PRÁCTICA 4 Y 6

PROGRAMACIÓN CONCURRENTE Y PARALELA

Hernán Iglesias Ramos

Introducción

En esta práctica nos vamos a centrar en una de las soluciones al gran problema de aumentar la velocidad de ejecución de un programa, para ser más claro, en las GPU.

Este hardware nos permite acelerar aún más los programas, pasando del paralelismo de CPU al paralelismo de GPU, que nos producirá a su vez numerosos cambios en la resolución de nuestros problemas.

En este proyecto, llevaré a cabo un análisis teórico del problema a resolver (Calcular un fractal) y lo estudiaremos junto a alguna que otra solución que utilice el cálculo multinúcleo en CPU.

También desarrollaremos este problema en GPU CUDA, donde podremos analizar el comportamiento de las GPU NVIDIA a la hora de ejecutar un código en CUDA, y, por último, haremos una comparación entre las ventajas del nuevo hardware respecto a los datos originales de la CPU.

Análisis Teórico

$$TPP_{dp} = chasis * nodos_{chasis} * sockets_{nodo} * cores_{socket} * clock_{GHz} * \frac{Nºflop}{ciclo_{dp}}$$

A partir de esta fórmula, junto con los datos de cada procesador, podemos calcular el TPP de cada máquina:

$$TPP_{i3} = 1 * 1 * 1 * 2 * 3{,}1 * 8 = 49{,}6\ GFlop$$

$$TPP_{Ryzen} = 1 * 1 * 1 * 8 * 3{,}6 * 16 = 460{,}8\ GFlop$$

$$TPP_{Xeon} = 1 * 1 * 2 * 4 * 2{,}4 * 8 = 153{,}6\ GFlop$$

(Estos datos se verifican en la siguiente imagen proporcionada por los profesores de la asignatura en las dispositivas de teoría).

Modelo	Nº Sockets	Cores x Socket	Frecuencia (GHz)	Flop por cycle (DP – SP)	TPP_{dp} (Gflop)	TPPsp (Gflop)
Xeon PHi 31S1P	57	4	1.1	4 – 8	1003.0	2006.0
Xeon E5-2603 v3	2	6	1.6	16 – 32	307.0	614.0
Xeon E5620	2	4	2.4	8 – 16	153.6	307.2
Ryzen 7 3700x	1	8	2.2 / 3.6	16 – 32	281.6 / 460.8	563.2 / 921.6
I3-2100	1	2	3.1	8 – 16	49.6	99.2
Cortex-A15	1	4	2.2	2 – 8	17.6	70.4

Conociendo el TPP (número de flops por segundo) de cada máquina, podremos conocer el t_c.

$$t_c = \frac{1}{TPP_{dp}}$$

Intel I3-2100

-Paralelo

$$t_c = \frac{1}{TPP_{i3} * 10^9} = 2{,}01613E-11seg$$

-Secuencial

$$t_c = \frac{2}{TPP_{i3} * 10^9} = 4{,}0323E-11seg$$

Xeon E5620

-Paralelo

$$t_c = \frac{1}{TPP_{Xeon} * 10^9} = 6{,}5104E-12seg$$

-Secuencial

$$t_c = \frac{8}{TPP_{Xeon} * 10^9} = 5{,}2083E-11seg$$

Ryzen 7 3700x

-Paralelo

$$t_c = \frac{1}{TPP_{Ryzen} * 10^9} = 2{,}1701E-12seg$$

-Secuencial

$$t_c = \frac{8}{TPP_{Ryzen} * 10^9} = 1{,}7361E-12seg$$

Complejidad temporal y espacial del problema

El problema que vamos a resolver se divide en tres partes:

-Calcular el fractal.

-Calcular el promedio.

-Binarización.

Sobre el cálculo del fractal del conjunto de Mandelbrot, se define por:

$$z_0 = 0$$

$$z_{k+1} = z_k^2 + c, c \epsilon \mathbb{C}$$

Disponemos de los tamaños en píxeles (xres e yres). Y xmin, ymin, xmax, ymax son las coordenadas de las esquinas de la zona a mapear en el vector.

También tendremos otro parámetro (maxiter) para limitar el numero de iteraciones a ejecutar.

