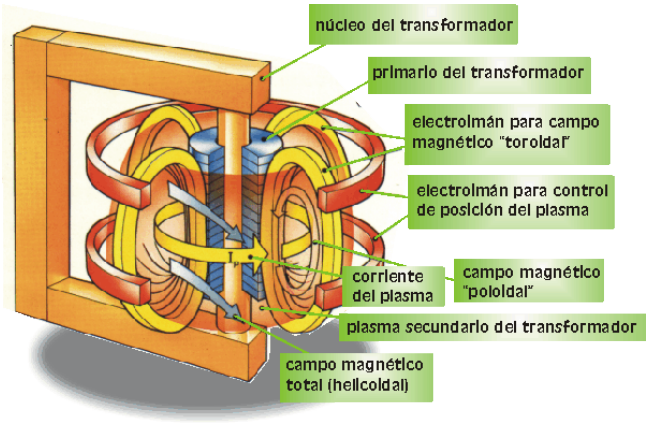


Figura 3.1: Esquema *tokamak*, consistente en un confinamiento magnético toroidal

Las condiciones ideales en este esquema implican mantener el plasma a una temperatura comprendida entre 100 y 200 millones de Kelvin.

Uno de los principales problemas existentes en este esquema es el ya mencionado de los electrones fugitivos, que cuando se generan en forma de racha, provocan importantes daños estructurales a la capa interior de la construcción que rodea al todo de plasma confinado. Por lo tanto, el estudio de su física, para encontrar condiciones y formas de evitar o minimizar la formación de estos electrones runaway, es un área de estudio muy actual, sobre todo durante la construcción del gran reactor tipo tokamak a escala semi-industrial denominado ITER, que se está construyendo en Francia.

3.3 Física de partículas: la ecuación de Langevin

El método de Langevin consiste en la realización de ciertas suposiciones, que se espera se den en los reactores tipo tokamak, por las cuales se incluyen términos estocásticos (esto es, de comportamiento fenomenológico aleatorio) para explicar el comportamiento aparentemente errático de los electrones y demás partículas debido al choque a altas velocidades entre ellas, principalmente con los iones.

La función resultante y su derivación se puede consultar en [4].

Esa función es la que se implementa en el código original en Fortran, y que se usa como caso de estudio en este proyecto para valorar la viabilidad de la utilización del nuevo paradigma de programación paralela

basada en GPU, y con ello incrementar la posibilidad de realizar simulaciones con más cantidad de partículas, y durante más tiempo, o de limitar las suposiciones realizadas durante el desarrollo teórico para obtener la ecuación de Langevin arriba indica, y mejorar con ello su representatividad.

CAPÍTULO 4

IMPLEMENTACIÓN DE LA PARALELIZACIÓN CON CUDA

En el anterior apartado se han resumido el conjunto de ecuaciones de Langevin que explican, de forma parcial por las aproximaciones asumidas en su derivación, el comportamiento, en términos de velocidades, de un plasma de partículas sometido a confinamiento magnético y bajo la acción (o no) de un campo eléctrico perpendicular.

Este conjunto de ecuaciones ya fueron expresadas de forma computacional usando el lenguaje de programación Fortran, y paralelizado haciendo uso de la biblioteca de paso de mensajes MPI. Ese código fuente original, es el que se va a estudiar y finalmente transformar para permitir que pueda aprovecharse de la capacidad computacional de las tarjetas gráficas de NVIDIA. Para ello se utilizará CUDA como lenguaje de programación.

En este capítulo se resumen los pasos y decisiones tomados para la adaptación de dicho código científico histórico original a CUDA, indicando las principales consideraciones iniciales realizadas, las dificultades y experiencias encontradas durante su implementación, la detección de posibles errores en el código original, las mejoras computacionales que sobre dicho código original pueden aplicarse (con o sin CUDA), y finalmente, explicando las propiedades que poseen las dos implementaciones finales que se presentan, así como la tecnología en que se basan.

4.1 Selección del lenguaje auxiliar de programación

La idea general es mantener el máximo código Fortran posible y a la vez conveniente, pensando en que así las modificaciones para la utilización de CUDA sean lo más sencillas posibles.

Al principio se barajaron las siguientes posibilidades:

- Utilizar PGI Fortran CUDA

  Esta opción se deshechó, dado que el periodo de tiempo disponible no parecía suficiente para realizar la implementación, comprobar su corrección, y ejecutar las pruebas de rendimiento.

  Sin embargo, es la más sencilla en principio, pues añadiendo unas pocas palabras y características más al lenguaje Fortran, permite invocar a la API del *driver* y del *runtime* de CUDA directamente. Esto

quita complejidad a la transición, y es quizá una opción muy interesante si se tiene el presupuesto adecuado.

- Utilizar, crear o ampliar un conjunto de subrutinas en Fortran que permitan ocultar la complejidad de las llamadas a CUDA.

  Por un lado está la dificultad de que al ser código en Fortran 90, no posee el módulo `ISO_C_BINDING`, incluido a partir de Fortran 95, y que facilita mucho la llamada a API basadas en C. No obstante, el principal factor de descarte fue que crear una biblioteca de este tipo, y pretender usarla en el futuro, implica tener que estar actualizándola muy frecuentemente para adaptarse a las modificaciones y mejoras que va introduciendo NVIDIA en cada nueva versión CUDA. Sobre todo, teniendo en cuenta que en las últimas versiones están haciendo obsoletas varias de las primitivas existentes en las primeras versiones de CUDA.

- Utilización de un middleware que facilite la programación desde Fortran con CUDA. Existe un proyecto a tal efecto, FortCUDA (`http://sourceforge.net/projects/fortcuda/`).

  Básicamente es lo mismo que arriba, pero ya elaborado. El programa se probó y funcionó correctamente, pero conviene que el código de origen sea ya Fortran 95 ó Fortran 2003. En caso contrario, el compilador realiza previamente una serie de transformaciones para acomodar el código en Fortran 77 ó Fortran 90 a Fortran 95. En ese punto se consideró que era muy peligroso confiar en que todo lo hiciera correctamente, y se aparcó su utilización.

- Utilizar el método de las subrutinas o funciones envoltorio (*wrapper*) en C.

  Éstas son unas funciones intermedias básicas en C que son llamadas de forma estándar desde Fortran, y que realizan la transformación necesaria del espacio de nombres, sobre todo para acceder a las variables globales declaradas en el código en Fortran. Esto es, gestionar el *name mangling*. Una vez acomodadas las variables, se invoca desde C de forma usual a CUDA. Los resultados son puestos a disposición de la subrutina Fortranllamante cuando se devuelve el control. Hay varios detalles que tener en cuenta, pero sobre todo, y mucho más en computación, hay que recordar que el almacenamiento en memoria de las matrices (*arrays*) es diferente entre ambos lenguajes: en Fortran se almacena por columnas primero, mientras que en C y CUDA el orden es primero por filas.

  Esto no puede ser resuelto sin excesivo sobrecoste por la función *wrapper*, por lo que debe ser tenido en cuenta en todo momento durante la implementación en CUDA y en C de todo bucle o acceso a las zonas de memoria multidimensionales originarias de Fortran.

Finalmente la opción escogida fue esta última, lo que implica que la implementación consistió en modificar el código original de Fortran del bucle básico de control de flujo del algoritmo, y sustituir las subrutinas Fortran que realizan la parte computacional o de comunicación, por unas llamadas a su respectiva función envoltorio. Esas funciones envoltorio ya llaman desde C a las funciones que realmente invocan a los *kernels* (programados en CUDA) para su ejecución en la tarjeta gráfica. La parte de escritura a disco, por compatibilidad con ficheros de configuración o previamente generados y por no ser necesario, se mantuvo en Fortran.

4.2 Utilización de *wrappers* en C para ser invocados por Fortran

A continuación se señalan las principales cuestiones que hay que considerar y resolver para conseguir posibilitar la utilización de *wrappers* escritos en C para ser invocados desde código fuente Fortran, debido dos cuestiones principales:

- la diferente forma en que se crean, nombran, definen y utilizan los símbolos generados por los diferentes compiladores, según el lenguaje y el suministrador del propio compilador,
- las diferencias propias existentes en las especificaciones de los lenguajes de programación involucrados (C y Fortran) en cuanto a la invocación de funciones o subrutinas.

4.2.1 Respeto de la nomenclatura (*name mangling*) de los símbolos según compilador

Desafortunadamente, cada compilador con cada lenguaje, tiene una forma diferente de nombrar definitivamente a las variables y zonas de memoria en los ficheros objetos (extensión `.o`). Este hecho dificulta el enlazado entre diferentes ficheros objeto compilados con diferentes compiladores o aún utilizando la misma familia de compiladores (GNU o Intel, por ejemplo), que los códigos fuente estén en diferentes lenguajes (como nuestro caso, parte en Fortran, parte en C y parte en CUDA-C).

Por ello es importante conocer el formato del *name mangling* de cada compilador para cada lenguaje que usemos en nuestro proyecto. Durante este proyecto, se han tenido que usar los compiladores `gfortran` y `gcc` de GNU, el `nvcc` de NVIDIA y los `icc` e `ifort` de Intel. `gcc`, `icc` y `nvcc` son compatibles entre sí porque el *name mangling* está claramente estandarizado en los lenguajes C y C++, pero eso no es así cuando entran en juego Fortran, y por lo tanto `gfortran` e `ifort`, que no está suficientemente estadarizado.

Junto a ello, es importante tener en cuenta el criterio de distinción de los símbolos por mayúsculas o minúsculas, que depende del lenguaje. Fortran no distingue, mientras que C. C++ y CUDA-C, sí lo hacen. A pesar de su indiferenciación en Fortran, sus compiladores están obligados a generar símbolos coherentes para el enlazado normal con otros ficheros objeto posiblemente generados por lenguajes que sí lo hagan. Es por ello que la norma habitual es que aunque Fortran no distingue mayúsculas de minúsculas en su código fuente, sí que tiene un criterio fijo interno (dependiente de cada compilador, eso sí), para la generación de símbolos en sus ficheros intermedios objeto `.o`.

Finalmente hay que distinguir la nomenclatura de las funciones (o subrutinas en Fortran) con las de las variables.

Si existe alguna duda, es de mucha ayuda la utilidad Linux `nm` (derivada de las palabras ***n**ame **m**angling*), que se utiliza utilizando como parámetro el nombre de un fichero objeto. Esta utilidad muestra por consola el listado de los símbolos en él usados. Junto con el nombre, aparece una serie de símbolos que permiten saber si ese objeto está declarado allí o simplemente son conocidos, pero que referencian a otro fichero objeto diferente que se conocerá en el momento del enlazado final.

Esto es de gran ayuda si nos enfrentamos a un nuevo compilador, o una versión muy diferente del anterior, puesto que permite ver la forma definitiva en que son conocidas las variables, y por lo tanto, adaptar nuestro código a este nuevo compilador a través de los ficheros de cabecera `.h`.

Acceso a variables globales declaradas en Fortran desde C y CUDA

Un problema existente en el código es el elevado número de variables declaradas con ámbito global (esto es, accesible directamente desde cualquier subrutina de Fortran), muchas de ellas provenientes de las opciones de ejecución de la simulación, a partir de la lectura de los ficheros de configuración, y otras de variables y matrices declaradas así para ser accesibles desde diferentes subrutinas, como el caso de los datos estadísticos, que son generados en unas y reutilizados y grabados en disco por otras.

El primer problema es el del nombrado de estas variables, aspecto ya discutido en el apartado anterior. El segundo es cómo utilizar estas variables tanto dentro del código en C como en CUDA.

Para su utilización en C (básicamente, dentro de las funciones *wrapper*), se crean unos ficheros de cabecera (`.h`), que mediante directivas de compilador `#define`, se realiza una traducción del nombre complejo en que aparecen estas variables globales en los ficheros objeto compilados con Fortran. Para ser genérico, conviene incluir directivas compiladores de compilador que sean capaces de detectar los posibles compiladores que se van a usar, y según eso, utilizar la traducción adecuada. A continuación se expone un ejemplo de cómo difernciar entre compiladores para direccionar correctamente el símbolo de una variable global exportado desde código Fortran:

Listado 4.1: Compilación condicional mediante detección del compilador

```
#if defined(__ICC) || defined(__INTEL_COMPILER)
/* Intel ICC/ICPC. ------------------------------------------ */
#define L_TWO_NOISES  __langevin_params_mp_l_two_noises
#define L_DE_LOR      __langevin_params_mp_l_de_lor
...
#elif (defined(__GNUC__) || defined(__GNUG__)) && !(defined(__clang__) || defined(
    __INTEL_COMPILER))
/* GNU GCC/G++. --------------------------------------------- */
#define L_TWO_NOISES  __langevin_params_MOD_l_two_noises
#define L_DE_LOR      __langevin_params_MOD_l_de_lor
...
#endif
```

En `http://nadeausoftware.com/articles/2012/10/c_c_tip_how_detect_compiler_name_and_version_using_compiler_predefined_macros` se muestra cómo detectar más familias de compilador, y con ello poder ampliar este ejemplo a una mayor variedad de compiladores de C.

Diferencias en los prototipos de función entre Fortran y C

A pesar de que Fortran no está totalmente estandarizado, especialmente en lo referente al *name mangling*, los compiladores Fortran `gfortran` de GNU e `ifort` de Intel comparten las siguientes características particulares en la creación de los símbolos de las subrutinas y funciones:

- Las funciones y subrutinas en el fichero objeto están siempre con minúsculas
- Además, se les añade a su nombre el carácter de subrayado (también conocido por "guión bajo") '_'.

Resumen comparativo de las diferentes nomenclaturas según lenguaje y familia de compilador

Como referencia y resumen final, en la Tabla 4.1 se ejemplifican algunos casos:

Tipo de símbolo	Tipo `nm`	Nombre declarado en Fortran	Símbolo creado por `gfortran` (GNU)	Símbolo creado por `ifort` (Intel)
Función propia	T	ADVANCE	advance_	advance_
Función externa (indefinida)	U	RHS_COEFFS_	rhs_coeffs_	rhs_coeffs_
Variable propia inicializada	D	MYDT	__advance_MOD_mydt	__advance_mp_mydt
Variable propia no inicializada	B	MY_ITER	__advance_MOD_my_iter	__advance_mp_my_iter
Variable externa (indefinida)	U	DT	__orb_params_MOD_dt	__orb_params_mp_dt

Tabla 4.1: Ejemplos del *name mangling* de los compiladores Fortran usados

En la tabla anterior, se muestran los símbolos que se crean en el fichero objeto producto de la compilación de un código fuente de un supuesto fichero `ADVANCE.F` que define la subrutina `ADVANCE()`. Se supone que esta subrutina realiza una llamada a la función `RHS_COEFFS` que está declarada y definida en un fichero externo. En la subrutina `ADVANCE` se definen dos variables, `MYDT` y `MY_ITER`, las cuales son globales para todo el programa Fortran, una de ellas inicializada y otra no, y utiliza la variable externa `DT` declarada en el fichero `ORB_PARAMS`.

Más información sobre el significado y tipos declarados por la utilidad GNU `nm` se puede obtener en `http://www.freebsd.org/cgi/man.cgi?query=nm&apropos=0&sektion=0&manpath=FreeBSD+9.0-RELEASE&arch=default&format=html`

4.2.2 Esquema general de llamadas a CUDA desde Fortran a través de *wrappers* C

El método elegido para adaptar el programa en Fortran 90 a CUDA consiste en la creación de una función *wrapper* en C, invocada de forma normal desde Fortran, que recoge los parámetros y accede a las variables globales declaradas en Fortran necesarias para pasar al *kernel*. A partir de ese momento, todo el código puede hacerse en C o en C++, dado que se compilará con el compilador `nvcc` proporcionado por CUDA. Se deberán tener en cuenta las cuestiones de nomenclatura y de paso de parámetros ya expuestas en el anterior apartado.

En general el proceso será:

- La función *wrapper* C, deberá definirse siempre con los calificadores `extern "C" void`, dado que una subrutina en Fortran nunca devuelve valor. El `extern "C"` permite la compatibilidad con la nomeclatura de funciones de C desde C++, puesto que `nvcc`en realidad es un envoltorio sofisticado del compilador C++ presente.
- En C la función deberá terminar siempre con el carácter de subrayado adicional (también conocido como guión bajo) `'_'`, y con todos sus caracteres en minúsculas. Todos sus parámetros serán punteros, y es recomendable que aquellos que no deban ser modificados por esa función o sus derivadas (parámetro sólo de entrada), lleven el calificador `const`.
- Desde Fortran se invocará a la función por el nombre definido en C, con cualquier combinación de mayúsculas y minúsculas, pero obligatoriamente sin el último carácter de subrayado. Además, debe invocarse como se hace a una subrutina de Fortran (con `CALL <nombre_subrutina(lista_parámetros)`), y no como se hace a una función de C.
- En la función *wrapper* C se gestionarán los parámetros de entrada, y sobre todo, se recomienda allí utilizar macros para renombrar las variables globales de Fortran, por ser tremendamente farragosas de escribir tal cual desde C. A partir de allí, con esos sinónimos en C se operará normalmente, siempre teniendo en cuenta que se trata de punteros. Extremar la precaución con los parámetros que sean matrices multidimensionales por la cantidad de indirecciones que serán necesarias para su correcta referencia al valor, o a la dirección (puntero donde se encuentre el valor real).
- En el *wrapper* C se gestionarán las transferencias de datos que sean necesarias entre la CPU y la GPU. Se recomienda, por eficiencia, evitar en todo lo posible la creación de nuevas zonas de memoria para cambiar el orden de acceso de las matrices.
- Desde el *wrapper* C se invocará ya de forma normal a los *kernels*, siguiendo las particularidades del lenguaje CUDA-C, en especial, de la invocación a dichos *kernels*.

- Dentro de los *kernels*, donde se utiliza el lenguaje CUDA-C, se deben tener en cuenta, aparte de todo lo característico de ese lenguaje extensión de C, la procedencia de las zonas de memoria multidimensionales (*arrays*). Si son en C deben accederse de la forma habitual (orden por filas), pero si provienen (fueron definidas) en Fortran, entonces el acceso debe realizarse teniendo en cuenta que el almacenamiento es por columnas, esto es, el segundo elemento en la zona de memoria no es el elemento `A(1,2)`, sino el elemento `A(2,1)`. Es por ello que se recomienda el uso de macros en C para direccionar siempre con ellas las zonas de memoria definidas en Fortran.

  A continuación se muestran las macros utilizadas en este proyecto para realizar tal acceso, en función de si se pretende acceder al valor, o a la dirección de memoria donde está almacenado dicho valor. Estas macros deberán definirse para cada tipo de *array* multidimensional. Las que se presentan son para matrices bidimensionales de números flotantes de doble precisión (`**double` o `double[][]`).

Listado 4.2: Macros en C para el acceso a matrices 2D definidas en Fortran

```
1  // Convert linear vector representation to 2D matrix in C-like file/column order
2  // Use: MATRIX2D_DBL_PTR(pointer_to_matrix_start, file, column, file_width)
3  #define MATRIX2D_DBL_PTR(m,i,j,N)   (((m))+((j)*(N)+(i)))  // A pointer to value
4  #define MATRIX2D_DBL_VAL(m,i,j,N) (*(((m))+((j)*(N)+(i)))) // The value itsef
```

Siguiendo estas indicaciones, supongamos una función denominada `ADVANCE`, que está declarada en el código original en Fortran, y que al realizar gran computación numérica la vamos a sustituir por un *kernel*. Sea `PARAM1` sea un parámetro de entrada, y `PARAM2` otro de salida. Su definición en Fortran será:

Listado 4.3: Definición de una subrutina común en Fortran 90

```
1          SUBROUTINE ADVANCE(PARAM1, PARAM2)
2          INTEGER, INTENT(IN) :: PARAM1
3          REAL*8,  INTENT(OUT):: PARAM2
4          ...
5          END SUBROUTINE ADVANCE
```

Para sustituir esa función por un *wrapper* en C, su declaración y contenido básico será la siguiente:

Listado 4.4: Definición de la función *wrapper* en C

```
1  extern "C" void kernel_advance_wrapper_(const int *param1, double *param2) {
2  ...
3    // Copy data from CPU to GPU, if apply
4    kernel_advance<<<GridSize,BlockSize(param1, ...);
5    // Copy data from GPU to CPU, if apply
6  ...
7    return; // It's void. So we have to return nothing!
8  }
```

Finalmente, esta función *wrapper* en C se invoca desde Fortran de la siguiente forma:

Listado 4.5: Ejemplo de llamada a un *wrapper* C desde Fortran

```
1          CALL KERNEL_ADVANCE_WRAPPER(PARAM1, PARAM2)
```

4.3 Diferencias programáticas importantes entre Fortran y C y CUDA-C

En las secciones anteriores ya se han comentado aspectos diferenciadores entre C y Fortran, pero que tenían más que ver con la sintaxis. En esta sección se hará especial hincapié en las principales diferencias semánticas y programáticas que deben considerarse en la paralelización con CUDA-C de programas Fortran, que son las siguientes:

Paso de parámetros: Por valor frente a por referencia

En C el paso de parámetros es por valor siempre (para pasar por referencia, en realidad se hace un pase por valor, pero del puntero a la zona de memoria referenciada), en Fortran el paso es siempre por referencia. En un apartado posterior se analizarán las consecuencias de este hecho.

De esta forma, en Fortran no se puede incluir una constante como parámetro, y por otra, desde el *wrapper* en C, al recibir un parámetro siempre será un puntero, y como tal, estará expuesto a ser modificado o eliminado desde C. Es por ello que se recomienda que sea declarado como puntero constante (`const`) en el prototipo en C.

Variables globales Fortran dentro de una subrutina llamada por diferentes hilos OpenMP

Las subrutinas Fortran, cuando son llamadas desde diferentes hilos de ejecución, no comparten las variables globales definidas en Fortran. Están disponibles, pero cada hilo tiene su propia versión local, sin que se copie el valor global cuando ésta tenía al ser llamada.

Para solucionar este problema, se propone el uso de parámetros adicionales en el prototipo de la subrutina Fortran. Así, por ejemplo, si se precisa tener acceso a la variable global `thread` del programa Fortran, se debe añadir un nuevo parámetro, con el modificador `INTENT` adecuado (`"IN"`, o `"INOUT"`), para que en la subrutina se utilice esta nueva variable local con el valor que tenía su correspondiente variable global en el programa principal en el momento en que fue llamada. De lo contrario se inicializa a cero.

Este problema afecta también a las matrices y variables definidas como `ALLOCATABLE`.

Diferente orden de almacenamiento en memoria de matrices multidimensionales

Ya ha sido comentado en el apartado anterior, pero es tan importante que merece un nuevo recordatorio. En C el acceso a las zonas de memoria multidimensionales se define y realiza por filas primero, por columnas después (si es multidimensional, de izquierda a derecha en el orden de las dimensiones tal cual se escriben). En Fortran, para matrices 2D, es por columnas primero y después por filas. Si son multidimensionales, de derecha a izquierda en el orden que aparecen las dimensiones o índices.

Diferente inicio del índice de vectores y matrices

En C, el índice de inicio siempre es 0, tanto en filas como en columnas. No es lo mismo en Fortran, en donde el índice del primer elemento normalmente empieza por uno (véase la excepción en la siguiente característica). Eso es muy importante a la hora de reutilizar el código en Fortran como base para su traducción a C: los bucles e índices deben revisarse convenientemente para que accedan correctamente al elemento correspondiente en cada momento.

