Prácticas 4 y 6: Programación Concurrente y Paralela

Proyecto Secuencial – Paralelo y CUDA

Isidro Benítez Zapico

Contenido

Introducción

La idea de este trabajo es aplicar los conocimientos de las prácticas anteriores para desarrollar un programa capaz de procesar imágenes con paralelismo tanto en OpenMP como en CUDA. Respecto a la práctica 4, está dedicada al paralelismo en CPU, programando el código en C y utilizando la API OpenMP para ser más amistoso para el programador.

Por otra parte, la práctica 6 está enfocada al paralelismo de GPU con la utilización de CUDA, aunque dependiendo de si la solución al problema es homogénea o heterogénea, se puede dividir el problema para que una parte la resuelva la GPU y otra la CPU.

Aclaraciones

1. El código de la práctica 4 fue entregado en su momento con diversos errores entre la librería LibProf y LibAlumnx que dificultan la toma de tiempos, por ello, estos tiempos han sido obtenidos en base a una versión del código posterior a la entregada con la idea de obtener datos coherentes.

2. Como es lógico, la versión original del código de la práctica 4, es la que será evaluada en cuanto al diseño y programación de los algoritmos. Esta versión no ha sido modificada desde su entrega, se adjunta junto con la memoria, el código original y su compilación además del mejorado.

3. La complejidad temporal y espacial se realizará para el código nuevo de la práctica 4.

4. En la práctica 6 es posible que la eficiencia de las GPU no sea correcta ya que se utiliza el mismo método para calcularla que con las CPU de la práctica 4.

Práctica 4: OpenMP

Complejidad Temporal

La complejidad temporal es el tiempo que tarda el programa en ejecutarse, en este caso como el programa son varias funciones, dependiendo de que función o funciones ejecutemos, su coste temporal variará. Primero de nada vamos a simplificar las variables del código por comodidad:

- dim (BlockSize) = t
- n = x
- DimX = m
- DimY = n

La complejidad temporal secuencial de cada método es la siguiente:

- DCT8x8: (2*t+2)*(2*t +2 * (2*t +2 * (2*t + 2))) = $16t^4+63t^3+96t^2+63t+16 \approx t^4+t^3+t^2+t$

  Con t=8: $\sum_0^8 \sum_0^8 \sum_0^8 \sum_0^8 1 = 4096$

- IDCT8x8: (2*t+2)*(2*t +2 * (2*t +2 * (2*t + 2))) = $16t^4+63t^3+96t^2+63t+16 \approx t^4+t^3+t^2+t$

  Con t=8: $\sum_0^8 \sum_0^8 \sum_0^8 \sum_0^8 1 = 4096$

- TriggerDCT: ((2*m+2)/t)*((2*n+2)/t) ≈ (4*m*n)/t2 ≈ (DimX*DimY) / BlockSize

  Con TriggerDCT + DCT8x8: $\sum_0^{DimX} \sum_0^{DimY} \sum_0^8 \sum_0^8 \sum_0^8 \sum_0^8 1 = \frac{DimX}{8} (\frac{DimY}{8} (4096))$

- TriggerIDCT: ((2*m+2)/t)*((2*n+2)/t) ≈ (4*m*n)/t2 ≈ (DimX*DimY) / BlockSize

  Con TriggerIDCT + IDCT8x8: $\sum_0^{DimX} \sum_0^{DimY} \sum_0^8 \sum_0^8 \sum_0^8 \sum_0^8 1 = \frac{DimX}{8} (\frac{DimY}{8} (4096))$

- FPA: ((2*n+2)/t)*((2*m+2)/t)*(2*x+2) ≈ (8*n*m*x)/t2 ≈ n*(DimX*DimY) / BlockSize

  Con FPA + FPAB: $\sum_0^{DimY/8} \sum_0^{DimX/8} \sum_0^x 1 = x * \frac{(DimY/8 * DimX/8)}{8}$

- FPB: ((2*n+2)/t)*((2*m+2)/t)*(2*size+2) ≈ (8*n*m*size)/t2 ≈ n*(DimX*DimY) / BlockSize

