Informe de las prácticas 4 y 6

Juan Francisco Mier Montoto (UO283319), PCP 22-23, EPI Gijón

Índice

Índice de contenidos

Índice de figuras

Bibliografía

- D. Cláusula schedule | Microsoft Learn
- OpenMP: improve reduction techniques (Example)
- Conjunto de Mandelbrot - Wikipedia, la enciclopedia libre
- Plotting algorithms for the Mandelbrot set - Wikipedia
- Anillo Único | Tolkienpedia
- OpenMP: For & Scheduling
- OpenMP Scheduling
- #pragma omp atomic - IBM documentation
- Techniques for reducing and bounding OpenMP dynamic memory
- Unified Memory for CUDA Beginners | NVIDIA Technical Blog
- Streaming Multiprocessor o SM: qué son y su historia en NVIDIA
- CUDA Refresher: The CUDA Programming Model | NVIDIA Technical Blog
- CUDA C++ Programming Guide (atomicAdd)
- Maximizing Unified Memory Performance in CUDA | NVIDIA Technical Blog
- would the block dimension affect performance? | NVIDIA Forums

Sitios de interés

Los siguientes enlaces contienen información no evaluada ni presentada, pero pueden ser interesantes y han sido muy relevantes para el desarrollo de este informe:

- Repositorio de la práctica
- Hoja de cálculo de la práctica

Introducción

La lucha constante con la Ley de Moore supone mejoras tecnológicas constantes para paliar la evolución de la dificultad de los problemas. En este caso, nos aprovechamos de las dos evoluciones más importantes en este ámbito: los procesadores multinúcleos y las GPU.

0.1. Enfoque y objetivo

El desarrollo de esta práctica se centra en analizar el funcionamiento del problema a resolver, el cálculo de un fractal del conjunto *Mandelbrot*, su promedio y binarización de manera paralela, primero en CPU gracias a OpenMP y posteriormente en GPU, dispositivos más potentes que permiten el cálculo acelerado y paralelizado de problemas muy complejos.

Puesto que el enfoque de la práctica está en OpenMP y CUDA y en el funcionamiento de su set de características, otros estudios ya realizados referentes al funcionamiento teórico del paralelismo, comparaciones teóricas/empíricas, etc se dejan ligeramente de lado. A lo largo de este informe se realizará otro tipo de análisis: comparación entre alternativas de implementaciones, primero intentando razonar los resultados que se deberían obtener desde un punto de vista teórico y luego comparando los resultados empíricos.

0.2. Notas sobre el procedimiento

- En todos los análisis se han utilizado las máquinas de la cola GPU para tomar las mediciones excepto cuando se indica lo contrario.
- La mayor parte de los datos son una combinación de resultados obtenidos de la máquina procedente con el objetivo de asegurar la fiabilidad de los datos. Dependiendo de la comparativa, se han tomado una, tres o cinco muestras.
- Para la práctica 4: todos los trabajos cuentan con el máximo de iteraciones igual a 1000, el valor indicado por defecto en el enunciado. No se ha experimentado con este u otros valores.
- Para la práctica 6: durante la obtención de resultados, se mandan trabajos pequeños a la GPU previos a las mediciones finales para obtener resultados más fiables. De otra manera, el orden de ejecución puede llegar a influir en la determinación de la mejor opción en cada caso[1].
- Para la práctica 6: todos los trabajos cuentan con 32 hilos por bloque, ya que el número de hilos debería ser siempre múltiplo de 32 para alinearse con los warps pero no puede ser mayor de 32 cuando se utilizan métodos 2D porque 32 * 32 = 1024 que es la configuración límite en el sistema en el que se realizan las pruebas.
- Se supone más importante las diferencias de tiempos para las tallas más grandes porque, por lo general, suponen una mayor diferencia. Por este motivo, se “ponderan” o valoran más que diferencias pequeñas en tamaños de problema inferiores.
- En ambas prácticas no se han realizado implementaciones extra ni se han estudiado las funciones de binarizado, ya que tienen menos juego que las otras dos.

[1] Esto seguramente tenga algo que ver con algún tipo de caché.

Práctica 4

4.1. Desarrollo Python

Esta es la parte más sencilla de esta práctica: el desarrollo en Python, haciendo uso de cualquier librería y estrategia, de la resolución de fractales.

Python cuenta con el método `complex()` que permite el uso de números complejos sin necesidad de utilizar otras funciones ni la sintaxis típica que se encontrará en las resoluciones en C y CUDA, lo que facilita mucho el código.

El resto de funciones (media y binarizado) son triviales, así como otras funciones como el grabado de imágenes a disco o el cálculo del error entre dos fractales.

4.2. Cálculos teóricos

4.2.1. Complejidad temporal

Secuencial

Para obtener la complejidad temporal, se analiza el funcionamiento del código.

Primero se analiza la función *mandel_iter()*, que es el algoritmo de tiempo de escape que calcula el valor de cada pixel. Este es el bucle principal de dicha función:

```
...
while (k < maxiter && u2 + v2 < 4.0) {          // 1 flop
    v = 2.0 * u * v + y;                        // 1 flop
    u = u2 - v2 + x;                            // 1 flop
    u2 = u * u;                                 // 1 flop
    v2 = v * v;                                 // 1 flop
    k++;
}
return k == maxiter ? 0 : k; // 1 flop
```

Suponiendo el peor caso, esta función tiene una complejidad aproximada de 5 flops por la cantidad de veces que se recorra el bucle en el peor de los casos, es decir, tantas veces como el valor de *maxiter*:

$$t_{mandel\_iter}(k) = (5k + 1) \cdot t_c$$

En la función principal, el código que se encarga de recorrer toda la matriz es el siguiente:

```
for (i = 0; i < xres; i++) {
      c_r = xmin + i * dx;                    // 1 flop
      for (j = 0; j < yres; j++) {
            c_im = ymin + j * dy;             // 1 flop
            k = mandel_iter(c_r, c_im, maxiter);
            A[i + j * xres] = k;              // 1 flop
      }
}
```

Contando con la complejidad temporal de la función *mandel_iter()* obtenida anteriormente, la complejidad temporal aproximada del problema es:

$$t_{mandel}(n,\ k)\ = n(1\ +\ n(2\ +\ t_{mandel\_iter}(k)))\ \cdot\ t_c$$
$$t_{mandel}(n,\ k)\ = n(1\ +\ n(2\ +\ (1\ +\ 5k)))\ \cdot\ t_c$$
$$t_{mandel}(n,\ k)\ = n(1\ +\ n(3\ +\ 5k)))\ \cdot\ t_c$$
$$O_{mandel}\ \in n^2\ \cdot\ k\ \approx\ n^2$$

Suponiendo que n = xres = yres = size. Siendo k = maxiter, una constante.

Paralelo

Puesto que se divide todo el problema utilizando paralelismo, la complejidad del problema se divide directamente entre p, de forma que:

$$t_{mandel\_parallel}(n,\ k,\ p)\ =\ \frac{n}{p}(1\ +\ n(3\ +\ 5k)))\ \cdot\ t_c$$

Tasks (paralelo)

En tasks no se divide todo el problema, sino que un hilo se encarga de lanzar las tareas que sí que se ejecutan de forma paralela:

$$t_{mandel\_tasks}\ (n, k,\ p)\ = n(1\ +\ \frac{n}{p}(3\ +\ 5k)))\ \cdot\ t_c$$

4.2.2. Complejidad espacial

La complejidad del problema sí que es igual en todas las ejecuciones y variaciones del problema. Como indicado en la revisión de la práctica anterior, se entrega en bytes.

$$e_{mandel}(xres,\ yres)\ =\ xres\ *\ yres\ *\ sizeof(double)\ =\ xres\ *\ yres\ *\ 8\ bytes$$

4.2.3. Rendimiento teórico y tiempos por flop

Ya se ha realizado un análisis teórico del rendimiento de las colas de la asignatura en la práctica anterior, por lo que se resume en este apartado. Toda esta información se obtiene de las propias máquinas de las colas:

$$TPP_i = chasis \cdot nodos_{chasis} \cdot sockets_{nodo} \cdot cores_{socket} \cdot clock_{GHz} \cdot \frac{nº\ flop}{ciclo}$$

$$t_{c_i} = \frac{chasis \cdot nodos_{chasis} \cdot sockets_{nodo} \cdot cores_{socket}}{TPP_i} = \frac{1}{clock_{GHz} \cdot \frac{nº flop}{ciclo}}$$

Máquina	TPP (Gflops)	t_c (s)
ColaI3 (i3-2100)	49.6	$4.0323e-11$
ColaXeon (Xeon E5620)	153.6	$5.2083e-11$
ColaGPU (Ryzen 7 3700x)[2]	460.8	$1.7361e-12$

Tabla 1. Rendimientos teóricos y tiempos por flop de las máquinas de las colas

4.3. Implementaciones escogidas

A la hora de realizar el código que se pedía, se ha optado por realizar varias “versiones” de las funciones base (mandel, promedio y binariza) para analizar las diferentes alternativas y su rendimiento. Las implementaciones realizadas son las siguientes:

4.3.1. mandel

- La versión “estándar”, paralelizada con *#pragma parallel for* de manera normal.
- La versión *tasks*, paralelizada con el pragma *tasks*.
- La versión *collapse*, muy similar a la versión estándar pero con cambios mínimos para permitir el “colapso” de los bucles que recorren la imagen en uno solo.
- Múltiples versiones con scheduling, similares a la versión estándar pero que estudian el funcionamiento y el rendimiento de la directiva *schedule* y sus opciones.

4.3.2. promedio

- La versión “estándar”, que hace uso del pragma *reduction* propio de OpenMP.
- La versión “schedule”, igual que la anterior pero que hace uso de las mejoras conseguidas con el análisis de la anterior función con respecto al uso de scheduling.

[2] Igual que en la práctica 3, teniendo en cuenta el boost de frecuencia implementado en el procesador.

- Las versiones *critical* y *atomic*, que como sus propios nombres indican hacen uso de los respectivos pragma y reemplazan al reduction, permitiendo una reducción “manual”.
- La versión de vectorización, que intenta eliminar la sobrecarga de las dos versiones anteriores haciendo uso de estructuras auxiliares.

