UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

HPC Laboratorio 2: Paradigma SIMT - Cuda

Nombres: Ekaterina Cornejo Contreras
Felipe Villalobos Padilla
Rut: 20.187.903-5
20.139.310-8
Profesor: Fernando Rannou Fuentes

30 de Octubre de 2022

Tabla de contenidos

1. Introducción

El proceso de paralelización consiste en transformar un programa secuencial en una nueva versión concurrente semánticamente equivalente. Este proceso puede ser realizado de forma automática o manualmente programando el algoritmo en un lenguaje de programación paralelo que permita especificar tareas u operaciones a ejecutar de forma concurrente, de manera que la finalidad es disminuir el tiempo de procesamiento mediante la distribución de tareas entre los procesadores disponibles. CUDA (Compute Unified Device Architecture) particularmente, es una arquitectura para cálculo paralelo con un conjunto de instrucciones con soporte de lenguajes de alto nivel como lo es el caso de C, que permite paralelismo a nivel de granularidad muy fino.

En el siguiente laboratorio se presentará el proceso de paralelización empleando CUDA, sobre la simulación de la difusión de una ola en un medio a partir de un estado inicial, empleando la ecuación de Schroendinger, donde la solución.

Objetivo

El objetivo del siguiente informe es presentar el desarrollo y resultados generados al implementar un simulador paralelo de la difusión de una onda según la ecuación de Schroendinger, empleando CUDA como tecnología de paralelización, analizando además el rendimiento de este en distintos casos.

2. Desarrollo

2.1. Contexto

Se solicita simular cómo se difunde una ola en un medio, a partir de un estado inicial, donde para iniciar se discretizará el dominio de difusión en una grilla de N × N celdas. En particular, la matriz H contendrá la altura de la onda en cada posición de la grilla. Además, como el proceso de difusión es dependiente del tiempo, H_{ij}^{t} es la altura, ya sea positiva o negativa, de la onda en la celda (i, j), en tiempo t. La ecuación de Schroendinger dice que:

$$H_{i,j}^{t} = 2H_{i,j}^{t-1} - H_{i,j}^{t-2} + c^2\Big(\frac{dt}{dd}\Big)^2\Big(H_{i+1,j}^{t-1} + H_{i-1,j}^{t-1} + H_{i,j-1}^{t-1} + H_{i,j+1}^{t-1} - 4H_{i,j}^{t-1}\Big) \tag{1}$$

Figura 1: Ecuación de Schroendinger

Donde "c" es la velocidad de la onda en el medio, "dt" es el intervalo de tiempo con que avanza la simulación y "dd" es el cambio en la superficie. Para este laboratorio, usaremos **c = 1.0, dt = 0.1, y dd = 2.0.**. Además de esto, se deberán considerar las condiciones de borde:

La condición de borde para este problema es cero, es decir, la altura en las celdas de los bordes del dominio permanece siempre con valor cero ($H_{0,j}^{t} = H_{i,0}^{t} = H_{N-1,j}^{t} = H_{i,N-1}^{t} = 0$ Para todo i,j,t).

También, la iteración para t = 1, es la siguiente ecuación:

$$H_{i,j}^{t} = H_{i,j}^{t-1} + \frac{c^2}{2}\Big(\frac{dt}{dd}\Big)^2\Big(H_{i+1,j}^{t-1} + H_{i-1,j}^{t-1} + H_{i,j-1}^{t-1} + H_{i,j+1}^{t-1} - 4H_{i,j}^{t-1}\Big) \tag{2}$$

Figura 2: Ecuación para primera iteración

2.2. Programación

2.2.1. Secuencial: A partir del laboratorio anterior

Para el desarrollo de la simulación, se inició generando el código con las condiciones antes mencionadas de manera secuencial (previo a cualquier tipo de paralelización).
Para esto inicialmente se generó el ingreso de los parámetros que serán requeridos por consola (N = tamaño de la grilla,x = tamaño de bloque en x, y = tamaño de bloque en y, T = número de pasos, f = archivo de salida).
Tras esto se procede al llenado inicial de la grilla (matriz) de N x N con valores 0 en los casos borde y 20 en el resto de los puntos (posiciones) de la grilla. Una vez hecho esto se procede a realizar el cálculo de la ecuación para cada una de las posiciones de la grilla considerando el caso de la primera iteración, cuyo cálculo poseía características específicas (tal como se mencionó anteriormente) y tras esto se continúa hasta que se realicen el número de pasos solicitados por consola. Finalmente, se generará un archivo (.raw) con el resultado obtenido tras T iteraciones realizadas llamado "ejemplo.raw".

