Problema del Viajante de Comercio con GPU

José Rafael Cenit Röleke, M. G. Arenas

Departamento de Arquitectura y Tecnología de Computadores.
ETSI Informática y de Telecomunicación. Universidad de Granada
Granada, España

ticjoserafael@gmail.com, mgarenas@ugr.es

Abstract. Este documento presenta el trabajo realizado en la ETSIIT de la Universidad de Granada. Se pone de manifiesto la ventaja y potencial de la programación paralela utilizando la tecnología CUDA para resolver un problema de optimización como es el Viajante de Comercio. Se analizará el rendimiento con diferentes entradas para el problema. A su vez este documento pretende ser un ejemplo de cómo resolver un problema mediante la tecnología que nos brinda la GPU. Finalmente este documento muestra un estudio comparativo de distintas versiones del algoritmo paralelo que muestran las ventajas en prestaciones de la GPU frente a la CPU para la asignatura Arquitectura y Computación de Altas Prestaciones.

Keywords: gpu, cuda, viajante comercio, tsp

Abstract. This document presents the work undertaken in the ETSIIT of the University of Granada. It manifest the advantage and potential of the parallel programming using CUDA technology for solving an optimization problem like the Travelling Salesman Problem. The performance will be analyzed with different inputs for the problem. At the same time this document pretends to be an example for how to solve a problem using the technology that the GPU has to offer. Finally this document shows a comparative studio for the different versions of the parallel algorithm showing the advantages of the GPU versus the CPU for the subject “Arquitectura y Computación de Altas Prestaciones”.

Keywords: gpu, cuda, travelling salesman, tsp

1 Introducción

Este documento presenta una aplicación distinta de la programación paralela utilizando una GPU y la tecnología CUDA [1] para resolver el problema del Viajante de Comercio, además de ser una práctica para la asignatura Arquitectura y Computación de Altas Prestaciones de la Universidad de Granada.

Se trata de un proyecto práctico de carácter investigador que puede ser perfectamente utilizado en el ámbito docente para la mejor comprensión de esta nueva forma de computación paralela.

En las diversas asignaturas que se cursan en los primeros años de la carrera como pueden ser Arquitectura de Computadores o Estructura de Computadores se estudian los distintos grados de paralelismo como pueden ser el paralelismo a nivel de instrucción ILP (Instruction Level Paralelism) y el paralelismo a nivel de datos DLP (Data Level Paralelism). Se utilizan herramientas de programación de memoria compartida como OpenMP.

Otras asignaturas como Sistemas Concurrentes y Distribuidos impartidas por el departamento de Lenguajes y Sistemas Informáticos muestran la programación paralela mediante el uso de sistemas de paso de mensajes como MPI, OpenMPI y variables compartidas.

El presente proyecto se enfoca en el estudio de la explotación de la tecnología CUDA impartida en la asignatura Arquitectura y Computación de Altas Prestaciones de la especialidad Ingeniería de Computadores para afianzar conceptos importantes como es el *mapeo* de las hebras en CUDA aplicado al problema del Viajante de Comercio. Los alumnos podrán proponer modificaciones del código y analizar el rendimiento.

2 Estado del Arte

En el año 2006 la empresa de procesadores gráficos NVIDIA lanza al mercado la primera GPU capaz de renderizar gráficos en 3D y que además incluye la posibilidad de ejecutar programas escritos en el lenguaje C utilizando el modelo de programación CUDA.

Desde entonces NVIDIA ofrece la capacidad de sus tarjetas gráficas para grandes centros de datos y otras instituciones tanto científicas como gubernamentales, siendo energéticamente eficientes para el procesamiento que son capaces de realizar. Además se utilizan versiones reducidas para potenciar las aplicaciones tanto de teléfonos móviles, ordenadores portátiles y tablets entre otros.

Los numerosos núcleos de los que dispone una tarjeta gráfica no solo se pueden utilizar para dibujar gráficos sino que además permiten ejecutar programas diversos, dando lugar al auge en nuestros días del denominado GPGPU computing.

3 Arquitectura y organización de la GPU

A continuación explicaremos la arquitectura y organización lógica de una GPU con CUDA tal y como se puede apreciar en la Figura 1.