Sobre el cálculo de la media, es mucho simple que el cálculo del fractal se define por:

$$\mu_A = \frac{\sum_{i=0}^{n} A_i}{n}$$

Sobre el cálculo de binariza, consiste en a partir del fractal original, iterar por cada posición del fractal y comprobar que el píxel tiene un valor mayor o menor a la media calculada.

Fractal

La complejidad espacial, va a depender del tamaño de la imagen que se deseé generar. Por lo general, se dice que la complejidad espacial de la función Mandelbrot es $O(N^2)$, donde N es el tamaño de la imagen en píxeles, en nuestro caso xres o yres. Esto significa que se necesitan:

$$xres * yres * sizeof\ (B)$$

La complejidad temporal, debido a ciertas condiciones, puede no ser la misma para cada píxel, por lo que, para el cálculo de la complejidad del problema, deberíamos centrarnos en el peor caso.

i) $T_{mandel_{secuencial}}(xres, yrex, maxiter) = xres * yres * maxiter * t_c$

ii) $T_{mandel_{multihilo}}(xres, yrex, maxiter, p) = \frac{xres*yres*maxiter}{p} * t_c$

Promedio

La complejidad espacial, de esta función es sencilla de calcular ya que depende del tamaño de la matriz, en nuestro caso el vector original, al utilizar *row-major.* Por tanto, al pasarle el tamaño mediante los datos de entrada xres e yres, la complejidad será:

$$xres * yres * sizeof\ (B)$$

La complejidad temporal, sería:

i) $T_{media_{secuencial}}(xres, yrex) = 2 + xres * yres * t_c$

ii) $T_{media_{multihilo}}(xres, yrex, p) = 2 + \frac{xres}{p} * yres * t_c$

En el caso paralelo, encontraríamos más problemas, al existir una reducción de suma, algo complejo de paralelizar. En el ámbito de OpenMP, se podría resolver con un:

#pragma omp parallel for reduction

En CUDA, se tendría que utilizar librerías como CUBLAS.

Binariza

La complejidad espacial, al igual que la función de media, depende del tamaño del vector. Por tanto, la complejidad será:

$$xres * yres * sizeof\ (B)$$

La complejidad temporal, sería:

i) $T_{binariza_{secuencial}}(xres, yrex) = xres * yres * t_c$

ii) $T_{binariza_{multihilo}}(xres, yrex, p) = \frac{yres}{p} * xres * t_c$

SPEED-UP TEÓRICO

Intel I3 – tiene 2 núcleos, con lo que el SpeedUp será 2.

Xeon – tiene 2 procesadores, y por cada procesador 4 núcleos, tendrá un SpeedUp de 8.

Ryzen – tiene 8 núcleos, por lo que el SppedUp será 8.

Esto quiere decir que el speedup será siempre el mismo, constante. Algo que en los tiempos

experimentales nos dice que no es cierto, ya que el tiempo teórico no contempla diferentes variables.

TABLAS cpu

(Tenemos los tiempos ordenados según el tamaño [256-8192], primero los tiempos teóricos tanto el secuencial como el paralelo, después, el tiempo en Python, y por últimos los tiempos en C, siguiendo diferentes soluciones, task, collapse, Schedule…)

yres	eo-secuencia	teo-paralelo	python	c-sec	c-p-task	c-p-collapse	chedule-dyn	-schedule-st
256	7,93E-03	3,98E-03	7,84E+00	1,43E-01	8,74E-02	8,22E-02	6,88E-02	8,53E-02
512	3,17E-02	1,59E-02	3,13E+01	5,22E-01	2,68E-01	2,72E-01	2,67E-01	3,01E-01
1024	1,28E-01	6,36E-02	1,25E+02	2,00E+00	1,09E+00	9,69E-01	9,32E-01	1,09E+00
2048	5,08E-01	2,54E-01	5,01E+02	7,99E+00	4,31E+00	3,81E+00	3,67E+00	4,24E+00
4096	2,03E+00	1,02E+00	N/A	3,19E+01	1,72E+01	1,53E+01	1,46E+01	1,70E+01
8192	8,12E+00	4,06E+00	N/A	1,28E+02	6,88E+01	6,09E+01	5,87E+01	6,78E+01