Matrices multidimensionales con índices que no empiezan en cero

En Fortran es posible definir matrices en las que una (o varias) dimensiones no empiecen en 1 como normalmente se hace. Puede definirse que el índice de una determinada dimensión tenga un intervalo cualquiera, como por ejemplo, entre -3 y +6, en lugar de 1 a 10. En C los índices siempre serán, en ese ejemplo, entre 0 y 9, por lo que a la hora de migrar el código de Fortran a C deberán contem-

plarse estas eventualidades para que el acceso sea siempre al elemento correcto (aparte de que se podría, además, estar accediendo a zona de memoria fuera de la matriz, con riesgo de corrupción de la misma).

Compatibilidad entre los diferentes tipos numéricos Es muy importante tener en cuenta realmente cuál es el espacio justo de almacenamiento de las variables en C según su declaración en Fortran, donde se pueden definir, por ejemplo, números reales de X decimales e Y cifras enteras. Esto da lugar a tipos de variables que deben analizarse para asegurar que en C se le asigna un tipo compatible en signo, en capacidad (4, 8, 16 Bytes) y en precisión. Especial atención a los tipos inexistentes en C, como los `LOGICAL` de Fortran.

Otras características menores

No merece la pena entrar en detalle de la multitud de pequeños detalles de divergencia entre estos lenguajes. Como mero ejemplo, en este proyecto en algún momento se pensó en la conveniencia de utilizar un `enumerate` de C, pero se observó que en Fortran no existen las denominadas "constantes nombradas". Por lo tanto, se tiene que trabajar con variables normales directamente, con valores enteros.

Por ello se precisa un experto conocedor de ambos lenguajes (Fortran, C y CUDA-C) para realizar una correcta y óptima migración o adaptación del código histórico en Fortran a CUDA.

Como corolario, es tremendamente importante tener presente en todo momento de dónde proceden las matrices, puesto que según hayan sido definidas en C o en Fortran, su acceso y tipología son diferentes. Además, es muy importante recordar que según en qué lenguaje estemos programando, el acceso a las matrices y su definición, son diferentes.

4.4 Implementación en CUDA del código de Langevin originalmente en Fortran

En esta sección se van a comentar brevemente las principales características de la implementación realizada resultante de la migración parcial del código de Langevin en Fortran a CUDA. En una sección posterior se comentará la modificación que a ésta primera implementación en CUDA debe realizarse para obtener una mejora de rendimiento por el desacoplamiento entre computación e intercambio de mensajes y escritura en disco.

4.4.1 Aplanamiento de subrutinas por inclusión en un *kernel* aglutinador

En CUDA la invocación a los *kernels*, y los cambios de contextos internos entre los diferentes hilos de la GPU son menos costosos que en los lenguajes tradicionales de C y Fortran. Por otro lado, es muy importante el número de registros y otros recursos, como las unidades lógicas de coma flotante (simple y doble, pues son independientes) no se saturen, porque eso disminuye la cantidad de hilos que pueden estar en ejecución simultáneamente en el mismo núcleo interno (bien sea en estado activo, con actualización constante del contador de programa, bien en estado pasivo, por estar a la espera de la finalización de alguna operación larga, como las aritméticas de doble precisión).

Consiguientemente, es muy factible en CUDA obtener buenos resultados incluso aunque se utilicen muchos *kernels* simples, siempre que éstos precisen pocos recursos, pero el número total de recursos queden bien saturados, de forma lo más homogénea posible, por la invocación y ejecución en cada núcleo del

número de bloques óptimo (o un poco más, nunca menos), como se demostrará en el siguiente Capítulo 5 "Valoración cualitativa del proceso de paralelización con CUDA".

A pesar de ello, y como demuestra este trabajo, es posible también contar con un buen rendimiento ocupacional aunque el *kernel* sea más bien complejo, si la utilización de los recursos es homogéneo. Tal es el caso que se produce en el `kernel_advance`, que es el principal responsable del tiempo de ejecución global del programa. Este *kernel* se encarga de, para cada partícula, calcular la nueva velocidad vectorial en función de la velocidad anterior, del valor de los campos magnético y eléctrico, y de un componente estocástico (fundamental al ser unas ecuaciones del tipo Langevin) que simula el efecto colisional del resto de partículas (electrones e iones) a su alrededor.

En el código original, tal como se observa en la Figura 5.1, la subrutina `ADVANCE` es responsable del 5,45 % del tiempo de computación, pero ésta llama exclusivamente a otras subrutinas de ayuda. Estas subrutinas de ayuda sirven para calcular algunos parámetros concretos de la fórmula de Langevin, que es la que se computa en la subrutina padre `ADVANCE`, tales como el efecto de la colisión con iones (`RHS_ION`) y el ángulo de la velocidad resultante tras impactar con iones o electrones, (`PHI_AND_PSI`, llamada a través de la otra subrutina `RHS_COEFFS`). También se utilizan subrutinas auxiliares, como `GET_GAUSSIAN_NOISE`, encargada de realimentar el plasma con un electrón estadísticamente similar a la condición inicial por cada electrón fugitivo (*runaway*) que se escapa del confinamiento magnético. Esta subrutina, `GET_GAUSSIAN_NOISE`, a su vez, llama a una subrutina de biblioteca, `RAN2`, que es la que genera los números aleatorios necesarios para el comportamiento estocástico de la ecuación de Langevin. Por lo tanto, en realidad, la subrutina `ADVANCE`, y las dependientes de ella, es responsable del 98,5 % del tiempo de ejecución del código original.

Esta cadena de llamadas de subrutinas no puede realizarse de forma equivalente con *kernels*. No en el sentido de que un *kernel* llame a otro *kernel*. Aunque eso es posible a partir de las tarjetas Kepler, esas invocaciones de un *kernel* por otro, está permitido sólo con un nivel de anidamiento de uno, e implica una nueva creación de muchos más bloques con sus respectivos hilos. Está pensado para permitir, hasta cierto punto, la programación dinámica, un paradigma de programación que no es aplicable a nuestro programa.

Lo que se hace, es invocar a un único *kernel*, el cual puede llamar a otras funciones en CUDA-C que lo que hacen es simplificar la expansión del código completo en una única función *kernel*. Pero en realidad, el *kernel* es único, lo que pasa es que éste llama a funciones, compiladas en el código propio de la GPU (`PTX`).

Por lo tanto, en el código presentado, el `kernel_advance_wrapper_` incluye otras funciones CUDA-C, `rhs_coeffs` y `rhs_ion`, que son la migración de sus correspondientes subrutinas de Fortran. Se observa que "desaparecen" las subrutinas `GET_GAUSSIAN_NOISE` y `RAN2`. Eso es debido a que las tarjetas NVIDIA, a través de CUDA tienen sus propios (y variados) generadores de números aleatorios, muy probablemente más eficaces y seguros que cualquier otro método que se quisiera implementar directamente con CUDA-C en la GPU.

4.4.2 Minimización de la transferencia de datos entre CPU y GPU

Las implementaciones presentadas han minimizado las transferencias de datos en ambos sentidos entre la CPU y la GPU.

Únicamente se transmiten los siguientes, que son inevitables:

- Vectores de velocidades iniciales cuando éstos son leídos de un fichero de reinicio (ficheros con el nombre `langevin.restart.????`): Evidentemente, pues la GPU no puede acceder directamente al disco duro. Nótese que si el programa debe generar las velocidades iniciales, según los parámetros indicados en el fichero de configuración (distribución uniforme o gaussiana), éstas se generan ya

directamente en la GPU, sin intervención de la CPU, quien desconcerá hasta la finalización de la ejecución el valor concreto de la velocidad de cada partícula del plasma.

- Vectores de velocidades finales: Es necesario pasarlos de la GPU a la CPU al final de la computación, por la misma razón: la GPU no puede guardar esos datos en ficheros `langevin.restart.????` en disco duro.
- Valores estadísticos intermedios: La propia GPU es la encargada de catalogar la velocidad de cada partícula en su correspondiente casillero para finalmente enviar a la CPU sólo los datos estadísticos ya elaborados que periódicamente, si así se ha indicado, deben grabarse en disco o mostrarse por consola. Estas estadísticas pueden ser de diferentes valores y características, pero siempre son computadas por la GPU y transmitidos los resultados estadísticos elaborados a la CPU.
- La contabilización del número de electrones fugitivos generados: Que en realidad es redundante con lo anterior, puesto que es otra estadística más.

Con ello se consigue eliminar la necesidad de intercambiarse la información sobre las velocidades de las partículas entre la GPU y la CPU. La versión que definitivamente eliminó esa necesidad proporcionó una aceleración interna entre versiones de prácticamente el 100 %. Esto es, el intercambio en cada iteración temporal de los vectores de velocidad suponía un coste semejante al coste computacional de los *kernels*.

4.4.3 Utilización adecuada de los generadores de números aleatorios de CUDA

CUDA, mediante la biblioteca estándar CURAND, proporciona un muy variado y eficiente conjunto de generadores de números aleatorios. Tal es su variedad, que se pueden seleccionar diferentes métodos de obtención de números, que se basan en diferentes métodos, con ventajas y desventajas respecto de la agrupación, seguridad, distribución y velocidad de generación de los números aleatorios. En este trabajo no se ha entrado a valorar las diferentes técnicas base, y se han seleccionado los que parecían más usualmente usados en la bibliografía, en tutoriales y otros programas.

Además, las funciones que aporta, permiten la generación directamente de números con distribución uniforme (la más usualmente implementada de forma estándar en los demás lenguajes), pero también de números con distribución normal (también conocida como distribución gaussiana). Es más, incluso es posible generar los números aleatorios por pares, con una mejora sobre un 25 % respecto a generar dos números aleatorios de forma individual. Ambas características han sido utilizadas en estas implementaciones, permitiendo que fuera la propia GPU la encargada de sintetizar los términos estocásticos directamente dentro del *kernel*.

Otro aspecto de rendimiento muy importante a tener en cuenta es que los números aleatorios deben ser inicializados. Esta inicialización no debe realizarse en cada invocación del *kernel*, sino que si las variables de los generadores aleatorios son guardados en la zona de almacenamiento global de la GPU, se asegura su permanencia durante toda la ejecución de la aplicación, aunque el *kernel* concreto que los haya inicializado o usado por última vez finalice. De esta forma, es posible tener los generadores ya inicializados para cada partícula, e invocar la petición de nuevos números aleatorios aprovechando las semillas existentes de una invocación a otra del *kernel*.

Cuando este fenónemo se descubrió, se obtuvo una aceleración relativa del 35 % en el tiempo global de ejecución del programa global, entre las dos versiones implicadas. Esto significa que aunque los números aleatorios pueden ser rápida y eficientemente generados en la GPU, no es así su proceso de inicialización. Es por ello que es conveniente inicializarlos una única vez durante toda la ejecución de la aplicación, y reusarlos siempre que sea posible.

Otra mejora fue la de asegurarse que estos generadores están en memoria compartida en el momento de la ejecución del *kernel*. Esto es, en general están en memoria global, pero cuando son necesarios se invocan varias veces. Para asegurarnos de que únicamente se bajan y suben de la memoria global a la más cercana a los núcleos CUDA una vez (al inicio y al final de cada invocación al *kernel*) y no depender de que estén o no todavía presentes en la *cache*, se hace una copia de ellas en memoria *shared* y después, al terminar el *kernel*, se sube el estado del generador aleatorio de nuevo a la memoria global. Esto es posible porque tenemos un generador por cada una de las partículas.

No obstante, hay que señalar un problema que se produjo con los números aleatorios: en determinadas simulaciones, es necesario contar con números aleatorios distribuidos uniformemente para unas cosas, pero otros distribuidos de forma normal (gaussiana) para otras. En esos casos, no puede utilizarse la misma variable generadora, siendo necesario crear generadores diferentes para distribuciones diferentes. Esto implica un sobrecoste, pero se realiza una única vez durante el programa.

Como idea de optimización se deja la posibilidad de probar diferentes generadores, para ver si su inicialización y constante generación es más rápida en unos que en otros. Nótese que nuestro programa, en principio, no requiere de especiales condiciones de extrema seguridad en cuanto a la calidad de los números aleatorios generados.

Otra idea de optimización consiste en seguir los consejos que en diversos foros aparecen, consistentes en que se inicializa sólo una variable generadora de números aleatorios, o un número reducido de ellos, por ejemplo, una por cada bloque. Posteriormente, cuando se solicitan números aleatorios, en lugar de pedir el siguiente, se pide el segundo, tercero o cuarto, etc. según el número de hilo. Dicen que si se hace avanzar la cuenta, es posible que no existan colisiones de números generados. No obstante, hay críticas a estos métodos por cuanto que su utilización implica la pérdida de la seguridad teórica de los algoritmos base de la implementación de estos números aleatorios. Por lo tanto, los críticos indican que los números obtenidos pueden estar sesgados, o tener ciertas agrupaciones locales indeseadas. Es un tema abierto a la experimentación, tanto para ver si acelera realmente las actuales implemenaciones, y también sobre si los resultados finales se muestran estadísticamente semejantes o diferentes a los obtenidos con la actual implementación, más conservadora y posiblemente más lenta, pero segura.

4.4.4 Mejoras en el código: utilización de funciones estándar frente a métodos numéricos pesados

En el código original, la subrutina `advance` debe implementar una determinada integral. En el código original, la implementación es tal cual, esto es, se realiza una integración numérica usando el método trapezoidal con unos 200 pasos. Esto computacionalmente es costoso. Si se analiza la integral, y como en algunos artículos puramente científicos donde se desarrollan estas ecuaciones de Langevin para plasma, esa integral ha dado lugar a la definición como tal de una función ampliamente conocida: la función error.

La sorpresa está en que esta función es tan frecuente, que prácticamente todos los lenguajes computacionales (esto es, que mínimamente sirvan o se usen para cálculos matemáticos), tienen prácticamente de serie, alguna función propia o en una biblioteca numérica estándar, implementada esta función. Tal como incluso se ha comprobado experimentalmente en este proyecto, la llamada a esta función es mucho más precisa, exacta y rápida que el método de integración implementado en Fortran dentro del código original. Es más, por ejemplo, en C, el error de la función está asegurada en ser como máximo dos unidades en la última posición representativa del número flotante de doble precisión resultante, siendo de una como máximo en la mayor parte del intervalo razonable y normal de uso. Es evidente que doscientas sumas y multiplicaciones para un cálculo numérico aproximado va a tener una exactitud de varios órdenes mucho peor.

La función error en C y en CUDA-C tiene el mismo nombre, y se trata de `erf()`, la cual se ha utilizado en el `kernel_advance`, con mejora de prestaciones. Esta mejora ha sido del 1–2 % en el momento de realizar el cambio.

4.4.5 Algoritmos de reducción en la GPU

El secreto evidente de la buena aceleración que se obtiene en los resultados de las implementaciones presentadas en este trabajo, es la absoluta independencia de los cálculos entre cada una de las partículas simuladas. Por ello, no existe dependencia de datos en la GPU y prácticamente se asegura que todos los accesos a memoria son coalescentes, propiedad que mejora las prestaciones computacionales de la tarjeta gráfica, al mejorar el trasiego de información interno entre la memoria y los núcleos de la GPU.

No obstante, existe un par de sitios en el algoritmo, en donde la GPU se ve forzada a realizar una reducción. Esto es, debe obtener un resultado a partir de los valores calculados por el resto de hilos y bloques, esto es, de todas las partículas:

- Cálculos estadísticos para obtener estadísticas sobre el valor de las componentes de la velocidad de los electrones (perpendicular y paralela al campo magnético): caracterizado porque se deben siempre tener en consideración el valor de todas las partículas (por lo tanto, de todos los hilos participantes en el *kernel*).
- Recuento del número de electrones que en cada iteración pasan a ser electrones fugitivos (*runaway*), por exceder la velocidad crítica de confinamiento: se caracteriza porque, en condiciones normales del plasma, se trata de una baja proporción de electrones (hilos) los que deben computarse.

Estas reducciones se pueden abordar al menos desde dos perspectivas diferentes:

- Mediante algoritmos propios de reducción, generalmente con coste computacional de orden $\mathcal{O}(\texttt{log}_2\texttt{(N)})$: Son los indicados para cuando el porcentaje de hilos (partículas) a participar en la reducción es elevada respecto al total.
- Mediante operaciones atómicas que cada partícula realiza sobre una variable global común: Son los indicados cuando el porcentaje de hilos (partículas) a participar es despreciable frente al conjunto total de partículas.

La elección del método es muy importante por su impacto en el tiempo de computación. Esta decisión es un claro ejemplo de las dificultades que suelen tener los programadores que no son expertos en ingeniería informática, dado que frecuentemente se obvia la realización de un estudio previo (o aunque sea posterior, experimental) de costes del algoritmo a implementar.

La elección para cada uno de los dos casos es diferente, y se basa en un estudio de costes:

Estadística de velocidades Cada hilo se ubica en su contenedor según sus componentes de velocidad perpendicular y paralela, pero hay que ver cuántas partículas hay en cada contenedor. Por lo tanto es una reducción en las que siempre participan todas las partículas. Es por ello que se requiere un algoritmo de reducción basado en reducción binaria en árbol, que tiene un coste de orden $\mathcal{O}(log_2(N))$-

Conteo de electrones *runaway* En condiciones normales de plasma, el número de electrones fugitivos es muy pequeño. Si fuera grande, el tokamak estaría descontrolado y la reacción nuclear de fusión se pararía en pocos milisegundos. Por lo tanto, no tiene sentido estar simulando en esas condiciones. Consiguientemente, interesa el algoritmo basado en operaciones atómicas sobre una misma variable,

pues en el caso extremo de que coincidan todas las operaciones de adición en el mismo momento, éstas se serializarían, y el coste sería de orden lineal, $\mathcal{O}(log_2(n))$, con n siendo el número de colisiones.

Analicemos cada una de este tipo de reducción por separado.

4.4.5.1 Reducción global iterativa

Esta reducción tiene sus triquiñuelas cuando se tiene que realizar en GPU, sobre todo cuando la capacidad computacional CUDA de la tarjeta usada es más baja. Esto es así, porque éstas manejan mucho peor los accesos no coalescentes a la memoria. Para resolver esta eventualidad, es importante planificar estos accesos para que sean lo más coalescentes posibles, y siempre bajo el prisma de que será necesario realizar una entrada recursiva (en realidad, iterativa) de orden $\mathcal{O}(log_2(N))$, siendo N el número de partículas, para su final reducción.

En el SDK de CUDA aparece un método terriblemente eficiente que utiliza unas macros de compilador para obtener la máxima potencia de la tarjeta gráfica, independientemente de su capacidad computacional, al optimizar mucho el acceso coalescente a la memoria. No obstante, es un método muy rebuscado como para poder reutilizarlo. Es por ello la implementación se ha realizado utilizando la técnica del árbol binario, con coste $\mathcal{O}(log_2(m))$, siendo m el número de hilos en cada bloque (32, o sea, 6 iteraciones), para posteriormente, hacer otra pasada y volver a totalizar esas sumas parciales (total de bloques dividido por 32), una por cada bloque lanzado.

Como ejemplo, se puede acudir al código fuente de la implementación, al fichero `kernel_calculate_statistics.cu` concretamente a los *kernels* `kernel_calculate_statistics_ratios` y `kernel_calculate_statistics_ratios2D`, que son demasiado extensos para su inclusión en esta memoria.

4.4.5.2 Reducción mediante el uso de operaciones atómicas

CUDA provee de operaciones atómicas sobre variables globales. El único problema con las operaciones atómicas en la GPU es que hasta el momento (hasta CUDA versión 7.0, capacidades computacionales hasta 5.2), no está permitido realizar una reducción sobre variables de longitud de 64 bits. Esto es, no se puede hacer un `atomicAdd` sobre un entero largo (`long int`) o sobre un número en coma flotante de doble precisión (`double`). Como en nuestro caso, el conteo se realiza con enteros normales, de 4 octetos, es perfectamente viable, y recomendable desde el punto de vista de teoría de coste computacional.

Simplemente consiste en seleccionar sólo aquellos hilos (partículas) que deban contabilizarse, y con ello, invocar directamente a la función `atomicAdd`. Veamos como ejemplo, la forma en que se utiliza en el *kernel* que recuenta los electrones con velocidad mayor que la crítica, considerados por ello en la simulación como electrones *runaway*. El Código 4.6 es un ejemplo de su utilización.

Listado 4.6: Ejemplo de *kernel* utilizando una reducción basada en atomicAdd

```
// Main kernel function
// Counts the runaway electrons, which have its velocity greater than the critical
__global__ void kernel_runprob2d(
                double *vz, double vcritz,        // part Z vel.component & vel.critic
                int N_d, int sqrtN_d, int nbins2_d, // running size
                int *ncont2d_runw_d, const int *pos2d_d// output vector 2D position
                ) {
  unsigned int col = (blockIdx.x * blockDim.x) + threadIdx.x;
  unsigned int row = (blockIdx.y * blockDim.y) + threadIdx.y;
  unsigned int idx = col + row * sqrtN_d;
  if (idx>=N_d || col>=sqrtN_d || row>=sqrtN_d) return;

  // NOTE: LOOK OUT!! In C, the row/col indexes are in inverse order than in Fortran!!
  if (vz[idx+N_d] >= vcritz) {
    atomicAdd(ncont2d_runw_d+pos2d_d[idx],1);
  }
} // end kernel_runprob2d()
```

4.4.6 Mejora en el algoritmo de clasificación de velocidades

Muchas de las estadísticas que se realizan en los pasos intermedios del programa consisten en la clasificación de las velocidades de las partículas según sus intervalos. Concretamente, se establece la velocidad mínima y máxima, que son las que si se superan, el electrón se convierte en un electrón fugitivo, y se divide ese intervalo en 50 intervalos iguales denominados canastas. La estadística consiste en contar cuantos electrones están en cada canasta. Esta estadística puede ser unidimensional sobre la celeridad (módulo del vector velocidad), o en dos dimensiones, creando una matriz bidimensional de canastas, para clasificar según la velocidad paralela al eje Z, o perpendicular a él (plano XY).

El método de clasificación implementado en el código original es terriblemente costoso, para lo simple del trabajo. Consiste en ordenar primero con orden total, todas las N partículas. Una vez ordenadas en un vector auxiliar creado y destruido posteriormente a su efecto, se recorre dicho vector auxiliar en un bucle y se cuentan cuántas partículas están dentro del intervalo de cada canasta.

Por lo tanto, el coste del algoritmo implementado en el código original es:

- Coste asintótico temporal = $\mathcal{O}(N \cdot log(N))$

  Por el orden total, asumiendo que los métodos de ordenación utilizados `HPSORT` y `HPSORT2` sean óptimos. A éste hay que añadir otro coste temporal de $\mathcal{O}(N)$ por recorrer posteriormente el nuevo vector ordenado previamente (no se ha añadido por estar hablando de coste asintótico).

- Coste asintótico espacial = $\mathcal{O}(N)$

  Por la necesidad de utilizar un vector auxiliar para almacenar el vector ordenado.

