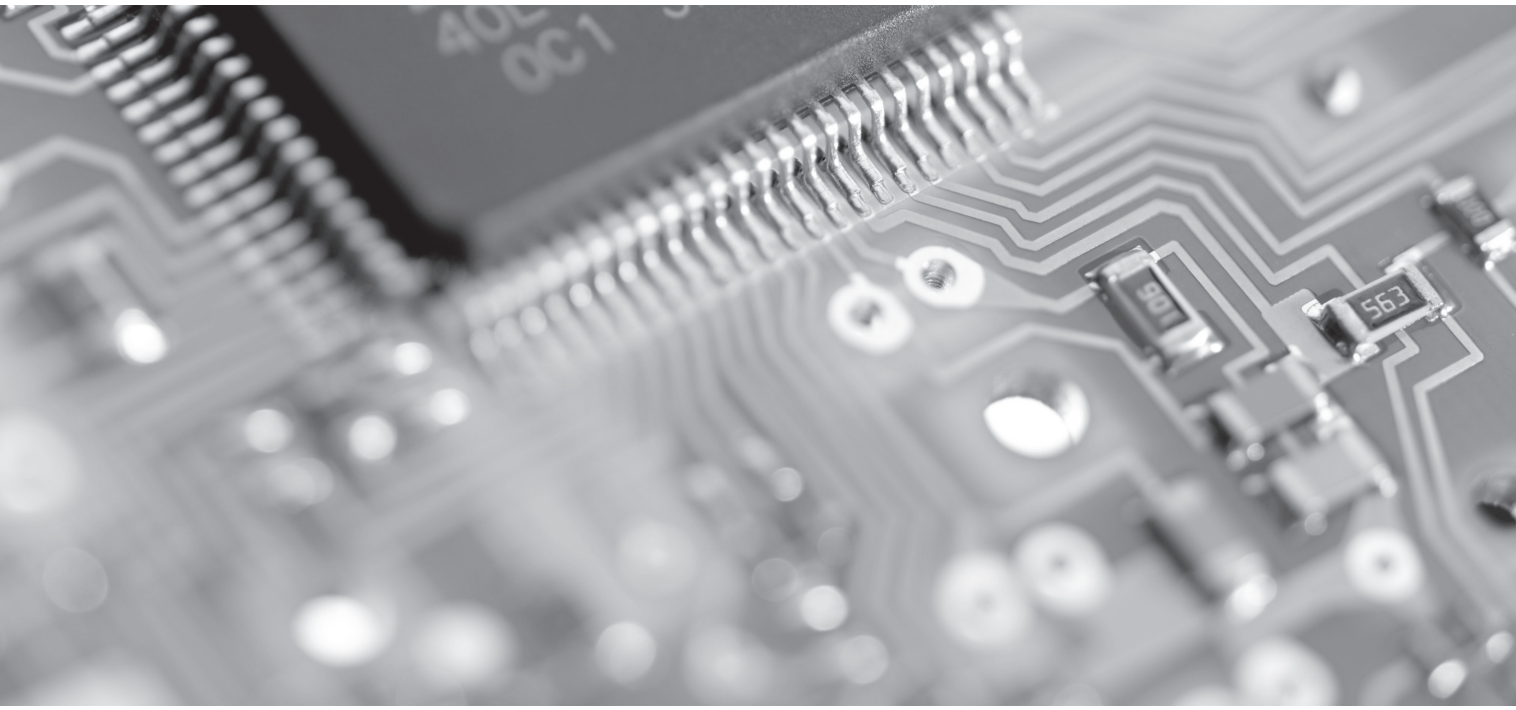
Unidades de procesamiento gráfico, conceptos básicos, aplicaciones y tendencias

Graphic processing units, basic concepts, applications and trends

Luis Felipe Echeverri Escobar[1]
Juan Camilo Acevedo Bedoya[2]
Luis Reinel Castrillón Osorio[3]



1 Especialista en Gestión de Software, Grupo de Investigación en Computación Móvil y Ubicua GIMU, Docente tiempo completo, Facultad de Ingeniería, Universidad Católica de Oriente, Rionegro, Antioquia, Colombia, luis.echeverri6408@uco.net.co.

2 Especialista en Seguridad Informática, Grupo de Investigación en Computación Móvil y Ubicua GIMU, Docente tiempo completo, Facultad de Ingeniería, Universidad Católica de Oriente, Rionegro, Antioquia, Colombia, jacevedo@uco.edu.co.

3 Magister en Ingeniería, Grupo de Investigación en Computación Móvil y Ubicua GIMU, Docente tiempo completo, Facultad de Ingeniería, Universidad Católica de Oriente, Rionegro, Antioquia, Colombia, lcastrillon@uco.edu.co.

