UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO DE FIN DE GRADO

Programación paralela basada en coprocesadores many-core

Autor: Pedro Alberto Ruiz González
Tutor: Víctor Moreno Martínez
Ponente: Francisco J. Gómez Arribas
Julio 2015

Programación paralela basada en coprocesadores many-core

Autor: Pedro Alberto Ruiz González
Tutor: Víctor Moreno Martínez
Ponente: Francisco J. Gómez Arribas

High Performance Computing and Networking
Dpto. de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Resumen

Este trabajo de fin de grado explora el empleo de la programación paralela mediante el uso de coprocesadores, realizándose un pequeño repaso de los diferentes modelos de la misma y los principales coprocesadores que existen en el mercado.

Dentro del contexto de la computación mediante coprocesadores, en primer lugar, este trabajo hace un repaso sobre HPC (High Performance Computing) o computación de alto rendimiento y se hace un estudio sobre la evolución de arquitecturas en los superordenadores en las últimas décadas. Tras ello, se realiza un repaso sobre los dos principales modelos de programación paralela, MPI (Message Passing Interface) y OpenMP (API para la programación mediante hilos), haciendo hincapié en las peculiaridades de cada modelo.

La principal motivación del presente trabajo es la comprensión de la arquitectura y programación acerca de la familia de coprocesadores desarrollados por la empresa Tilera. Para poner al lector en contexto, se presenta el cómputo basado en coprocesadores o aceleradores hardware, mencionando los principales coprocesadores existentes en el mercado, analizándose las principales virtudes de cada uno. Tras ello, nos centramos en el coprocesador de la empresa Tilera TILE-Gx36.

Adicionalmente, se analizan los diferentes aspectos a nivel de arquitectura, así como latencias de acceso a memoria dependiendo de la dirección de la misma o de la ubicación física del procesador que se ejecute para, posteriormente, llevar a cabo una comparación de los diferentes procesadores que tiene la propia tilera de manera individual.

Por último, se tratan problemas con requisitos computacionales reales y se comparan las latencias de procesamiento con otras arquitecturas utilizadas en la programación paralela, comparando la aceleración con la potencia energética necesaria para la misma y el mismo problema en serie sobre un único ordenador.

Palabras clave

Coprocesador, MPI, OpenMP, tile, clúster, computación paralela, multicore, MiCA, TFLOPS.

Abstract

This end of career report explores the use of the parallel programming by means of using coprocessors, and to achieve a little checkup of the various models of it and the main coprocessors that may exist in the market.

Inside the world programming through coprocessors, firstly, this report does a checkup over HPC (High Performance Computing) and make a study through the evolution of the architectures in the supercomputers in the last decades. After that, it is fulfilled a checkup over the two main models of parallel programming, MPI (Message Passing Interface) and OpenMP (API for wire programming), insisting on the special features of each model.

The main motivation of this report is the comprehension of the architecture and the programming about the family of coprocessors which have been developed by the company Tilera. To bring the reader into the context, it is presented the calculation based upon coprocessors or hardware accelerators, referring to the main existing coprocessors in the market, analyzing the main advantages of each one. After that, we devote oneself to the coprocessor of Tilera company TILE-Gx36.

Additionally, in this report are reviewed the different aspects from the point of view of architecture, as well as the entry latency to memory depending of its direction or the physical position of the computer which can carry out to, later, to fulfill one comparison from different processors which own Tilera itself in an individual way.

Lastly, it is dealt problems with real computers requirements and are compared as well processing latencies with other used architectures in the parallel programming, comparing the acceleration with the needed energetic power and the same problem in mass over one only computer.

Key Words

Coprocessor, MPI, OpenMP, tile, cluster parallel computing, multicore, mica, TFLOPS.

Agradecimientos

Me gustaría agradecer en primer lugar a mi tutor Víctor Moreno, por la ayuda en el desempeño de esta gran tarea de aprendizaje.

A la Universidad Autónoma de Madrid, por darme la oportunidad de poder trabajar con ellos y prestarme el equipo necesario, así como el apoyo para solucionar los problemas surgidos en el mismo.

Agradecer a los compañeros y profesores con los que me he cruzado a lo largo de la carrera. Gracias a los que he adquirido la formación que tengo hoy en día y con los que espero seguir aprendiendo.

No solo ha sido un trabajo mío, ha sido también de aquellos que han estado apoyándome en el camino. Comenzaría dando las gracias a mi familia, que incondicionalmente me ha sabido dar fuerzas y apoyo de todas las maneras posibles. Económicamente y moralmente han sido el gran pilar en estos años de trabajo, en los cuales he tratado de mejorar día a día, sin olvidar que siempre se aprende algo nuevo y que no debemos rendirnos nunca.

A Marta, que a pesar de su reciente llegada, ha entrado en mi vida pisando fuerte. Ella ha sabido mostrarme otros puntos de vista y la visión positiva en los peores momentos. Creo que siempre estaré en deuda con lo que has aportado a mi vida y a mi forma de ser. Me gustaría mantener siempre el ánimo, la fuerza, la confianza y el cariño que tú me has enseñado.

Agradecerte a ti, Daniel mi gran amigo de la infancia la fe que has depositado siempre en mí. El apoyo y la fuerza que me has sabido dar, en especial el no dejar que olvidara que podía con todo y más. Gracias por creer en mí más incluso que yo mismo.

Por último, pero no menos importantes, gracias a todos mis amigos por esos momentos inolvidables tanto buenos como malos. Esos exámenes y prácticas, esas risas y horas de comidas en la cafetería, por ese sufrimiento en las revisiones de exámenes y en especial por formar parte de un gran grupo.

Tabla de contenido

Índice de figuras

Índice de códigos

Índice de tablas

Índice de gráficos

Glosario

- **Latencia:** Suma de retardos temporales dentro de una red, producido por la demora en la programación y transmisión de paquetes.

- **CPU (Central Processing Unit):** Unidad central del procesamiento.

- **FLOPS (Floating Point Operations Per Seconds):** Medida de rendimiento, número de operaciones en coma flotante por segundo.

- **MIPS:**
  - **Arquitectura (Microprocessor without Interlocked Pipeline Stages):** Micropocesador de arquitectura RISC.
  - **Medida de Rendimiento:** Millones de instrucciones de enteros por segundo.

- **RISC (Reduced Instruction Set Computing):** Diseño de CPU con instrucciones de tamaño fijo y solo de acceso a memoria las instrucciones de carga y descarga.

- **VLIW (Very Long Instruction Word):** Arquitectura de CPU caracterizada por el uso de instrucciones de tamaño grande.

1. Introducción

Debido a la constante evolución en los sistemas de procesamiento, cada vez se pueden resolver en un menor tiempo problemas más costosos. A pesar de ello, el tamaño de muchos problemas supera las capacidades de computación que poseemos. Como solución a ello, aparece la computación paralela, una técnica de trabajo computacional en la que se realizan varias tareas a la vez, consiguiendo con ello una aceleración del proceso mediante la división del mismo para su posterior ejecución en diferentes maquinas o procesadores, agrupando finalmente los resultados parciales en uno final. No obstante, el uso excesivo de procesadores/ordenadores supone un coste ya sea energético, espacial o monetario de los mismo aparatos. Esto provoca un descenso en la relación productividad/coste llevando al dilema de cuanto coste justifica la aceleración obtenida en el uso de la programación en paralelo.

Para ello, existen diversas soluciones para el trato de estos problemas. Por un lado Amdahl propone la división de una tarea de alto coste en múltiples tareas de menor coste; por otro lado, la solución propuesta por Gustafson consiste en aumentar el tamaño del problema, manteniéndose el tiempo de cómputo, incrementando la cantidad de trabajo realizado mejorando así el rendimiento. Estas leyes establecen los límites del paralelismo.

Un ordenador actual arquetípico utiliza una arquitectura multi-core, apoyándose en una tarjeta gráfica (coprocesador) para el tratamiento de imagen. A lo largo de este trabajo de fin de grado se estudiarán diferentes coprocesadores para la programación paralela, tomando como ejemplo para un estudio detallado un coprocesador poco conocido fabricado por la empresa Tilera. Concretamente, se profundizará acerca de su arquitectura, diferentes funcionalidades y se realizará una comparación con otras arquitecturas paralelas viendo su rendimiento.

1.1. Motivación

La computación de alto rendimiento es algo necesario en el mercado de las comunicaciones, seguridad etc. Sin embargo, el empleo de computadoras de alto rendimiento ha de realizarse con cautela debido a los sobrecostes de adquisición y operación (consumo eléctrico, mantenimiento) derivados. Esta es una de las razones que ha venido motivado el uso de coprocesadores dedicados. Estos coprocesadores han venido siguiendo dos filosofías. En primer lugar, contener unidades de cómputo muy específicas de manera que hacen tareas muy concretas con un rendimiento mejor que una CPU tradicional. Por otro lado, existe la cada vez más común tendencia de incluir en estos coprocesadores un gran número de unidades funcionales más sencillas de modo que, aunque cada unidad tarde más en realizar un trabajo de forma individual, el paralelismo permita obtener extraordinarias mejoras en términos de rendimiento global.

Existen problemas complejos, denominados “grand challenges” que demandan una alta potencia de computación. Como pueden ser las simulaciones climáticas/geofísicas, sistemas avanzados de diseño o realidad virtual, análisis de datos etc. Las prestaciones para estos problemas se pueden incrementar de tres maneras posibles. Una manera es mediante un hardware más potente, esto supone un sobrecoste en su obtención y mantenimiento al tener que estar actualizando el hardware de manera constante. Otra forma es la optimización de algoritmos, esto es costoso a nivel de programación ya que hay algoritmos altamente optimizados y supone un gran reto el mejorar los mismos. Y por último es la computación paralela, mediante la utilización de un hardware más paralelizable se pueden realizar una gran cantidad de operaciones en un tiempo más limitado. Por estas razones, el uso de coprocesadores supone un hardware específico para ciertas tareas, aportando también gran cantidad de cores para realizar una programación masiva en paralelo.

1.2. Objetivos

A lo largo de este proyecto se tratarán diferentes aplicaciones de la computación paralela basada en el uso de coprocesadores (procesadores dedicados para tareas específicas con los que se puede conseguir una mejor eficiencia a menor coste). Una vez presentados los ejemplos de arquitecturas más extendidas, este proyecto se centra en los coprocesadores fabricados por la empresa Tilera. Se estudiará su arquitectura y entorno de desarrollo, y se realizaran pruebas de funcionamiento.

Se llevarán a cabo diferentes pruebas, con las que se distinguirán qué factores pueden afectar a la mejora del rendimiento conseguida, factor que se estudiará teniendo en cuenta el número de procesos necesarios, así como su consumo eléctrico. También se realizará un exhaustivo estudio sobre cómo diferentes topologías de ejecución pueden afectar al cómputo basado en Tilera, con vistas a obtener una compresión a bajo nivel de la arquitectura. Dicho estudio sobre las configuraciones topológicas de cómputo tendrán en cuenta no sólo la ubicación de los elementos de cómputo utilizados, sino también de las unidades de memoria.

Una vez vistas estas peculiaridades de la propia tarjeta respecto a los accesos a memoria y comunicación entre los propios cores de la misma, se realizará un problema con mayor dificultad y se ejecutarán pruebas como la del cálculo del número PI o la encriptación de ficheros, para ver las ventajas que nos ofrece este modelo de cómputo con respecto al resto.

1.3. Estructura del documento.

En el resto del documento se detalla, de manera estructurada, lo que es la computación de alto rendimiento. De esta forma, en el capítulo dos se habla sobre qué es realmente la computación mediante coprocesadores, cuáles son los ordenadores más potentes del mundo, qué tipo de computación se utiliza para conseguir tal potencia de procesado y sobre las diferentes tarjetas de coprocesamiento como las conocidas tarjetas gráficas o las más sofisticadas y exclusivas Xeon Phi.

Una vez se ha ofrecido una descripción general de estas tarjetas, en el capítulo tres comenzará el estudio detallado en una específica desarrollada por la empresa Tilera (en concreto la tarjeta Tilera Gx36). En primer lugar se hace una descripción de su arquitectura, como funcionan y se relacionan todos los procesadores. Dicho esto, se profundiza acerca del entorno de desarrollo empleado para programar de forma más sencilla y gráfica, haciendo uso de la herramienta de desarrollo perfeccionada por Eclipse con la que se hacen simulaciones de ejecuciones para ver el correcto funcionamiento de las mismas. Tras esto, se hace una descripción sobre el equipo de trabajo, describiéndose su procesador principal, memoria, etc. Para finalizar este capítulo se lanzan pruebas con el objetivo de ver las peculiaridades de la propia tarjeta sobre la que se trabaja y así ver que procesador es más rápido dependiendo de qué funciones se quieren realizar, ya sea para lectura, escritura en memoria etc.

Por último, en el capítulo cuatro, se realiza un problema más complejo para mostrar el potencial de la tarjeta tratada en el capítulo anterior (TileGx-36). Como ya se ha dicho, primero es el cálculo del número PI y se compara con un modelo de computación simple y un modelo MPI. Seguidamente se realiza un problema de criptografía básico, teniendo como objetivo ver la librería especifica MiCA (librería aportada por la empresa Tilera para usar los motores de aceleración y así acelerar la encriptación y compresión de archivos). Con esto se llega al capítulo cinco donde se mostrarán las conclusiones obtenidas sobre dicha tarjeta llegando a en qué puntos es más interesante qué tipo de programación paralela y qué beneficios se podrían obtener con la misma.

2. Estado del arte del cómputo basado en coprocesadores

El cómputo basado en coprocesadores consiste en el uso de herramientas concretas que realizan trabajos dedicados con el objetivo de perfeccionar y aprovechar al máximo la capacidad de cómputo de una máquina. Estas máquinas realizan partes específicas de una tarea y devuelven el resultado a la CPU. Esto está pensado principalmente para liberar la CPU de tareas, pudiendo así, dedicar su capacidad de cómputo a otras tareas menos específicas como es el caso de tratamiento de imagen en los coprocesadores GPGPU que deja las funciones de procesamiento de imagen para el coprocesador y así poder trabajar la CPU con el resto de la lógica del programa. Para estimar el rendimiento obtenido, una forma es calculando el tiempo de ejecución del problema. Siendo Ts el tiempo de ejecución del problema en serie y Tp el problema en paralelo, Tp tiene un speedup o aceleración frente a Ts[1]. Otra forma sería manteniendo el tiempo de trabajo fijo, por lo que aumentaría la carga de trabajo a realizar[2]. Una vez se tiene el speedup, se puede calcular fácilmente el rendimiento hallando su proporcionalidad con el número de procesadores utilizados[3] o también como vamos a comparar más adelante con la potencia consumida[4].

