Pontificia Universidad Católica de Chile
Instituto de Ingeniería Matemática
IMT2112

Tarea 4

27 de noviembre de 2020

Fernando De Diego Ávila

Mi tarea 4 fue desarrollada con PyOpenCL, debido a diversas complicaciones para instalar OpenCL en mi computador. El número de iteraciones realizadas y el largo del vector de números complejos generados que utilicé (100000, en ambos casos), fueron determinados a partir de las limitaciones de mi computador. Al momento de ejecutar el programa en mi computador, un mayor número de cualquiera de dichos factores se hubiera traducido en un resultado incorrecto.

Pese a lo mencionado, el mismo algoritmo fue testeado en Google Colab, en donde un aumento de iteraciones y números considerados se tradujo en una mayor precisión, como era de esperar. Probablemente, esto indica que la razón está relacionada con la cantidad de información que se puede transferir hacia la GPU de un dispositivo (y que depende de la de GPU utilizada).

Preguntas

2

(a) Para escoger los puntos c, basta generar números pseudo aleatorios sobre un rango específico con un algoritmo de uso común. Como se nos pidió representar cada número complejo como un punto con dos coordenadas, entonces es necesario generar $2n$ números aleatorios en total, donde n es el número total de puntos que se quiere generar.

En mi implementación, para escoger el rango me basé en un resultado que señala que un número c pertenece al Conjunto de Mandelbrot si y sólo si el módulo $|z_m|$ de cada elemento de la sucesión $z_m = z_{m-1}^2 + c$ es menor o igual a 2, para todo m (Fuente: [1]). De esta forma, generé dos arreglos con números aleatorios en el intervalo $[-2, 2]$, ya que sabemos que el resto de los números están fuera del conjunto de Mandelbrot. La función utilizada fue `np.random.uniform(-2, 2, n).astype(np.float32)`.

(b) Como fue mencionado anteriormente, un punto está dentro del fractal si y sólo si

$|z_m| \leq 2$ para todo m, donde z_m es la sucesión que define al fractal. Como no podemos saber cómo se comporta la sucesión hasta el infinito, podemos asumir que un número complejo pertenece al fractal si, luego de una cantidad razonable de iteraciones, el módulo de los elementos de la sucesión generada sigue siendo menor o igual a 2. Como mencioné anteriormente, consideré 100000 iteraciones en mi implementación.

3

La generación de números aleatorios es un proceso determinista. Es decir, si se le entrega la misma semilla a un programa, entonces se puede calcular el número pseudo-aleatorio que se obtendrá. En particular, `NumPy` utiliza combinaciones de bits, y las maneja internamente (Fuente: [2]). Debido a esto, si se le entrega a dos (o más procesos) distintos la tarea de generar números aleatorios, dada una semilla, entonces se obtendrá dos veces distintas el mismo resultado. Junto a esto, si no se le entregan semillas a los procesos, los resultados no son replicables, y muchas veces existe igualmente una correlación entre los números generados, producto de que el computador genera números dadas las condiciones de ciertos factores internos, los cuales se ven alterados por el tiempo en el que se solicita el número (por lo que si se generan números aleatorios paralelamente, los números serán muchas veces los mismos).

Ocurre lo mismo con la función `rand()` de `C++`.

4

(a) Implementé dos códigos parecidos, cuyo proceso de reducción será explicado a continuación:

- `main.py`: Corresponde al archivo principal de la tarea, el cual incluye dos reducciones, distribuidas en dos *kernels*. La primera reducción fue realizada de manera similar a la ayudantía. El funcionamiento es el siguiente: Cada hilo verifica si su punto asociado pertenece al fractal, y luego el primer hilo de cada grupo suma los resultados del grupo y los almacena en un arreglo de resultados locales, cuyo largo es igual al número total de grupos.

  Por otro lado, la segunda reducción fue realizada en un *kernel* aparte, cuyo fin consiste en reducir el arreglo de resultados locales en un solo número que representa la aproximación del area del fractal. Para esto, se utilizó solamente el primer hilo, debido a que el número de elementos a reducir es pequeño.

- `other.py` Corresponde a una aproximación ligeramente distinta a la mencionada en el archivo principal. En particular, se incorpora el uso de *chunks* asociados a cada hilo, de manera tal de que cada hilo realice más trabajo. Las reducciones siguen la misma idea.

(b) Hay 3 *if-else statements* en mi código (aparte de los *for*, que igual verifican una condición de término). El primero, verifica que el módulo de cada elemento de la sucesión no sea mayor a 2. El segundo indica si el hilo tiene un id local de 0, para

realizar la primera reducción. El tercero indica si el hilo tiene un id global de 0, para realizar la segunda reducción.

Los últimos dos no tienen tanto problema de divergencia de trabajo, ya que no hay más trabajo luego de la ejecución de dichos *statementes*. Por el contrario, el primero puede generar ineficiencias por temas de divergencia de trabajo, ya que según el resultado de la condición se continúa calculando los siguientes elementos de la sucesión o se etiqueta a un número como que no pertenece al fractal. Estas son dos tareas distintas, que pueden ser una fuente de ineficiencia en la ejecución, provocada por el modelo `SIMD` de la GPU.

Pese a lo mencionado, el código parece tener resultados con un error bajo, en un tiempo de ejecución decente.

5

El resultado del programa `main.py` con un total de 100 mil datos y 100 mil iteraciones, es de 1,5025584, lo cual tiene un error aproximado de 0,00403 con respecto al valor real (%0,267718 de error). Por otra parte, el resultado de `other.py` con 100 mil iteraciones, *chunks* de tamaño 100 y 10 mil iteraciones es de 1,52, lo que tiene un error aproximado de 0,001340 (%0,889969 de error).

Corriendo el ejercicio en Google Colab, el resultado del programa `main.py` con un total de 100 millones de datos y 100 mil iteraciones es 1,5068718, lo que tiene un error aproximado de 0,00028004 (% 0,018588 de error).

Referencias

[1] Mandelbrot Set: `https://en.wikipedia.org/wiki/Mandelbrot_set`

[2] Random sampling: `https://numpy.org/doc/stable/reference/random/index.html`