En esta primera tabla tenemos los tiempos que hemos sacado en la máquina **Intel I3-2100.**

yres	eo-secuencia	teo-paralelo	python	c-sec	c-p-task	c-p-collapse	chedule-dyn	-schedule-st
256	2,55E-03	3,20E-04	1,14E+01	1,80E-01	2,37E-01	4,34E-02	3,07E-02	3,20E-02
512	1,02E-02	1,28E-03	4,55E+01	6,49E-01	8,87E-01	1,31E-01	1,29E-01	1,81E-01
1024	4,10E-02	5,12E-03	2,73E+02	3,91E+00	4,01E+00	3,72E-01	3,23E-01	4,19E-01
2048	1,64E-01	2,06E-02	1,09E+03	1,56E+01	1,55E+01	1,50E+00	1,29E+00	1,77E+00
4096	6,55E-01	8,19E-02	N/A	6,24E+01	5,74E+01	6,01E+00	5,16E+00	7,01E+00
8192	2,62E+00	3,28E-01	N/A	2,49E+02	2,37E+02	2,40E+01	2,06E+01	2,77E+01

Y, por último, esta tabla recoge los tiempos sacados de la máquina **Xeon E5620.**

yres	eo-secuencia	teo-paralelo	python	c-sec	c-p-task	c-p-collapse	chedule-dyn	-schedule-st
256	8,54E-04	1,07E-04	4,96E+00	1,51E-01	8,81E-02	1,23E-02	2,24E-02	1,47E-02
512	3,42E-03	4,27E-04	1,98E+01	3,52E-01	2,58E-01	6,95E-02	8,06E-02	1,12E-01
1024	1,37E-02	1,71E-03	7,90E+01	1,41E+00	1,07E+00	2,22E-01	1,97E-01	2,61E-01
2048	5,46E-02	6,83E-03	3,16E+02	5,62E+00	4,34E+00	7,88E-01	7,41E-01	9,62E-01
4096	2,18E-01	2,73E-02	N/A	2,27E+01	1,66E+01	3,19E+00	2,95E+00	3,84E+00
8192	8,73E-01	1,09E-01	N/A	9,02E+01	6,71E+01	1,28E+01	1,19E+01	1,53E+01

