Computación Paralela 2019/2020
Memoria de la Práctica 3: CUDA

Grupo: g210
Alumnos: Arenas Guerra, Ignacio - Herruzo Herrero, Juan

1. Puntos fundamentales y soluciones aplicadas para conseguir el objetivo.

1. Seguimiento de los tiempos de ejecución de las distintas partes del código.(A partir de línea 589)

Utilizando la función cp_Wtime() y un conjuntos de variables para registrar los tiempos de ejecución en diferentes momentos, esto nos sirvió para poder darnos cuenta de que aspectos estaban tardando más en ejecutarse.

Para que estos tiempos fuesen representativos sincronizamos la cpu con la gpu con cudaDeviceSynchronize() justo antes de cada llamada a cp_Wtime()

2. Gestión del tamaño de los grid y bloques (Líneas 605-606 y antes de cada llamada a kernel)

Al principio pusimos una geometría de bloque de (1024;1) y un grid de una dimensión cuyo tamaño variaba según la cantidad de células o celdas.

Una vez tenemos todo paralelizados, mediante prueba y error con distintas geometrías, llegamos a la que acabamos utilizando: (64;4) para bloques. La geometría de grid no cambió.

3. Creación de streams y eventos para paralelizar los kernels (A partir de la línea 608)

Como hay partes de la evolución que se pueden hacer en paralelo sin problemas de dependencias decidimos utilizar streams.
Para asegurarnos de no imcumplir dependencias primero creamos un grafo como el de la Figura 1.
En esta figura se pueden ver en qué streams está cada kernel, hay un kernel con el borde punteado, esto indica su posición teórica, pero por una errata se nos quedó en otro stream.

4. Gestión de memoria de la gpu (A partir de la 633) (explicar todos los array y cómo los utilizamos, reasignamos)

Para evitar el uso excesivo de malloc y copy en la gpu, hacíamos una reserva de memoria pensando en el futuro, utilizaremos un multiplicador (Línea 678), para reservar memoria extra para futuras iteraciones.

Para el array de células se utilizan dos arrays(Líneas 710 y 711), uno activo y otro para hacer la limpieza de células muertas, en cada iteración se intercambian sus papeles, es decir se guardan la células vivas del activo (en ese momento) en el de intercambio. (Líneas 887-906)

5. Creación y modificación de los kernels (Líneas 147-308)

Aunque todos los bucles del código secuencial han sido convertidos en kernels en la GPU cabe mencionar los kernels de reparto de comida y de limpieza de las células muertas:

- En el reparto de comida unimos en un único array los números aleatorios de la comida general y la de un punto específico y la mandamos a la gpu donde hará un atomicAdd en los sitios correspondientes.(Líneas 166-174).

- El bucle de limpieza de células muertas, utiliza los dos vectores de células y utilizando un atomicAdd cada hilo (si su célula está viva) recibe una posición en el nuevo vector.

Cada kernel tiene un argumento extra que delimita la posición máxima del array sobre el que trabaja, para así si un kernel es lanzado con más hilos de los que necesita (esto se puede deber al tamaño fijo de los bloques) estos hilos extra no accedan a posiciones de memoria ilegales.

2. Detalles sobre el proceso seguido

1. Empezamos dividiendo la ejecución del código entre CPU y GPU bucle por bucle de forma básica para obtener una versión paralelizada a excepción del reparto de comida y la eliminación de células muertas. Se copian entre CPU y GPU todas las estructuras de memoria en cada "kernel"
2. Paralelizamos el reparto de comida de la forma explicada en el punto 1.5.
3. Ajustamos los movimientos de memoria para minimizarlos aprovechando el hecho de que en ocasiones no era necesario traer la memoria de vuelta a cpu entre kernels
4. Refactorización de la gestión de memoria de la GPU para evitar "malloc" y "memCpy" innecesarios.
5. Mediciones de tiempo básicas para determinar qué bucles mejorar
6. Mejora de aquellos bucles que más tiempo de ejecución consumían, estrechamente relacionado con operaciones atómicas y reservas de memoria (Mejora vista en el punto 1.4).
7. Paralelizamos la limpieza de células muertas de la forma explicada en el punto 1.5.
8. Cambios y optimización de la geometría de grids y bloques (Mejora en el punto 1.2)
9. Desarrollo de la paralelización de movimientos de memoria y ejecución de kernels mediante streams.

Durante todo este proceso hemos utilizado nvprof (y su versión gráfica nvvp) para comprobar que kernels y que funciones de cuda son las que más estás tardando y cuda-memcheck para arreglar problemas de memoria y copia de datos a la gpu que hemos tenido durante el proceso.

3. Material previo utilizado

Incluyendo ideas sacadas de prácticas anteriores de compañeros, comentarios en el grupo de Discord que hayan sido de ayuda, información compartida con otros alumnos, comentarios sobre otras fuentes consultadas con citas a la bibliografía incluida en el apéndice, etc.

Por la diferencia conceptual entre esta práctica y las anteriores, no hemos podido aprovechar ningún recurso de ellas a excepción de la forma de crear la paralelización de la generación de comida cuya idea la hemos recuperado de OpenMP, pese a que en aquel momento no pudimos llevarla a caba por los errores de suma al reordenar los float.

Para la realización de la práctica utilizamos principalmente las diapositivas de la asignatura y nvidia y la documentación oficial de nvidia. Aunque también hemos investigado, pero no aplicado en la práctica, fuentes como stackoverflow y el subreddit r/CUDA.