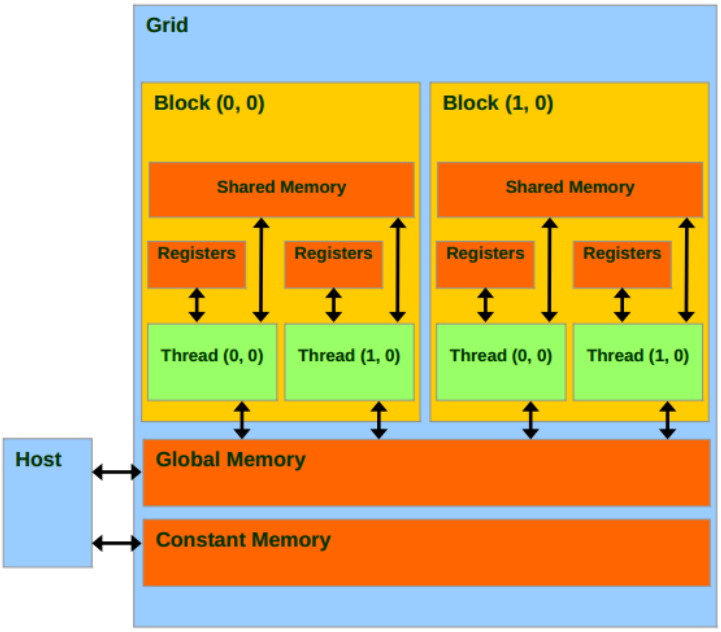

Fig. 1. Ejemplo de la arquitectura de memoria CUDA [2].

El primer elemento lógico que nos encontramos en CUDA es el hilo de ejecución o *thread*. En segundo lugar nos encontramos con el concepto de bloque o *block* y el tercer elemento lógico es rejilla o *grid*. Esta forma de organización está relacionada en cómo se gestionan los recursos dentro del dispositivo o *device*.

Tanto el número de bloques por *grid* como el número de hebras por bloque está limitado según las características físicas del dispositivo y/o de la generación del mismo. La denominada *compute capability* nos indica los límites del *device*.

Como podemos observar en la Figura 2 el número máximo de *threads* por bloque es de 1024 para todas las versiones de *compute capability*. Tanto los bloques como los *threads* pueden organizarse de forma unidimensional, bidimensional y/o tridimensional. Los *grids* pueden organizarse de forma unidimensional y/o bidimensional.

Esta plasticidad de la tecnología CUDA nos permite adaptar la estructura lógica del *grid* a la estructura de datos del problema que nos ocupe en ese momento. Por ejemplo si vamos a realizar un filtro sobre imágenes 2D lo adecuado sería organizar los bloques y las hebras de forma bidimensional puesto que la correspondencia seria 1:1. A este proceso se le denomina *mapping* o mapeo.

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

Compute Capability of Fermi and Kepler GPUs

Fig. 2. Tabla de especificaciones de cada versión de la capacidad de cómputo (*Compute Capability)* [3].

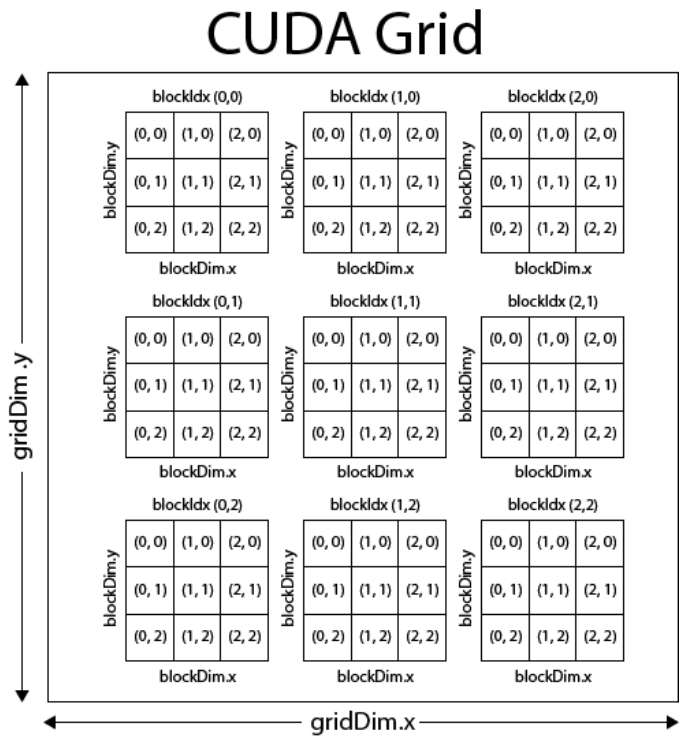

Fig. 3. Ejemplo de organización bidimensional en CUDA [4].