2.1. HPC (High Performance Computing)

La computación de alto rendimiento se define como el aumento de potencia de cálculo con la que se consigue un rendimiento mucho mayor al que se podría conseguir con un equipo de trabajo típico para así resolver grandes problemas computacionales, por esto, este tipo de computación es usada únicamente para fines muy concretos. Esta computación se apoya en distintas tecnologías dependiendo de la arquitectura en la que se desarrolle, ya sea clúster, supercomputadores o computación paralela. El clúster es la arquitectura más usada en las listas de ordenadores más potentes del mundo (Top500 [1]). Este se basa en el uso de ordenadores interconectados por red LAN que tienen un comportamiento como si fuesen un único ordenador con lo que se obtiene la ventaja de conseguir una gran escalabilidad y buen mantenimiento a bajo coste.

[1] $speedUp = \frac{Ts}{Tp}$

[2] $speedUp = \frac{Cantidad\ de\ trabajo\ realizado\ en\ paralelo}{Cantidad\ de\ trabajo\ realizado\ programa\ serie}$

[3] $rendimiento = \frac{speedup}{n\ procesadores}$

[4] $rendimiento = \frac{speedup}{vatio}$

El Top500 es el ranking de los 500 ordenadores más potentes, computacionalmente hablando, del mundo, con capacidades muy superiores a los que una máquina común puede alcanzar. Este ranking se actualiza dos veces al año, concretamente en los meses de junio y noviembre. En esta lista se muestra la potencia en TFLOPS (FLOPS: número de operaciones en coma flotante por segundo) que es capaz de procesar el superordenador, el número de cores que necesita para dicho procesamiento y su consumo energético. En la actualidad el más potente pertenece a China, con el ordenador "Tianhe-2 (MilkyWay-2)", que con 3.120.000 cores es capaz de procesar 33.862,7 TFLOPS por segundo con un consumo de 17.808 KW. Este ranking es muy útil para ver la evolución en la computación paralela y que arquitecturas se utilizan para ello.

Como podemos ver en la Figura 1, actualmente la arquitectura más utilizada es la de clúster utilizando una programación basada en el paso de mensajes (MPI).

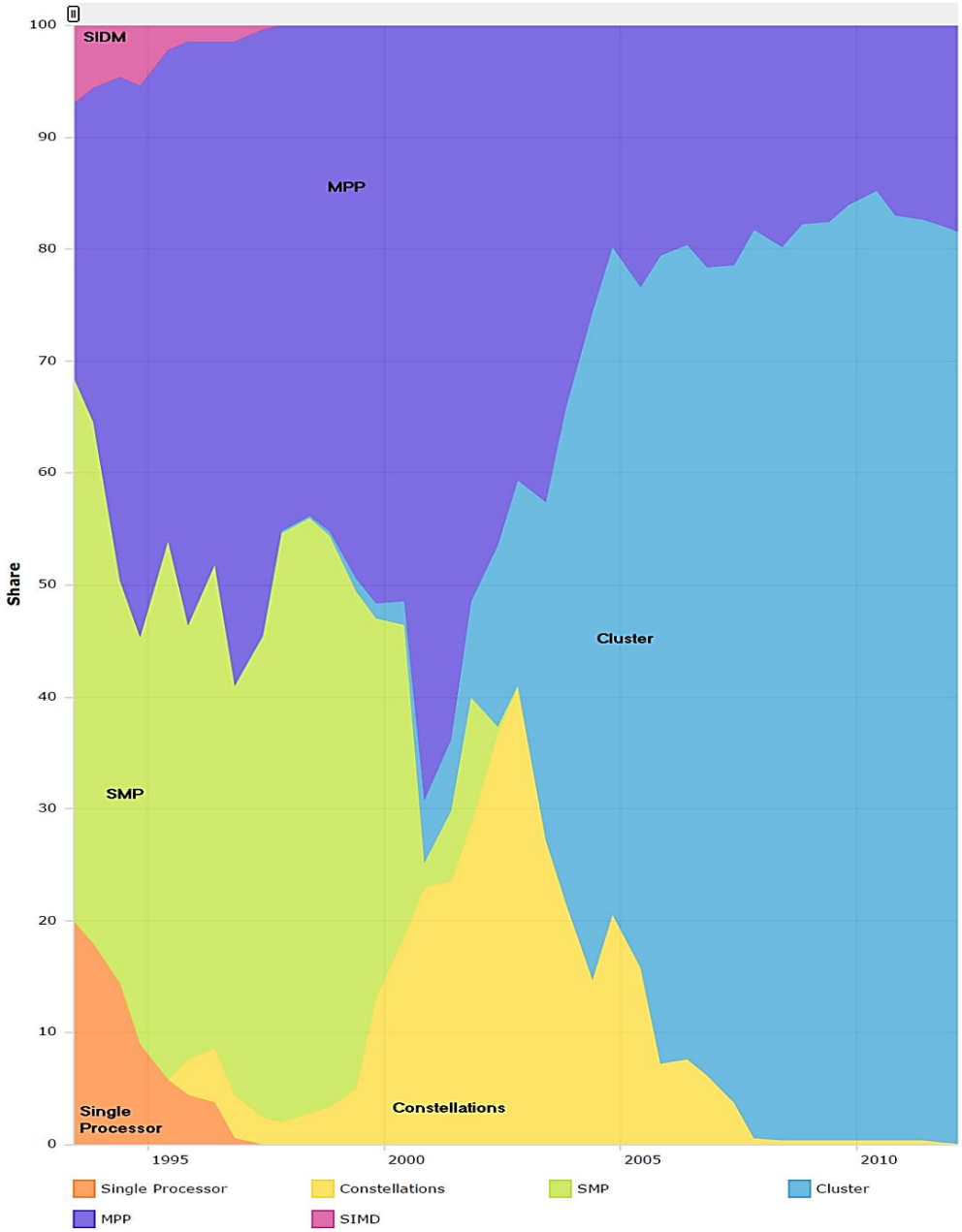

Figura 1. Arquitecturas utilizadas en el TOP500por año (Fuente: TOP500. [1]).

En el Top500 se ve una tendencia exponencial respecto a la potencia de procesamiento conseguido en los últimos años, demostrando el constante crecimiento de la multiprogramación. En la Figura 2 se puede ver esta tendencia de potencia de procesamiento que se ha seguido desde la creación de la lista Top500:

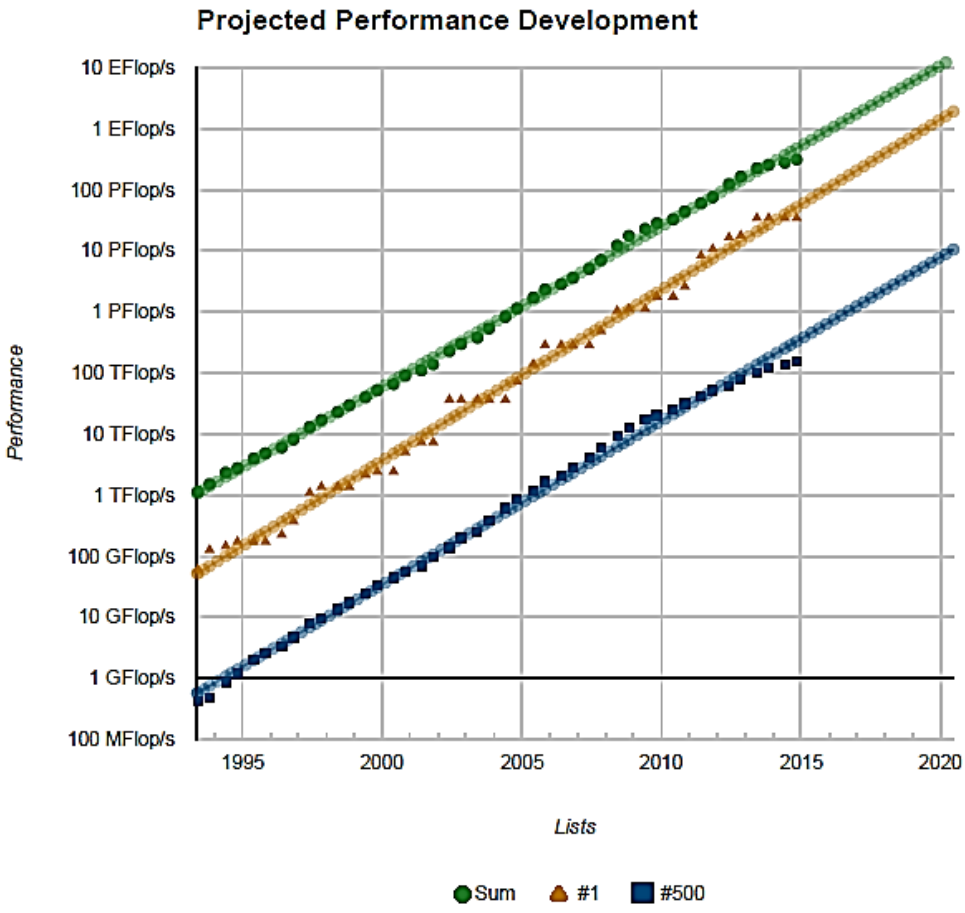

Figura 2. Evolución Rendimiento TOP500 (Fuente: TOP500[1]).

2.2. MPI

Modelo de programación basado en el paso de mensajes. MPI es un estándar que se utiliza para la programación paralela con memoria distribuida. Su principal característica es la no necesidad de memoria compartida, imposible para el uso de clúster.

Básicamente, dicho modelo se caracteriza por utilizar una librería con funciones de comunicación para el envío y recepción de mensajes entre procesos. Con lo que el programador decide la sincronización entre procesos y recursos pero la comunicación es transparente al mismo. La distribución de tareas se decide por el programador, el maestro-esclavo es el más utilizado, donde se crea un proceso master que es el encargado de distribuir los datos y recolectar los resultados. El resto de procesos ejecutan la misma tarea con los datos proporcionados por el maestro (mismo programa, múltiples datos). Su principal ventaja sobre otras librerías de paso de mensajes es su alta portabilidad a otras arquitecturas ya que se ha implementado en casi todas las arquitecturas de memoria distribuida y ha sido optimizada para el hardware en el que se ejecute.

Este modelo de programación es más complejo para el programador debido a que la repartición de tareas recae sobre él. Esto le da la ventaja de poder definir los criterios de reparto e incluso que funciones realiza cada procesador y con qué frecuencia se sincronizan entre los mismos.

2.3. OpenMP

Este modelo de programación está pensado para la programación multiproceso en memoria compartida. Para ello utiliza una arquitectura multi-core con memoria compartida con la que, mediante un conjunto de directivas simples y fáciles de usar, es posible paralelizar un programa serie donde éste genera hilos en el procesador y divide la tarea entre los mismos hilos. A diferencia de MPI, donde el programador se encarga de la sincronización y reparto, OpenMP distribuye las tareas durante la creación de los hilos siendo esto transparente al programador. Este modelo puede implementar un paralelismo tanto de grano fino como de grano grueso, en cambio MPI, debido a la latencia de mensajes, un paralelismo de grano fino lo hace ineficiente.

El paralelismo de grano fino o grueso se clasifica según la frecuencia con la que se sincronicen las tareas. Un paralelismo de grano fino necesita una alta frecuencia de sincronización, por ejemplo en bucles. Sin embargo, una paralelización con una alta frecuencia de sincronización, como funciones completas, se consideran paralización de grano grueso.

A diferencia de MPI, este modelo de programación es más accesible de cara al desarrollo del mismo debido a que se compone de simples directivas haciendo que sea el compilador y las variables de entorno las que influyen en el tiempo de ejecución completamente independiente al programador. Como desventaja, este modelo no es tan

escalable como MPI debido a que es fuertemente dependiente del procesador empleado ya que necesita trabajar con memoria compartida. Otra desventaja es para cores con diferentes latencias, es más complejo realizar una distribución de carga en cada core utilizado, debido a que estas directivas son generadas en la compilación del código, no en la ejecución del mismo.

2.4. Coprocesadores

Un coprocesador, como se ha comentado anteriormente en la introducción, sigue dos filosofías distintas. Una es la utilización de unidades de cómputo específicas. Estas están fabricadas para la realización de problemas de manera que están altamente optimizadas consiguiendo un rendimiento superior al de un procesador tradicional. La segunda filosofía es la creación de unidades más sencillas, que utilizadas en paralelo obtienen importantes mejoras en el rendimiento global.

Por otro lado, un coprocesador no está diseñado para realizar todas las funciones de la CPU, sino que para dar apoyo a la misma realizando funciones concretas para reducir su carga de trabajo. De esta forma, las CPUs necesitan de un host que mande peticiones y estos devuelvan al host el resultado obtenido. Por un lado, los principales coprocesadores en la actualidad son las *GPGPU* de "nVidia", muy útiles para el procesamiento gráfico, las *Xeon Phi* de "Intel", creadas para realizar funciones más potentes como el procesamiento de la red, etc. Por otro lado, la familia *TILEGx* de "Tilera", menos popularizadas que las anteriores, tienen sus principales usos en el apoyo a servidores y funciones criptográficas.

2.4.1. GPGPU:

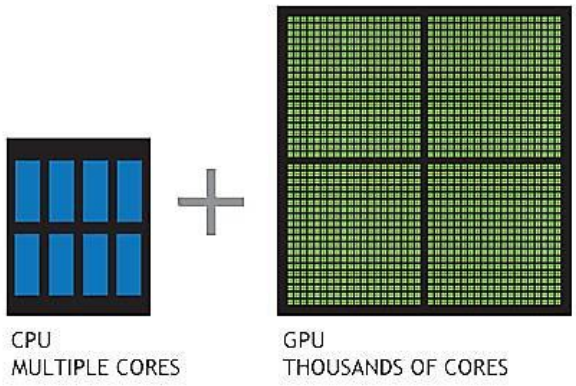

Figura 3. Representación de una CPU y una GPU. (Fuente: nvidia [2]).

GPGPU o unidad de procesamiento gráfico (Imagen derecha Figura 3) es un coprocesador dedicado a la gestión y procesamiento de gráficos y la operación de problema con coma flotante.

Estas tarjetas están pensadas para aligerar la carga de trabajo del procesador durante la ejecución de videojuegos o aplicaciones que interaccionen con imágenes, así como Photoshop o editores de video. Las GPUs realizan ciertas funciones en cuanto al procesamiento gráfico mientras que el procesador central se encarga del resto de cálculos. Las GPUs ya tienen librerías propias para la optimización de imagen más común como el antialiasing, utilizado para el suavizado de bordes consiguiendo aspectos de más realismo.

Mientras que un procesador está formado por un número de cores cada vez más complejos, una GPU está formada por cientos/miles de núcleos (GeForce GTX 750, 512 núcleos [2]) mucho más pequeños que los de la CPU, que pueden realizar trabajo de forma paralela. Aunque cada uno de los cores "sencillos" pueda tardar más que una CPU tradicional en realizar un trabajo concreto, el trabajo simultáneo de una gran cantidad de estos sencillos cores permite obtener cifras de rendimiento muy superior al de una máquina "normal". Las aplicaciones que más se benefician son las SIMD, Single Instruction Multiple Data, los procesadores se dividen en bloques, los cuales realizan la misma función apuntando a diferentes lugares de memoria (distintos datos). Por ejemplo, la serie 20 de Nvidia (Figura 4) tiene 512 núcleos divididos en 16 bloques de 32 núcleos cada uno [9], por lo que es capaz de realizar una función por bloque en 32 puntos de memoria distintos. El acceso a memoria puede tener mayor o menor latencia dependiendo del bloque de memoria y precisión de la tarjeta. En las primeras versiones (Figura 5) se necesitaba un acceso paralelo (cada hilo accede a la posición de memoria que le corresponde) y en el mismo bloque de memoria, entonces tiene una latencia de un único ciclo de reloj, si el acceso fuera cruzado o en diferentes bloques este acceso pasaría a ser en serie por lo que provocaría una alta latencia en el acceso. En las versiones actuales (Figura 6) esto se ha mejorado y el acceso puede hacerse cruzado, pero tiene que ser en el mismo bloque de memoria y necesitara un ciclo de reloj por bloque de memoria.

