Evaluación comparativa de modelos de programación paralela en arquitectura de memoria compartida

Luis Miguel Sánchez [1] , Javier Fernández [2], Rafael Sotomayor [3] y J. Daniel García [4]

Resumen— **Hoy día, la mayoría de los computadores estándar incluyen características hardware que mejoran el rendimiento de los hilos paralelos de propósito general (hyper-threading, multi-core, ccNUMA) o kernels SIMD (GPUs, instrucciones vectoriales). El objetivo de este artículo es realizar una evaluación comparativa de los distintos modelos de programación paralela donde cada prueba se ajusta para explotar las características definitorias de estos modelos, pero también donde cada una requiere de un nivel diferente como programador de cada modelo. Se han elegido 4 modelos de programación: OpenMP, Intel TBB, Intel ArBB y CUDA. La idea es cubrir un rango más amplio de modelos de programación y de las características hardware para el paralelismo que se incluyen en computadores modernos. OpenMP y TBB explotan hilos paralelos en sistemas multi-core. Por otra parte, ArBB combina threads paralelos en multicore con características SIMD, y CUDA explota totalmente las características SIMD de las GPU. Los resultados obtenidos con estas pruebas sugieren que OpenMP y TBB tienen peor rendimiento que ArBB y CUDA. De estos dos, ArBB tiene a ser comparable con CUDA, aunque siempre es ligeramente inferior. Así, las pruebas parecen apuntar a que una arquitectura multi-core y multi-socket bien diseñada es comparable a tarjetas GPU equivalentes en cuanto a rendimiento para muchas aplicaciones, con la ventaja de que cuenta con un modelo de programación más simple.**

Palabras clave— **SIMD, ArBB, OpenMP, CUDA, TBB, Multicore.**

I. Introducción

CO n el tiempo, los ordenadores estándar han ido incluyendo diferentes características en sus arquitecturas para mejorar la programación paralela, y el rendimiento en general. En los últimos años se ha visto que múltiples arquitecturas han aplicado estrategias paralelas de todo tipo a todos los componentes posibles. Esta tendencia responde a un intento de mantener la misma tasa de mejora en el rendimiento que se había conseguido hasta ahora. Sin embargo, otras estrategias para la mejora de rendimiento, como aumentar la frecuencia del reloj, se han encontrado con límites muy difíciles de superar. Así, los computadores modernos se apoyan en distintos mecanismos de paralelización, como el paralelismo a nivel de instrucción (ILP), el paralelismo a nivel de hilos (TLP), y paralelismo a nivel de datos (SIMD) para alcanzar el nivel de rendimiento que se espera. El mayor inconveniente de esta solución es la imposibilidad de aislar su uso de la aplicación software. Esto es especialmente notable en las técnicas TLP y SIMD, siendo ILP la única excepción. Las arquitecturas que automatizan la delegación de trabajo paralelo partiendo de programas secuenciales han llegado a un punto en el que los resultados empeoran. Están limitadas por restricciones de energía, y de complejidad de diseño. La atención que se ha volcado recientemente sobre el descubrimiento de paralelismo en el software es el resultado de transferir la responsabilidad de arquitectos de procesadores a los desarrolladores de software. Por esta razón, términos como hyper-threading, many-cores, multicores y GPGPU se han convertido en moneda de cambio en la programación software, como antes lo fueran en la arquitectura de computadores. Así, la desventaja de esta estrategia es que los desarrolladores necesitan lidiar con la complejidad inherente que incorporar el software paralelo. Aún peor es el hecho de no todas las aplicaciones son susceptibles de ser paralelizadas. Recientemente se han presentado numerosas herramientas software y frameworks para facilitar el desarrollo de aplicaciones paralelas. El objetivo es conseguir el mejor rendimiento posible para computadores de uso no comercial. Cada framework cuenta con sus propias estrategias para lidiar con los susodichos problemas y, por esa razón, cada uno tiene sus ventajas e inconvenientes. La búsqueda del mejor modelo para el desarrollo paralelo aún es un tema en proceso de investigación. En este artículo se presenta una comparación de diferentes respuestas al desarrollo de aplicaciones paralelas pensadas para computadores estándar. Cada uno tiene su enfoque del problema. Así, el conjunto presenta una perspectiva de las ventajas y desventajas de esta clase de herramientas. Hoy día, un computador estándar puede incluir un número de características paralelas, como:

- Hyper-threading
- Múltiples núcleos por socket.
- Múltiples microprocesadores por placa.
- Organización de memoria ccNUMA
- GPUs de propósito general.