Resumen

Este artículo presenta una descripción acerca de los conceptos básicos de una unidad de procesamiento gráfico GPU, su arquitectura, principales características y una comparación de su desempeño versus una unidad central de procesamiento CPU. Esta información permite determinar bajo qué circunstancias es necesario utilizar una GPU para dar solución a un problema de desempeño y cuáles son las condiciones de suficiencia para usar un clúster de estas unidades de procesamiento gráfico. Se habla de igual manera sobre los diferentes tipos de lenguajes soportados para la creación de algoritmos y los pasos necesarios para implementarlos utilizando CUDA. De esta manera se tiene un primer acercamiento y una mejor comprensión acerca del manejo de las GPU, la cual es una de las tecnologías de más alto crecimiento en el procesamiento paralelo en los años recientes.

Palabras clave

Programación en paralelo, Arquitectura GPU, CPU vs GPU, CUDA.

Abstract

This paper provides a description of basic concepts of GPU, its architecture, the main characteristics and a comparison between the GPU and CPU performance. The results provided by this paper will help you to decide if a GPU is what you need to solve your performance problems. Also, if GPU is enough or a GPU cluster is required. Moreover, this paper shows the programming languages used to describe algorithms in GPUs and how to create a basic algorithm in CUDA. The aim of this work is to give the reader a better understanding of GPUs, which is one of the fastest growing technologies in parallel processing.

Key words:

Parallel programing, GPU computing, GPU architecture, CPU vs GPU, CUDA.

Introducción

Existen varias técnicas para mejorar el rendimiento de algoritmos que tardan mucho tiempo en ejecutarse en los sistemas de cómputo, la más simple de ellas es aumentar la frecuencia del procesador. En la actualidad la velocidad de los procesadores no está teniendo el acelerado crecimiento que tenía en otros años, esto debido en gran medida a la potencia dinámica (ver ecuación 1, donde es la frecuencia del reloj, la capacitancia combinada y el voltaje de polarización) y la estática producida por las corrientes de fuga (Asanovic et al., 2006) las cuales generan temperaturas que son complejas de disipar en una área apropiada. Otra técnica es realizar algoritmos que exploten eficazmente la localidad espacial y temporal de los datos en memoria y utilizar sistemas con cachés de varias capas que reduzcan el acceso a memorias lentas como el disco duro o memoria RAM (Guim & Rodero, 2012).

$$P_D = fCV_{DD}{}^2 \qquad (1)$$

Cuando las técnicas anteriores no son suficientes para alcanzar el desempeño deseado, se requiere cambiar la programación de serial a paralelo, esto consiste no solo en tener un hardware que posibilite hacerlo, sino una planeación para subdividir el problema en partes que puedan ser ejecutadas lo más independientemente posible una de la otra, lo que dependerá del problema que está resolviendo el algoritmo y que no siempre garantiza mejores resultados que los que se podrían obtener con el procesamiento serial (Guim & Rodero, 2012). Para llevar a cabo esta tarea existen una serie de herramientas y hardware para lograrlo, entre las más citadas en la literatura se encuentran los sistemas multinúcleos, supercomputadoras, clústeres, grids, arreglos de compuertas programables FPGA *(Field Programmable Gate Array)* y unidades de procesamiento gráfico GPU *(Graphics Processor Unit)* (Torun, Yilmaz, Akansu, 2016).