En lugar de este método se propone simplemente la utilización de las siguientes fórmulas, dependiendo de si la clasificación es unidimensional o bidimensional:

Listado 4.7: Clasificación óptima para canastas unidimensionales

```
    // Unidimensional clasification:
    //   idx         : particle index
    //   vx[]        : component X of velocity
    //   vy[]        : component Y of velocity
    //   vz[]        : parallel velocity (component Z of velocity)
    //   vcrit       : critical velocity
    //   nbins       : number of bins
    //   pos2d_d[]   : pointer to the 2D array where the counting data is stored
    // Intermediate calculation explanation:
    //   v=sqrt(vx*vx+vy*vy+vz*vz) : celerity, modulus of the velocity vector XYZ
    v = sqrt(vx[idx]*vx[idx] + vy[idx]*vy[idx] + vz[idx]*vz[idx]);
    pos2d[idx] = (int) floor(v/vcrit*nbins);
```

Listado 4.8: Clasificación óptima para canastas bidimensionales

```
    // Bidimensional clasification:
    //   idx         : particle index
    //   vx[]        : component X of velocity
    //   vy[]        : component Y of velocity
    //   vz[]        : parallel velocity (component Z of velocity)
    //   vz0min_d    : minimum parallel velocity
    //   ivparohs_d  : width of the parallel bin (bin1)
    //   ivperpohs_d : width of the perpendicular bin (bin2)
    //   pos2d_d[]   : pointer to the 2D array where the counting data is stored
    // Intermediate calculation explanation:
    //   sqrt(vx*vx+vy*vy) : perpendicular velocity (module of velocity in plane XY)
    int bin1 = (int) floor((vz[idx]-vz0min_d)*ivparohs_d);
    int bin2 = (int) floor((sqrt(vx[idx]*vx[idx]+vy[idx]*vy[idx])*ivperpohs_d));
    pos2d_d[idx] = bin2*nbins2_d+bin1;      // Fortran: add +1 to the vector indexes
```

Como se observa, el coste asintótico temporal es 1, y el espacial 0, para cada partícula. Dado que no se ejecuta un bucle por la paralelización masiva al utilizar todos los núcleos del multiprocesador, es por lo que desaparece el bucle y el coste asintótico global es $\mathcal{O}(1)$. El coste real es del orden $\mathcal{O}(\frac{N}{p})$, donde N es el número de partículas y p el número de núcleos que tiene (o se están utilizando de) la GPU.

La corrección del nuevo método queda constratado experimentalmente por comparación de resultados entre el código original y el nuevo *kernel* que implementa esta simple clasificación en orden asintótico $\mathcal{O}(1)$. Por lo tanto se reduce el coste desde $\mathcal{O}(N + N \cdot log(N))$ temporal a $\mathcal{O}(1)$.

4.4.7 Detección de errores (*bugs*) en el código

Durante la paralelización con CUDA del código original, y conforme se conocía mejor tanto el programa como las ecuaciones en él simuladas, así como su significado físico, se llegó a la detección de lo que se consideran dos errores (o *bugs*) del programa. Los dos errores son errores durante la programación en Fortran. Esos errores fueron comunicados al científico suministrador del código, pero hasta el momento no ha respondido al respecto.

No se va a entrar en detalles, pero se comenta sucintamente la naturaleza de los errores, para servir de ejemplo lo importante que son los aspectos que se señalarán más adelante en la valoración del proyecto: la formación específica en programación, y mejor en ingeniería de software, del programador a cargo del código, y de la conveniencia, sino necesidad, de contar con un experto científico que supervise la labor de dicho programador.

- Declaración como `INTEGER` de una variable estadística, que recoge el la velocidad perpendicular en la función de distribución *runaway*. En realidad cuando su valor es la resultante de cálculo de coma flotante de doble precisión. Se producen dos problemas acumulados:

  – Al ser declarada como `INTEGER` sus valores se redondean al entero más cercano. Sus valores varían de 0 a 13, por lo que al redondear al número más entero más bajo, se comete un error relativo enorme, de entre el 8 % y el 100 %.

  – A pesar de declararse como `INTEGER`, en MPI se transfiere como `MPI_DOUBLE_PRECISION`. Por lo tanto, se desvirtua totalmente el valor en él contenido, la mitad superior de los elementos tienen un valor indeterminado, y la mitad inferior contienen dos enteros que se interpretan como un número flotante de doble precisión.

  Este error es un mero error de programación, pero debería haberse ya detectado, puesto que los ficheros estadísticos que resultan no tienen significado físico al estar desvirtuado su contenido. Por lo tanto, una buena revisión de los resultados estadísticos por parte de un científico debería haber revelado las consecuencias de este error.

- Durante el cálculo estadístico de la distribución de velocidades en una componente, se utiliza un parámetro previamente calculado para otra componente, que puede tener un valor totalmente diferente según la simulación realizada.

  Este error no es tanto de mera programación, como falta de aplicación de los conceptos básicos de ingeniería de software, concretamente la factorización y la reutilización de código.

  El problema aparece en una serie de 4 estadísticas similares, cada una de las cuales calcula la distribución de las velocidades según su componente X, Y, Z y el módulo de la velocidad. En lugar de crear una nueva subrutina que realizara esta misma tarea en un único conjunto de líneas de código, se realizó la primera implementación sobre la componente X de forma correcta, y después, a continuación de esas líneas, se copiaron hasta 3 veces seguidas, para después modificar cada línea pegada modificando las referencias del eje, esto es, cambiando de X a Y, de X a Z y de X al módulo de la velocidad.

  Sin embargo se cometió el olvido de modificar una variable en una línea de esta última copia. Por ello, las estadísticas del módulo de la velocidad se calculan a partir de un parámetro que no corresponde al módulo de la velocidad, sino solamente a la componente X de ella.

  La solución es la refactorización del código, como se ha realizado en las implementaciones aquí presentadas. Se crea una nueva función en CUDA-C, y se invoca cuatro veces, cambiando en cada caso por los parámetros de invocación a los correspondiente de cada dimensión, o de su módulo. De esta forma el mantenimiento del código es más sencillo, pues hay un único punto donde deben modificarse en el futuro.

  Por lo tanto, este error ejemplifica un aspecto importante a la hora de migrar los códigos, o incluso de rehacerlos desde el principio: la necesidad de contar con un experto en ingeniería del software, y en computación paralela, para evitar errores de diseño como éste, que son fuente posible de errores difíciles de detectar, y dificultan el mantenimiento y comprensibilidad del código.

4.5 Desacople de computación y comunicación usando OpenMP

La versión denominada r24 ya era definitiva en el sentido de que estaba totalmente paralelizada con CUDA (a excepción de una subrutina que sólo se invoca al inicio y al final del programa, y que realiza unos cálculos estadísticos de una forma extraña que no se ha sabido descifrar), y proporcionó unos resultados correctos y tenía unas prestaciones excelentes, con una aceleración cercana a 700, como se mostrará en el Capítulo 5 "Resultados", gracias a todos los detalles acabados de explicar en este mismo capítulo.

No obstante, en las pruebas se observó que el tiempo de comunicación empezaba a ser significativo, y que eso influía en el tiempo global de ejecución. Este problema desaparecía casi totalmente si en las simulaciones no se solicitaban resultados estadísticos intermedios.

La razón es obvia: cada cierto número de iteraciones (normalmente cada 1/20 de las totales a realizar), se pueden realizar unos cálculos estadísticos que dependen de los datos de velocidad de cada partícula. Es más, alguno de estos cálculos va actualizándose iteración a iteración, y depende de los otros. Concretamente es el número de electrone fugitivos perdidos. La dependencia de datos es tal que no es posible utilizar los recursos de asincronía de MPI, tal como `MPI_IREDUCE`.

El análisis de estas limitaciones, y la conveniencia de disminuir el tiempo de comunicaciones, desacoplar computación de estadísticas y escritura en disco duro, lleva a las siguientes posibles soluciones:

- Agrupar las zonas de memoria, para que su transmisión sea en un único mensaje y no en varios. Esto finalmente se descarta por varias razones:
  - Las zonas de memoria afectadas están declaradas y definidas en Fortran, y su cambio requiere la reestruccturación de gran parte del código de Fortran que todavía se utiliza.
  - Fortran optimiza mucho el código porque en la asignación de memoria asegura que no exista solapamiento. El hecho de crear una gran zona de memoria, y después empezar a utilizar punteros de Fortran al interior de esa zona, y recorrera, hará que el compilador desconfíe, y genere código con una mayor sobrecarga por la comprobación de sobreescritura. Puede que incluso sea contraproducente si no se hace cuidadosamente la implementación.
  - Las zonas que pueden juntarse, de forma coherente, sólo se transmiten en la actualidad al principio y al final del programa, con lo que la ganancia será muy marginal.
  - La acción conjunta de todas estas razones hace que no parezca conveniente esta aproximación. No obstante, se deja constancia de ella porque en otros casos puede resultar interesante.
- Búsqueda de soluciones basadas puramente en MPI: Se plantean las siguientes posibilidades:
  - Utilización de primitivas MPI asíncronas, tales como `MPI_IREDUCE`. Estas primitivas de comunicación retornan el control al programa inmediatamente, aunque no haya finalizado la transmisión de datos. Es ideal cuando se tiene la seguridad de que los datos (zonas de memoria) de origen y de destino no van a ser modificadas durante lo que dure la transmisión.

    Este hecho no puede asegurarse en nuestro caso. Es más, se sabe que no es así, pues se comprobó que la transmisión puede tardar hasta 9-10 iteraciones temporales en una carga intermedia. Además, como ya se ha comentado, existe interdependencia de datos en esa iteración de recogida y elaboración de estadística de progreso, de tal forma que en realidad los datos no estarían disponbles cuando fueran necesarios, leyendo los del resumen anterior, no los actualizados.
  - Utilización de primitivas MPI, tales como `MPI_PACK` o `MPI_TYPE_STRUCT`, que permitan compactar las diferentes zonas de memoria con anterioridad a su envío por MPI.

Sin embargo, eso implica seguramente más copias de zona de memoria compartida, pero como veremos, en los entornos de computación de altas prestaciones, la red de comunicación es casi tan eficiente como la propia memoria compartida, sólo peca de más de latencia. Consiguientemente, no está claro que eso mejore finalmente el rendimiento, sino que incluso lo empeore, sobre todo porque por las primeras pruebas realizadas, el programa no parece colapsar la red, y ésta es suficientemente rápida como para obtener poca ganancia con este tipo de soluciones.

– utilización del parámetro MPI `MPI_IN_PLACE` en las reducciones, para intentar disminuir la latencia en el nodo receptor de la operación de reducción.

No obstante, esto parece más una ayuda de programación que una mejora real, dado que en realidad MPI gestiona de forma oculta la creación, llenado y final copia y eliminación de un *buffer* donde recibir realmente los datos. Por lo tanto, no parece ser un mecanismo general de copia cero que acelere la reducción.

- Desacoplar la computación de las obtención de estadísticas y escritura en disco. Para ello existen diferentes alternativas:

  – Utilizar una biblioteca equivalente a `pthreads` en C, pero en Fortran.

  Desafortunadamente, no se encontró ninguna. Existía la posibilidad de crear nueva funciones envoltorio, esta vez en Fortran, para utilizar la biblioteca de *pthreads* desde Fortran. Un ejemplo de ello es la descripción e implementación inicial propuesta en `http://v-ganesh.tripod.com/papers/fthreads.pdf`.

  Otro problema es que también sería interesante utilizar las variables condición, y no está clara en principio cuál es su posible adaptación a Fortran.

  – Utilizar una mezcla de MPI y OpenMP para conseguir dos hilos OpenMP, y que cada uno gestione una tarea (computación en *kernels* y comunicaciones MPI y su posterior escritura en disco).

  Dado que no se pueden crear fácilmente hilos de forma manual en Fortran, pueden utilizarse los generados a partir de OpenMP. El único problema es que más que trabajar con zonas paralelizadas, se trabajará con diferentes regiones en las que se asignarán a cada uno de los hilos.

Finalmente, por descarte, y por no existir nada que haga pensar que haya una inviabilidad técnica que lo impida, la opción elegida es la última, la de utilizar OpenMP para crear hilos con los que intentar solapar lo más posible el cómputo de las transmisiones de mensajes y escritura a disco. En realidad hay dos formas de atacar el problema a partir de aquí, como se comentará en el siguiente apartado.

4.5.1 Desacople usando OpenMP con hilo en espera activa

El primer intento se realiza a un nivel de grano fino, y se denomina versión r26. Parte de la versión r24 (Fortran+MPI+CUDA), comentada en el apartado anterior, a la que se le añade OpenMP. Uno de los hilos se encarga siempre de la computación numérica, especialmente del manejo de la invocación de los *kernels*, y el otro hilo, de las comunicaciones MPI y de la escritura en disco (operaciones `I/O`, *Input/Output*).

Además, se declaran unas variables compartidas por los dos hilos, cada una de las cuales controla un tipo de transmisión de mensaje MPI (las estadísticas de electrones perdidos, las estadísticas unidimensionales de velocidades, las bidimensionales, las orbitales, etc.). Cada variable tiene un conjunto de tres posibles estados: `DONE`, `WORKING` y `MESSAGING`. Se comporta como una máquina de estados con dos hilos moviéndose a través de cada estado, y deben cumplirse ciertas condiciones de seguridad.

La zona de paralelización OpenMP incluye todo el bucle principal de las iteraciones temporales. Este bucle no está declarado como `$OMP PARALLEL FOR`, y por lo tanto, no existe ninguna barrera al final de cada iteración. Internamente del bucle, cada hilo avanza lo que puede, y según el tipo de código que sea, el hilo puede entrar en esa parte o no. Esto es, programáticamente, mediante `IF (thread .EQ. 0)` o su inverso, o su respectivo `ELSE`, el hilo ejecuta o no ese código. De esta forma se separa la computación (CUDA) del resto de operaciones esporádicas más lentas y bloqueantes (MPI e `I/O`).

El esquema es el siguiente:

1. Al inicio la variable está en `DONE`.
2. Empieza la computación, y cada hilo recorre lo que puede del bucle.
3. El hilo 1 (MPI e `I/O`), rápidamente llega a la iteración donde deben realizarse las estadísticas, pues no tiene ningún trabajo asignado en las otras. Pero allí debe detenerse en espera del mucho más lento hilo 0 (o máster, CUDA), que está en labores computacionales.
4. Al llegar a su correspondinte estadística, el hilo 1 se queda bloqueado, pues no puede avanzar hasta que la variable compartida correspondiente a dicha estadística esté en `MESSAGING`.
5. Eventualmente, el hilo 0 llega a la misma iteración y entra en la subrutina estadística.
6. Ese hilo no puede avanzar si la variable compartida no está en el estado `DONE`. Esto es así por si resulta en alguna simulación que el hilo 1 es el más lento (por mensajes muy grandes, muchos nodos, red muy lenta o colapsada, por ejemplo).
7. Entonces, si no está en `DONE`, debe esperarse a que el hilo 1 termine su trabajo.
8. Si está en `DONE` (que era la situación inicial), el hilo 0 genera en la GPU los datos estadísticos correspondientes.
9. Cuando el hilo 0 termina con el cómputo estadístico, copia dichos resultados en unos *buffers* auxiliares al efecto, para preservarlos de ser modificados en una siguiente iteración.
10. Tras ello, cambia el estado a `MESSAGING`, y continua en ejecución, iterando si llega al final de cada paso.
11. El hilo 1 o llega a la subrutina cuando el estado ya está en `MESSAGING`, o estaba en espera activa comprobando continuamente si hay un cambio en él. Cuando observa que está en `MESSAGING`, sabe que el hilo 0 ha preparado datos a enviar o recibir, y guardar en disco, si se trata de la tarea 0.
12. Al terminar su tarea de mensajería y de escritura, si es el caso, cambia de nuevo el estado a `DONE`, para hacer saber al hilo 0 cuando llegue a una nueva iteración estadística, que los datos anteriores han sido correctamente procesados, y que puede sobreescribir sobre ellos.
13. Este bucle termina al acabar las iteraciones temporales a simular, en cuyo momento ambos hilos terminan normalmente.

Este esquema se ha demostrado correcto en la práctica, pero sólo se implementó sobre una de las estadísticas, y la ejecución sólo fue sobre `gpu.dsic`, con dos GPU en local. La mejora fue imperceptible, aunque sí que se comprobó que existía solapamiento de computación y mensajería, puesto que el hilo 0 recorría unas 5–10 iteraciones mientras el hilo 1 estaba transmitiendo y escribiendo en disco, si era el caso. No obstante, se contrastó la existencia de un problema que sugería buscar otra solución.

El problema con todo este esquema es que las detenciones en espera del semáforo, es que se trata de esperas activas, en un bucle vacío, simplemente comprobando constantemente la condición del cambio de estado. Esto es ineficiente energéticamente, pues obliga a que la CPU esté siempre al 100 % de utilización. Además, según el planificador, puede que le esté robando ciclos preciosos al otro hilo que sí que está haciendo tareas provechosas (computando, o transmitiendo o escribiendo en disco).

Se propusieron soluciones alternativas que realmente no parecían llegar a solucionar totalmente el problema:

- Uso de señales: La idea es crear los servicios de la variable condición utilizando los servicios del sistema operativo. Se desconoce cómo usar, lanza y gestionar señales del sistema operativo en Fortran. Por ello, se descarta inmediatamente.
- Dormir el hilo en espera, para que no esté en espera activa, y despertarlo pasado cierto tiempo. El problema está de nuevo en que Fortran no provee de funciones de detención inferiores a un segundo. Se llegó a crear un *wrapper* para invocar a una nueva función en C que permitía realizar esperas desde un microsegundos a varios milisegundos.

  En las pruebas realizadas, no obstante, se comprobó que la granularidad efectiva mínima era de unos 250 µs (0,25 ms), y que este valor era variable según el entorno de ejecución. Una espera de 0,5 ms implica 2.000 esperas así cada segundo. Los tiempos de ejecución indicaban que en ese tiempo, incluso el hilo 0 computacional habría realizado algunas iteraciones. Esto implica que si el hilo computacional, el teóricamente más lento, invocara a esta llamada, se perdería un mínimo de 5 iteraciones, lo cual parecía poco razonable, puesto que a lo mejor el hilo de comunicaciones ya había terminado antes.

  Por estas primeras pruebas, también se descartó esta opción.

Descartadas estas opciones, se pensó en utilizar los cerrojos de OpenMP (`omp_locks`).

4.5.2 Implementación final con cerrojos de OpenMP (`omp_locks`)

La versión r27 final recoge las ideas y la experiencia de la anterior r26, pero la aplica a partir de la r24. Añade a esta versión dos cerrojos, con los cuales se gestiona el acceso a las iteraciones en que se realiza el cómputo estadístico, y las inmediatamente siguientes a ella. Se trata pues de una gestión de grado más grueso, puesto que ya no se pretende gestionar el caso de cada una de las estadísticas por separado, sino que simplemente se actúa sobre toda la iteración, lo cual incluso tiene cierto sentido computacional por la existencia de dependencia de datos entre las diferentes estadísticas de una misma iteración.

Los requisitos son los mismos: se crean permanentemente zonas de memoria auxiliar (*buffers*) para almacenar los resultados estadísticos de una iteración, de tal forma que en la siguiente los datos originales puedan ser sobreescritos sin problema. De paso, toda la gestión de memoria dinámica se pasa a estática para prácticamente todas las variables, dado que se considera también muy costoso el estar constantemente reservando, inicializando y liberando memoria en cada iteración estadística.

Los cerrojos en OpenMP permiten bloquear en una espera no activa al bloque que intenta tomar el cerrojo pero éste ya está tomado por otro hilo. Sólo cuando el otro hilo lo libera, el sistema operativo avisa el hilo bloqueado para que éste pueda coger el cerrojo y seguir su camino. Se trata pues de una gestión propia de OpenMP, y por lo tanto optimizada para la arquitectura subyacente y teóricamente más eficiente que el intento realizado con la versión r26 anteriormente explicada.

Para el caso que nos ocupa, la solución planteada ha sido la utilización de dos cerrojos diferentes, así como dos variables auxiliares de seguimiento de quién es su propietario (es importante, dado que está prohibido liberar un cerrojo que no se posea, ni volver a coger un cerrojo que ya se posee, en ambos casos, la ejecución termina abruptamente con error).

La idea básica es que inicialmente el hilo 0 posee ambos cerrojos, y siempre tendrá al menos uno de ellos. Nunca se le estará permitido liberar los dos cerrojos. Por su parte, el hilo 1 podrá tener o ninguno, o sólo uno de los dos cerrojos. Podrán haber cerrojos libres, sin ningún propietario, que siguiendo las reglas anteriores, y otras por definir, podrán ser tomados por cualquiera de los dos hilos, si se le permite. Dependiendo de qué cerrojo o cerrojos poseea cada hilo, podrá avanzar en su tarea, o deberá quedarse bloqueado en espera de un cambio de situación, esto es, coger uno de los otros cerrojos. Cada vez que se coge el cerrojo, el nuevo propietario anota en la variable auxiliar correspondiente que él es el propietario de ese cerrojo. Cada vez que se libera, esa variable se pone a -1, indicando que no tiene propietario. Así, cualquier hilo puede en cualquier momento saber quién es el propietario del cerrojo, o si éste está libre. Los hilos 0 y 1 tienen asignadas las mismas funciones que las descritas en el caso anterior de r26.

El esquema detallado de funcionamiento es el siguiente:

1. La ejecución empieza con el hilo 0 poseyendo ambos cerrojos, que denominaremos abreviadamente `WORK` y `WAIT`.
2. Al inicio de la iteración de generación de estadísticas, el hilo 0 comprueba por las variables auxiliares si es el propietario de `WORK`. Si no lo es, intenta cogerlo, quedando bloqueado si no lo consigue (llamada a `omp_set_lock()`).
3. Si ya era propietario de `WORK`, entonces intenta coger `WAIT`, de tal forma que si falla, también se queda bloqueado hasta obtenerlo.
4. Sólo cuando el hilo 0 posee ambos cerrojos, continua la ejecución de la iteración de estadísticas. Esto es, precisa tener los dos cerrojos para elaborar estadísticas.
5. No obstante lo anterior, cuando ha obtenido los dos cerrojos, y va a continuar la ejecución, lo primero que hace es liberar el cerrojo de `WAIT`.
6. Eventualmente el hilo 0 termina el ciclo de generación de las estadísticas, que las guarda en las variables auxiliares al efecto, y llega a la siguiente iteración, la de estadísticas+1.
7. Cuando el hilo 0 llega al inicio de la iteración estadísticas+1 (o termina el bucle, que es equivalente), el hilo 0 intenta coger el `WAIT`. Si no puede cogerlo, queda bloqueado hasta conseguirlo. Si lo consigue, o cuando, tras estar bloqueado, lo consigue, es porque el hilo 1 ya lo ha tomado y lo ha devuelto, y por lo tanto, sirve para que el hilo 0 obtenga el reconocimiento del hilo 1 de que éste sabe que el hilo 0 está preparando los nuevos datos.
8. El hilo 0, tras obtener de nuevo el `WAIT`, libera el `WORK`, con lo cual ha respetado en todo momento tener al menos uno de los dos cerrojos en su posesión.
9. Pasemos al hilo 1. Cuando éste llega a la iteración de generación de estadísticas, opera de forma diferente al hilo 0. Primero mira si él es el poseedor de alguno de los dos cerrojos, `WORK` o `WAIT`, si es así, se ha producido un error inesperado en el protocolo, y directamente aborta la ejecución, terminando anómalamente el programa.