Cada framework se centra en optimizar al menos

[1]Área de arquitectura y tecnología de computadores, Dept. de Informática, UNIVERSIDAD CARLOS III DE MADRID e-mail: lmsan@arcos.inf.uc3m.es

[2]Área de arquitectura y tecnología de computadores, Dept. de Informática, UNIVERSIDAD CARLOS III DE MADRID e-mail: jfernandez@arcos.inf.uc3m.es

[3]Área de arquitectura y tecnología de computadores, Dept. de Informática, UNIVERSIDAD CARLOS III DE MADRID e-mail: rsoto@arcos.inf.uc3m.es

[4]Área de arquitectura y tecnología de computadores, Dept. de Informática, UNIVERSIDAD CARLOS III DE MADRID e-mail: jdaniel@arcos.inf.uc3m.es

una de estas características, a la vez que implementan un enfoque para facilitar la inclusión de algoritmos paralelos en los programas. Este artículo se organiza de la siguiente manera: La sección 2 describe múltiples arquitecturas estándar de computación paralela, así como modelos adecuados a las mismas. La sección 3 muestra las comparativas utilizadas para las evaluaciones. La sección 4 describe las plataformas hardware utilizadas, las metodologías aplicadas y el análisis del rendimiento obtenido en las pruebas realizadas. La sección 5 enumera trabajos relacionados. Finalmente, la sección 6 resume las conclusiones del artículo, y la sección 7 presenta trabajos futuros.

II. Arquitecturas y frameworks de computación paralela

Los computadores modernos basan su alto rendimiento en incluir tantas técnicas de paralelización como sea posible. Este enfoque aumenta enormemente el rendimiento teórico. Sin embargo, dicho rendimiento se consigue a costa de cargar ese paralelismo en el software. Hoy día la responsabilidad de conseguir un buen rendimiento recae en el software, porque los programas deben adaptarse para aprovechar las ventajas de las características paralelas que presenta el hardware. Existen dos perspectivas a considerar. La primera apoya el uso de plataformas multi-hilo de propósito general. Un procesador moderno permite esto de diversas formas. Primero, es posible la ejecución de múltiples hilos en un único núcleo (hyper-threading). Segundo, también es posible ejecutar múltiples hilos en núcleos diferentes dentro del mismo microprocesador que comparten la estructura de memoria (multicore). Tercero, se proporciona la posibilidad de ejecutar múltiples hilos en núcleos de distintos microprocesadores que no comparten la estructura de memoria (ccNUMA). La segunda perspectiva consiste en mantener el mismo código y aplicarlo sobre distintos datos utilizando unidades SIMD. A su vez existen dos métodos. Uno es utilizar las características SIMD que ofrecen los procesadores modernos como instrucciones vectoriales (Intel SSE). Esta solución se puede combinar con paralelismo a nivel de hilos. Otro método es utilizar una GPU dedicada al código SIMD. En este caso, se gestionarían cientos de hilos trabajando sobre distintos datos. Las CPUs y GPUs modernas cuentan, típicamente, con:

- Paralelismo SIMD explícito mediante ampliación de instrucciones vectoriales (SSE, AVX s[1], Intel Many-Core Architecture [2], etc.).
- Uso de técnicas multithreading simultáneas.
- Arquitecturas multicore en un único microprocesador, integrado en un sistema multisocket.
- Motores heterogéneos, con memoria, instrucciones y comportamiento diferenciados.

Por ejemplo, la arquitectura Intel Nehalem, sucesora del dual core, cuenta con tecnología hyper-threading, mejora las instrucciones vectoriales a SSE4 (aunque las unidades SIMD son iguales que en su predecesora), y soporta configuraciones cc-NUMA comunicando varios socket Nehalem con la interconexión Intel QuickPath. La sucesora de Nehalem es la arquitectura Sandy Bridge. Mejora principalmente en que las unidades SIMD son de 256 bits, en lugar de 128. El 10º microprocesador de la familia de AMD incluye quad-core, ccNUMA (mediante HyperTransport) y unidades SIMD de 128 bits e instrucciones SSE4. En las GPU, la arquitectura Nvidia GF100 cuenta con hasta 512 núcleos CUDA, distribuidos en hasta 16 procesadores de streaming (SM), cada uno con 32 procesadores simples (SP), 16 unidades de doble precisión (DP) y 4 unidades especiales (SFU) funcionando de 1.25 a 1.4 GH.La arquitectura SIMD se expone al programador como "warps" de hilos. Cada SM incluye 12+48 KB de memoria compartida/L1. La memoria global va de 3 a 6GB. A continuación se describen los lenguajes principales utilizados en entornos de memoria compartida:

