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


# Laboratorio 2: CUDA - Ecuación de Schroedinger

Integrantes: César Salazar
Jorge Sandoval

Curso: HPC

Modalidad: Magister

Profesor: Fernando Rannou

31 de Octubre de 2022

# Tabla de contenidos

# 1. Introducción

Por *High Performance Computing*, comprendemos la practica de añadir potencia a soluciones informáticas con tal de ser utilizadas para gestionar/procesar grandes volúmenes de datos a través de diferentes técnicas, estándares o recursos. Es por esto que, para comprender la importancia de la computación de alto rendimiento, sus conceptos y principios, se realizaran una serie de laboratorios para generar un acercamiento empírico a esta área, los que quedaran registrados en este documento.

Dando lugar a este laboratorio, es que se abordo *CUDA* (Nvidia, 2022), *Compute Unified Device Architecture*, el cual como *Openmp* permite desarrollar aplicaciones paralelas eficientes, sin embargo con grandes diferencias, tanto en la gestión de las hebras en su desarrollo(Cook, 2012), como puede ser el control de las secuencias de calculo que manejara cada hebra, estas, serán abordadas durante el desarrollo del laboratorio. Para esta segunda experiencia se abordará la ecuación de *Schroedinger* con tal de simular como se difunde una ola en un medio, utilizando *CUDA* como tecnologia de paralelizacion, con tal de contrastar el uso de sus diferentes parámetros y el que puede surgir al compararse con tecnologías como *Openmp*.

## 1.1. Objetivo

1. Implementar un simulador paralelo de la difusión de una onda según la ecuación de *Schroedinger*, usando *CUDA* como tecnología de paralelización.

## 2. Descripción del problema

Se define un medio acuoso en reposo (valor 0 inicialmente), el que se perturba con un pulso inicial definido por la proporción y generando el siguiente comportamiento:

$$H^0_{i,j} = 20 \quad \forall\, 0.4N < i < 0.6N, \;\; 0.4N < j < 0.6N$$

Figura 1: Sector de la grilla que afecta el primer pulso.

Es durante el primer paso de tiempo que su estado cambia a :

$$H^t_{i,j} = H^{t-1}_{i,j} + \frac{c^2}{2}\left(\frac{dt}{dd}\right)^2\left(H^{t-1}_{i+1,j} + H^{t-1}_{i-1,j} + H^{t-1}_{i,j-1} + H^{t-1}_{i,j+1} - 4H^{t-1}_{i,j}\right)$$

Figura 2: Comportamiento en el primer paso de tiempo.

Posteriormente los estados durante cada paso de tiempo t están definido por:

$$H^t_{i,j} = 2H^{t-1}_{i,j} - H^{t-2}_{i,j} + c^2\left(\frac{dt}{dd}\right)^2\left(H^{t-1}_{i+1,j} + H^{t-1}_{i-1,j} + H^{t-1}_{i,j-1} + H^{t-1}_{i,j+1} - 4H^{t-1}_{i,j}\right)$$

Figura 3: Comportamiento en un paso de tiempo t.

Tal que el medio acuoso tiene el siguiente comportamiento en cada instante de tiempo T:

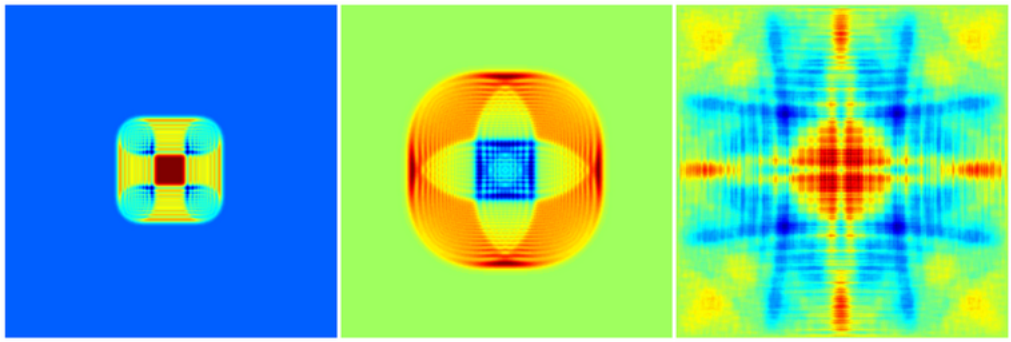

Figura 4: Resultados esperados para los pasos 300, 1000 y 10000 (para imagen de 256).

# 3. Estrategia de paralelización

En esta experiencia se pensó en segmentar los diferentes casos a resolver a diferencia de la experiencia previa. Esto da a lugar a SIMT (*Single Instruction Multiple Threads*), puesto que cada hebra resuelve la misma instrucción al momento de ser creada y operada. Con lo que se integran 3 llamados de kernel diferentes para los casos especiales de la función*Schroedinger* de tiempo cero y tiempo 1. Lo que permite desaparecer las bifurcaciones de flujo y acciones diferentes, quedando el calculo para un tiempo N separado. (Ver figuras 5 a 8) para revisar su implementación.

```
// FUNCIÓN: Ejecutar_schroedinger_t1cuda
// ENTRADA: Matriz tiempo 0, Matriz tiempo 1, largo del arreglo.
// PROCESAMIENTO: Calcula tiempo 1 de forma paralela.
// SALIDA: - .
__global__ void ejecutar_schroedinger_t1cuda(float *matriz_t0, float *matriz_t1, int N){
    int i, j;
    float c = 1.0, dt = 0.1, dd = 2.0;
    float iInf, iSup, jInf, jSup, ij_t1, cola;
    i = blockDim.x*blockIdx.x + threadIdx.x;  // global index x (horizontal)
    j = blockDim.y*blockIdx.y + threadIdx.y;  // global index y (vertical)

    if(i >= 1 && i < (N-1) && j >= 1 && j < (N-1)){
        iInf = matriz_t0[(i-1)*N+j];
        iSup = matriz_t0[(i+1)*N+j];
        jInf = matriz_t0[i*N+(j-1)];
        jSup = matriz_t0[i*N+(j+1)];
        ij_t1 = matriz_t0[i*N+j];
        cola = ((dt*dt)/(dd*dd)) * (iInf+iSup+jSup+jInf-(4*ij_t1));
        matriz_t1[i*N+j] = (ij_t1 + (((c*c)/2)*cola));
    }
}
```

Figura 5: SIMT: Schroedinger tiempo N

```
// FUNCIÓN: Ejecutar_schroedinger_tncuda
// ENTRADA: Matriz tiempo 0, Matriz tiempo 1, Matriz tiempo actual,largo del arreglo.
// PROCESAMIENTO: Calcula tiempo n de forma paralela.
// SALIDA: - .
__global__ void ejecutar_schroedinger_tncuda(float *matriz_t0, float *matriz_t1, float *matriz_rs, int N){
    int i, j;
    float c = 1.0, dt = 0.1, dd = 2.0;
    float iInf, iSup, jInf, jSup, ij_t1, ij_t0, cola;
    i = blockDim.x*blockIdx.x + threadIdx.x;  // global index x (horizontal)
    j = blockDim.y*blockIdx.y + threadIdx.y;  // global index y (vertical)

    if(i >= 1 && i < (N-1) && j >= 1 && j < (N-1)){
        iInf = matriz_t1[(i-1)*N+j];
        iSup = matriz_t1[(i+1)*N+j];
        jInf = matriz_t1[i*N+(j-1)];
        jSup = matriz_t1[i*N+(j+1)];
        ij_t1 = matriz_t1[i*N+j];
        ij_t0 = matriz_t0[i*N+j];
        cola = ((dt*dt)/(dd*dd)) * (iInf+iSup+jSup+jInf-(4*ij_t1));
        matriz_rs[i*N+j] = ((2*ij_t1) - ij_t0 + ((c*c)*cola));
    }
}
```