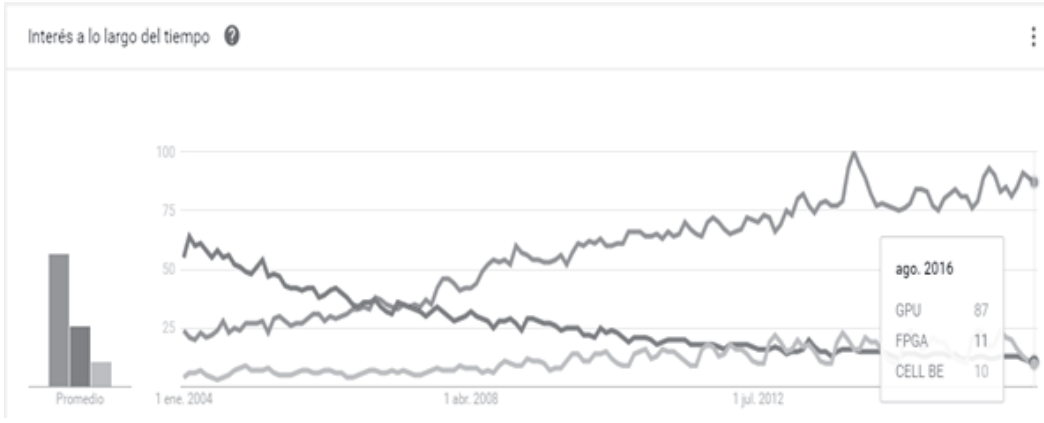
Con la finalidad de lograr el objetivo de ayudar al lector a tomar la mejor decisión acerca de cuándo utilizar una GPU o un clúster de estas, este artículo se divide en cuatro secciones. En la primera sección se comentan los conceptos básicos de la GPU, qué es, cómo ha sido la evolución de las GPU de la marca dominante en el mercado y además se presenta un análisis comparativo entre la CPU y la GPU. En la sección 2 se describe la arquitectura de la más reciente GPU de NVIDIA. En la sección 3 se indican los lenguajes utilizados para describir el algoritmo dentro de este hardware y se presenta un ejemplo de cómo ejecutar un algoritmo simple en la GPU que aproveche el paralelismo de esta. En la sección 4 se presentan algunas aplicaciones de la GPU y las tendencias actuales.

Conceptos básicos

Una GPU es un coprocesador que ayuda a la unidad central de procesamiento CPU *(Central Processing Unit)* en tareas gráficas u operaciones de punto flotante, con la finalidad de aligerar la carga del procesador central. Al uso de las GPU para propósito general se les conoce como unidades de procesamiento gráfico de propósito general GPGPU *(General-Purpose Computing on Graphics Processing Units),* y es una de las tendencias con mayor crecimiento en el campo de la computación

paralela por encima de las FPGA o el *Cell Broadband Engine* de Sony (que es el corazón de la Play Station 3). En la figura 1 se muestra el resultado de las tendencias de búsqueda en Google de estas tres tecnologías. En gran medida, esta tendencia de crecimiento se da gracias a la relación existente entre las prestaciones que ofrece y su costo (Guim & Rodero, 2012), además los lenguajes actuales para la programación de las GPU permiten un tiempo de desarrollo menor comparado con lenguajes como Verilog o VHDL para el diseño RTL *(Register Transfer Logic)* en FPGA para propósitos similares (Asanovic et al., 2006).

Figura 1. Tendencia de búsqueda en Google sobre GPU, FPGA, CELL BE.



Fuente Google Trends (www.google.com/trends).

En la actualidad existen tres principales proveedores de unidades de procesamiento gráfico. Intel domina el mercado de las GPU integradas y de bajo desempeño, mientras que para el mercado de alto rendimiento se encuentran NVIDIA y AMD (Brodtkorb, Hagen, Saetra, 2013). NVIDIA es la clara dominadora del mercado tanto en el campo académico como en el industrial, por tal razón este artículo se centrará en ella.

Para ejecutar un algoritmo o parte de él en una GPU es conveniente analizar las características y hacer una comparación entre el rendimiento de la CPU y GPU. La tabla 1 muestra los parámetros más relevantes entre la GPU de NVIDIA GP100 y uno de los actuales procesadores de Intel, el Xeon E5-2697v4 (Intel Corporation, 2016); (Chiapppetta, 2016); (NVIDIA Corporation Tech, 2016a). Como se observa es notable la diferencia entre los núcleos, su precio, su potencia, la frecuencia y el rendimiento en ejecución de operaciones de punto flotante, por lo que el algoritmo debe permitir explotar estas características y mejorar así el rendimiento del programa.

Tabla 1. Comparación entre la GPU GP100 vs Xeon E5-2697v4.

	GPU P100 NVIDIA	Procesador Intel Xeon E5-2697 v4
Número de transistores	15.300,000,000	7,200,000,000
Número de núcleos	3584	18
Tecnología del transistor	16nm FinFET	14nm
Potencia	300W	145W
Precio	129,000 USD	2,702 USD
CACHÉ	4MB	45MB
Frecuencia	1.328GHz	2.3GHz
Precisión doble	4.7 TFLOPS	437 GFlops
Arquitectura	Pascal	Intel 64
Lanzamiento	2016	2016

Los datos revelados en la tabla 1 pueden generar falsas expectativas, ya que se podría pensar que, si se paraleliza suficientemente un algoritmo, se lograrían incrementos sustanciales en la ejecución, proporcionales a la cantidad de núcleos disponibles en la GPU en comparación con la CPU. Sin embargo, se ha demostrado que los incrementos en la velocidad pueden ser de seis a doce veces mayores que aquellos algoritmos adecuadamente optimizados en procesadores multinúcleo. Por lo tanto, algunos resultados de investigación que obtienen velocidades 100 veces mayores deben ser analizados con más detalle, puesto que puede ser una medición engañosa debida a que los investigadores, con la finalidad de mostrar resultados y valorar su esfuerzo, dedican un tiempo considerable a perfeccionar los algoritmos en la GPU y los algoritmos de la CPU los ejecutan sin mayores procesos de optimización (Jaros & Pospichal, 2012); (Sangjin, 2016). Por ende, siempre se debe realizar una adecuada optimización del uso de los recursos tanto en la GPU y CPU que permitan una correcta comparación.