  Con FPA + FPAB: $\sum_0^{DimY/8} \sum_0^{DimX/8} \sum_0^x 1 = size * \frac{(DimY/8 * DimX/8)}{8}$

- FPAB: (2*x+2) ≈ x, $\sum_0^x 1 = x$

- FPBB: (2*size+2) ≈ size, $\sum_0^{size} 1 = size$

La complejidad temporal en paralelo de cada método es la siguiente:

- DCT8x8: (2*t+2)*(2*t +2 * (2*t +2 * (2*t + 2))) = $16t^4+63t^3+96t^2+63t+16 \approx t^4+t^3+t^2+t$

  Con t=8: $\sum_0^8 \sum_0^8 \sum_0^8 \sum_0^8 1 = 4096$

- IDCT8x8: (2*t+2)*(2*t +2 * (2*t +2 * (2*t + 2))) = $16t^4+63t^3+96t^2+63t+16 \approx t^4+t^3+t^2+t$

  Con t=8: $\sum_0^8 \sum_0^8 \sum_0^8 \sum_0^8 1/p = 4096/p$

- TriggerDCT: ((2*m+2)/t)*((2*n+2)/t) ≈ (4*m*n)/t2 ≈ (DimX*DimY) / BlockSize

  Con TriggerDCT + DCT8x8: $\sum_0^{DimX} \sum_0^{DimY} \sum_0^8 \sum_0^8 \sum_0^8 \sum_0^8 1 = \frac{DimX}{8}\left(\frac{DimY}{8}(4096)\right) * (\frac{1}{p})$

- TriggerIDCT: ((2*m+2)/t)*((2*n+2)/t) ≈ (4*m*n)/t2 ≈ (DimX*DimY) / BlockSize

  Con TriggerIDCT + IDCT8x8: $\sum_0^{DimX} \sum_0^{DimY} \sum_0^8 \sum_0^8 \sum_0^8 \sum_0^8 1 = \frac{DimX}{8p}(\frac{DimY}{8p}(4096) * (\frac{1}{p}))$

- FPA: ((2*n+2)/t)*((2*m+2)/t)*(2*x+2) ≈ (8*n*m*x)/t2 ≈ n*(DimX*DimY) / BlockSize

  Con FPA + FPAB: $\sum_0^{DimY/8} \sum_0^{DimX/8} \sum_0^x 1 = x * \frac{(DimY/8 * DimX/8)}{8}$

- FPB: ((2*n+2)/t)*((2*m+2)/t)*(2*size+2) ≈ (8*n*m*size)/t2 ≈ n*(DimX*DimY) / BlockSize

  Con FPA + FPAB: $\sum_0^{DimY/8} \sum_0^{DimX/8} \sum_0^x 1/\mathrm{p} = size/p * \frac{(DimY/8 * DimX/8)}{8p}$

- FPAB: (2*x+2) ≈ x, $\sum_0^x 1/\mathrm{p} = x/\mathrm{p}$

- FPBB: (2*size+2) ≈ size, $\sum_0^{size} 1/\mathrm{p} = size/p$

Complejidad Espacial

La complejidad espacial no varía independientemente de si el algoritmo es secuencial o paralelo, dicho esto para calcularla hay que tener en cuenta el nº de estructuras usadas. En el primer caso, las funciones TriggerDCT/IDCT y sus auxiliares DCT8x8/IDCT8x8 usan dos matrices, ORIG y DST además de usar DimX y DimY para el tamaño de las imágenes. Su complejidad temporal es:

2*DimX*DimY

En el segundo caso, las funciones FPA y FPB con sus auxiliares FPAB y FPBB utilizan una única matriz Im por lo que su complejidad temporal sería:

DimX*DimY

*No vamos a tener en cuenta variables auxiliares o de control dado que son despreciables para el coste espacial.