1. OpenMP: Open Multi-Processing (OpenMP) [3] proporciona un API que, mediante directivas del compilador y llamadas a subrutinas, proporciona paralelismo de datos. La unidad base es el hilo, y cada uno tiene acceso a variables en caché compartida o RAM. Los mejores resultados se ven cuando el acceso a datos compartidos tiene bajo coste. Ofrece soporte en C, C++ y FORTRAN, aunque requiere de compiladores especiales.
2. Intel TBB: Intel Threading Building Blocks (TBB) [4]. Modelo de programación paralela basado en rutinas que utiliza hilos. Provee una serie de plantillas, tipos de datos y algoritmos. TBB está pensado de cara al rendimiento, por lo que es compatible con otras bibliotecas y paquetes. Permite configurar 3 tipos de organización (estática, guiada y dinámica). La organización se consigue mediante un enfoque de divide-y-vencerás implementada con robo de tareas.
3. Intel ArBB: Inter Array Building Blocks (ArBB) [5]. Biblioteca para C++ desarrollada para aprovechar diferentes tipos de procesadores en la resolución de problemas paralelos. Ofrece un modelo de programación basado en la composición dinámica de patrones estructurados de programación paralela. Mediante el uso de patrones paralelos, ArBB evita condiciones de carrera e interbloqueo. Son más sencillos de entender y depurar.
4. CUDA: Compute Unified Device Architecture (CUDA) [6]. Entorno de trabajo desarrollado por NVidia para unidades de procesamiento gráfico (GPU) NVidia. Se apoya en lenguajes conocidos, como C, aunque para alcanzar un alto rendimiento es necesario realizar optimizaciones manuales sobre la configuración y un amplio conocimiento de la arquitectura hardware de las GPU. CUDA se basa en tres abstracciones: jerarquías de grupos de hilos, memorias

compartidas y barreras de sincronización. Estas abstracciones están orientadas a la resolución de problemas mediante la división de los mismos en tareas que se pueden desarrollar de forma paralela. Un programa CUDA compilado es escalable, ya que solamente es necesario conocer el número de procesadores.

III. Benchmarks

Las arquitecturas y modelos de programación mencionaos anteriormente se evaluarán usando el siguiente conjunto de benchmarks:

A. Multiplicación de matrices

Como primer test, se ha escogido los programas SGEMM (*simple general matrix multiplication*) y DGEMM (*double general matrix multiplication*) La multiplicación de matrices en doble y simple precisión (SGEMM y DGEMM respectivamente) se utilizan en múltiples algoritmos de álgebra numérica. SGEMM se caracteriza por un patrón de acceso regular y, por tanto, de mapeo a arquitectura SIMD de manera directa. El uso de hilos es también muy sencillo. Basta con separar las matrices en subbloques de igual tamaño sobre los que pueden operar múltiples hilos de manera independiente. SGEMM trabaja a $O(n^3)$, donde n es la dimensión de la matriz, y realiza $O(n^2)$ accesos a los datos.

B. Convolución simple 2D

Se trata de una operación común de filtrado sobre imagen que se utiliza para efectos como el desenfoque. Sus cálculos aritméticos son simples operaciones de multiplicar-sumar y sus accesos a memoria son regulares, ya que se basa en la modificación de un elemento (pixel) basado en el propio valor y el del vecindario. Cada píxel se calcula de forma independiente, proporcionando así alto grado de paralelismo en las operaciones, tanto a el nivel SIMD como de hilo de proceso. El coste computacional depende en gran medida del tamaño del filtro o vecindario definido a tal efecto, lo que en la práctica supone un alto grado de dependencia de los accesos a memoria. Su ventana deslizante al estilo de patrón de acceso da lugar a problemas de alineación de memoria en los cálculos SIMD, ya que aun siendo regular, realiza saltos en memoria que pueden afecta a la caché del sistema.

En [7] se dispone de otras evaluaciones realizadas donde se comparan las mismas arquitecturas y tecnologías presentadas en el presente artículo.