Esta otra tabla recoge los tiempos sacados de la máquina **Ryzen 7 3700x.**

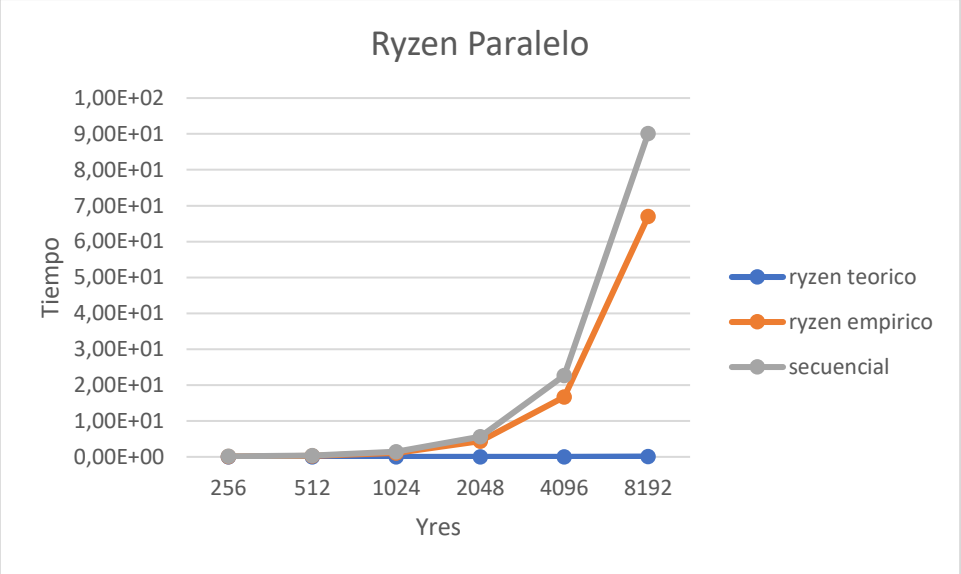
Análisis de los datos obtenidos

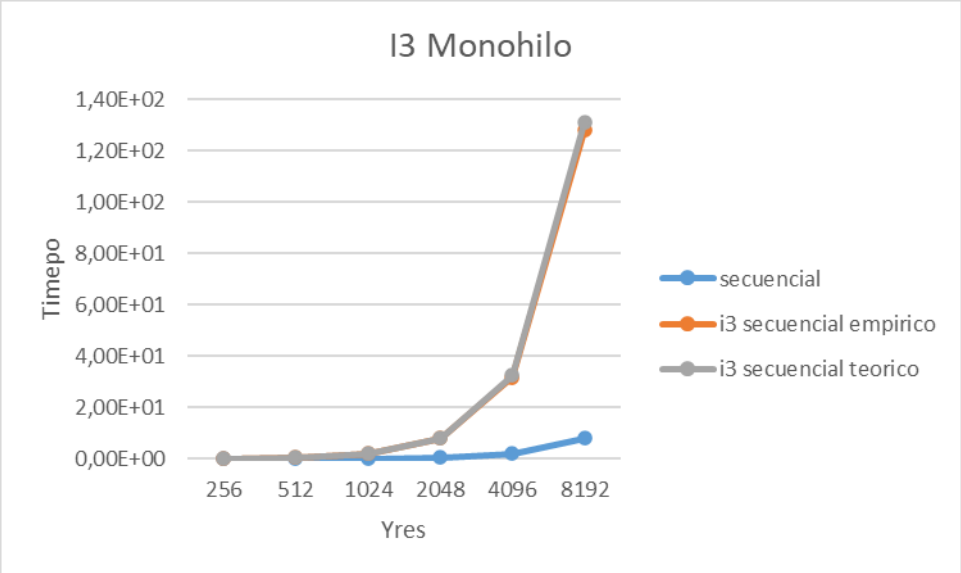
Los datos obtenidos en las diferentes máquinas tienen una similitud a los datos teóricos, hay que recalcar, que se asemejan, nunca van a ser exactamente los mismos, ya que, pueden influir numerosos factores propias de la máquina o a la hora de

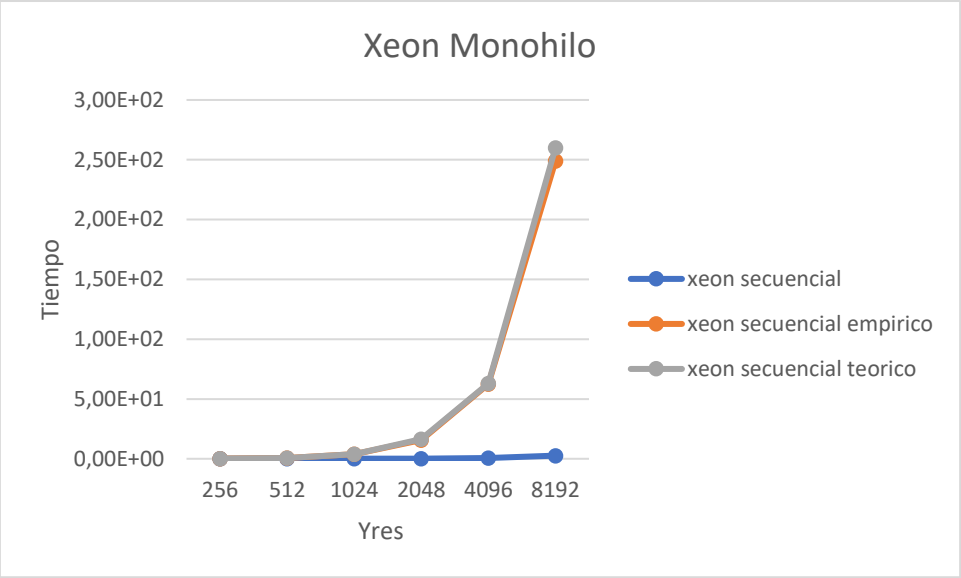
resolver los diferentes cálculos teóricos. Cumpliéndose así la función de correlación que nos dice:

$$T_{Empírico\ medido} \simeq T_{Teórico} * k_{constante}$$

Los comportamientos que he podido apreciar algo distintos a lo teórico ha sido en el caso de la máquina Ryzen, más en concreto en la parte de paralelo, al compararlo con cualquier otra máquina en el plano secuencial.