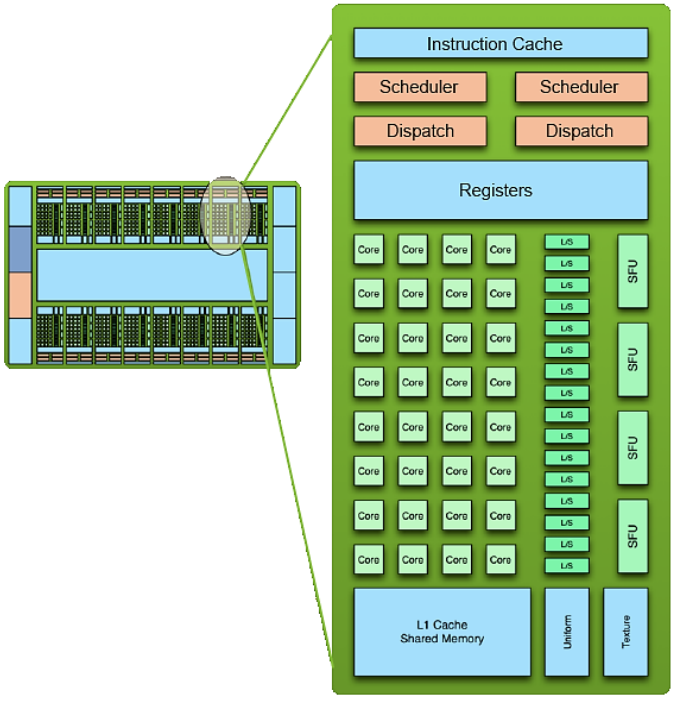

Figura 4. Coprocesador nVidia Serie 20. (Fuente U.A.M. [9])

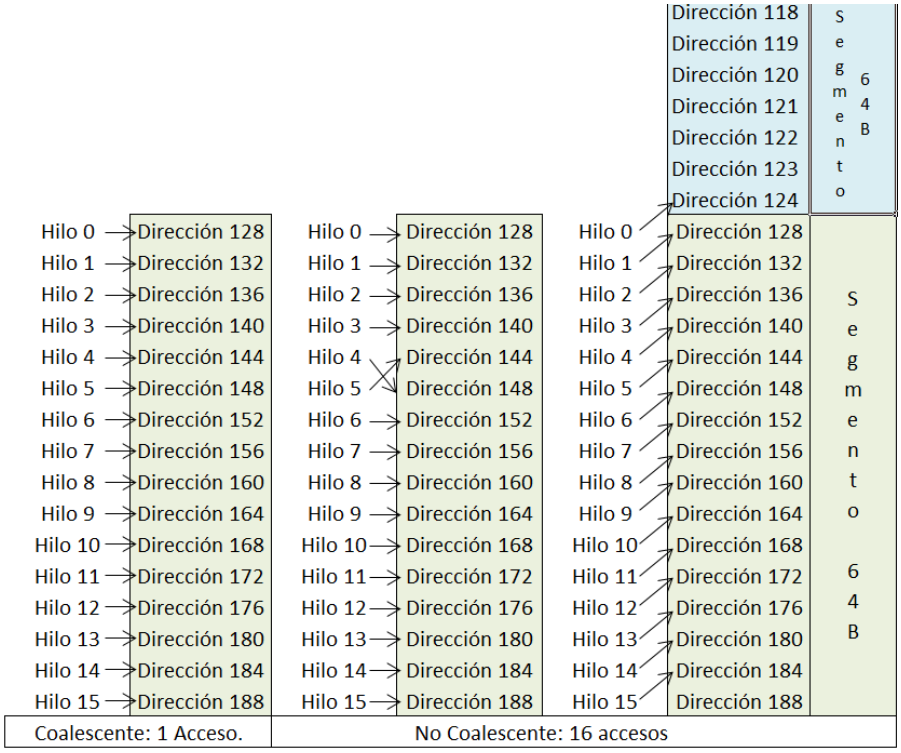

Figura 5. Acceso a memoria, primeras versiones GPU.

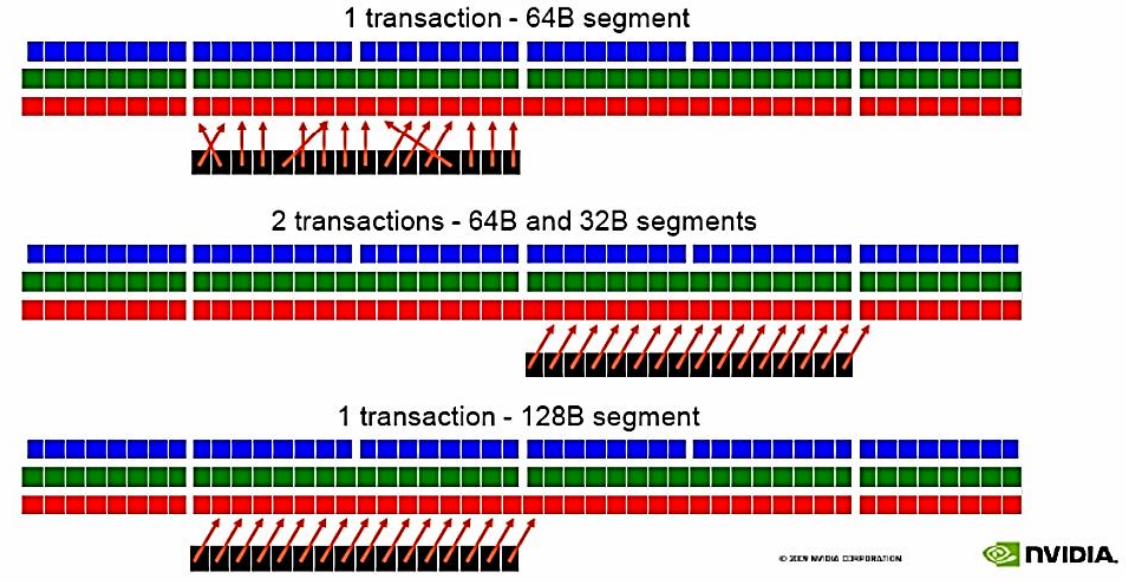

Figura 6. Acceso a memoria, actuales versiones. (Fuente nvidia [2])

2.4.2. Xeon Phi

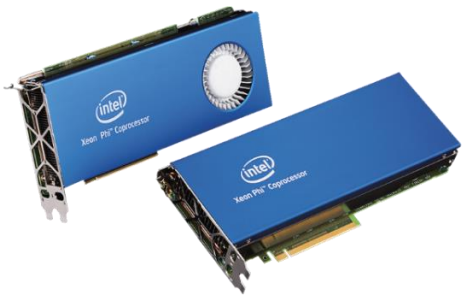

Figura 7. Dos coprocesadores Intel Xeon Phi. (Fuente: Intel [3]).

Xeon Phi o Intel MIC (Figura 7: representación de una tarjeta Xeon Phi) es un diseño de multiprocesadores creado por Intel para el desarrollo de aceleradores con el objetivo de la ejecución de problemas altamente paralelos, con la ventaja respecto a una GPU de que su arquitectura está basada en los procesadores de Intel para propósito general, por lo que con un modelo de programación común obtenemos un hardware altamente paralelo mientras que otras arquitecturas como las GPUs necesitan su propio modelo de programación. Estas tarjetas también pueden utilizar el modelo de primitivas de las GPUs.

Los principales ordenadores de la lista Top500 son apoyados con tarjetas Xeon Phi, las cuales dan apoyo en las operaciones vectoriales y de coma flotante, reduciendo el trabajo de la CPU principal que puede dedicar trabajo a otras tareas. Estos coprocesadores ofrecen hasta 61 procesadores consiguiendo 244 hilos con picos de procesamiento de 1,2 teraFLOPS (Fuente: Intel [3]). Esto es debido gracias al soporte que ofrece el compilador de Intel para las instrucciones vectoriales.

Estas tarjetas pueden ser utilizadas como coprocesador o como sistema autónomo pudiéndose utilizar sin necesidad de un host. Soportan MPI, lo que es una ventaja frente a CUDA debido al conocimiento previo por los mismos desarrolladores hacia el lenguaje.

2.4.3. Tile-Gx

Figura 8. Tarjeta Tilera TILE-Gx36 (Fuente Slash Gear [5]).

Los coprocesadores Tile-Gx (Figura 8) son coprocesadores multi-núcleo desarrollados por la empresa Tilera (comprada recientemente por EZchip), cuya filosofía es la creación de coprocesadores funcionales, altamente escalables. Esto se consigue mediante la interconexión entre los procesadores usando una red de malla con la que se han llegado a unir hasta 100 núcleos con mínima interferencia en el rendimiento. Sus principales objetivos son el procesamiento de red[18], la criptografía masiva y el aceleramiento de aplicaciones.

Con estos coprocesadores, Tilera fusiona las dos filosofías de coprocesamiento mencionadas anteriormente. Por un lado utiliza muchas unidades funcionales o cores (hasta 100), pero también posee unidades específicas para acelerar la E/S de la red y las operaciones criptográficas. En el Capítulo 3 se trata más detalladamente este modelo.

3. TILERA

Figura 9. Coprocesador tilera de 36 cores (Fuente maximumpc [6]).

Tilera es una empresa estadounidense fundada en 2004 por Anant Agarwal, Devesh Garg y Vijay K. Agarwal dedicada al desarrollo de coprocesadores multicore con un diseño altamente escalable. Su primer coprocesador se llamó "TILE64", creado en agosto 2007, un procesador de 64 núcleos con el que consigue un total de 166 MIPS (Millones de instrucciones de enteros por segundo) con un consumo de potencia de entre 15 y 22 W a 700 MHz. Es precisamente ese bajo consumo, contrapuesto con la potencia de sus cores, lo que otorga a Tilera una relación rendimiento/potencia muy interesante. Sus siguientes procesadores de 36 y 16 núcleos son capaces de procesar 40 Gigabit/s con un consumo inferior a 25 W que, comparado con el de otros coprocesadores como el Xeon Phi de Intel de 36 núcleos, aportan el mismo rendimiento pero a costa de un consumo cinco veces mayor[8].

3.1. Arquitectura

El coprocesador a estudiar es el "Tile36" (Figura 10). Se trata de un coprocesador compuesto por diversos elementos de cómputo interconectados mediante una red en forma de malla bidimensional con la interconexión "iMesh", de la cual se hablará más adelante, con la que se consigue un fácil aumento en el número de procesadores sin necesitar modificaciones. Este en concreto tiene 36 *tiles*, cada "tile" es un procesador de propósito general con tres cores, memoria caché de niveles 1 y 2 y un switch no bloqueante para la comunicación con el resto de *tiles*.

Su producción se realiza mediante una fabricación litográfica de grosor entre 45 y 90nm con una arquitectura VLIW (Very Long Instruction Word) basada en la arquitectura MIPS (Microprocessor without Interlocked Pipeline Stages), que sigue la arquitectura Hardvard de diseño de procesadores al separar la memoria de instrucciones de la de datos. Cada *tile* es capaz de ejecutar un sistema operativo de manera independiente al resto.

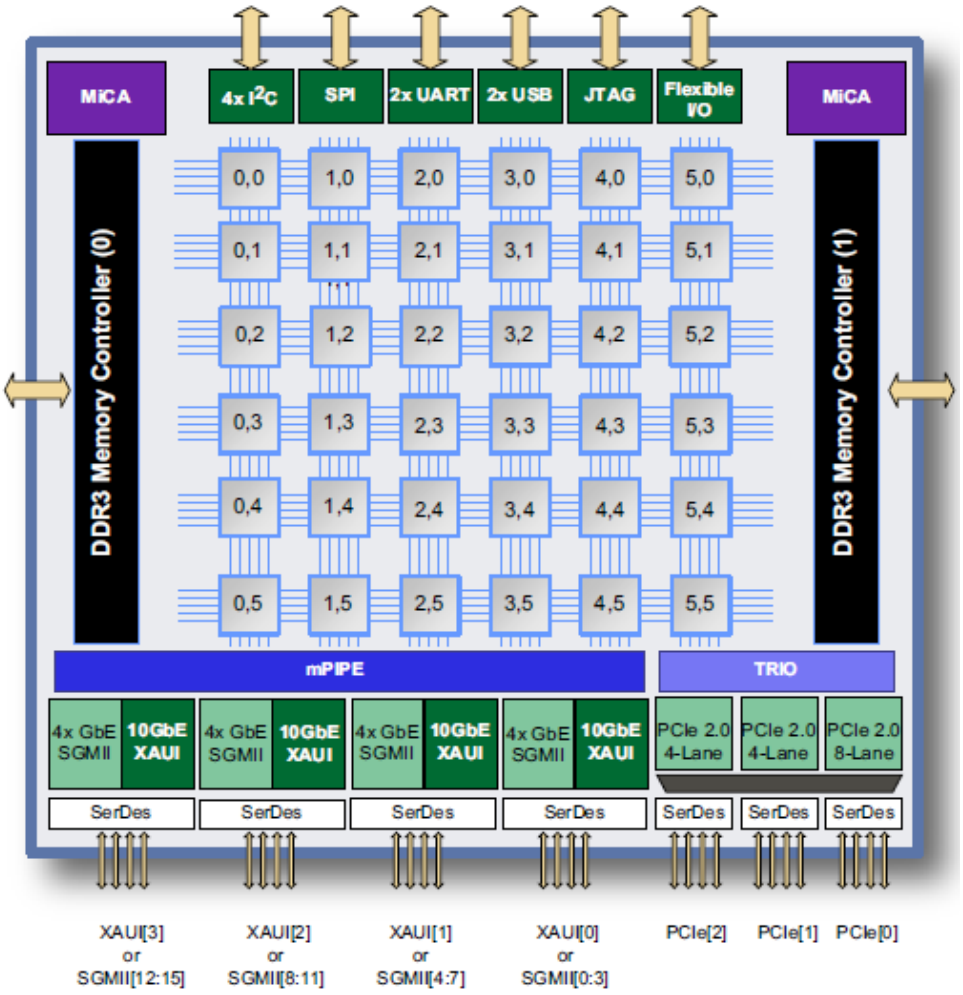

Figura 10. Representación procesador tilera de 36 tiles (Fuente Oficial Tilera [7]).