Cuando lanzamos un conjunto de hebras en CUDA la unidad mínima de ejecución física denominada *warp* ejecuta 32 hebras. Es decir aunque nuestro programa solo lanzase una sola hebra realmente se lanzan 32 hebras simultaneas donde la ejecución es la misma para todas ellas porque comparten el mismo contador de programa.

El programa que ejecuta cada hebra se denomina *kernel*. Este programa es ejecutado de la misma forma por cada hebra. Antes de lanzar un *kernel* necesitamos haber asignado previamente la memoria en el *device* copiando en dicha zona de memoria los datos que necesitemos. Esto es, se copian los datos del *host* al *device*.

Una vez finalizada la ejecución de nuestro *kernel* se realiza la operación inversa, se copian los resultados del *device* al *host*.

4 Proyecto: optimización del TSP mediante la GPU

Este proyecto aborda la computación en GPU para resolver el problema del viajante de comercio nombrado de aquí en adelante por sus siglas en inglés (TSP) [5].
El TSP es un problema que se encuentra en la categoría NP-Completo puesto que no se conocen algoritmos que sean capaces de generar la solución óptima en tiempo polinomial [6].

Es un problema combinatorio que se basa en: dadas varias ciudades, encontrar el camino más corto que las recorra todas una sola vez volviendo a la ciudad origen. El amplio estudio del TSP dentro de las ciencias de la computación lo convierten en un problema emblemático que tiene repercusiones y aplicaciones para la resolución de problemas de la vida real.

Las aplicaciones del TSP son múltiples: la planificación, la logística, la fabricación de circuitos integrados, etc. Por ejemplo uno puede pensar en un brazo robótico industrial que realiza puntos de soldadura. Las ciudades en este caso serían cada uno de los puntos de soldadura y la resolución del problema nos daría la secuencia de puntos que tendría que soldar el brazo robótico para minimizar el tiempo total de este trabajo.

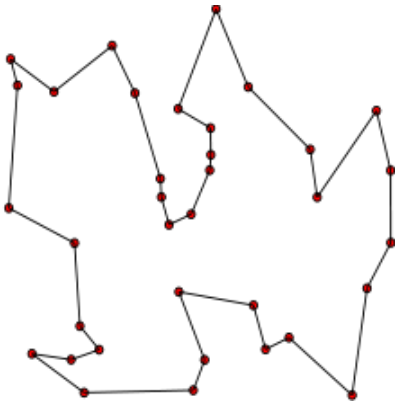

Fig. 4. Ejemplo de circuito obtenido para el TSP [7].

Dada la naturaleza del problema del TSP se ha desarrollado un algoritmo que haciendo uso de una heurística sencilla, logra generar soluciones sub-óptimas, cercanas al óptimo real, en un tiempo de ejecución aceptable.
El algoritmo que se describe a continuación es el siguiente:

```
for()//8192 soluciones
for()//n iteraciones
for()//Calcular distancia de la solución
{
   //Matriz de distancias
   intercambiar elementos
   if(mejora)
   {
    distancia=nueva_distancia
   }
   else
   {
    restaurar elementos
   }
}
actualizar solución
```

La estructura de datos del problema es un vector unidimensional que presenta la siguiente forma:

$$[e0, e1,\ldots, eN-1, c0, eN+N+2,\ldots, eN+N+N+1, cN-1]$$

Donde *e* representa el elemento de la solución y *c* el coste asociado a dicha solución.

5 Paralelización del problema

La paralelización del problema se consigue haciendo que cada hebra aplique el algoritmo y genere una solución sub-optima. Por lo tanto como generamos 8192 soluciones se lanzarán 8192 hebras, cada una de ellas generando 1 solución sub-óptima.

Este número de hebras (8192) no es trivial. Se ha escogido este número puesto que si lanzásemos más hebras dependiendo del tamaño del problema tendríamos también que ocupar más memoria VRAM de la tarjeta gráfica y los tiempos de reserva, copiado y acceso a la memoria global de los resultados serían mayores. A su vez evitamos el control de divergencia al ser potencia de dos.