Alguno de los problemas que nos encontramos al resolver esta parte del proyecto fue el estudio de tiempos de respuesta, en concreto el del Xeon, donde los tiempos de los algoritmos en paralelo eran superiores a los secuenciales.

Todo esto se deberá a algún problema de caching ocasionado por los paralelos lo que hace que los secuenciales salgan perjudicados.

(enlace a una página en la que habla de este tipo de problema, y como poder solventarlo en nuestros equipos).

https://techblog.sdstudio.top/es/como-solucionar-el-problema-del-almacenamiento-en-cache-del-navegador-de-apalancamiento-en-wordpress/

La solución a esto sería modificar el fichero en cuestión, para que se ejecuten los algoritmos paralelos enviándolos a una carpeta cualquiera, y finalizar con los secuenciales, es decir, tomar una ejecución completa de los paralelos como nodo inicial.

De este modo, la ejecución paralela y la secuencial se aprovechan de la ventaja que esto produce.

A parte de esto, también hay que comentar, que Schedule Dynamic resulta ser el que más aceleración tiene, a causa del reparto de tareas a hilos que hace.

También destacamos que el de Tasks, es el peor, debido principalmente a sobrecargas en las tareas.

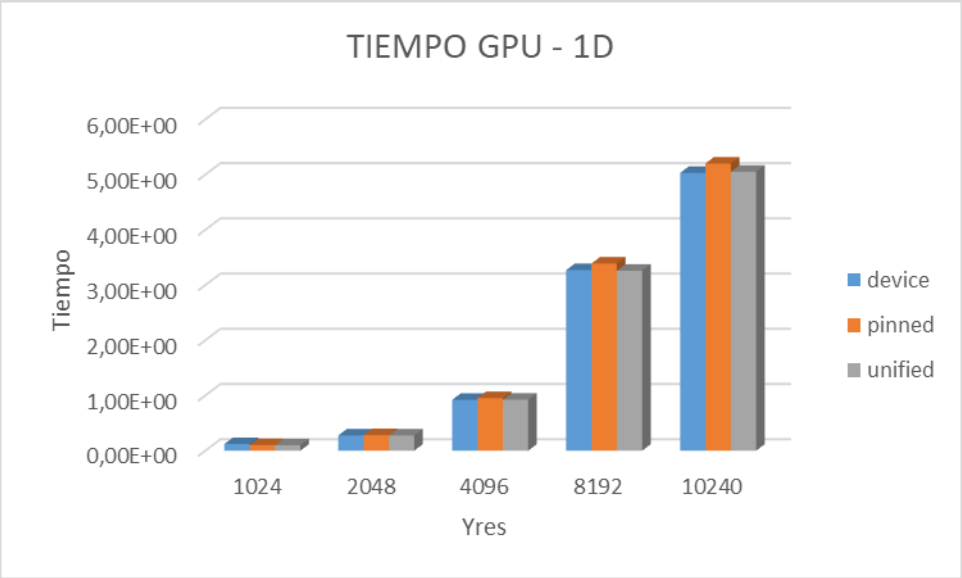
GPU

La diferencia entre diseñar un programa enfocado en la ejecución desde CPU y otro para GPU se encuentra en la organización de la memoria. A la hora de asignar memoria en un programa ‘normal’ (ejecutado en CPU) se utiliza la RAM, cuando al procesador le haga falta el dato en cuestión se cacheará. Los datos se almacenen en memoria principal y del resto se encargar el propio sistema.

En el caso de la GPU, el dispositivo no tiene acceso a la memoria RAM, para ello la GPU tendrá su propia VRAM que viene a ser la RAM de la GPU. Si se quiere procesar datos con la GPU se deberán encontrar en la VRAM, algo que no es lo mas adecuado. Existen otras memorias como la pinned, utilizando un puntero, y la unificada, que es una memoria compartida entre CPU y GPU.