2.2.2. Paralelización: CUDA

En esta ocasión, para realizar el proceso de paralelización, se implementó la arquitectura CUDA, donde se genera una grilla, dividida en bloques, cuyo número de hebras asignados depende del los valores ingresados por consola con los nombres de "tamaño_bloque_en_X " y "tamaño_bloque_en_X " representando así la dimensión de los bloques a emplear. Esta grilla vendrá a dividir la matriz que representa la imagen a trabajar. De esta forma, cada uno de las celdas será trabajada por una sola hebra, dentro de un bloque de la grilla gracias al uso de CUDA, de manera que el programa es escrito de manera secuencial dentro del kernel, este kernel se ejecuta de forma paralela dentro de la GPU con el conjunto de hebras organizadas y agrupadas en los bloques generados, cumpliendo con el número de hebras correspondientes al ingreso por consola y distribuidas en la grilla, en este caso, con bloques de dos dimensiones (x e y).

De esta manera el cálculo de el número de bloques se obtiene de la siguiente manera:

- tamaño bloque en eje x = x
- tamaño bloque en eje y = y
- bloques en dimensión x = N / x
- bloques en dimensión y = N / y

Con N = tamaño de la grilla. Además de esto, en caso de que el número de bloques generado no alcance a ser suficiente para cubrir cada una de las celdas de la grilla (divisiones de N donde el módulo distinto de 0), se añadirá un bloque más, ya sea en el eje x o y, donde se requiera. Dado que se fijarán condiciones de borde para aquellas celdas cuyo número supere el valor de la grilla a tratar, no se considerarán cálculos en las secciones del bloque fuera de estos límites, por lo que no interferirá en el resultado final que se obtendrá.

Cabe destacar que inicialmente se genera tres vectores "H1", "H2" y "HAUX", los cuales son almacenados en la memoria del dispositivo, y a su vez, se generan tres vectores que son almacenados en la memoria compartida de la GPU, obteniendo así seis vectores a utilizar. Estos últimos tres vectores almacenados la memoria compartida de la GPU es fundamental para el manejo paralelo de la información, ya que las funciones Kernels utilizan esta memoria para el manejo de la información compartida. Esta memoria es asignada a través de "Cuda-Malloc". Tras realizar todos los procesos concernientes al cálculo de la ecuación de Schroendinger, los valores obtenidos que permanecen en la memoria compartida de la GPU son asignados a los vectores representados en CPU, esto solo al concluir el proceso.

```
float *H1 = (float *) malloc(N*N*sizeof(float));
float *H2 = (float *) malloc(N*N*sizeof(float));
float *HAUX = (float *) malloc(N*N*sizeof(float));
```

Figura 3: Vectores asociados a CPU

```
//////////////////////////////////////////////////////
//  INICIO:   Asignacion de memoria Device
//////////////////////////////////////////////////////

  float *d_H1;
  float *d_H2;
  float *d_HAUX;

  //Se asigna memoria para cada vector del device
  cudaMalloc( &d_H1, N*N*sizeof(float));
  cudaMalloc( &d_H2, N*N*sizeof(float));
  cudaMalloc( &d_HAUX, N*N*sizeof(float));

  //Se copia el vector del host al vector del device
  cudaMemcpy(d_H1, H1, N*N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_H2, H2, N*N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_HAUX, HAUX, N*N*sizeof(float), cudaMemcpyHostToDevice);
```

Figura 4: Vectores asociados a GPU

2.3. Memoria compartida

En cuanto al empleo de memoria compartida, esta fue empleada para almacenar la región local (es decir, el bloque que se esté procesando) donde cada una de las hebras trabajaron a medida que realizaron los cálculos de las zonas que le correspondían, por lo tanto, en lugar de almacenar toda la matriz en cada ocasión, solo fueron porciones de esta, correspondientes a las que disponían las hebras. Para esto se empleó un vector dinámico de memoria compartida "temp", definido de la siguiente manera:

```
int j, i;
extern __shared__ float temp[];
```

Figura 5: Vector de memoria compartida dinámico

En este vector se irán almacenando los resultados obtenidos por parte de la hebra en la celda que le corresponda, empleando índices locales al momento de ir almacenando los valores dentro de esta, y al finalizar, este resultado será almacenado nuevamente en la matriz general definida de manera global, H1.