IV. Evaluación

En esta sección se evalúa el rendimiento de diferentes arquitecturas en la multiplicación de matrices. Los detalles de las mismas se muestran a continuación.

A. Arquitecturas

La primera arquitectura es un Intel Core 2 Duo. Se utiliza como base para medir la aceleración del resto. Incluye un chip Core 2 Duo E7400 que contiene dos núcleos a 2.8 GHz, sin hyper-threading y con 4 GB de memoria. Otros dos computadores representan las arquitecturas unisocket con memoria uniforme. El primero es de la arquitectura Nehalem. Específicamente, incluye un chip Intel Xeon E5405 que contiene 4 núcleos a 2.0 GHz sin hyper-threading y 4 GB de momoria. El segundo cuenta con un chip Intel Core i7-2600 con 4 núcleos a 3.40 GHz con hyper-threading y 8 GB de memoria. Para evaluar la arquitectura ccNUMA se incluyen otros dos computadores. El primero es de la familia del 10º microprocesador de AMD. Incluye dos chips AMD Opteron 6168 [11] con 6 núcleos a 1.9 GH cada uno, todos ellos con hyper-threading y 64 GB de memoria. El otro pertenece a la arquitectura Nehalem [12] y cuenta con 4 Intel Xeon CPU E7-4807 con 6 núcleos a 1.87 GHz cada uno, todos ellos con hyper-threading y 128 GB de memoria.

La tabla I muestra las distintas plataformas hardware usadas para todas las evaluaciones.

Finalmente, la GPU utilizada pertenece a la familia NVidia GF100 (Fermi), concretamente una tarjeta NVidia c2050. Incluye 448 núcleos CUDA a 1.4 GHz. Se distribuyen en 14 SM con 32 SP cada uno. La memoria global es de 3 GB.

B. Metodología

Se prueba cada benchmark repetidas veces, y se comprueba que la desviación estándar es irrelevante. Se toma la media como valor representativo del test. La aceleración se representa tomando como base el rendimiento de la versión secuencial ejecutada en un núcleo de la arquitectua Intel Core 2 Duo. Para la codificación se han utilizado las multiplicaciones de ejemplo que proporciona NVidia, mientras que para el resto de modelos de programación, se han codificado por completo cada uno de los programas.

Es importante notar que para OpenMP se ha elegido la versión con mejores resultados cambiando el tipo de planificador. Por otra parte, para TBB se han utilizado los sistemas de reparto proporcionados por la biblioteca.

B.1 Análisis del rendimiento

En esta sección se presenta un breve análisis de los resultados de rendimiento obtenidos a partir de los puntos de referencia. Las cifras se han organizado para comparar los resultados obtenidos usando OpenMP/TBB por una parte y los obtenidos por ArBB/CUDA por el otro. La razón es que tanto OpenMP y como TBB utilizan paralelismo a nivel de hilo (TLP), mientras que ArBB y CUDA realizan para paralelismo a nivel de datos (DLP) en CPU y GPU respectivamente. En el caso de CUDA, se ha tenido en cuenta el tiempo de transferencia entre la memoria principal y la GPU.

Las gráficas 1 y 2 muestran los resultados obtenidos en el test SGEMM (*single precision*), mientras que las gráficas 3 y 4 muestran los resultados del benchmark DGEMM (*double preci-*

Especification CPU	Sockets	Cores por Socket	Hyperthreading	Memoria (GB)
Intel(R) Core(TM)2 Duo CPU E7400 @ 2.80GHz	1	2	No	4
Intel(R) Xeon(R) CPU E5405 @ 2.00GHz	1	4	No	4
Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz	1	4	Yes	8
AMD Opteron(tm) Processor 6168 @ 1.9GHz	2	6	Yes	64
Intel(R) Xeon(R) CPU E7-4807 @ 1.87GHz	4	6	Yes	128