10. Si sigue vivo (no tenía ninguno de los dos cerrojos), el hilo 1 intenta obtener el `WAIT`, o se queda bloqueado en espera de él.

11. Al conseguir el `WAIT` se entera de que el hilo 1 ya ha llegado también al hilo de generación de estadísticas, así que lo devuelve inmediatamente como reconocimento de haberse enterado.

12. Inmediatamente, intenta coger el cerrojo de `WORK`, dado que el hilo 0 debe estar en esos momentos generando las estadísticas, y no lo liberará hasta que el hilo 0 no termine esa iteración de trabajo estadístico, y al inicio de la iteración de estadísticas+1, recoja el `WAIT` para saber que el hilo 1 ya está preparado, y devuelva el `WORK` como señal al hilo 1 de que los datos ya están generados y en sitio seguro para poder ser gestionados por el hilo 1.

13. El hilo 1, cuando toma el `WORK`, se desbloquea, y continua con su iteración, que será la de estadísticas acabas de generar por el hilo 0.

14. Cuando el hilo 1 termina su iteración de intercambio y resumen de estadísticas, y almacenamiento en disco de ellas, si procedía, llega al inicio de la iteración de estadísticas+1.

15. Al inicio de la iteración estadísticas+1, el hilo 1 devuelve el cerrojo de `WORK` como señal de que ya ha terminado su trabajo.

16. Consecuentemente, el hilo 1 sólo ha tenido como máximo un único cerrojo en su posesión, el `WAIT` para reconocer la situación de trabajo del hilo 0, o el `WORK` cuando está gestionando los datos estadísticos generados por el hilo 0. Cuando está iterando sin hacer nada hasta la siguiente iteración de estadísticas, o bloqueado al inicio de ésta, no posee ningún cerrojo.

Con este protocolo se ha implementado la versión r27, que como se verá en el Capítulo 5 “Resultados”, es más rápida que la versión de partida r24 en aproximadamente un 3–4 % en el peor de los casos, excepto en un caso puntual que se explicará en su momento, y que pudiera tener relación con el entorno de ejecución.

A continuación se incluye el código al inicio de la iteración principal del bucle, donde se gestionan estos cerrojos. El Código 4.9 está ampliamente comentado para seguir el protocolo anterior.

Listado 4.9: Gestión de cerrojos para permitir el correcto solapamiento de cálculo y comunicación

```
1          DO jk = 1, niter - 1     ! Main loop of temporal iterations
2            IF(l_production) THEN ! Check if the statistics are active
3              if (thread==0) then ! Thread 0 does the computational tasks
4                if (nprod*int(jk/nprod) == jk .or. jk==niter-1 .or.
5       &            ncontrol*int(jk/ncontrol) == jk) then
6                 ! Iteration when the special statistical calculations must be
                     done
7                 ! Master (thread==0) cannot start if thread==1 hast not
                     finished
8                 ! messaging and storing the previous stats
9                 ! The thread==0 (master) has to have both locks to continue
10                ! It always have one at least, don't need to check if have none
11                ! Then, when he has got both, he inmediatly release lock_wait
12                if (lock_work_owner .ne. thread) then
13                  call omp_set_lock(lock_work)
```

```
14                    lock_work_owner = thread
15                    iter_work = jk
16                  elseif (lock_wait_owner .ne. thread) then
17                    call omp_set_lock(lock_wait)
18                    lock_wait_owner = thread
19                    iter_wait = jk
20                  endif
21                  call omp_unset_lock(lock_wait)
22                  lock_wait_owner = -1
23                elseif ((jk .ne. 1) .and. (nprod*int(jk/nprod)==jk-1 .or.
24       &           ncontrol*int(jk/ncontrol)==jk-1) ) then
25                  ! Iteration after the special stats have been done
26                  ! The master (thread==0) has only the lwork and waits for lwait
27                  ! When it gets lwat, then it releases lwork so it can be taken
28                  ! by thread==1 who will message and/or store that info
29                  call omp_set_lock(lock_wait)
30                  lock_wait_owner = thread
31                  iter_wait = jk
32                  call omp_unset_lock(lock_work)
33                  lock_work_owner = -1
34                endif     ! end of checking iteration number for thread==0
35              else        ! It's thread 1, which does the MPI and I/O part
36                if (nprod*int(jk/nprod) == jk .or.
37       &            ncontrol*int(jk/ncontrol) == jk) then
38                  ! Iteration when the special statistical calculations must be
                      done
39                  ! Thread==1 cannot start if master (thread==0) hast not yet
                      finished
40                  ! this very same iteration, because needs to be sure the data
                      is OK.
41                  ! Thread==1 have no lock, we check that, and release them if so
                      .
42                  ! Then thread==1 tries to get lock_wait before continue the mes
                      &IO work.
43                  ! When it gets it, releases it inmediatly and then tries to get
44                  ! the lock_work
45                  if (lock_work_owner==thread .or.
46       &              lock_wait_owner==thread) then
47                    print *,'thread ',thread,'have one lock at it=',jk,
48       &                    'That is not allowed. STOPING!!!'
49                    STOP ! No allowed, something went wrong. Abnormal termination
50                  endif
51                  call omp_set_lock(lock_wait)
52                  lock_wait_owner = thread
53                  iter_wait = jk
54                  call omp_unset_lock(lock_wait)
55                  lock_wait_owner = -1
```

```
56                  call omp_set_lock(lock_work)
57                  lock_work_owner = thread
58                  iter_work = jk
59                elseif ((jk .ne. 1) .and. (nprod*int(jk/nprod)==jk-1 .or.
60       &           (ncontrol*int(jk/ncontrol)==jk-1 .or. jk==niter-1)))
61       &           then
62                  ! Iteration after the special stats have been done
63                  ! The thread==1 has the lock_work, and releases it
64                  ! to let know the master it has already messaged and
65                  ! stored previous stats
66                  call omp_unset_lock(lock_work)
67                  lock_work_owner = -1
68                endif ! end of checking iteration number for thread==1
69              endif ! if (thread==0) then
70            ENDIF ! IF(l_production) THEN
71  ! End of lock management, never make this region OMP CRITICAL or will deadlock
```

4.5.2.1 Consideraciones con la adición de OpenMP en el código Fortran

Al utilizar hilos de OpenMP se pueden definir qué variables van a ser locales y cuáles van a ser compartidas entre ambos hilos participantes de la paralelización de la región dada. No obstante, hay un caso especial y es cuando desde Fortran se invoca a una subrutina. En ese momento, las variables que hasta ahora eran globales, desaparecen como tales, y pasan realmente a ser locales. Es más, se reinicializan a su valor por defecto (generalmente cero), sin ni siquiera conservar el valor que la variable global tenía en el momento de invocarse la subrutina. Ese comportamiento no es así si la creación de los hilos es dentro de la subrutina.

Para solventar ese problema se hace necesario, pues, pasar como parámetros en la invocación a la subrutina de todas aquellas variables que el hilo precise conocer para la ejecución correcta de la subrutina invocada. Esto hace que sea necesaria la modificación del prototipo de casi todas las subrutinas invocadas desde el programa principal, pues es allí donde se crean los hilos.

4.6 Valoración cualitativa del proceso de paralelización con CUDA

La realización de la migración paso a paso ha permitido evaluar en cada momento cuáles han sido los principales saltos cualitativos y cuantitativos obtenidos durante el proceso, así como tener que enfrentarse a decisiones para solventar aquellos problemas que han ido apareciendo. Tras este proces, se presentan de forma resumida las principales características que se consideran interesantes:

- La migración a CUDA ha resultado ser, en lo que se refiere a las funciones individuales de cómputo, relativamente sencilla.
- Dicha migración sí que ha presentado mayores problemas a la hora de definir la estrategia de migración.
- También ha resultado ser problemático analizar el algoritmo en el código implementado, puesto que en ocasiones éste no era el más evidente ni el más óptimo.

- Se han tenido que resolver problemas de programación interesantes en CUDA-C, tales como la utilización eficiente de los generadores de números aleatorios, la utilización y programación de diferentes tipos de reducción, y la optimización en la invocación a los *kernels*.
- El método de los *wrappers*, junto con los ficheros de cabecera para acceder más fácilmente a las variables globales declaradas en Fortran ha resultado eficiente. No obstante, aporta un grado de complejidad que puede representar un escalón importante para programadores no familiarizados en C.
- El análisis del código, junto con la comprensión de la parte física de las ecuaciones de Langevin con ellas implementada, ha permitido la detección de dos errores de programación que se han corregido en la implementación propuesta, así como de la propuesta e implementación de ella de algoritmos y uso de funciones más eficientes que los métodos numéricos utilizados en el código original.
- La inclusión de OpenMP en el código ya paralelizado, para obtener una aceleración aún mayor, ha resultado ser mucho más complejo que los resultados con él obtenidos. Evidentemente esto depende del caso concreto, pero el grado de complejidad, no solo en programación, sino también en la necesidad de idear algoritmos correctos libres de bloqueo, hace que esta última parte no sea recomendable intentarla si no se tiene a una persona muy experta en computación paralela y distribuida en el equipo. Se incluye la distribuida, porque el funcionamiento asíncrono de los hilos plantea problemas alejados de la computación paralela, y mucho más propios de la distribuida, con su necesidad de estudio de corrección y viveza de los algoritmos que se propongan implementar.
- Desde el punto de vista cuantitativo, la migración es un total éxito, dado que es correcta en sus resultados, y aporta una aceleración en torno a 700 respecto al código original ejecutándose en un núcleo de CPU. Económicamente, si las simulaciones son muchas, implica un importante ahorro en el tiempo de uso del *cluster* de producción utilizado, y con ello, también económico.

(Página intencionadamente en blanco)

CAPÍTULO 5

RESULTADOS

En este capítulo se presentan los resultados obtenidos en diferentes entornos con la ejecución del programa implementado, o de algunas de sus versiones previas, si ya era suficiente para sacar conclusiones válidas.

Las pruebas se han realizado en varios entornos de ejecución diferentes, por lo que será interesante presentar una breve reseña de los mismos con antelación a la presentación de dichos resultados.

Finalmente, se efectuará un análisis de los resultados, que permitirá valorar la implementación, en sus dos aspectos cuantitativos más importantes a valorar:

- su corrección, esto es, si produce los resultados que de él se espera,
- su eficiencia, y por lo tanto, qué rendimiento se extrae de su utilización, de forma comparada con otras implementaciones existentes.

No obstante, y recordando que la paralelización del código en concreto utilizado es una caso de prueba y demostración, se debe valorar cualitativamente el proceso de paralelización utilizado, en comparación con otras posibles alternativas. Todo ello en vistas a poder informar a la parte científica de los requisitos que debe contar el personal que deba acometer en el futuro esta labor, personal que actualmente, en su mayor parte, está formada por la propia comunidad científico teórica del campo en cuestión, la física de partículas.

5.1 Pruebas realizadas

Las pruebas se han realizado en dos etapas diferentes:

- inicio de la implementación, para la optimización del tamaño de la malla y estimar ya desde las primeras etapas del trabajo la aceleración mínima obtenible,
- con las implementaciones finales terminadas de las dos versiones del código, la puramente MPI con CUDA, y la que se añade OpenMP para desligar las operaciones computacionales de los *kernels* de las de comunicación y escritura en disco.

A continuación se detallan ambas etapas, y las razones y objetivos de las mismas:

5.1.1 Primera etapa: optimización del kernel principal según los parámetros propios de su invocación

En la primera etapa el trabajo se focalizó en comprobar si el proyecto era viable, o mejor dicho, si el código inicialmente escogido era paralelizable con CUDA y susceptible de tener una buena aceleración. Para ello se identificó el mínimo trozo de código que era responsable de una gran proporción del tiempo total de ejecución. En la Figura 5.1 se representa gráficamente el diagrama de llamadas obtenido con el *profiling* del programa original. En él se detectó que la subrutina `advance`, junto con aquellas a las cuáles ésta llamaba (`rhs_coeffs`, `phi_and_psi`, `rhs_ion` y `get_gaussian_noise`) era la consumidora del 85 % del tiempo de ejecución, al cual hay que añadir prácticamente otro 13 % por la parte correspondiente de las subrutinas estáticas de biblioteca `hpsort` y sobre todo de `ran2`, que son principalmente utilizadas por las subrutinas dependientes de `advance`.

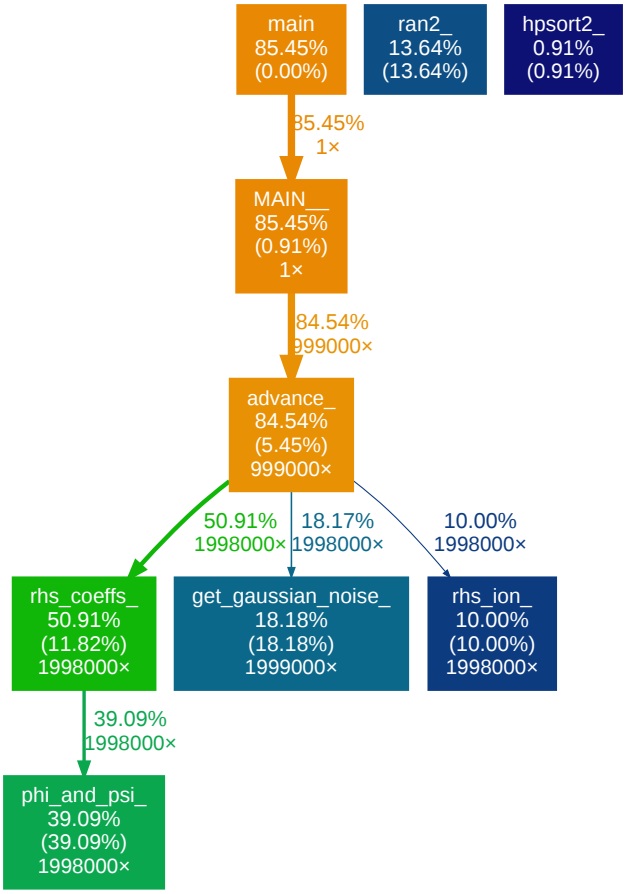


Figura 5.1: Diagrama de llamadas del programa original obtenido del perfilado

Es por ello que la primera etapa de implementación consistió en paralelizar mediante *kernels* de CUDA la subrutina `advance` y todas aquellas que son llamadas por ésta. La ejecución del código preliminar así obtenido mostró que efectivamente esta parte del código era la que más afectaba al tiempo de ejecución, al obtenerse ya un *speedup* en torno a 30–50, según el entorno de ejecución.

Tras optimizar dicho código, se lanzó una prueba de optimización de los parámetros de llamada a los

kernels. Adicionalmente a los típicos parámetros de entrada/salida de toda función C/C++, los *kernels* de CUDA se lanzan indicando entre dos y cuatro parámetros más, que se incluyen en la llamada con una notación especial que precisamente caracteriza su código de llamada:

```
Kernel_Name<<< GridSize,  BlockSize,  SMEMSize,   Stream >>> (arguments,....);
```

Los valores relevantes para nuestro caso son `BlockSize` y `GridSize`, que definen respectivamente la cantidad de bloques y cómo éstos se estructuran al lanzar la ejecución del *kernel* en la GPU. Aunque estos parámetros son de un tipo especial de vector de tres dimensiones (`x`,`y`,`z`) definido por el propio CUDA, en nuestro código no es necesaria tal complejidad y se utilizará únicamente las dimensiones `x` e `y` en el del tamaño del bloque y `x` en el tamaño de la malla.

La forma en que se invocan los *kernels* en la GPU es importante por las siguiente razones:

- el número de bloques junto con el tamaño de cada bloque determinará el número de ejecuciones individuales que se realizará del código. Debe ser el número suficiente para que acoja, en nuestro caso, todas y cada una de las partículas a simular, puesto que cada hilo de ejecución se encargará de una de ellas en exclusiva,
- el número de bloques (`GridSize`) tiene que ser suficientemente elevado como para permitir la máxima utilización de los multiprocesadores de la tarjeta gráfica,
- el tamaño del bloque (`BlockSize`) determinará un factor muy importante para la eficiencia de la ejecución paralela en la tarjeta gráfica, que es el grado de ocupación de los registros, y con ello, el grado de paralelismo solapado existente en la ejecución. Téngase en cuenta que hay mucho cálculo de doble precisión, por lo que las unidades de coma flotante estarán constantemente en utilización, y conviene utilizar el tiempo en que ese *warp* está inactivo en espera del resultado, en otros cálculos utilizando registros y otras unidades lógicas del *core*.

La prueba consiste en variar estos parámetros, teniendo siempre en cuenta que existe un límite máximo según la arquitectura de número de hilos que pueden ejecutarse concurrentemente en un bloque. Para ello, se varían las dimensiones `x` e `y` de `BlockSize`, cambiándose el valor de `GridSize` para que se cumpla que se lancen los mínimos bloques necesarios para que cada iteración (partícula) sea ejecutada una y al menos una vez por un único hilo.

Estas prebas se realizan en diversos entornos reducidos a una única máquina, dado que en este caso, el resultado no depende de la masividad del cómputo, sino de las características hardware de la tarjeta gráfica utilizada, estudiando para ello tres diferentes generaciones de arquitectura CUDA: Fermi, Kepler y Maxwell (primera generación).

5.1.2 Segunda etapa: medición de prestaciones de las versiones finales en un entorno HPC real

Una vez obtenidas las dimensiones óptimas para el lanzamiento del principal *kernel*, el trabajo se centra en paralelizar el resto de funciones que están dentro del bucle principal del programa. De esta forma se obtiene un programa final que realiza prácticamente todos los cálculos sobre cada partícula utilizando CUDA. Esto permite evitar traspasos continuos de memoria entre la CPU y la GPU en cada iteración temporal. Sólo se realizan aquellas transferencias de memoria necesarias, siempre de GPU a CPU, para enviar los datos mínimos necesarios para que la CPU pueda guardar los datos estadísticos y de progreso de la simulación en disco (o presentarlos por consola).

Las pruebas confirman la corrección de la implementación, y por ello sólo queda comprobar el grado de eficiencia, o aceleración, obtenido con la paralelización. Estas pruebas de rendimiento se realizan en un entorno HPC real, para obtener resultados de calidad para obtener las conclusiones que realmente son el objetivo de este trabajo: si interesa o no paralelizar código antiguo científico utilizando el nuevo paradigma ofrecido por la GPGPU, y concretamente, usando la tecnología CUDA.

5.2 Entorno de ejecución

5.2.1 Primera etapa: optimización de los parámetros de invocación del kernel principal

Estas pruebas se realizan utilizando hardware común, no especialmente diseñado para el cómputo de altas prestaciones. Para ello se dispone de tres ordenadores personales, con monoprocesador multinúcleo, en el que cada uno de ellos dispone de una única tarjeta gráfica de NVIDIA, cada una de ellas de una generación y arquitectura CUDA diferente. El sistema operativo utilizado es Linux, usando diferentes distribuciones y diferentes versiones de compilador, y diferentes versiones de CUDA. En la Tabla 5.1 se muestran las principales características de estos entornos:

	Nombre de la computadora		
Característica	athor.dsic.upv.es	gpu.dsic.upv.es	FEDORAFujitsu
Distribución Linux (SO)	Debian jessie/sid	RedHat 6.6	Fedora 22
Versión del kernel	3.13.0	2.6.32	4.1.2
Versión de `gcc`	4.8.2	4.4.7	5.1.1
Versión del driver de CUDA	7.0	7.0	7.5
Versión del *runtime* de CUDA	7.0	7.0	7.0
Marca y familia CPU	Intel© Core™	Intel© Core™	AMD Phenom™ II
Modelo de CPU	i7 950	i7 3820	X6 1100T
Núcleos en la CPU	8	8	6
Frecuencia de la CPU	3,07 GHz	3,60 GHz	3,3 GHz
Memoria RAM	8 GB	16 GB	16 GB
Tarjeta gráfica	Tesla C2050	Tesla K20c	GeForce GTX 750 Ti
Cantidad de tarjetas instaladas	2	2	1
Arquitectura NVIDIA	Fermi	Kepler	Maxwell (1ª gen.)
Generación del chip de la GPU	GF100	GK110	GM107
Capacidad computacional CUDA	2.0	3.5	5.0
Número de multiprocesadores	14	13	5
Número total de núcleos	448	2496	640
Frecuencia máxima de la GPU	1147 MHz	706 MHz	1100 MHz
Memoria RAM de la GPU	3 GB	5 GB	2 GB
Frecuencia de la RAM de la GPU	3,0 Gbps	5,2 Gbps	5,4 Gbps
Ancho de banda de la memoria	144 GB/s	208 GB/s	86,4 GB/s
Interfaz de memoria	348 bits	320 bits	128 bits
Rendim. pico coma flotante simple	1,03 Tflop/s	3,52 Tflop/s	1,305 Tflop/s
Rendim. pico coma flotante doble	515 Gflop/s	1,17 Tflop/s	408 Gflop/s
Máxima potencia consumida por GPU	238 W	225 W	60 W

Tabla 5.1: Entornos de ejecución para la prueba de parámetros de invocación del *kernel* `advance`

En los dos primeros entornos, las tarjetas gráficas son de la serie Tesla, esto es, están pensadas directamente para la computación numérica vía CUDA. Éstas tienen la ventaja de ser más robustas, y de permitir cambiar el modo de operación de la memoria, para poder utilizar corrección de errores (aunque con penalización de la cantidad de memoria libre y de cómputo). En el otro caso, se hace uso de una tarjeta de la serie GeForce, y se escogió por ser una de las primeras gráficas que se basaba en la arquitectura Maxwell, aunque de la primera generación. Aunque presenta ventajas arquitectónicas respecto a las anteriores (mejor rendimiento energético, en flops/W, mejor *cache*, mayor capacidad CUDA), al ser de uso doméstico su rendimiento computacional es sensiblemente inferior, por disponer de menor número de núcleos, menos cantidad de memoria interna, y una interfaz menor con esa misma memoria.

5.2.2 Segunda etapa: entorno HPC real, MinoTauro

Las pruebas a gran se realizan en un entorno de producción real de computación de altas prestacion es ideado para la computación con CUDA, al disponer todos sus nodos de dos tarjetas gráficas NVIDIA. A continuación se resumen sus carcterísticas, que se pueden consultar en `https://www.bsc.es/support/MinoTauro-ug.pdf`.

Cada nodo dispone de 2 GPU de 512 núcleos y de 2 CPU de 6 núcleos. Como se verá en los resultados de las pruebas realizadas, la red de interconexión es muy buena. Existen en realidad dos redes: InfiniBand se usa para las comunicaciones mediante MPI, mientras que la GigabitEthernet para el sistemas de ficheros paralelo distribuido GPFS.

5.3 Ejecución de las pruebas

A continuación se presentan los resultados obtenidos en las pruebas realizadas. Primero se justificará la elección de los parámetros utilizados para la invocación de los *kernels*, cuyos valores serán utilizados en las siguientes pruebas de rendimiento en el entorno real de HPC.