TPPdp y Tiempo por Flop teórico mínimo

Ahora vamos a calcular el TPPdp de los diferentes nodos de cálculo:

- i3-2100: 1*2*3.1*8 = 49.6 TPPdp (Gflop)
- Xeon CPU E5- 2603: 2*6*1.7*16 = 326.4 TPPdp (Gflop)
- Ryzen 7 3700X: 1*8*3.6*16 = 460.8 TPPdp (Gflop)

Gracias a estos datos podemos obtener los tiempos por flop teóricos mínimos:

$$Tc = 1/TPPdp$$

- I3 Secuencial = 1 / (49.6 * 10^9) / 2 = 4.0322E-11
- I3 Paralelo = 1 / (49.6 * 10^9) = 2.0161E-11
- Xeon Secuencial = 1 / (326.4 * 10^9) / 12 = 3.6764E-11
- Xeon Paralelo = 1 / (326.4 * 10^9) = 3.0637E-12
- Ryzen Secuencial = 1 / (460.8 * 10^9) / 8 = 1.7361E-11
- Ryzen Paralelo = 1 / (460.8 * 10^9) = 2.17013E-12

Speedup y Eficiencias teóricas

Si dividimos Secuencial entre Paralelo obtendremos los speedup teóricos:

- I3 = 4.0322E-11 / 2.0161E-11 = 2
- Xeon = 3,6764E-11 / 3.0637E-12 = 12
- Ryzen = 1.7361E-11 / 2.17013E-12 = 8

Con los speedup podemos calcular la eficiencia teórica aplicando la siguiente formula:

$$Eficiencia = SpeedUp/cores$$

- Intel I3 = (2 / 2) = 1
- Xeon = (12 / 12) = 1
- Ryzen = (8 / 8) = 1

La eficiencia teórica es 1 es decir el caso ideal, la experimental deber ser ≤ 1.

Tiempos Teóricos

Utilizando El tiempo por Flop teórico mínimo y multiplicándolo por la complejidad temporal (Incorporando los datos como DimX, DimY, y demás para la imagen 8K) es decir los Flops, obtenemos los tiempos teóricos para cada método, variando en función del procesador y si esta paralelizado o no.

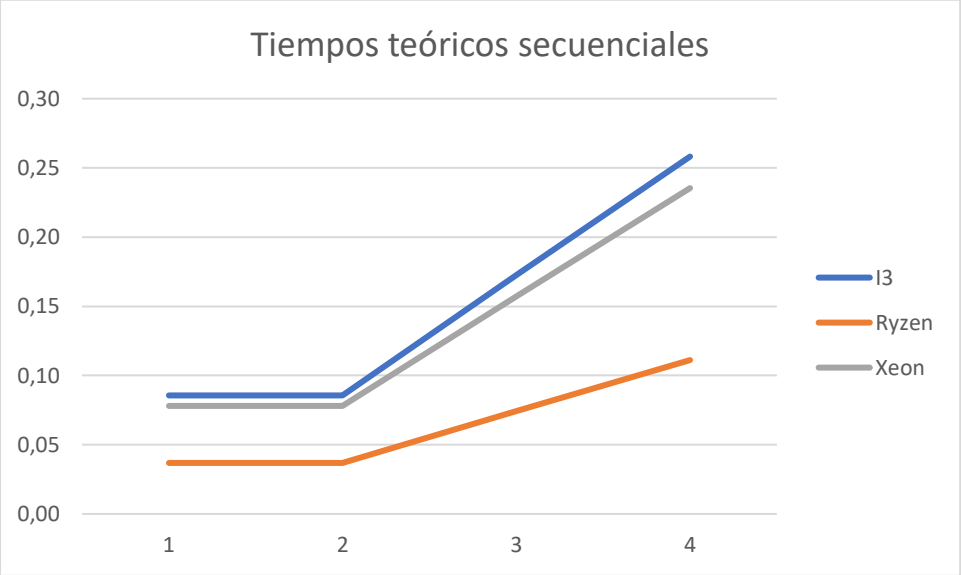
I3	Tc	Flops	Teóricos
DCT	4,03E-11	2123366400,00	0,09
IDCT	4,03E-11	2123366400,00	0,09
DCT+FPB(60)+IDCT	4,03E-11	4277836800,00	0,17
DCT+FPA(4)+IDCT	4,03E-11	6403276800,00	0,26
DCT	2,02E-11	2123366400,00	0,04
IDCT	2,02E-11	265420800,00	0,01
DCT+FPB(60)+IDCT	2,02E-11	2392675200,00	0,05
DCT+FPA(4)+IDCT	2,02E-11	2389046400,00	0,05

