Arquitecturas Híbridas o Heterogéneas

Programación de arquitecturas híbridas

Carlos Andrés Balaguera Peña 2130051
Universidad Industrial de Santander
Ingeniería de Sistemas

Daniel Alfonso Vega Camelo 2120486
Universidad Industrial de Santander
Ingeniería de Sistemas

Abstract – This document is a short description of what is a heterogeneous or hybrid architecture , where you can find , how they work and how to program them , also talk about CUDA , one of the most used platforms today to take advantage of these architectures , one reason for this is because it will make an analysis of a program written in the programming language CUDA which shows how these structures advantage when using parallel computing , assigning different types of tasks to appropriate types of processors to increase performance

Keywords — CUDA; Heterogeneous Architecture Platform; Parallel Programming.

Resumen – Este documento es una corta descripción sobre qué es una arquitectura heterogénea o híbrida, donde se pueden encontrar, cómo funcionan y cómo programarlas, también se habla sobre CUDA, una de las plataformas más utilizadas en la actualidad para aprovechar estas arquitecturas, una razón de esto es porque se va a hacer un análisis sobre un programa escrito en el lenguaje de programación de CUDA el cual muestra cómo se aprovechan estas estructuras al usar la computación en paralelo, asignando los diferentes tipos de tareas a los tipos de procesadores adecuados para aumentar el rendimiento.

Palabras claves — CUDA; Plataforma de Arquitectura Heterogénea; Programación en paralelo.

I. Introducción

Desde sus inicios los computadores han tenido unidades centrales de procesamiento (CPUs) las cuales están diseñadas para correr tareas generales de procesamiento de manera secuencial muy eficientemente, pero en las últimas décadas en los computadores se han ido incluyendo otros elementos de procesamiento distintos. El más destacado es la unidad de procesamiento gráfico (GPU), originalmente la GPU fue diseñada para

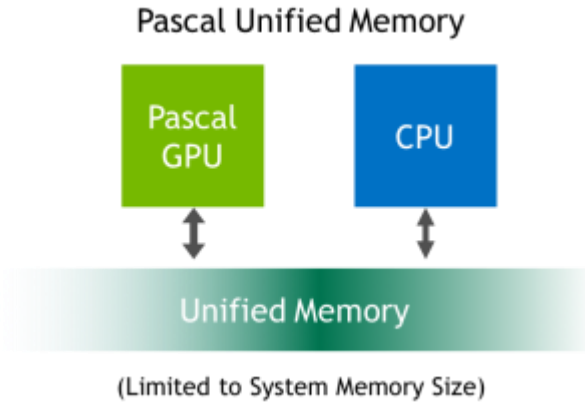
cálculos gráficos en paralelo, con el tiempo las GPUs se han vuelto más potentes y más generalizadas permitiéndoles ser aplicadas para tareas más generales de procesamiento en paralelo con excelente rendimiento.

Actualmente un gran número de aplicaciones requieren un gran rendimiento y eficiencia la cual es solo alcanzada por la computación en paralelo, pero las CPUs y las GPUs fueron diseñadas como elementos separados de procesamiento por lo cual no funcionan en conjunto muy eficientemente. Para poder aprovechar todo el potencial que tienen en conjunto estos dos elementos de procesamiento los diseñadores han tenido que pensar de una forma distinta para integrar bien estos dos elementos en una única plataforma que permita a los programadores escribir aplicaciones que usen eficientemente los dos elementos. NVIDIA fue la empresa pionera en este campo, ya que sabía que un hardware tan impresionantemente rápido (GPU) tenía que combinarse con herramientas de hardware y software intuitivas para aprovecharlo mejor, ya que las GPUs existían desde hace varios años, pero para usarlos era demasiado complicado ya que los desarrolladores que debían aprender de las últimas APIS gráficas del momento las cuales eran muy complicadas incluso para los que ya conocían lenguajes de programación básicos como OpenGl, así que en el año 2006 Nvidia junto con Ian Buck, creador de "Brook" el primer modelo de programación ampliamente adoptado para extender C con constructos cons datos paralelos, lanzó CUDA, la primera solución del mundo para computacion general en las GPUs. Estas GPUs han evolucionado al punto que muchas de las aplicaciones del mundo real se están implementado fácilmente en ellas y se ejecutan muchísimo más rápido que en sistemas con múltiples núcleos, por lo cual se dice que las arquitecturas de computación del futuro serán sistemas híbridos con GPU de núcleos paralelos trabajando en conjunto con CPU de múltiples núcleos.