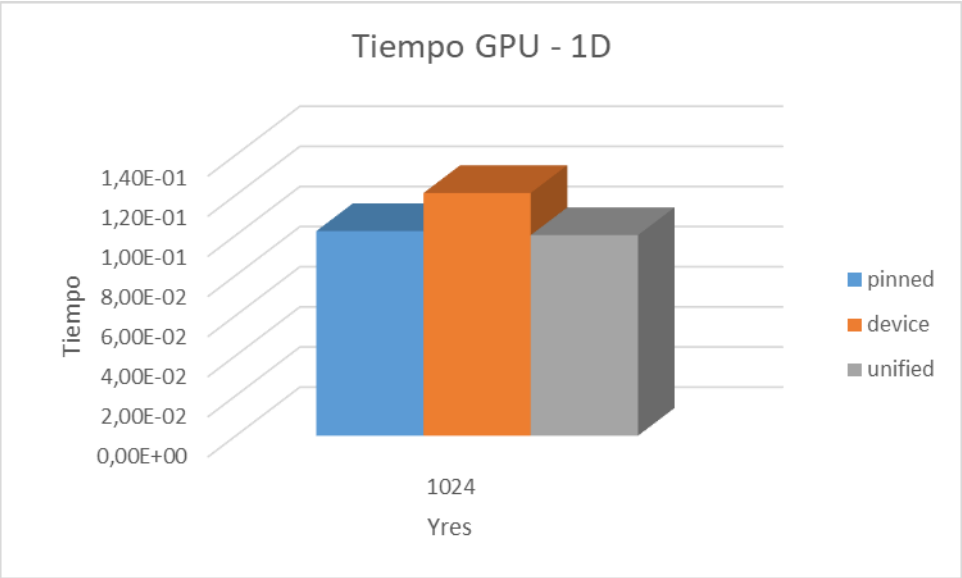
Hay que tener en cuenta la organización de los hilos, donde un bloque tendrá un máximo de 1024 hilos y un warp 32.

Análisis de GPU con diferentes tipos de memoria (1D)



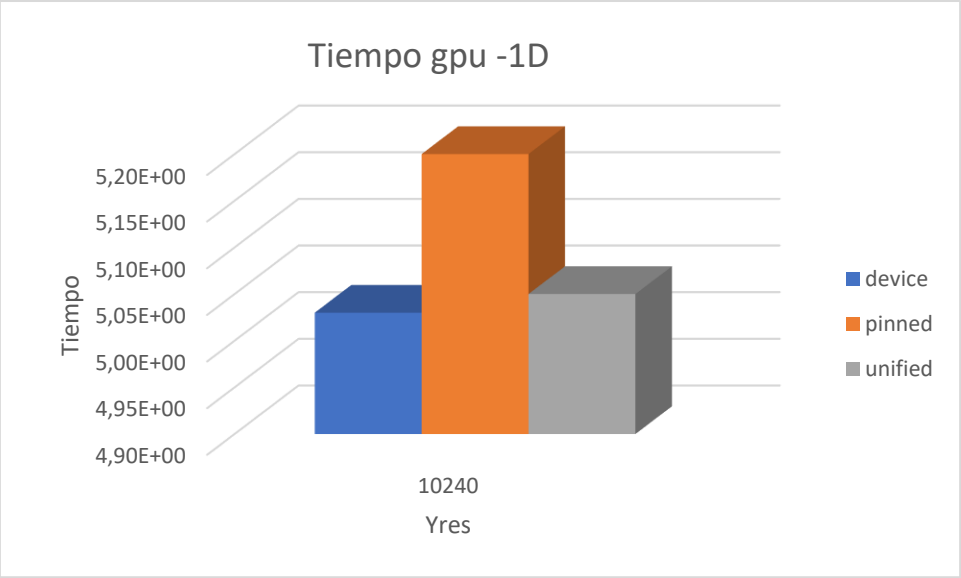
Con esta gráfica nos damos cuenta de que el problema no está limitado, así que, en otros posibles problemas, con mayor cantidad de datos, tendrá un mayor valor en la organización de su memoria.

Por eso vamos a profundizar más, y centrarnos en un yres concreta, por ejemplo 1024, 4096 y 10240.



En esta gráfica vemos que la memoria unified y pinned son similares.