Figura 6: SIMT: Schroedinger tiempo 1

```
// FUNCIÓN: Ejecutar_schroedinger_t0cuda
// ENTRADA: Matriz tiempo 0, largo del arreglo.
// PROCESAMIENTO: Calcula primer pulso de forma paralela.
// SALIDA: - .
__global__ void ejecutar_schroedinger_t0cuda(float *matriz_t0, int N){
    int i, j;
    i = blockDim.x*blockIdx.x + threadIdx.x;  // global index x (horizontal)
    j = blockDim.y*blockIdx.y + threadIdx.y;  // global index y (vertical)

    if((i > 0.4*N && i <0.6*N) && (j > 0.4*N && j < 0.6*N)){
        matriz_t0[i*N+j] = 20.0;
    }
}
```

Figura 7: SIMT: Schroedinger tiempo 0

```
// FUNCIÓN: Copiar_matriz_cuda
// ENTRADA: Matriz tiempo 0, Matriz tiempo 1, Matriz tiempo actual,largo del arreglo.
// PROCESAMIENTO: Copia los valores de las matrices a la matriz del tiempo anterior tn -> tn-1 -> tn-2.
// SALIDA: -
__global__ void copiar_matriz_cuda(float *matriz_t0, float *matriz_t1, float *matriz_rs, int N){
    int i, j;
    i = blockDim.x*blockIdx.x + threadIdx.x;  // global index x (horizontal)
    j = blockDim.y*blockIdx.y + threadIdx.y;  // global index y (vertical)

    if(i >= 1 && i < N-1 && j >= 1 && j < N-1){
        matriz_t0[i*N+j] = matriz_t1[i*N+j];
        matriz_t1[i*N+j] = matriz_rs[i*N+j];
    }
}
```

Figura 8: SIMT: Mover valores a matriz del tiempo previo.

# 4. Rendimiento computacional

Uno de los aspectos que generaron mas curiosidad durante la elaboración de pruebas era la diferencia del comportamiento del tiempo de ejecución al modificar el tamaño de la grilla o el tamaño de los bloques. Para ello se dispuso de un paso de tiempo de 100000

iteraciones, modificando respectivamente el tamaño de la matriz para una grilla de hebras del mismo tamaño y modificando el tamaño de la grilla para una misma matriz.(Ver figura 10-11).

En el primer análisis se puede apreciar el porcentaje de uso de los Warps es del cien porciento, esto debido a que la cantidad de hebras calza con la cantidad de celdas utilizadas. Respecto al tiempo se evidencia un crecimiento notable al aumentar el tamaño de la matriz, esto quedando en evidencia al tomar aumenta 3-5-4 veces respectivamente entre iteración en el tiempo Wall-Clock, aunque estos tiempos son significativamente menores que los provistos por la versión de paralelización de Openmp en matrices de dimensiones aun menores (Ver figura 9).