Ryzen	Tc	Flops	Teóricos
DCT	1,74E-11	2123366400,00	0,037
IDCT	1,74E-11	2123366400,00	0,037
DCT+FPB(60)+IDCT	1,74E-11	4277836800,00	0,074
DCT+FPA(4)+IDCT	1,74E-11	6403276800,00	0,111
DCT	2,17E-12	2123366400,00	0,005
IDCT	2,17E-12	4147200,00	0,000
DCT+FPB(60)+IDCT	2,17E-12	2131401600,00	0,005
DCT+FPA(4)+IDCT	2,17E-12	2127517650,00	0,005

Xeon	Tc	Flops	Teóricos
DCT	3,68E-11	2123366400,00	0,078
IDCT	3,68E-11	2123366400,00	0,078
DCT+FPB(60)+IDCT	3,68E-11	4277836800,00	0,157
DCT+FPA(4)+IDCT	3,68E-11	6403276800,00	0,235
DCT	3,06E-12	2123366400,00	0,007
IDCT	3,06E-12	9830400,00	0,000
DCT+FPB(60)+IDCT	3,06E-12	2133340800,00	0,007
DCT+FPA(4)+IDCT	3,06E-12	2133206400,00	0,007

Se puede observar claramente como los mejores tiempos son los paralelos y además el Ryzen, es el más rápido ya que tiene el mayor número de cores.

Aquí se ven las evoluciones de los tiempos teóricos secuenciales para las diferentes CPU:



Y aquí en paralelo donde la función DCT como tal no está paralelizada, pero si TriggerDCT, mientras que IDCT si está paralelizada, de ahí la bajada de tiempos. El resto dependen de DCT así que su coste será mayor al de DCT.

Tiempos Empíricos

Se han tomado tiempos en secuencial y paralelo para los diferentes métodos y procesadores con una imagen 8K proporcionada.

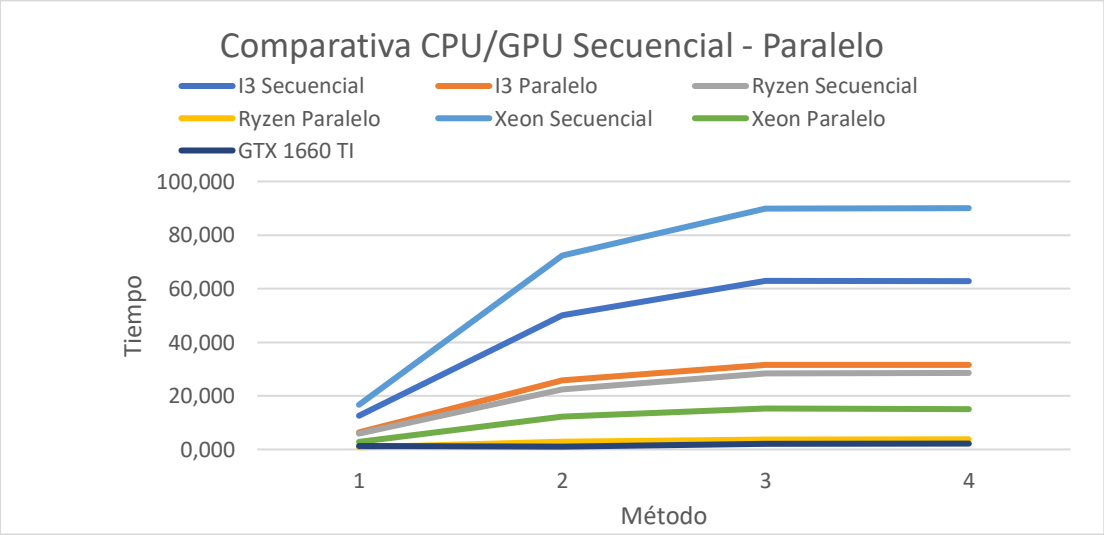
	I3 Sec	I3 Par	Ryzen Sec	Ryzen Par	Xeon Sec	Xeon Par
DCT	12,563	6,365	5,914	0,781	16,659	2,812
IDCT	49,960	25,841	22,296	2,948	72,326	12,159
DCT+FPB(60)+IDCT	62,892	31,551	28,452	3,765	89,900	15,274
DCT+FPA(4)+IDCT	62,864	31,458	28,554	3,767	90,036	15,095
SpeedUp promedio	1,974754031		7,5687700		5,930755049	
Eficiencia Promedio	0,987377015		0,9460962		0,988459175	

