Computación de Alto Desempeño - INF-351
Informe Laboratorio IV

Anghelo Nicolas Carvajal Carvajal
anghelo.carvajal.14@sansano.usm.cl
201473062-4

Eduardo Francisco Pozo Valdés
eduardo.pozo.14@sansano.usm.cl
201473040-3

Herman Nicolás Valencia Lara
herman.valencia.13@sansano.usm.cl
201373098-1

22 de Agosto de 2019

1. Preguntas

1. **[Pregunta]** ¿Qué puede decir sobre los tiempos obtenidos? Comente al respecto y concluya.
2. **[Pregunta]** Compare los tiempos de ejecución de ambas versiones. ¿Qué puede comentar al respecto? ¿Obtuvo alguna mejora? Explique
3. **[Pregunta]** ¿Qué idea se le ocurre que pueda ayudar a optimizar la solución a este problema?. Desarrolle y explique su propuesta.

2. Ambiente de ejecución:

`Nvidia GTX 1050Ti, Compute Capability 6.1`

3. Respuestas

1. El tiempo obtenido en la implementación de GPU es de alrededor de los 82091.71[ms], este tiempo es mucho mas reducido que el tiempo en CPU que ya a los 15[min] aun no presentaba respuesta. Por lo que es por lo menos 5 ordenes de magnitud más rápido.
2. En este caso hubo una mejora, la diferencia entre implementaciones de GPU es de 5266.24[ms], que no es mucha. El nuevo algoritmo que solo calcula la carga para una vecindad de radio 100 demora alrededor de los 76825.27[ms], esto puede deberse a que para examinar si un ion debe o no agregarse al calculo de la carga del vértice, primero debe calcular la distancia entre la carga y dicho vértice, calculo que es mucho más costoso que el de obtener la carga a sumar en el vertice.
3. Algunas posibles optimizaciones serian:
   - Modificar la cantidad de *threads*: Como cada GPU esta optimizada de distinta forma según la cantidad de *threads* por bloque, modificar este numero hasta llegar a un óptimo podría se beneficioso para la ejecución.
   - Paralelizar el cálculo de las cargas iniciales: Actualmente, cada *thread* de la GPU se encarga de calcular la carga que se encontraría en el punto que se le asigno. Este calculo implica un `for` de 5000 iteraciones que tiene que realizar cada `thread`, lo cual podría ser paralelizable. Para poder paralelizar esto, tendríamos que implementar una reducción usando memoria compartida.
   - Usar *streams* al calcular las cargas iniciales en los puntos de la grilla. Actualmente existe la función **set_Qs_in_chunks(float* dev_Q)**, esta funcion calcula la carga de los puntos a trozos de iones iniciales. Podría asignarse diferentes trozos a diferentes streams.
   - Poblar la malla de iones en GPU en vez de CPU.

4. Conclusiones

1. El saber que tipo de memoria usar y cuando puede optimizar bastante el tiempo de respuesta de un programa.
2. Siempre hay que considerar el *tradeoff* entre las asignaciones y copia de memoria para el uso de GPU con la velocidad que podría generar la CPU sin esas asignaciones. Para tareas que no son muy largas, muchas veces puede llegar a ser más óptima la segunda opción.
3. Para obtener el mínimo no siempre la implementación en GPU es mas rápida, depende del largo del arreglo donde se desea buscar el mínimo, la reducción de GPU vale la pena con un gran cantidad de datos. En este caso con este tamaño de grilla se obtiene 8192^2 números en el que buscar el mínimo, cantidad suficiente para que haga sentido hacer uso de una reducción.

5. Compilación

1. Para compilar la versión que agrega todos los iones en el calculo:

```
$nvcc reduction.cu kernels.cu ion_placement_infinity.cu -o ion_placement_infinity
```

2. Para compilar la versión que agrega los iones de la vecindad en el calculo:

```
$nvcc reduction.cu kernels.cu ion_placement_radius.cu -o ion_placement_radius
```