HPC - LAB3: Paradigma SIMT - CUDA

Rubén Cavieres Carvajal
Universidad de Santiago de Chile

Se presenta un resumen del desarrollo del laboratorio 3 de la asignatura High Performance Computing dictado por el profesor Dr. Fernando Rannou, exponiendo resultados de las pruebas de rendimiento computacional sobre la aplicación paralelizada.

Keywords: HPC, CUDA, NVIDIA, Parallel, Schroedinger

El trabajo comienza con la creación de un algoritmo secuencial que simula el comportamiento de una onda bajo la ley de Schroedinger.

Luego se identifican las piezas de código que pueden ser paralelizables para utilizar en estas estrategias basadas en el paradigma SIMT-CUDA.

Finalmente se realizan pruebas de rendimiento de la aplicación utilizando las métricas:

- Tiempo de ejecución (wall-clock) para diferentes tamaños de grillas.
- Ocupancia
- Tiempo de ejecución para diferentes tamaños de bloques.

Algoritmo

Ecuaciones

El algoritmo se estructuró de tal forma que represente el planteamiento del matemático. Es decir, consideró las condiciones de la variable tiempo t para utilizar las ecuaciones con los siguientes métodos:

- `initializeSpace`: Para $t = 0$ inicializa el espacio (grilla) de trabajo, dado por el impulso en celdas centrales en grilla.
- `fillSpaceFirstStep`: Para $t = 1$ genera la primera iteración de la onda utilizando H^{t-1}.
- `fillSpaceTSteps`: Para $t > 1$ genera las sucesivas iteraciones de la onda, utilizando los estados del espacio H^{t-1} y H^{t-2}.

Paralelización

La estrategia de paralelización se basó en descubrir los bloques de códigos críticos potenciales para la utilización de CUDA.

Estos trozos de código son aquellos que construyen la ecuación de Schroedinger para el intervalo de tiempo $t >= 1$ (método `fillSpaceTSteps`):

```
int i = blockIdx.y * blockDim.y +
    threadIdx.y;
int j = blockIdx.x * blockDim.x +
    threadIdx.x;

waveSpace[N * i + j] = 2 *
    waveSpaceTMin1[N * i + j] -
    waveSpaceTMin2[N * i + j] + (c * c)
    * (dt/dd * dt/dd) * (waveSpaceTMin1[
    N * (i + 1) + j] + waveSpaceTMin1[N
    * (i - 1) + j] + waveSpaceTMin1[N *
    i + (j - 1)] + waveSpaceTMin1[N * i
    + (j + 1)] - 4 * waveSpaceTMin1[N *
    i + j]);
```

La estrategia adoptada fue la de utilizar las posiciones absolutas de las hebras dentro de la grilla completa, para así procesar cada porción del arreglo que contendrá imagen con expansión de onda a generar.

De esta manera, se obtienen los indices `i, j` a partir del identificador de bloque, tamaño de éste y la identificación de hebra actual.

Las matrices necesarias para operar la ecuación son entregadas por parámetro a la función, utilizando memoria compartida. Cada vez que termina de ejecutarse el método, se reasignan los valores de las matrices a sus nuevos estados `T`:

```
cudaMemcpy(waveSpaceTMin2_d,
    waveSpaceTMin1_d, N * N * sizeof(
    float), cudaMemcpyDeviceToDevice);
cudaMemcpy(waveSpaceTMin1_d, waveSpace_d
    , N * N * sizeof(float),
    cudaMemcpyDeviceToDevice);
```

Cada copia de esta información se concentra en dispositivo.

Rendimiento Computacional

El análisis de rendimiento se realizó bajo los siguientes parámetros y tamaños:

- Grillas = {512, 1024, 2048, 4096}
- Bloques = {10x10, 16x16, 32x16, 32x32}
- Tiempo = {300, 1000, 10000}

En un principio se ejecutaron las pruebas con el algoritmo generando imagen en formato RAW, pero por problemas de capacidad de almacenamiento de equipos computacionales y lentitud de ejecución, debió modificarse el algortimo para evitar la generación de las imágenes para cada caso.

A pesar de lo anterior se lograron ejecutar pruebas con algoritmo original (generando imagen) hasta la iteración `N = 4096, T = 300`, cuyas pruebas son expuestas en apartado.

Tiempo (Mismo Tamaño de Bloque)

Para los siguientes cálculos se utilizaron como tamaño de bloque `X = 10, Y = 10`:

- Con generación de la imagen de salida
- Sin generación de la imagen de salida.

Sin Generación Imagen Salida. El tiempo tomado para realizar la ejecución se representa en la siguiente figura:

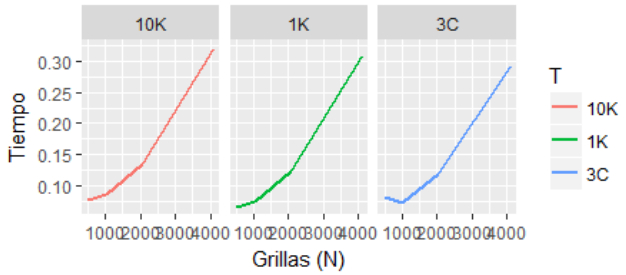

Figura 1. Tiempo ejecución para T = {300, 1000, 10000}

En los tres casos el tiempo aumenta aceleradamente en una proporción similar como se evidencia en curvas.

