Explotación de la potencia de procesamiento mediante paralelismo: un recorrido histórico hasta la GPGPU

Francisco Charte Ojeda[1], Antonio J. Rivera Rivas[2], Francisco J. Pulgar Rubio[2], María J. del Jesus Díaz[2]

[1] Departamento de Ciencias de la Computación e Inteligencia Artificial. E.T.S.I.I.T. Universidad de Granada.
francisco@fcharte.com
[1] Departamento de Informática. E.P.S. Universidad de Jaén.
{arivera,fjpulgar,mjjesus}@ujaen.es

Resumen. La mejora en los sistemas de fabricación de semiconductores, con escalas de integración crecientes durante décadas, ha contribuido a incrementar de forma espectacular la potencia de los sistemas de cómputo en sus diversas formas, ordenadores personales y portátiles, móviles, tabletas, consolas, etc. Esa evolución, no obstante, también ha encontrado obstáculos por el camino que, entre otros aspectos, acabaron hace varios años con la escalada en las frecuencias de reloj. En la actualidad la potencia de un procesador ya no se mide exclusivamente en GHz, sino que también influyen factores como el número de núcleos de procesamiento y el diseño de estos. En el presente artículo se lleva a cabo un recorrido histórico de cómo el paralelismo ha ido adecuándose al hardware disponible en cada momento con el objetivo de obtener el mayor provecho del mismo.

Palabras Clave: Paralelismo, tiempo compartido, núcleo, CPU, GPU.

Abstract. Due to the improvement of semiconductor manufacturing technologies, and higher integration scales in the last decades, the power of computing devices has experienced an impressive growth. However, some obstacles have been also found along the way. As a consequence, the battle for reaching higher clock frequencies almost ended a few years ago. Nowadays, the power of processors is not measured exclusively using GHz. Other factors, as the number of cores and their inner design, also have a large impact. This paper provides an historical review on how parallelism techniques have been adapted over time to overcome these changes aiming to better exploit the available hardware.

Keywords: Parallelism, time sharing, core, CPU, GPU

1 Introducción

El conocimiento que aporta al alumnado la asignatura de ingeniería de computadores es esencial para, entre otros fines, poder extraer del hardware existente en cada momento el mayor beneficio posible. Los detalles de arquitectura correspondientes a la máquina en la que se ejecutará un software, así como la identificación de los lenguajes y herramientas más apropiadas a cada caso, hacen posible el desarrollo de aplicaciones capaces de explotar toda la potencia disponible. Como sería de esperar, tanto el hardware como las soluciones diseñadas para sacar provecho del mismo evolucionan a lo largo del tiempo.

Durante décadas, las mejoras en la potencia de procesamiento de un sistema se debían, fundamentalmente, a un incremento en la frecuencia de funcionamiento de su procesador central o CPU, medida en términos de MHz/GHz. La incorporación de múltiples vías para la descodificación y ejecución de instrucciones, y en ocasiones varias unidades aritmético-lógicas, se sumaban a la citada mejora de velocidad en el reloj que actúa como marcapasos interno de la CPU. A estos siguieron el diseño de procesadores con capacidad para actuar sobre dos hilos de ejecución en paralelo y, posteriormente, los que incorporaban múltiples núcleos de ejecución independientes. La disponibilidad de la infraestructura necesaria para interconectar múltiples sistemas en una red de comunicación rápida, principalmente Ethernet, abrió en su momento una nueva vía para satisfacer la siempre creciente necesidad de procesamiento de datos, empleando múltiples máquinas como si de un único sistema de cómputo se tratase. El último escalón en este camino corresponde a las GPU (*Graphics Processing Unit*), particularmente a las tecnologías GPGPU (*General-Purpose Computation en GPU*) que permiten utilizar estos procesadores, inicialmente diseñados para acelerar el tratamiento de gráficos, como un sistema de cómputo de propósito general.

Paralelamente al propio hardware, las interfaces de programación de aplicaciones (API) ofrecidas por los sistemas operativos y los lenguajes de programación también han ido adaptándose con el objetivo de ofrecer al desarrollador los medios necesarios para explotar las nuevas capacidades. La elección de la herramienta adecuada a cada caso es un aspecto vital que complementa el conocimiento de la arquitectura del hardware.

El objeto de este artículo es ofrecer una visión global sobre cómo la explotación de paralelismo en informática ha ido evolucionando con el paso del tiempo, desde los primeros sistemas que mediante técnicas de tiempo compartido hicieron posible la multitarea hasta las actuales GPU y las diferentes API para aprovecharlas, como son CUDA, OpenCL y WebCL. En la Sección 2 se lleva a cabo un recorrido por la historia de la multitarea y el paralelismo en CPU. La Sección 3 introduce el concepto de GPGPU como evolución del uso de las GPU exclusivamente para tratamiento gráfico. Las tres API antes mencionadas, así como las herramientas asociadas a las mismas, se describen en la Sección 4. Las conclusiones finales se facilitan en la Sección 5, seguida de las correspondientes referencias.

2 Multitarea, paralelismo y computación distribuida

Actualmente resulta habitual contar con un teléfono móvil, una tableta, un ordenador personal y posiblemente también un portátil, dispositivos todos ellos que incorporan microprocesadores con una inmensa capacidad de cálculo, incluyendo la posibilidad de ejecutar múltiples tareas de manera paralela. El mundo, no obstante, no siempre ha sido así. Hace apenas medio siglo los sistemas informáticos ocupaban grandes espacios, requerían sistemas de refrigeración a medida y un gran aporte de energía eléctrica para, a cambio, obtener una fracción de la potencia que hoy ofrece un móvil. Aprovechar esa potencia, obtenida a un muy alto coste, resultaba fundamental, de ahí que algunas de las primeras técnicas de ejecución concurrente surgieran por entonces.