Una vez tomados los tiempos se ha calculado el speedup dividiendo cada tiempo secuencial entre el paralelo, y la eficiencia de multiplicar el número de cores por el speedup.

SpeedUp i3	Eficiencia	SpeedUp Ryzen	Eficiencia	SpeedUp Xeon	Eficiencia
1,97390583	0,986952915	7,5762614	0,94703268	5,924279496	0,987379916
1,933384158	0,966692079	7,5622827	0,945285339	5,948236726	0,991372788
1,993337834	0,996668917	7,5566717	0,944583956	5,885992078	0,98099868
1,998388302	0,999194151	7,5798642	0,947483024	5,964511898	0,994085316

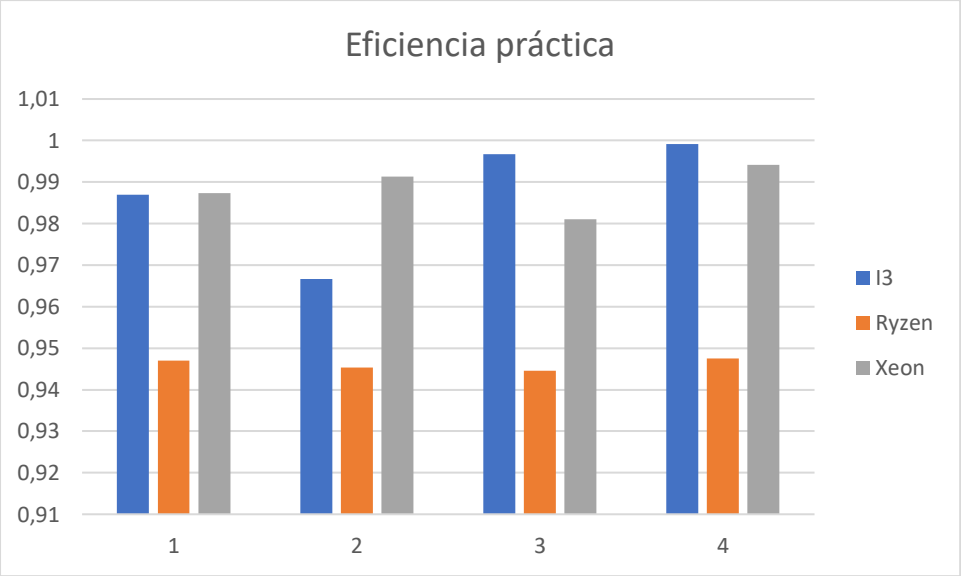
Obtenemos el SpeedUp promedio y Eficiencia Promedio de cada procesador donde el speedUp se acerca al teórico, pero sin superarlo, por ejemplo, el I3 es **1,97 ≤ 2.**

Respecto a la eficiencia es similar, la empírica no puede superar la teórica, por ejemplo, el Ryzen es **0,94 ≤ 1.**



En esta gráfica podemos ver el tiempo empírico de cada método y procesador tanto en secuencial como en paralelo.

Otra comparativa interesante es la eficiencia en un gráfico de barras, donde el I3 es uno de los más eficientes, perdiendo únicamente en la función IDCT contra el Xeon y quedando empatado en la DCT. También es interesante ver como el Ryzen es el menos eficiente de todos sin importar el método.