TABLA I

ARQUITECTURAS USADAS EN LOS TEST

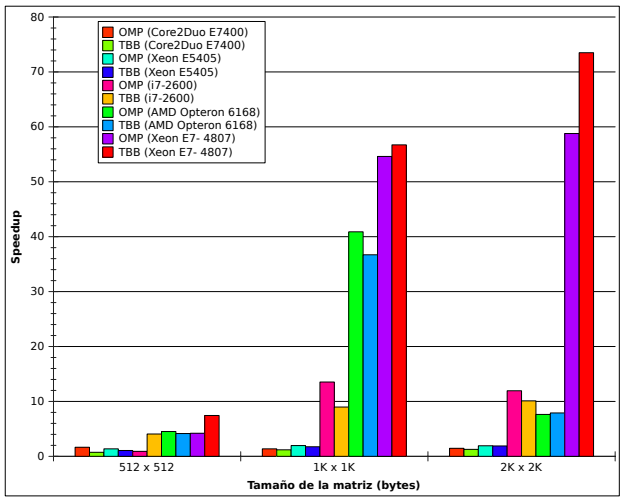

Fig. 1. SGEMM: Comparación entre OpenMP - TBB

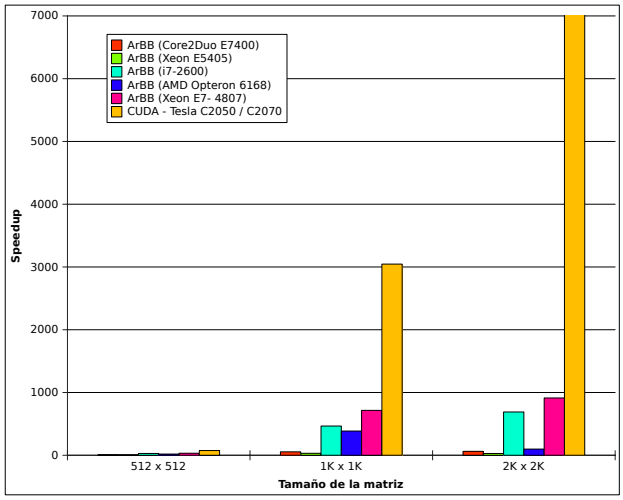

Fig. 2. SGEMM: Comparación entre ArBB - CUDA

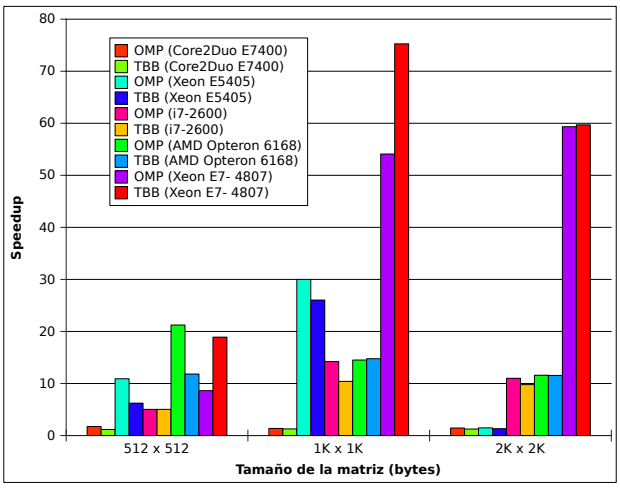

Fig. 3. DGEMM: Comparación entre OMP - TBB

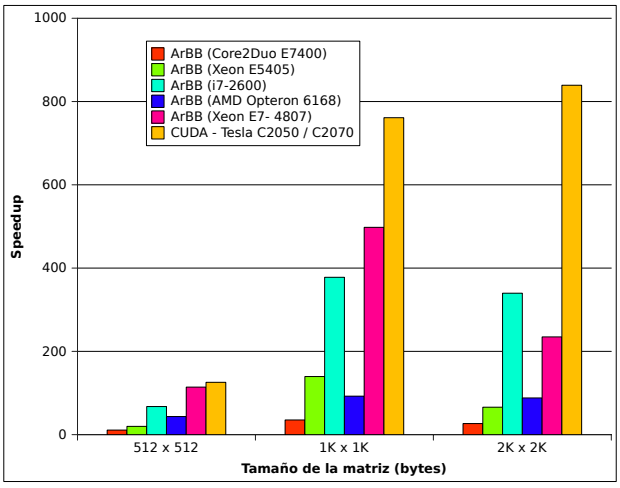

Fig. 4. DGEMM: Comparación entre ArBB - CUDA

sion). Como era previsible, los resultados obtenidos por OpenMP/TBB son muy inferiores a los de ArBB/CUDA para arquitecturas similares. Por esta razón se han realizado las comparativas en pares separados. En el caso de OpenMP/TBB, se puede apreciar que en genral los resultados de OpenMP son mejores a los de TBB, exceptuando en la arquitectura ccNUMA. Esto indica que TBB gestiona mejor los recursos de cómputo y memoria en arquitecturas ccNUMA. En el caso de ArBB, se puede observar que los mejores resultados se obtienen en las arquitecturas Sandy Bridge, superando a arquitecturas que utilizan un mayor número de cores. Pero más importante es que ArBB, en el caso del test SGEMM o de DGEMM, obtiene unos resultados comparables a los obtenidos por CUDA cuando la matriz es de pequeño o mediano tamaño (512x512). Así, el *speedup* de ArBB es 2/3 del obtenido por CUDA (en la arquitectura Sandy Bridge) en el caso de SGEMM. En el caso de DGEMM el uso de la arquitectura Sandy bridge hace que se acerquen mucho los resultados de ArBB y CUDA, siendo en el peor de los casos el rendimiento un 25% del de CUDA.