4.4. Resultados obtenidos

Se rellenan las tablas indicadas por el enunciado según los datos obtenidos:

t_c	4.03E-11	Colal3									
core	2				Tiempos empíricos						
	Análisis teórico					C paralelo					
	Tiempos teóricos		Ratio teórico/empírico			mandelAlumnx					
size	Secuencial	Paralelo	Secuencial	Paralelo	Python	C secuencial	mandelProf	parallel for	collapse	tasks	schedule (guided)
256	1.32E-02	6.61E-03	0.11	0.09	3.39E+00	1.18E-01	7.72E-02	9.73E-02	7.83E-02	1.24E-01	8.75E-02
512	5.29E-02	2.64E-02	0.11	0.09	1.37E+01	4.71E-01	2.92E-01	2.86E-01	2.73E-01	3.29E-01	2.69E-01
1024	2.12E-01	1.06E-01	0.11	0.09	5.46E+01	1.88E+00	1.16E+00	1.13E+00	1.09E+00	1.10E+00	9.62E-01
2048	8.46E-01	4.23E-01	0.11	0.09	2.18E+02	7.53E+00	4.52E+00	4.39E+00	4.37E+00	4.35E+00	3.78E+00
4096	3.38E+00	1.69E+00	0.11	0.09	N/A	3.02E+01	1.79E+01	1.74E+01	1.75E+01	1.84E+01	1.51E+01
8192	1.35E+01	6.77E+00	0.11	0.09	N/A	1.21E+02	7.16E+01	6.97E+01	6.99E+01	6.93E+01	6.02E+01

Tabla 2. Resultados generales de la práctica 4 sobre la cola i3

t_c	5.21E-11	ColaXeon									
core	8				Tiempos empíricos						
	Análisis teórico					C paralelo					
	Tiempos teóricos		Ratio teórico/empírico			mandelAlumnx					
size	Secuencial	Paralelo	Secuencial	Paralelo	Python	C secuencial	mandelProf	parallel for	collapse	tasks	schedule (guided)
256	1.71E-02	2.13E-03	0.10	0.01	4.74E+00	1.75E-01	2.56E-01	4.56E-02	4.18E-02	2.61E-01	7.11E-02
512	6.83E-02	8.54E-03	0.10	0.01	1.91E+01	6.89E-01	1.01E+00	1.70E-01	1.26E-01	9.80E-01	1.35E-01
1024	2.73E-01	3.42E-02	0.10	0.01	7.63E+01	2.75E+00	3.66E+00	5.15E-01	4.54E-01	4.36E+00	3.92E-01
2048	1.09E+00	1.37E-01	0.10	0.01	3.05E+02	1.11E+01	1.90E+01	1.90E+00	1.85E+00	1.72E+01	1.44E+00
4096	4.37E+00	5.46E-01	0.10	0.01	N/A	4.45E+01	6.69E+01	7.36E+00	7.38E+00	7.45E+01	5.67E+00
8192	1.75E+01	2.19E+00	0.10	0.01	N/A	1.78E+02	2.50E+02	2.93E+01	2.95E+01	2.30E+02	2.25E+01

Tabla 3. Resultados generales de la práctica 4 sobre la cola Xeon

t_c	1.74E-11	ColaGPU									
core	8				Tiempos empíricos						
	Análisis teórico						C paralelo				
	Tiempos teóricos		Ratio teórico/empírico					mandelAlumnx			
size	Secuencial	Paralelo	Secuencial	Paralelo	Python	C secuencial	mandelProf	parallel for	collapse	tasks	schedule (guided)
256	5.69E-03	7.12E-04	0.06	0.01	2.27E+00	8.83E-02	8.66E-02	2.59E-02	2.98E-02	1.49E-01	3.88E-02
512	2.28E-02	2.85E-03	0.06	0.01	8.92E+00	3.53E-01	2.93E-01	1.02E-01	9.80E-02	3.15E-01	5.80E-02
1024	9.11E-02	1.14E-02	0.06	0.01	3.55E+01	1.41E+00	1.07E+00	2.34E-01	2.36E-01	1.07E+00	1.91E-01
2048	3.64E-01	4.55E-02	0.06	0.01	1.43E+02	5.69E+00	4.29E+00	9.43E-01	9.46E-01	4.27E+00	7.56E-01
4096	1.46E+00	1.82E-01	0.06	0.01	N/A	2.31E+01	1.77E+01	3.85E+00	3.83E+00	1.73E+01	3.02E+00
8192	5.83E+00	7.29E-01	0.06	0.01	N/A	9.42E+01	6.87E+01	1.56E+01	1.57E+01	6.82E+01	1.22E+01

Tabla 4. Resultados generales de la práctica 4 sobre la cola GPU

Puesto que los ratios teórico/empíricos salen (sorprendentemente) iguales, las complejidades temporales y, por lo tanto, los tiempos teóricos están bien calculados, pese a que sean muy inferiores.

No hay mucho más que destacar de estas tablas: los análisis en profundidad entre implementaciones se realizan en el siguiente apartado. Los tiempos de la librería del profesor son inferiores a la versión estándar del alumno. Puesto que coinciden en tiempos, seguramente utilice la estrategia *tasks*. Python es mucho más lento que el resto de funciones.

4.5. Comparativas

4.5.1. Scheduling

Planteamiento

Uno de los principales análisis experimentales durante la práctica ha sido con el uso de *scheduling* con la directiva *schedule(type)* en OpenMP. Como indicado en el enunciado de la práctica, el uso de esta técnica hace que los tiempos de ejecución varíen, principalmente dependiendo del tamaño del problema.

Utilizando un rango de valores para la talla desde 256 hasta 8192, se analizan los tiempos obtenidos al usar las siguientes opciones de scheduling:

- *static*
- *guided*
- *auto*
- *dynamic*

Resultados

Se obtienen los siguientes resultados realizando la media de cinco análisis:

	mandelAlumnx				
		schedule			
size	normal	auto	static	guided	dynamic
256	2.8136E-02	3.0021E-02	4.2871E-02	3.7681E-02	3.4456E-02
512	9.0269E-02	5.9520E-02	9.7914E-02	8.2593E-02	7.8770E-02
1024	2.6615E-01	2.3417E-01	2.7155E-01	2.1389E-01	2.1778E-01
2048	9.8562E-01	9.4349E-01	9.6872E-01	7.5217E-01	7.6257E-01
4096	3.8167E+00	3.8099E+00	3.8118E+00	2.9539E+00	2.9876E+00
8192	1.5516E+01	1.5475E+01	1.5475E+01	1.1989E+01	1.1965E+01

Tabla 5. Comparación entre estrategias de scheduling

A primera vista, se pueden hacer algunas observaciones:

- *static* funciona igual o peor que no utilizar scheduling.
- *auto* solo mejora el rendimiento para tallas pequeñas, por lo que no es fiable a la hora de obtener el mejor rendimiento de manera constante.
- *guided* y *dynamic* son las mejores opciones para este problema.

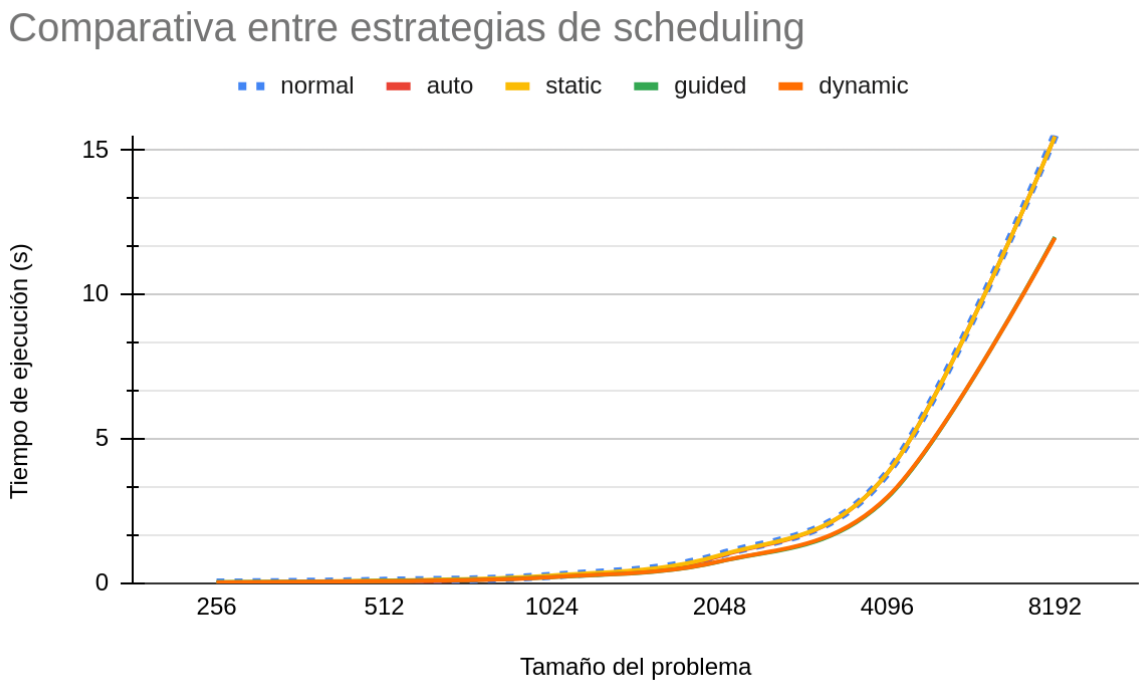


Gráfico 1. Comparación entre estrategias de scheduling

Ya que no hay gran diferencia entre estrategias, si nos centramos en valores más pequeños se obtiene el siguiente gráfico:

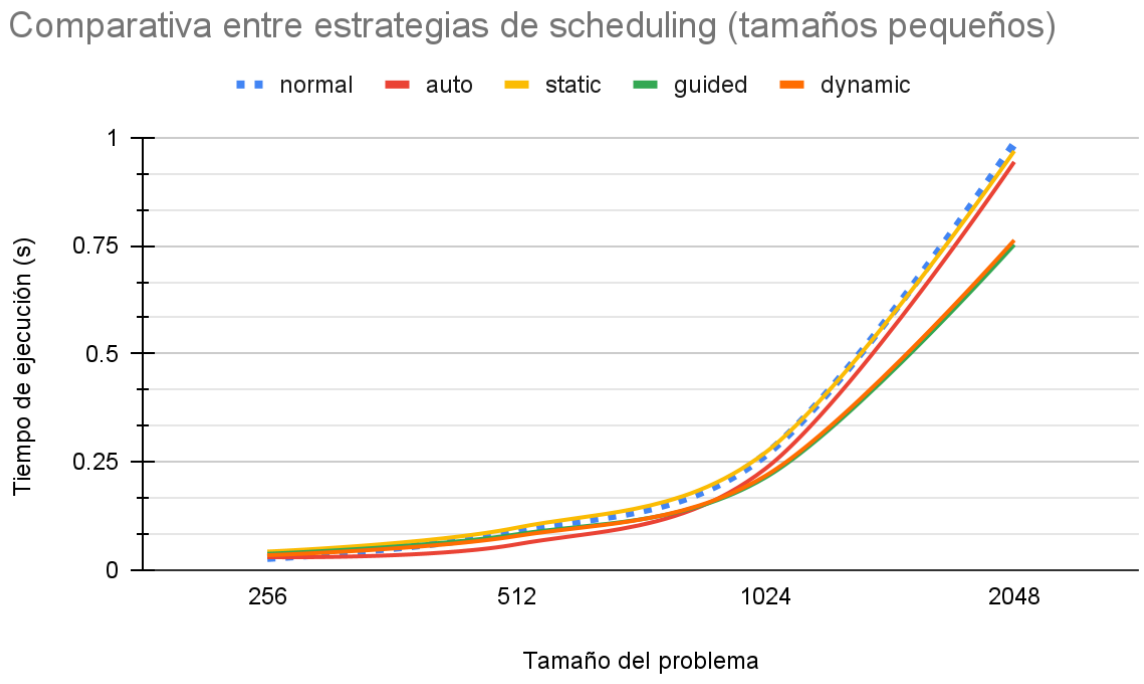


Gráfico 2. Comparación con tallas pequeñas entre estrategias de scheduling

Se comparan los valores con respecto a la media de cada tamaño para observar cuál es superior.

Diferencias respecto a la media					
AVG	normal	auto	static	guided	dynamic
3.4633E-02	-6.4971E-03	-4.6118E-03	8.2379E-03	3.0483E-03	-1.7733E-04
8.1813E-02	8.4558E-03	-2.2293E-02	1.6101E-02	7.7936E-04	-3.0431E-03
2.4071E-01	2.5439E-02	-6.5385E-03	3.0840E-02	-2.6817E-02	-2.2924E-02
8.8251E-01	1.0311E-01	6.0974E-02	8.6206E-02	-1.3035E-01	-1.1994E-01
3.4760E+00	3.4068E-01	3.3391E-01	3.3586E-01	-5.2208E-01	-4.8838E-01
1.4084E+01	1.4323E+00	1.3907E+00	1.3915E+00	-2.0953E+00	-2.1192E+00
AVG	3.1724E-01	2.9202E-01	3.1146E-01	-4.6179E-01	-4.5894E-01

Tabla 6. Diferencia con el promedio de tiempos según estrategia de scheduling

Explicación

Puesto que el tamaño del problema en cada iteración no es constante, *guided* y *dynamic* eran las mejores opciones a priori, sobre todo para valores grandes como se puede observar. *static* no tiene sentido por el mismo motivo, el tamaño del problema no es constante en cada iteración. *auto* es un poco decepcionante, ya que debería escoger la estrategia correcta en tiempo de compilación o ejecución, pero no lo consigue de manera constante, sólo para tallas del problema pequeñas.

Los valores de *chunk_size* altos no tienen sentido porque se está asignando mucha carga computacional a un mismo hilo. Esto es justo lo contrario de lo que se desea intentar, repartir la carga de la mejor manera de forma que todo termine lo antes posible y todos los hilos realicen una cantidad similar de trabajo.

La diferencia entre *dynamic* y *guided* es que este último reparte una cantidad de iteraciones proporcional a la cantidad de trabajo restante, por lo que según la ejecución va avanzando la carga de cada hilo es menor.

scheduling es un arma muy potente en cuanto a mejora de rendimiento, siempre y cuando se sepa escoger bien. Al escoger las estrategias que mejor encajan con el problema, se pueden observar importantes mejoras de rendimiento, sobre todo si se desea obtener un fractal con mayor resolución. Se consigue una aceleración global de 1.16 al utilizar *schedule(guided)*, 1.3 en el caso de tallas más grandes.

Aceleración normal/schedule(guided)		
256	0.75	
512	1.09	
1024	1.24	
2048	1.31	
4096	1.29	
8192	1.29	
AVG	1.16	

Tabla 7. Aceleración total de la mejor estrategia

chunk_size

Se analiza el rendimiento dependiendo del *chunk_size* para *schedule(dynamic)* a modo de prueba. Se consiguen los siguientes resultados:

	dynamic chunk_size					
size	1	2	4	8	16	32
256	1.67E-02	2.39E-02	1.67E-02	2.17E-02	3.15E-02	3.15E-02
512	6.97E-02	7.49E-02	6.35E-02	7.56E-02	9.28E-02	9.28E-02
1024	1.97E-01	1.98E-01	1.97E-01	2.02E-01	2.11E-01	2.11E-01
2048	7.50E-01	7.56E-01	7.56E-01	7.46E-01	7.45E-01	7.45E-01
4096	3.01E+00	3.02E+00	2.99E+00	2.98E+00	3.02E+00	3.02E+00
8192	1.20E+01	1.20E+01	1.21E+01	1.21E+01	1.23E+01	1.23E+01

Tabla 8. Comparación entre chunk_size para schedule(dynamic)

Se obtienen las siguientes conclusiones a partir de los resultados:

- Los valores grandes no parecen ser muy óptimos.
- No hay mucha diferencia en general entre valores pequeños.
- La tendencia es a mayor tamaño de bloque, peor rendimiento individual de cada hilo y peor rendimiento global.
- Puesto que “1” es el valor por defecto en todas las directivas y además es uno de los que mejor rendimiento obtiene para cualquier tamaño, se asume en el resto de análisis, incluyendo el análisis anterior.

4.5.2. Promedios

Planteamiento

Puesto que la función que calcula el promedio de la imagen es esencialmente una reducción, hay varias maneras de intentar optimizarlo, incluyendo las vistas en las transparencias de prácticas de aula de la asignatura.

Por ello, se ha implementado la misma función de promedio con diferentes estrategias para analizar los resultados y obtener la forma de reducción óptima.

Se tienen las siguientes opciones:

- Con la directiva *reduction* de OpenMP.
- Haciendo la reducción a mano utilizando *critical*.
- Haciendo la reducción a mano utilizando *atomic*.
- Haciendo la reducción a mano utilizando una estrategia de vectorización.
- Con la directiva *reduction* de OpenMP en combinación con un scheduling estático.
- Con la directiva *reduction* de OpenMP guardando el valor en un entero.
    - Esta opción es deliberadamente mala, es un ejemplo de mal rendimiento para el análisis posterior. No siempre devuelve valores correctos.

Resultados

Tras los análisis se obtienen los siguientes resultados:

	promedioAlumnx					
	reduction			manual		
size	normal	int	schedule	atomic	critical	vectorization
256	6.79E-05	1.06E-04	6.67E-05	6.98E-05	4.76E-05	4.93E-05
512	1.15E-04	2.10E-04	2.08E-04	1.65E-04	1.21E-04	1.20E-04
1024	4.39E-04	8.06E-04	8.38E-04	4.42E-04	4.37E-04	4.53E-04
2048	1.63E-03	2.83E-03	3.69E-03	1.67E-03	1.61E-03	1.63E-03
4096	9.64E-03	1.75E-02	1.38E-02	9.31E-03	9.41E-03	9.34E-03
8192	4.83E-02	8.43E-02	5.37E-02	4.53E-02	4.59E-02	4.62E-02

Tabla 9. Comparación entre versiones de la función promedio

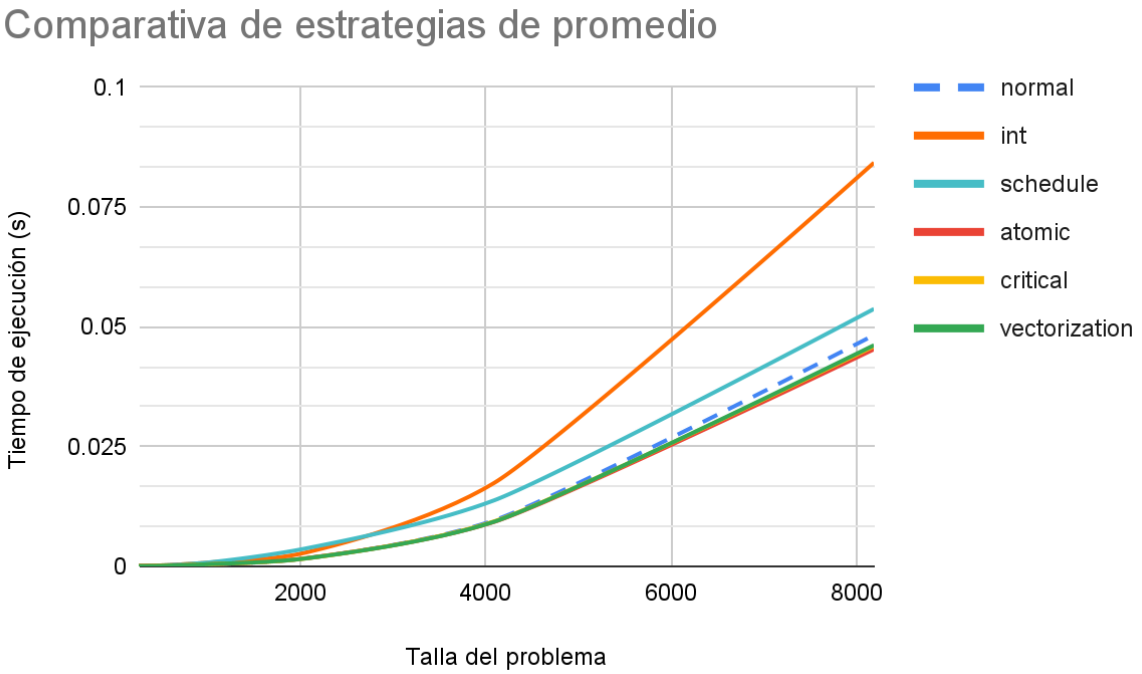


Gráfico 3. Comparación entre versiones de la función promedio

A primera vista, se pueden realizar un par de observaciones:

- La versión con enteros funciona mal, como se esperaba.
- No se ha conseguido una mejoría con *schedule(static, 8)*[3].
- El resto de funciones funcionan de manera muy similar pero mejor que el *reduction* normal.