Se procede entonces paralelizando el primer bucle, puesto que si intentásemos una supuesta paralelización del segundo bucle el costo de sincronizar el resultado de las hebras en cada iteración mediante memoria compartida perjudicaría gravemente el rendimiento.

La organización propuesta es la siguiente, unidimensional para los bloques y unidimensional para las hebras. No obtendríamos ventaja alguna haciendo otro tipo de distribución puesto que esta distribución se adapta perfectamente a la naturaleza del problema.

Por lo tanto el esquema del mapeo para las 8192 hebras es el siguiente:

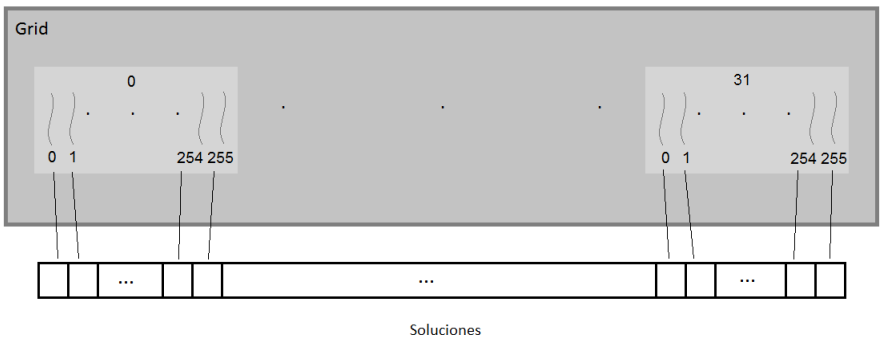

Fig. 5. Mapeo de trabajo por hebra.

Se lanzarán hebras organizadas en bloques, es decir, 32 bloques de 256 hebras cada uno dando un total de 8192 hebras.

Cada hebra escribirá su resultado en su parte correspondiente del vector de soluciones. Esto se realiza calculando el offset de cada hebra de la siguiente forma:

```
offset = ((blockIdx.x * blockDim.x) + threadId.x)*(dimension + 1)
```

Donde *dimensión* es el número de ciudades del problema a resolver. A esta variable se le suma una unidad puesto que cada solución de dimensión tiene su coste.

6 Soluciones propuestas

Una vez decidido como paralelizar, se han implementado tres versiones de un mismo algoritmo cuyas diferencias son:

- La primera versión utiliza memoria *shared* para guardar la matriz de distancias y *registros* para él vector solución. Esta versión del algoritmo ha resultado ser la más rápida.
- La segunda versión algoritmo utiliza *registros* para él vector solución.
- La tercera versión utiliza memoria *shared* para el vector de números aleatorios y *registros* para él vector solución. Esta versión del algoritmo ha resultado ser la más lenta. Se lanza n/45 veces siendo *n* el número de ciudades.

La utilización de la primera, la segunda o la tercera versión del algoritmo depende del tamaño del problema, puesto que hay problemas, que por su tamaño no se pueden abordar con la versión primera, puesto que la matriz de distancias ocupa demasiado y no es posible albergarla en la memoria tipo *shared* del dispositivo.

- Número de ciudades menor o igual a 45: Primer algoritmo
- Número de ciudades entre 45 y 50: Segundo algoritmo
- Número de ciudades mayor que 50: Tercer algoritmo

Como disponemos de un tamaño de 16KB para la memoria *shared* nos caben 16384 elementos de 8 bits. Con 45 ciudades llegamos a ese límite para almacenar la matriz de distancias puesto que 45x45x8 = 16200 elementos. Por este motivo el primer algoritmo es el más eficiente en tiempo de ejecución al tener la matriz de distancias en memoria *shared* y además el vector solución en *registros*. A partir de un tamaño mayor de 45 ciudades y menor de 50 ciudades ya no podemos introducir en memoria *shared* la matriz de distancias por lo que hemos utilizado el segundo algoritmo.