Arquitectura Pascal de NVIDIA

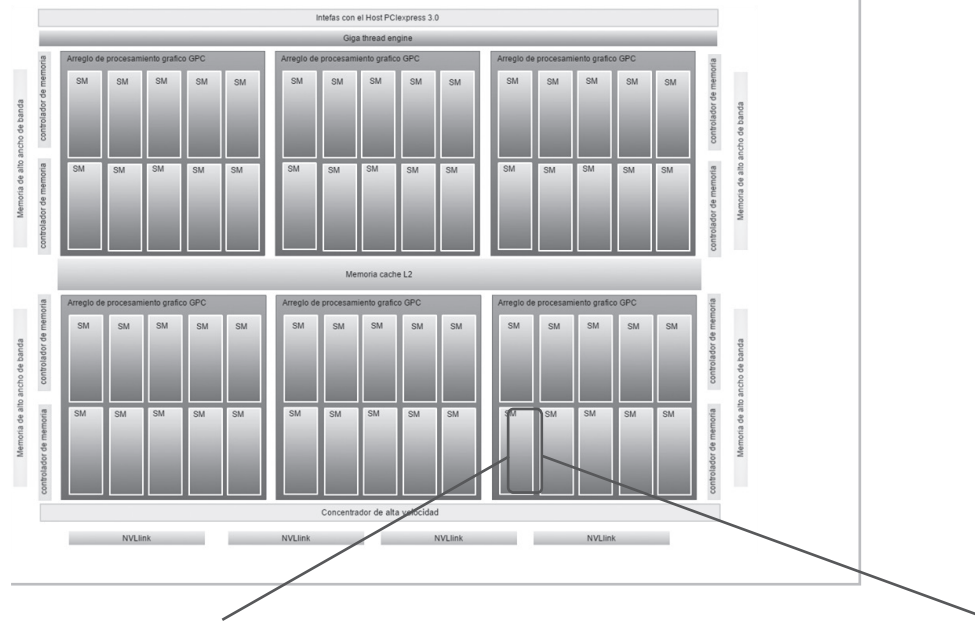
Los procesadores multinúcleo de los computadores están compuestos de un número de complejos núcleos con grandes cachés. Los núcleos son optimizados para ejecuciones de un solo hilo y pueden manejar dos hilos de hardware por cada núcleo usando una técnica de *hyper-threading,* esto significa que gran parte del área del silicio es dedicada al para-