Respecto a la misma grilla de 2048x2048, el porcentaje de uso de los Warps son totales a diferencia de la ejecución de 32x32 que es de 66.6 porciento y esto es debido a que los warps activos para esta ejecución son menores que los totales. Respecto al tiempo se evidencia un claro aumento en el momento en que la grilla aumenta su tamaño y esto puede deberse a el acceso de los datos en el momento que se disponen las hebras para realizar su tarea o la conformación de los bloques de hebras, esto queda en evidencia al aumentar poco mas del doble en el tiempo Wall-Clock. Dando lugar a que un bloque de hebras de menor tamaño en esta implementación otorgara una mayor rapidez de ejecución.

```
cesar@warmachine:~/Documentos/lab_HPC_waves_OpenMP$ make
gcc -Wall -fopenmp   -c -o wave.o wave.c
gcc -Wall -fopenmp   -c -o funciones.o funciones.c
gcc -Wall -fopenmp -o wave wave.o funciones.o -lm
cesar@warmachine:~/Documentos/lab_HPC_waves_OpenMP$ ./wave -N 256 -T 10000 -H 12 -f archivoSalida.raw

Ejecucion secuencial: 7.038603 (segundos)

Ejecucion paralela: 0.688842 (segundos)
cesar@warmachine:~/Documentos/lab_HPC_waves_OpenMP$ ./wave -N 256 -T 100000 -H 12 -f archivoSalida.raw

Ejecucion secuencial: 69.423960 (segundos)

Ejecucion paralela: 7.019957 (segundos)
cesar@warmachine:~/Documentos/lab_HPC_waves_OpenMP$ ./wave -N 256 -T 1000000 -H 12 -f archivoSalida.ra

Ejecucion secuencial: 697.591456 (segundos)

Ejecucion paralela: 70.608298 (segundos)
cesar@warmachine:~/Documentos/lab_HPC_waves_OpenMP$ ./wave -N 256 -T 10000000 -H 12 -f archivoSalida.r

Ejecucion secuencial: 6963.881099 (segundos)

Ejecucion paralela: 696.486219 (segundos)
cesar@warmachine:~/Documentos/lab_HPC_waves_OpenMP$
```

Figura 9: Tiempos de ejecución secuencial vs Openmp.