II. Estado del arte:

Una de las últimas novedades en cuanto al ecosistema de las arquitecturas heterogéneas es la nueva arquitectura gráfica NVIDIA Pascal, la cual es la más avanzada del mundo y llevó una inversión de más de mil millones de dólares y un desarrollo de tres años. Las tarjetas gráficas basadas en Pascal proporcionan mucho más rendimiento y eficiencia energética gracias a un diseño basado en transistores FinFET proveniente de la compañía de semiconductores taiwanesa TSMC y funciones DirectX 12 para brindar mejores experiencias de uso, con la máxima velocidad y eficiencia energética, mejorando con creces la arquitectura utilizada anteriormente la Maxwell. Además, la memoria en la arquitectura Pascal se gestionará de forma unificada de cara a las aplicaciones, que podrán acceder tanto a la memoria de la CPU como de la GPU de un

modo transparente, esta memoria pasa a ser HBM2 (High Bandwidth Memory 2) con una distribución "apilada", de forma que se podrán integrar hasta 32 GB en una única tarjeta, también se gana en velocidad, la cual pasa a ser de hasta 1 TB/s. Por otro lado, se mejora la tecnología de interconexión entre chips en configuraciones de múltiples procesadores gráficos. NVLink es el nombre de esta nueva forma de interconectar los procesadores gráficos con hasta 12 veces más rapidez que usando PCI-Express, y que además podrán acceder mutuamente a la memoria de cada procesador sin complicaciones.



En conjunto, se mejoran los cuellos de botella de Maxwell, por ejemplo, en configuraciones multi procesador. Las arquitecturas gráficas explotan el paralelismo de sus chips, pero llega un momento en el que también es necesario poner a trabajar en paralelo a varios procesadores gráficos, lo cual suponía ralentizar los procesos en Maxwell. A nivel de cálculos en coma flotante, Pascal introduce cálculos con precisión mixta de 16 y 32 bits, siendo el doble de rápida en operaciones con precisión de 16 bits que en las que trabajan a 32 bits.

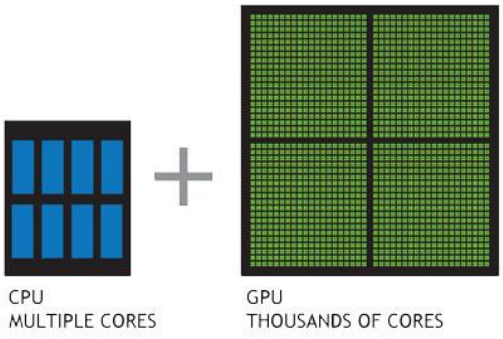
Este tipo de arquitectura no vino solo, también se presentaron los avances de las características que tendrá CUDA 8, dentro de las cuales está el soporte para la nueva arquitectura de GPU Pascal, incluyendo el nuevo acelerador Tesla P100. También nuevas capacidades de memoria unificada y una nueva librería para nvGRAPH GPU-Accelerated Graph Analytics.

III. Definiciones.

a. Arquitectura Heterogénea:

Cuando se habla de arquitectura heterogénea, se hace referencia a los diferentes sistemas que utilizan más de un tipo de procesador y que buscan maximizar el rendimiento de este sistema mediante una serie de instrucciones que permite la integración de la unidad de procesamiento central y la unidad de procesamiento gráfico en el mismo bus con memoria y tareas compartidas , una de las características de las arquitecturas heterogéneas o híbridas es que gracias a que usan más de un tipo de procesador se pueden dedicar a varias tareas distintas, como la GPU que se puede usar para hacer muchos cálculos matemáticos con una gran cantidad de datos, para procesamiento gráfico y la CPU se dedica a otras tareas que requieren mayor capacidad de procesamiento y ejecutar el sistema operativo, esto se da no debido a que se agreguen más procesadores iguales, si no por

agregar procesadores de distinto tipo, permitiendo realizar distintos tipos de tareas en paralelo y tareas particulares con el procesador especializado.