Cuando el número de ciudades es mayor a 50 utilizamos el tercer algoritmo lanzado *n*/45 veces siendo *n* el número de ciudades. En cada ejecución el tamaño máximo del vector de solución almacenado en registros es de 45. En la memoria *shared* se almacenan 4000 números aleatorios de 32 bits. La matriz de distancias permanece en memoria *global* por la imposibilidad de almacenarla en memoria *shared*. Este algoritmo produce soluciones peores puesto que las permutaciones se realizan sobre partes del vector solución y no sobre el vector solución completo.

El pseudocódigo de cada versión se expone a continuación:

```
for()//8192 soluciones
for()//n iteraciones
for()//Calcular distancia de la solución
{
     //Matriz de distancias en memoria shared
     //Vector solución en registros

     intercambiar elementos
     if(mejora)
     {
       distancia=nueva_distancia
     }
     else
     {
       restaurar elementos
     }
}
actualizar solución
```

Fig. 6. Primera versión del algoritmo, ejecutada cuando número de ciudades [1, 45].

```
for()//8192 soluciones
for()//n iteraciones
for()//Calcular distancia de la solución
{
    //Vector solución en registros

    intercambiar elementos
    if(mejora)
    {
        distancia=nueva_distancia
    }
    else
    {
        restaurar elementos
    }
}
actualizar solución
```

Fig. 7. Segunda versión del algoritmo, ejecutada cuando número de ciudades (45, 50].

```
for()//8192 soluciones
for()//n iteraciones
for()//Calcular distancia de la solución
{
   //Vector solución en registros
   //Números aleatorios en memoria shared

   intercambiar elementos
   if(mejora)
   {
       distancia=nueva_distancia
   }
   else
   {
       restaurar elementos
   }
}
sumar distancia de última ciudad de la solución parcial
  actual con primera ciudad del siguiente trozo
actualizar solución
```

Fig. 8. Tercera versión del algoritmo, ejecutada n/45 veces cuando número de ciudades (50, +∞].

7 Resultados paralelización I

Los tiempos obtenidos quedan reflejados en la Tabla 1, la primera columna indica el número de ciudades y el resto de columnas el tiempo empleado tanto en CPU como en CUDA con distinto número de iteraciones. Todas ejecuciones se realizan sobre 8192 hebras.

Table 1. Tiempo (seg.) distintas ejecuciones del algoritmo con distintos tamaños de entrada.

Ciudades	CPU 10000	CUDA 10000	CPU 100000	CUDA 100000
8	1,11	0,01	10,84	0,16
10	1,42	0,02	14,03	0,21
15	1,97	0,03	19,70	0,29
20	2,58	0,05	25,53	0,47
29	3,54	0,08	35,35	0,69
45	5,27	0,14	52,53	1,20
46	5,40	1,01	53,95	9,83
48	5,66	1,04	56,41	10,41
52	6,07	7,21	60,33	72,27
120	14,2	21,08	141,64	207,77

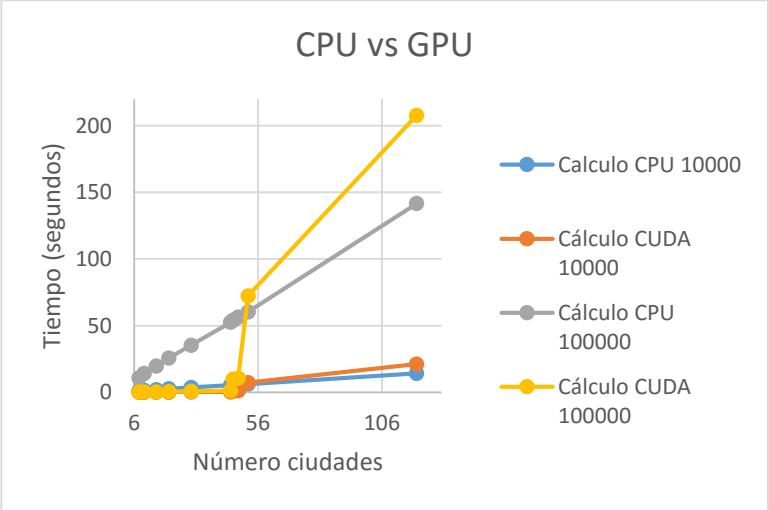

Fig. 9. Tiempo de cálculo de CPU frente a GPU. Como podemos apreciar cuando el número de ciudades es inferior a 45 el tiempo de cálculo en GPU es extremadamente pequeño obteniendo una ganancia significativa. A partir de 52 ciudades la CPU gana en tiempo de cálculo a la GPU.