Cada *tile* (Figura 11) está dividido en tres partes principales:

- Processor Engine.
- Cache Engine.
- Switch Engine.

El motor de procesamiento se trata de un microprocesador de 64 bits con una arquitectura VLIW de tres vías, lo que le da la capacidad de ejecutar hasta 3 operaciones de 64 bits por ciclo usando un contador propio y con un acceso directo a la memoria caché (DMA). Cada procesador es capaz de ejecutar un sistema operativo independiente por lo que se pueden realizar diferentes instrucciones en cada *tile* y comunicarse entre ellos.

La jerarquía de cache de cada *tile* tiene un Tanslation Lookaside Buffer (TLB) y dos niveles de cache. El primer nivel tiene una memoria de 32KB, dividida en 16KB para instrucciones y otros 16KB para datos. En la caché de segundo nivel tenemos 256KB por *tile* común para datos e instrucciones. Esta caché tiene un motor DMA que se encarga de la gestión e intercambio de datos con el resto de *tiles* y la memoria RAM haciendo que sea el propio hardware el que se encargue de la sincronización de datos, y no el programador. Todos los *tiles* pueden acceder a la memoria caché del resto de *tiles*, mediante una abstracción de memoria virtual con las que se consigue manejar hasta 9MB. A una frecuencia de 1GHz esta configuración permitiría el procesamiento de 192 mil millones de instrucciones por segundo con hasta 27 Tbps de interconexión y hasta 40 Gbps de ancho de banda en la E/S.

La red de interconexión "iMesh" es un motor de intercambio que facilita la transferencia de datos entre los *tiles* con funciones de buffer y control de flujos de datos mejorando la sincronización entre las mismas. Los *tiles* están conectados entre ellos en forma de malla, estando enlazando cada *tile* con el de arriba, abajo y lados. Para enviar un dato de un *tile* a *tile*, como se ve en la Figura 10, éste se va enviando de *tile* a *tile* adyacente hasta llegar al *tile* deseado y coger el dato. Con el fin de una conexión sin bloqueos, cada canal de los cinco que posee cada *tile* tiene un fin. MDN (Memory Dynamic Network) y TDN (Tile Dynamic Network) mejoran el ancho de banda de acceso a memora, STN (Static Network) acceso a caché y coherencia de memoria. IDN (I/O Dynamic Network) trata la comunicación E/S externa y UDN (User Dynamic Network) es el canal de usuario por el que se hace la transferencia de paquetes entre *tiles*.

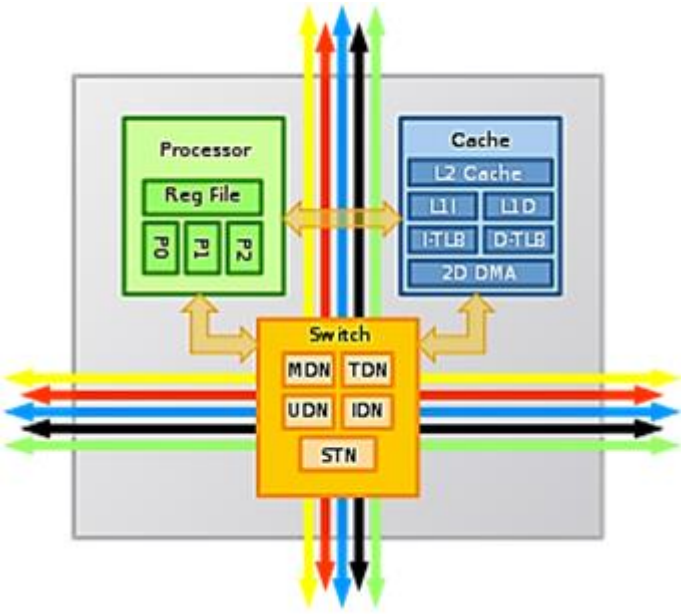

Figura 11. Representación de un "Tile". (Fuente: Documentación Tilera [7])

MiCA (Multistream iMesh Coprocessing Accelerator), este motor se encarga de la aceleración de funciones criptográficas, compresión y descompresión de datos. La función de estos motores es la de la realización de funciones predefinidas sobre un búfer de memoria.

El motor mPIPE (multicore Programmable Intelligent Packet Engine) es una interfaz para el tratamiento de los paquetes entre los *tiles* y la red. Este motor tiene capacidad de análisis sintáctico para identificar el flujo de un paquete, distribución de paquetes, gestor de carga de trabajo entre *tiles* para el procesamiento y control sobre los mismos paquetes. El instalado sobre el coprocesador estudiado consigue una velocidad de carga de hasta 40 Gbps.

TRIO (Transactional I/O) se encarga de la carga de datos entre el coprocesador y el sistema. Este motor puede funcionar como anfitrión o como root, en el primer caso, el coprocesador es utilizado como complemento permitiendo una comunicación entre el host Linux y la tarjeta. En el caso de root está pensado para poder conectar otros dispositivos (por ejemplo, un controlador de disco duro) directamente sobre el coprocesador.

Estos procesadores se instalan en una tarjeta Tilera que trabaja junto con un equipo completo (Host). Su instalación es mediante PCI Express x8 con la que se obtienen 4 puertos para red y un micro USB para conexión con el HOST. La tarjeta con la que trabajamos en concreto es la TILE-Gx8036 con dos bancos RAM DDR3 de 4GB cada una y cuatro puertos de Ethernet capaces de transmitir un total de 40Gb de datos.

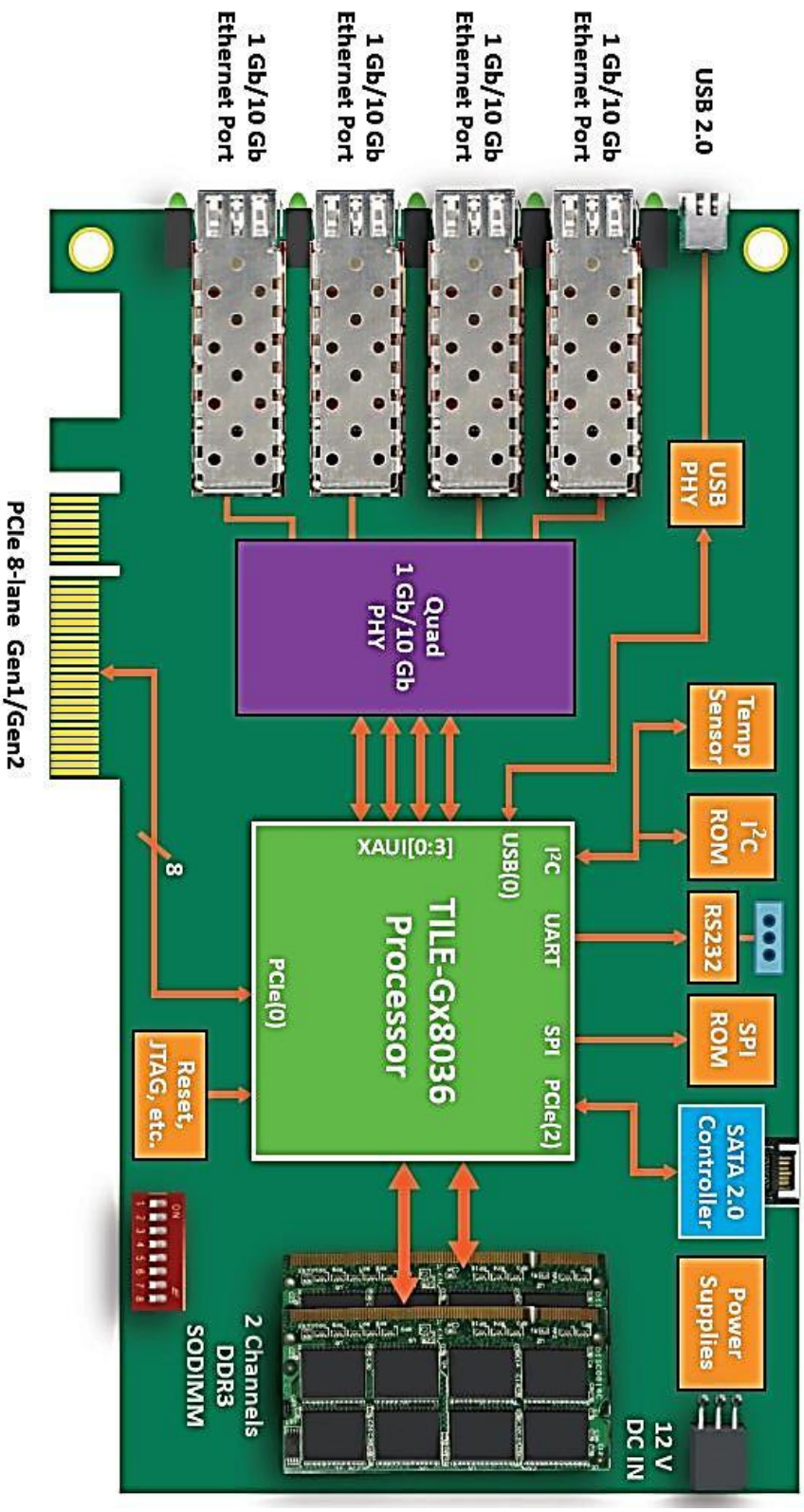

Figura 12. Representación tarjeta TILE-Gx36. (Fuente Documentación Tilera [7]).

3.2. Entorno de desarrollo

Para el desarrollo de software que se ejecute sobre una tarjeta tilera se ha perfeccionado un entorno específico llamado "Tilera´s Multicore Development Environment", también llamado TileMDE. Este entorno está basado en el proyecto Eclipse bajo un diseño gráfico común para todas las tarjetas, por lo que proporciona una herramienta común para trabajar con tarjetas tilera específicas y un compilador de C/C++ junto a un simulador del hardware de las propias tarjetas tilera. Este simulador hace posible el desarrollo y trabajo de software para estas tarjetas sin la necesidad de tener una tarjeta física en el equipo de desarrollo y se puede ver el comportamiento de forma independiente en cada *tile*, sus comunicaciones y el tiempo real en uso por cada uno. Después de la ejecución de un proceso, el IDE muestra el tiempo real dedicado por la tarjeta tilera para la ejecución del código contando únicamente el tiempo dedicado al programa y una representación gráfica de los *tiles* activos en cada momento lo que le hace una excelente herramienta visual para su codificación, desarrollo y debug.

La ejecución de una tarea sobre la tarjeta sigue el siguiente procedimiento:

1. Arranque de la tarjeta Tilera: Se carga una imagen vmlinux (sistema operativo Linux) en cada *tile* que se vaya a utilizar en el proceso. Esta imagen viene por defecto aunque se puede cambiar e instalar otra al ejecutar el programa mediante comandos en la ejecución

2. Carga de Ficheros: Una vez haya sido configurada la tarjeta Tilera se cargan los ficheros y binarios generados por el compilador especifico de tilera Tile-gcc, esto crea un directorio donde se ejecutarán los binarios.

3. Lanzamiento de la aplicación: Se ejecuta la aplicación en un *tile*, este es utilizado como "shepherd" (pastor) y se encarga de la planificación y asignación de tares entre el resto de *tiles*.

4. Recogida de datos: Se recogen los datos y se descargan al ordenador anfitrión que será el encargado de tratarlo con el usuario.

Las diferentes etapas pueden ser configuradas, lo que permite al programador control sobre el uso de los *tiles* y la memoria utilizada por el mismo. Para ello hay que tener en cuenta ciertos criterios en la programación:

- La tarjeta tilera tiene acceso total a todos los ficheros del anfitrión antes y durante la ejecución de un programa, al cargar el programa sobre la tilera se genera un directorio desde el que se sitúan los *tiles*.

- Cada *tile* puede comunicarse con el anfitrión y otros *tiles* de manera independiente gracias a la interconexión iMesh.

- La sincronización/comunicación entre los *tiles* puede ser por memoria compartida, paso de mensajes, semáforos o algún mecanismo comúnmente utilizado en otras arquitecturas.

- Su arquitectura y conexión entre *tiles* permite una sencilla expansión pudiendo añadir varias tarjetas tilera a un mismo anfitrión, estando estas interconectadas entre ellas como una sola.

La programación para esta arquitectura es mediante ANSI C, un lenguaje altamente conocido y expandido con una programación de alto nivel con un gran rendimiento. Para la coordinación entre los *tiles* se usan librerías propias que serán detalladas más adelante. Estas librerías permiten el uso de la arquitectura iMesh con muchos *tiles* dando funciones de gestión de procesos, canales, paso de mensajes y memoria compartida. Las funciones son transparentes al programador por lo que los caminos realizados para el paso de mensajes están en manos del coprocesador, no del programador. También se puede compilar y trabajar desde terminal mediante comandos propios como tile-gcc para la compilación o tile-monitor para la puesta en marcha de un proceso. Estos irán acompañados de varios argumentos donde se seleccionara los *tiles* donde se ejecutara el programa o una imagen en caso de querer una simulación, el directorio, máquina virtual a cargar en los *tiles* etc. Por ejemplo si queremos compilar un código llamado ejemplo.c ejecutaremos el comando "tile-gcc –o ejemplo –Wall –std=gnu99 –Werror –g -0s ejemplo.c", igual que un programa normal en linux y para ejecutar este ejemplo sobre la tarjeta utilizando 4 *tiles* seria "tile-monitor –tile 2x2 –try-console-out console.out –mkdir /opt/prueba –cd /opt/prueba –upload ejemplo ejemplo – ejemplo argumentos", donde se ve que elegimos 4 tiles en forma 2x2 y creamos un directorio prueba desde el que se localizará la tarjeta para intercambio de datos.

3.3. Entorno de trabajo:

Para el desarrollo del trabajo se ha utilizado un ordenador con una tarjeta tilera TILEGx-36 montada sobre un ordenador con una CPU Intel i7 de 4 núcleos con arquitectura de 64 bits. Teniendo una memoria caché de 64KB para el primer nivel 256KB la de nivel 2 y caché de nivel 3 de 1GB de memoria. Cada core es capaz de trabajar con dos hilos consiguiendo de manera virtual 8 cores en la CPU con una frecuencia de 3600Hz. Esta CPU va acompañada de 16GB de memoria RAM y una tarjeta Nvidia GForce 8400.

Para gestionar todo lo dicho anteriormente se utiliza una distribución Linux basado de la distribución Red Hat llamada "Scientific Linux" con arquitectura de 64bits que utiliza un kernel linux 2.6.32-358.el6.x86_64. Los paquetes del entorno tilera son ".rpm" por lo cual se necesita una distribución basada en Red Hat, por esto se decide una distribución de libre uso idéntica a Red Hat quitando los paquetes privados de la misma teniendo licencia GNU.

3.4. Librerías específicas:

Durante la instalación del entorno, tilera añade un grupo de librerías para la administración de los *tiles*, así como el *tile* a ejecutar como el uso de semáforos mensajes o memoria compartida. Tilera Multicore Component ofrece un conjunto de primitivas (funciones) que facilitan la paralelización de problemas incluyendo rutinas para la gestión de tareas, núcleos o asignación de memoria. Esta API es válida tanto para la realización de procesos como para la gestión de hilos, lo que hace más cómodo su uso. En el anexo 7.2 se describirán las librerías utilizadas a lo largo del proyecto y sus funciones más importantes.