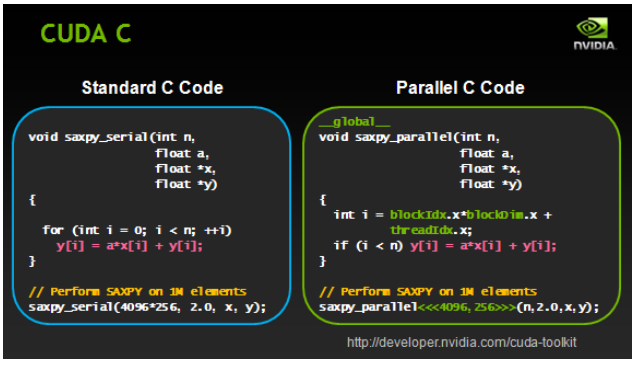
Usualmente la heterogeneidad en el contexto de la computación hace referencia a las diferentes arquitecturas de conjuntos de instrucciones (ISA), antiguamente cuando se hacía referencia a la computación heterogénea significaba que diferentes ISA tenían que ser manipuladas de distinta manera, mientras que actualmente los sistemas de arquitectura heterogéneo eliminan la diferencia para el usuario, un ejemplo de esto puede ser arquitecturas como CUDA (Compute Unified Device Architecture) que envuelve a tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por NVidia que permite usar una variación del lenguaje C para codificar distintos algoritmos en un GPU de Nvidia.

b. CUDA:

CUDA de las siglas en inglés Compute Unified Device Architecture (Arquitectura Unificada de Dispositivos de Cómputo) es una plataforma de computación paralela y un modelo de programación inventado por NVIDIA que aprovecha la potencia de la GPU para proporcionar un incremento bastante notable en el rendimiento del sistema, también busca explotar las ventajas de las GPU frente a las CPU de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten el uso de un gran número de hilos simultáneamente. Por esto, si una aplicación está diseñada usando una gran cantidad de hilos que realizan tareas independientes, una GPU puede ofrecer gran rendimiento en campos que van desde la biología computacional a la criptografía.

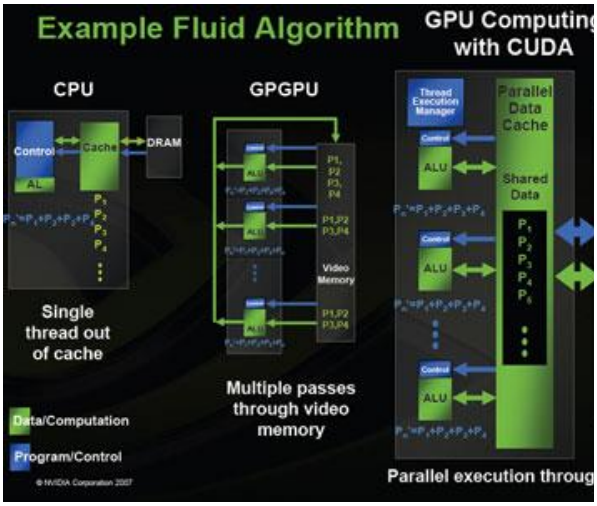
El lenguaje de programación de CUDA es C/C++ pero por medio de wrappers (adaptadores de código) también se pueden utilizar otros lenguajes conocidos tales como Python, Fortran y Java.

CUDA Y OpenCL, así como la mayoría de otros lenguajes de programación bastante avanzados pueden usar arquitecturas heterogéneas para aumentar el rendimiento de ejecución.



CUDA funciona de una manera sencilla, esta puede ser resumida en 4 pasos:

1. Se copian los datos de la memoria principal a la memoria de la GPU.
2. La CPU encarga el proceso a la GPU.
3. La GPU lo ejecuta en paralelo en cada núcleo.
4. Por último, se copia el resultado de la memoria de la GPU a la memoria principal.



Uno de los problemas que se puede presentar al programar en CUDA es que se pueden presentar cuellos de botella entre las CPU y la GPU debido al ancho de banda de los buses y sus latencias.

Características:

- Lenguaje de programación de alto nivel, basado en estándares abiertos
- MultiGPUs controlados, es decir, una sola CPU puede ser capaz de controlar varias GPU, lo cual permite un amplio campo de posibilidades en cuanto a la mejora de rendimiento.
- Caché de datos en paralelo, multiplicando el ancho de banda efectivo y reduciendo las latencias, al permitir que los grupos de procesadores trabajen juntos en el uso de la información contenida en la caché local.
- Transferencia de datos a través del bus PCI Express. Gracias a su elevado ancho de banda, especialmente en sus últimas versiones, las aplicaciones de cálculo pueden explotar unas tasas de transferencia de datos bastante elevadas, permitiendo unas lecturas más rápidas de y hacia la GPU

Limitaciones:

- Un código C válido puede ser rechazado debido a las limitaciones del propio hardware.
- Las primeras versiones de CUDA no admite recursividad, punteros a funciones, y otras limitaciones se están desapareciendo.
- En ocasiones puede existir un cuello de botella (es decir, la capacidad de procesamiento del dispositivo es mayor a la capacidad del bus) entre la CPU y la GPU por los anchos de banda de los buses y sus latencias.