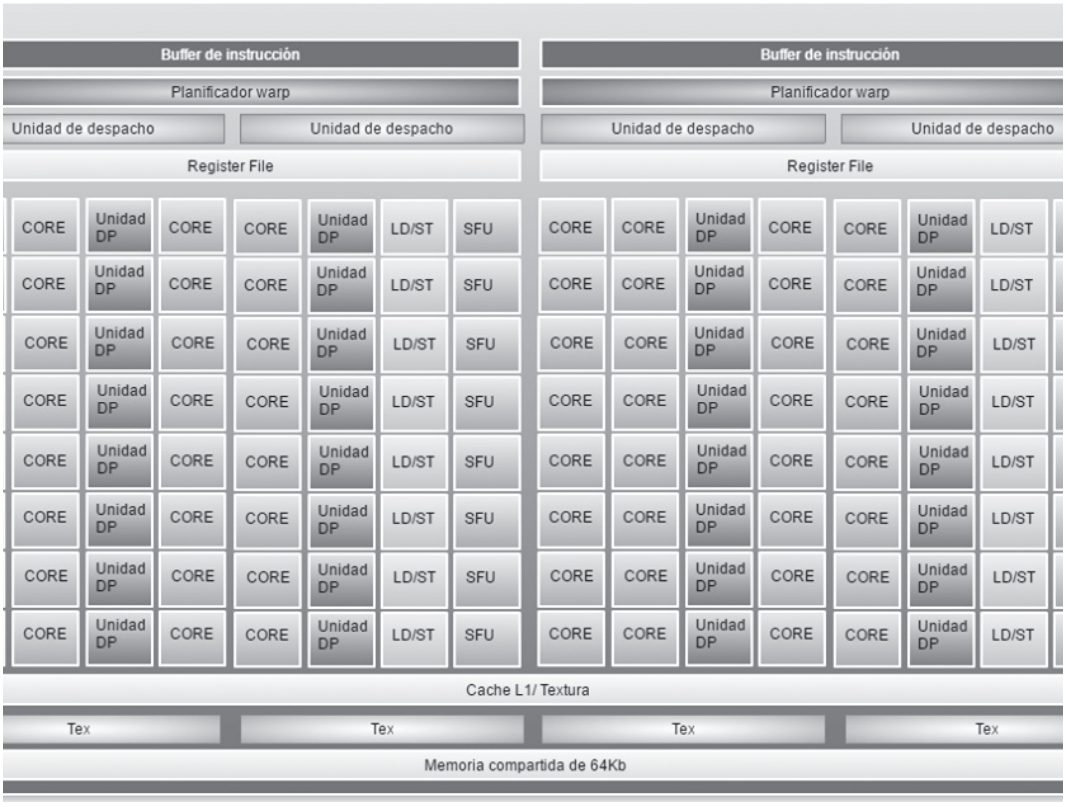
lelismo de la instrucción como lo es el *pipeline,* la predicción del salto y la especulación de la ejecución, dejando así una pequeña fracción de esta área para las unidades funcionales de procesamiento de enteros y de punto flotante FPU *(Floating-point Unit)*. En contraste, una GPU está compuesta de miles de núcleos más simples que manejan miles de hilos de hardware concurrentes, los cuales son diseñados para maximizar el desempeño de las operaciones de punto flotante, por lo que la mayor parte de los transistores dentro de cada núcleo se dedican más a las operaciones de la FPU, en lugar de las complejas instrucciones de control, almacenamiento y largas cachés de la CPU (Brodtkorb, et al., 2013); (Acosta, Segura, Ospina, 2012).

La actual arquitectura Pascal de la GP100 mostrada en la figura 2 está conformada por 3584 núcleos denominados núcleos CUDA, cada uno de estos puede ejecutar una instrucción entera en la ALU o una operación de punto flotante en la FPU en tan solo un ciclo de reloj. La GPU está compuesta de un arreglo de seis grupos de procesamiento gráfico GPC *(Graphics Processing Cluster),* cada uno con diez multiprocesadores de flujo SM *(streaming multiprocessor),* para un total de 60 SM. Mostrados con más detalle en la parte inferior derecha de la figura 2, cada uno de ellos cuenta con 32 núcleos CUDA, 16 unidades de punto flotante con soporte para operaciones de precisión doble, simple y media, ocho unidades de carga y almacenamiento, cuatro grupos de procesamiento de textura y ocho unidades para ejecutar instrucciones especiales como seno, coseno, raíz cuadrada e interpolación (NVIDIA Corporation Tech, 2016a). Todos los procesadores de un SM comparten la unidad de búsqueda y lanzamiento de instrucciones, de tal forma que se ejecuta la misma instrucción al mismo tiempo en todos los procesadores (Acosta et al., 2012).

Figura 2.

GPU GP100 con 6 arreglos de procesamiento gráfico GPC y 60 SM, haciendo ampliación a una de ellas.