3.5. Peculiaridades del cómputo basado en TILERA.

Todos los sistemas de cómputo hoy en día tienen sus propias peculiaridades, ya sea de ejecución, utilización o mantenimiento. En este apartado nos centraremos en el coprocesador de tilera TILEGx-36.

Como se ha mencionado anteriormente este procesador está compuesto de 36 *tiles* o procesadores idénticos conectados en forma de malla. A continuación se comprueba si el rendimiento ofrecido es homogéneo para distintos patrones de aplicación. El siguiente apartado se centra en la lectura y escritura de memoria de forma independiente por cada *tile* viendo cuales obtienen los mejores resultados y cuales, al tener que leer datos pasando por más *tiles*, son más lentos en el acceso a memoria. Vamos a ver el rendimiento sobre los diferentes *tiles* (o un conjunto de *tiles* ya que no siempre se van a usar todos en la ejecución del mismo problema). Los procesadores tilera tienen la ventaja de poder ejecutar diferentes tareas a la vez en diferentes procesadores, teniendo inclusive, a diferencia de otros coprocesadores como los GPGPUs, el control sobre que procesadores ejecutar en las diferentes tareas pudiendo dejar parte libre para otras tareas como es el procesamiento en red, motivación de los procesadores de red.

3.5.1. Suma n números.

La suma de N números consiste en sumar un array de números devolviendo el resultado obtenido. Esto supone un alto coste en lectura de memoria ya que los números se alojan en la RAM. Con este problema se analiza el efecto del uso de memoria de la tarjeta tilera y cómo afecta usar unos *tiles* u otros. Los resultados a comparar es el tiempo necesitado por el programa para realizar dicha suma.

Primero se ejecuta el problema sobre 1 único *tile*, variando el *tile* utilizado y observando el tiempo necesario para realizar la suma/ lectura de memoria. La tarjeta tilera posee dos bancos de memoria RAM de 4GB cada uno, primero se ha trabajado sobre el primer banco de memoria, seguidamente se ha probado el segundo banco y se ha terminado con la suma de números rellenando completamente los dos bancos de memoria.

Para la primera prueba se ha usado un array de 4GB alojado en la primera RAM (dirección 0x00000). Los tiempos de la suma son los siguientes:

La tabla presentada representa los *tiles* en orden según las coordenadas como se localizan en la tilera representada en la Figura 10 siendo la coordenada 0,0 la esquina superior izquierda, la suma de 4GB son 1.073.741.820 números enteros de 32 bits. El dato representado es el tiempo en ejecutar el problema (tiempo de ejecución de la tarjeta sin considerar el overhead de cargar del programa en la misma) dado en segundos, marcaremos en rojo los *tiles* más lentos y en verde los más rápidos.

Tiempos en realizar la ejecución de suma de n números:

		Columna					
		0	1	2	3	4	5
Fila	0	8.03s	7.98s	7.79s	7.90s	7.93s	8.00s
	1	7.96s	7.87s	7.88s	7.73s	7.67s	7.91s
	2	7.94s	7.88s	7.83s	7.59s	7.82s	7.84s
	3	7.95s	7.76s	7.62s	7.79s	7.83s	7.75s
	4	7.99s	7.92s	7.88s	7.85s	7.86s	7.80s
	5	8.06s	7.92s	7.85s	7.83s	7.85s	7.79s

Tabla 1. Representación tiempos tilera suma n números por tile físico, memoria de números 4GB desde posición RAM 0. (Resultados en segundos)

Max: 8.06s Min: 7.59s Media: 7.85s

Tras obtener los resultados se puede ver como los *tiles* centrales tienen mejores tiempos de ejecución debido a una mejor latencia de acceso a memoria, si se pasan los 4GB al segundo banco de memoria se tienen los siguientes resultados:

Tiempos en realizar la ejecución de suma de n números:

		Columnas					
		0	1	2	3	4	5
Filas	0	8.07s	7.96s	7.70s	7.91s	7.92s	7.78s
	1	7.99s	7.78s	7.84s	7.66s	7.85s	7.92s
	2	7.82s	7.64s	7.81s	7.69s	7.84s	7.87s
	3	7.83s	7.78s	7.83s	7.64s	7.70s	7.78s
	4	7.85s	7.92s	7.84s	7.84s	7.70s	7.80s
	5	7.84s	7.87s	7.94s	7.84s	7.93s	7.81s

Tabla 2. Representación tiempos tilera suma n números por tile físico, memoria de números 4GB desde posición RAM 1. (Resultados en segundos)

Max: 8.07s Min: 7.64s Media: 7.82s

Los resultados no son tan visibles como en el primer caso, pero se sigue viendo como los bordes hacen peores tiempos que los *tiles* más centrados, si se pasa a usar toda la memoria disponible se ven unos tiempos muchos más claros donde los *tiles* centrales son claramente los más rápidos y las esquinas resultan ser los *tiles* más lentos llegando a casi medio segundo entre el más lento y el más rápido. La cantidad de números a sumar es el doble por lo que el tiempo es prácticamente el doble que en las pruebas anteriores.

Tiempos en realizar la ejecución de suma de n números, acceso a los dos bancos:

		Columnas					
		0	1	2	3	4	5
Filas	0	13.70s	13.56s	13.51s	13.49s	13.57s	13.65s
	1	13.56s	13.44s	13.33s	13.35s	13.39s	13.48s
	2	13.49s	13.38s	13.30s	13.29s	13.31s	13.42s
	3	13.45s	13.38s	13.27s	13.29s	13.32s	13.40s
	4	13.58s	13.44s	13.34s	13.35s	13.37s	13.46s
	5	13.69s	13.60s	13.50s	13.48s	13.55s	13.63s

Tabla 3. Representación tiempos de ejecución TILEGX-36 suma n números por tile físico, uso de toda la memoria. (Resultados en segundos.)

Max: 13.70s Min: 13.27s Media: 13.45s

Una vez visto los tiempos usando un *tile* se pasa a usar un conjunto de 6 *tiles* con una distribución 1x6, 6x1, 2x3 y 3x2 con lo se verá con que distribución es más efectiva la comunicación entre los mismos y se verá cómo afecta lo cerca que este de la memoria RAM para mayor velocidad de acceso. Con los tiempos de ejecución anteriores, se ha visto cómo trabajan los *tiles* de manera independiente respecto a la lectura de memoria, ahora se verá cómo trabajan en conjunto sobre la misma memoria, viendo si crean conflictos y se pierde eficacia.

En el primer caso se ha decidido usar una distribución rectangular de 2x3 estando todos los *tiles* unidos. Los resultados son los siguientes:

La siguiente tabla representa el tiempo de la ejecución de los *tiles* con distribución rectangular 2x3 estando el *tile* 0,0 en la posición indicada en la tabla. Para una representación gráfica en la siguiente tabla se ven que tiles son ejecutados en cada cuadro de la tabla de tiempos utilizando las coordenadas de la Figura 10. Estos resultados se dan en segundos.

Tabla con los *tiles* utilizados para la suma de n números en la Tabla 5:

(0,0)(1,0) (0,1)(1,1) (0,2)(1,2)	(1,0)(2,0) (1,1)(2,1) (1,2)(2,2)	(2,0)(3,0) (2,1)(3,1) (2,2)(3,2)	(3,0)(4,0) (3,1)(4,1) (3,2)(4,2)	(4,0)(5,0) (4,1)(5,1) (4,2)(5,2)
(0,1)(1,1) (0,2)(1,2) (0,3)(1,3)	(1,1)(2,1) (1,2)(2,2) (1,3)(2,3)	(2,1)(3,1) (2,2)(3,2) (2,3)(3,3)	(3,1)(4,1) (3,2)(4,2) (3,3)(4,3)	(4,1)(5,1) (4,2)(5,2) (4,3)(5,3)
(0,2)(1,2) (0,3)(1,3) (0,4)(1,4)	(1,2)(2,2) (1,3)(2,3) (1,4)(2,4)	(2,2)(3,2) (2,3)(3,3) (2,4)(3,4)	(3,2)(4,2) (3,3)(4,3) (3,4)(4,4)	(4,2)(5,2) (4,3)(5,3) (4,4)(5,4)
(0,3)(1,3) (0,4)(1,4) (0,5)(1,5)	(1,3)(2,3) (1,4)(2,4) (1,5)(2,5)	(2,3)(3,3) (2,4)(3,4) (2,5)(3,5)	(3,3)(4,3) (3,4)(4,4) (3,5)(4,5)	(4,3)(5,3) (4,4)(5,4) (4,5)(5,5)

Tabla 4. Representación de los tiles activos en los cuadros de los siguientes resultados con reparto de tiles 2x3

Tabla de tiempo de ejecución:

		Columna comienzo región 2x3				
		0	1	2	3	4
Filas comienzo región 2x3	0	2.48s	2.47s	2.47s	2.47s	2.49s
	1	2.47s	2.48s	2.46s	2.47s	2.48s
	2	2.48s	2.47s	2.47s	2.47s	2.49s
	3	2.48s	2.47s	2.54s	2.47s	2.50s

Tabla 5. Resultados tiempos obtenidos suma n números distribución 2x3.

Max: 2.50s Min: 2.46s Media: 2.48s

Los resultados son muy parecidos, pero en estos resultados no se tiene el objetivo de ver cuál es la zona más rápida, sino que distribución es la más adecuada a la hora de seleccionar que *tiles* usar. Por lo cual se seguirá con diferentes distribuciones para ver qué resultados se obtienen antes de tomar decisiones erróneas. La siguiente distribución será la inversa a la anterior elegida, 3x2, con la que se han obtenido los siguientes resultados:

Tabla representativa de los *tiles* activos para los resultados dados en segundos de la Tabla 7:

(0,0)(1,0)(2,0) (0,1)(1,1)(2,1)	(1,0)(2,0)(3,0) (1,1)(2,1)(3,1)	(2,0)(3,0)(4,0) (2,1)(3,1)(4,1)	(3,0)(4,0)(5,0) (3,1)(4,1)(5,1)
(0,1)(1,1)(2,1) (0,2)(1,2)(2,2)	(1,1)(2,1)(3,1) (1,2)(2,2)(3,2)	(2,1)(3,1)(4,1) (2,2)(3,2)(4,2)	(3,1)(4,1)(5,1) (3,2)(4,2)(5,2)
(0,2)(1,2)(2,2) (0,3)(1,3)(2,3)	(1,2)(2,2)(3,2) (1,3)(2,3)(3,3)	(2,2)(3,2)(4,2) (2,3)(3,3)(4,3)	(3,2)(4,2)(5,2) (3,3)(4,3)(5,3)
(0,3)(1,3)(2,3) (0,4)(1,4)(2,4)	(1,3)(2,3)(3,3) (1,4)(2,4)(3,4)	(2,3)(3,3)(4,3) (2,4)(3,4)(4,4)	(3,3)(4,3)(5,3) (3,4)(4,4)(5,4)
(0,4)(1,4)(2,4) (0,5)(1,5)(2,5)	(1,4)(2,4)(3,4) (1,5)(2,5)(3,5)	(2,4)(3,4)(4,4) (2,5)(3,5)(4,5)	(3,4)(4,4)(5,4) (3,5)(4,5)(5,5)

Tabla 6Representación de los tiles activos en los cuadros de los siguientes resultados con reparto de tiles 3x2.

Tabla de tiempo de ejecución:

		Columna comienzo región 3x2			
		0	1	2	3
Filas comienzo región 3x2	0	2.48s	2.48s	2.48s	2.50s
	1	2.47s	2.46s	2.47s	2.49s
	2	2.47s	2.49s	2.47s	2.48s
	3	2.47s	2.47s	2.47s	2.48s
	4	2.48s	2.47s	2.47s	2.50s

Tabla 7. Resultados de tiempo de ejecución obtenidos suma n números distribución 3x2.

Max: 2.50s Min: 2.46s Media: 2.47s

Otra distribución también posible es la de filas y columnas.

Columnas:

Tabla de tiempo de ejecución:

Columna					
0	1	2	3	4	5
2.50s	2.49s	2.49s	2.48s	2.49s	2.57s

Tabla 8. Resultados de tiempo de ejecución obtenidos suma n números distribución 1x6.

Max: 2.57s Min: 2.48s Media: 2.50s

Filas:

Tabla de tiempo de ejecución:

Fila	0	2.51
	1	2.51
	2	2.47
	3	2.50
	4	2.50
	5	2.52

Tabla 9. Resultados de tiempos de ejecución obtenidos suma n números distribución 6x1.

Max: 2.52 Min: 2.47 Media: 2.50

Los resultados con el uso de un banco de memoria u otro han salido muy similares a los obtenidos anteriormente por lo que por no ser repetitivos se ha considerado no influyente la zona de memoria para la distribución de *tiles* en la ejecución del problema. Como conclusión para el reparto se puede ver que la peor distribución de trabajo es en filas y columna, cuanto más cuadrados mejor ya que hay más *tiles* "centrales" y cuánto más aplanados, usamos más *tiles* de los exteriores. A pesar de ello, como se ha dicho en la definición de la arquitectura de tilera, la conexión entre *tiles* está muy optimizada por lo que son mínimas las diferencias de tiempos de ejecución dependiendo del reparto. Como referencia se han puesto todos los *tiles* y el tiempo ha sido de 0.849448s lo que supone una aceleración de 2.94 sobre la tarjeta con 6 *tiles* y 16.13 sobre la tarjeta con un solo *tile*, la

aceleración es más eficiente con 6 *tiles* debido a que con 36 *tiles* el tiempo de ejecución de la tarea de cada *tile* es tan corto que el tiempo de carga del programa en cada *tile* supone un 50% del tiempo total de ejecución. Sobre una sola maquina el problema se ha ejecutado en 6.38 segundos lo que significa una aceleración de 7.5 utilizando la tarjeta completa.

3.5.2. Multiplicación de Matrices

Con la suma de n números se ha puesto a prueba el rendimiento de los *tiles* en una aplicación que hace uso intensivo (en lectura) de los bancos de memoria. La multiplicación de matrices pretende probar un escenario de cómputo más intensivo y con accesos a memoria tanto de lectura como de escritura. Para ello se ha hecho un programa sencillo que conquiste en multiplicar dos matrices y guardar el resultado en una tercera. Para evitar más colisiones de las necesarias en la multiplicación, primero multiplicaremos las líneas con las columnas correspondientes y el resultado se ha guardado en una variable privada, finalizada esta multiplicación, entonces se escribirán los resultados en la matriz final que será una zona de memoria compartida. Las pruebas realizadas son las mismas que en la suma de n números por lo que no es necesario detenerse a explicar la tabla nuevamente. Los resultados representan los segundos tardados en hacer la multiplicación.