Se descartan las estrategias que no funcionen y se centra la Gráfico en los puntos que más diferencia enseñan.

[3] El objetivo inicial de esta función se explica posteriormente.

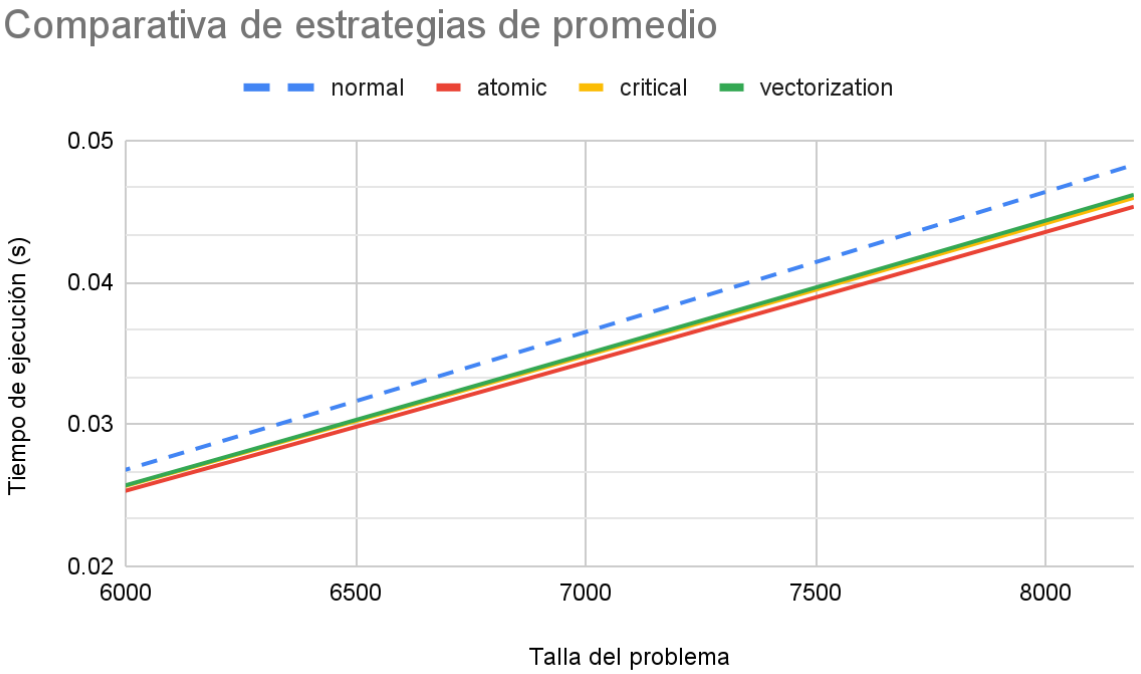


Gráfico 4. Comparación de versiones de la función promedio para tallas seleccionadas

Tras esta Gráfico, podemos sacar en claro lo siguiente:

- *atomic* es mejor que *critical*, como cabría esperar.
- No se ha conseguido mejorar el rendimiento con la estrategia de vectorización.
- Estas mejoras tan solo suponen una mejora diminuta con respecto a la reducción estándar de OpenMP.

Explicación

Con estos resultados, se confirman algunas asumpciones pero se descartan algunas teorías.

El objetivo de utilizar *schedule(static, 8)* en conjunto con la reducción es tratar de evitar el *false sharing* visto en clase introduciendo un padding gracias al número de posiciones contiguas que se le asigna a cada hilo mediante el *static*, pero esto provoca una degradación de rendimiento.

El problema principal con la función a base de enteros son los contínuos casteos de tipos que se generan al convertir entre *int* y *double*. Esta función está aquí a modo de anécdota, ya que es la función original desarrollada para el promedio hasta el desarrollo del resto de alternativas, desconociendo su pobre rendimiento y su nula fiabilidad de cálculo de valores correctos.

El objetivo de utilizar vectorización es eliminar la sobrecarga que introducen los métodos *critical* y *atomic*. Después de analizarlo, el rendimiento es mejor que la reducción original,

pero su complejidad no supone una mejora de rendimiento con respecto a las otras dos estrategias.

El análisis principal se encuentra entre las dos funciones restantes. Era de esperar que ambas funcionaran mejor que la reducción “automática” de OpenMP. También era de esperar que *atomic* funcionara ligeramente mejor que *critical*, ya que la primera utiliza recursos hardware específicos y permite menos operaciones y la segunda es más universal pero tiene un coste de gestión superior.

Notas

Para conseguir los resultados obtenidos, los bucles *for* dentro de cada función utilizan la directiva *nowait*. Además, *atomic* utiliza *update* como indicado en las prácticas de aula, pese a que debería ser el funcionamiento por defecto.

4.5.3. Estrategias de mandel

Planteamiento

El objetivo de esta comparación es analizar el rendimiento de todas las opciones generadas de la función *mandel*, la encargada de calcular el fractal. Puesto que ya se ha realizado el análisis de scheduling, se escoge la mejor opción, es decir *scheduling(guided)* para este análisis. Se da por supuesto que esta versión tiene mejor rendimiento que la función “normal”. Las opciones son las siguientes:

- *normal*, función estándar con un pragma *parallel for* utilizada como base para medir el resto de funciones a lo largo de esta disertación y del desarrollo del código.
- *collapse*, igual que la función base pero colapsando los dos bucles que recorren la matriz gracias al pragma de OpenMP del mismo nombre.
- *schedule*, ya estudiado y analizado.
- *tasks*, versión recurrente de otras prácticas[4], no se espera buen rendimiento por su pobre complejidad temporal y su conocido rendimiento.

[4] En la práctica anterior (1+3) se obtenían peores resultados utilizando *tasks*. Puesto que aquí la complejidad temporal también es peor, se esperan los mismos patrones.

Resultados

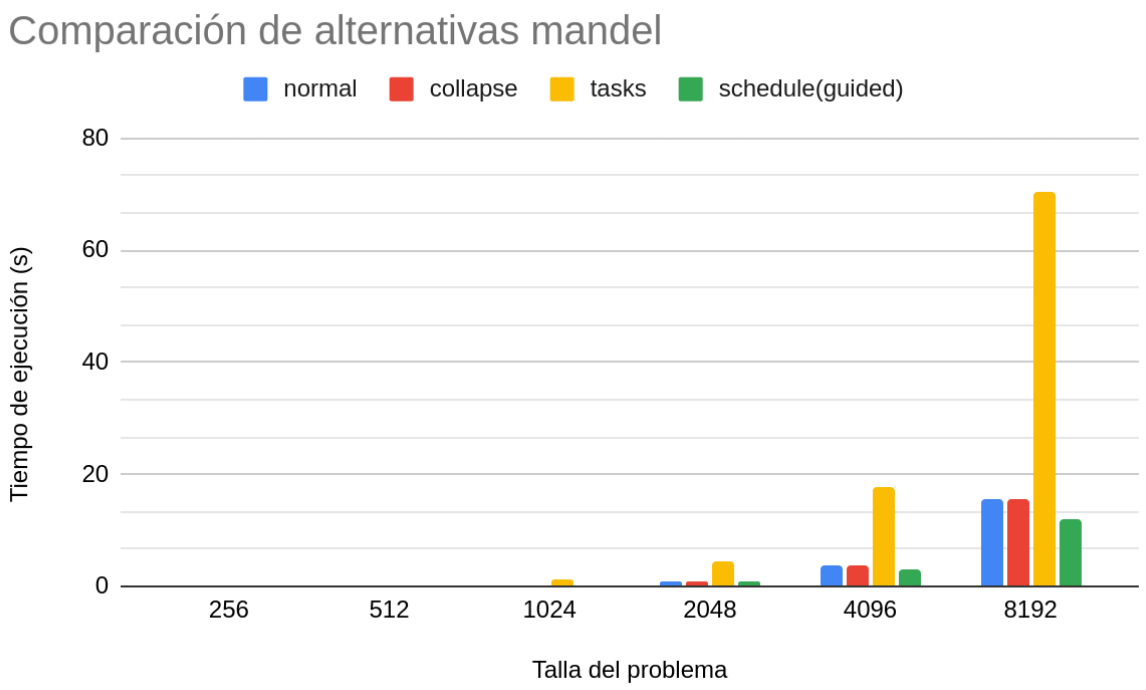


Gráfico 5. Comparación de alternativas mandel (gráfico de barras)

Instantáneamente, podemos observar que *tasks* cumple con las expectativas negativas. Mientras que el resto de opciones tarda menos de 20 segundos en ejecutar el problema más grande, *tasks* tarda alrededor de 70, lo que lo descarta del resto de análisis.