5.3.1 Parámetros óptimos de invocación al *kernel* principal `advance`

5.3.1.1 Breve descripción de la prueba

Se realizan pruebas en los tres entornos indicados con anterioridad en la Tabla 5.1. Éstas consisten en compilar un programa preliminar en el que únicamente se ha paralelizado con CUDA la subrutina `advance` y todas aquellas a las que ésta llama, a saber, `rhs_coeffs`, `phi_and_psi`, `rhs_ion` y `get_gaussian_noise`. Además, la invocación a `ran2` se sustituye por la llamada a funciones aleatorias propias de CUDA. Con esto se consigue que el anterior 98 % del tiempo de ejecución sea ahora ejecutado completamente en la tarjeta gráfica.

En cada compilación se cambian los parámetros de invocación al *kernel* `GridSize` y `BlockSize`. Las variables de la optimización son `BlockSize.x` y `BlockSize.y`, y con ello se recalcula `GridSize` para asegurar que se ejecuten los mínimos bloques necesarios para que cada partícula se calcule siempre una y una única vez.

Las ejecuciones se realizan con la versión final del programa de CUDA con MPI (sin OpenMP) y se lanzan en un sólo proceso con una sóla GPU, dado que lo que interesa es el tiempo de ejecución del *kernel*. Es por ello que se toma únicamente el tiempo que el programa emplea en ejecutar los *kernels*, sin contar el tiempo por escritura en disco, ni gestión de intercambio de mensajes MPI. Se toma un problema relativamente pequeño, de 10000 partículas durante 20000 iteraciones temporales.

Software	
Distribución Linux (SO)	Red Hat Enterprise Edition 6.0
Versión del kernel	2.6.32
Versión de `gcc`	4.6.1
Versión del driver de CUDA	7.0
Versión del *runtime* de CUDA	7.0
Gestor de colas	SLURM 2.2.7-Bull-4
Hardware básico	
Modelo de CPU	Intel Xeon E5649
Cantidad de CPU instaladas	2
Núcleos en la CPU	6
Frecuencia de la CPU	2,53 GHz
Memoria RAM	24 GB
Hardware gráfico, aceleradores	
Tarjeta gráfica	Tesla M2090
Cantidad de tarjetas instaladas	2
Arquitectura NVIDIA	Fermi
Generación del chip de la GPU	GF110
Capacidad computacional CUDA	2.0
Número de multiprocesadores	16
Número total de núcleos	512
Frecuencia máxima de la GPU	1301 MHz
Memoria RAM de la GPU	6 GB
Frecuencia de la RAM de la GPU	3,7 GHz
Ancho de banda máximo	177 Gbit/s
Interfaz de memoria	384 bit
Rendim. pico coma flotante simple	1.332 Gflop/s
Rendim. pico coma flotante doble	666 Gflop/s
Máxima potencia consumida por GPU	225 W
Red de interconexión	
Cantidad de nodos totales	126
Cantidad de nodos computacionales	124
Cantidad de nodos de gestión	2
Cantidad de conmutadores (*switches*)	14
Tarjeta de comunicaciones	InfiniBand MT26428
Características InfiniBand	ConnectX VPI PCIe 2.0 5 GT/s
Red de conexión 1	Infiniband (para uso MPI)
Red de conexión 2	10 GigabitEthernet (para uso GPFS)

Tabla 5.2: Entorno de computación de altas prestaciones MinoTauro

5.3.1.2 Resultados obtenidos

A continuación se presentan los resultados de forma tabulada para cada uno de los entornos. Por error, en uno de los entornos, el de `gpu.dsic.upv.es`, se compiló primeramente para la capacidad CUDA 2.0, cuando la tarjeta se trata de una Kepler con capacidad CUDA 3.5. Se muestran los resultados por separado de ambas pruebas porque permite realizar algunas observaciones interesantes al respecto.

Nótese que los tiempo de ejecución que se muestran es de sólo la ejecución de los *kernels*, no de todo el programa. Se comprobó que el resto de tiempos eran estadísticamente iguales entre sí, y además estamos optimizando el propio tiempo de ejecución CUDA.

5.3.1.3 Fenómenos observados

De los resultados mostrados en la Tabla 5.3 se extraen las siguientes observaciones:

- Existe una cierta simetría con la diagonal que va desde la esquina superior derecha (mayor `Y` y menor `X`) a la inferior izquierda (mayor `X` y menor `Y`). Se trata de una simetría cualitativa que se tamiza y explica mejor al ver las siguientes observaciones.
- La simetría arriba mencionada puede expresarse cuantitativamente a partir del valor del producto (`BlockSize.x` · `BlockSize.y`), que tiene el significado de ser el número de hilos por bloque. Por lo tanto, una variable importante (y evidente) es el número de hilos del bloque. No obstante, y un poco sorprendentemente, parece no ser lo mismo la geometría de definición de esos hilos, sobre todo cuando dicha geometría `X` e `Y` es aplanada en los *kernels*, puesto que allí sólo sirven para calcular el índice `idx` que normalmente va asociado unívocamente a cada partícula de la simulación.
- Cuantos más hilos por bloque, los resultados tienden a mejorar, pero sólo hasta cierto punto (en nuestro caso generalmente al llegar a 32–64 hilos. La razón es que hay mucho cálculo matemático con doble precisión. Estas operaciones son costosas computacionalmente, pero la GPU dispone de varias unidades de ese tipo por cada multiprocesador (depende de la generación de la GPU), por lo que pone en espera esos hilos que han entrado en un cómputo largo de doble precisión, y reactiva otros. Si no hay más, el multiprocesador se queda parado, con ciclos ociosos esperando que terminen las unidades especiales de coma flotante doble. No obstante, llega un momento en que ya hay suficientes hilos para ir rotándolos para ocupar eficientemente la mayor cantidad posible de unidades funcionales de cada multiprocesador. Cuando eso sucede, añadir más hilos no hace más que generar más sobrecarga por la mínima gestión que hace el microprocesador para cambiar el contexto de los diferentes hilos del bloque. Por esa razón, en general, si se desconoce el grado de ocupación que va a tener el *kernel* programado, es mejor excederse incluso en un término medio a quedarse corto por muy poco.
- A pesar de la anterior observación, existen límites al número de hilos a invocar por bloque. Primero, hay un número máximo de hilos por bloque según la arquitectura de la GPU, pero que lo habitual es estar en 1024 (de ahí que sólo se probaran valores de `BlockSize.xy` `BlockSize.y`en el intervalo 1–32. Segundo, aún estableciendo un número inferior a ese límite máximo dado por la arquitectura de la tarjeta gráfica, se puede producir un fenómeno de sobreutilización de los registros, en cuyo caso, el *kernel* falla por error de falta de recursos (celdas marcadas con `ERROR` en las tablas). Se observa que en la GPU de arquitectura Fermi, que es de la generación más antigua, sólo se produce un fallo por falta de recursos, en el caso de 32×32 (1024 hilos), pero funciona correctamente, aunque más lentamente, en los casos próximos de 512 hilos (16×32 y 32×16).
- En resumen, la variación de los parámetros sí que tiene una fuerte influencia en el resultado obtenido en su ejecución, sobre todo en los casos extremos, a saber:

	Y									
X	**1**	**2**	**4**	**6**	**8**	**12**	**16**	**32**	Escala de colores	
1	90,030	47,250	26,140	19,230	15,350	12,340	10,620	7,340	4,830	>=0%
2	46,470	25,240	14,220	10,650	8,800	7,330	6,120	5,750	4,927	>=2%
4	24,820	13,740	8,150	6,290	5,040	6,490	5,220	5,580	5,023	>=4%
6	17,750	10,030	6,230	7,530	6,300	6,560	5,220	5,600	5,313	>=10%
8	13,700	7,990	**4,830**	6,150	**5,010**	**5,030**	5,140	5,210	6,038	>=25%
12	10,490	6,490	6,530	6,410	5,050	5,510	5,570	5,780	7,245	>=50%
16	8,590	5,290	5,080	**5,020**	5,170	5,540	5,220	5,310	9,660	>=100%
32	5,420	5,440	5,530	5,820	5,610	5,920	5,610	ERROR	28,980	>=500%

(a) Athor - Tesla C2050 (Fermi, capacidad CUDA 2.0)

	Y									
X	**1**	**2**	**4**	**6**	**8**	**12**	**16**	**32**	Escala de colores	
1	25,920	14,360	9,020	7,070	6,280	5,210	4,910	4,340	3,110	>=0%
2	14,230	8,350	5,560	4,600	4,240	3,610	3,630	3,620	3,172	>=2%
4	8,150	5,210	3,920	3,250	3,280	3,380	3,370	3,280	3,234	>=4%
6	6,190	4,340	3,310	3,900	3,340	3,380	3,310	3,360	3,421	>=10%
8	5,300	3,890	3,200	3,320	3,290	3,260	3,270	3,770	3,888	>=25%
12	4,230	**3,150**	3,240	3,320	3,190	3,590	3,290	3,350	4,665	>=50%
16	3,790	**3,110**	3,330	3,160	3,310	3,190	3,660	ERROR	6,220	>=100%
32	**3,170**	**3,140**	3,160	3,200	3,640	3,240	ERROR	ERROR	18,660	>=500%

(b) Gpu.dsic - Tesla K20c (Kepler, capacidad CUDA 2.0)

	Y									
X	**1**	**2**	**4**	**6**	**8**	**12**	**16**	**32**	Escala de colores	
1	26,070	14,320	9,070	6,960	6,370	5,180	4,840	4,300	3,080	>=0%
2	14,150	8,260	5,490	4,520	4,200	3,590	3,630	3,570	3,142	>=2%
4	8,070	5,170	3,900	3,220	3,290	3,330	3,280	3,320	3,203	>=4%
6	6,150	4,210	3,440	3,840	3,360	3,310	3,250	3,310	3,388	>=10%
8	5,220	3,980	3,150	3,280	3,170	3,180	3,250	3,750	3,850	>=25%
12	4,240	3,110	3,210	3,190	3,170	3,500	3,190	3,260	4,620	>=50%
16	3,780	**3,090**	**3,080**	3,140	3,140	3,150	3,800	ERROR	6,160	>=100%
32	3,100	3,120	3,320	3,240	3,600	3,240	ERROR	ERROR	18,480	>=500%

(c) Gpu.dsic - Tesla K20c (Kepler, capacidad CUDA 3.5)

	Y									
X	**1**	**2**	**4**	**6**	**8**	**12**	**16**	**32**	Escala de colores	
1	206,400	106,690	55,410	37,500	28,440	19,440	14,780	8,670	7,680	>=0%
2	107,070	54,760	28,280	19,190	14,640	10,120	7,760	7,810	7,834	>=2%
4	55,440	28,300	14,710	10,080	7,720	10,110	7,800	7,960	7,987	>=4%
6	37,250	19,170	10,090	12,980	9,910	10,030	7,810	7,920	8,448	>=10%
8	28,200	14,600	7,760	9,920	7,730	7,810	7,840	8,320	9,600	>=25%
12	19,070	10,010	9,990	10,120	7,720	9,200	7,740	7,990	11,520	>=50%
16	14,520	**7,680**	**7,680**	7,720	7,730	7,750	8,240	ERROR	15,360	>=100%
32	8,450	8,400	8,400	8,450	8,630	8,440	ERROR	ERROR	46,080	>=500%

(d) FUJITSUFedora - GTX 750 Ti (Maxwell, capacidad CUDA 5.0)

Tabla 5.3: Influencia de los parámetros `BlockSize.X` y `BlockSize.Y` de invocación a los *kernels* en su tiempo de ejecución en función del entorno y la capacidad CUDA de la GPU usada

  - Pocos hilos: los tiempos de ejecucion se disparan por infrautilización de las diferentes unidades computacionales de cada microprocesador encargado de ejecutar cada bloque.
  - Suficientes hilos: se obtiene la combinación ideal y se obtienen los mejores tiempos.
  - Más hilos de los necesarios: el rendimiento se resiente, pero mucho más ligeramente que si faltan.
  - Exceso en demasía de hilos: el *kernel* se aborta por falta de recursos de cada microprocesador para poder gestionar tantos hilos.

- Los parámetros óptimos globales de ejecución de los *kernels* de la implementación presentada de CUDA+MPI depende ligeramente de la arquitectura, pero está en el orden de lanzar 32–64 hilos, con preferencia general por la configuración `BlockSizeX=16` y `BlockSizeX=2 ó 4`, excepto en la tarjeta Fermi, que esa combinación era considerablemente peor que la más simétrica de `BlockSizeX=BlockSizeX=8`.

5.3.1.4 Conclusión de la prueba

De los resultados y de las observaciones realizadas, se concluye que a la hora de lanzar simulaciones reales en un entorno de producción es interesante previamente invertir cierto tiempo en determinar para ese entorno concreto con qué combinación de parámetros `BlockSize.y`e `BlockSize.y`hay que compilar el programa, para minimizar el tiempo de ejecución, siempre escaso y valioso en los entornos HPC.

5.3.2 Aceleración en un entorno de ejecución de altas prestaciones

Las pruebas más concluyentes son las que se realizan en el entorno de ejecución de MinoTauro, dado que se trata de un *cluster* específicamente diseñado para la ejecución de programas paralelizados con CUDA con GPU en todos los nodos, los cuales están interconectados por una rápida red de InfiniBand. Las pruebas han consistido en ejecutar primeramente el código original en Fortran con diferentes cargas (diferente número de partículas con diferentes números de iteraciones cada vez) y con diferentes configuraciones del programa (básicamente, generar o no todas las estadísticas 2D cada cierta cantidad de iteraciones). Los mismos ficheros de carga han sido utilizados para lanzar las pruebas de ejecución con los dos programas finales presentados: la versión CUDA+MPI (que denominaremos r24) y la versión con CUDA+MPI+OpenMP (que denominaremos r27).

Estas baterías de pruebas nos permitirán analizar los siguientes aspectos:

- Variación de la aceleración con el incremento del número de nodos para una misma carga, de las dos versiones (r24 y r24) frente a la versión original de sólo CPU+MPI.
- Variación de la aceleración con el incremento de la carga
- Influencia de la saturación de la memoria de la GPU utilizando cargas que obliguen a utilizar casi el 100 % de su memoria, frente a la utilización de sólo del 60 %.
- Mejora obtenida con la adición de OpenMP para conseguir la asincronía entre el cálculo puro realizado en la GPU del proceso de intercambio de datos obtenidos a través de mensajes MPI y la escritura en disco de dichos datos intermedios.
- En general, analizar la escalabilidad global de los programas presentados, tanto espacial como en número de nodos.

También se obtendrán conclusiones interesantes sobre aspectos inicialmente no planificados, como por ejemplo, influencia del lanzamiento de un número no divisible por 2 del tamaño de los bloques, detección de posibles problemas con la utilización de OpenMPI con InfiniBand.

Las pruebas se realizan en el entorno de ejecución de MinoTauro. Se utiliza un número variable de nodos, así como un número variable de GPU. También se varía el número de GPU por nodo que se utiliza en cada prueba, para ver si la mayor localidad de los procesos en menos nodos incrementa la eficiencia de la ejecución de la simulación. Se seleccionan diferentes tamaños de carga, variando tanto el número de iteraciones temporales, como el número de partículas que son consideradas en cada iteración. También se cambia el tipo de simulación (con generación o no de resultados estadísticos intermedios, con fuerte carga de intercambio de mensajes y escritura en disco). Y finalmente se utilizan tres programas: el código original en Fortran+MPI; la nueva implementación sustituyendo la parte computacional con CUDA, y algo de C para facilitar la invocación a la API; y la que mejora ésta última con la adición de OpenMP para desacoplar el cómputo de la comunicación, y así solapar lo más posibles estas dos actividades.

Todas estas pruebas se hacen de una forma ortogonal (ortogonalidad no completa, pero sí suficiente para extraer conclusiones adecuadas). De esta forma se puede estudiar la escalabilidad del programa bajo diferentes puntos de vista: espacial, temporal y localidad, así como evaluar si la inclusión de CUDA primero y de OpenMP después compensan respecto de su respectivo tiempo de desarrollo.

A continuación se procede a describir, presentar resultados, analizar los resultados, y obtener conclusiones sobre cada una de las pruebas realizadas.

5.3.2.1 Caracterización del comportamiento del programa original

Para poder evaluar la nueva implementación, se necesita hacerlo por comparación con el código original de partida. En esta prueba servirá para caracterizar el comportamiento del programa original, ya que la nueva implementación deberá mejorar los aspectos más débiles del original, sin que los aspectos fuertes de éste sufran merma. La caracterización consistirá en analizar su escalabilidad espacial y temporal.

Breve descripción de las pruebas

Consiste en la ejecución del código original en el entorno de MinoTauro mediante las siguientes pruebas:

1. Con una carga pequeña constante (10.000 iteraciones temporales de 20.000 partículas), variar el número de procesos que se ejecutan en un único nodo. Con esto se comprobará la escalabilidad en un entorno puro de memoria compartida.

2. Con la misma carga pequeña constante anterior, ejecutar siempre con 12 procesos, pero con un número variable de nodos, variando por lo tanto el número de procesos por nodo. Así se podrá comparar la velocidad de intercomunicación en memoria compartida con la existente entre diferentes nodos (memoria distribuida).

3. Con una carga grande constante (500.000 iteraciones temporales de 200.000 partículas), variar el número de nodos, y por lo tanto de procesos, haciendo siempre que cada nodo trabaje al 100% de su capacidad, esto es, con `2x6 = 12` procesos por nodo. La idea es comprobar la escalabilidad del programa en un entorno mixto de memoria compartida y distribuida a gran escala.

Los tamaños son escogidos con dos ideas:

- La carga pequeña es para que las ejecuciones no sean excesivamente largas, sobre todo cuando se ejecutan en un único nodo.
- La carga mayor está relacionada con la memoria máxima que una única GPU del *cluster* puede soportar, y el número de iteraciones es uno ya suficientemente elevado como para que el fenómeno del *plateau* (meseta) en la formación de los electrones fugitivos sea claro, se vea claramente estabilizado, en estado estacionario.

Resultados obtenidos

Las pruebas 1 y 2 se realizan en un único nodo, para ver la escalabilidad en memoria compartida. En la Tabla 5.4 se muestran los resultados obtenidos.

Procesos	Nodos	Procesos por nodo	Tiempo de ejecución	Aceleración	Eficiencia
1	1	1	2045,806 s	1,000	1,000
2	1	2	1021,929 s	2,002	1,001
3	1	3	683,256 s	2,994	0,998
4	1	4	511,999 s	3,996	0,999
6	1	6	343,132 s	5,962	0,994
9	1	9	228,449 s	8,955	0,995
12	1	12	170,748 s	11,981	0,998
12	2	6	171,560 s	11,925	0,994
12	3	4	173,361 s	11,801	0,983
12	4	3	171,696 s	11,915	0,993
12	6	2	172,575 s	11,855	0,988
12	12	1	170,895 s	11,971	0,998

Tabla 5.4: Aceleración y eficiencia del código original de CPU (memoria compartida y distribuida)

En la Gráfica 5.2, en representación doble logarítmica, se representan los datos obtenidos de la Prueba 3, con gran cantidad de procesos y una carga grande. Se observa la disminución del tiempo de ejecución conforme se incrementa el número de nodos utilizados en la computacion de forma muy proporcional al número de procesos participantes.

Fenómenos observados

En la pruebas 1, sobre un único nodo, esto es, sólo con memoria compartida, tal como se ve en la Tabla 5.4 en las filas sombreadas de amarillo, la escalabilidad es prácticamente lineal en todo el intervalo. Lo mismo es cierto para cuando se cambia progresivamente a memoria distribuida (filas sombreadas de azul), donde igualmente la eficiencia es prácticamente la unidad en todos los casos, nunca bajando del 98 %.

En la prueba 3, mezcla de memoria compartida y distribuida, Figuras 5.2 y 5.3, se observa que la línea es casi recta, con una pendiente de prácticamente la unidad, lo que adelanta que la aceleración será bastante lineal. Ese hecho se comprueba en la Gráfica 5.3 que visualiza que tanto la aceleración como la eficiencia se aproximan a la idealidad (eficiencia muy cercana a 1 incluso con 64 nodos, que son 768 procesos, y la línea muy coincidente con la ideal).

Aún así, con 64 nodos (768 procesos MPI) la eficiencia sigue siendo muy alta, prácticamente del 95 %.

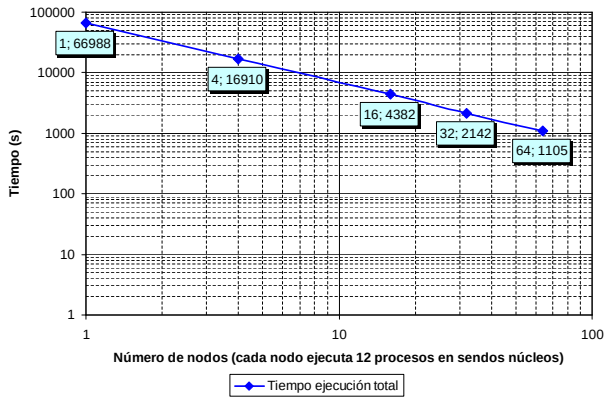


Figura 5.2: Escalabilidad temporal del código original de CPU (500.000 partículas y 200.000 iteraciones)

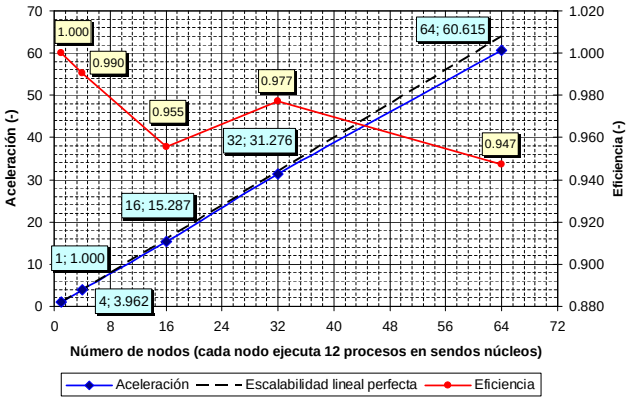


Figura 5.3: Aceleración y eficiencia del código original de CPU (500.000 partículas y 200.000 iteraciones)

Conclusiones de las pruebas

A raíz de estas tres pruebas, se obtienen las siguientes conclusiones respecto al programa original en Fortran paralelizado con MPI ejecutándose sólo en CPU y también sobre el entorno de ejecución de MinoTauro.

- El programa original escala muy bien en memoria compartida, pero al ser tan lento, no es suficiente para simulaciones que permitan extraer conclusiones físicas interesantes.
- El programa original escala muy bien en un *cluster* HPC, lo que hasta ahora ha permitido realizar simulaciones suficiente para extraer conclusiones.
- La red de comunicaciones del *cluster* MinoTauro es muy rápida. No hay diferencia sustancial entre el uso de memoria compartida y la memoria distribuida.

No obstante, se observa que para poder alcanzar tamaños de simulación interesantes, se necesitan grandes recursos computacionales durante un tiempo nada despreciable. Como se verá, la paralelización con CUDA permitirá reducir drásticamente esta relación entre tiempo necesario y recursos hardware utilizados, lo que implicará o un importante ahorro de explotación, y la posibilidad de lanzar simulaciones mucho más grandes que permitan obtener resultados más conclusivos, o disminuir la cantidad de aproximaciones realizadas para alcanzar un conjunto de ecuaciones más complicadas de computar, pero más fieles respecto del comportamiento esperado del plasma de partículas simulado.