Los tiempos sacados en la ejecución de la matriz de 1000x1000 usando un único core, como en la primera prueba del ejercicio de suma de n números, los tiempos sacados son en segundos y cada cuadro representa al *tile* que está en la posición del cuadro:

		Eje de las x					
		0	1	2	3	4	5
Eje de las y	0	171,81s	171,32s	170.61s	171.33s	171.55s	172.86s
	1	172.14s	171.56s	170.04s	170.08s	170.47s	172.04s
	2	171.52s	169.82s	170.14s	169.61s	170.01s	170.68s
	3	171.75s	170.34s	169.68s	169.91s	170.03s	171.58s
	4	171.64s	169.38s	169.61s	169.65s	170.12s	171.51s
	5	172.00s	170.74s	170.73s	171.21s	170.83s	171.17s

Tabla 10. Representación tiempos tilera multiplicación matriz 1000x1000 en cada tile físico. (Resultados en segundos)

Max: 172.86 Min: 169.38 Media: 170.81

Los resultados son similares a los de la lectura de memoria con suma de n números siendo los *tiles* centrales más rápidos que en los bordes. Igual que ha sucedido con la suma, a continuación se ve que reparto es el más adecuado para hacer tareas en grupo a la hora de elegir que *tiles* utilizar.

Los diferentes repartos utilizados son los mismos que en el apartado anterior 2x3, 3x2, 6x1 y 1x6. A continuación se muestran resultados obtenidos, los cuadros representan el mismo orden que en la suma de n números por lo cual no repetimos la tabla de *tiles* en ejecución por cuadro que se puede ver en la tabla 4, 6,8 y 10:

Tabla de tiempo de ejecución usando reparto representado en Tabla 6:

		Columna comienza región 3x2			
		0	1	2	3
Fila comienza región 3x2	0	29.60s	29.62s	29.91s	30.15s
	1	29.32s	29.59s	30.24s	29.50s
	2	30.25s	30.17s	30.32s	30.29s
	3	29.98s	30.32s	30.19s	30.13s
	4	30.27s	30.27s	30.12s	29.75s

Tabla 11Representación tiempos tilera multiplicación matriz1000x1000 usando una matriz de tiles de 3x2. (Resultados en segundos)

Max: 30.32 Min: 29.32 Media: 29.99

Tabla de tiempo de ejecución usando reparto representado en Tabla 4:

		Columna comienza región 2x3				
		0	1	2	3	4
Fila comienza región 2x3	0	29.42s	29.58s	30.10s	30.15s	30.39s
	1	29.72s	29.34s	29.56s	29.57s	29.67s
	2	30.30s	29.97s	29.79s	30.11s	29.93s
	3	30.16s	30.00s	30.02s	30.22s	29.72s

Tabla 12. Representación tiempos tilera multiplicación matriz1000x1000 usando una matriz de tiles de 2x3. (Resultados en segundos)

Max: 30.39 Min: 29.34 Media: 29.88

Tabla de tiempo de ejecución usando reparto por filas:

Fila	0	30.41s
	1	29.70s
	2	29.27s
	3	29.49s
	4	29.52s
	5	30.35s

Tabla 13. Representación tiempos tilera multiplicación matriz1000x1000 usando una matriz de tiles de 6x1. (Resultados en segundos)

Max: 30.41 Min: 29.27 Media: 29.79

Tabla de tiempo de ejecución usando reparto por columnas:

Columna					
0	1	2	3	4	5
30.14s	30.07s	30.30s	30.29s	30.46s	30.07s

Tabla 14. Representación tiempos tilera multiplicación matriz1000x1000 usando una matriz de tiles de 1x6. (Resultados en segundos)

Max: 30.46 Min: 30.07 Media: 30.22

Por lo que se puede ver de los resultados anteriores ante la distribución 2x3 y 3x2, 2x3 es más rápido en cuanto al máximo y mínimo, pero en cuanto a la media de tiempos es superior por lo que podemos ver que dependiendo del grupo de *tiles* puede ser más lento mientras que en el caso 3x2 tiene más cantidad de tiempos por encima de 30 segundos.

A diferencia de la lectura, la escritura es más eficiente en los *tiles* exteriores e izquierdos. Y la distribución de forma horizontal está más pensada para la escritura obteniendo mejores tiempos. Si se observan con los tiempos obtenidos al multiplicar la misma tabla con un solo *tile* se tiene una aceleración media de x5.7, que considerando los 6 *tiles* utilizados, implica que se tiene una eficiencia de un 95%. Con toda la tarjeta en uso se ejecuta en un tiempo de 6.38, una aceleración de 26.77 (eficiencia de un 74%). La multiplicación sobre la CPU ha sido de 11.90 segundos, lo cual se consigue una aceleración de 1,87 usando toda la tarjeta. Si se utiliza una matriz transpuesta, esto hace que la multiplicación de las dos matrices sea por filas y así consigue menos fallos de caché (y, por tanto un mejor rendimiento en las operaciones de lectura/escritura), los tiempos con toda la tarjeta pasan de 6.38 a 0.625s y de la CPU de 11.90 a 14.72, esto demuestra una gran mejora en la tarjeta suponiendo una aceleración de 14.72 más rápida que la CPU.

4. Cómputo avanzado con TILERA

La ventaja de trabajar con los coprocesadores tilera se tratará a continuación. Se comparará la aceleración en tiempo de ejecución en un ordenador simple, un clúster y en el coprocesador TILE36.

En este apartado empezaremos a ver la las principales ventajas de trabajar con Tilera. Se comparará la aceleración que hay entre el trabajo con un ordenador simple, un clúster y el uso de Tilera. A continuación veremos y explicaremos en qué casos es más conveniente una cosa u otra.

4.1. Cálculo del número PI

Teniendo un círculo de radio r y dividiendo este en 4 sectores iguales, si se selecciona uno de dichos sectores, el área del sector será $(\pi r^2)/4$ y el área del cuadrado de radio r en el que está metido dicho sector seria (r^2). Si se elige un punto al azar dentro del cuadrado que albergue un sector, siendo dos de los lados de dicho cuadrado los radios delimitadores de éste, el suceso de que dicho punto caiga dentro del círculo completo será una variable aleatoria que sigue una distribución Bernoulli. La probabilidad de que el punto aleatorio se encuentre dentro del sector circular será igual al área del sector circular (casos favorables) dividida entre el área del cuadrado (casos posibles), considerando que el radio es igual a uno se da entonces una probabilidad p= $\frac{\pi}{4}$.

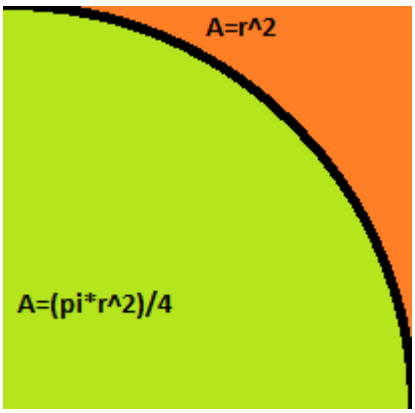

Figura 13- Ilustración área círculo y cuadrado

Para calcular una aproximación al número π con lo dicho anteriormente se puede considerar que si se ponen n números aleatorios y k de ellos caen dentro del sector circular, tenemos que $\frac{k}{n} = \frac{\pi * r^2}{4}$. Por lo cual, generaremos n puntos (x, y) aleatorios dentro de un cuadrado de radio r y mediante Pitágoras veremos si este punto es menor que el radio del circulo $1 \geq \sqrt[2]{x^2 + y^2}$, sumaremos el número de puntos dentro del círculo (k) y finalmente calcularemos pi siendo $\pi = \frac{k}{n} * 4$.

En el Código 1 podemos ver como seria en una computadora serie:

```
gettimeofday(&inicio,NULL);
for(i = 0, dentroCirculo = 0; i < PRUEBAS; i++){
    puntoX = (double)(rand()%RADIO) + (double)(rand()/RAND_MAX);
    puntoY = (double)(rand()%RADIO) + (double)(rand()/RAND_MAX);
    if(pow(puntoX,2) + pow(puntoY,2) <= cuadrado)
        dentroCirculo++;
}
pi = (float)((float)(dentroCirculo * 4)/(float)PRUEBAS);
gettimeofday(&fin,NULL);
```

Código 1. Algoritmo pi simple.

Se genera un punto dentro del cuadrado y se ve si ese punto está dentro del conjunto de puntos que hay dentro del radio de coordenadas del círculo. Una vez se calculan el número de puntos dentro del círculo se calcula el porcentaje de puntos que han caído dentro del circulo de todos los probados.

Para Tilera se utiliza el mismo proceso con la diferencia de que se reparten el número de números aleatorios entre los *tiles*, que calculan el número de puntos dentro y realizan la suma de los resultados parciales calculados por cada *tile*, para posteriormente enviar el resultado final al *tile* 0 que calcula el porcentaje e imprime el resultado (Código 3). Para paralelizar varios *tiles* hemos usado la función parallelize (Código 2), en ella se crean procesos hijos y se establecen en el primer *tile* disponible.

```
static int parallelize(int count)
{
  cpu_set_t cpus;

  //Para establecer que CPUs están en el conjunto
  if (tmc_cpus_get_my_affinity(&cpus) != 0)
    tmc_task_die("Failure in 'tmc_cpus_get_my_affinity()'.");

  //Vemos que tengamos CPUs suficientes
  if (tmc_cpus_count(&cpus) < count)
    tmc_task_die("Insufficient cpus (%d < %d).", tmc_cpus_count(&cpus), count);
  //librería task (gestión de tareas) -> función vigilar hilos
  int watch_forked_children = tmc_task_watch_forked_children(1);

  int rank;
  for (rank = 1; rank < count; rank++)
  {
    pid_t child = fork();
    if (child < 0)
      tmc_task_die("Failure in 'fork()'.");
    //creación hijo, queda elegir cpu
    if (child == 0)
      goto done;
  }
  rank = 0;

  (void)tmc_task_watch_forked_children(watch_forked_children);

 done:
  //selecciona la cpu correspondiente
  if (tmc_cpus_set_my_cpu(tmc_cpus_find_nth_cpu(&cpus, rank)) < 0)
    tmc_task_die("Failure in 'tmc_cpus_set_my_cpu()'.");

  return rank;
}
```

Código 2. Paralización de tiles creando hijos y asignándolo a un tile libre.

```
void calculo_pi(cpu_set_t* cpus,int rank){


    double x, y, pi =0;

    unsigned long  long i;
    unsigned long long  totalDentro=0;
    unsigned long long  TOTAL_PUNTOS = (NUM_CPUS * PUNTOS_POR_CPU);


    srand(time(NULL));


    for (i = 0, totalDentro = 0; i < PUNTOS_POR_CPU; i++){
        x = (double)(rand()%RADIO_CIRCULO) + (double)(rand()/RAND_MAX);
        y = (double)(rand()%RADIO_CIRCULO) + (double)(rand()/RAND_MAX);
        if((pow(x,2) + pow(y,2))  <= CUADRADO_CIRCULO){
            totalDentro++;
        }
    }

    //Buscamos la siguiente CPU
    int next_rank = rank + 1;
    if (next_rank == NUM_CPUS)
    next_rank = 0;

    //esta función lo que hace es mandar a la cpu mayor su suma, la cpu mayor suma lo suyo mas siguiente
    // y la última se lo manda a cpu 0 que recibirá la suma total y con ello imprimirá el resultado
    int next_cpu = tmc_cpus_find_nth_cpu(cpus, next_rank);
    DynamicHeader header = tmc_udn_header_from_cpu(next_cpu);



    if (rank == 0){
        if(rank != next_rank){
            tmc_udn_send_1(header, UDN0_DEMUX_TAG, totalDentro);
            totalDentro = tmc_udn0_receive();
        }
        pi = ((long double)(totalDentro*4)/(long double)TOTAL_PUNTOS);
    }else{
        uint64_t accum = tmc_udn0_receive();
        accum += totalDentro;
        tmc_udn_send_1(header, UDN0_DEMUX_TAG, accum);
    }

}
```

Código 3. Calculo pi y comunicación para suma total entre tiles.

Para la prueba se ha calculado el número pi poniendo 10^3números aleatorios y calculando el tiempo de ejecución para realizar dicho problema en segundos, luego se ha multiplicado por 10 la cantidad de números hasta llegar a 10^{10} números aleatorios. En la Tabla 15 se pueden ver los tiempos dependiendo del número de puntos y arquitectura utilizada.

Número de puntos	Simple	Clúster 10 ordenadores	Tilera 10 tiles	Tilera 20 tiles	Tilera 36 tiles
10^3	$9{,}50 * 10^{-5}$s	$2{,}47 * 10^{-2}$s	$2{,}63 * 10^{-2}$s	$2{,}94 * 10^{-2}$s	$4{,}88 * 10^{-2}$s
10^4	$9{,}52 * 10^{-4}$s	$2{,}45 * 10^{-2}$s	$2{,}23 * 10^{-2}$s	$3{,}96 * 10^{-2}$s	$4{,}28 * 10^{-2}$s
10^5	$9{,}46 * 10^{-3}$s	$2{,}48 * 10^{-2}$s	$3{,}23 * 10^{-2}$s	$2{,}51 * 10^{-2}$s	$5{,}60 * 10^{-2}$s
10^6	$6{,}46 * 10^{-2}$s	$3{,}41 * 10^{-2}$s	$1{,}13 * 10^{-1}$s	$7{,}70 * 10^{-2}$s	$7{,}10 * 10^{-2}$s
10^7	$6{,}07 * 10^{-1}$s	$1{,}08 * 10^{-1}$s	$9{,}03 * 10^{-1}$s	$5{,}01 * 10^{-1}$s	$3{,}30 * 10^{-1}$s
10^8	6,00s	$8{,}66 * 10^{-1}$s	9,00s	4,65s	2,81s
10^9	59,83s	8,45s	91,95s	47,50s	27,09s
10^{10}	596,59s	86,07s	935,23s	485,04s	272,49s

Tabla 15.Tiempos en calcular número pi con x puntos aleatorios.

Con estos tiempos se ve que es a partir de los 1000000 puntos cuando empieza a ser rentable el utilizar sistemas de apoyo computacional. En el caso de la tarjeta tilera, para un trabajo genérico como es el cálculo del número PI (generar números aleatorios, divisiones y sumas), con 10 *tiles* no se llega a tener la misma potencia de cálculo que un ordenador, con 20 *tiles* ya superamos levemente la potencia del ordenador y es con el uso de toda la tarjeta con la que llegamos a tener algo más del doble de potencia. Podemos ver en el Gráfico 1 como el uso de MPI con 10 ordenadores consigue una superior aceleración respecto al resto.