Las gráficas 5 y 6 muestran los resultados obtenidos usando el test de convolución simple. La tendencia es similar a los resultados obtenidos en el test anterior. Los resultados de OpenMP/TBB son peores que ArBB/CUDA exceptuando algun caso (concretamente con tamaños de datos pequeños). En el caso de ArBB, obtenemos resultados mucho mejores que los obtenidos por CUDA, en especial so-

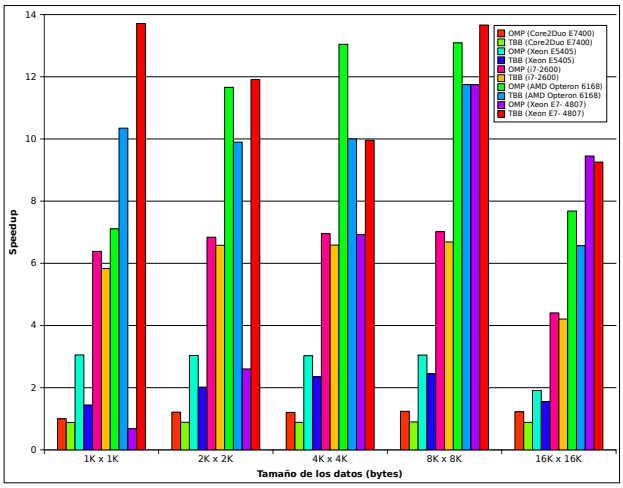

Fig. 5. Convolución simple: Comparación entre OMP - TBB

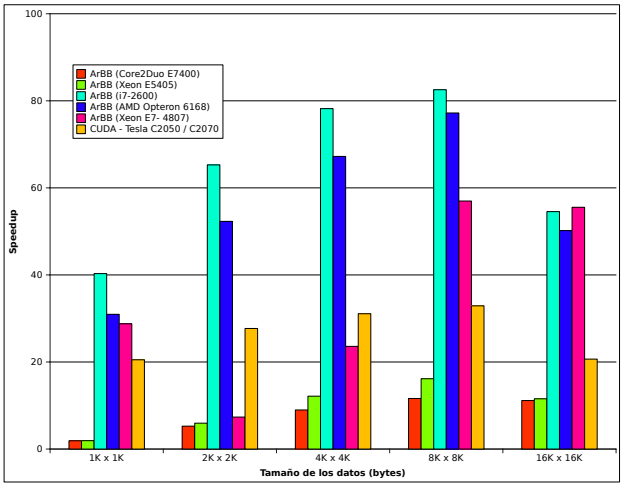

Fig. 6. Convolución simple: Comparación entre ArBB - CUDA

bre las arquitecturas Sandy Bridge y ccNUMA. Esto es debido a que ArBB gestiona mejor los datos de pequeño tamaño (1 byte) que CUDA.

Como resumen, se puede apreciar que el uso de ArBB puede ser muy adecuado para obtener mejores resultados en aplicaciones que utilicen CPU, en especial si hacen un uso intensivo de datos y realizan accesos regulares a memoria. En el caso de OpenMP/TBB, los resultados son en general peores que con ArBB/CUDA. Cabe indicar que OpenMP se comporta mejor que TBB en arquitecturas con un solo socket, mientras que TBB mejora su rendimiento simpre que se utilice sobre arquitecturas ccNUMA.