Tras esto se emplea "cudaDeviceSynchronize()" para así esperar que las hebras secundarias ejecutadas finalicen (espera que finalice el dispositivo de cómputo). Esto se empleará para asegurar de que la GPU termine de ejecutar el kernel antes de que se detenga el temporizador.

```
    iMas1=HAUX[(i+1)*N+j];
    jMas1=HAUX[i*N+(j+1)];
    temp[(threadIdx.x*blockDim.x)+threadIdx.y]= 2*HAUX[i*N+j]-H2[

}

H1[i*N+j] =temp[(threadIdx.x*blockDim.x)+threadIdx.y];
```

Figura 6: Llamado a funciones

2.4. Rendimiento computacional

Para medir el rendimiento del programa generado, se empleará una tabla con los resultados del tiempo de ejecución y ocupancia de los kernel para cuatro casos: tamaños de grilla de 512x512, 1024x1024, 2048x2048 y 4096x4096. A continuación se ven los resultados obtenidos por consola y el consolidado de estos en formato de tabla comparativa, para estos resultados se emplearon tamaños de bloque de 16x16 y un total de 10000 iteraciones:

Tamaño de grilla	512x512	1024x1024	2048x2048	4096x4096
Tiempo de ejecución (Time spent)	669.30115 [ms]	2072.97534 [ms]	7714.63525 [ms]	32.000.07227 [ms]
Tiempo de ejecución (Wall-Clock)	0 [s]	2.0 [s]	8.0 [s]	32.0 [s]
Ocupancia kernels	100.00 %	100.00 %	100.00 %	100.00 %

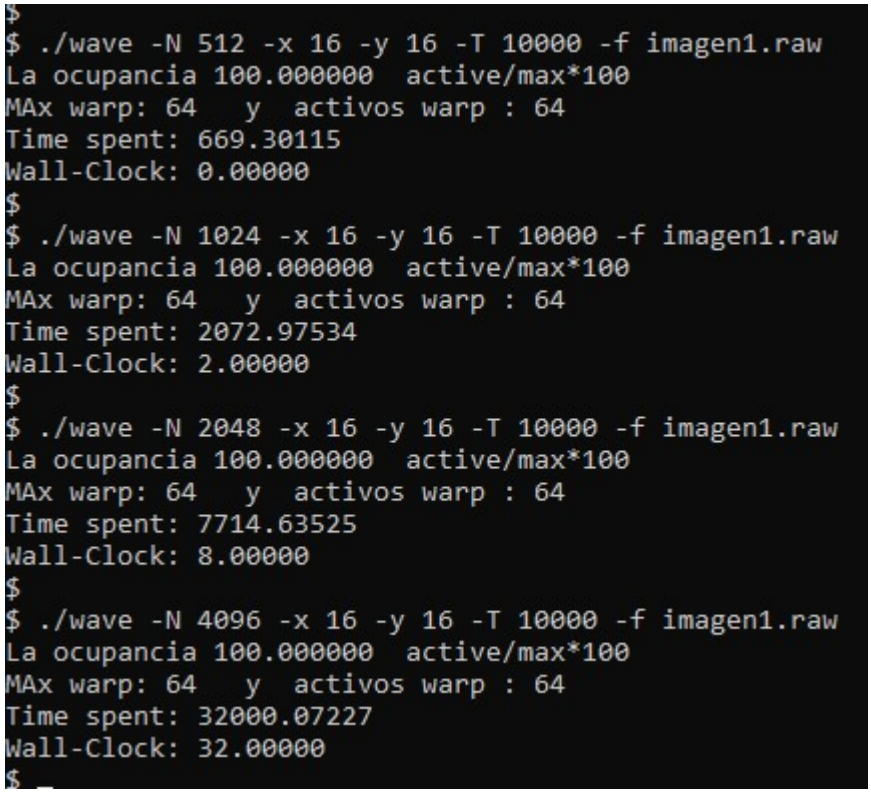

Figura 7: Resultados rendimiento

De estos resultados obtenidos, se puede observar como a medida que aumenta el tamaño de la grilla, mayor es el tiempo de ejecución, de manera que en un aumento del doble de N, lo que quiere decir N*N aumento de celdas en la grilla, implica un aumento de 4 veces el tiempo de ejecución (Wall-Clock), exceptuando el caso de la primera grilla que pasa de 0 segundos a 2 con la siguiente prueba. Además de esto, al tener igual cantidad de celdas como hebras empleadas, los 64 "warps" máximos disponibles son empleados en su totalidad en todos los casos, obteniendo una ocupancia de 1 , es decir, el 100

Ahora para una grilla de 2048x2048, se medirá el tiempo de ejecución para los siguientes tamaños de bloques: 16x16, 32x16 y 32x32.

