ACAP

Práctica 5

Implementando el filtro de Sobel en CUDA y MPI

Antonio Priego Raya

1. DESCRIPCIÓN DEL PROGRAMA Y CARGAS *(Extracción de P3)*

El programa va a realizar un procesamiento de imágenes basado en gradientes. En este caso aplicaremos la máscara de Sobel:

```
//
// Devuelve el gradiente de 'x' del punto(x,y) de la imagen
//    Proceso: Aplicar la máscara
//
//   Orden       Máscara (x)
// [z1 z2 z3]    [-1  0  1]
// [z4 z5 z6]    [-2  0  2]
// [z7 z8 z9]    [-1  0  1]
//
// Gx= (z3+2*z6+z9) - (z1+2*z4+z7)
```

Ilustración 1: Comentarios previos a funcion x_gradient() en mi código

```
//
// Devuelve el gradiente de 'y' del punto(x,y) de la imagen
//    Proceso: Aplicar la máscara
//
//   Orden       Máscara (y)
// [z1 z2 z3]    [-1 -2 -1]
// [z4 z5 z6]    [ 0  0  0]
// [z7 z8 z9]    [ 1  2  1]
//
// Gy= (z7+2*z8+z9) - (z1+2*z2+z3)
```

Ilustración 2: Comentarios previos a funcion y_gradient() en mi código

De forma que, aplicando este filtro a cada pixel, para lo cual debemos tener en cuenta que necesitamos los píxeles inmediatos; obtenemos la imagen procesada. En este caso, el procesamiento se aplica a detección de bordes. También tener en cuenta que es un filtro aplicable a escala de grises.

He utilizado como muestras de carga 3 imágenes:

- Muestra muy pequeña 793x533
- Muestra grande 3840x2160(4K)
- Muestra muy grande 7680x4320(8K)

He elegido estos tamaños, porque quería un tamaño que evidenciara un procesamiento de carga baja, uno de carga media, aunque la consideremos una resolución alta, y por último una carga alta que evidenciara un procesamiento de carga considerable.

2. Granularidad de paralelismo

Siguiendo las indicaciones que aclaró para este ámbito, he reunido la granularidad del paralelismo de CUDA y MPI en bloques de '*Rendimiento*', agrupando en estos bloques las ganancias de mejor a peor de ambas implementaciones para así poder realizar una comparación entre estos.

Realicé un pequeño estudio de tiempos en CUDA según las hebras por bloque. Tiempos obtenidos con la ejecución del script /script_impresionCruda. Obteniendo los tiempos de /data/ejecucion/numThread.
De forma que pude observar que, incrementando el número de hebras por bloque, obtenemos un tiempo en kernel ligeramente superior, al igual que los tiempos de gestión de memoria. Por lo que decidí escoger 100HxB(10x10) hebras por bloque para representar a las '*Alto Rendimiento'* que equipararé con 8 cores en el paralelismo de CPU.
De la misma forma equipararé en '*Medio Rendimiento*' a 576HxB(24x24) y 4cores.
Y por último en '*Bajo Rendimiento*' nos queda 1024(32x32) y 2cores.

	MPI #coresCPU	CUDA #threadsXbloque
Bajo Rendimiento	2	(32,32) = 1024
Medio Rendimiento	4	(24,24) = 576
Alto Rendimiento	8	(10,10) = 100

3. Recolección de resultados

Una vez tenemos la granularidad del paralelismo para CUDA, escribo un script que recoge los tiempos que arroja el programa para todas las posibilidades de tamaño de muestra y granularidad.
Este script es script /script_medicionNumThreadsFinal y guarda la información en data/ejecucion/medicionFinal.
Llevo estos tiempos a la tabla de cálculo que usé en la P3 para tener acceso también a los tiempos de MPI. Y con esto, llego a construir la tabla que servirá para la gráfica que analizaremos a continuación.