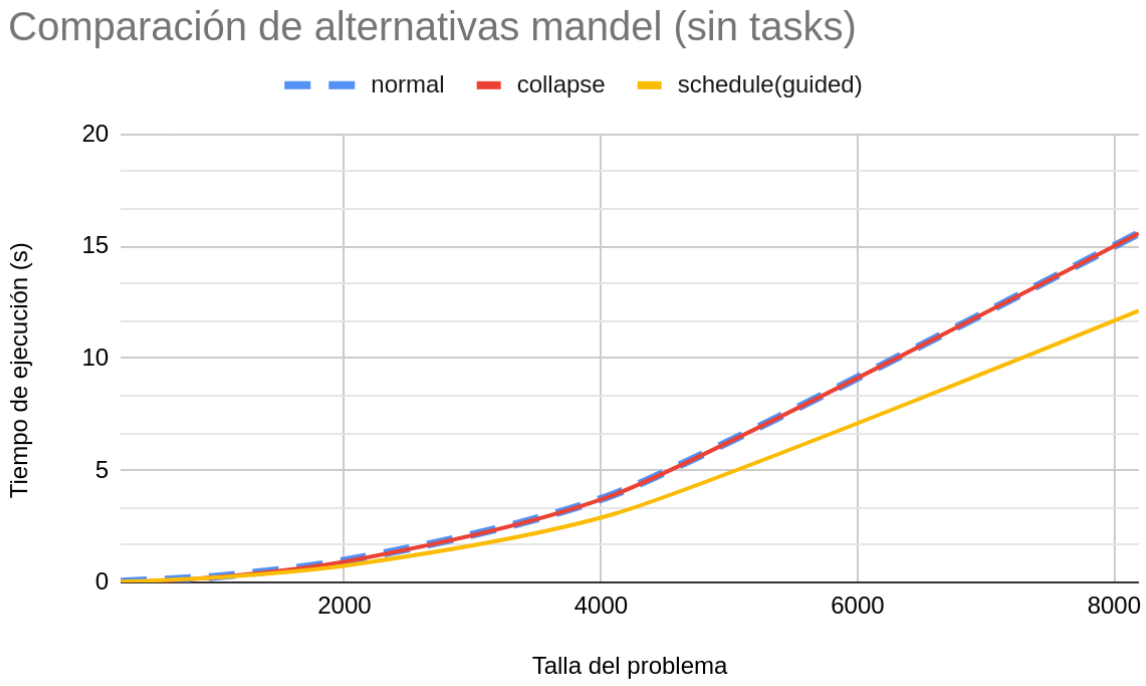


Gráfico 6. Comparación de alternativas mandel (excl. tasks, gráfico de líneas)

Eliminando a *tasks* de la ecuación nos deja con un panorama fácil de asimilar: *collapse* no influye nada en el rendimiento del programa y *schedule(guided)* funciona bien, como esperado.

Explicación

Este análisis cumple con las expectativas teóricas y del resto de análisis ya realizados: *schedule* sí que supone una mejoría de rendimiento, sobre todo al utilizar una opción que concuerde con el problema, *tasks* tiene un rendimiento muy pobre ya que lo único que no se paraleliza el problema entero y *collapse* no supone un mejor rendimiento porque simplemente convierte al problema a una dimensión en lugar de dos, lo que no debería suponer un cambio en el rendimiento.

Práctica 6

6.1. Implementaciones escogidas

Se ha escogido por implementar la mayoría de opciones presentadas en el trabajo y estudiarlas, intentando analizar donde tenga sentido hacerlo y añadiendo algunas opciones extras:

6.1.1. mandelGPU

- La versión estándar en 2D con su respectivo kernel en CUDA.
- Versiones de la función base que utilizan la implementación clásica de memoria[5], *Pinned Memory*[6] y *Unified Memory*.
- Una versión de la función *mandelGPU* en 1D (el resto de funciones que utilizan 2D).
- Una función con heterogeneidad, que significa que la CPU pasa a ejecutar un porcentaje inferior al 50% del cómputo total del problema.

6.1.2. promedioGPU

- La versión estándar utilizando la API de *cublas*.
- Una versión con kernel "manual" que hace uso de la memoria shared para la reducción.
- Una versión con kernel "manual" que hace uso de una estructura de almacenamiento pasada por parámetros que reemplaza a la memoria shared del anterior.
- Una versión con un kernel muy reducido que hace uso únicamente de la instrucción *atomicAdd()*.

6.2. Resultados obtenidos

Se obtienen todos los datos requeridos por el enunciado de la práctica:

		mandelAlumnx				
				memory strategy		dimension
size	mandelProf	normal	heterogéneo	unified	pinned	1D
1024	4.70E-02	4.70E-02	4.48E-02	4.84E-02	4.98E-02	9.75E-02
2048	1.71E-01	1.71E-01	1.35E-01	1.76E-01	1.82E-01	2.84E-01
4096	6.33E-01	6.33E-01	4.82E-01	6.55E-01	6.76E-01	7.36E-01
8192	2.37E+00	2.37E+00	1.81E+00	2.46E+00	2.54E+00	2.70E+00
10240	3.64E+00	3.64E+00	2.76E+00	3.76E+00	3.91E+00	3.94E+00

Tabla 10. Resultados generales de la práctica 6

[5] A veces "classic memory", "handmade memory" o "explicit copy memory".
[6] A veces "zero-access memory" en alguna documentación de NVIDIA.

El análisis exhaustivo de estos datos se realiza más adelante.

6.3. Comparativas

6.3.1. Tipos de memorias (mandel)

Planteamiento

Desde un primer momento, se tiene la opción de escoger el tipo de memoria con el que trabajar en la GPU. Dependiendo del caso, como al intentar *heterogeneidad* con *unified memory*, algunas estructuras tienen más sentido que otras.

En esta comparativa se quiere comparar estrictamente el rendimiento, no la funcionalidad que tenga ni el uso que le pueda dar el programador. Para ello, se evalúa el funcionamiento de la memoria clásica, *pinned memory* y *unified memory* en la función *mandel* estándar en 2D.

Resultados

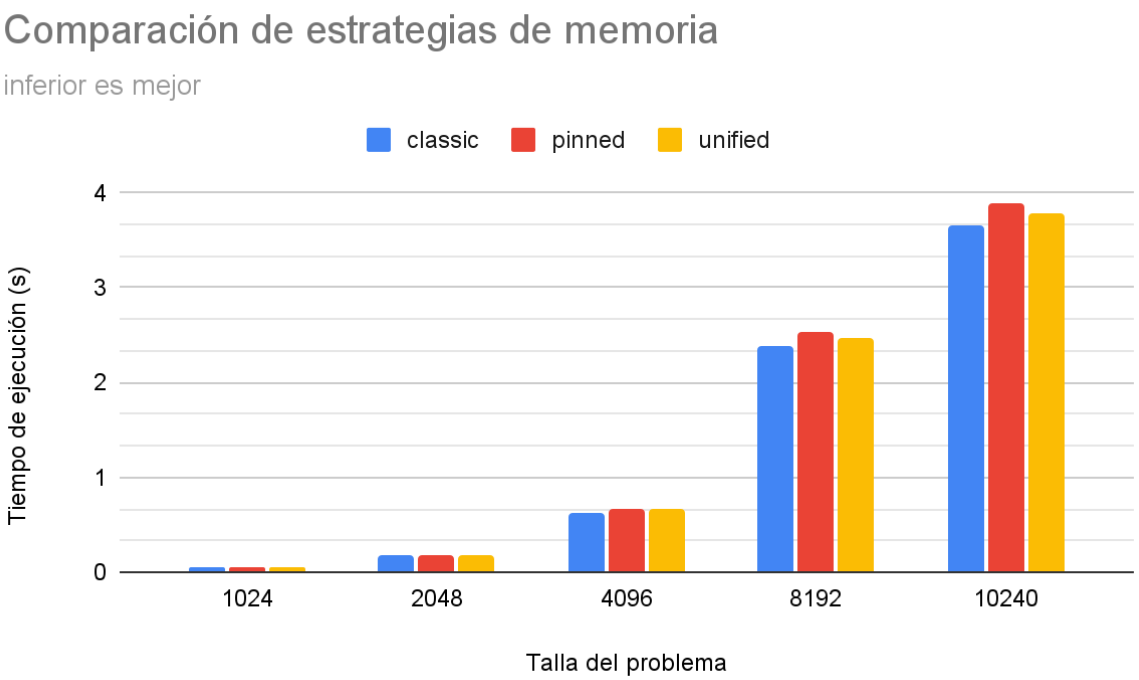


Gráfico 7. Comparación de estrategias de memoria

Sin entrar en mucho detalle, aparentemente tanto la gestión de memoria clásica como *unified* son igual de rápidas, mientras que *Pinned Memory* es más lenta. La Gráfico anterior utiliza los siguientes datos:

	memory strategies			
size	classic	pinned	unified	AVG
1024	4.70E-02	4.95E-02	4.87E-02	4.84E-02
2048	1.71E-01	1.81E-01	1.76E-01	1.76E-01
4096	6.33E-01	6.71E-01	6.57E-01	6.54E-01
8192	2.37E+00	2.53E+00	2.47E+00	2.46E+00
10240	3.65E+00	3.88E+00	3.78E+00	3.77E+00

Tabla 11. Datos obtenidos de la comparación de estrategias de memoria

Ajustando con las medias para cada tamaño, se obtienen los siguientes resultados:

size	classic	pinned	unified
1024	-1.42E-03	1.10E-03	3.14E-04
2048	-5.05E-03	4.97E-03	8.36E-05
4096	-2.06E-02	1.75E-02	3.02E-03
8192	-8.31E-02	7.16E-02	1.15E-02
10240	-1.23E-01	1.11E-01	1.23E-02

Tabla 12. Comparación con las medias entre estrategias de memoria

Explicación

A primera vista, no está claro por qué se han obtenido esos resultados tan claros. Al estudiar el funcionamiento y las características de cada estrategia, se obtienen las siguientes conclusiones:

- *Pinned Memory* se especializa en la rápida transferencia de información. Se almacena en la memoria RAM, por lo que pese a que es accesible tanto por el host (CPU) como por el device (GPU), su uso excesivo puede degradar el rendimiento, tal y como se observa en los resultados obtenidos.
- *Unified Memory* almacena información en alguna memoria y permite a ambos dispositivos ser capaz de acceder y modificar la información. Obviamente no es tan sencilla y rápida como la asignación clásica de memoria, pero es mucho más eficiente que *pinned*, como se puede comprobar. El motivo por el que *unified* funciona tan bien es gracias a la migración de páginas, que combina las ventajas de tener copias de información en ambos dispositivos (classic) y la memoria “zero-copy” (pinned), que depende de la velocidad de las conexiones entre dispositivos.

Por todo esto, se concluye que los resultados obtenidos tienen sentido y cumplen con lo que dice la teoría sobre el funcionamiento de estas estrategias.

6.3.2. Heterogeneidad

Planteamiento

La función “heterogénea” no calcula todo el fractal utilizando el kernel CUDA correspondiente, sino que asigna un porcentaje del cálculo a la CPU, de modo que ambos dispositivos trabajan de manera simultánea con el objetivo de conseguir el mismo trabajo en una cantidad inferior de tiempo. Para mejorar los resultados, se utilizan los conocimientos obtenidos en la anterior práctica y se implanta un pragma de OpenMP *parallel for schedule(guided)* para tratar de conseguir el mejor rendimiento posible.

Para estudiar la eficiencia de esta estrategia, hay que estudiar los valores de tiempo obtenidos dependiendo del porcentaje del problema asignado a cada dispositivo. Puesto que la GPU es más potente que la CPU, debería tener un mayor porcentaje de la carga computacional.