En la Figura 9 podemos apreciar la diferencia entre los tiempos obtenidos con la CPU y con GPU. El problema se puede apreciar con claridad. Cuando el número de ciudades es inferior a 45 la diferencia de tiempos es bien clara no llegando al segundo en la GPU. Esto es debido a que tanto el vector local de soluciones de cada hebra cabe en sus *registros* y la matriz de distancias está en memoria *shared* que es aproximadamente 10 veces más rápida que la memoria *global*. Ahora bien cuando el número de ciudades es mayor que 50 tanto el vector soluciones, como el vector de números aleatorios, como el vector de distancias están en memoria global. Esto produce un cuello de botella considerable haciendo que prácticamente los accesos a memoria y la ejecución del programa sean secuenciales, eliminando así toda ganancia. En este punto la CPU supera en tiempo de cálculo a la GPU.

8 Resultados paralelización II

Los tiempos obtenidos quedan reflejados en la Tabla 2, la primera columna indica el número de ciudades y la segunda columna el tiempo. Todas ejecuciones se realizan sobre 8192 hebras.

Table 2. Tiempo (segundos) de las distintas ejecuciones del algoritmo con distintos tamaños de entrada.

Ciudades	CPU 10000	CUDA 10000	CPU 100000	CUDA 100000
52	9,18	1,73	94,57	16,45
120	20,8	3,82	219,21	37,74
150	26,61	4,74	280,38	47,07
202	35,17	6,50	368,77	64,11
561	99,89	40,02	1053,34	399,39

En la Figura 10 y Figura 11 podemos apreciar cómo se ha mejorado el tiempo de ejecución en la GPU cuando el número de ciudades es mayor a 50. Esto se ha conseguido lanzando varias veces el *kernel* en CUDA donde en cada ejecución se almacenan en registros 45 ciudades, y en memoria *shared* 4000 números aleatorios. Se consigue así minimizar el acceso a la memoria *global* y por lo tanto disminuir el tiempo de cálculo a costa de obtener soluciones menos óptimas puesto que las permutaciones se realizan en trozos de 45 elementos.

9 Repercusión del número de iteraciones

El número de iteraciones con el que lanzamos el programa determina la calidad de la solución obtenida para una entrada de ciudades concreta. A continuación se muestra una comparativa de la ejecución.

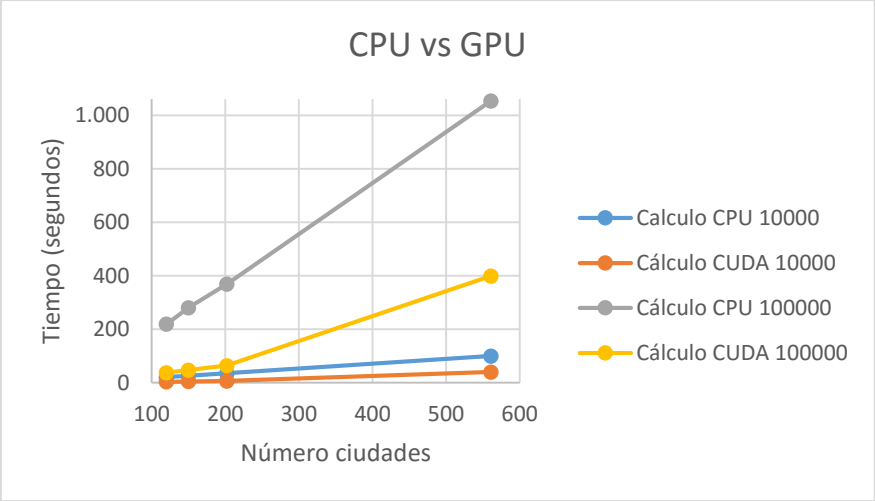

Fig. 10. Tiempo de cálculo de CPU frente a GPU. Como podemos apreciar ahora se obtienen ganancias significativas con más de 50 ciudades.