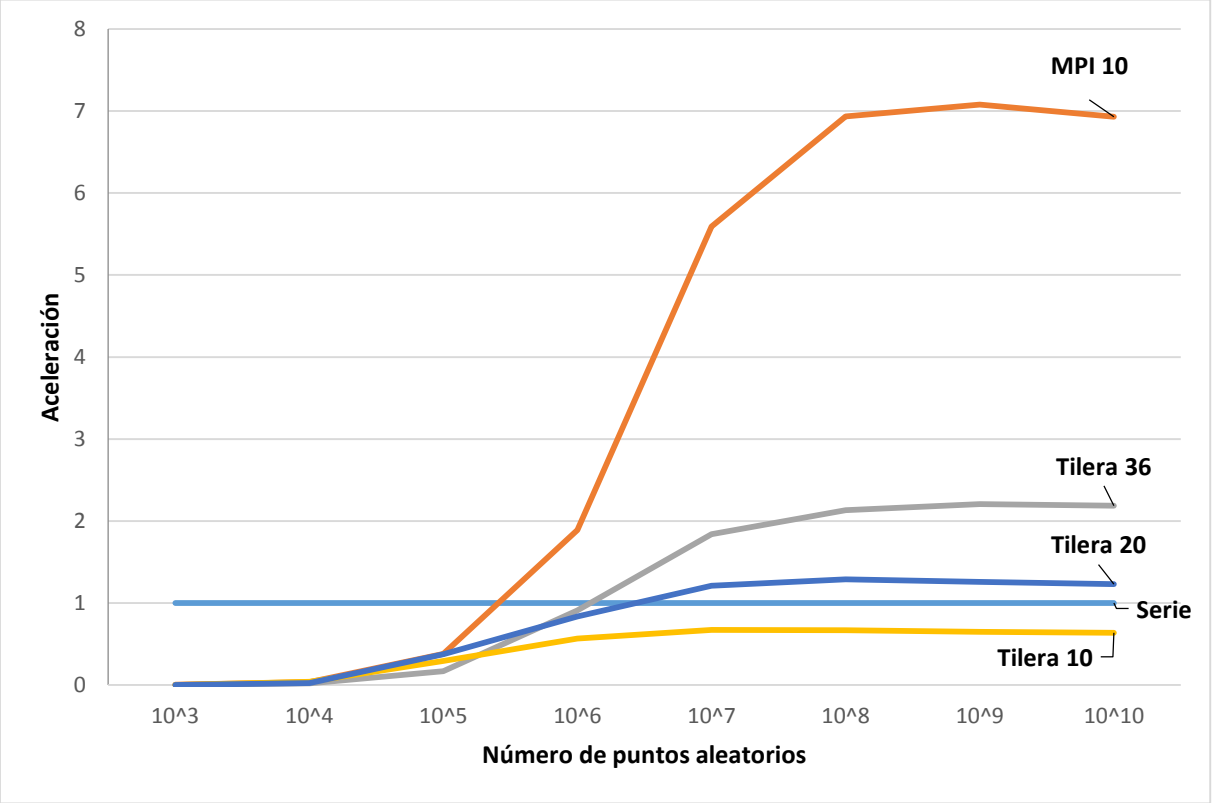

Gráfico 1. Aceleración calculo PI por número de puntos aleatorios.

En cambio si se compara esta aceleración con el coste energético, como se ve en el Gráfico 2, en la arquitectura MPI se han utilizado 10 ordenadores, lo que significa (considerando un consumo medio de ordenador de 70W) un consumo total de 700W, lo hace que pase de una aceleración máxima de casi 6.8 puntos a 7/700 = 0.01 de aceleración por W utilizado, mientras que la tarjeta tilera (70W ordenador + 22W tarjeta tilera) consigue una aceleración de 2.2/92=0.024 por W. Esto es una aceleración de casi 2.5 veces superior por vatio utilizado a lo que hay que sumar el coste económico.

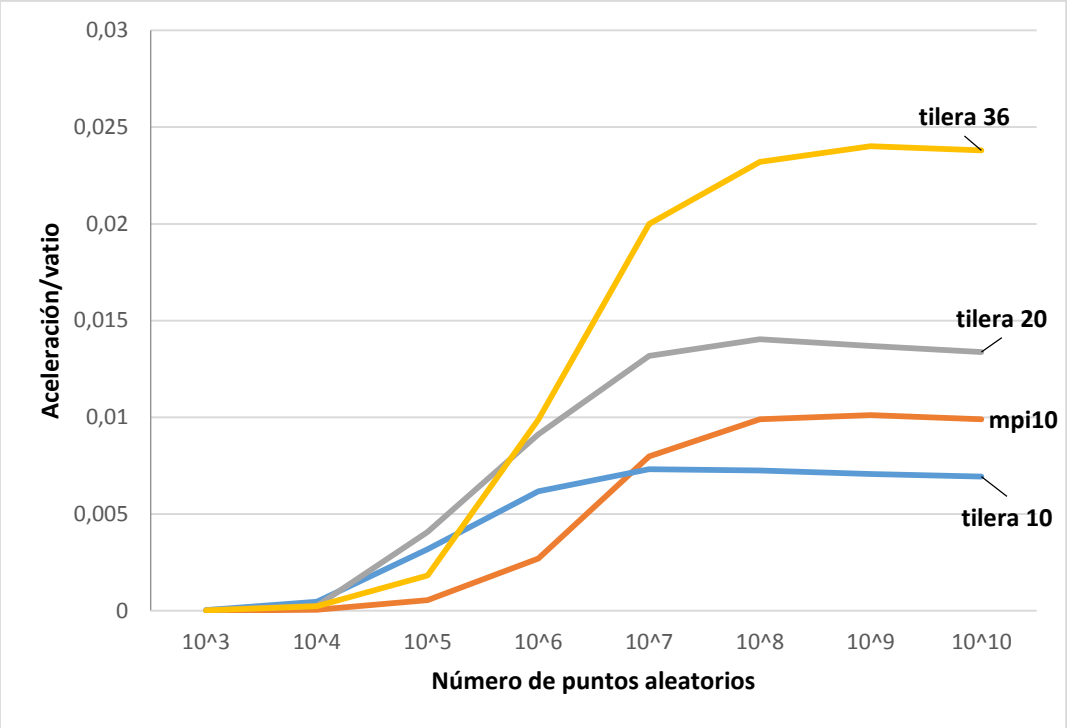

Gráfico 2. Aceleración calculo PI por número de puntos aleatorios considerando el consumo utilizado.

4.2. Criptografía con tilera

Tilera ha introducido una librería MiCA, esta es una interfaz común para diferentes funciones de aceleración para la criptografía o la compresión de archivos. Esta API permite asignar y configurar recursos proporcionando una interfaz general para iniciar operaciones sobre zonas de memoria. Contiene hasta 40 contextos hardware siendo cada uno de ellos un conjunto de registros que poseen la información necesaria para realizar una operación. En este caso usaremos MiCA junto a la librería gxcr.h para la encriptación de datos y se compararán los resultados con la librería OpenSSL, proyecto de software libre que proporciona un paquete de herramientas de administración y bibliotecas relacionadas con la criptografía.

El algoritmo elegido es el AES-CBC 128, algoritmo de encriptación por bloques de 16Bytes con al que se le añade una función XOR entre el bloque encriptado anterior y el texto plano a encriptar para aumentar la dificultad de romper este cifrado (Representación cifrado en Figura 14). Este añadido hace que sea difícil su paralelización al depender un bloque del otro.

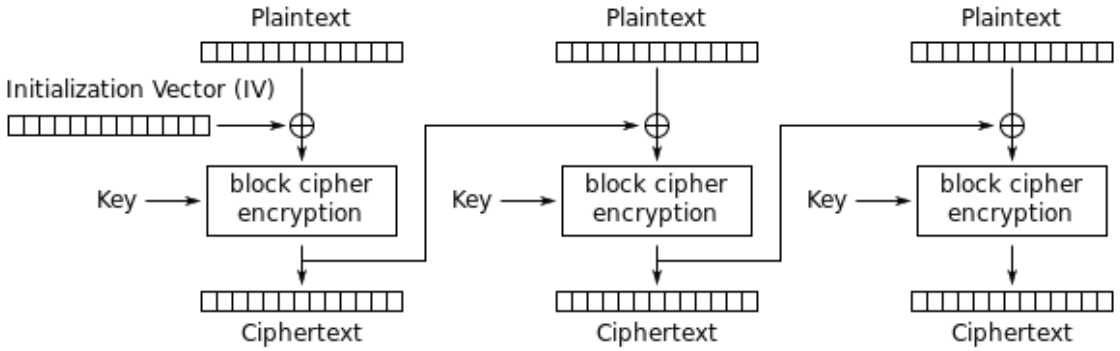

Figura 14. Algoritmo cifrado bloque CBC.

Para este tipo de algoritmos los coprocesadores Tilera tienen motores de aceleración MiCA (Véase en Figura 10). Estos generan un contexto sobre una zona de memoria y realizan un algoritmo propio sobre esa zona de memoria. A continuación se muestra un ejemplo sencillo:

Primero se debe generar el contexto MiCA, donde se le asigna que motor de aceleración se va a usar y a continuación se le asigna una página con un puntero donde empieza la memoria y un tamaño.

```
gxio_mica_context_t cb;
gxio_mica_init(&cb, GXIO_MICA_ACCEL_CRYPTO, 0);
result = gxio_mica_register_page(&cb, page0, page_size, 0);
```

Código 4. Inicialización contexto MiCA.

Una vez se tiene registrada la zona de memoria debemos cargar el código a realizar dicho motor. Para realizar esta tarea, se genera el código y se le asigna una zona de memoria donde registrar dicho código. Con gxcr_init_context se genera un código en la zona de memoria apuntada por enc_metadata donde está el algoritmo de encriptación AES_CBC_128 (cuarto argumento) y, en este caso, se añade el algoritmo MD5.

```
gxcr_context_t encrypt_context;
result = gxcr_init_context(&encrypt_context, enc_metadata, TEST_MEM_SIZE,
                           GXCR_CIPHER_AES_CBC_128, GXCR_DIGEST_MD5,
                           key, NULL, 1, 1, 0);
result = gxcr_do_op(&cb, &encrypt_context, src_data, dst_data, TEST_LEN);
```

Código 5. Inicialización contexto AES_CBC_128.

Para finalizar, se lanza la ejecución al motor MiCA con la función gxcr_do_op, en ella se envía el contexto mica donde indica en qué modo está el acelerador y sobre qué zona de memoria se trabaja, el contexto donde se encuentra el algoritmo a ejecutar, sobre qué zona de memoria se sacan los datos, donde guardar los resultados y el tamaño de memoria sobre la que se va a realizar el algoritmo.

Para este caso se ha encriptado texto plano empezado con 256 bytes hasta 65536 bytes y se ha comparado la aceleración con el mismo cifrado sobre el mismo algoritmo que nos ofrece la librería OpenSSL utilizando un ordenador del clúster en serie. Para comparar dichos algoritmos se ha utilizado el mismo método que en problemas anteriores comparándolos con el tiempo necesario para la ejecución del problema. Los tiempos tardados (en segundos) para la encriptación de datos dependiendo de la cantidad de datos en bytes se muestran en la Tabla 16.

Cantidad de Bytes a encriptar	Tiempo tardado en encriptar los datos ordenador serie.	Tiempo tardado en encriptar los datos motor aceleración MiCA.
256	$1{,}80 * 10^{-5}$s	$2.40 * 10^{-5}$s
512	$2.30 * 10^{-5}$s	$2.40 * 10^{-5}$s
1024	$3.30 * 10^{-5}$s	$2.30 * 10^{-5}$s
2048	$5.30 * 10^{-5}$s	$2.40 * 10^{-5}$s
4096	$9.40 * 10^{-5}$s	$2.70 * 10^{-5}$s
8192	$1.75 * 10^{-4}$s	$3.20 * 10^{-5}$s
16384	$3.52 * 10^{-4}$s	$4.10 * 10^{-5}$s
32768	$6.74 * 10^{-4}$s	$6.10 * 10^{-5}$s
65536	$1.34 * 10^{-3}$s	$1.00 * 10^{-4}$s

Tabla 16. Tiempos en realizar algoritmo AES_CBC_128 sobre x bytes de memoria.

De forma individual podemos ver como aumentando el tamaño de datos a encriptar al doble, el método serie aumenta su tiempo un poco menos del doble, esto se debe a datos en cache que ya están cargados, en cambio con MíCA, hasta los 4KBytes de datos es prácticamente el mismo tiempo, esto es debido a que tarda al principio más en cargar el código de encriptación en memoria que en realizar el algoritmo. Una vez superamos este tiempo, vemos como (quitando los dos segundo de cargar el código) ya si aumenta al doble.

Esto significa una aceleración exponencial llegando a los 14 puntos (14 veces más rápido MiCA) al encriptar 64KBytes

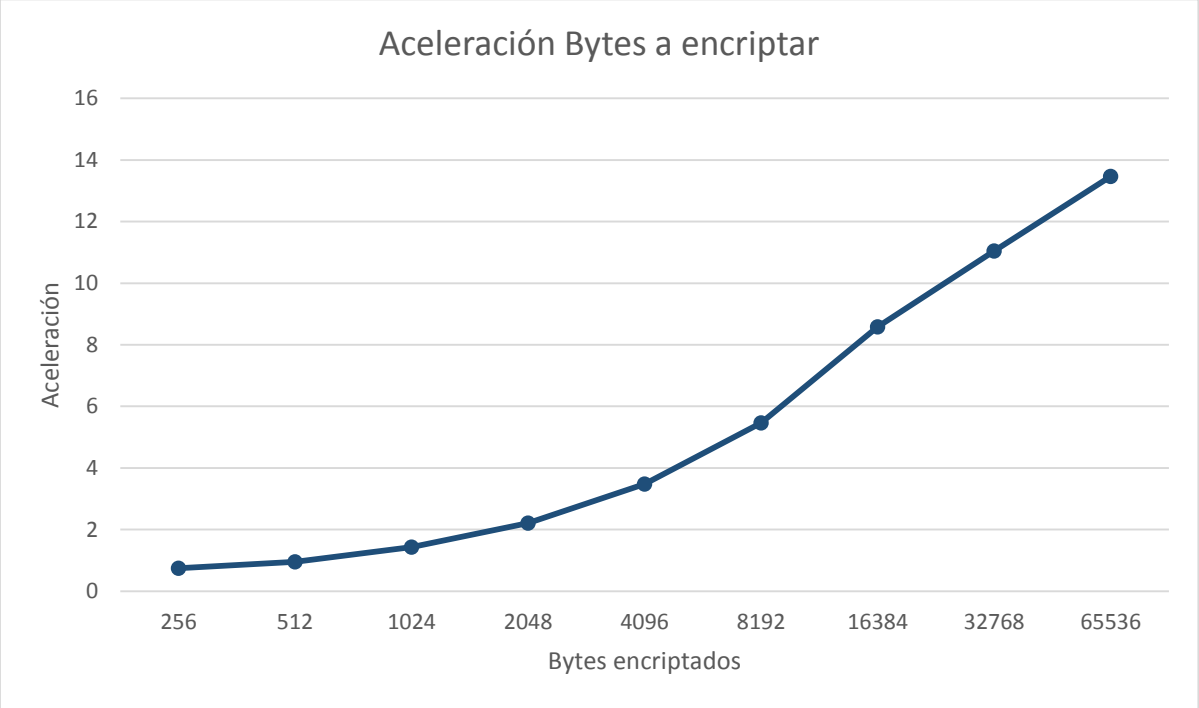

Gráfico 3. Aceleración en la realización algoritmo AES_CBC_128 con MiCA según tamaño a encriptar.