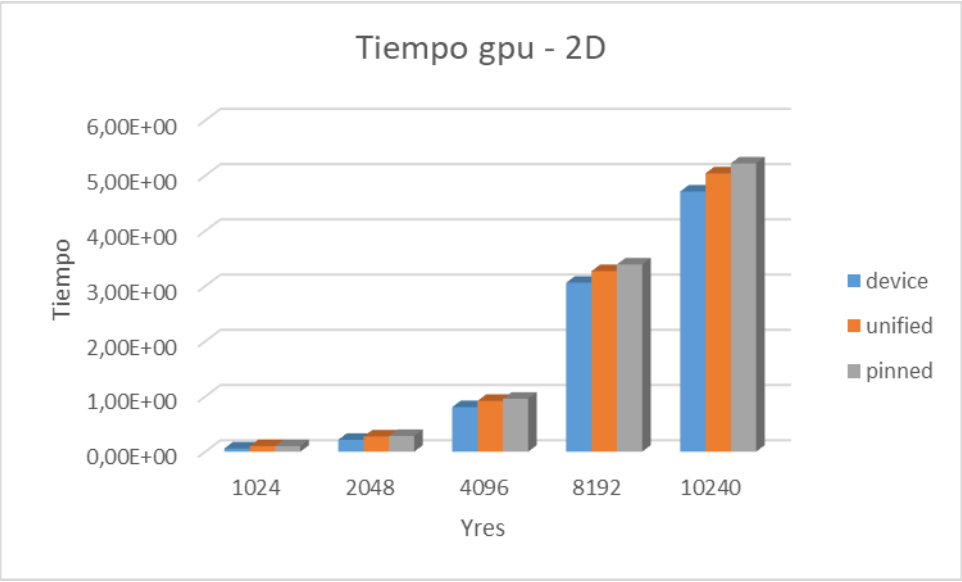
Pero en esta otra gráfica los resultados muestran que la memoria device es mejor.

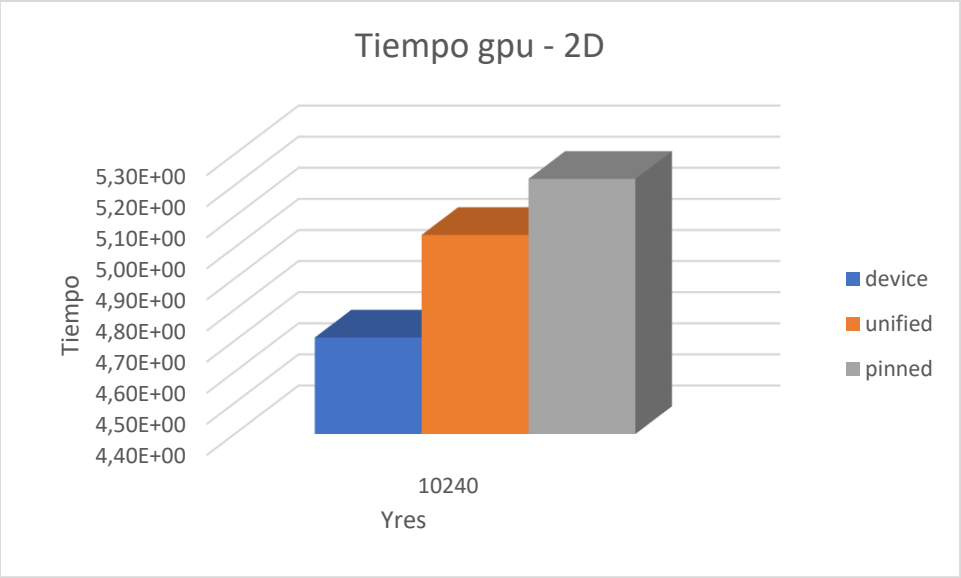


Aquí vemos que la mejor de todas es la device.