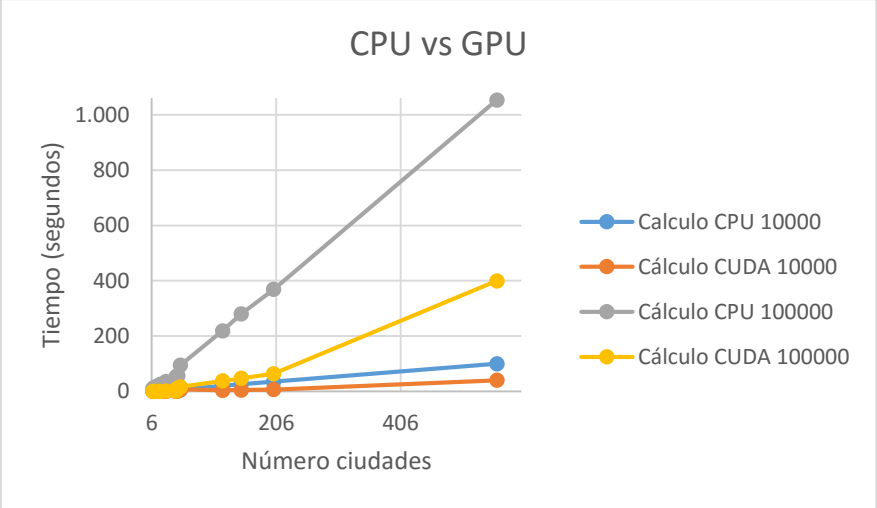

Fig. 11. Tiempo de cálculo de CPU frente a GPU. Unión de ambas gráficas anteriores.

Table 3. Distancia y tiempo de la ejecución del algoritmo para un mismo tamaño de entrada y distinto número de iteraciones.

Ciudades	Iteraciones	Tiempo (segundos)	Distancia
22	10	0,02	104
22	1000	0,19	72

Como se puede observar en la Tabla 3 el tiempo de ejecución es menor con 10 iteraciones. Por contrapartida se obtiene una mejor solución en la con 1000 iteraciones puesto que la distancia es menor (72 frente a 104).

10 Conclusiones

En esta práctica se ha comprobado como la GPU es capaz de obtener ganancias significativas en problemas que en apariencia no son aptos de paralelizarse como los clásicos algoritmos de procesamiento de imágenes que encajan perfectamente con la filosofía de programación CUDA. Algunos de los resultados presentados presentan ganancias de 58x cuando tenemos un número inferior a 45 ciudades.

Lo más destacable de esta práctica es la importancia de optimizar los algoritmos paralelos para que minimicen el acceso a la memoria global poniéndose de manifiesto en el apartado 7 y 8.

Finalmente esta práctica podrá ser reutilizada en la asignatura "Arquitectura y Computación de Altas Prestaciones" impartida en la especialidad de Ingeniería de Computadores para que los alumnos observen y deduzcan a partir de los distintos tamaños de entrada, cuando se empieza a utilizar la memoria global de forma más intensiva, observándose la degradación de la ganancia. También podrán examinar el código y proponer mejoras o sencillamente aprender de él.

Referencias

[1] NVIDIA Corporation. (2016, May) nvidia.com. *URL* http://www.nvidia.com/object/what-is-gpu-computing.html

[2] Delbosc. Nicolas. 2016. Overview of the CUDA device memory model. URL https://sites.google.com/site/computationvisualization/_/rsrc/1321741184041/programming/cuda/article1/memory.png?height=359&width=400

[3] StreamComputing BV. 2016. Compute Capability of Fermi and Kepler GPUs. URL http://streamcomputing.eu/wp-content/uploads/2012/10/Compute-Capability-of-Fermi-and-Kepler-GPUs.png

[4] Microway Inc. 2016. CUDA Grid Block Thread Structure. URL http://www.microway.com/wp-content/uploads/CUDA-GridBlockThread-Structure.png

[5] Wikipedia Inc. (2016, May) wikipedia.org. URL https://en.wikipedia.org/wiki/Travelling_salesman_problem

[6] Ellis Horowitz, Sartaj Sahni, "Fundamentals of Computer Algorithms", Rockville, Maryland, U.S.A., 1978.

[7] Wikipedia Inc. 2016. Solution of a travelling salesman problem. URL https://upload.wikimedia.org/wikipedia/commons/thumb/1/11/GLPK_solution_of_a_travelling_salesman_problem.svg/220px-GLPK_solution_of_a_travelling_salesman_problem.svg.png