INF351: Computación de Alto Desempeño
Laboratorio 3

Lobos y Conejos

Grupo 6 : Felipe Otero 201604006-4
Sinéad Eriksson 202290109-8
Tomás Velásquez 201721055-9

Fecha de entrega: 04/06/2022

1. Parte 1

2. Parte 2

2.1. ¿Cual de las dos implementaciones muestra mejor desempeño?

La implementación en GPU es claramente mejor, siendo aproximadamente 18 veces mas rápida para el caso de 1000 iteraciones. La ganancia no es tan grande como en otros casos en los que hemos trabajado, pero debido a la gran cantidad de lectura y escritura a memoria global, el kernel es bastante lento. Para mejorar su desempeño se podría probar el uso de operadores ternarios en las condiciones de movimiento y reproducción, como también estudiar la posibilidad de unir los dos kernel, con el fin de invocar un solo kernel por iteración, que sabemos puede reducir considerablemente los tiempos de ejecución.

	CPU	GPU
Tiempo [ms]	72486.0	3244.5

2.2. ¿Condiciones de carrera?

Con el primer intento de implementar el algoritmo, cada thread iba a calcular una dirección y luego mover su conejo o lobo a la posición correspondiente en un buffer. El problema con esto es que al mover un conejo o lobo, existe una posibilidad de que otro thread intente guardar un conejo o lobo en esa misma posición al mismo tiempo. Para solucionar esto, se genero un kernel que guarda todas las direcciones a las que se moverán los animales, luego de esto se usa un kernel que en vez de mover los animales de una casilla a otra, mira a sus vecinos y en que dirección se van a mover para calcular cuantos animales caerán en la casilla actual y luego guardarlo en un buffer. Como cada thread solo escribe en la posición en la que se encuentra, no cabe la posibilidad de una condición de carrera.

Para la reproducción se utilizo un acercamiento similar, donde cada thread mira cuantos animales ahí en las casillas vecinas y calcula si terminara un animal o no en la casilla actual, para luego guardar ese resultado devuelta al arreglo original (no a otro buffer). Si se siguiera el acercamiento común de verificar en que casillas tenemos dos o mas animales, y luego escribir en las casillas de a los lados, cuando deben reproducirse, no podemos asegurar que el buffer se va a llenar, por lo que habría que vaciar el buffer antes.

3. Parte 3

4. Parte 4

De las tres versiones de códigos para contar a todos los animales, la mas rápida es la reducción atómica realizada por GPU. Existen dos razones para esta mejora en el tiempo, la primera es que solo se debe llamar a un kernel, la segunda, es que habiendo solo tantas sumas atómicas como bloques (un numero no mayor a 100 en esta implementación), provoca que la posible cantidad de conflictos al guardar un dato disminuyan, lo que aprovecha al máximo el hecho de no tener que llamar a un segundo kernel, ya que si hubiesen demasiados threads intentando hacer una suma atómica al mismo dato, se ocasionaría un cuello de botella el cual reduciría drásticamente la ventaja ganada al llamar un kernel menos.

	Tiempo [ms]
Reducción CPU	4
Reducción GPU	0.471
Reducción atómica GPU	0.255

5. Conclusión

Al final de 1000 iteraciones, en cada simulación que se ejecutó, la cantidad de lobos se reduce a 0 , mientras que la cantidad de lobos explota a su máximo alcanzable. Esto se debe a que los únicos depredados en la simulación son los conejos, por lo tanto su población solo se irá reduciendo a medida que pasa el tiempo

	Cantidad de conejos	Cantidad de lobos
CPU	0	1109105
GPU	0	1114154

Existen diferentes formas de optimizar un código en cuda, en este caso, probamos con la utilización de memoria compartida, que disminuye considerablemente los tiempos de procesamiento gracias a que se reduce los accesos a memoria global de la GPU, siendo el caso de la reducción una gran ejemplificación de su uso.

Por otro lado, tenemos el uso de operaciones atómicas, que nos entrega una forma rápida y sencilla de trabajar con las condiciones de carrera que surgen en las lecturas y escrituras en memoria, pero que deben ser utilizadas inteligentemente para no ser perjudiciales en el tiempo de ejecución del código, ya que utilizado ingenuamente puede generar que la resolución del problema se haga de forma secuencial, y bien utilizado, como en el caso de la reducción , puede lograr que el algoritmo duplique su rapidez.

También es importante destacar, que en el uso de memoria compartida en reducción, fue muy importante definir el tamaño correcto del gridSize, ya que dependiendo de esto el código puede sufrir una mejora considerable en los tiempos de ejecución. Esto tiene mucha relación con la arquitectura de cada GPU y de cuanto trabajo tiene que hacer cada hebra al momento de hacer la pre-suma inicial.

	Ejecución 1 [ms]	Ejecución 2 [ms]	Ejecución 3 [ms]	Promedio [ms]
Iteraciones CPU	76338	77406	76948	76897.33
Iteraciones GPU	4207	4472	4088	4255.66
Reducción CPU	4	4	4	4
Reducción GPU	0.471	0.428	0.409	0.436
Reducción GPU atómica	0.255	0.243	0.245	0.248