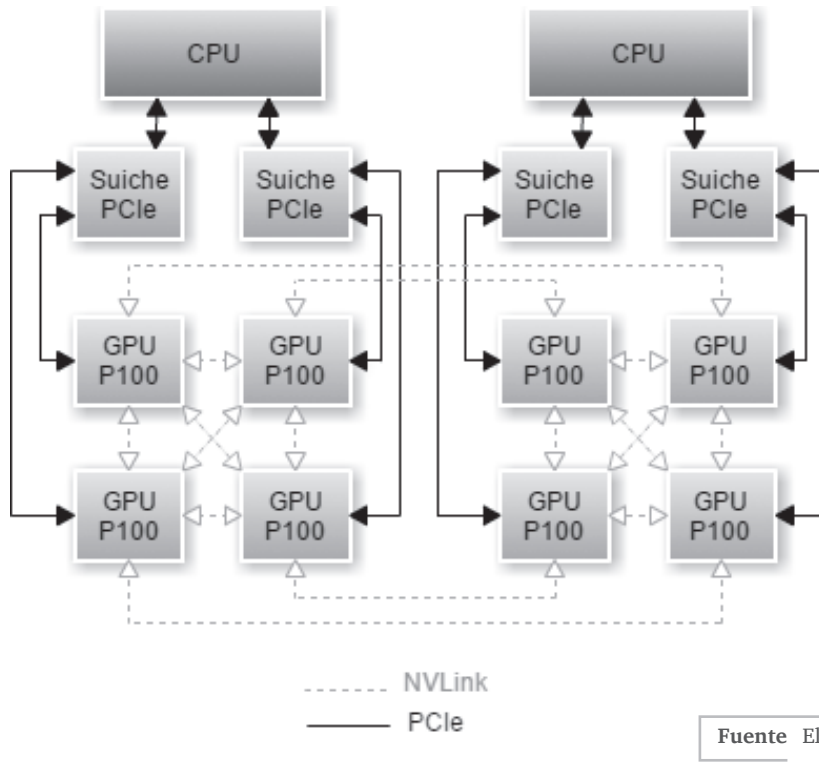




Fuente Elaboración propia.

La GP100 provee un total de 4 MB de memoria caché L2, mejorando el desempeño y capacidad con relación a sus antecesoras, lo que permite disminuir el acceso a memoria DDR *(Dynamic Direct RAM)* y mejorar la eficiencia, además provee una nueva tecnología de interconexión denominada *NVLink,* la cual permite la unión de varias GPU para aumentar la velocidad de transferencia entre ellas o arreglos entre otras CPU y GPU. En la figura 3 se puede observar un diagrama de interconexión que puede aumentar la velocidad de transferencia entre GPU o, si el hardware lo permite, entre GPU y CPU, en una proporción entre cinco a doce veces mayor con respecto al bus PCIe (NVIDIA Corporation Tech, 2016a); (NVIDIA Corporation Tech, 2016b).

Figura 3. Interconexión en malla de 8 GPU.



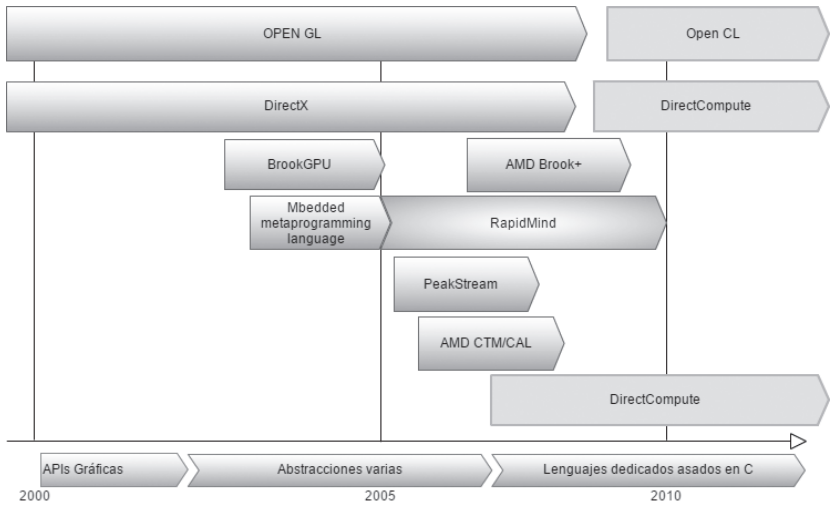
Fuente Elaboración propia

Lenguajes para la creación de algoritmos en GPU

La primera GPU fue programada como una calculadora aritmética, luego se utilizaron las primitivas gráficas (por ejemplo dibujar un triángulo) en la implementación de algoritmos con mejor rendimiento que los presentados en algoritmos secuenciales. Estos programas presentaban alta complejidad para ser desarrollados, depurados y optimizados, y frecuentemente se encontraban errores. Sin embargo, sirvieron como prueba de concepto para identificar la posibilidad de incrementar el desempeño de ciertos algoritmos que antes operaban en CPU. Con el pasar del tiempo los fabricantes de GPU lanzaron lenguajes no gráficos para habilitar el uso de las GPU para propósito general. En la figura 4 se muestra un diagrama cronológico de los lenguajes usados para computación en GPU. Actualmente existen tres lenguajes de programación dominantes en el campo de la GPGPU: NVIDIA CUDA, DirectCompute y OpenCL (NVIDIA Corporation Tech, 2016a).

Hoy en día existen una serie de ambientes de desarrollo y aplicaciones que permiten realizar la programación de la GPU de forma más simple. En la tabla 2 se muestra un resumen de algunas herramientas de acuerdo con el enfoque requerido (Nardone, 2016); (Ramey, 2010).

Figura 4. Historia de los lenguajes de programación para computación usando GPU.