A la hora de planificar el grado de concurrencia de un software o algoritmo hay que diferenciar entre paralelismo SIMD (*Single Instruction Multiple Data*) y paralelismo MIMD (*Multiple Instruction Multiple Data*) [1]. Las GPU son ideales para el primer caso, en el que un mismo conjunto de sentencias se aplica simultáneamente sobre múltiples datos independientes entre sí, produciendo por lo general tantos resultados como datos de entrada existan. Es una configuración ideal para, por ejemplo, realizar operaciones sobre matrices que, dependiendo de su tamaño, pueden ser calculadas en un único ciclo de reloj. Para paralelizar tareas del segundo tipo, en las que flujos de sentencias diferentes se aplican sobre conjuntos de datos que pueden ser independientes o no, es necesario recurrir a los núcleos de la CPU, ya que cada uno de ellos puede ejecutar un programa distinto: un núcleo puede controlar la interfaz de usuario de un programa mientras otro se dedica a procesar datos introducidos en dicha interfaz, tareas paralelas pero que no tienen mucho que ver entre sí.

Los desarrollos descritos en los siguientes puntos de esta sección, relativos a la evolución de las CPU, están en su mayor parte relacionados con el modelo MIMD. En la Sección 3 y Sección 4, centradas en la programación GPGPU, el modelo de programación sería el SIMD.

2.1. Sistemas de tiempo compartido y multitarea

La velocidad a la que un operador puede escribir, a fin de comunicarse con el ordenador a través de una terminal interactiva, es muy inferior a la capacidad de un procesador para ejecutar sus instrucciones. A causa de ello la mayor parte del tiempo el sistema estaría en espera, una situación a la que en la actualidad, con nuestros ordenadores personales, no se le otorga la mayor importancia, pero que resultaba totalmente prohibitivo a mediados de la década de los 60 del pasado siglo. Por entonces los mainframes, que precedieron a la informática personal, suponían una inversión de millones de dólares y el coste de su funcionamiento y mantenimiento era también muy considerable. Es por entonces cuando surge el concepto de tiempo compartido[1] (*time-sharing*) [2] como solución para aprovechar la potencia de aquellas máquinas, ofreciendo a cada uno de los usuarios conectados a las mismas la ilusión de tener a su disposición todos los recursos.

[1] El término *time-sharing* con la acepción empleada aquí fue popularizado por John McCarthy, por entonces profesor del MIT y al que se otorgó el Premio Turing en 1971.

El tiempo compartido en un sistema informático funciona como un maestro de ajedrez jugando una partida múltiple, moviéndose en círculo y cambiando de tablero a tablero retomando la tarea donde la dejó en un momento anterior. A pesar de que el procesador en ningún caso ejecuta paralelamente varias tareas, para los usuarios así lo parece. Equipos de finales de los 60, como el GE-635/645 [3a], facilitaban la implementación de esta técnica, consiguiendo desbancar a sistemas mucho más establecidos, como el IBM System/360 [3b], en organizaciones como DARPA (*Defense Advanced Research Projects Agency*) y Bell Laboratories. En cuanto a software se refiere, Multics, el predecesor de UNIX, fue probablemente el primer sistema operativo diseñado para soportar originalmente servicios de tiempo compartido.

La técnica de tiempo compartido no solo facilitaba el acceso simultáneo de múltiples usuarios a los grandes sistemas, también fue el principio que permitió en sistemas monousuario ejecutar múltiples aplicaciones aparentemente de forma paralela. El microprocesador de los primeros ordenadores personales de IBM, el Intel 8086/8088 [4], carecía de la circuitería necesaria para facilitar el intercambio entre tareas y las unidades para ejecución paralela, pero el uso de las interrupciones hardware, generadas por el reloj interno, el teclado y ratón, hacían posible la ejecución aparentemente simultánea de varias aplicaciones. La infraestructura necesaria para efectuar el intercambio de contexto entre tareas por hardware se incorpora a esta familia de procesadores en 1986, con el procesador de 32 bits 80386 [4].

Debido a la inexistencia de estándares establecidos, construir soluciones multitarea en los 60s/70s implicaba conocer los detalles del hardware concreto en que iban a ejecutarse, así como el sistema operativo que el fabricante había desarrollado para dicho hardware. Por ejemplo, OS/360 ofrecía en 1967 la posibilidad de usar MVT (*Multiprogramming with Variable number of Tasks*) [5] como gestor de tareas con una filosofía análoga a la de los hilos de ejecución actuales.

2.2. Sistemas multiprocesador y multinúcleo

Durante décadas las CPU de los ordenadores se ajustaron fielmente al diseño establecido en la conocida como *arquitectura von Neummann* [6]. El núcleo de esta es el conocido ciclo de recuperación-decodificación-ejecución de instrucciones que, aún hoy, sigue encontrándose en el interior de los modernos microprocesadores. Se trata de una arquitectura inherentemente secuencial, cuya potencia estaba en gran medida limitada por la velocidad a que podían recuperarse las instrucciones del soporte en que se encontraban almacenadas. No obstante, dicha arquitectura ha ido adaptándose paulatinamente para aprovechar las innovaciones surgidas en cada momento, como fue el uso de memoria de tipo semiconductor, mucho más rápida que las cintas de papel y fichas perforadas, para contener las instrucciones del programa en ejecución. En 1970, como parte del proyecto de investigación ILLIAC de la Universidad de Illinois, se desarrolló el Illiac-IV [7], probablemente el primer ordenador con capacidad explícita de paralelismo. Este sistema, con hasta 256 unidades de procesamiento paralelo, fue el primer superordenador conectado a Internet, antes de que los equipos diseñados por Seymour Cray ocupasen dicho trono durante largo tiempo.

La construcción de ordenadores con múltiples zócalos de procesador se popularizó en la década de los 90, concretamente en los servidores de alta gama y las conocidas

como estaciones de trabajo (*workstations*) de alto rendimiento. Cada microprocesador aportaba su unidad de procesamiento interno, compartiendo con los demás micros, alojados en otros zócalos de la misma circuitería base, la memoria RAM y, generalmente, buses de comunicación dedicados que facilitaban las tareas de sincronización.