V. Trabajos relacionados

Existen numerosos trabajos donde se analizan tanto el rendimiento de TBB [8] como el de OpenMP [9] sobre distintas aplicaciones cientificas y de ingeniería. En [10], los autores estudian distintas implementaciones de aplicaciones para el analisis de imagenes médicas usando OpenMP y TBB, haciendo especial enfasis en las fases donde pueden existir bloqueos en las secciones críticas. Como resultado de este analisis, los autores concluyen que OpenMP se comporta algo mejor que TBB. Sin embargo otros autores, como [11] obtienen el resultado contrario. En este ultimo caso, un exhaustivo analisis del rendimiento y comportamiento de TBB explica que TBB tenga un comportamiento mejor en la busqueda de subcadenas a lo largo de un texto. Otros autores, como [12], realizan un estudio sobre el uso de la palarelización de un código usando taréas tanto de OpenMP como de TBB sobre arquitecturas ccNUMA, donde TBB tiene cierta ventaja al usar técnicas de *work-stealing* [13] que permiten mejorar el rendimiento gracias a que se explota el uso de la localidad de los datos.

El paradigma de computación GPGPU [14] se ha convertido en una importante tendencia en los últimos años. CUDA [15] y OpenCL [16] son los principales estandares en la programación GPGPU. En general, CUDA presenta un mejor rendimiento en comparación a OpenCL, como en [17]. Sin embargo otros autores [18] indican que la comparación realizada no se hace en las mismas condiciones, por lo que no se puede asegurar taxativamente que OpenCL tenga un rendimiento peor.

Los nuevos supercomputadores incluidos en el top500 [19] han incorporado el modelo GPGPU en su tecnología para incrementar el rendimiento. Sin embargo, estas arquitecturas presentan problemas de eficiencia y de consumo electrico, tal y como se indica en [20]. Por estas razones, el estudio de las nuevas arquitecturas de CPU, que utilizan instrucciones de vectorización [21] se ha convertido en un punto de especial interes en la computación paralela [22] [1].

Por último, otras investigaciones como [23] tienen un especial interés en el uso de los recursos de computación conjunto (CPU + GPU) de forma transparente al programador. Esto es posible incluyendo directivas en el lenguaje, de forma similar a OpenMP.

VI. Conclusiones

Este trabajo presenta un resumen de varios modelos de programación paralela que se despliegan fácilmente en las principales arquitecturas paralelas. El documento incluye una evaluación de los diferentes modelos de programación paralelos utilizados: OpenMP y TBB que realizan una programacióón basada en el uso de de los conceptos de multi-thread y multi-tarea, ArBB que emplea las instrucciones de vectorización disponibles en las CPU (modelo SIMD) y CUDA que realiza programación sobre GPU (GPGPU) usando capacidades SIMD.

Las evaluaciones de estas tecnologías a través de una variedad de arquitecturas de memoria compartida han demostrado varios resultados, como la mejora significativa en el rendimiento obtenido por las aplicaciones que, usando ArBB, utilizan las nuevas instrucciones vectoriales incluidas en la arquitectura Intel Sandy Bridge. ArBB combina programación multi-thread y SIMD en un único modelo de programación, lo que facilita la labor a los programadores realizando desarrollos más ágiles. Por lo tanto, la combinación de ArBB y la arquitec-

tura Sandy Bridge parece tener un rendimiento que se puede aproximar en situaciones muy concretas al obtenido usando CUDA, especialmente cuando el tamaño de los datos es de pequeño o mediano tamaño (las matrices de hasta 512 x 512 elementos). Sin embargo, la eficiencia de ArBB disminuye significativamente en procesadores que no utilizan instrucciones de vectorización AVX, así como en las arquitecturas ccNUMA.

Otro resultado importante es que mientras TBB parece tener un rendimiento inferior OpenMP para arquitecturas que disponen de un unico socket, la situación cambia cuando se ejecuta en arquitecturas ccNUMA, donde muestra una mejora significativa. Esto es debido al modelo especulativo del robo de tareas usado en TBB [13].

En resumen, las aplicaciones que se aprovechan de las características de TLP y SIMD proporcionados por los últimos modelos de CPU, al igual que las aplicaciones ArBB, parece acercarse a rendimiento obtenido por CUDA, además de la ventaja de utilizar una plataforma de programación donde el usuario se abstrae de las complejidades del hardware.

VII. Trabajos futuros

Como próximos trabajos futuros, pretendemos continuar el estudio de tecnologías como ArBB o Cilk [24] en otro tipo de arquitecturas que incluyan instruciones vectoriales en sus procesadores (como por ejemplo la nueva gama Ivy bridge).