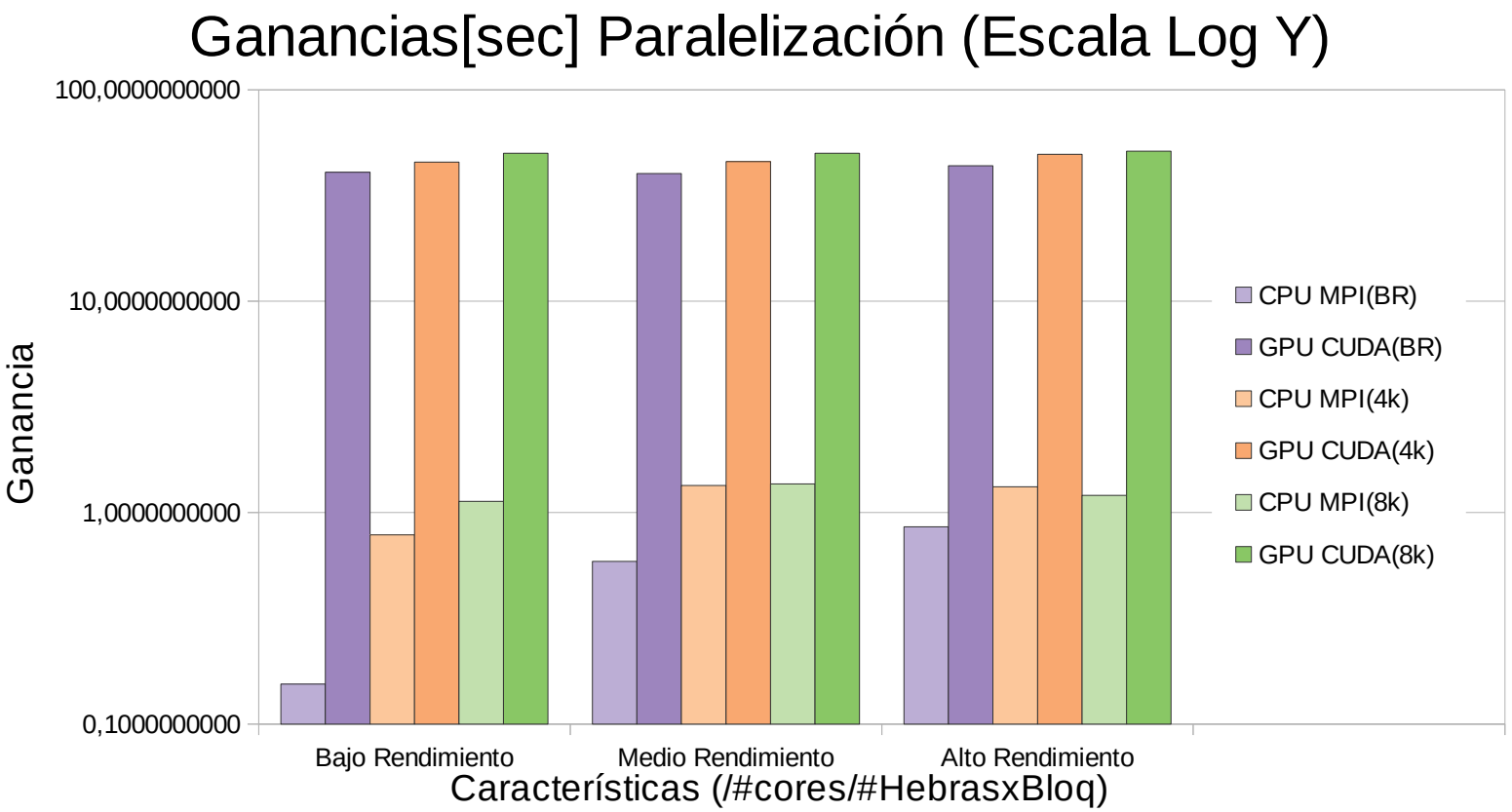
TABLA DE GANANCIAS **Respecto de Secuencial**

Resolución 4k Baja Resolución		**CPU MPI(BR)**	**GPU CUDA(BR)**	
	Bajo Rendimiento(BR)	0,1542552623	40,6645963248	2cores | 1024HxB
	Medio Rendimiento(BR)	0,5870430615	40,0715686542	4cores | 576HxB
	Alto Rendimiento(BR)	0,8571980683	43,7447034371	8cores | 100HxB
		CPU MPI(4k)	**GPU CUDA(4k)**	
	Bajo Rendimiento(4k)	0,7844627403	45,5017163541	2cores | 1024HxB
	Medio Rendimiento(4k)	1,3454465052	45,7345112717	4cores | 576HxB
	Alto Rendimiento(4k)	1,3234474803	49,4662023784	8cores | 100HxB
Resolución 8k		**CPU MPI(8k)**	**GPU CUDA(8k)**	
	Bajo Rendimiento(8k)	1,1307901236	50,0462064598	2cores | 1024HxB
	Medio Rendimiento(8k)	1,3675060360	50,0572329053	4cores | 576HxB
	Alto Rendimiento(8k)	1,2082840046	51,1825856076	8cores | 100HxB

Para más información como los tiempos y sus gráficas o verificar cómo se han reunido los datos de la tabla, consultar /data/Resultados.ods.

4. Análisis de resultados

Obtenemos finalmente:



Como observamos, los resultados obtenidos son los que cabría esperar.

Vamos a analizar todo lo que podamos de forma reflexiva.