Una arquitectura más barata y de diseño más sencillo, en cuanto no requiere buses de comunicación entre zócalos, es el de los circuitos integrados que incorporan múltiples núcleos de procesamiento en la misma pastilla (*die*). Uno de los predecesores de esta idea fue el R65C00 de Rockwell, integrando a mediados de los 80 dos procesadores 6502[2] [8] en un mismo *chip*. Ya a principios del actual siglo tanto Intel como AMD incorporaron en sus microprocesadores dos o más núcleos, un diseño que en la actualidad resulta bastante corriente en configuraciones con 4/8 núcleos e incluso más en microprocesadores de alta gama como los Intel Xeon [4].

Fusionar varios núcleos completos en un mismo circuito integrado, compartiendo buses de comunicación con el exterior y una cierta cantidad de memoria común, resulta una solución casi obvia, que no sencilla de implementar, para mejorar el rendimiento ofrecido por un sistema de cómputo. Una vía alternativa, cuyo estudio tuvo lugar a mediados de los 90, fue la posibilidad de procesar dos o más hilos de ejecución en paralelo mediante lo que se conoce como SMT (*Simultaneous Multi-Threading*). El mítico Alpha EV8 de Digital Equipment Corporation (DEC) [9] fue el primer microprocesador en cuyo diseño se incluyó este planteamiento, muy utilizado en los productos de Intel desde hace algunos años bajo la denominación *Hyper-Threading* (HT).

En la actualidad los micros para sistemas de escritorio de Intel cuentan tanto con varios núcleos de procesamiento como con la tecnología HT, de forma que un equipo con cuatro núcleos, por ejemplo, puede ejecutar paralelamente hasta ocho hilos. También los microprocesadores móviles suelen disponer de dos, cuatro y hasta ocho núcleos, al tratarse esta de una configuración más eficiente en cuanto a consumo de energía se refiere que el aumento de la frecuencia de funcionamiento del sistema. Se trata, en consecuencia, de una de las formas de paralelismo más populares a día de hoy.

Con independencia de si el hardware a explotar consta de múltiples procesadores, procesadores con múltiples núcleos, núcleos de tipo HT o una combinación de varias de estas configuraciones, la herramienta fundamental con la que cuentan los programadores para extraer toda la potencia disponible son las librerías de tipo *pthreads* [10]. Los sistemas operativos modernos incorporan estos servicios, cuya finalidad es permitir a una aplicación crear múltiples hilos de ejecución simultánea, comunicarse con ellos, sincronizarlos, etc. En general, y a pesar de que el uso de *threads* resulta más común en lenguajes como C/C++, cualquier lenguaje de programación con acceso a los servicios del sistema puede sacar provecho de dichos servicios. El lenguaje Java, por ejemplo, pone a disposición de los programadores esa API a través de su máquina virtual (JVM).

[2] El 6502 fue un microprocesador muy popular en los 70-80 debido a su bajo precio. Fue la base de múltiples microordenadores, entre ellos los primeros modelos fabricados por Apple, Commodore y Atari.

2.3. Sistemas de computación distribuida

Paralelamente a la evolución de las CPU muchas otras tecnologías estrechamente relacionadas con la informática, en particular las relativas a sistemas de comunicación, fueron surgiendo y popularizándose. Estas hicieron posible conectar múltiples ordenadores formando redes de una manera relativamente barata, especialmente desde que Ethernet [11] pasó a ser un estándar en 1983. Interconectando ordenadores relativamente baratos es posible configurar *clústeres* con una gran potencia de cómputo, siguiendo el paradigma Beowulf [12] desarrollado en 1994 para la NASA. Esto permite, hasta cierto punto, tratar todo el *clúster* como un superordenador, con un gran número de unidades de procesamiento y mayor capacidad de memoria.

El principal obstáculo que representa la explotación de un sistema de cómputo distribuido, un *clúster* de máquinas, estriba en la distribución del trabajo entre las distintas máquinas y su sincronización. El método habitual consiste en habilitar un intercambio de mensajes entre los equipos de la red, ya sea gestionado por la propia aplicación o por un *middleware* que se ejecuta entre esta y el propio sistema operativo. Para ello, a lo largo del tiempo, se desarrollaron soluciones como RPC (*Remote Procedure Call*) [13], CORBA (*Common Object Request Broker Architecture*) [14] o Java RMI (*Remote Method Invocation*) [15]. El esquema fundamental consta de dos capas de software adicionales, el *proxy* y el *stub*, que permiten a la aplicación llamar a los métodos que la componen como si estuviesen ejecutándose de forma local, aunque en realidad lo hagan en otra máquina.

Un esfuerzo conjunto de decenas de empresas a principios de los 90 dio como fruto la especificación de MPI (*Message Passing Interface*) [16], un protocolo de intercambio de mensajes para aplicaciones distribuidas con interfaces para multitud de lenguajes de programación. En los últimos años, con el advenimiento del concepto *big data*, han ganado un considerable terreno soluciones como Apache Hadoop y Apache Spark [17]. Estas facilitan un mayor nivel de abstracción respecto a MPI, ofreciendo al desarrollador mecanismos que, siguiendo el enfoque divide y vencerás, distribuyen los datos y el trabajo entre las máquinas que forman el *clúster*.