5.3.2.2 Caracterización básica de la implementación consistente en la paralelización con CUDA

A continuación se va a proceder a estudiar el comportamiento de la versión final del programa en el que, basándose en el código original en Fortran+MPI ejecutable en CPU, se realiza una paralelización de toda la parte de computación numérica y del cálculo de las estadísticas intermedias mediante CUDA. De esta forma, Fortran queda sólo como soporte básico para la lectura y escritura de ficheros y la invocación a la librería MPI para el intercambio de datos entre los diferentes procesos, así como también del flujo de control básico del programa, incluyendo la gestión de las iteraciones temporales, y de cuándo corresponde realizar las estadísticas.

La implementación ha sido tal que tan sólo es necesaria la transferencia de los datos de velocidades de las partículas al principio y al final del programa. Durante todas las iteraciones temporales, la transferencia de datos se realiza únicamente de la GPU a la CPU solamente en aquellas iteraciones en las que hay que

recopilar y guardar en disco las estadísticas intermedias solicitadas, y sólo de los mínimos datos necesarios. De esa forma se ha conseguido disminuir mucho la transferencia de memoria entre la GPU y la CPU, lo que permite incrementar el porcentaje de código que el programa está realizando cómputos, y por lo tanto, por la Ley de Amdahl, permitirá un mucho mejor aprovechamiento del poder de la paralelización, como se observará a través de los resultados que a continuación se presentan y discuten.

Al igual que en la sección anterior, primero se detallarán las pruebas realizadas, después se presentarán los datos cuantitativos resumidos obtenidos de dichas pruebas, que permitirán observar las principales características de la ejecución, y con ello, caracterizar la implementación, a partir de lo cual se extraerán conclusiones sobre ella.

5.3.2.2.1 Prueba de corrección Durante todo el proceso de implementación se ha ido comprobando que los resultados de cada parte que se implementaba no cambiaban los resultados intermedios estadísticos que genera el programa. En realidad, los mismos resultados no pueden generarse, dado que se cambió el generador de números aleatorios. Sin embargo, forzando a que unos pocos números sean los mismos se comprobó que las velocidades y su clasificación en contenedores fueran iguales a los del código original. De hecho, durante esta revisión se localizaron varios posibles *bugs* en el código original que fueron puestos en conocimiento de sus responsables.

A partir de ese momento, los números generados aleatoriamente se fijaron para la GPU hasta la versión final, y aunque al procesar con diferente número de GPU esos valores vuelven a cambiar, sí que se observa que la distribución de los valores en cada columna en los ficheros generada es estadísticamente similar a los del código original.

Además, visualizando los ficheros `.silo` con VisIt, se observa que los mapas de velocidades y su distribución son similares a los que se muestran en [4].

Es por todo ello que se considera que las implementaciones presentadas son correctas en cuanto a la fidelidad de la resolución de las ecuaciones físicas que se usan para simular el plasma confinado magnéticamente.

5.3.2.2.2 Aceleración y eficiencia respecto del código en CPU

Breve descripción de las prueba

La prueba consiste en ejecutar la misma carga que se utilizó en la prueba de la caracterización de la CPU, cuyos datos se han mostrado gráficamente en la Figuras 5.2 y 5.3, esto es, 200.000 iteraciones temporales sobre 500.000 partículas. Se realizan varias simulaciones variando dos parámetros:

- El número de procesos, con cada proceso utilizando una GPU. Se varía desde 1 nodo con 1 GPU (caso base) hasta 128 procesos, utilizando 128 GPU.
- El número de GPU que se utilizan en cada nodo. Las mismas pruebas se realizarán utilizando una GPU por nodo, o dos.

Esto último permitirá establecer si hay alguna diferencia importante entre la ejecución dentro de un mismo nodo o con un mayor número de nodos, y por lo tanto, con una mayor utilización de la red de interconexión.

Resultados obtenidos

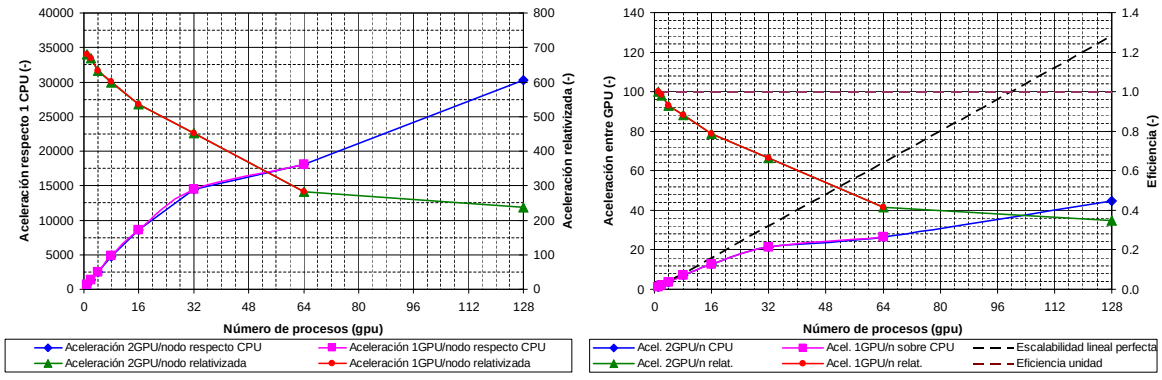
En la Tabla 5.5 se muestran los datos de esta prueba. Los experimentos se han lanzado por duplicado, siendo muy coincidentes en todos los casos parejos, por lo que en la tabla se muestra el valor promedio medido.

Procesos	Nodos	$\frac{\text{GPU}}{\text{nodo}}$	Tiempo de ejecución					Aceleración	Eficiencia	Aceleración GPU/CPU	Aceleración relativizada
			Total	GPU	I/O	MPI	Otros				
1	1	1	1181,871	1177,111	4,559	0,057	0,144	1,000	1,000	680	680
2	2	1	600,646	597,715	2,596	0,386	−0,051	1,968	0,984	1338	669
4	4	1	317,959	315,880	1,631	0,525	−0,077	3,717	0,929	2528	632
8	8	1	167,233	165,699	1,141	0,515	−0,122	7,067	0,883	4807	601
16	16	1	93,997	92,619	0,901	0,583	−0,106	12,573	0,786	8552	534
32	32	1	55,610	54,437	0,792	0,565	−0,184	21,253	0,664	14455	452
64	64	1	44,437	38,298	0,716	5,525	−0,101	26,597	0,416	18090	283
2	1	2	601,306	598,022	2,780	0,512	−0,007	1,966	0,983	1337	668
4	2	2	317,888	315,954	1,625	0,370	−0,060	3,718	0,929	2529	632
8	4	2	167,683	166,181	1,151	0,431	−0,079	7,048	0,881	4794	599
16	8	2	93,997	92,619	0,901	0,583	−0,106	12,573	0,786	8552	534
32	16	2	55,610	54,437	0,792	0,565	−0,184	21,253	0,664	14455	452
64	32	2	44,437	38,298	0,716	5,525	−0,101	26,597	0,416	18090	283
128	64	2	26,550	25,318	0,656	1,086	−0,509	44,515	0,348	30277	237

Tabla 5.5: Aceleración y eficiencia de la implementación Fortran+MPI+CUDA con misma carga que Tabla 5.4

Para el cálculo de la aceleración sobre la CPU, y dado que el código original escala linealmente de forma perfecta, se ha tomado el tiempo total de ejecución en un nodo completo, corrigiéndose el hecho de que ese tiempo se corresponde con 12 procesos.

Estos datos se visualizan en las Gráficas 5.4.



(a) Comparación con el código original en CPU

(b) Comperación entre las propias versiones en GPU

Figura 5.4: Aceleración y eficiencia de implementación Fortran+MPI+CUDA con misma carga que Tabla 5.4

Fenómenos observados

En la Tabla 5.5 se observan los siguientes fenómenos:

- La aceleración de una única GPU frente a un único núcleo de CPU alcanza el elevado valor de 680. Esto quiere decir que una única GPU realiza el mismo trabajo prácticamente que 32 nodos biprocesador como los existentes en MinoTauro.

- Conforme se utilizan más GPU la aceleración incrementa, sin notarse que llegue un momento de desaceleración.
- Conforme se utilizan más GPU la aceleración relativa decrementa. Esto es, añadir el doble de nodos con GPU al mismo trabajo no disminuye a la mitad el tiempo global de ejecución. Esto es, la eficiencia se observa que baja, hasta llegar a un 35 % al usar 128 nodos. Este fenómeno es el esperable, dado que el mantenerse constante trabajo total, al incrementarse el número de GPU, disminuye su carga individual, por lo que en general, al disminuir la carga computacional por nodo, es esperable cierto decremento de prestaciones. En todo caso, en la práctica, es una situación que no suele darse, puesto que el lanzamiento se suele ya realizar sobre los mínimos procesos necesarios para evitar el coste de comunicaciones y aprovechar la localidad de la carga.
- Al igual que en pruebas anteriores, la misma carga repartida entre el doble de nodos que trabajan con la mitad de sus recursos (sólo 1 procesador CPU y sólo 1 tarjeta gráfica) tiene prácticamente el mismo tiempo de ejecución que si se ejecutan de forma más concentrada en la mitad de nodos a tope de capacidad (2 procesadores CPU con 2 GPU).
- Sorprendentemente, cuando se utilizan 64 nodos en MinoTauro, en todas las ejecuciones realizadas en esta prueba, el tiempo parcial empleado por la aplicación en comunicaciones MPI se incrementa de manera súbita de 0,5 segundos a 5,5 segundos. Cuando se utilizan 128 nodos, vuelve a baja a un valor justificable entre 0,5 y 1,0 segundos.
- Los tiempos parciales de escritura (I/O) decrementan conforme aumenta el número de nodos. Esto es así por las características del sistema de ficheros distribuido GPFS que usa MinoTauro.

Conclusiones de las pruebas

De las observaciones anteriores se concluye que:

- La implementación en GPU del método aproximativo de Langevin es muy eficiente comparada con la original en CPU+MPI, con una aceleración de 680.
- Esta primera implementación con CUDA no escala bien cuando el número de GPU incrementa más allá de 16, aunque sigue acelerándose el cómputo. Este hecho es el que impulsa la siguiente versión añadiendo OpenMP para desacoplar más el cálculo de la comunicación y la escritura en disco.
- Se confirma que la red de interconexión MinoTauro es excelente. No obstante, hay algunas combinaciones de número de nodos, quizá ligada a la topología resultante y probablemente también a algún problema entre OpenMPI e InfiniBand, que hace que el tiempo de comunicación por MPI se dispare, con la consiguiente pérdida de rendimiento global.

5.3.2.3 Caracterización básica de la implementación con CUDA y OpenMP

Consiste en las mismas pruebas y metodología que en el anterior apartado 5.3.2.2.

Los resultados son muy similares a la versión anterior, sólo que un 3–5 % más rápidos, según la carga. La aceleración máxima obtenida en este caso es de 695 sobre el código original en CPU. Por mor de la brevedad, se omite la tabla y las gráficas de resultados, al no aportar mayor información, y ser redundantes con las de la primera implementación por su semejanza.

5.3.2.4 Pruebas masivas de escalabilidad de las implementaciones con CUDA

Tras observar el funcionamiento básico de la nueva implementación, y siendo ésta correcta y obteniendo resultados de aceleración muy importantes respecto al código original en CPU, se procede a realizar varias pruebas masivas a nivel de producción de los dos códigos presentados, al que sólo se le ha incorporado CUDA (referenciada como programa o versión r24, con Fortran+MPI+CUDA) y al que añade a la anterior OpenMP para gestionar dos hilos en cada proceso, uno encargado de la computación numérica en la GPU principalmente y otro responsable de las comunicaciones y la escritura en disco de los resultados (referenciada como r27).

Breve descripción de las pruebas

Las pruebas se realizan en el entorno de ejecución de MinoTauro. Se realizan diferentes lanzamientos variando la carga, para lo cual básicamente se mantiene constante el número de iteraciones temporales, fijándolas en 200.000 iteraciones (dado que llega un momento en que se alcanza el estado estacionario y no tiene sentido físico alguno continuar la simulación), y variar el número de partículas, desde muy pocas a muchas.

Es importante indicar que existe un límite para el número de partículas que se pueden lanzar en una única GPU. La tarjeta gráfica NVIDIA M2090, que dispone de 6 GB de memoria RAM, y con la actual implementación, sólo permite un máximo de unas 825.000 partículas. Para trabajar con un poco de margen, se escoge lanzar simulaciones con 500.000 u 800.000 partículas por cada GPU.

Presentan las siguientes características, lanzadas de forma ortogonal entre todas ellas:

- Carga de entre 1 millón y 50 millones de partículas, siempre con 200.000 iteraciones temporales.
- Se varía el número de GPU, desde el mínimo número posible por la capacidad en memoria de las tarjetas gráficas, que coincide con unas 825.000 partículas/GPU, hasta el máximo disponible en el *cluster* por política de utilización, que es de un máximo de 128 GPU.
- Se prueban las dos versiones r24 (con CUDA) y r27 (con CUDA más OpenMP).

Estas pruebas se componen de forma ortogonal, barriendo todas las posibilidades combinativas, pero por mor de la brevedad, sólo se mostrarán las más relevantes, sobre todo teniendo en cuenta que se comprueba que alguna de las variables inicialmente prevista no tiene en la práctica, en este entorno de ejecución, ninguna influencia estadística.

5.3.2.4.1 Resultados obtenidos

Una importante observación tras las pruebas, y concordante con lo visto con anterioridad, es que da prácticamente lo mismo lanzar `X` procesos en `X` nodos (utilizando sólo 1 GPU por nodo), que lanzar esos mismos `X` procesos en $\frac{X}{2}$ nodos (utilizando 2 GPU por nodo). Es por ello que en adelante sólo se mostrarán los datos correspondientes a la utilización de 2 GPU por nodo, excepto en el caso de las pruebas con 128 nodos, que por política del *cluster*, sólo podían ser lanzadas en 64 nodos.

En las Tablas 5.6 y 5.7 se muestran los resultados de los tiempos de ejecución de las diferentes pruebas. Las pruebas se realizan por duplicado, repitiéndose si se observa algún valor extraño, que generalmente se presenta como un valor extrañamente alto en el tiempo parcial de las comunicaciones. En las tablas se muestran el valor promedio de las ejecuciones realizadas, eliminando los valores extremos extraños, que suelen ser en torno al 10–15 % de las ejecuciones, especialmente las que se ejecutan con mayor número de GPU() o nodos.

Promedio de Tiempo	Partículas							
Procesos	Parte	1,000,000	2,000,000	4,000,000	8,000,000	16,000,000	32,000,000	50,000,000
2	GPU	1177.366						
	I/O	4.628						
	MPI	0.570						
	Otros	0.024						
	Total	1182.587						
4	GPU	597.950	1177.426					
	I/O	2.663	4.643					
	MPI	0.364	0.397					
	Otros	-0.065	-0.072					
	Total	600.912	1182.394					
8	GPU	315.947	597.928	1177.343				
	I/O	1.621	2.784	4.596				
	MPI	0.550	0.488	0.802				
	Otros	-0.091	-0.087	-0.006				
	Total	318.027	601.112	1182.736				
16	GPU	165.991	316.038	597.802	1177.352			
	I/O	1.126	1.621	2.618	4.636			
	MPI	0.447	0.503	0.460	0.470			
	Otros	-0.141	-0.089	-0.105	-0.076			
	Total	167.422	318.072	600.775	1182.382			
32	GPU	92.590	166.030	315.896	597.833	1177.577		
	I/O	0.848	1.133	1.634	2.621	4.598		
	MPI	0.751	0.536	0.557	0.629	0.570		
	Otros	-0.195	-0.149	-0.165	-0.149	-0.125		
	Total	93.994	167.550	317.922	600.929	1182.620		
64	GPU	54.031	92.574	166.073	315.856	597.692	1177.630	1861.650
	I/O	0.763	0.906	1.139	1.633	2.619	4.569	6.751
	MPI	0.773	0.889	0.719	1.238	9.418	0.716	8.898
	Otros	-0.299	-0.467	-0.223	-0.291	-0.253	-0.243	-0.169
	Total	55.267	93.902	167.707	318.436	609.475	1182.672	1877.130
128	GPU		54.140	92.571	166.113	315.868	597.822	954.424
	I/O		0.788	0.932	1.209	1.660	2.627	3.681
	MPI		11.897	23.777	49.865	101.112	204.925	355.542
	Otros		-0.588	-0.535	-0.618	-0.691	-0.734	-0.515
	Total		66.237	116.745	216.569	417.949	804.639	1313.133

Tabla 5.6: Tiempos de ejecución de las pruebas de producción para la versión r24 (CUDA+MPI)

Para poder observar mejor el comportamiento, para cada una de las implementaciones y tipo de simulación, se presentan también las siguientes gráficas:

Figura 5.5: Tiempos de ejecución en función del número de GPU, para cada una de las cargas (número de partículas), en escala doble logarítmica. Permitirá visualizar cómo escala el programa, puesto que lo ideal sería obtener una recta de pendiente la unidad.

Figura 5.6: Tiempos de ejecución en función del número de partículas de la carga, según el número de GPU utilizado.

Figura 5.7: Tiempos parciales de la parte de comunicaciones MPI en función del número de GPU y del número de partículas simuladas. Permite comprobar si la pérdida de escalabilidad se puede achacar principalmente a esta parte del programa, por lo que es interesante observar cómo varía también entre las diferentes implementaciones y tipos de simulación realizadas.

Promedio de Tiempo		Partículas						
Procesos	Parte	1,000,000	2,000,000	4,000,000	8,000,000	16,000,000	32,000,000	50,000,000
2	GPU	1127.802						
	I/O	3.932						
	MPI	1.416						
	Otros	5.009						
	Total	1138.158						
4	GPU	570.129	1127.719					
	I/O	2.413	3.817					
	MPI	0.573	0.884					
	Otros	5.479	5.422					
	Total	578.594	1137.842					
8	GPU	299.392	569.961	1127.621				
	I/O	1.759	2.446	3.700				
	MPI	0.576	0.626	2.696				
	Otros	5.536	5.528	4.884				
	Total	307.263	578.560	1138.901				
16	GPU	155.517	299.533	570.070	1127.730			
	I/O	1.243	1.671	2.334	3.776			
	MPI	0.471	0.396	0.499	0.552			
	Otros	5.533	5.711	5.700	5.987			
	Total	162.763	307.310	578.602	1137.905			
32	GPU	85.486	155.836	299.634	570.313	1127.866		
	I/O	0.945	1.263	1.675	2.397	3.748		
	MPI	0.750	0.546	0.702	0.540	3.493		
	Otros	5.460	5.475	5.831	5.578	5.938		
	Total	92.641	163.119	307.842	578.828	1141.045		
64	GPU	50.852	86.174	156.630	300.415	571.260	1128.100	
	I/O	0.939	0.977	1.281	1.796	2.399	3.830	
	MPI	0.733	0.628	0.619	0.588	3.734	9.993	
	Otros	4.704	5.360	5.354	5.510	5.554	5.661	
	Total	57.228	93.138	163.884	308.309	582.947	1147.586	
128	GPU		52.316	87.130	157.008	302.132	657.749	1141.044
	I/O		1.017	1.146	1.478	2.077	2.578	3.297
	MPI		5.944	19.578	45.918	111.002	365.214	801.091
	Otros		6.205	7.089	7.253	-8.189	-241.537	-670.985
	Total		65.481	114.941	211.656	407.021	784.003	1274.447

Tabla 5.7: Tiempos de ejecución de las pruebas de producción para la versión r27 (CUDA+MPI)

Figura 5.8 a) y b): Eficiencia temporal relativa del programa en función del número de GPU, para cada carga. Se ha denominado relativa porque no se utiliza como referencia el código original Fortran en CPU, sino que se compara con la ejecución que utiliza menos recursos. En este caso, se trata se compara con la ejecución en un único nodo con dos GPU, dado que la carga menor no puede ser ejecutada por una única GPU por falta de memoria.

Figura 5.8 c) y d): Eficiencia espacial relativa del programa, esto es, cómo se comportan los programas con el incremento de la carga, representada por el número de partículas, pues el número de iteraciones se mantiene constante en 200,000 en todas estas pruebas.

Figura 5.9 Comparación directa entre los tiempos de ejecución de las dos versiones con GPU presentadas (r24 y r27).

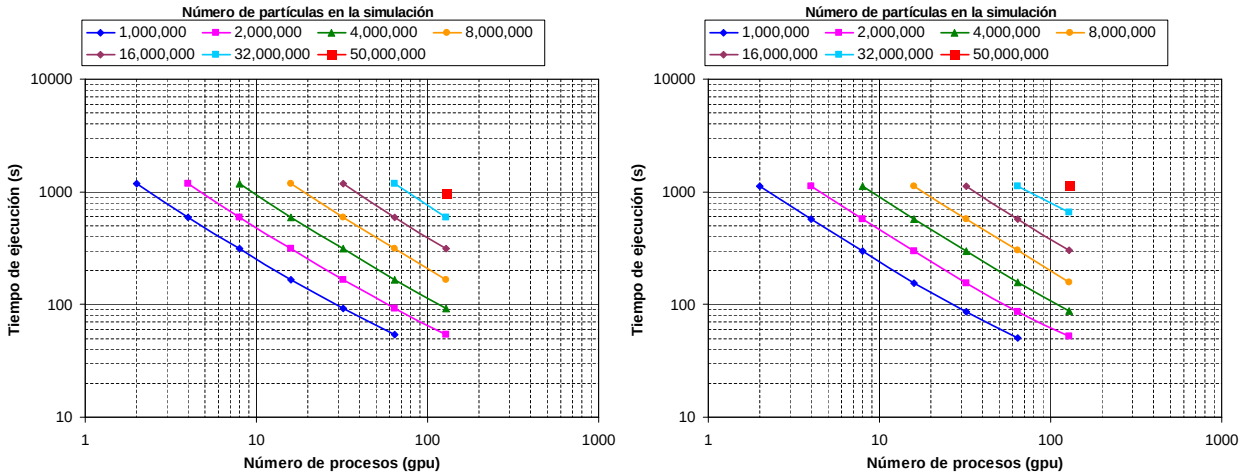
Es importante explicar por qué aparece el valor negativo en la parte de computación "Otros", que es

aquella que no está contabilizada ni como de cálculo en la GPU y su gestión, ni de transmisiones MPI, ni de escritura en disco duro (“I/O”). Aparece sobre todo en la versión con OpenMP, y especialmente cuando más procesos están ejecutando, y en las cargas mayores. Esto es debido a que hay dos hilos dentro de cada proceso, y el hilo que se encarga de la transmisión MPI contabiliza los tiempos de comunicación y también la de escritura en disco. Por su parte, el hilo que ejecuta los *kernels* cronometra el tiempo de GPU utilizado. El hecho de que la suma de la computación numérica más la de comunicaciones y escritura en disco es mayor al tiempo total de ejecución real de la aplicación demuestra que existe asincronía entre ambos conjuntos de operaciones.

Así se justifica también que el tiempo de ejecución global sea menor que en la versión r24, sin OpenMP, dado que allí están seralizadas la computación, la computación y la escritura en disco, de tal forma que cuando aumenta el tamaño del problema, o el número de nodos participando de la computación, aumenta la cantidad de datos a transmitir y a escribir en disco.