NOTA: esta función no consigue un error nulo al comparar los resultados con el resto de funciones, pero la imagen es visualmente idéntica[7].

Resultados

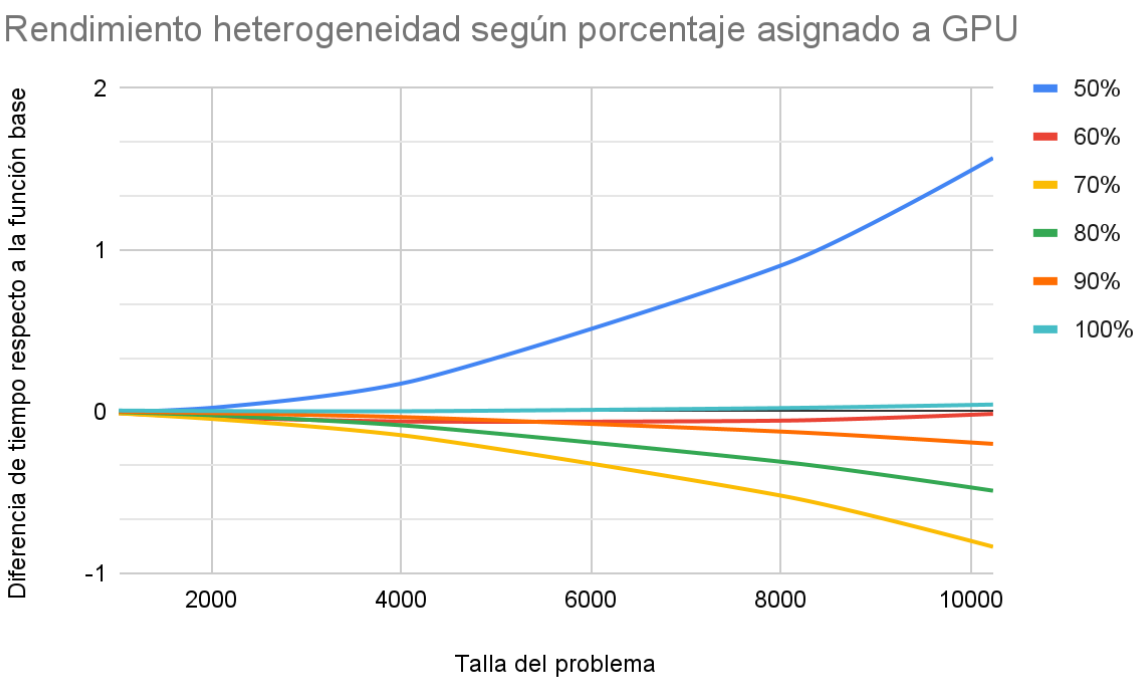


Gráfico 8. Rendimiento de la función heterogénea dependiendo del porcentaje de cálculo de la GPU

Este gráfico muestra bastante bien las ganancias o pérdidas de tiempo dependiendo el porcentaje de cálculo de la GPU que se escoja. Suponiendo que el eje de abscisas es el

[7] Se consigue un error inferior a 5000, la diferencia está en píxeles sueltos. Una hipótesis es la diferencia de precisión entre la CPU y la GPU, puesto que solo se encuentran estos píxeles “muertos” en la parte heterogénea que calcula la CPU.

tiempo de la función “base”, es decir, la función mandel2D sin heterogeneidad, se puede observar lo siguiente:

- Valores inferiores al 60% son iguales o peores que la función base.
- Valores superiores al 60% consiguen una mejoría de rendimiento importante.
- El rendimiento cuando se establece 100% (que la GPU hace todo el trabajo) es igual o ligeramente peor que el de la función base.
- Los valores cercanos al 70% consiguen una mejora de tiempo MUY importante.

Explicación

Gracias a la implementación de la paralelización de la práctica anterior, se pueden conseguir valores que realmente impacten al rendimiento de manera positiva, siendo esta estrategia la que más afecta al rendimiento de todos los análisis realizados para ambas prácticas.

Supuestamente, la GPU tiene mucho más rendimiento que la CPU y no debería tener un porcentaje tan “bajo”, pero las optimizaciones de OpenMP han movido el *sweet spot* del 90% a cerca del 70%, donde se consigue el mejor rendimiento.

Puesto que se utiliza *Unified Memory* para trabajar con ambos dispositivos sobre la misma estructura, es posible que se beneficie más el acceso de la CPU en caso de encontrarse en la RAM del ordenador.

6.3.3. Estrategias de mandel

Planteamiento

En este apartado se analiza más a fondo los datos obtenidos en los resultados generales de la práctica. Se comparan las distintas versiones de mandel realizadas por el alumno, partiendo de lo siguiente:

- Ya se ha estudiado el rendimiento según la estrategia de memoria.
- Ya se ha visto el efecto (positivo) de la heterogeneidad.
- Según los datos vistos en la tabla de resultados generales, 1D es peor que 2D para este problema. No se estudia con 3D porque la complejidad de implementar una función así no supondría un efecto positivo en el rendimiento.

Por estos motivos, el análisis no es tan exhaustivo como en el resto de apartados.

Resultados

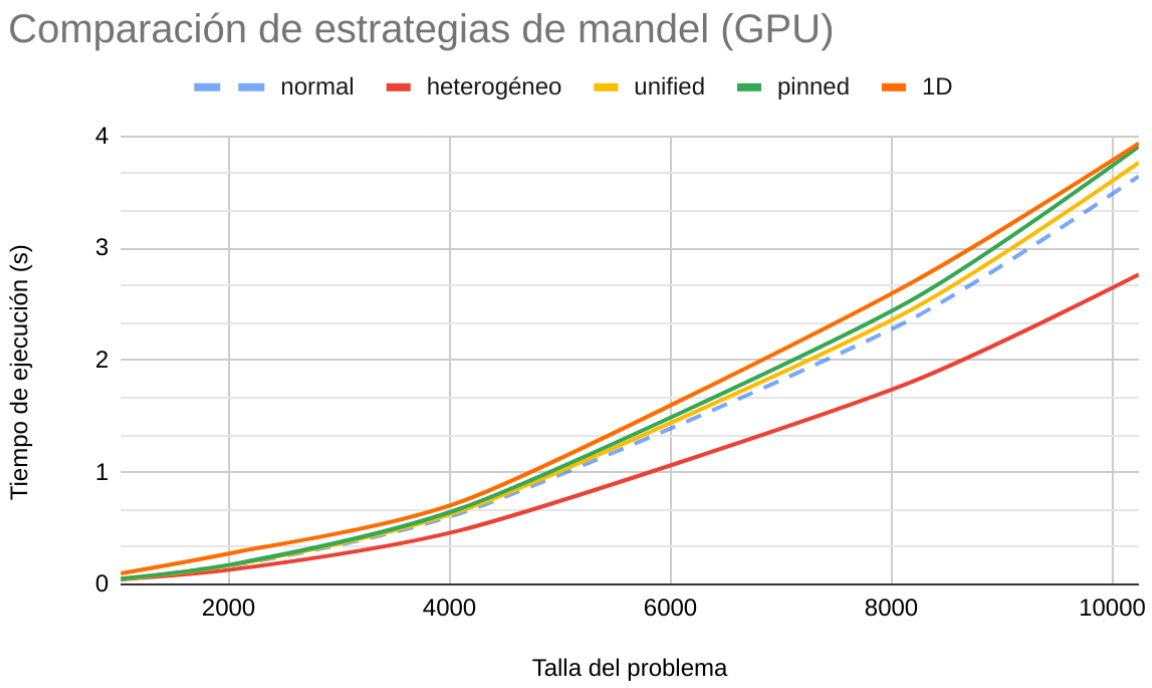


Gráfico 9. Comparación de estrategias general de mandelGPU

Todas las versiones funcionan como esperado, pero ¿por qué la versión unidimensional es tan lenta? Posible respuesta: para todas las ejecuciones se pasa por parámetros el mismo número de bloques, es decir, 32, como especificado al comienzo de este informe. Mientras que el resto de funciones están en dos dimensiones, lo que implica 1024 hilos por bloque, en una sola dimensión son de tan solo 32. La diferencia no debería existir o al menos ser tan grande, puesto que el compilador supuestamente convierte todas las implementaciones a lo mismo en tiempo de compilación.

6.3.4. Estrategias de promedio

Planteamiento

A la hora de realizar el promedio en CUDA, el enunciado plantea varias alternativas. Adicionalmente, durante la práctica se han encontrado algunas alternativas. Las opciones son las siguientes:

- La función *api* realiza el promedio mediante la librería *cublas* en C, que realiza operaciones en CUDA sin necesidad del uso de un kernel, con implementaciones ya realizadas y supuestamente optimizadas.
- La función *shared* es una implementación manual del promedio mediante un kernel CUDA. A la hora de realizar este promedio, se ha utilizado el “Problema X” de las transparencias de PAs de CUDA, que trata una reducción diferente a este problema pero útil. Esta función utiliza dos kernels, uno para realizar la suma de los valores dentro de cada bloque y otro que realiza la suma de los valores de cada bloque del kernel anterior. El uso de dos kernels supone una barrera implícita, es decir, que el

segundo no comienza su ejecución sin antes terminar el primero, perfecto para este problema.

- La función *params* es esencialmente la misma que la versión anterior pero sin el uso de memoria compartida, utilizando en su lugar el paso por parámetros de estructuras para almacenar información. Se espera que sea más lento y es además más ineficiente con el espacio de memoria reservada[8].
- La función *atomic* utiliza un kernel muy sencillo con la función *atomicAdd*. Se utiliza *unsigned long long int* en la variable que acumula el valor para evitar *overflows*[9].