3 De los *shaders* a la GPGPU

La evolución en el desarrollo de las CPU y los mecanismos de paralelización descritos en las anteriores secciones ha sido promovida, principalmente, por la necesidad de satisfacer la demanda que organizaciones y empresas hacían para poder ejecutar aplicaciones capaces de procesar un volumen de datos creciente y cada vez más complejo. Un ejemplo de estas son los grandes sistemas de bases de datos que, diariamente, atienden a miles de usuarios simultáneamente. Al mismo tiempo, aunque por una razón muy distinta como era la ejecución de juegos cada vez más sofisticados, el hardware de generación de gráficos experimentaba asimismo un acelerado avance. En la actualidad la industria del videojuego ha superado económicamente a la del cine [18], moviendo miles de millones de euros anualmente. Este volumen de negocio ha generado un gran beneficio para empresas con NVIDIA, revirtiendo en una inversión en investigación prácticamente sin precedentes.

Las GPU cuentan con unidades de procesamiento especializado, no de propósito general como las CPU, si bien han ido adquiriendo paulatinamente algunas de las capacidades de estas últimas. La diferencia fundamental estriba en que una GPU puede integrar miles de núcleos de procesamiento, frente a las pocas decenas de las CPU más potentes a día de hoy. Esto ha provocado que se desarrollen abundantes soluciones de procesamiento masivo de datos basados en GPU, siendo uno de los ejemplos más representativos las redes de aprendizaje profundo o *Deep Learning*. El más reciente producto de esta gama, la NVIDIA DGX-1 [19], es capaz de entrenar un modelo complejo de este tipo en 2 horas, cuando el mismo modelo requería más de 160 horas en un sistema con doble procesador Xeon con 14 núcleos/28 hilos cada uno.

El hardware de vídeo inicialmente no era programable, limitándose a convertir los datos alojados en una zona de memoria del ordenador en una señal interpretable para una pantalla. La elaboración de los gráficos era trabajo de la CPU, mientras el adaptador de vídeo leía a intervalos regulares el contenido de esa zona de memoria y actualizaba la señal de vídeo. Manteniendo un cauce (*pipeline*) gráfico no programable, poco a poco los adaptadores de vídeo fueron incorporan funciones de procesamiento gráfico parametrizables. Gracias a dichas funciones se podían aplicar sombreados, iluminación, etc., a los objetos de una escena gráfica. La llegada de las GPU programables [20] sustituyeron las funciones fijas parametrizables por funciones definidas por el usuario, conocidas habitualmente como *shaders*.

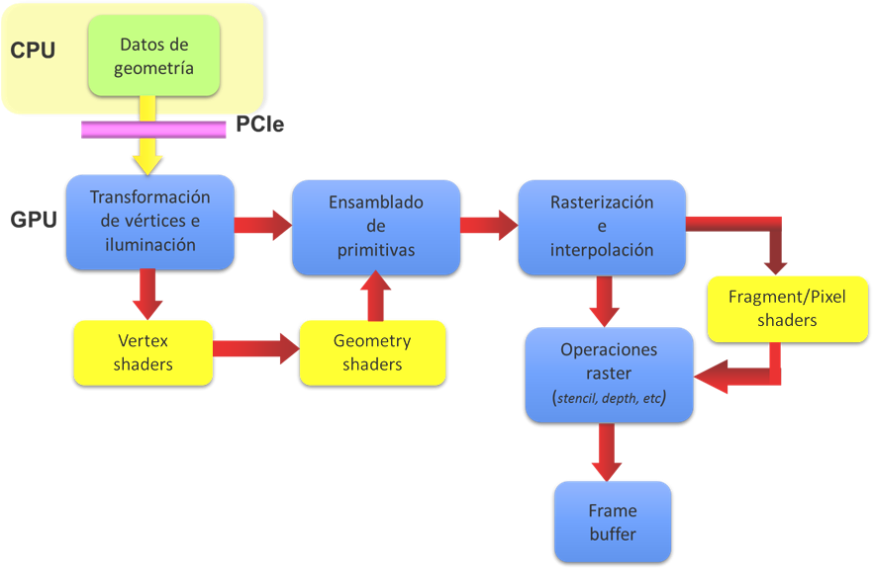

Figura 1. Estructura de bloques del *pipeline* de una GPU actual.

Los *shaders* son pequeños programas especializados en la manipulación de vértices correspondientes a los objetos gráficos, su geometría y sus píxeles, aplicando rotaciones, efectos de iluminación, texturas, etc. Existen múltiples versiones de la especificación de programación de *shaders*, denominadas *Shader Model*, y lenguajes para crearlos desarrollados por empresas como NVIDIA y Microsoft. Entre dichos lenguajes se

encuentran NVIDIA Cg (*C for graphics*), HLSL (*High-Level Shader Language*) y GLSL (*OpenGL Shading Language*). A pesar de que las operaciones realizadas por los *shaders* siempre están orientadas a la producción de gráficos, actuando sobre miles de vértices y píxeles en tiempo real, en realidad no dejan de ser funciones de tipo SIMD, potencialmente aplicables a muchos otros problemas.

El concepto GPGPU, la implementación de soluciones de propósito general mediante programación en GPU ya definida en 2004 en [21], comienza a ser accesible para los desarrolladores a partir de 2007/2008, a raíz de la introducción de tecnologías como CUDA y ATI Stream. Estas fueron las primeras herramientas de desarrollo en permitir la ejecución de código en GPU no relacionado específicamente con la generación de gráficos. CUDA, y posteriormente OpenCL y WebCL, han llevado a las técnicas GPGPU a convertirse en el principal paradigma actual de la computación de altas prestaciones.

4 Desarrollo de soluciones GPGPU

Para aprovechar la potencia de las GPU es necesario contar con herramientas de desarrollo adecuadas, capaces de explotar el alto nivel de paralelismo que ofrecen estos dispositivos. Hasta no hace mucho dichas herramientas eran bastante primitivas, ya que su objetivo era facilitar exclusivamente la programación de *shaders*. Estos son pequeños bloques de código que aplican un cierto procesamiento a los vértices de la geometría de una escena y los fragmentos resultantes de la *rasterización*. Ese código se ejecuta paralelamente en cada núcleo, lo cual permite aplicar un cierto algoritmo masivamente a miles o millones de vértices y píxeles.