- La correcta selección del número y disposición de threads es crítica para obtener un elevado rendimiento.
- Una de las principales dificultades al trabajar en CUDA, es la transferencia de datos y más aún si se dispone de más de una GPU, especialmente en situaciones donde una GPU requiere datos residentes en otra GPU. Todos estas inconvenientes se deben tener presentes a la hora de programar en CUDA y son responsabilidad del programador. Es decir, no basta con enviar reservar datos, hay que saber qué dependencias existen entre tareas, y cómo reducir el número de transferencias.

IV. Áreas de aplicación:

Las arquitecturas heterogéneas son altamente usadas en dispositivos system on chip tales como tablets, smartphones, consolas de videojuegos y se pueden encontrar en distintas plataformas en cada ámbito de la informática, desde servidores dedicados y máquinas de alto rendimiento.

Las plataformas para utilizar estas arquitecturas mayormente se utilizan para crear programas que van a hacer analisis de mucha información, como buscar un string dentro de 1TB de datos, análisis relacionados con Big Data, simulaciones de ingeniería, también se utilizan para mejorar el rendimiento de algunos algoritmos, como es el caso del equipo de HSA (Heterogeneous System Architecture) de AMD que analizó el rendimiento de Haar Face Detect, un algoritmo de análisis de vídeo usado para identificar rostros en una transmisión de video. El equipo comparó la implementación de este algoritmo con OpenCl con la implementación con HSA. La versión con HSA comparte datos sin problemas entre la CPU y la GPU sin copias de memoria o caché ya que se asigna cada parte de la carga de trabajo (workload) al procesador más adecuado aumentando su eficiencia. El resultado neto fue una mejora de 2,3x en el rendimiento y un 2,4x en la reducción de potencia requerida. Este nivel de rendimiento no es posible usando sólo CPUs multicore o solo GPUs, es importante resaltar que esto es hecho usando simples extensiones a C++, no es un modelo de programación totalmente diferente.

Algunos otros ejemplos de aplicaciones para el uso de estas arquitecturas heterogéneas con la plataforma de programación CUDA son:

Mejoras en el rendimiento de algoritmos: BLAS para rutinas algebraicas, FFT (transformada rápida de fourier) que se utiliza en la física computacional y procesamiento de señales, SPARSE para el tratamiento de matrices dispersas y RNG los cuales son algoritmos para generación de números aleatorios, todos con una mejora en el rendimiento mayor a 6x.

Identificar la placa oculta en las arterias: Los infartos son la principal causa de muerte en todo el mundo. Harvard Engineering, Harvard Medical School y Brigham & Women's Hospital se han unido para usar las GPU para simular el flujo sanguíneo e identificar la placa arterial oculta sin las técnicas invasivas de diagnóstico por imágenes o la cirugía exploratoria.

Analizar el flujo del tráfico aéreo: El National Airspace System administra la coordinación del flujo del tráfico aéreo en todo el país. Los modelos de computación ayudan a identificar nuevas formas de mejorar la congestión y mantener el movimiento del tráfico aéreo eficiente. Usando la potencia computacional de las GPU, un equipo de la NASA obtuvo un enorme aumento en el rendimiento y redujo el tiempo del análisis de diez minutos a tres segundos.

Visualizar moléculas: Una simulación molecular llamada NAMD (dinámica molecular a nanoescala) obtiene un gran aumento en el rendimiento con las GPU. La aceleración es resultado de la arquitectura paralela de las GPU, que les permite a los desarrolladores de NAMD llevar a la GPU partes de la aplicación con uso intensivo de la computación usando el kit de herramientas CUDA.

V. Código en CUDA

El código proporcionado está programado en C/C++, este código es acerca de la adición de matrices en una GPU usando CUDA debido a que su extensión es .cu, con el compilador de NVCC, este compilador es de propiedad de NVidia diseñado para ser utilizado con CUDA, NVCC separa en dos partes el código y envía una parte del código que será corrido en la CPU a un compilador en C como GCC o Intel C++ Compiler (ICC) o Microsoft Visual C compiler, y envía el código del dispositivo (la parte que se corre en la GPU) a la GPU. El código del dispositivo es compilado por NVCC que es basado en LLVM.