Mecanismos utilizados en el código de OpenMP

#pragma omp parallel for collapse(2) shared(ORIG, DST)

Destaca esta directiva de OpenMP en el código ya que todas las demás son similares, sirve para paralelizar los bucles for en TriggerDCT y TriggerIDCT, añadiendo collapse a la directiva, causa que dos bucles se comporten como uno solo, de forma que es más eficiente a la hora de ser paralelizado y shared hace que todos los hilos compartan el valor de ORIG y DST.

#pragma omp parallel for collapse(2) private(x, y, u, v, tmp) shared(DST, ORIG)

También es interesante comentar del método IDCT8x8 que es importante privatizar las variables que iteran los bucles o modifican varios hilos para evitar condiciones de carrera e inconsistencias de datos, si no se hiciese el programa no funcionaría correctamente. Se sobrescribirían datos correctos, por ejemplo, la variable tmp que va almacenando los resultados se remplazaría constantemente antes de poder almacenarse correctamente en la imagen destino, guardando siempre el ultimo valor de tmp.

Conclusiones de la Práctica 4

Gracias a las bondades del paralelismo hemos podido observar la diferencia de rendimiento entre un algoritmo secuencial y uno paralelo, dividiendo el problema en bloques.

No solo eso, sino que dependiendo del procesador y su número de cores se puede determinar no solo a nivel práctico sino teórico cual ofrecerá un mayor rendimiento. En el caso de esta práctica el procesador Ryzen en paralelo ha sido el más rápido mientras que el I3, como nos decían tiempos teóricos y finalmente empíricos, ha resultado ser el peor en cuanto a tiempos se refiere.

Estos tiempos teóricos nos permiten confirmar las observaciones empíricas al probar el código en cada una de las CPU, además de ayudarnos a detectar errores, un método con menor complejidad temporal y tc, debería tardar menos tiempo que otro cuyo valor sea mayor, y si empíricamente no es así, la implementación muy probablemente este mal.

OpenMP nos permite paralelizar código de una forma más amistosa para el programador, en un lenguaje familiar como C. Es importante destacar que procesadores como el I3-2100 ya no son el estándar de hoy en día, con dos núcleos difícilmente un ordenador puede ser útil actualmente. Los procesadores modernos de gama media comienzan sobre los ocho núcleos y los sobrepasan hasta llegar a gamas empresariales como los Threadripper.

En conclusión, aunque el paralelismo siempre ha sido interesante históricamente para buscar soluciones más rápidas y eficientes, con el hardware actual sus implicaciones son mucho mayores al disponer de muchos más cores y sockets a precios económicos.

Por ello es importante comprender las ventajas e inconvenientes a la hora de diseñar software y optimizarlo, sin obviar que no todos los problemas son escalables o ideales para el paralelismo.

Práctica 6: CUDA

Número de bloques y tamaño de bloques: la resolución de la imagen es 7680x4320, si cada bloque es de 8x8 píxeles, el número de bloques en el eje X sería 7680/8=960 y en el eje Y sería 4320/8=540

Por lo tanto, el número total de bloques sería 960×540.

Número de hilos por bloque: dado que se utiliza un tamaño de bloque de 8x8, el número total de hilos por bloque sería de 8×8=64 hilos por bloque.

Número total de hilos: este sería el número de bloques y el número de hilos por bloque. Es decir, 960×540×64 = 33.177.600 hilos.

Uso de la memoria: dependiendo de la solución, el comportamiento varía:

- Solución normal: Se utilizan las matrices ORIG y DST donde estas son almacenadas en la memoria global de la GPU, en esta solución se utiliza cudaMemcpy para la transferencia de datos.
- Solución Shared (Solo DCT): Cambia respecto a la normal en que se utiliza memoria compartida para almacenar los bloques compartidos por hilos de este.
- Solución Pinned: Utiliza memoria pinned o “anclada” con el método cudaHostAlloc para mejorar las transferencias de datos entre host y device, se usan punteros para acceder directamente desde device.
- Solución Unified: Se basa en memoria unificada llamando al método cudaMallocManaged, permite el acceso desde host y device, la memoria es gestionada por el sistema.

Coalescencia: En el código cada hilo accede a datos contiguos en la memoria global.

Mecanismo de uso de Shared: Se utiliza el método __syncthreads(); para sincronizar los hilos de cada bloque y así comprobar que todos los datos compartidos están preparados antes de usarse.