Estos bloques de código tienen una longitud generalmente muy limitada y se programan en una suerte de lenguaje ensamblador a medida, por lo que difícilmente pueden aplicarse más que a la función para la que están pensados desde un principio. Como se indicó anteriormente existen diferentes versiones, denominadas *Shader Models*, que han ido evolucionando en paralelo a Microsoft DirectX y OpenGL y que tanto ATI/AMD como NVIDIA han ido implementando en su hardware. Al desarrollar una aplicación gráfica se utiliza una API, como las citadas DirectX u OpenGL, para escribir el código que se ejecutará en la CPU, usando el ensamblador del *shader model* correspondiente para escribir el código a ejecutar en la GPU. Tanto el tipo de operaciones que puede llevar a cabo ese código como la memoria a la que tiene acceso están limitados.

El desarrollo de Cg [22] por parte de NVIDIA, hace unos 15 años, fue un primer avance al facilitar la codificación de funciones a ejecutar en la GPU. En lugar de escribir el código en ensamblador se usa un lenguaje de más alto nivel, similar a C. Una función como la mostrada en el siguiente fragmento se ejecutaría una vez para cada vértice, pero no de manera secuencial sino paralelamente:

```
void processVertex(
  in float4 location : POSITION,
  out float4 locationD : POSITION,
  out float4 colorD : COLOR0,
```

```
    const uniform float4x4 ModelViewMatrix,
    const uniform float4 color)
{
    locationD = mul(location, ModelViewMatrix);
    colorD = color;
}
```

La limitación de Cg es que se trata de un lenguaje dirigido específicamente a la generación de gráficos. A medida que el número de núcleos de proceso en una GPU se fue incrementando, y ganando en rendimiento al poder operar con datos en coma flotante, se hizo cada vez más patente la necesidad de aprovechar esa potencia bruta de cálculo para propósitos alternativos, aparte de la evidente aplicación en videojuegos de última generación. Solamente se precisaban herramientas de trabajo de corte más general, con un espectro de aplicación más amplio. La primera respuesta a esta necesidad fue CUDA [23], desarrollado por NVIDIA, seguida de OpenCL [24] y WebCL [25]. En esta sección se introduce cada una de estas tecnologías.

Sobre estas tecnologías de bajo nivel se han desarrollado soluciones de más alto nivel y específicas para ciertos lenguajes. Por ejemplo se han desarrollado paquetes o librerías para programación GPU desde Matlab, Java, R, Python, etc.

4.1. CUDA

La mayoría de los lenguajes de programación no cuentan con estructuras nativas que faciliten la paralelización de procesos, entendiendo como tales partes de un algoritmo que pueden ser ejecutados de manera simultánea y no como lo que se entiende por procesos en el contexto de un sistema operativo. Es cierto que existen API y bibliotecas de funciones, como la anteriormente citada *pthreads*, que facilitan hasta cierto punto la programación paralela, pero prácticamente ninguna de ellas está pensada para ejecutar el código explotando una GPU. En la mayoría de los casos lo único que hacen es iniciar varios hilos de ejecución dejando en manos del sistema operativo el reparto de tiempo de proceso entre las unidades con que cuente la CPU. Para trasladar la aplicación a otro tipo de procesador, así como para ampliar o reducir el número de hilos en ejecución, es corriente tener que alterar, o incluso rescribir por completo, el código fuente.

La solución que ofrece CUDA (*Compute Unified Device Architecture*) es mucho más flexible y potente y, además, se basa en estándares existentes. Los programas se escriben en lenguaje C, no en el ensamblador de un cierto procesador o en un lenguaje especializado como es el caso de Cg. Esto facilita el acceso a un grupo mucho mayor de programadores. Al desarrollar una aplicación CUDA el programador escribe su código como si fuese a ejecutarse en un único hilo, sin preocuparse de crear y lanzar *threads*, controlar la sincronización, etc. Ese código será ejecutado posteriormente en un número arbitrario de hilos, asignado cada uno de ellos a un núcleo de procesamiento, de manera totalmente transparente para el programador. Este no tendrá que modificar el código fuente, ni siquiera recompilarlo, dependiendo de la arquitectura del hardware donde vaya a ejecutarse. Incluso existe la posibilidad de recompilar el código fuente, dirigido originalmente a ejecutarse sobre una GPU, para que funcione so-

bre una CPU clásica, asociando los hilos CUDA a hilos de CPU en lugar de a núcleos de ejecución de GPU. Obviamente el rendimiento será muy inferior ya que el paralelismo al nivel de CPU no es, actualmente, tan masivo como en una GPU.

Los objetivos planteados en el desarrollo de CUDA han dado como fruto un conjunto de tres componentes, disponibles gratuitamente para Windows, GNU/Linux y OS X. El controlador CUDA es el componente básico, ya que es el encargado de facilitar la ejecución de los programas y la comunicación entre CPU y GPU. Este controlador se aplica a prácticamente toda la gama hardware de NVIDIA: GeForce 8XX, 9XX y GTX 2XX y posteriores, así como a la línea de adaptadores Quadro y los procesadores Tesla. En cualquier caso, se requiere una cantidad mínima de 256 MB de memoria gráfica para poder funcionar, por lo que en adaptadores con menos memoria no es posible usar CUDA. Instalado el controlador, el siguiente componente fundamental para el desarrollo de aplicaciones es el *toolkit* CUDA, compuesto a su vez de un compilador de C llamado `nvcc`, un depurador específico para GPU, un perfilador de código y un conjunto de bibliotecas con funciones de utilidad ya predefinidas, entre ellas la implementación de la Transformada rápida de Fourier (FFT) y unas subrutinas básicas de álgebra lineal (BLAS). El tercer componente de interés es el *CUDA Developer SDK*, un paquete formado básicamente por código de ejemplo y documentación. Se ofrece más de medio centenar de proyectos en los que se muestra cómo integrar CUDA con DirectX y OpenGL, cómo paralelizar la ejecución de un algoritmo y cómo utilizar las bibliotecas FFT y BLAS para realizar diversos trabajos: generación de un histograma, aplicación de convolución a una señal, operaciones con matrices, etc. Conjuntamente estos tres componentes ponen al alcance del programador todo lo que necesita para aprender a programar una GPU con CUDA y comenzar a desarrollar sus propias soluciones, apoyándose en código probado como el de los ejemplos facilitados o el de las bibliotecas FFT y BLAS.