Fuente Elaboración propia

Tabla 2. Diferentes herramientas de desarrollo de acuerdo con el enfoque requerido

Enfoque	Ejemplo
Análisis numérico y aplicaciones de integración	Matlab, Mathematica, LabVIEW
Lenguaje paralelo implícito	PGI Acelerator, HMPP
Abstracción de capa o adaptador	PyCuDA, CUDA.NET, jCUDA
Lenguaje de integración	CUDA C/C++, PGI CUDA Fortran
Api para el dispositivo de bajo nivel	CUDA C/C++, DirectCompute, OpenCL

Uno de los lenguajes más utilizados para la programación de GPU es CUDA, el cual «maneja un concepto de programación en el que se puede ejecutar código de forma secuencial mediante la CPU o paralela usando la GPU. El kernel hace la paralelización del código, asignando tareas específicas a los diferentes threads dispuestos para la aplicación» (Acosta et al., 2012). A continuación, se describe como se implementaría una sumatoria de matrices en un procesador y en una GPU.

Para el procesador *monocore* el algoritmo sería muy similar al mostrado en la figura 5, en donde se puede observar que tardará tantos ciclos de procesador como número de elementos tenga la matriz y otros adicionales para las instrucciones de bifurcación inherentes al ciclo *for*.

Figura 5.

Código de una suma matricial en un procesador de un solo núcleo.

```
float *C = malloc(N * sizeof (float));
for (int i = 0; i < N; i++) {
C[I] = A[i] + B[i];
}
```

Fuente Elaboración propia.

En la Figuras 6 y 7 se muestran las líneas de código necesarias para implementar el algoritmo de la sumatoria de matrices en una GPU. El procedimiento es el siguiente:

- Declaración de las funciones (función que realiza el Host).
- Asignación de memoria en la GPU para las entradas y salidas.
- Copiado de variables de la memoria del Host al Device.
- Paralelización y ejecución de las tareas por realizar en la GPU.
- Copiado de los resultados de la memoria del Device al Host.
- Liberación de memoria (Barr, 2016).

Figura 6. Código de la declaración de la función que ejecuta el Host.

```
_global_ void
cudaAddVectorsKernel(float *a, float *bo, float * c) {
    unsigned int index=blockIdx.x*blockDim.x + threadidx.x;
    c[index]=a[index] +b[index];
}
```

Fuente Elaboración propia

Figura 7. Código de para la ejecución de una suma matricial en una GPU.

```
void cudaAddVector(const float* a, const float* o, float* c, size) {
//Por el momento, suponemos a y b fueron creados antes del llamado de
esta función
//dev_a, dev_b (para entradas) y dev_c (para salida) serán arreglos en la
GPU
float * dev_a;
float * dev_b;
float * dev_c;
//Asignar memoria en la GPU para las entradas:
cudaMalloc ( (void**) &dev_a, size*sizeof(float));
cudaMemcpy (dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);
cudaMalloc ( (void**) &dev_b, size sizeof(float));
cudaMemcpy (dev_b, b, size*sizeof (float), cudaMemcpyHostToDevice);
//Asignar memoria en la GPU para la salida:
cudaMalloc ( (void**) &dev_c, size*sizeof(float) ) :
//El mínimo puede ser 32
//Límite de pascal 3584
const unsigned int threadsPerBlock=3584;
//Cuántos bloques se requieren para terminar la operación
const unsigned int blocks=ceil (size/float(threadsPerBlock) ) ;
//Llamada al kernel
cudaAddVectorskerne << blocks, threadsPerBlock >>
    (dev_a, dev_b, dev_c);
//Copiar la salida desde la GPU al host (se asume aquí que la memoria del
host
//para la salida ha sido calculada)
cudaMemcpy (c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);
//Liberar la memoria de la GPU
cudaFree (dev_a);
cudaFree (dev_b);
cudaFree (dev_c);
}
```

Fuente Elaboración propia

Aplicaciones y tendencias

Las aplicaciones en GPU han evidenciado la capacidad del aumento de la velocidad en la ejecución de algoritmos de manera eficiente. Algunas de las áreas en las que se destaca el uso de las GPU son: algebra lineal, procesamiento de imágenes, algoritmos de ordenamiento y búsqueda, procesamiento de consultas sobre bases de datos, análisis financieros, mecánica de fluidos computacional, predicción meteorológica, genoma humano, química cuántica, dinámica molecular, redes neuronales y criptografía (Guim & Rodero, 2012).

El éxito de las GPU se debe en gran medida a su bajo costo, entendido este como la relación de desempeño por dólar, gracias a la oferta y la demanda en el mercado del entretenimiento (Guim & Rodero, 2012), por lo cual es cada vez más común utilizar las GPU para desarrollar algoritmos que aprovechen el paralelismo del hardware intrínseco a este.