Otros: Dependiendo de si la memoria ha sido reservada en GPU o CPU se utiliza el método cudaFree (GPU) o cudaFreeHost(CPU).

Datos y comparativas y aceleraciones

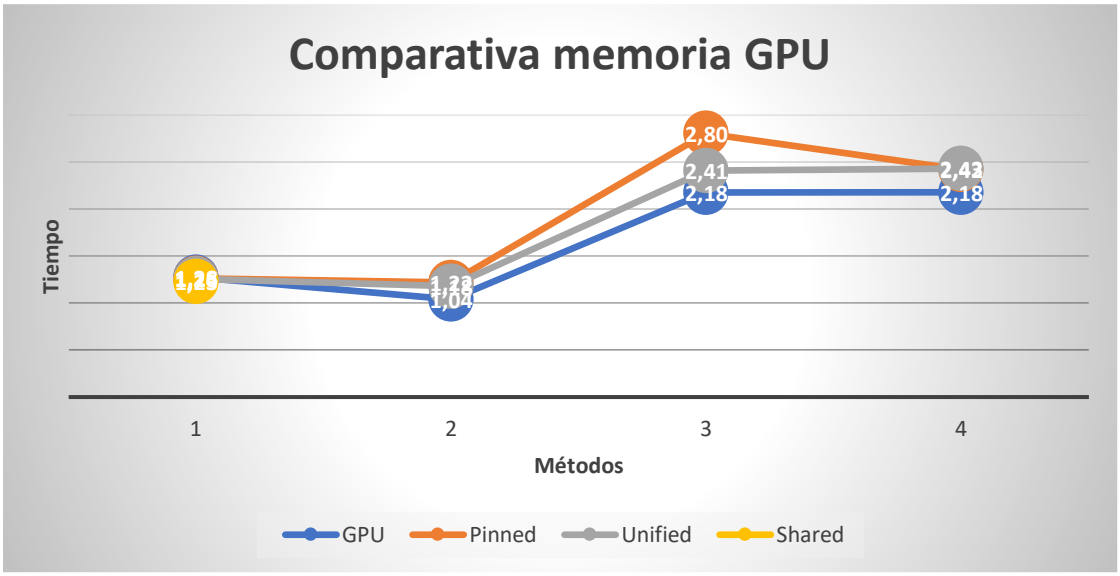
En primer lugar, tenemos una tabla con los datos registrados para las diferentes soluciones de memoria de GPU:

	GPU	GPU - pinned	GPU - unified	GPU - shared
DCT	1,28	1,26	1,26	1,23
IDCT	1,04	1,22	1,18	
DCT+FPB(60)+IDCT	2,18	2,80	2,41	
DCT+FPA(4)+IDCT	2,18	2,42	2,43	
SpeedUp promedio	1,73	1,48	1,56	-
Eficiencia Promedio	0,0011	0,0010	0,0010	-