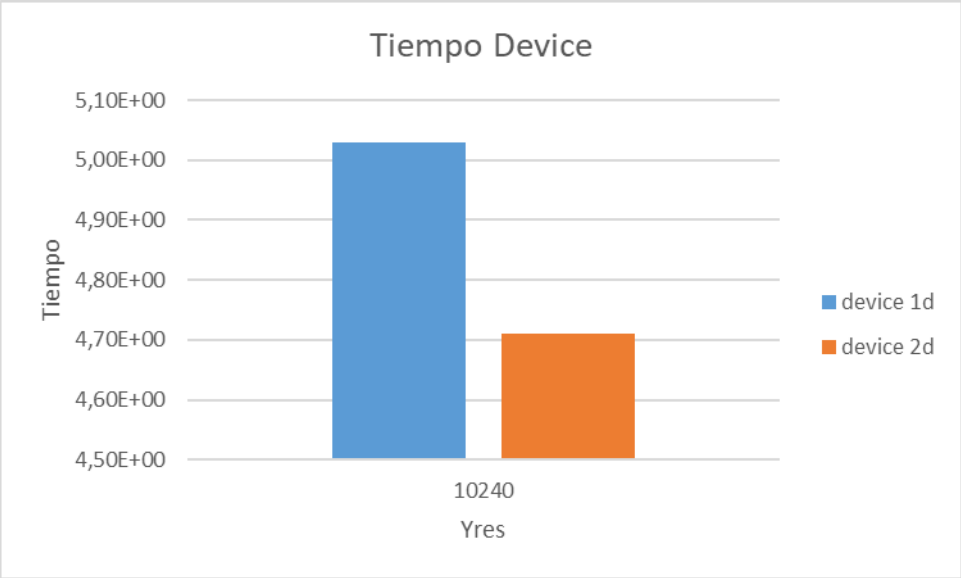
Análisis de GPU con diferentes tipos de memoria (2D)

Al ver que en 1D, la memoria device era la que mejor rendimiento mostraba , y en 2D, sucede lo mismo, es la device la que resulta ser mejor, y luego la unficada y la pinned las peores, siendo sobre todo la pinned la peor.





Una buena comparación sería entre 1D y 2D:



Vemos como en 2D, resulta un mejor rendimiento que en 1D

Pero observando los tiempos en la tabla, vemos que por ejemplo la memoria pinned resulta ser mejor en 1D que en 2D, por parte de la memoria unficada, hay algún caso en el que resulta mejor una que otra.

Esta indexación se consigue al organizar la memoria en 1D,2D y 3D, en estructuras que se procesan en la GPU (un vector, por ejemplo). Entre ellas nos debería haber grandes diferencias.

En cuanto a las aceleraciones entre 2D respecto de 1D:

Se obtiene a partir del cociente entre el Tb y el Ta.

	aceleracion
device	1,0268
pinned	1,02561
unified	0,9986

Las aceleraciones son muy pequeñas, puede deberse a que a la hora de compilar las tareas 2D, se transformen como si fuesen 1D, de forma que, a la hora de ejecutarse, el código es el mismo.

TABLAS gpu

	1D			2D			
yres	device	pinned	unified	device	pinned	unified	heterog
1024	1,21E-01	1,02E-01	1,00E-01	5,89E-02	1,01E-01	1,01E-01	1,36E-01
2048	2,76E-01	2,83E-01	2,76E-01	2,16E-01	2,85E-01	2,75E-01	3,08E-01
4096	9,22E-01	9,54E-01	9,24E-01	8,08E-01	9,60E-01	9,21E-01	9,64E-01
8192	3,27E+00	3,39E+00	3,26E+00	3,06E+00	3,39E+00	3,27E+00	3,88E+00
10240	5,03E+00	5,20E+00	5,05E+00	4,71E+00	5,22E+00	5,04E+00	6,76E+00

Si analizamos según las aceleraciones de las ejecuciones en el sistema de la universidad y en el de mi propio sistema sacamos las siguientes conclusiones (utilizando la memoria pinned en 1D):

yres	aceleracion
1024	1,24
2048	1,71
4096	1,79
8192	1,88
10240	1,95

Ambos sistemas son capaces de ejecutar CUDA correctamente, pero si que es verdad que el sistema del centro es superior al de mi propio sistema, puede deberse que el sistema de la universidad sea más moderno.

Conclusiones

Programar de manera paralela en GPU nos permite disponer de recursos magníficos al realizar cálculos.

De hecho, gran número de máquinas disponen de una o varias GPU's, lo que permite sacar el máximo partido al paralelismo.

El paralelismo, nos acercará a gran número de nuevos problemas, y el rendimiento de las GPU's seguirá aumentando, a medida que pase el tiempo.