Gracias a discord arreglamos el error Invalid number of Results el cual nos ocurría por una mala gestión de los hilos en un kernel.

4. Aportaciones personales

Hemos realizado el 100% de la práctica de forma conjunta utilizando plataformas de comunicación por vídeo llamada y compartiendo la pantalla. De esta forma los dos hemos sido partícipes en todo el proceso desde el inicio hasta el final del proceso de desarrollo del programa. La forma de trabajar se basó en atacar los bucles individualmente y hacer un brainstorming de ideas, una vez llegábamos a una solución razonable, la implementamos.

5. Bibliografía

[1] A. G. Escribano y Y. T. de la Sierra,"CUDA - Modelo de programación" 2020,
https://campusvirtual.uva.es/pluginfile.php/1283853/mod_resource/content/1/ttrasp_CUDA_2.pdf
Información básica sobre el uso de CUDA.

[2] A. G. Escribano y Y. T. de la Sierra,"CUDA - Técnicas básicas de optimización en CUDA" 2020,
https://campusvirtual.uva.es/pluginfile.php/1296301/mod_resource/content/3/OcupacionCoalescencia.pdf
Formas de optimizar el cómputo en tarjetas Nvidia con CUDA.

[3] A. G. Escribano y Y. T. de la Sierra,"Streams y Visual Profiler en CUDA" 2020,
https://campusvirtual.uva.es/pluginfile.php/1300470/mod_resource/content/3/sesion4.pdf
Paralelismo de kernels CUDA y herramientas de debug.

[4] NVIDIA Corporation, "Streams and Concurrency Webinar", 2020,
https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf
Información sobre el uso de Streams en la paralelización de "kernels"

[5] NVIDIA Corporation, "CUDA Documentation", 2020
http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/modules.html
Documentación sobre la tecnología CUDA

6. Figuras

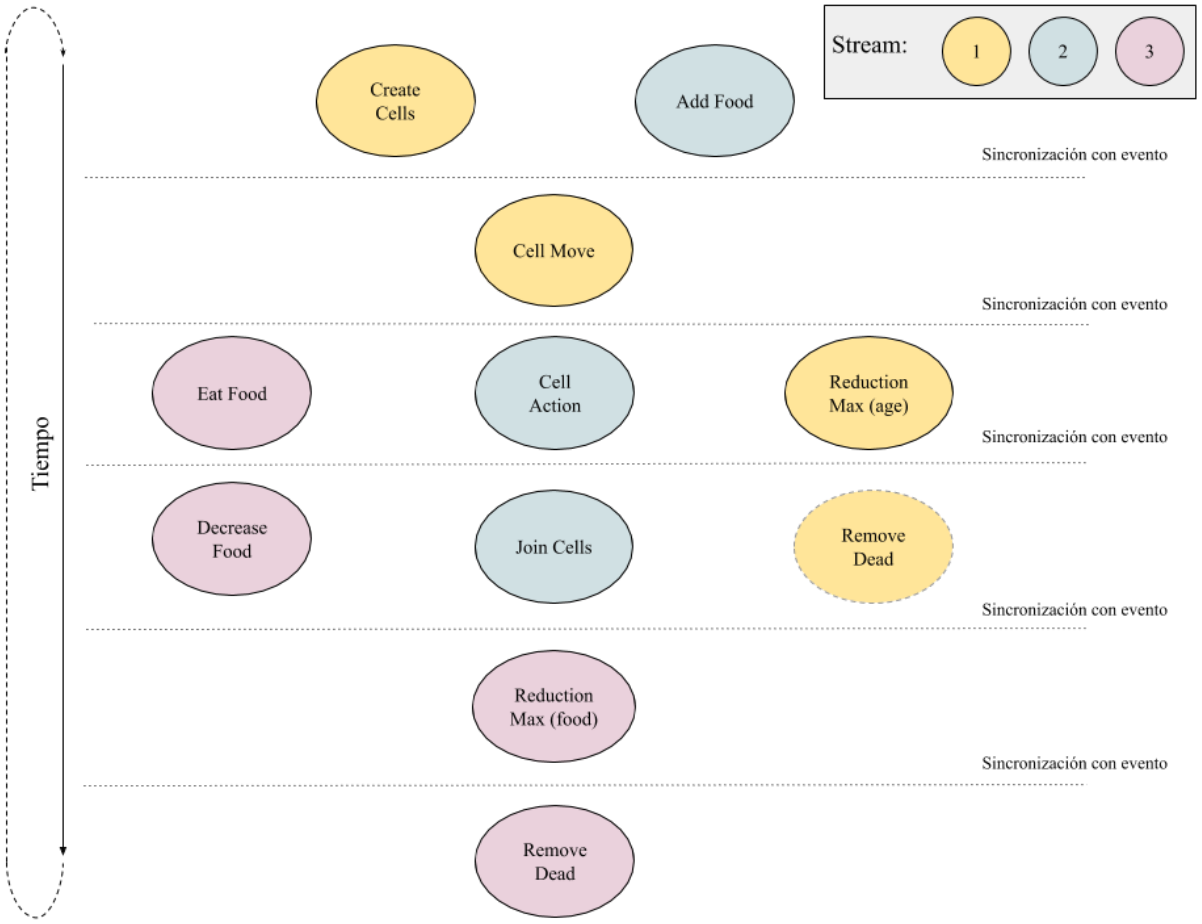

Figura 1. Grafo de los streams a lo largo del tiempo de ejecución