5.3.2.4.2 Escalabilidad de las implementaciones

Para estudiar la escalabilidad de las implementaciones presentadas nos serviremos de las Figuras 5.5 y 5.6, donde se muestran los resultados obtenidos enfocando la representación a un análisis espacial o temporal, respectivamente.



(a) Código r24 (CUDA+MPI)

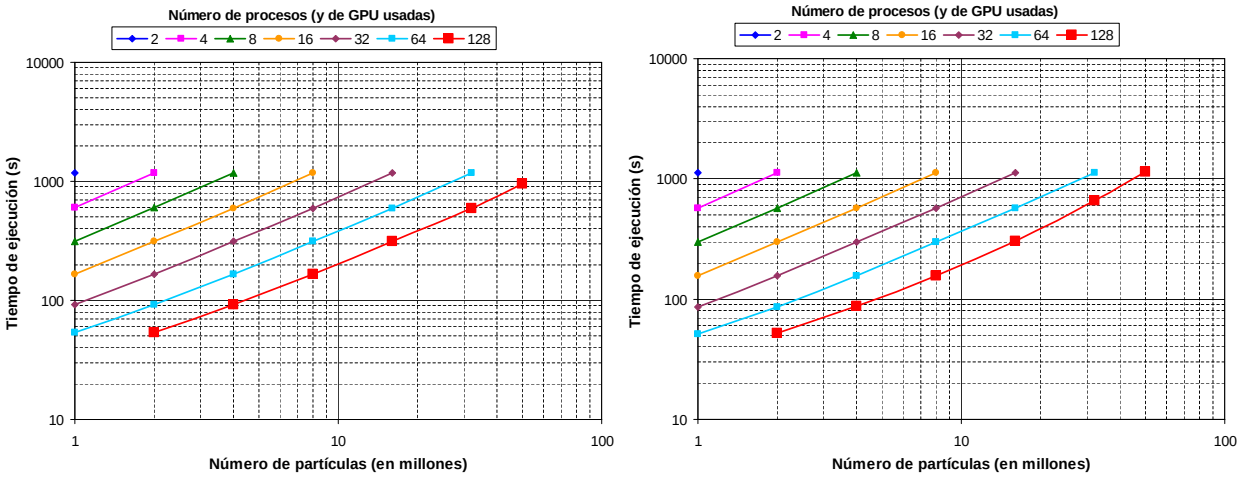
(b) Código r27 (CUDA+MPI+OpenMP)

Figura 5.5: Escalabilidad espacial de las implementaciones realizadas

De las Tablas 5.6 y 5.7 también se observa un súbito incremento del tiempo empleado por las comunicaciones con MPI cuando se llega a usar gran parte de nodos del *cluster* MinoTauro. Las Gráficas 5.7 muestran mejor ese fenómeno.

De todos estos datos se observa que:

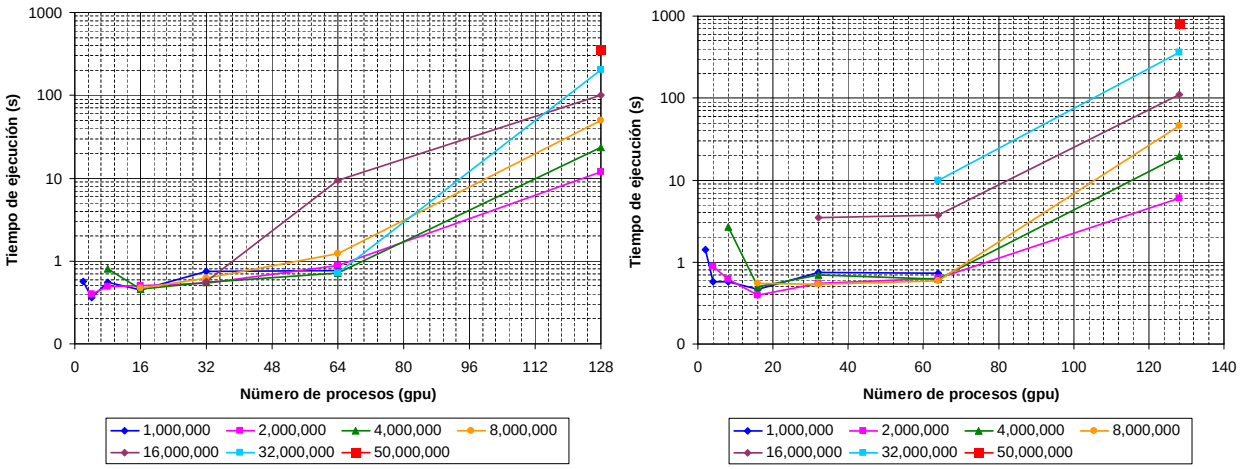
- El código con OpenMP se ejecuta en unos tiempos inferiores a la versión con sólo CUDA y MPI.
- Parece existir algún problema con la versión r27 con OpenMP cuando se trabaja con 64 nodos (128 procesos) y con más de 30.000.000 de partículas. No se ha analizado la causa, pues el fenómeno de pérdida de escalabilidad en esas condiciones no se reproduce en la versión r24. Una hipótesis podría ser que cuando en ese *cluster* se ejecutan simulaciones con la mitad de los nodos globales de



(a) Código r24 (CUDA+MPI)

(b) Código r27 (CUDA+MPI+OpenMP)

Figura 5.6: Escalabilidad temporal de las implementaciones realizadas



(a) Código r24 (CUDA+MPI)

(b) Código r27 (CUDA+MPI+OpenMP)

Figura 5.7: Variación del tiempo de comunicación con la carga y número de procesos

cluster, aparecen una serie de mensajes en OpenMPI indicando sobre conflictos en los puertos InfiniBand y de topología de red. Puede que esas situaciones de alto uso del *cluster* provoquen ciertos problemas de congestión esporádicos, que un poco por casualidad, hayan afectado a la versión r27 y no a la versión r24, ejecutadas en días diferentes. Pero por precaución, conviene tener este punto en consideración para un futuro análisis, y si es necesario, corrección.

- La red de interconexión es suficiente para el tráfico generado por las aplicaciones, y apenas afecta al tiempo global de ejecución. El tiempo de escritura, dado que el sistema de ficheros es paralelo, incluso disminuye con el incremento de nodos.
- Por su parte, el tiempo de MPI escala razonablemente, excepto cuando se usa la mitad de los nodos del *cluster*. Ese puede ser un problema de configuración de la topología, o incluso es probable que sea un problema de configuración de OpenMP cuando trabaja con InfiniBand, por el error que aparece:

Listado 5.1: Mensaje de error habitual al utilizar 64 nodos en MinoTauro

```
WARNING: There are more than one active ports on host 'nvb4', but the default
    subnet GID prefix was detected on more than one of these ports.  If these ports
     are connected to different physical IB networks, this configuration will fail
    in Open MPI.  This version of Open MPI requires that every physically separate
    IB subnet that is used between connected MPI processes must have different
    subnet ID values.
```

Para visualizar mejor la evolución de la escalabilidad comparada, espacial y temporal, de las propias implementaciones, en las Tablas 5.8 la evolución de la eficiencia conforme se va incrementando el número de nodos y el tamaño de la carga. El valor indicado se corresponde con la eficiencia relativa al tiempo de ejecución de la carga de 1.000.000 de partículas durante 200.000 iteraciones temporales simuladas en dos GPU. Referenciemos por 2 ese caso base considerado, i el caso en evaluación, N el número de partículas, p el número de procesos (y por lo tanto, de GPU) participantes y T el tiempo de ejecución. Entonces la eficiencia viene dada, en esta tabla, por

$$\text{Eficiencia} = \frac{T_2}{T_i} \cdot \frac{p_i}{2} \cdot \frac{N_i}{N_2}$$

En las Gráficas 5.8 se representan dichos datos.

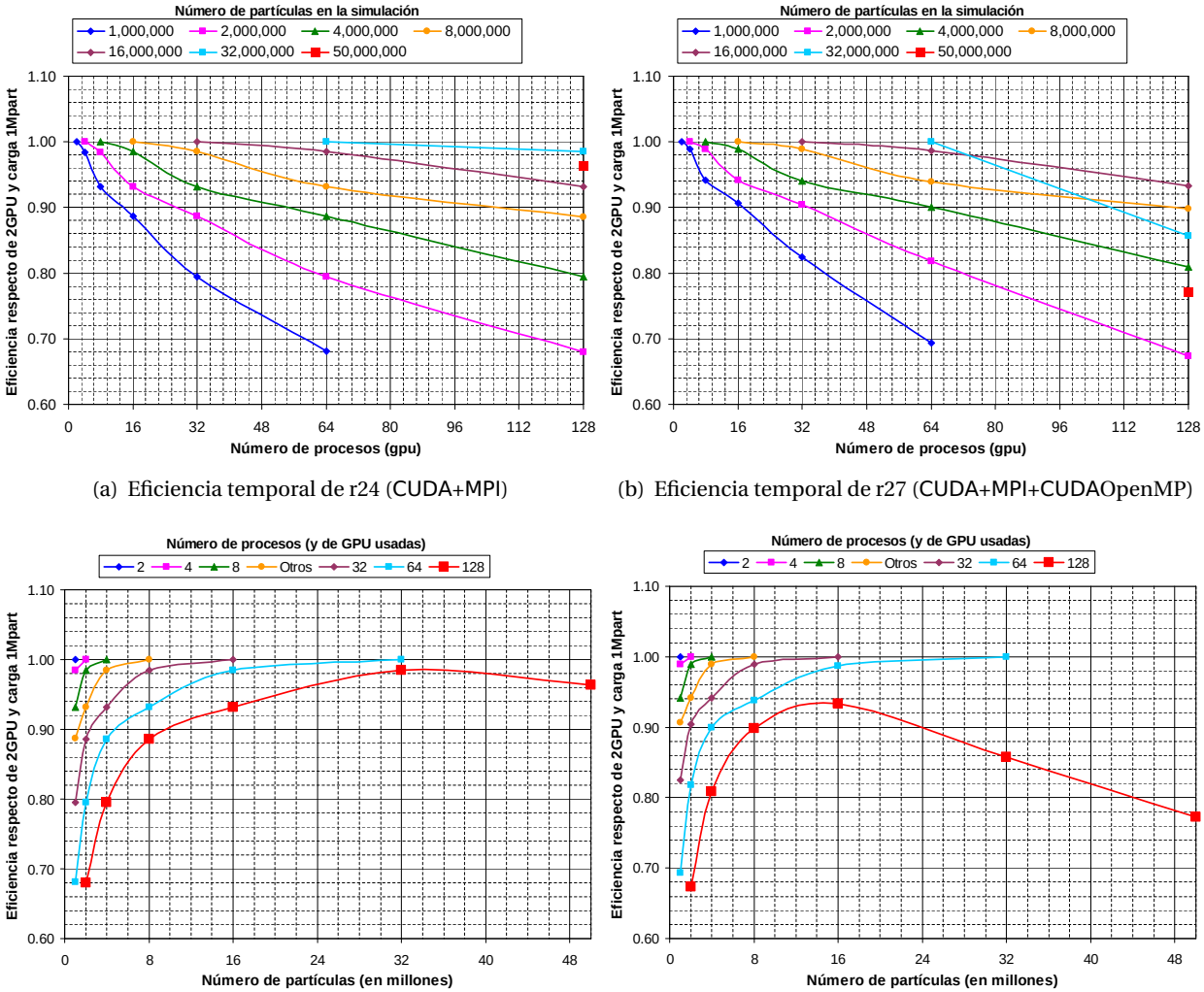
Eficiencia	Partículas (200.000 iteraciones)						
Nodos	1,000,000	2,000,000	4,000,000	8,000,000	16,000,000	32,000,000	50,000,000
2	1.000						
4	0.985	1.000					
8	0.932	0.985	1.000				
16	0.887	0.931	0.985	1.000			
32	0.795	0.886	0.932	0.985	1.000		
64	0.681	0.795	0.886	0.932	0.985	1.000	
128		0.680	0.795	0.886	0.932	0.985	0.964

(a) Código r24 (CUDA+MPI)

Eficiencia	Partículas (200.000 iteraciones)						
Nodos	1,000,000	2,000,000	4,000,000	8,000,000	16,000,000	32,000,000	50,000,000
2	1.000						
4	0.989	1.000					
8	0.942	0.989	1.000				
16	0.906	0.941	0.989	1.000			
32	0.825	0.905	0.941	0.989	1.000		
64	0.693	0.818	0.900	0.939	0.987	1.000	
128		0.674	0.809	0.898	0.933	0.857	0.772

(b) Código r27 (CUDA+MPI+OpenMP)

Tabla 5.8: Eficiencia relativa de las implementaciones con CUDA



(a) Eficiencia temporal de r24 (CUDA+MPI)

(b) Eficiencia temporal de r27 (CUDA+MPI+CUDAOpenMP)

(c) Eficiencia espacial de r24 (CUDA+MPI)

(d) Eficiencia espacial de r24 (CUDA+MPI+CUDAOpenMP)

Figura 5.8: Eficiencia relativa, temporal y espacial, de las implementaciones con CUDA

De estos datos se desprende que:

- La versión r27, con OpenMP, escala ligeramente mejor que la r24. Eso es explicable porque al estar desacopladas la computación de la comunicación y escritura, se minimiza el efecto negativo de su serialización, y cada parte puede escalar mejor, sobre todo con la buena red de interconexión del *cluster* MinoTauro.
- No obstante lo anterior, en estas gráficas se visualiza mejor si cabe el problema en la versión r27, con OpenMP, cuando se utiliza la mitad de los nodos de MinoTauro. En las observaciones anteriores ya se ha especulado por una posible razón, pero es un caso que conviene estudiar en un futuro.

5.3.2.4.3 Comparación directa de las implementaciones

Aunque en los anteriores apartados se ha podido comparar ambas implementaciones, dado que se han presentados los resultados conjuntamente, y ya ha sido parcialmente comentado en las observaciones y conclusiones realizadas, es interesante visualizar el rendimiento de ambas implementaciones. Para ello se

presenta la Tabla 5.9 y su Figura 5.9 correspondiente, en donde se presentan la aceleración obtenida si se compara la versión r27, con OpenMP, con la primera implementación r24, con sólo Fortran+MPI+CUDA.

r24/r27	Millones de partículas (200,000 iteraciones)						
GPU	1	2	4	8	16	32	50
2	1.044						
4	1.049	1.044					
8	1.055	1.049	1.044				
16	1.067	1.055	1.049	1.044			
32	1.083	1.065	1.054	1.048	1.044		
64	1.063	1.074	1.060	1.051	1.046	1.044	
128		1.035	1.062	1.058	1.045	0.909	0.836

Tabla 5.9: Aceleración obtenida al incorporar OpenMP (r27) a la primera implementación con CUDA (r24)

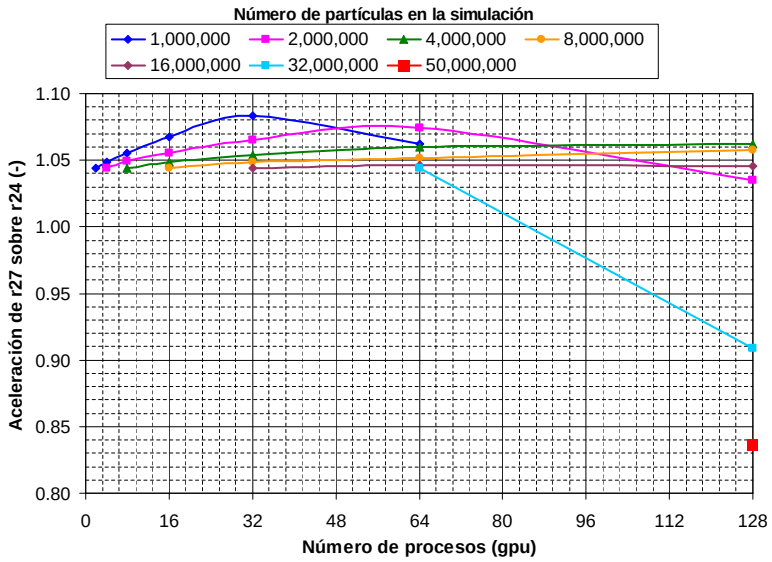


Figura 5.9: Aceleración obtenida al incorporar OpenMP (r27) a la primera implementación con CUDA (r24)

A la vista de estos datos, y de su representación visual, se observa que:

- En general, la implementación denominada r27, en la que se ha añadido OpenMP a la implementación inicial r24 con sólo CUDA aventaja a esta última en rendimiento
- Este rendimiento es como mínimo un 4,4% mejor, que se correponde con los casos en los que la GPU está más utilizada (cargas más grandes, las tabuladas en la diagonal). Esto es lógico, pues a más utilización de la GPU, para la misma carga, implica una menor cantidad de transferencias a realizar (más carga ya existente en el nodo maestro de MPI) y menor número y gestión de mensajes (menos nodos en la computación).
- El rendimiento aumenta considerablemente cuando las GPU están infrautilizadas (casos de la columna de la derecha, incrementando el número de procesos), por la razón inversa, esto es, porque hay más peso de la parte de comunicaciones sobre la de cálculo, con lo que hacer asíncronas las comunicaciones hace que se solape más tiempo en la ejecución global.
- De nuevo, hay dos casos especiales, que se dan en r27 cuando se utilizan la mitad de los nodos del

cluster y con las cargas de 32 y 50 millones de partículas. En esas ejecuciones, la versión con OpenMP se comporta mucho peor, pero está por determinar si es un problema específico y en cierta forma estocástico por la topología del *cluster* y la utilización combinada de la implementación OpenMPI de MPI sobre InfiniBand, o si es algún problema específico de la implementación r27.

5.3.2.4.4 Cálculo de la aceleración en una carga grande

Finalmente, y aunque en el apartado 5.3.2.1 "Caracterización del comportamiento del programa original", concretamente en la Tabla 5.5, ya se había establecido que para una carga pequeña ejecutada en local la aceleración máxima era de 680 para la versión r24, y de 695 cuando se utlizaba OpenMP (r27), es interesante calcular la aceleración respecto de una carga grande.

Dados los posibles problemas en las comunicaciones MPI del *cluster* al trabajar con la mitad de sus nodos, el cálculo se realizará sobre las versiones de 64 procesos (GPU) sobre 32 nodos con una carga de 32.000.000 millones de partículas simulando 200.000 iteraciones temporales. El patrón de comparación, dado que el código original de Fortran+MPI en CPU escala perfectamente, será el tiempo de ejecución de 12 procesos corriendo en 1 único nodo con una carga de 500.000 partículas computando 200.000 iteraciones temporales, cuyo tiempo de ejecución es de 66.987,859 segundos como se puede ver en la Figura 5.2.

La fórmula correspondiente para el cálculo de la aceleración se muestra a continuación, y está referida a un único núcleo o proceso computacional en la CPU, esto es, serían el número de núcleos necesarios en un entorno de memoria compartida para ejecutar la misma carga en el mismo tiempo. La nomenclatura es la misma que en la fórmula anterior, y el caso base, descrito en el párrafo anterior, se referencia como 12.

$$\text{Aceleración} = \frac{T_{12}}{T_i} \cdot \frac{p_{12}}{p_i} \cdot \frac{N_i}{N_{12}} \cdot \frac{it_i}{it_{12}}$$

Sustituyendo valores se obtiene:

$$\text{Aceleración máxima}_{\text{r24}} = \frac{66.987,859s}{1.177,630s} \cdot \frac{12}{64} \cdot \frac{32.000.000}{500.000} \cdot \frac{200.000}{200.000} = 682,6 \text{ CPU equivalentes}$$

$$\text{Aceleración máxima}_{\text{r27}} = \frac{66.987,859s}{1.128,100s} \cdot \frac{12}{64} \cdot \frac{32.000.000}{500.000} \cdot \frac{200.000}{200.000} = 712,2 \text{ CPU equivalentes}$$

Por lo tanto, se concluye que las implementaciones con GPU presentadas proporcionan una aceleración de aproximadamente 700 sobre el código de CPU original, medido con cargas grandes en un entorno de HPC, comparando un único proceso GPU con un único proceso CPU con código original.

Esto significa que con una máquina con un núcleo CPU y una GPU se obtiene el mismo resultado que con 700 núcleos CPU utilizando el código original.

Cierto es que las GPU utilizadas tienen internamente 512 núcleos CUDA, pero vienen todas integradas en un único componente de precio relativamente asequible. Aún así, para este tipo de códigos, un único núcleo interno de la GPU parece un 37 % más potente computacionalmente que un núcleo de la CPU.

No obstante, la limitación que tiene la GPU es en el número de partículas que como máximo puede tratar individualmente, al menos en estas primeras versiones presentadas (unas 825.000 partículas). Es otro aspecto del que convendría realizar en el futuro un análisis más detallado, puesto que podría ser que no se esté utilizando adecuadamente toda la capacidad de memoria interna real de la gráfica.

5.4 Valoración

5.4.1 Valoración numérica cuantitativa

Tras toda implementación de un programa, hay dos aspectos importantes a valorar:

- su corrección, esto es, si produce los resultados que de él se espera,
- su eficiencia, esto es, qué rendimiento se extrae de su utilización.

Dado que el objetivo era realizar una implementación utilizando CUDA de un código ya existente, paralelizado con MPI, los criterios para valorar la nueva implementación son evidentes:

- obtener los mismos resultados que el código original. Más concretamente, resultados equivalentes, por cuanto las ecuaciones de base son estocásticas, y por lo tanto, tienen un importante término aleatorio que impiden que incluso dos ejecuciones del mismo programa obtengan el mismo resultado,
- obtener una importante aceleración (*speedup*) que justifique el tiempo de desarrollo a invertir en nuevas parelelizaciones con CUDA de otro código científico preexistente. Cuanto mayor sea la aceleración, más fácilmente será amortizable el tiempo de aprendizaje y desarollo de la parelelización con CUDA, puesto que a la hora de su utilización se obtienen más beneficios por poder atacar problemas más grandes en menos tiempo, así como más ahorros económicos de producción, esto es, menor coste energético, y menor coste de amortización de los equipos, al ser necesario un número mucho menor de ellos.

Estas valoraciones se han realizado a lo largo de este capítulo, resultando ambas ser positivas:

- Las dos implementaciones proporcionan datos correctos, equivalentes al código original.
- Las dos implementaciones con CUDA proporcionan una aceleración en torno a 700 (una GPU ejecuta en el mismo tiempo una misma carga que la que ejecutaría 700 núcleos de CPU de procesador, ambas de MinoTauro)
- Eventualmente, un 10–15 % de las ejecuciones con CUDA, sobre todo cuando se utilizan 32 o más, sufren unas congestiones importantes en MPI, que afectan en el tiempo de ejecución hasta en un 30 % superior.
- Las implementaciones realizadas escalan relativamente muy bien cuando las GPU tienen mucha carga. De lo contrario, el peso del intercambio de mensajes se incrementa proporcionalmente mucho y puede decrecer el rendimiento hasta cerca de un 40 %.
- Por todas esas consideraciones, y desde un punto de vista meramente de inversión y de mantenimiento de un HPC, la paralelización con CUDA de programas científicos similares es altamente reccomendable económicamente.

Consiguientemente, la implementación se valora muy positivamente desde el punto de vista cuantitativo.