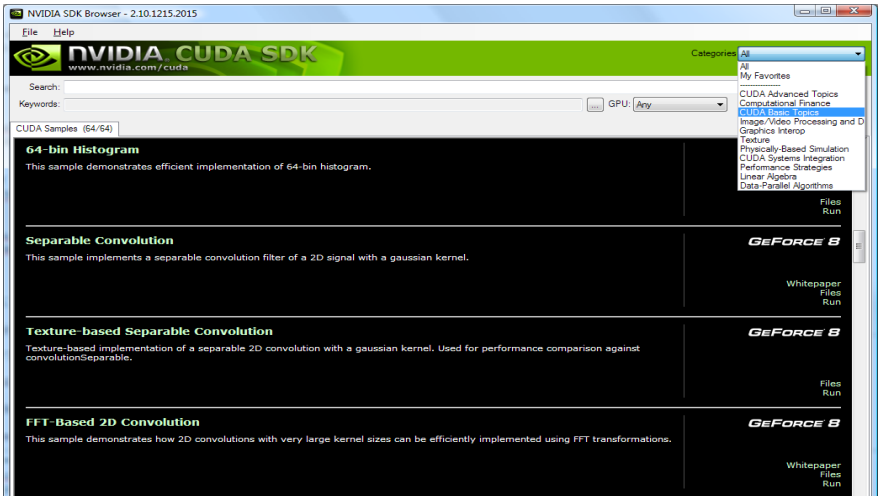

Figura 2. El NVIDIA CUDA SDK ofrece múltiples herramientas al desarollador.

El código de un programa escrito para CUDA siempre estará compuesto de dos partes: una cuya ejecución quedará en manos de la CPU y otra que se ejecutará en la

GPU. Al código de la primera parte se le denomina código para el *host* y al de la segunda código para el dispositivo. Al ser procesado por el compilador `nvcc`, el programa generará por una parte código objeto para la GPU y, por otra, código fuente u objeto para la CPU. El código objeto específico para la GPU se denomina *cubin*. El código fuente para la CPU será procesado por un compilador de C/C++ corriente, enlazando el código *cubin* como si de un recurso se tratase. La finalidad del código *host* es inicializar la aplicación, transfiriendo el código *cubin* a la GPU, reservando la memoria necesaria en el dispositivo y llevando a la GPU los datos de partida con los que se va a trabajar. Esta parte del código puede escribirse en C o en C++, lo cual permite aprovechar el paradigma de orientación a objetos si se quiere. El código a ejecutar en el dispositivo debe seguir estrictamente la sintaxis de C, contemplándose algunas extensiones de C++. Normalmente se estructurará en funciones llamadas *kernels*, cuyas sentencias se ejecutarán en paralelo según la configuración hardware del dispositivo final en el que se ponga en funcionamiento la aplicación.

Lo que hace el entorno de ejecución de CUDA, a grandes rasgos, es aprovechar el conocido como paralelismo de datos o SIMD, consistente en dividir la información de entrada, por ejemplo una gran matriz de valores, en tantos bloques como núcleos de procesamiento existan en la GPU. Cada núcleo ejecuta el mismo código, pero recibe unos parámetros distintivos que le permiten saber la parte de los datos sobre los que ha de trabajar. El sencillo ejemplo mostrado a continuación corresponde a una función *kernel* muy simple, cuyo objetivo es hallar el producto escalar de una matriz por una constante. La función solamente opera sobre un elemento de la matriz, el que le indica la variable `threadIdx.x` que identifica el hilo en que está ejecutándose el código. Esta función se ejecutaría paralelamente en todos los núcleos de la GPU, por lo que en un ciclo se obtendría el producto de una gran porción de la matriz o incluso de esta completa, dependiendo de su tamaño y el número de núcleos disponibles.

```
__global__ void ProdEscalar(
    float* M1,
    float* M2,
    float Constante)
{
   // Se obtiene el número de thread
   int i = threadIdx.x;
   // y se procesa el dato que corresponde
   M2[i] = M1[i] * Constante;
}
```

En una CPU moderna, como puede ser un Intel Core i7, el desarrollador puede dividir los datos de entrada en cuatro o seis partes, dependiendo del número de núcleos con que cuente el microprocesador, pero sin ninguna garantía de que se procesarán en paralelo salvo que se programe explícitamente el reparto trabajando a bajo nivel. Es decir, el propio programador ha de crear los hilos de ejecución independientes y asignar cada una de las particiones de datos a un hilo. En una GPU y usando CUDA, por el contrario, esos datos se dividirán en bloques mucho más pequeños, al existir 240,

512, 1024 o más núcleos de procesamiento, garantizándose la ejecución en paralelo si necesidad de recurrir a la programación en ensamblador.

