Pontificia Universidad Católica de Chile
Facultad de Matemáticas
IMT2112/ Algoritmos paralelos en computación científica
Alumno: Tomás González

Tarea Nº4

1. Resultado de correr opencl-devices.cpp:
   1. Device: Intel(R) Core(TM) i5-5250U CPU @ 1.60GHz
      1.1 Hardware version: OpenCL 1.2
      1.2 Software version: 1.1
      1.3 OpenCL C version: OpenCL C 1.2
      1.4 Parallel compute units: 4
   2. Device: Intel(R) Iris(TM) Graphics 6100
      2.1 Hardware version: OpenCL 1.2
      2.2 Software version: 1.2(May 26 2020 20:53:35)
      2.3 OpenCL C version: OpenCL C 1.2
      2.4 Parallel compute units: 48

2. El algoritmo consiste en:

   a) Elegir al azar K números complejos del círculo de radio 2 centrado en el origen de $\mathbb{R}^2$. Así elegimos los puntos c.

   b) En la literatura una propiedad conocida es que si $|x_n| = |x_{n-1}^2 + c| \geq 2$ para algún n, entonces $c \notin$ al conjunto de Mandelbrot. Luego, para determinar si un punto pertenece o no al fractal, calculamos x_n para n grande (a determinar empíricamente), y si alguno de los números complejos $x_0, x_1, ..., x_{n-1}, x_n$ tiene valor absoluto mayor que 2, entonces c no está en el conjunto de Mandelbrot.

   c) Los resultados son buenos con 1000 iteraciones de x_n, pero finalmente los mejores resultados se obtienen con 10.000. Esto es empírico.

3. a) No usé una semilla, ya que MonteCarlo nos da garantías de convergencia (resultados reproducibles) de todas maneras.

   b) No se puede paralelizar porque la función, al ser pseudorandom, si la ejecutamos al mismo tiempo con diferentes grupos, vamos a obtener los mismos números complejos c, ya que es practicamente ejecutar la misma función pseudorandom varias veces. Y es claro que tener puntos repetidos está mal, nos quita garantías de montecarlo ya que ya no estamos sampleando independiente desde el circulo, sino que hay dependencia entre los puntos que se generan al mismo tiempo (son muchos puntos c iguales). Para evitar esto, tendríamos que ejecutar la función prácticamente usando un hilo a la vez, y así poder obtener diferentes números c. Esto es ineficiente, ya que solo un thread estaría haciendo algo en cada momento, perdiendo todo el poder de paralelización de la gpu.

4. a) Se calcula con la función en el kernel 'recursion'.

   b) Mi procedimiento es el siguiente: tengo un vector con indicadores (se llama indic en el programa), que tiene 1 en las entradas donde el número c no está en el conjunto, y ceros donde si está en el conjunto. Se hace una reducción de este vector dentro de cada grupo, pero la reducción global se hace en CPU, como lo conversado en la ayudantía: siempre va a haber que hacer una parte 'lineal', que en mi caso era juntar las sumas obtenidas en cada grupo. Para hacerlo de maneral thread safe dentro de la GPU tendría que haber llamado un kernel *cant_grupos* veces, lo cual es mucho más ineficiente que hacerlo lineal en la CPU.

   La operación de reducción se hace de la siguiente manera: dentro de cada grupo hay chunks, y un hilo se encarga de sumar el vector de indicadores dentro del chunk. Luego, se completa la suma local de cada grupo sumando todos los chunks del grupo. Todo esto lo hace la función 'reduction'. Finalmente, las sumas locales de cada grupo se juntan en una suma global en la CPU.

   c) Como se hizo un padding con 0, que son iteraciones que cumplen $0 = 0^2 + 0$ entonces nos ahorramos tener que hacer un $if(global\_id < n)$ en la función de la recursión. El único if que hacemos es: $if|x_n|^2 > 4$, entonces el indicador se hace 1, else continuar con $x_n = x_{n-1}^2 + c$. Esto es eficiente dentro de todo, ya que solo existen dos opciones posibles para todas las entradas del vector, por lo que un grupo de entradas del vector hace algo y al siguiente paso el resto de las entradas hace otra cosa. Entonces, en dos ciclos se completa una iteración, lo que es la mitad de lo que sería posible sin la divergencia que provoca el if, sin embargo en la práctica es muy rápido de todas formas, ya que solo requiere dos pasos cada iteración en vez de una, en el peor caso podría requerir que cada entrada haga algo diferente y eso provocaría una divergencia mucho más terrible (cada iteración se completaría en una cantidad de pasos igual a la cantidad de entradas por grupo, que es mucho más que 2) que hacer elegir a cada entrada entre dos opciones, que es nuestro caso.

5. Para calcular el error, como con la reducción obtengo la cantidad de números que diverge, entonces ahora la divido por el total de números ($\_K$ en el informe) para obtener la proporción que diverge, y haciendo $1-$ esta proporción obtengo la proporción que NO diverge. Luego, esta proporción debería, por Montecarlo, ser igual a la proporción entre el área del conjunto de Mandelbrot y el área del círculo de radio 2. Entonces, para calcular la aproximación hacemos $4\pi(1-p)$, donde $p = d/n$ es la proporción de números que divergen, y d representa a la cantidad que diverge (reducción), n la cantidad total.

   El error finalmente es $|1.50659177 - 4\pi(1-p)|$, que en los experimentos con $n = 1.000.000$ y cantidad de iteraciones x_n igual a 10.000, se obtiene $0.005438, 0.001882, 0.003383$ en tres experimentos distintos.