Resultados

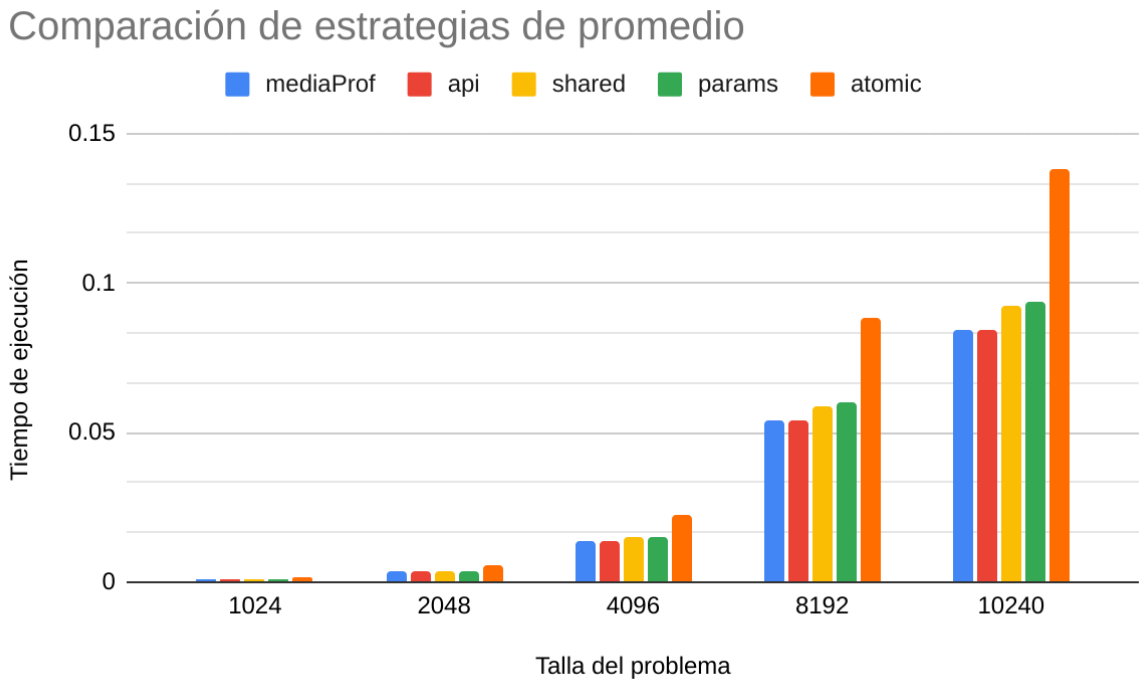


Gráfico 10. Comparación de estrategias de promedioGPU (gráfico de barras)

A primera vista, *atomic* es mucho peor que el resto de alternativas, por lo que se elimina de la ecuación.

[8] Existen ineficiencias ya que se reutiliza el mismo vector "caché" para ambos kernels cuando el segundo necesita mucho menos espacio que el primero. Ya que la tarea es *compute-bound*, no debería suponer un problema.

[9] Pese a que NVIDIA habla sobre el funcionamiento de la función *atomicAdd(double*, double)* en su documentación, la configuración que se utiliza en los equipos o las opciones de compilación no la soportan. Las Gráficos utilizadas (1660Ti) soportan esta característica gracias a su *Compute Capabiltiy* (SM7_1 frente a SM6_X necesario). Esto no supone pérdida de precisión ya que el contenido del vector sobre que calcular contiene enteros pese a ser de tipo *double*.

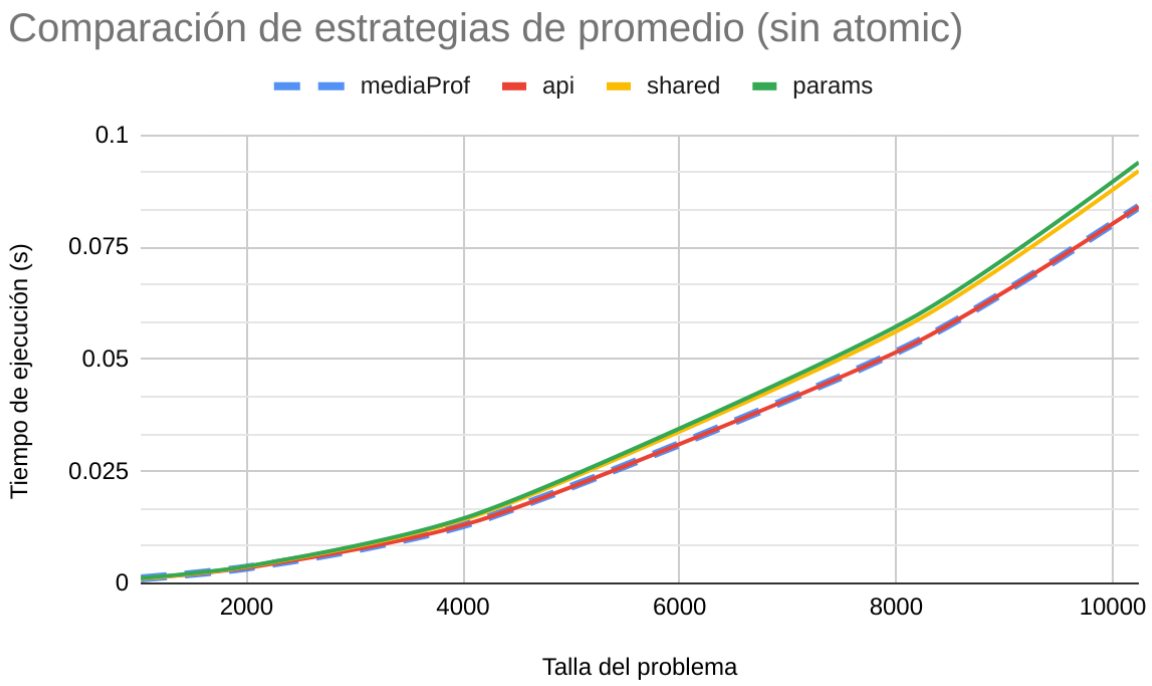


Gráfico 11. Comparación de estrategias de promedioGPU (excl. atomic, gráfico de líneas)

Se desconoce la estrategia implementada en la librería del profesor, pero comparando rendimientos se asume que se utiliza la API de cublas y la función *cublasDasum* o similar.

Explicación

Como en el resto de análisis, todos los resultados tienen sentido y cumplen con lo que cabría esperar desde un punto de vista teórico.

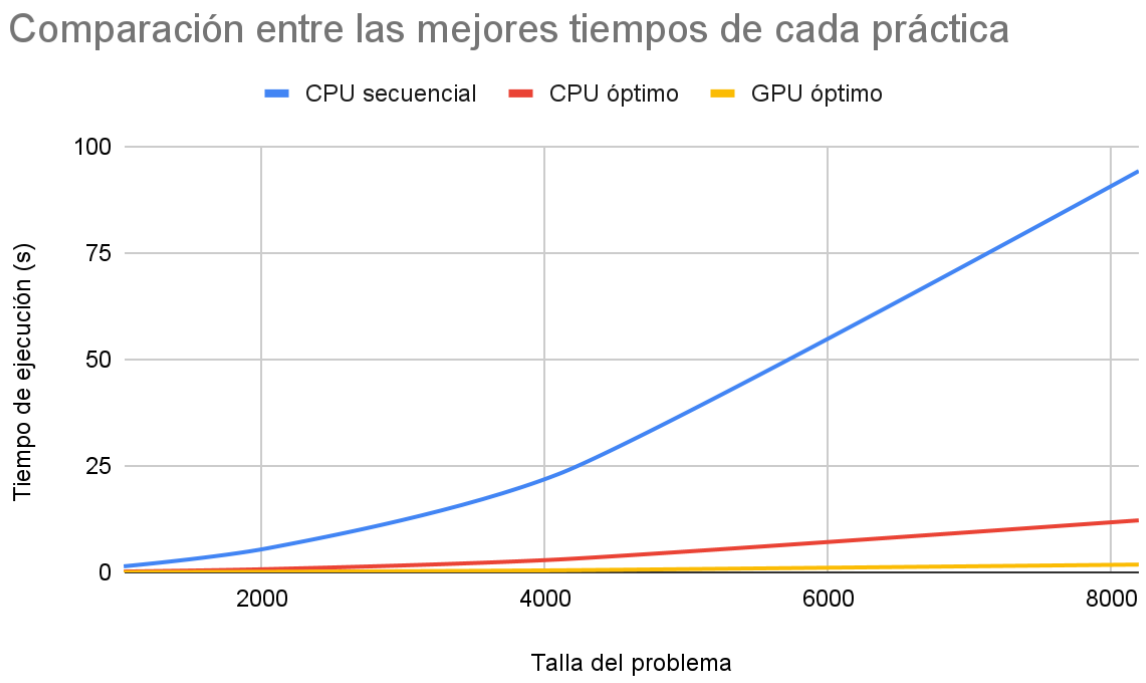
- *atomic* es más lento que el resto de operaciones porque todos los hilos tratan de sumar su valor a una misma variable, lo que crea bloqueos y serializa el acceso a la misma.
- La API de *cublas* está optimizada como cabría esperar y funciona mejor que alternativas manuales que pueden tener algunos fallos de implementación.
- *shared* y *params* son implementaciones similares, pero como esperado, el paso por parámetros de un mismo vector tiene peor rendimiento, seguramente debido al

6.3.5. Comparación entre CPU y GPU

La diferencia entre los tiempos de ejecución de CPU y GPU son los siguientes:

size	CPU secuencial	CPU óptimo	GPU óptimo	accel CPU/GPU
1024	1.41E+00	1.91E-01	4.35E-02	4.40
2048	5.69E+00	7.56E-01	1.31E-01	5.76
4096	2.31E+01	3.02E+00	4.78E-01	6.32
8192	9.42E+01	1.22E+01	1.82E+00	6.70

Como era evidente desde el principio, la GPU debería conseguir y consigue tiempos muy inferiores a los de la CPU, contando en ambos casos con todas las optimizaciones de los análisis de este informe. La GPU consigue una aceleración frente a la CPU de entre 4 y 7 veces, algo increíblemente significativo en términos de tiempos de ejecución.



Como se puede ver en el gráfico anterior, el tiempo de la CPU se dispara con el tamaño del problema. Debido al gran poder de computación de la GPU, la diferencia entre tallas de problema no es tan grande. Obviamente, la CPU en secuencial obtiene tiempos terribles.

Anexo

Un python para gobernarlos a todos

El código Python y el resto de archivos auxiliares ofrecidos estaban estructurados de tal forma que el añadido de algunas características clave como:

- la obtención de resultados de múltiples funciones de manera simultánea
- la obtención de resultados de múltiples tamaños de manera simultánea
- el paso por parámetros de las opciones deseadas
- la selección e inicialización automática de librerías a examinar

era imposible, además de ser difíciles de mantener y leer en sí. Es por ello por lo que para este trabajo se han reescrito todos estos ficheros. Gracias a esto, se puede ejecutar el script de Python con muchas opciones con solo cambiar un par de parámetros de llamada.