El código proporcionado está compuesto por las siguientes funciones:

- Mat_add: esta función se encarga de sumar las matrices que se proporcionen.

```
__global__ void Mat_add(float A[], float B[], float C[], int m, int n) {
   /* blockDim.x = threads_per_block                            */
   /* First block gets first threads_per_block components.      */
   /* Second block gets next threads_per_block components, etc. */
   int my_ij = blockDim.x * blockIdx.x + threadIdx.x;

   /* The test shouldn't be necessary */
   if (blockIdx.x < m && threadIdx.x < n)
      C[my_ij] = A[my_ij] + B[my_ij];
}  /* Mat_add */
```

- Read_matrix: esta función se encarga de leer las matrices que serán enviadas a la función Mat_add.

```
/*--------------------------------------------
 * Function:  Read_matrix
 * Purpose:   Read an m x n matrix from stdin
 * In args:   m, n
 * Out arg:   A
 */
void Read_matrix(float A[], int m, int n) {
   int i, j;

   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
         scanf("%f", &A[i*n+j]);
}  /* Read_matrix */
```

- Print_matrix: esta función se encarga de imprimir las matrices que le sean pasadas como su argumento.

```
/*--------------------------------------------------------
 * Function:  Print_matrix
 * Purpose:   Print an m x n matrix to stdout
 * In args:   title, A, m, n
 */
void Print_matrix(char title[], float A[], int m, int n) {
   int i, j;

   printf("%s\n", title);
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++)
         printf("%.1f ", A[i*n+j]);
      printf("\n");
   }
}  /* Print_matrix */
```

En el main se definen ciertas variables, entre ellas se encuentran unas de tipo entero que se utilizan para el tamaño de las matrices con el nombre m y n, también se definen unos punteros para asignar un espacio en la memoria del host con el nombre de *h_A, *h_B, *h_C y para el dispositivo *d_A, *d_B, *d_C.

```
int main(int argc, char* argv[]) {
   int m, n;
   float *h_A, *h_B, *h_C;
   float *d_A, *d_B, *d_C;
```

Luego se obtiene el tamaño de las matrices y se les asigna un tamaño determinado para guardar los valores de estas con malloc, luego se leen las matrices usando la función Read_matrix(), luego se imprimen usando la función Print_matrix(), luego se le reserva un espacio a las matrices en el dispositivo con malloc y posteriormente se copian de la memoria del host al dispositivo. luego se llama el kernel con m hilos en bloque de los cuales cada uno tiene n hilos, luego se sincronizan los hilos con la función cudaThreadSynchronize() y espera a su finalización, continúa copiando la información del dispositivo al host con la función cudaMemcpy() y sus respectivos parámetros, luego se procede a imprimir la matriz y para finalizar se libera la memoria del host y el dispositivo.

VI. Conclusiones

- Las arquitecturas heterogéneas se pueden aprovechar mucho mejor con la programación en paralelo debido a que ayuda a dividir una gran tarea en varias tareas para que estas puedan ser procesadas en el tipo de núcleo más adecuado de forma paralela, permitiendo que algunos procesos sean más eficaces.
- No en todos los casos la mejor opción es programar en paralelo, se debe tener en cuenta como es el problema a resolver y el hardware disponible.
- La tecnología CUDA permite desarrollar avances en muchos campos de la ciencia, llevando consigo, una nueva manera para desarrollar aplicaciones de software

más eficientemente, en donde las operaciones de cálculo intensivo que anteriormente tomaba meses y años, se desarrollan en mucho menos tiempo.

- La clave de esta arquitectura, es aprovechar la unificación computacional, la eficiencia multihilo, la potencialidad del hardware, aplicando múltiples hilos de ejecución de manera simultánea, mediante el desarrollo del lenguaje de programación CUDA, que le brinda al programador, un entorno amigable con algunas palabras agregadas al lenguaje C, adaptado para la nueva arquitectura

VII. Bibliografía

http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/

http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/hsa10.pdf

https://es.wikipedia.org/wiki/CUDA

http://www.nvidia.es/object/cuda-parallel-computing-es.html

http://progrevo.blogspot.com.co/2011/11/cuda-un-lenguaje-de-programacion-para.html

http://la.nvidia.com/object/gpu-architecture-la.html

https://devblogs.nvidia.com/parallelforall/cuda-8-features-revealed/

https://devblogs.nvidia.com/parallelforall/inside-pascal/

http://eprints.ucm.es/38720/1/Memoria_TFG.pdf