Este algoritmo como se puede ver en la Figura 14, es altamente dependiente, lo que hace que su paralelización sea muy limitada. El uso de un clúster en este caso sería desaconsejable al tener que usar una zona de memoria compartida y conllevar esto una gran cantidad de mensajes entre los propios procesadores lo que lo hace muy lento además de un procesamiento prácticamente lineal al depender cada zona a encriptar de la anterior.

Con este problema se ha hecho un estudio de una de las filosofías del coprocesador hablado anteriormente, no la realización de un problema de forma altamente paralela, sino la utilización de un motor especifico altamente especializado para la realización de determinadas tareas, en este caso el motor MiCA, especializado para la realización de algoritmos sobre zonas de memoria.

5. Conclusiones

A lo largo del presente trabajo se ha realizado un estudio de la programación paralela basada en coprocesadores, centrado en el coprocesador desarrollado por la empresa Tilera. Cabe destacar que éste ha conllevado la adquisición de numerosos conocimientos no adquiridos durante el grado, así como el refuerzo de ciertos conceptos que se habían tratado a lo largo del mismo, pero que son llevados aquí a un nuevo nivel con su puesta en práctica en entornos reales.

El aspecto principal de este proyecto ha sido el estudio sobre el potencial computacional ofrecido por el coprocesador TILE36. Este coprocesador puede ejecutar diferentes tareas a la vez, por lo que se ha estudiado en que zonas y que reparto es el más adecuado dependiendo de la tarea a realizar. Para la lectura de memoria, se ha visto como los *tiles* centrales tienen un mejor acceso a los bancos de memoria RAM. Por el contrario, son los *tiles* izquierdos los que tienen menor latencia en la escritura de datos en los bancos. En cuanto al trabajo en grupo de *tiles*, es mejor un reparto lo más cuadrado posible para la comunicación entre los mismos.

Comparando esta tarjeta con una arquitectura paralela basada en el paso de mensajes (Clúster), la aceleración del clúster es muy superior. Si se tiene en consideración la aceleración en ejecutar un problema por vatio consumido, se observa que es aquí donde la tarjeta tilera supera al clúster debido al bajo consumo de la misma. Por ello, sería interesante estudiar el rendimiento de un clúster de tileras, pues con múltiples unidades trabajando en paralelo podría lograrse un rendimiento igual o superior al del clúster pero reduciendo el consumo energético del sistema de procesamiento.

Estos coprocesadores tienen motores de aceleración para recursos específicos como son los MiCA (Multistream iMesh Crypto Accelerator). En concreto, la tarjeta con la que se ha trabajado (Tile-Gx36), viene con dos motores MiCA. Estos motores están pensados para la realización de código sobre grandes cantidades de datos a la vez, por lo que se ha visto una mayor aceleración relacionada con el aumento de bytes a encriptar.

A grandes rasgos, este proyecto analiza los componentes del coprocesador TILE36, con objeto de lanzar diferentes tareas a la vez sobre el mismo, siendo los *tiles* centrales más útiles para la lectura de datos y los *tiles* del exterior izquierdo los más aptos para la escritura. Para la criptografía, con el uso de los motores MiCA, los resultados con los *tiles* de forma independiente han sido muy parecidos. Las diferencias entre los tiempos de ejecución entre los diferentes *tiles* son menores a un microsegundo. Esta es la razón por lo cual se recomienda hacer una gestión de los *tiles* con repartos lo más cuadrados posibles y dejar dos *tiles* libres para las funciones criptográficas al haber en este caso dos motores MiCA.

Por último, es interesante mencionar una serie de posibles líneas de investigación futuras haciendo uso del coprocesador Tilera, como su viabilidad como elemento para computar directamente desde la red, su posible empleo como elemento dentro de un sistema híbrido, en el cual se puedan emplear todos los elementos de cómputo presentes (procesador, GPU, Tilera...) lográndose una aceleración global importante al emplearse en cada momento varios procesadores, siendo la única limitación su disponibilidad o el estudio para el desarrollo de una librería estándar basada en OpenMP para el aprovechamiento de los cores de los propios tiles mediante la generación de hilos a partir de simples directivas.

6. Bibliografía

[1] TOP500, online: http://www.top500.org

[2] Nvidia, online: http://www.nvidia.es

[3] Xeon Phi, Intel online: http:/www.intel.es/content/www/es/es/processors/xeon/xeon-phi-detail.html

[4] CPU World, online: http://www.cpu-world.com/news_2012/2012013001_Tilera_announces_availability_of_36-_and_16-core_CPUs.html

[5] slash gear, online: http://www.slashgear.com/tilera-tile-gx-puts-36-core-pc-on-a-half-height-pcie-card-30211246/

[6] Noticias sobre tilera Gx-36, online: http://www.maximumpc.com/article/news/tilera_adds_36core_chip_its_arsenal

[7] Oficial tilera, actualmente EZchip, online: http://www.tilera.com/

[8] Nuevos procesadores tilera de 36 y 16 núcleos, online: http://www.siliconweek.es/projects/nuevos-procesadores-de-tilera-de-16-y-32-nucleos-18992

[9] Universidad Autónoma de Madrid, arquitectura de sistemas paralelos, online: https://moodle.uam.es/course/view.php?id=8811

[10] A. Tousimojarad and W. Vanderbauhede. *Number of tasks, not threads, is keys.* Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on.

[11] A. Tousimojarad and W. Vanderbauhede. *Cache-aware Parallel Programming for Manycore Processers.* International symposium on Highly- Efficient Accelerators and Reconfigurable Technologies (HEART2013), Edinburgh, Scotland, June 13-14, 2013.

[12] D. Sánchez-Román, V. Moreno, S. López-Buedo, G. Sutter, I. González, F.J. Gómez-Arribas, J. Aracil. *FPGA acceleration using High-level languages of a Monte-Carlo method for pricing complex options.*

[13] Lin Gan, Haohuan Fu, Wei Xue, Yangtong Xu, Chao Yang, Xinliang Wang, Zihong Lv, Yang You, Guangwen Yang, Kaijian Ou. *Scaling and analyzing the stencil performance on multicore and manycore architectures.* . Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on.

[14] Noticia sobre tilePro 64, online:

http://www.theinquirer.net/inquirer/news/1006963/tilera-releases-core-chip

[15] M. Cabanas-Holmen, E.H. Cannon, A. Kleinosowski, J. Ballast, J. Killens, J. Socha.*Clock and Reset Transients in a 90 nm RHBD Single-Core Tilera Processor.* Nuclear Science, IEEE Transactions on (Volume: 56, Issue: 6)

[16] C.Y. Villalpando, A.E. Johnson, R. Some, J. Oberlin, S. Goldberg. *Investigation of the Tilera processor for real time hazard detection and avoidance on the Altair Lunar Lander. Aerospace Conference, 2010 IEEE.*

[17]H. Ya-Fei, T. Shau-Yin, K. Chung-Ta, L. Huan-Yu, H. Shih-Chieh. *Parallel Implementation and Performance Prediction of Object Detection in Videos on the Tilera Many-Core Systems.* Pervasive Systems, Algorithms and Networks (ISPAN), 2009 10th International Symposium on

[18] Jing Xu, Hanbo Wang, Wei Liu, Xiaojun Hei, *Towards High-speed Real-time HTTP Traffic Analysis on the Tilera Many-core Platform,* in 2013 IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing.

7. Anexos

7.1. Problemas encontrados durante el desarrollo

Cierres inesperados del entorno de desarrollo Eclipse.

Intentos realizados para solucionarlo:

- Reinstalación entorno: Se han reinstalado todos los paquetes Eclipse y actualizado a la última versión. Resultado: Se sigue cerrado. Eclipse se abre y crea proyectos, al intentar abrir los códigos del proyecto o ejecutar se cierra sin dejar señal en el log del error del cierre.
- Actualización de todos los paquetes del sistema operativo: Se actualizan todos los paquetes en busca de paquetes obsoletos que provoquen el error y como consecuencia cierre de Eclipse. Número total de paquetes actualizados/eliminados: 735. Resultado: Tras la actualización de paquetes persiste el error.
- Reinstalación paquetes GTK+, GTK2 y GLPK: Estos paquetes fueron necesarios en la primera instalación. Se repite la instalación de los mismos para ver si hubo algún error en la instalación anterior. Resultado: Siguen los cierres.
- Compilación paquetes tilera e instalación de los mismo.

7.2. Librerías específicas:

En esta sección hablaremos de las librerías utilizadas durante el trabajo:

- `Cpus.h`: Librería específica para la gestión de *tiles* e inicialización de una tarea. Esta librería proporciona una variedad de rutinas para la manipulación de conjuntos de *tiles* pudiendo:
  - Añadir/Eliminar *tiles* de un conjunto para la realización de operaciones.
  - Unir tareas haciendo que estas sean ejecutadas por un conjunto seleccionado de *tiles*.
  - Identificar cada *tile* con un número lógico dependiendo de su posición y consiguiendo de esta manera una gestión de cada *tile* más sencilla.

  Funciones que da esta librería:

  - `tmc_cpus_clear(cpu_set_t * s)`: Elimina el conjunto s de CPUs.
  - `tmc_cpus_add_cpu(cpu_set_t * s, int unsigned cpu)`: Añade la CPU "cpu" al conjunto de CPUs.
  - `tmc_cpus_remove_cpu(cpu_set * s, unsigend int cpu)`: Elimina la CPU "cpu" del conjunto de CPUs s.
  - `tmc_cpus_add_cpus(cpus_set_t * s, cpu_set_t * CPU)`: Inserta en el conjunto de CPUs s el conjunto de CPUs CPU.
  - `tmc_cpus_remove_cpus(cpus_set_t * s, cpus_set_t * cpu)`: Elimina el conjunto de CPUs "cpu" del conjunto s.
  - `tmc_cpus_intersect_cpus(cpus_set_t *s, cpus_set_t * cpu)`: Guarda la intersección de los dos conjuntos en el conjunto s.
  - `tmc_cpus_has_cpu(cpu_set_t * s, int cpu)`: Devuelve 1 en caso de que la cpu este en el conjunto s, 0 en el caso contrario.
  - `tmc_cpus_count(cpu_set_t * s)`: Devuelve el número de cpus que hay en el conjunto s.
  - `tmc_cpus_find_nth_cpu(cpu_set_t * s, int n)`: Devuelve que cpu ocupa la posición n del conjunto s.
  - `tmc_cpus_find_first_cpu(cpu_set_t *s)`: Devuelve la primera cpu del conjunto s.
  - `tmc_cpus_get_my_affinity(cpu_set_t * s)`: Establece el conjunto de CPUs
  - `tmc_cpus_set_my_cpu(int cpu)`: Establece en que CPU se realiza la tarea.

- `Task.h`: Librería específica para la gestión de tareas entre los *tiles*, generando un core como maestro y el resto como esclavos. Así el maestro puede gestionar o eliminar procesos esclavos. Esta librería junto a la anterior son utilizadas para todos los problemas, con "`cpus.h`" generamos un pastor como anteriormente se ha dicho y con "task" se tiene el control para la finalización de todos los *tiles* en caso de error.

  Funciones principales utilizadas de la librería:

  - `tmc_task_die(char * format)`: Escribe un mensaje de error y termina el proceso y todos los procesos que sean observador por el proceso que observaba a este proceso.
  - `tmc_task_watch_forked_children(int flag)`: Establece un core principal para observar el resto de cores.

- `alloc.h`: Rutinas para la asignación de memoria especificando la memoria cache, controlador y tamaño. Esta API trabaja bajo la estructura predefinida `tmc_alloc_t` con al que se podrá establecer la dirección, longitud, tamaño de página etc. Se recomienda inicializar `tmc_alloc_t` por defecto con la estructura `TMC_ALLOC_INIT` y una vez inicializado modificar los valores que se deseen mediante funciones getters y setters. Esta rutina nos da la posibilidad de reservar memoria en puntos concretos de la memoria por lo que es de gran utilidad para estudios posteriores sobre la propia tilera.

  Funciones principales utilizadas de la librería:

  - `tmc_alloc_set_shared(tmc_alloc_t * alloc)`: Establece la memoria asignada como memoria compartida para poder ser accedida por los procesos padres e hijos. Tener en cuenta que una vez sean creados los hijos, la memoria nueva asignada no será compartida, solo se comparte con los hijos creados después de la asignación de memoria.
  - `tmc_alloc_map(tmc_alloc_t * alloc, size_t length)`: Asigna una cantidad específica de memoria devolviendo un puntero con la posición de la misma.
  - `tmc_alloc_map_at(tmc_alloc_t * alloc,void * start, size_t length)`: Asigna una cantidad de memoria especifica en una dirección concreta, en caso de esta ocupada esta dirección devuelve un puntero NULL como error.
  - `tmc_alloc_remap(void * addr, size_t old, size_t new, int flags)`: modifica la longitud y/o dirección de una memoria reservada anteriormente.
  - `tmc_alloc_unmap(void * start, size_t length)`; Libera memoria asignada.

- `cmem.h`: Mientras que la librería alloc se utiliza para paginar memoria, cmem es utilizada para el mapeo de memoria virtual y generación de punteros para todos los procesos que apunten a la misma dirección de memoria generando la memoria compartida.

- `malloc.h`: Las rutinas estándar de malloc no son compatibles con la librería mspace. Por lo cual se creó la librería `tmc_malloc` para conseguir compatibilidad entre las mismas. Esta librería sobrescribe la librería estándar con las nuevas funciones aportadas por mspace.

- `mem.h`: Da soporte para mover bloques de memoria a dentro o fuera de cada core, de forma sincronizada para que todos los elementos tengan los mismos datos de memoria guardando la coherencia de datos. También se utiliza para la sincronización de cores por medio de memoria compartida.

- `mspace.h`: Librería utilizada para trabajar con diferentes pilas de memoria dándoles a las pilas propiedades distintas, ya sea de accesibilidad de los diferentes cores como localidad. La librería esta optimizada para el procesador aprovechando mejor las características del mismo que la librería estándar.

- `sync.h`: Librería muy útil para la sincronización de tareas, acceso a memoria etc. mediante el uso de semáforos.
- `udn.h`: User Dynamic Network, crea caminos para la comunicación entre los tiles.

  Funciones principales utilizadas de la librería:

  - `tmc_udn_header_from_cpu(int cpu)`: Genera una cabecera de red dinámica para conexión con dicha cpu.
  - `tmc_udn_send_1(DynamicHeader dest, uint32_t tag, uint_reg_t n1)`: Envía un mensaje a dest, cpu asignada con la función anterior.
  - `tmc_udn0_receive ()`: lee mensaje de la cola de mensajes udn0
  - `tmc_udn_init()`: Inicializa la conexión udn.
  - `tmc_udn_activate()`: activa la conexión udn para tener acceso a la red.

- `MiCA:` Multicore iMesh Coprocessing Accelerator, librería creada con la finalidad de acelerar la encriptación o compresión de archivos. Esta API permite asignar recursos de la memoria así como memoria de origen, destino etc.