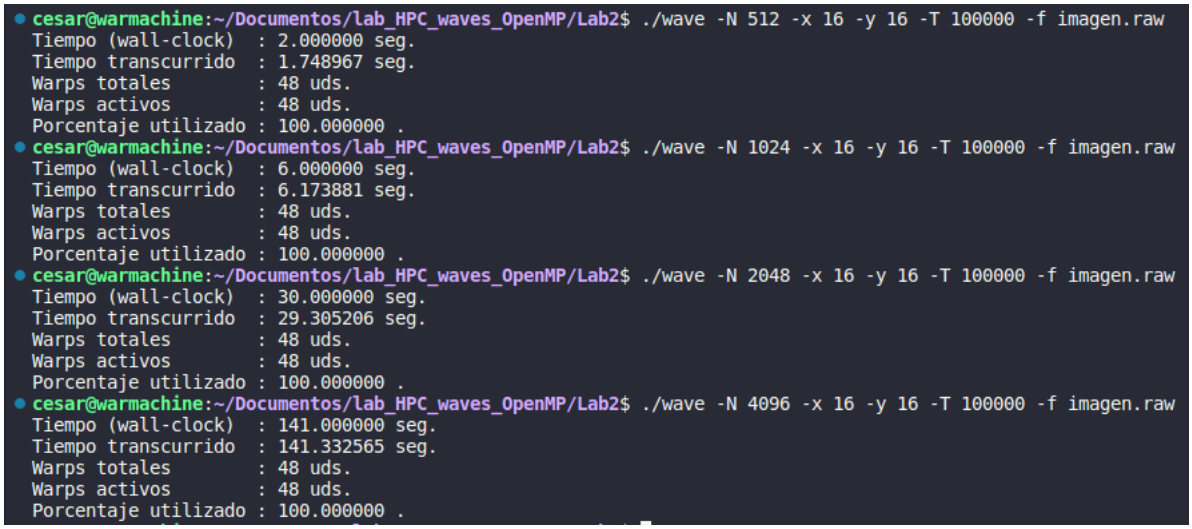


Figura 10: Resultados para diferente tamaños de grilla

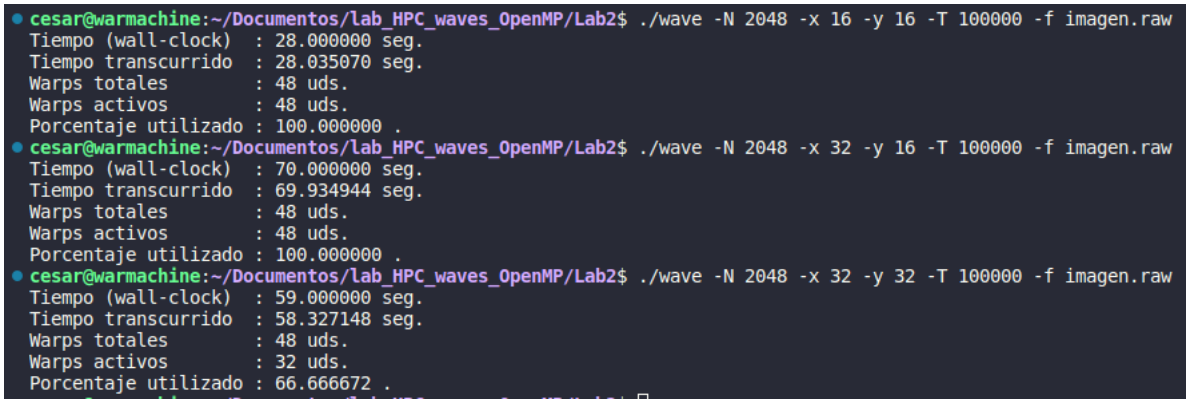


Figura 11: Resultados para diferentes tamaños de bloques

Las pruebas se ejecutaron en una tarjeta gráfica NVIDIA GeForce RTX 3080 (10GB GDDR6X -GPU Base Clock : 1440 MHz), procesador Ryzen 9 5950x con sistema operativo Ubuntu 22LTS. Otros aspectos de hardware importantes son 240 GB WDC WDS240G2G0A-00JH30 (SSD) y 32GB (2 x 16GB — DIMM DDR4-3200 RAM).

# 5. Conclusiones

Durante el desarrollo de este laboratorio, se puede evidenciar que se realizó correctamente la simulación de la difusión de una onda según la ecuación de Schroendinger utilizando la tecnología de paralelizacion *CUDA* en el lenguaje de programación C.

Se puede evidenciar que presenta ventajas frente a soluciones como las de la experiencia previa, con mayor potencia respecto al nivel de paralelización otorgado por *OpenMP*, esto gracias a la diferencia considerable de núcleos que posee una GPU de una CPU estandar, a los que se puede acceder, otorgar una carga de calculo a cada celda en particular y mantenerlas trabajando al mismo tiempo. Claramente la facilidad de la creación, gestión y control de hebras son aspectos en los que *CUDA* se diferencia, ya que no es tan sencillo gestionar las grillas, hebras y el calculo asociado a la celda que corresponde, la identificación de las ultimas requiere de mas cuidado y planificación.

Durante la experiencia se verifica que el uso de esta tecnología para grandes cálculos repetitivos, como los que ocurren al operar en una matriz, aprovecha en gran medida los recursos del hardware disponible para su procesamiento. Gracias a que posee con un mayor control de la gran cantidad de nucleos que posee una GPU, otorgando mayor potencia. Permitiendo realizar mas tareas en paralelo que una solución paralelizada en un procesador estándar con *OpenMP* y claramente mayor una ejecución secuencial.

Otro factor que surgió durante esta experiencia, fue la dificultad de instalar el software necesario para utilizar Cuda en un ambiente propio, puesto que la es nuestra primera experiencia configurando un ambientes para desarrollar con tarjetas gráficas. Pero que tras comprender lo necesario en documentación (Nvidia, 2022) correspondiente se comprendieron los recursos necesarios para ello.

# Bibliografía

Cook, S. (2012). Cuda programming: A developer's guide to parallel computing with gpu. https://www.sciencedirect.com/book/9780124159334/cuda-programming. 28 December 2012.

Nvidia (2022). NVIDIA Documentation. https://developer.nvidia.com/. October 2022.