1. Lo primero que resalta es la enorme diferencia entre las ganancias que ofrece MPI con las que ofrece CUDA.

   1.1. Como dijimos en la P3, la paralelización de MPI en algunas situaciones mejora los tiempos secuenciales. Si bien, no ofrece ganancias sorprendentes.

   1.2. En CUDA, también podemos discernir mejoras de rendimiento. Sin embargo ocurre algo que podría ir en contra de una lógica básica, y es que, la mejora de rendimiento se produce cuando rebajamos las hebras por bloque, en lugar de cuando las aumentamos. Por esto fue que decidí hacer la clasificación de '*Alto Rendimiento*' para menos HxB; siguiendo las indicaciones, debemos hacer un estudio comparando los mejores, medios y peores resultados de MPI con los de CUDA.

   ¿Qué lógica sería aplicable a estos resultados? Pues podría deberse a múltiples razones, una de ellas es que, al aumentar el número de *threadsXbloque*, disminuye el *numBloques* de bloques con que procesamos; por tanto pese a que podría parecer que aumentando los threads ganamos paralelismo, lo estamos perdiendo.

```
116     dim3 threadsXbloque(tam_threads, tam_threads, 1);
117     dim3 numBloques((int)(orig.cols/tam_threads), (int)(orig.rows/tam_threads), 1);
118
119     sobel<<<numBloques, threadsXbloque>>>(device_A, device_B, orig.cols, orig.rows, condicion_frontera);
```

Sin embargo, lo que marca la diferencia no es el tiempo de ejecución, por lo tanto, este motivo no es suficiente. Hay que darle una explicación desde el prisma del tratamiento de memoria. Tras un tiempo investigando y pensando qué podría causar esta tendencia de mejora en los tiempos de gestión de memoria al disminuir las HxB, he llegado a la conclusión de que es simple coincidencia que la tendencia en estos tiempos, que son los determinantes para que se refleje dicha tendencia en el tiempo total, sea de mejora. Ya que tiene sentido que los tiempos empeoren cuando aumentamos el tamaño de la muestra(la imagen), pero no tiene sentido que varíen los tiempos de memoria para la misma imagen, ya que la granularidad no afecta a la gestión de memoria.

Por tanto si hacemos un análisis de tiempos de ejecución del kernel obtenemos justo lo que nos dice la lógica, que es que obtenemos mejoras al disminuir la densidad de HxB, ya que, como he explicado, esto implica un aumento de bloques global. Pero si hacemos un análisis de tiempos totales, pese a que los resultados siguen siendo continuistas con la línea de razonamiento que he explicado, es mera coincidencia; ya que en teoría los tiempos de memoria deberían permanecer constantes e independientes de la granularidad.

1.3. Para finalizar comentando la diferencia de ganancias CUDA/MPI y reuniendo la información de los dos puntos anteriores. Pese a la perspectiva distorsionada de la escala logarítmica para poder observar el crecimiento de MPI, este no puede hacer frente a la ganancia que ofrece CUDA.

2. No se llega a apreciar en la gráfica con claridad el incremento que produce hacer uso de menos HxB, pero hay diferencias hasta de 10 puntos(Pág2) al reducirlos del máximo que mi máquina permite(1024) hasta el mínimo que he estipulado(100). Mientras que MPI en su peor tiempo presenta una ganancia de 0’15(Pág2) respecto del secuencial y donde más ventaja debería obtener [8k|8cores], la ganancia es de tan solo 1’20(Pág2).
   Incluso si nos produce reticencia comparar la ganancia del tiempo total y queremos juzgar el tiempo de cómputo, CUDA sigue siendo superior sobre MPI.
   Por tanto podemos decir que la escalabilidad en CUDA también es mejor.

5. Conclusión

Como cabía esperar, los tiempos en CUDA son mejores, desproporcionados en comparación con los que arrojan las ejecuciones de MPI.

Algo que podría sorprender y por ello cabe destacar es que más threads por bloque no implica ejecuciones más rápidas, por la razón explicada; los threads por bloque guardan relación con el número de bloques. De manera que si aumento el número de threads, disminuyen los bloques, elemento que podríamos tratar en este estudio como unidad de ejecución paralela de CUDA, y por tanto perderíamos paralelismo.

Pero como he explicado(Punto 1.2, Apartado 4) este razonamiento obedece a tiempos de ejecución kernel, pero no debería hacerlo con los tiempos de memoria, por tanto, los tiempos totales “favorecen” al razonamiento, pero son simple coincidencia, ya que la mejora en tiempos de ejecución es despreciable frente a los tiempos de gestión de memoria.

Por tanto, como síntesis. CUDA es una alternativa muy superior a MPI para aplicar el filtro de Sobel, en todos los ámbitos que puedan enfrentarse.