4.2. OpenCL

¿Cómo podría una misma aplicación, con el objetivo de explotar toda la potencia de un ordenador actual, aprovechar tanto el paralelismo de la CPU como de la GPU? Utilizando CUDA esto implica crear hilos en la CPU para las tareas MIMD y *kernels* en la GPU para las tareas SIMD, con herramientas distintas y en ocasiones lenguajes de programación distintos, integrando los diversos componentes de la mejor manera posible. Una alternativa que cuenta con el favor de una gran parte de la industria es OpenCL, un estándar que hace posible la ejecución de código escrito en C/C++ distribuyendo las tareas entre CPU y GPU, sin que importe el fabricante de la GPU. Esa capacidad, no obstante, es actualmente teórica en un escenario en el que cada empresa intenta favorecer su oferta sobre la de la competencia. Si se cuenta con un hardware gráfico de NVIDIA, los controladores OpenCL de esta empresa solamente reconocerán como dispositivo la propia GPU, tal y como se aprecia en la ventana de la izquierda de la Figura 3. Los controladores de AMD/Intel, por el contrario, sí reconocen la CPU como dispositivo (ventana de la derecha), pero obviamente no pueden usar la GPU de NVIDIA sino las suyas propias.

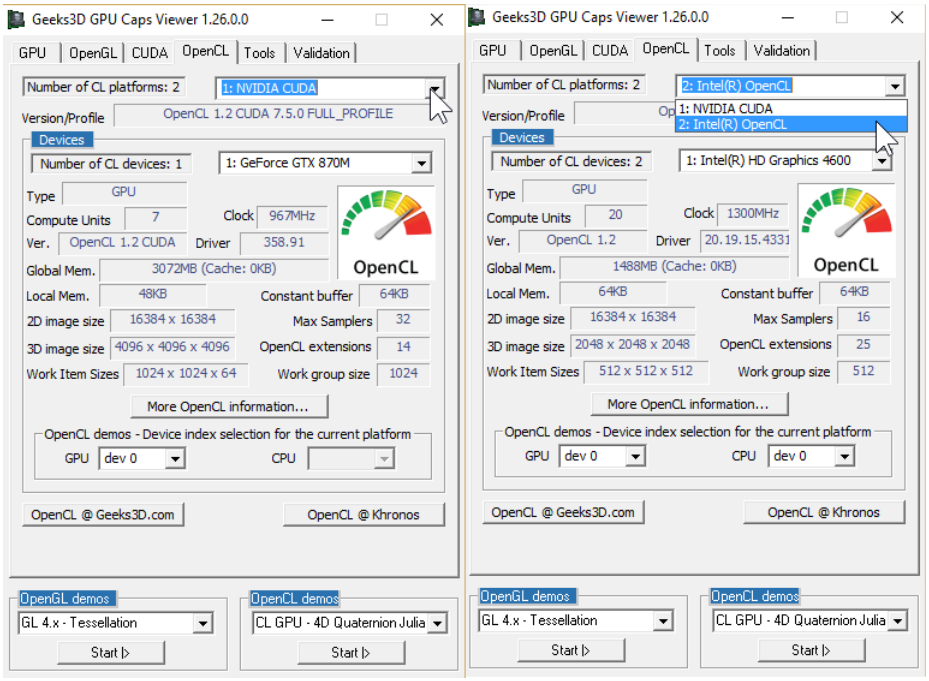

Figura 3. Configuración de controladores OpenCL de NVIDIA/Intel en un mismo sistema.

OpenCL ofrece al desarrollador una interfaz de más alto nivel para la programación de tareas que han de ejecutarse en paralelo, abstrayéndole de las diferencias entre GPU y CPU y los distintos hardware gráficos. Se trata, por tanto, de una herramienta que facilita el aprovechamiento del paralelismo sobre plataformas heterogéneas, incluyendo CPU, GPU y potencialmente otro tipo de procesadores especializados como las FPGA.

Uno de los obstáculos que habitualmente encontramos al iniciarnos en el uso de una nueva tecnología, en este caso concreto la programación en GPGPU con CUDA u OpenCL, estriba en que habitualmente se requiere el conocimiento de un lenguaje relativamente complejo, como puede ser C/C++, y el conjunto de herramientas asociado para compilación, depuración, etc. En los últimos años este proceso se ha visto simplificado gracias a lenguajes como Python que, a través de los paquetes `pycuda` y `pyopencl`, hacen posible el acceso interactivo a las funciones ofrecidas por los respectivos paquetes de desarrollo. La información relativa a las plataformas disponibles, anteriormente mostrada en la Figura 1 y obtenida con una herramienta específica, puede obtenerse desde Python previa instalación del paquete `pyopencl` con las siguientes sentencias:

```
import pyopencl as cl

cl.VERSION
cl.get_platforms()
cl.get_platforms()[0].get_devices()
```

4.3. WebCL

Al tratar el tema del paralelismo siempre se asume que el objetivo es emplearlo en programas que serán instalados y ejecutados en un ordenador de forma nativa, ya sea a través de compiladores específicos para un hardware concreto o el uso de máquinas virtuales que se ocupan de los detalles de más bajo nivel. En cualquier caso, son aplicaciones dirigidas a funcionar bajo una cierta configuración: microprocesador, GPU, sistema operativo, etc. De un tiempo a esta parte, sin embargo, la web se ha convertido en una plataforma más para la ejecución de aplicaciones superando esas especificidades, no precisando más que un navegador que se ajuste a los estándares: HTML5, CSS3 y Javascript.

El código Javascript de una aplicación web puede ejecutar código en múltiples hilos en la CPU gracias a los *Web Workers*, pero hasta hace poco no existía un método que permitiese aprovechar la gran potencia con la que cuentan las GPU actuales con independencia del hardware, sistema operativo o navegador que el usuario emplee para acceder a la aplicación web. Por suerte es una situación que ha cambiado gracias a WebCL.

WebCL es a las aplicaciones web lo que OpenCL a las aplicaciones nativas: una capa de abstracción que permite ejecutar código en paralelo tanto en CPU como en GPU, sin importar el fabricante del hardware ni el sistema operativo empleado. Es un estándar regido por el Khronos Group y que siguen múltiples fabricantes de hardware, entre ellos AMD, NVIDIA e Intel. En realidad, WebCL es en esencia un enlace o

binding para poder acceder a OpenCL desde Javascript. Al tratarse de un estándar en desarrollo ningún navegador lo incluye actualmente. Para poder probarlo es necesario instalar un complemento en el navegador. Completado ese paso, es posible crear desde Javascript un objeto `WebCLComputeContext` (en la implementación de Samsung) y usarlo para obtener información sobre el hardware disponible, preparar el código a ejecutar paralelamente y enviarlo a la CPU/GPU.