Ejemplos de ejecuciones

```
python Launcher.py $xmin $xmax $ymin $maxiter prof own
sizes 1024 2048 4096 debug tpb 32
```

El comando anterior ejecuta la librería proporcionada y la generada por el código propio, con cuatro tamaños, imprimiendo las imágenes y utilizando 32 hilos por bloque (lo que indica una ejecución de las librerías GPU).

Para obtener la tabla de experimentación de la práctica 4, se utilizan los siguientes comandos:

```
params="${xmin} ${xmax} ${ymin} ${maxiter} py own sizes
256 512 1024 2048 4096 8192"

# Secuencial
export OMP_NUM_THREADS=1
python Launcher.py $params

# Paralelo
unset OMP_NUM_THREADS
python Launcher.py $params noheader -py prof methods
normal schedule_runtime
```

Como se puede apreciar, se lanzan todos los tamaños de manera simultánea[10]. Además, esto permite enviar a las colas GPU tanto la versión secuencial como la versión paralela en un solo script, obteniendo una sola respuesta evitando tener que estar pendiente de múltiples ejecuciones y devoluciones. Además, se elimina la ejecución de la función de Python para evitar malgastar tiempo en obtener resultados repetidos.

Para obtener la tabla de resultados de la práctica 6, se utiliza la siguiente instrucción:

```
python Launcher.py $xmin $xmax $ymin $maxiter onlytimes prof own sizes 1024 2048 4096 8192 10240 tpb 32 methods all
```

Formato de resultados

El script devuelve los resultados en formato “csv” delimitado por puntos y comas. Por defecto, se devuelve el nombre de la función, el modo de ejecución[11], el tiempo de ejecución, el error cometido y el promedio del resultado (utilizando la función promedio correspondiente). Dependiendo de los argumentos de entrada, también se puede incluir el error de binarizado, los tiempos de ejecución del binarizado y del promedio y el número de hilos por bloque (en caso de estar en modo GPU/CUDA).

Ejemplo de ejecución para la práctica 4:

```
Function;Mode;Size;Time;Error;Average;
mandelProf;PARALELO;2048;4.31188E+00;-;221.97750806808472;
mandelPy;PARALELO;2048;1.31529E+02;0.0;221.97750806808472;
mandel_normal;PARALELO;2048;1.01743E+00;0.0;221.97750806808472;
mandel_schedule_guided;PARALELO;2048;7.56102E-01;0.0;221.97750806808472;
```

Ejemplo de ejecución para la práctica 6 (con más opciones):

```
Function;Mode;Size;TPB;Time;Error;Average;Bin (err);Bin Time
mandelProf;GPU;2048;32;3.55479E-01;-;221.9761254787445;-;6.23131E-03
mandelGPU_normal;GPU;2048;32;1.71371E-01;0.0;221.9761254787445;0.0;6.27661E-03
mandelGPU_heter;GPU;2048;32;2.07167E-01;307784.3493032094;220.5662591457367;333370.9319211859;6.32763E-03
mandelGPU_unified;GPU;2048;32;1.77213E-01;0.0;221.9761254787445;0.0;6.32882E-03
mandelGPU_pinned;GPU;2048;32;1.83663E-01;0.0;221.9761254787445;0.0;6.31547E-03
```

[10] El script de Python limita el tamaño de las ejecuciones de numpy a 2048, ya que los tamaños superiores supondrían días (en algunos casos hasta años) de ejecución.
[11] El modo de ejecución puede ser *SECUENCIAL*, *PARALELO* o *GPU* dependiendo del entorno y las opciones de ejecución.

Gracias al comando *column*, se puede embellecer el formato resultante para que sea más fácil de leer a través de la consola. Dicho comando se ha utilizado durante todo el desarrollo de ambas prácticas.

Un Makefile para atraerlos a todos y atarlos a las tinieblas

Puesto que el script de Python se encarga de manejar la inicialización de librerías, de funciones dentro de las librerías, de todos los argumentos, tamaños y demás opciones, tiene sentido que también se encargue de llamar al Makefile que compile tan solo lo necesario.

Por ello, al nuevo Makefile se le pueden pasar las opciones “cuda” y “omp” para compilar ambas prácticas directamente desde el script, evitando así tener que cambiar las llamadas desde los shell-scripts.

Metodología de trabajo

Visual Studio Code

Para el desarrollo de ambas prácticas se ha utilizado Visual Studio Code a través de SSH en conexión con el servidor de la asignatura. Trabajar de esta manera permite una flexibilidad y configuración que, a mi parecer, son totalmente necesarios. El hecho de trabajar desde tu ordenador te permite saber exactamente dónde está todo y qué esperar al hacer cualquier acción, lo que elimina el tiempo que supone aprender cómo hacer funcionar otro entorno de trabajo.

Trabajar con VSCode permite tener todas las opciones personales sincronizadas así como extensiones. Estas extensiones facilitan mucho el trabajo, ya sea porque aportan sintaxis a lenguajes, herramientas adicionales, opciones visuales…

¿IAs?

Una de estas extensiones es GitHub Copilot, una IA con forma de extensión que ayuda mucho al desarrollo, principalmente a la hora de orientarse o repetir código de manera semi-automatizada. ChatGPT es otra IA de diálogo con amplios conocimientos en programación en la que me he apoyado, o más bien lo he intentado. Estas IAs no son mágicas, ni mucho menos. Son útiles en algunos casos, pueden llegar a sorprender en otros, pero la mayor parte de las veces no dan implementaciones que se puedan llegar a utilizar.

He seguido la evolución de estas inteligencias artificiales muy de cerca, y me han ayudado en gran medida, a veces a la hora de realizar tareas repetitivas y a veces a comprender problemas mejor de lo que lo hacen profesores. Sin embargo, y en base a mi experiencia,

hay que estar muy alerta y no fiarse de todo, en especial de implementaciones de métodos enteros o de problemas complejos.

Scripts y sincronización de ficheros

Se han desarrollado scripts en bash con varios propósitos. El más importante de estos scripts es *enp*, que permite lanzar tareas a la cola GPU y obtener el output directamente en consola, sin necesidad de manejar ficheros generados por las máquinas a las que se envíe, algo que también es totalmente necesario en mi opinión. El uso de estos scripts también ha provocado algún que otro problema.

A la hora de sincronizar los archivos, se utiliza *git* para actualizar el repositorio personal de la asignatura. Esto permite mantener un control de versiones además de ser capaz de visualizar los ficheros de manera online sin tener que estar conectado al servidor de la asignatura ni tener siquiera el repositorio clonado de manera local.

Problemas durante el desarrollo

Falta de tiempo y dificultad

El problema principal del desarrollo ha sido el tiempo. Las dos prácticas combinadas suponen un trabajo de casi cien horas. Es una de las entregas individuales más grandes de toda la carrera, por lo que organizarse correctamente es muy difícil.

Además de la organización, el trabajo en sí no es para nada fácil. El desarrollo de todo el código en la primera parte, el mantenimiento del código auxiliar generado (Python, shell-scripts, makefiles) y, en especial, el desarrollo de algunas implementaciones de la segunda práctica han supuesto muchos quebraderos de cabeza. Además de ser uno de los trabajos más grandes, es también uno de los más difíciles de completar.

Calidad del código proporcionado

Los códigos en Python y Shell proporcionados dejan mucho que desear y suponen muchas molestias, sobre todo a la hora de obtener resultados. Los códigos originales se basan en el envío de muchas tareas pequeñas con valores diferentes para absolutamente todo: tamaños, librerías, métodos y demás opciones. Esto genera una situación de código muy difícil de manejar, además de una cantidad de archivos resultantes de las colas inmensa.

Desde el primer momento, he tratado de generar mi propia base de trabajo que me permita hacer todo lo que yo quiero como yo quiero, pese a que no se valore. Pese a la increíble cantidad de trabajo que ha supuesto, creo que al final ha merecido la pena, no solo por la

facilidad de obtener múltiples resultados de manera clara y simultánea, sino porque comprendo mucho mejor todos los entresijos.

Problemas con los scripts y mi método de trabajo

A lo largo del curso, he recibido instrucciones contradictorias con respecto al uso de mis métodos de trabajo. Lo que al principio estaba permitido se convierte meses después en un supuesto uso excesivo de scripts y VSCode.

Como he intentado aclarar, no creo que VSCode haya supuesto ninguna carga excesiva en el sistema, siempre que el sistema funcione correctamente, además de que no soy la única persona que lo utiliza. Si VSCode hubiera supuesto un problema, se habría manifestado mucho antes en el curso, en especial durante el desarrollo de la práctica 3.

Posteriormente, he descubierto que algunos de los scripts de colas que se ponen a disposición, como *qstatus*, *qstat* y principalmente *qacct* tienen un rendimiento malo, por lo que es conveniente reducir su uso al mínimo.

Problemas con el servidor

Está claro que el servidor de la asignatura tiene un rol crucial en el desarrollo de la misma y todas las actividades de laboratorio giran en torno a él. Es evidente que con un servicio tan demandado por todos los estudiantes, no va a ser una experiencia libre de errores.

El rendimiento del servidor se ha ido degradando con el paso de los meses, sin motivo aparente. Sin embargo, el problema principal son los atascos que se generan en las fechas anteriores a las entregas y los abusos de las colas.

¿SIMD?

Como especificado en los enunciados de las prácticas, se podría utilizar instrucciones SIMD (SSE o AVX) para mejorar el rendimiento de algunas partes de la práctica.

Sin embargo, no es posible utilizar la directiva *simd* de OpenMP porque la versión contenida en los compiladores de las colas[12] es mucho anterior a la implementación de la directiva[13].

He jugado[14] con la implementación manual de instrucciones SSE, algo ya visto en cursos pasados[15], pero no se han obtenido los resultados deseados. (ej: Al implementar SIMD en el algoritmo de tiempo de escape, se multiplican los tiempos de ejecución)

[12] La versión de OpenMP de las colas de la asignatura es 2011-07.
[13] Se tiene 3.1, se necesita 4.0 (**≈9 años de diferencia**).
[14] Código de la función aquí.
[15] Visto en el trabajo final de Arquitectura de Computadores, 21-22.