5.4.2 Valoración cualitativa de los resultados

Además de la valoración cuantitativa, y recordando que el objetivo de este trabajo no es tanto la paralelización de este código concreto, sino que esa labor ha servido de ejemplo práctivo para valorar la conveniencia

de la paralelización de otros códigos científicos históricos, se procede a valorar cualitativamente la experiencia del proceso de implementación, las decisiones que se han tomado en dicha implementación y otras alternativas que se descartaron al inicio del trabajo.

Como toda valoración cualitativa, ésta es subjetiva, por lo que a continuación se incluye se expresa como opinión formada a través de todo el proceso de realización del presente trabajo. Sin embargo, se considera muy interesante su expresión explícita en esta memoria porque es la parte quizá más interesante para el trabajo futuro, más que la mera perfección de las implementaciones concretas presentadas.

A continuación se presentan diferentes características que se consideran de interés.

5.4.2.1 Facilidad de programación

CUDA está basado en C y C++, y éstos ya son menos intuitivos de usar que Fortran para los científicos sin base específica de programación. CUDA representa otro paso más allá en la programación, pues se cambia el paradigma de programación. Por ello, la paralelización de código histórico en Fortran requiere:

- que una parte del grupo de científicos se especialice seriamente en programación paralela,
- que la tarea de programación sea encargada a un experto ingeniero informático.

Esta segunda opción sería la preferible, pero siempre bajo el asesoramiento de algún científico, para poder clarificar las fórmulas, algoritmos, y en definitiva, dar a entender al programador qué se está haciendo.

Por la razón anterior, CUDA es un lenguaje complicado de usar, aunque exprime a fondo las tarjetas gráficas de NVIDIA, muy potentes computacionalmente. Es interesante estar atento a otras alternativas, concretamente con el avance de OpenMP, versión 4, y sobre todo, de OpenACC. Ambos permiten el *offload* (envío de tareas de computación) a los aceleradores (principalmente, pero no exclusivamente, GPU, pero también a otros como Intel Xeon Phi o tarjetas gráficas de AMD) de una forma más intuitiva que CUDA. En este último trimestre de año se espera el lanzamiento de versiones ya finales de compiladores libres, como `gcc`, con soporte pleno a estas nuevas tecnologías. Se propone estar atentos a este hecho, y comparar la paralelización usando las GPU a través de OpenMP 4.0 y OpenACC.

5.4.2.2 Necesidad de la reformulación de los algoritmos

El gran *leitmotiv* de Fortran es su propio originario acrónimo: *FORmula TRANslation system*, hecho por el cual es tan ampliamente utilizado en el mundo científico. Pero también es uno de sus principales problemas. Al ser tan sencillo transcribir las fórmulas, los métodos matemáticos y los algoritmos en ellas externamente visualizados, es muy fácil caer en la programación directa de la fórmula, sin pensar en aspectos importantes desde el punto de vista de ingeniería del software:

- tener en cuenta los costes espaciales y temporales a la hora de implementar algoritmos (por ejemplo, clasificación en canastas, que puede hacerse con coste 1 por partícula, total $\mathcal{O}(N)$, siendo innecesario hacer previamente una ordenación total de los valores, con coste $\mathcal{O}(N \cdot log(N))$, y después recorrer otro bucle con coste $\mathcal{O}(N)$),
- implementar eficientemente según la arquitectura hardware y software disponible en el momento, (por ejemplo, estar actualizado en cuanto al avance de la tecnología de la paralelización, tales como OpenMP, OpenACC y CUDA),

- amplio conocimiento de las funciones y biblotecas disponibles en el entorno (ejemplo, utilización de la función error, `erf` en Fortran, C y CUDA), en lugar de calcularla cada vez mediante métodos numéricos de integración (trapecio o Simpson).
- actualización del programa base a las nuevas versiones de Fortran (por ejemplo, actualizar los programas desde Fortran 77 o Fortran 90 a Fortran 95 (*bindings* con C) o incluso Fortran 2003 (interoperabilidad con C y por lo tanto, más fácil trabajar con CUDA, y objetos, incluso aunque sean como meras estructuras de datos, para ocultación de variables actualmente declaradas globalmente)

5.4.2.3 Depuración semántica de las implementaciones históricas

Durante el trancurso del trabajo se detectaron lo que podían ser dos errores de programación que podrían influir de manera muy importante en los resultados que éste genera. A la fecha de hoy, el físico responsable de los códigos no ha confirmado ni negado esos informes de error enviados.

Los códigos históricos utilizando tecnologías de paralelización basadas en *clusters* de CPU tienen cierta limitación en cuanto al tamaño y, por lo tanto, validez, de las simulaciones que con ellos se pueden realizar. Ésa es la razón que fundamenta este trabajo, de hecho, el estudio de viabilidad de su conversión para la utilización de tarjetas gráficas de computación. Dado ese limitado poder de simulación, es posible que en los códigos históricos, aunque estén en constante evolución y revisión, persistan errores de base, difíciles de detectar en código y también en simulaciones a pequeña escala, pero que en simulaciones mayores, provoquen resultados mucho más divergentes con la realidad que lo razonablemente esperado. Esto llevaría a falsas conclusiones y generación de dudas sobre el propio cuerpo teórico científico en que se basan.

Es por ello que antes de cada paralelización, se revise completamente el código original, o incluso en mucho casos, que la implementación se realice desde cero para evitar de raíz este problema. Esto tiene la ventaja de que proporcionaría un código mucho más adaptado a las actuales tecnologías de programación paralela.

5.4.2.4 Selección de códigos a paralelizar

Finalmente, y es una obviedad, dadas todas las dificultades arriba destacadas, es siempre conveniente realizar un adecuado estudio de selección de qué códigos históricos deben ser paralelizados con GPU. La programación de GPU para GPGPU es delicada, laboriosa y difícil de depurar, pero su tecnología (hardware y software), potencial y viabilidad económica, no paran de crecer. Aún así, no todos los problemas aprovechan esta nueva tecnología.

Es una labor crucial tener a algún experto que asesore sobre la viabilidad de una implementación con CUDA u otras tecnologías emergentes (OpenACC, OpenMP 4), y de esta forma, dirigir los esfuerzos en paralelizar primero aquellas aplicaciones más interesantes pero que sean viables, tanto desde de el punto de vista tecnológico (rendimiento obtenido), como temporal (coste económico de la implementación).

(Página intencionadamente en blanco)

CAPÍTULO 6

TRABAJO FUTURO

La paralelización realizada de CUDA, con y sin OpenMP, sobre un código científico basado en Fortran con MPI, ha permitido comprobar que la aplicación del modelo SIMT a cierto tipo de códigos permite obtener unas prestaciones extraordinarias comparativamente con el código funcionando sobre CPU. También ha servido para detectar dificultades que deberán ser superadas para la aplicación de este procedimiento de paralelización a otros códigos científicos.

Las líneas de futuro a partir de este trabajo se dividen en tres categorías:

1. Ampliación del trabajo sobre las implementaciones en CUDA presentadas, con objeto de mejorar su estabilidad, rendimiento y aplicabilidad:
   - Depuración del programa, mediante detección de posibles *bugs*, limpieza de código y otras tareas básicas para que el código quede en un estado de mejor mantenibilidad.
   - Mejorar la ubicación de los cerrojos de `OMP`, que permitan, por ejemplo, disminuir la granularidad, y con ello, intentar extraer todavía más prestaciones basadas en un mayor solapamiento de la computación con las comunicaciones y la escritura en disco o consola.
   - Mejorar programáticamente las implementaciones actuales, por ejemplo, evitando la copia, dentro de la GPU de los vectores de velocidad nuevos sobre los viejos, mediante el sistema de rotar el puntero para realizar dichos cambios con copia cero, o investigando sobre la mejora de prestaciones en el uso de los generadores de números aleatorios.
   - Investigar por qué razón no se permiten ejecuciones de simulaciones de más de 825.000 partículas por cada GPU, cuando éstas parecen disponer de suficiente capacidad de almacenamiento (6 GB). Posiblemente a través de una mejor gestión de los diferentes tipos de memoria presentes en la tarjeta gráfica.
   - Reimplementar el código utilizando otras tecnologías de paralelización emergentes que también se aprovechan de los aceleradores gráficos, pero permiten también hacerlo sobre otros aceleradores, como sobre las tarjetas gráficas de AMD, o sobre todo, del acelerador Intel Xeon Phi. Ejemplos de estas tecnologías emergentes son OpenMP 4 y OpenACC, de los cuales se espera una implementación final libre en este último trimestre gracias a GNU.

2. Aplicación de la paralelización a otros códigos científicos
    - Migrar otros códigos similares a CUDA, u otras tecnologías emergentes, para aprovechar el extremo poder computacional de las GPU.
    - Modificar las ecuaciones base de Langevin obtenidas inicialmente, a través de la progresiva eliminación de algunas de las aproximaciones efectuadas para la deducción del conjunto de ecuaciones implementadas en el presente código. Esto debería permite obtener simulaciones más cercanas a la realidad.
    - A partir de esta experiencia, y viendo que es posible, añadir complejidad al cuerpo teórico inicial, por ejemplo, mediante la consideración de la relatividad dentro del conjunto de ecuaciones, para de esta forma acercarse más si cabe al comportamiento que tendrá el plasma cuando esté realmente en reacción de fusión controlada de forma continuada, bien sea en un reactor de tipo tokamak o del tipo stellarator.
3. Utilización a fondo de la nueva capacidad de simulación en este campo, y estudiar nuevas condiciones, variando los parámetros de entrada, realizando las oportunas mínimas variaciones en el código si es necesario, de tal forma que se convierta en una herramienta más para la comprensión del comportamiento real del plasma de partículas confinado magnéticamente.

CAPÍTULO 7

CONCLUSIONES

En el presente Trabajo Fin de Máster se ha estudiado la viabilidad y metodología para poder migrar código histórico científico, generalmente en lenguaje Fortran, al nuevo paradigma de programación mediante aceleradores externos, concretamente, mediante tarjetas gráficas de NVIDIA a través de su tecnología propia privativa CUDA. Para ello se ha utilizado como caso de estudio un código en Fortran ya paralelizado mediante MPI, que permite la simulación de un plasma de partículas a alta temperatura confinado magnéticamente, que es la física básica en la que se basan los actuales intentos de fusión en caliente en reactores de tipo tokamak.

Los resultados cuantitativos son plenamente satisfactorios: se han obtenido aceleraciones de hasta 700 en comparación con el código ejecutándose en un único núcleo de CPU, todo ello, obteniendo resultados correctos en la simulación y generando correctamente los ficheros intermedios estadísticos pertinentes. Esto representa grandes ventajas evidentes: permite la ejecución de simulaciones más grandes en un tiempo razonable, y disminuye el coste de explotación (número de horas) de utilización del *cluster* de computación de altas prestaciones.

Asímismo, se ha redactado una memoria que puede servir de guía sobre los problemas, dificultades, técnicas, planteamientos y decisiones que podrían surgir durante la realización a gran escala de la migración de código científico a las nuevas tecnologías de GPGPU.

A pesar de ello, la viabilidad de extender este caso de estudio a otros proyectos más ambiciosos no debe sólo ser informado por esta valoración muy positiva desde el punto de vista cuantitativo. Como ya se ha indicado al final de los Capítulos 4 y 5, hay otros aspectos cualitativos que sean tanto o más importantes a la hora de tomar la decisión:

- La migración es técnicamente viable y se han demostrado cómo realizarla. No obstante, no es trivial. El grado de complejidad del código resultado es elevado, sobre todo por la mezcla de diferentes lenguajes y tecnologías. Es por ello que deben concluirse se precisa una persona cona la suficiente formación en computación paralela y en ingeniería del software para poder enfrentarse con garantías y exitosamente al reto.
- A pesar de lo anterior, es muy importante destacar la importancia que tiene el significado físico de las fórmulas y algoritmos que implementan, así como el conocimiento del fenómeno físico estudiado, y

con ello, los resultados que a priori pueden obtenerse de la simulación. Es por ello que en el proceso de migración también debe estar presente un experto en la materia científica de que se trate, al objeto de guiar, comprobar y revisar que la implementación se ajuste a la teoría, esto es, efectúe una revisión semántica de los códigos.

- Con relativamente poco esfuerzo, se pueden conseguir resultados notables. Mejorarlos es también posible, pero muchas veces requiere de una inversión en tiempo y tecnología más elevado. Pero todo depende del caso concreto.
- Hay un aspecto que no se ha podido analizar, y es la existencia de compiladores específicos privativos de pago que permiten el uso de CUDA directamente sobre un dialecto o variación del lenguaje Fortran. Cabe la posibilidad de que su uso sea más sencillo que la solución propuesta, y que los requisitos del personal a cargo del proyecto sean también diferentes.
- No todo código científico histórico puede ser fácilmente paralelizable con CUDA, o incluso puede que sea contraproducente hacerlo. Se requiere un buen conocimiento de CUDA y del programa a migrar, para poder determinar si el nuevo paradigma de paralelización masiva de la GPGPU es adecuado al proyecto en cuestión. Como ejemplos, los métodos recursivos deben pasarse a iterativos. Tampoco se tolera la programación dinámica, aunque es poco común en el mundo científico. Los algoritmos con gran dependencia entre datos pueden presentar problemas de acceso a memoria que disminuyan mucho el rendimiento de las GPU.
- Es por ello que es necesaria una adecuada selección del código o algoritmo que va a ser migrado o implementado con CUDA, para extraer de esta tecnología todo lo que puede aportar.

GLOSARIO

buffer Espacio de memoria intermedia, que puede ser software o hardware.

bug En español, simplemente, error, aunque a veces se traduce literalmente como bicho, o gusano. Error, generalmente sutil, en el código que provoca el comportamiento anómalo del programa, incluso provocando su finalización abrupta. Especialmente peligrosos cuando sus manifestaciones son erráticas, dependiendo de alguna combinación determinada de condiciones para revelarse. Muy difíciles de depurar en computación paralela y distribuida.

cache Memoria rápida intermedia de pequeña capacidad para acelerar el acceso a una memoria principal de gran capacidad más lenta.

cluster Conjunto de ordenadores interconectados por una red de altas prestaciones para la ejecución conjunta de aplicaciones paralelas.

computación grid Entorno de ejecución HPC formado por varios *clusters* localizados geográficamente en ubicaciones muy dispersas y administrados por diferentes organizaciones que, no obstante, comparten sus recursos entre sí para lograr una mayor ocupación y utilización de sus instalaciones, ofreciendo una interfaz de lanzamiento común. Las ejecuciones deben ser planificadas y coordinadas, para lo cual existen gestores de colas especializados que pueden funcionar en base a créditos.

CUDA kernel Función cuyo código que se ejecuta en una GPU, y que por lo tanto de manera masivamente paralelizada.

daemon Apicación servidora que se inicia usualmente en el arranque del sistema operativo y que se encarga de gestionar un determinado servicio para el sistema operativo o las aplicaciones. Aunque normalmente se suele traducir al español por «demonio», en este trabajo se opta por el término «duende», término que parece más correcto, pues demonio tiene connotaciones malignas que no cabe esperar de un servicio legítmo arrancado por el sistema operativo, mientras que el término «duende» evoca más al efecto «mágico» de aquello que se ocupa de hacer algo por nosotros sin casi percibir esa ayuda.

disrupción Fenómeno brusco en un plasma en confinamiento magnético por el cual pierde sus condiciones de estabilidad y que provoca que en muy pocos milisegundos desaparezcan las condiciones necesarias para la fusión nuclear, apagándose el reactor como consecuencia.

HPC *High Performance Computing*, computación de alto rendimiento. Consiste en la ejecución de programas reales en entornos masivos de producción dotados de la última tecnología especialmente diseñados y gestionados con el fin de obtener las máximas prestaciones computacionales..

mapear Tecnicismo utilizado para describir la acción de referenciar un objeto o zona de memoria desde otra ubicación, generalmente con el objeto de virtualizar o facilitar el acceso directo al objeto original.

middleware Pieza de software que ofrece servicios a otras aplicaciones más allá de los que ofrece el sistema operativo, tales como permitir la mútua interacción o el intercambio de información. Es comúnmente utilizado en sistemas distribuidos, en donde un único sistema operativo no puede centralizar y gestionar toda la intercomunicación. Otro uso es facilitar al programador el acceso a diferentes recursos de entrada/salida, ya sean locales o remotos, y de tipo software o hardware.

name mangling Nomenclatura particular utilizada por cada compilador para cada lenguaje para la denominación de sus símbolos en los ficheros objeto `.o`.

plasma Estado de la materia, similar al gas, en el que por el efecto de las altísimas temperaturas, la materia se haya completamente ionizada, dada la alta velocidad térmica de las partículas.

profiling Investigación del comportamiento de un programa a través de la información recopilada desde el análisis dinámico del mismo. Existen programas específicos para tales tareas, siendo el más común `gprof`, dentro del software libre en Linux. NVIDIA pone a disposición dentro de su *Toolkit*, la utilidad `nvprof`. Aunque el término inglés está ampliamente difundido, puede traducirse por "perfilaje" o por "análisis del rendimiento"..

runtime Biblioteca de funciones que dan soporte para la ejecución de aplicaciones que utilicen un determinado lenguaje, en nuestro caso CUDA C.

speedup Aceleración. Es una métrica de la mejora relativa en prestaciones del rendimiento para la ejecución de una misma tarea. En el ámbito de la computación paralela, hace referencia al cociente entre el tiempo de ejecución de la tarea secuencial y el tiempo de ejecución del programa paralelizado.

stellarator Dispositivo donde se confina magnéticamente un plasma de partículas a alta temperatura con la finalidad de obtener la fusión nuclear controlada. Tiene la particularidad de que el campo magnético es tal que las órbitas de las partículas contenidas tiene una forma helicolidal.

tokamak Dispositivo similar al stellarator, pero en el que el campo magnético confina al plasma en una conjunto de órbitas de forma toroidal.

warp En español, urdimbre. Representa la agrupación de 32 ó 64 hilos de ejecución (según la arquitectura CUDA) que tienen un ámbito de ejecución común en una tarjeta gráfica de NVIDIA. Comparten recursos esenciales y escasos como registros y memoria *cache*, pero sobre todo, tienen un mismo flujo de ejecución, de tal forma que si éste es divergente para cada hilo, los hilos que no estén en esa bifurcación permanecen parados, y luego al revés, de tal forma que se acumulan los tiempos. Esto es así porque cada urdimbre sólo tiene activa una instrucción, la cual será o no ejecutada por el hilo que la componga. La urdimbre es, en la industria textil, el hilo longitudinal que se mantienen en tensión en un marco o tela, y que sirve de guía y estructura sobre la que se entretejen los hilos.

wrapper Función envoltorio, que ofrece una interfaz conocida para acceder a otra función o secuencia de funciones. Su uso puede estar motivado por diferentes causas, tales como facilitar el acceso a otras funciones, simplificando la forma de invocarlas, o de esconder el funcionamiento interno de un software por temas de seguridad o de protección de la propiedad intelectual.

SIGLAS

API *Application Programming Interface*, Interfaz de programación de aplicaciones.

ARPACK *ARnoldi PACKage*, Paquete de (método de) Arnoldi.

BLAS *Basic Linear Algebra Subprograms*, Subprogramas de álgebra lineal básica.

CPU *Central Processing Unit*, Unidad central de procesamiento.

CUDA *Compute Unified Device Architecture*, Arquitectura unificada de dispositivos de computación.

FORTRAN Anteriormente *FORmula TRAslation system*, Sistema de traducción de fórmulas. En la actualidad ya no se interpreta de esta manera, se considera una palabra propia.

GPFS *General Parallel File System*, Sistema de ficheros paralelo general (desarrollado por IBM.

GPGPU *General-Purpose computing on Graphics Processing Units*, Computación de propósito general en unidades de procesamiento gráfico.

GPU *Graphics Processing Unit*, Unidad de procesamiento gráfico.

ITER *International Thermonuclear Experimental Reactor*, Reactor experimental termonuclear internacional.

LAPACK *Linear Algebra PACKage*, Paquete de álgebra lineal.

Lis *Library of Iterative Solvers*, Biblioteca de resolvedores iterativos.

MPI *Message Passing Interface*, Interfaz de intercambio de mensajes.

OpenCL *Open Computing Language*, Lenguaje abierto de computación.

OpenMP *Open Multi-Processing*, Multiprocesamiento abierto.

PETSc *Portable, Extensible Toolkit for Scientific Computation*, Conjunto portable y extensible de herramientas para la computación científica.

ScaLAPACK *Scalable Linear Algebra PACKage*, Paquete de álgera lineal escalable.

SIMT *Single Instruction Multiple Threads*, Instrucción única, múltiples hilos. Modelo de ejecución en paralelo, utilizada por las GPU de NVIDIA con capacidad CUDA, entre otras, en la que múltiples hilos ejecutan concurrentemente la misma instrucción sobre diferentes datos.

SLEPc *Scalable Library for Eigenvalue Problem Computations*, Biblioteca escalable para computaciones sobre el problema de los valores propios o característicos.

SLURM *Simple Linux Utility for Resource Management*, Utilidad simple de Linux para la gestión de recursos.

UPV Universidad Politécnica de Valencia.

BIBLIOGRAFÍA

[1] *AMD accelerated parallel processing (app) sdk (formerly ATI Stream).* http://developer.amd.com/sdks/amdappsdk/, Sep. 2015.

[2] *AMD OpenCL zone.* http://developer.amd.com/zones/openclzone/, Sep. 2015.

[3] *CUDA gpus.* http://developer.nvidia.com/cuda/cuda-gpus, Sep. 2015.

[4] I. Fernández Gómez: *Generación y dinámica de electrones runaway en plasmas tokamak.* Tesis de Doctorado, Universidad Carlos III de Madrid, 2013.

[5] *Gpgpu: General-purpose computation on graphics hardware.* http://gpgpu.org/, Sep. 2015.

[6] T. D. Han y T. S. Abdelrahman: *Reducing branch divergence in gpu programs.* En *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, págs. 3:1–3:8, New York, NY, USA, Mar. 2011. ACM, ISBN 978-1-4503-0569-3.

[7] *Intel sdk for OpenCL.* http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/, Sep. 2015.

[8] *The khronos group inc. home page: Connecting software to silicon.* http://www.khronos.org/, Sep. 2015.

[9] E. Lindholm, J. Nickolls, S. Oberman y J. Montrym: *NVIDIA Tesla: A unified graphics and computing architecture.* IEEE Micro, 28(2):39–55, Mar. 2008, ISSN 0272-1732.

[10] H. H. Litz: *Improving the scalability of high performance computer systems.* Tesis de Doctorado, University of Mannheim, 182 págs., 2010.

[11] J. Nickolls y W. J. Dally: *The gpu computing era.* IEEE Micro, 30(2):56–69, Mar. 2010, ISSN 0272-1732.

[12] *NVIDIA OpenCL developer zone.* http://developer.nvidia.com/opencl, Sep. 2015.

[13] *OpenCL - the open standard for parallel programming of heterogeneous systems (specifications).* http://www.khronos.org/opencl/, Sep. 2015.

[14] *OpenGL: The industry's foundation for high performance graphics.* http://www.opengl.org/, Sep. 2015.

[15] *SNU-SAMSUNG OpenCL framework.* (http://aces.snu.ac.kr/Center_for_Manycore_Programming/SNU-SAMSUNG_OpenCL_Framework.html, Sep. 2015.

[16] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney y A. Shringarpure: *On the limits of gpu acceleration.* En *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, Berkeley, CA, USA, 2010. USENIX Association.