Sin embargo, se destaca que al aumentar el número de pasos `T`, el rendimiento del procesamiento siga siendo similar, demostrando la capacidad de la unidad GPU de procesar tareas de cómputo repetitivas.

Se obtienen los siguientes mínimos:

Como se ha dicho, estos mínimos se incrementan aceleradamente a medida que aumentan las iteraciones.

Cuadro 1
Mínimos de tiempo por iteraciones.

T (its.)	N (grilla.)	T (seg.)
300	1024	0,073585
1000	512	0,066303
10000	512	0,076833

Con Generación Imagen Salida. Ejecutando algoritmo con generación de imagen, se consigue siguiente rendimiento:

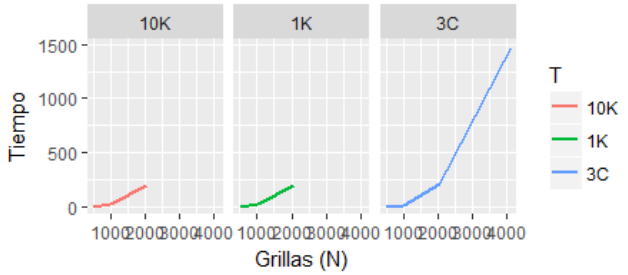

Figura 2. Tiempo ejecución para T = {2000, 4000, 8000}

No fue posible continuar con resto de pruebas, debido al llenado de espacio en disco duro de computador, producto de la imagen de gran tamaño que se estaba generando para `N = 4096, T = 1000`.

El comportamiento de la curva en comparación al anterior es equivalente, registrando los siguientes mínimos:

Cuadro 2
Mínimos de tiempo por iteraciones.

T (its.)	N (grilla.)	T (seg.)
300	512	0.244316
1000	512	0.284443
10000	512	0.296664

Los tiempos mínimos no experimentan mayores cambios, en comparación con los máximos:

Cuadro 3
Máximos de tiempo por iteraciones.

T (its.)	N (grilla.)	T (seg.)
300	4096	25 (min)
1000	2048	3,4 (min)
10000	2048	3,4 (min)

Dichos tiempos fueron excesivamente altos comparándolos a su ejecución sin almacenamiento de imagen, evidenciando el costo computacional que implica el acceso a memoria secundaria, por ser aquella más lenta dentro de todos los tipos de memoria del sistema.

Ocupancia

La métrica utilizada para calcular el *Device Occupancy* se expresa de la siguiente forma:

$$Ocupancia = \frac{Numero\ de\ warps\ activos}{Maximo\ Numero\ de\ warps}$$

Para dicha labor se utilizó la herramienta *CUDA Occupancy calculator*, que permite realizar el calcular la ocupancia de un kernel particular a partir de las siguientes entradas:

- Compute capability = 5,2 (A partir de información recogida para GPU NVIDIA Quadro M2000)
- Shared memory size Config (bytes) = 98304
- Global load caching mode = L2 only (cg)
- Threads per block = 1024 (considerando el tamaño máximo de bloques utilizado)
- Registers per thread = 18 (a partir de comando `nvcc -ptxas-options=-v`)
- Shared memory per block (bytes) = 88

Consiguendo la siguiente información:

Cuadro 4
Ocupancia GPU

Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	2
Occupancy of each Multiprocessor	**100 %**

Se alcanza máxima ocupancia con la solución construída.

Tiempo (Diferentes Tamaños de Bloques, Sin Imagen Salida)

Para este caso se utilizaron los parámetros:

- Grilla = {2048x2048}
- Bloques = {16x16, 32x16, 32x32}
- Tiempo = {300, 1000, 10000}

Curva generada fue la siguiente:

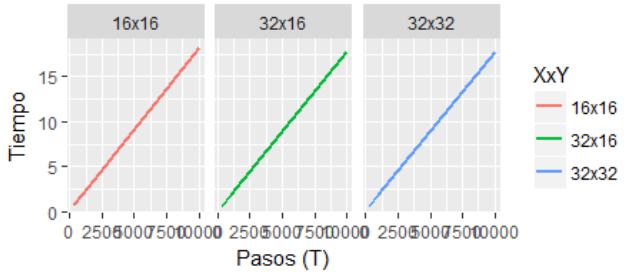

Figura 3. Tiempo ejecución para T = {300, 1000, 10000}

Cambiando el tamaño de los bloques se aprecia un comportamiento lineal similar al aumentar el valor T.

Conclusiones

El trabajo mostró el desempeño la implementación de la ecuación de Schroedinger, bajo el paradigma CUDA-SIMT.

Los gráficos generados muestran un comportamiento uniforme del desempeño para cada caso comparado, siendo en términos generales consistente la ejecución del algoritmo ante el aumento de carga y utilización de tamaños de bloques distintos.

Un problema experimentado en el análisis fue la ejecución de las pruebas con algoritmo generado imagen extensión RAW, que ante una iteración con valores altos, consumió tiempo y espacio en disco duro excesivo, obligando a quitar esta opción para continuar con análisis, descubriendo un considerable incremento en el rendimiento de la aplicación.