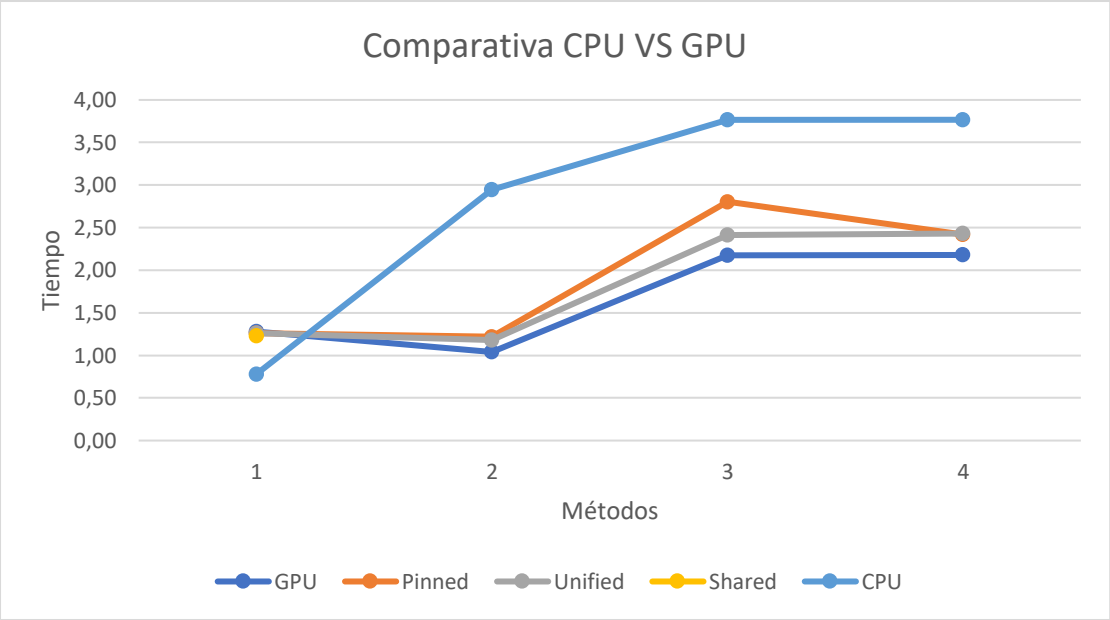
Y aquí tenemos el calculo de SpeedUps y eficiencias respecto al Ryzen Paralelo:

SpeedUp GPU	Eficiencia GPU	SpeedUp GPU - pinned	Eficiencia GPU - pinned	SpeedUp GPU - unified	Eficiencia GPU - unified
0,611506525	0,000398116	0,618448802	0,000402636	0,620217694	0,000403788
2,834883028	0,001845627	2,41689552	0,0015735	2,498051211	0,001626335
1,728970546	0,001125632	1,342534088	0,000874046	1,561042655	0,001016304
1,728593455	0,001125386	1,555456456	0,001012667	1,549659176	0,001008893

En el gráfico podemos observar como Shared es el que tiene un menor tiempo para DCT, mientras que, para el resto, la solución de memoria global es la más rápida. Pinned y Unified empatan en todos los métodos, salvo en DCT+FPB(60)+IDCT.



Si comparamos en una gráfica los tiempos paralelos del Ryzen con los tiempos de la GPU podemos observar una gran diferencia de rendimiento, en el método DCT, la CPU es curiosamente más rápida, pero en el resto la GPU se muestra mucho más eficiente.



Para comparar de una forma mejor el rendimiento entre CPU y GPU he calculado la aceleración de cada uno, con la siguiente formula, $Aceleración = \frac{TiempoRyzenParalelo}{GPU}$

Aceleración	GPU	Pinned	Unified	Shared
DCT	0,611506525	0,618448802	0,620217694	0,635511576
IDCT	2,834883028	2,41689552	2,498051211	0,00
DCT+FPB(60)+IDCT	1,728970546	1,342534088	1,561042655	0,00
DCT+FPA(4)+IDCT	1,728593455	1,555456456	1,549659176	0,00
Promedio Aceleración	1,725988388	1,483333717	1,557242684	0,635511576

Una vez calculados estos datos podemos concluir que:

- El modelo de memoria global es 1,72 veces más rápido que el Ryzen en Paralelo.
- El modelo de memoria pinned es 1,48 veces más rápido que el Ryzen en Paralelo.
- El modelo de memoria unified es 1,55 veces más rápido que el Ryzen en Paralelo.
- El modelo shared es 0,63 veces más lento que la CPU (Aunque solo tenemos una medición de DCT, el único método implementado, por lo que no podemos fiarnos de si implementar shared en todos los métodos haría el programa más lento).

Conclusiones de la Práctica 6

Tras estudiar el paralelismo de CPU en la práctica 4 y ver como la utilización de varios cores reduce significativamente el tiempo de procesamiento, al aplicarlo a las GPU explica porque hoy en día se utilizan cientos de GPU junto con las CPU en centros de procesamiento de datos por todo el mundo. Es cierto que las GPU por si solas no son perfectas, ni vienen a remplazar las CPU, pero su función es vital en muchos campos de la informática como investigación, simulaciones, inteligencia artificial o renderización de gráficos.