La tendencia actual en GPU es la reducción del consumo de potencia mediante la disminución del ancho del canal de los transistores, lo que permite una menor potencia dinámica y consumo energético total de la GPU. Además, se observa que las nuevas arquitecturas tratan de solventar los problemas de cuello de botella presentados en el acceso a memoria mediante nuevas técnicas como el bus NVlink de Nvidia y cachés más eficientes (NVIDIA Corporation Tech, 2016a).

Conclusiones

El incremento de velocidad en el algoritmo que se pueda lograr en una GPU depende altamente del paralelismo intrínseco de este; por tal motivo, no todos los algoritmos se ejecutarán más rápido cuando se realiza una migración desde un sistema secuencial en CPU a uno en GPU o híbrido. Hay que tener claro que uno de los posibles cuellos de botella que se pueden presentar es la carga de datos hacia la GPU y luego la recuperación de los datos ya procesados por esta, lo que conlleva a que, dependiendo del problema computacional, la solución sea optimizar el algoritmo en el procesador mediante técnicas que reduzcan el número de instrucciones o accesos a memoria lentas por parte este, en lugar de usar una GPU.

Aunque en la literatura se encuentran un gran número de artículos que demuestran el aumento de velocidad logradas en la GPU vs CPU, se debe tener cuidado al analizar estos resultados debido a que es frecuente que estas investigaciones estén sesgadas en cuanto al esfuerzo dedicado en la optimización del algoritmo en la GPU frente al de la CPU, el cual, generalmente, es programado en la mayoría de casos en un solo núcleo de la CPU y sin realizar los debidos procesos de optimización, por lo que las velocidades de 100x o más que algunos artículos presentan podría verse drásticamente reducidas si se hace una juiciosa optimización en el algoritmo de la CPU.

Referencias Bibliográficas

Acosta, F., Segura, O., Ospina, A. (2012). Guía y fundamentos de la programación en paralelo. *Revista en telecomunicaciones e informática,* 2(4), 81-97.

Asanovic, K., Bodik, R., Bryan, Catanzaro, C., et al. (2006). The Landscape of Parallel Computing Research: A View from Berkeley. *EECS Department, University of California, Berkeley, Tech.* Recuperado de https://pdfs.semanticscholar.org/1400/74c97f0849b06d63a9b2832f6c185a0f82f6.pdf

Barr, A. (2016). GPU Programming, lecture1: Introduction, class notes for CS 179. California Institute of Technology-Caltech. Recuperado de http://courses.cms.caltech.edu/cs179

Brodtkorb, A., Hagen, T., Saetra, M. (2013). Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing,* 73(1), 4–13.

Chiapppetta, M. (2016). Intel Xeon Processor E5 v4 Family Debut: Dual E5-2697 v4 With 72 Threads Tested. *HotHardWare.* Recuperado de http://hothardware.com/reviews/intel-xeon-processor-e5-v4-family-debut-dual-e5-2697-v4-with-72-threads-tested?page=3#P-0jdW7qxzbywrTHp.99

Guim, F., Rodero, I. (2012). *Arquitecturas de computadores avanzadas.* Barcelona: Universidad Oberta de Catalunya.

Intel Corporation. (2016). Especificaciones del procesador Intel® Xeon® Processor E5-2697 v4. Recuperado de http://ark.intel.com/es/products/91755/Intel-Xeon-Processor-E5-2697-v4-45M-Cache-2_30-GHz

Jaros, J., Pospichal, P. (2012). A Fair Comparison of Modern CPUs and GPUs Running the Genetic Algorithm under the Knapsack Benchmark. Camberra, Australia: *ANU College of Engineering and Computer Science,*

GPU. NVIDIA Tesla P100 whitepaper. Recuperado de https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

NVIDIA Corporation, Tech. (2016) Tecnología de interconexión de alta velocidad nvidia nvlink. Recuperado de http://www.nvidia.es/object/nvlink-es.html.

Ramey, W. (2010). Languages APIs and Development Tools for GPU Computing. In GPU Technology Conference, 2010.

Sangjin, H. (2016). On Fair Comparison between CPU and GPU. *Berkeley University* Recuperado de https://people.eecs.berkeley.edu/~sangjin/2013/02/12/CPU-GPU-comparison.html

Torun, M., Yilmaz, O., Akansu, A. (2016). FPGA, GPU, and CPU implementations of Jacobi algorithm for eigenanalysis. *Journal of Parallel and Distributed Computing,* 96, 172–180.