Grilla: 2048 Tamaño de bloque	16x16	32x16	32x32
Tiempo de ejecución (Time spent)	2739 [ms]	2721 [ms]	2845 [ms]
Tiempo de ejecución (Wall-Clock)	3 [s]	2 [s]	3 [s]
Ocupancia kernels	100 %	100 %	100 %

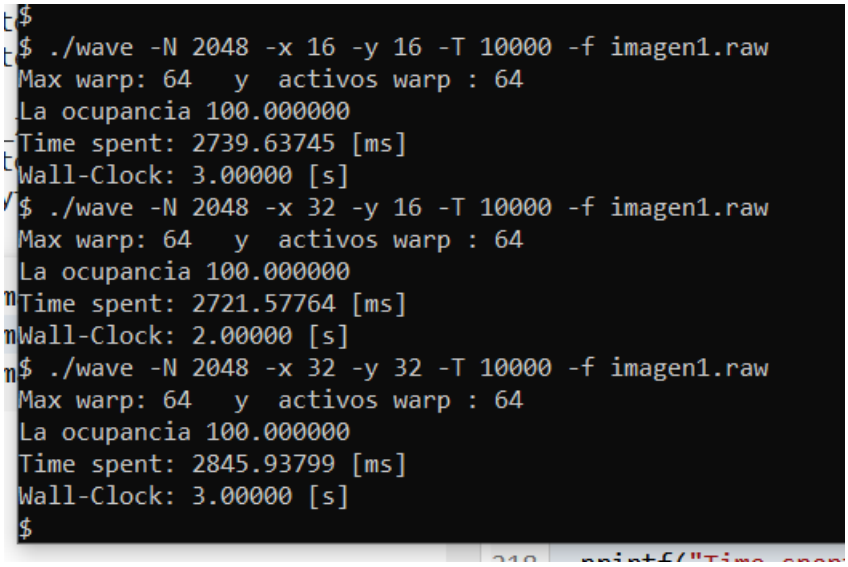

Figura 8: Resultados rendimiento grilla 2048X2048

Tal como se puede apreciar de los resultados obtenidos, el tiempo empleado para la grilla de 2048x2048 con bloques de tamaño 32x16, tomo tan solo 2 segundos, mientras que en los otros dos casos (16x16 y 32x32) tomaron 3 segundos, a pesar de ser mayores o menores que el caso mencionado. Esto se puede deber al manejo que tiene la GPU sobre el manejo de los datos y el trabajo de las hebras. Es posible que el hecho de que el tamaño de "x" e "y" al ser diferentes produzca menores demoras en las consultas a la caché u otros procesos. Además de esto, al igual que en el caso anterior, se puede observar una ocupancia del 100 % para todos los casos debido a que el tamaño de la grilla es dividido de manera exacta con el tamaño de los bloques a generar, lo que implica que todos los "warps" disponibles (64) estarán siendo empleados.

3. Conclusión

La paralización en GPU (empleada mediante la arquitectura CUDA) posee mejor rendimiento que en CPU, dado que la cantidad de núcleos que esta posee, siendo mucho mayor a los disponibles en una CPU promedio, aumentando el número de hebras disponibles, lo que a su vez aumenta el radio de cobertura para la paralelización a realizar, puesto que se pueden asignar tareas a un número mayor de hebras para que trabajen dividiendo el trabajo. Para este caso particularmente, esto permitió asignar el cálculo de cada una de las celdas de la grilla a una hebra en particular, lo que en una paralelización en CPU no habría sido posible para dimensiones de grilla grandes como las utilizadas durante el desarrollo de este informe. Además, el empleo de memoria compartida no significo grandes mejoras, puesto que el formato de la solución no requería almacenar información particular que fuera a ser requerida por la hebra durante el desarrollo de sus cálculos y que requiriera ser compartida.

Referencias

Rannou, F. (2022a). High Performance Computing, Departamento de Ingeniería en Informática LAB1 : SIMD- OpenMP.

Rannou, F. (2022b). High Performance Computing SIMD con hebras. OpenMP.

University of Colorado Boulder (2022). Fundamentals of parallel programming — Research Computing University of Colorado Boulder documentation. [Online] `https://curc.readthedocs.io/en/latest/programming/parallel-programming-fundamentals.html`.