Otro foco de atención está puesto en el uso de estas tecnologías en arquitecturas ccNUMA, proponiendo la integración de tecnicas de multi-thread (como TBB) con ArBB.

Por último, proponemos el uso combinado de computación SIMD entre CPU y GPU, lo que puede incrementar en rendimiento en aplicaciones cuyo comportamiento puede verse beneficiado al usar ambos modelos al mismo tiempo.

Agradecimientos

El presente trabajo ha sido financiado mediante el proyecto "Técnicas escalables de E/S en entornos distribuidos y de computación de altas prestaciones". Ministerio de ciencia e innovación, TIN2010-16497

Referencias

[1] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1816021

[2] Intel. (2012, Jan.) Many integrated core (mic) architecture - advanced. [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html

[3] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[4] J. Reinders, *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[5] C. J. Newburn, B. So, Z. Liu, M. D. McCool, A. M. Ghuloum, S. D. Toit, Z.-G. Wang, Z. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, "Intel's array building blocks: A retargetable, dynamic compiler and embedded language," in *CGO*, 2011, pp. 224–235.

[6] R. Farber, *CUDA Application Design and Development*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[7] L. M. Sanchez, J. Fernandez, R. Sotomayor, and J. D. Garcia, "The 10th ieee international symposium on parallel and distributed processing with applications," in *A Comparative Evaluation of Parallel Programming Models for Shared-Memory Architectures*, 2012.

[8] I. TBB. (2012, Jan.) IntelÂ® threading building blocks. [Online]. Available: http://threadingbuildingblocks.org

[9] OpenMP. (2012, Jan.) Open multi-processing api specification for parallel programming. [Online]. Available: http://openmp.org/wp

[10] P. Kegel, M. Schellmann, and S. Gorlatch, "Using openmp vs. threading building blocks for medical imaging on multi-cores," in *Euro-Par 2009 Parallel Processing*, ser. Lecture Notes in Computer Science, H. Sips, D. Epema, and H.-X. Lin, Eds. Springer Berlin / Heidelberg, 2009, vol. 5704, pp. 654–665.

[11] A. Marowka, "On performance analysis of a multithreaded application parallelized by different programming models using intel vtune," in *Parallel Computing Technologies*, ser. Lecture Notes in Computer Science, V. Malyshkin, Ed. Springer Berlin / Heidelberg, 2011, vol. 6873, pp. 317–331.

[12] M. Wittmann and G. Hager, "Optimizing ccnuma locality for task-parallel execution under openmp and tbb on multicore-based systems," *CoRR*, vol. abs/1101.0093, 2011.

[13] A. Robison, M. Voss, and A. Kukanov, "Optimization via reflection on work stealing in tbb." in *IPDPS*. IEEE, 2008, pp. 1–8.

[14] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. KrÃ$\frac{1}{4}$ger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2007.01012.x

[15] N. CUDA. (2012, Jan.) Compute unified device architecture. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[16] OpenCL. (2012, Jan.) Open computing language. [Online]. Available: http://www.khronos.org/opencl

[17] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of cuda and opencl," *CoRR*, vol. abs/1005.2581, 2010.

[18] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of cuda and opencl," *The 40th International Conference on Parallel Processing ICPP11 Taipei Taiwan*, 2011. [Online]. Available: http://www.pds.ewi.tudelft.nl/pubs/papers/icpp2011a.pdf

[19] Top500. (2011, Nov.) Supercomputer sites. [Online]. Available: http://top500.org/lists/2011/11

[20] V. Kindratenko and P. Trancoso, "Trends in high-performance computing," *Computing in Science and Engineering*, vol. 13, pp. 92–95, 2011.

[21] Intel. (2012, Jan.) Advanced vector extensions (avx). [Online]. Available: http://software.intel.com/en-us/avx

[22] N. G. Dickson, K. Karimi, and F. Hamze, "Importance of explicit vectorization for cpu and gpu software performance," *J. Comput. Physics*, vol. 230, no. 13, pp. 5383–5398, 2011.

[23] C. Augonnet, S. Thibault, R. Namyst, and P.-a. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *EuroPar 2009 Parallel Processing*, vol. 23, no. 2, pp. 187–198, 2009. [Online]. Available: http://www.springerlink.com/index/h013578235633mw3.pdf

[24] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, Aug. 1995. [Online]. Available: http://doi.acm.org/10.1145/209937.209958