El grupo de trabajo encargado de este estándar dentro del Khronos Group fue creado en 2011 y, en principio, su objetivo es facilitar una guía de implementación para fabricantes conservando la esencia de OpenCL y poniendo especial énfasis en el tema de la seguridad, ya que el código se ejecutaría en el ordenador de los usuarios al acceder a una aplicación desde su navegador, sin necesidad de instalar ni ejecutar explícitamente un programa externo.

5 Conclusiones

La potencia de los ordenadores se ha incrementado varios órdenes de magnitud en las últimas décadas, haciendo realidad el desempeño de tareas que parecían prácticamente imposibles hace no muchos años. El aprovechamiento de esa potencia, sin embargo, ha demandado el desarrollo de nuevas arquitecturas hardware y tecnologías software a lo largo de este tiempo. Si en los primeros años de la informática personal, en los 70-80, el incremento en la frecuencia de reloj de un microprocesador se traducía automáticamente en un mayor rendimiento del software, desde principios de este siglo esa mejora no se obtiene de manera tan directa, siendo preciso recurrir a técnicas de paralelización basadas en los principios SIMD/MIMD.

En este trabajo se ha resumido la evolución de las técnicas de paralelización desde casi los albores de los primeros ordenadores, y las técnicas de tiempo compartido, hasta las modernas tecnologías GPGPU que hacen posible explotar la enorme potencia del hardware gráfico actual. El desafío futuro probablemente estribará en cómo facilitar el aprovechamiento del cada vez mayor número de procesadores alojados en un ordenador, CPU, GPU y otros integrados especializados, ofreciendo una infraestructura de desarrollo homogénea sobre un hardware cada más potente pero también más heterogéneo.

Referencias

1. Ganesan, R., Govindarajan, K., & Wu, M.: Comparing SIMD and MIMD Programming Modes. J. Parallel Distrib. Comput., 35, 91-96 (1996).
2. Ritchie, D., & Thompson, K.: The UNIX Time-Sharing System (Reprint). Commun. ACM, 26, 84-89 (1983).

3a. GE-635 System manual. General Electric. 1964.

3b. Emerson W. Pugh, Lyle R. Johnson, John H. Palmer, IBM's 360 and Early 370 Systems, 360-363, MIT Press 2003.

4. Intel® 64 and IA-32 Architectures Software Developer Manuals. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`

5. IBM System/360 Operating System: MVT Guide, OS Release 21.7. Sixth Edition. IBM. August 1974. GC28-6720-5.
6. Neumann, J.V.: First draft of a report on the EDVAC (1945). IEEE Annals of the History of Computing, 15, 27-75 (1993).
7. Adve, S.V. et al.: Parallel Computing Research at Illinois the Upcrc Agenda Parallel@illinois: Pioneering and Promoting Parallel Computing (2008).
8. James, G., Silverman, B., & Silverman, B.: Visualizing a classic CPU in action: the 6502. SIGGRAPH (2010).
9. Supnik, R.M.: Digital's Alpha Project. Commun. ACM, 36, 30-32 (1993).
10. Mueller, F.: A Library Implementation of POSIX Threads under UNIX. USENIX (1993).
11. Metcalfe, R.: Ethernet: Distributed Packet Switching for Local Computer Networks. BERKELEY (1976).
12. Becker, D.J., Dorband, J.E., Packer, C.V., Ranawake, U.A., Sterling, T.L., & Savarese, D.: BEOWULF: A Parallel Workstation for Scientific Computation. ICPP (1995).
13. Birrell, A., & Nelson, B.J.: Implementing Remote Procedure Calls. ACM Trans. Comput. Syst., 2, 39-59 (1984).
14. Beeharry, A., Bouguettaya, A., & Delis, A.: Managing Persistent Objects in a Distributed Environment. ADC (1987).
15. Domajnko, T., Hericko, M., Juric, M.B., Krisper, M., Rozman, I., & Zivkovic, A.: Java and Distributed Object Models: An Analysis. SIGPLAN Notices, 33, 57-65 (1998).
16. Bruck, J., Dolev, D., Ho, C., Rosu, M., & Strong, H.R.: Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations. SPAA (1995).
17. Ingram, J.E., Monteith, J.Y., & McGregor, J.D.: Hadoop and its Evolving Ecosystem. ICSOB (2013).
18. Chatfield, T. Videogames now outperform Hollywood movies. The Guardian, September, 27 (2009).
19. http://www.NVIDIA.com/object/deep-learning-system.html.
20. Kilgard, M.J., Lindholm, E., & Moreton, H.: A User-programmable Vertex Engine. NVIDIA Corporation (2001).
21. Buck, I., Govindaraju, N.K., Harris, M.J., Krüger, J., Luebke, D., Lefohn, A.E., Purcell, T.J., & Woolley, C.: GPGPU: general purpose computation on graphics hardware. SIGGRAPH (2004).
22. Akeley, K., Glanville, R.S., Kilgard, M.J., & Mark, W.R.: Cg: a system for programming graphics hardware in a C-like language. ACM Trans. Graph., 22, 896-907 (2003).
23. Buck, I.: GPU computing with NVIDIA CUDA. SIGGRAPH (2007).
24. Gohara, D., Stone, J.E., & Shi, G.: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Computing in Science and Engineering, 12, 66-73 (2010).
25. Foley-Bourgon, V., Hendren, L.J., Khan, F., Kathrotia, S., & Lavoie, E.: Using JavaScript and WebCL for numerical computations: a comparative study of native and web technologies. DLS (2014).