



# Tarea 2

## Computación en GPU

### Juego de la vida

Integrantes: Nicolás Escobar Zarzar  
Vicente Muñoz

Profesor: Nancy Hitschfeld K.

Ayudantes: Diego García E.  
Vicente I. González

Fecha de realización: 28 de Abril de 2025  
Fecha de entrega: 6 de Junio de 2025  
Santiago, Chile

---

## Índice

1. Introducción .....	1
1.1. Descripción del problema .....	1
1.2. Resultados Esperados .....	1
2. Implementación .....	1
2.1. Implementación Secuencial .....	1
2.2. Implementación Paralela en CUDA .....	2
2.3. Implementación Paralela en OpenCL .....	3
2.4. Variaciones de configuración .....	3
3. Resultados .....	3
3.1. Máquina .....	4
4. Análisis de Resultados .....	5
4.1. <i>Speed-up</i> de las implementaciones paralelas .....	5
4.2. Estudio de tamaño de grillas .....	6
4.3. Estudio del tipo de arreglo .....	6
5. Conclusiones .....	6

# 1. Introducción

## 1.1. Descripción del problema

El Juego de la Vida de Conway es un popular autómatas celular creado por el matemático británico John Horton Conway en 1970. Se trata de un modelo matemático que simula el crecimiento y evolución de células en una cuadrícula bidimensional. Cada célula puede estar viva o muerta y su estado evoluciona según una serie de reglas predefinidas. El objetivo de esta tarea desarrollar una implementación en GPU del Juego de la Vida de Conway utilizando CUDA y OpenCL, y probar diferentes técnicas de optimización para mejorar el rendimiento de la implementación.

Las reglas que debe seguir el juego de la vida son las siguientes:

- Una célula nace de un espacio muerto si tiene exactamente 3 vecinos vivos.
- Una célula sobrevive en la próxima generación si tiene 2 o 3 vecinos.

## 1.2. Resultados Esperados

Se espera que la implementación en GPU sea muy superior comparada a la implementación en CPU, ya que El Juego de la Vida de Conway se puede reducir a resolver el subproblema de determinar si una célula está viva o no para las  $n \times m$  células de la grilla, el cual es un problema altamente paralelo en los datos y con una sola tarea para cada thread.

# 2. Implementación

## 2.1. Implementación Secuencial

Para la implementación secuencial (y también como base para el resto), se creó la clase `GameOfLife`, que cumple la función de ser una interfaz común a todas las implementaciones para facilitar las pruebas de rendimiento. Para el caso de CPU se hizo la clase `GameOfLifeSerial` que contiene 2 arreglos, `grid` y `nextGrid`, los cuales se declaran mediante una macro llamada `ARRAY_TYPE` para facilitar el cambio de tipo de arreglos sin necesitar reescribir toda la clase. Estos arreglos son de tipo `unsigned char` para ocupar solo 1 byte por célula y facilitar el cálculo de vecinos vivos, es necesario tener 2 porque en el primero se tiene el estado de la grilla actual y en el segundo estará el nuevo estado de la grilla una vez que cada célula determine si sigue viva o no. El tamaño de estos arreglos es determinado por el valor de las macros `GRID_ROWS` y `GRID_COLS`, esto se hizo así para tener dimensiones estáticas y conocidas en tiempo de compilación (estos valores cambian según variables entregadas a CMake).

El constructor de esta clase simplemente aloca memoria para sus 2 arreglos y el método `initializeRandom` se utiliza para crear una grilla pseudoaleatoria de células usando como semilla fija el valor 123 para tener resultados consistentes y reproducibles cada vez que se corre el programa. Cada valor de los arreglos se rellena con un 0 o un 1 dependiendo del resultado del generador de números aleatorios, con un 0 representando una célula muerta y un 1 representando una célula viva. El cálculo de los vecinos vivos se hace con el método `countAliveCells`, el cual suma los valores (0 o 1) que rodean a la célula para determinar cuantos vecinos vivos tiene.

La clase `GameOfLife` tiene un método `step` para calcular el estado de la grilla, para el caso de la implementación secuencial se procede de la manera siguiente:

1. Se hace un ciclo `for (int y = 0; y < GRID_ROWS; ++y)`, en el cual se determina la “coordenada” vertical de la célula en la grilla cuyo estado se requiere calcular. Esta coordenada es dada por el valor `y`
2. Se calcula la coordenada vertical de las células arriba y abajo de esta y se guarda en las variables `y0` (arriba) y `y2` (abajo). El cálculo de `y0` se hace con la operación  $((y + \text{GRID\_ROWS} - 1) \% \text{GRID\_ROWS})$ , esto debe ser así ya que, si por ejemplo, se está revisando la célula de la esquina superior izquierda (coordenadas 0,0), el resultado de  $y - 1$  sería  $-1$ , cuando en realidad debiese ser  $\text{GRID\_ROWS} - 1$ , ya que la grilla funciona de forma cíclica (es decir, el vecino superior de la célula en la esquina superior izquierda es la célula de la esquina inferior izquierda, etc), además la operación módulo no

está definida para valores negativos y la coordenada debe estar en el rango  $[0, \text{GRID\_ROWS})$  luego  $(0 + \text{GRID\_ROWS} - 1) \% \text{GRID\_ROWS} = \text{GRID\_ROWS} - 1$ . La coordenada de la célula de abajo sigue la misma lógica con aritmética modular para mantenerla dentro de los límites de la grilla  $(y+1) \% \text{GRID\_ROWS}$ .

3. Se hace otro ciclo `for (int x = 0; x < GRID_COLS; ++x)` para determinar los vecinos horizontales y diagonales, y definir finalmente que célula se está viendo en este momento, esto hará un “sweep” horizontal a través de las filas de la grilla, revisando cada fila por completo hasta la siguiente. El cálculo de las coordenadas horizontales y diagonales sigue la misma lógica modular de las coordenadas verticales con la variable `x` y `GRID_COLS`.
4. Dentro de este último ciclo `for` se hace el cálculo de los vecinos vivos mediante el método `countAliveCells`, el cual recibe 6 valores, que corresponden a las coordenadas necesarias para calcular cuantos vecinos hay:  $(x-1, x, x+1, y-1, y, y+1)$ . Dependiendo del resultado del método anterior, se decide si esta célula sigue viva (tiene 2 o 3 vecinos vivos), revive (está muerta y tiene 3 vecinos vivos) o muere (más de 3 vecinos vivos). Este resultado se guarda en las coordenadas  $(x, y)$  equivalentes de `nextGrid` sin alterar el arreglo `grid` original, puesto que cada célula debe decidir si vive o muere con el último estado de la grilla y no debe haber un estado intermedio hasta que todas decidan.
5. Finalmente se intercambian los punteros de `grid` y `nextGrid`, haciendo que la nueva grilla recién calculada sea la próxima grilla actual en otra iteración.

Notar que esta implementación de CPU hace uso de la macro `ARRAY_ACCESS(NAME, i, j)`, la cual abstrae la lógica del cálculo de la ubicación de la célula en la grilla según si está definida o no la macro `ARRAY_2D`, en caso de estar definida, se están utilizando arreglos de 2 dimensiones por lo que el acceso es directamente del estilo `nextGrid[y][x]`, si no, hace el cálculo de cual sería el índice real en el mapeo a un arreglo de 1 dimensión: `nextGrid[i * GRID_COLS + j]`

Para todas las implementaciones se creó una función utilitaria `benchmark` que recibe como argumentos un puntero `game` a algún `GameOfLife`, un entero con la cantidad de iteraciones y un string que apunta al path de un archivo en el cual guardar los resultados (tiempos). En todos los casos se probó con 16 iteraciones.

La lógica de todas las implementaciones en `benchmark` es similar:

1. Se inicializa la grilla con `initializeRandom`
2. Se inicia un ciclo `while` que corre hasta que acaben las iteraciones
3. Dentro del `while` se llama al método `game->step()` el cual hace una siguiente iteración del juego de la vida hasta que el ciclo `while` termine
4. Terminado el ciclo `while` se imprimen estadísticas y si se proveyó un nombre de archivo, este es cerrado y sus contenidos serán estadísticas de tiempo en cada iteración en formato csv con las columnas `iterationNumber`, `iterationDuration(s)` y `cellsPerSecond` que indican que iteración se corrió, cuanto duró dicha iteración en segundos y la cantidad de células que se procesaron por segundo basado en el cálculo de  $\text{GRID\_ROWS} * \text{GRID\_COLS} * \text{totalIterations} / \text{seconds}$ .

Para propósitos de comparación también se hizo una implementación paralela en CPU que sigue exactamente la misma lógica, pero distribuye el cálculo de las células en distintos threads, la cantidad de células procesadas por thread está dado por  $\text{GRID\_ROWS} * \text{GRID\_COLS} / \text{NUM\_THREADS}$ , donde `NUM_THREADS` es una macro definida en el archivo `game_of_life_parallel.hpp` cuyo valor por defecto es 12 (la cantidad de threads de la máquina 2 mostrada en resultados).

## 2.2. Implementación Paralela en CUDA

La implementación paralela en CUDA sigue la misma lógica que la implementación en CPU, pero hace uso del kernel `life_step_kernel1d` o `life_step_kernel2d` para calcular el siguiente estado de la grilla. El número de bloques a utilizar está dado por los valores de las macros `BLOCK_SIZE_X` y `BLOCK_SIZE_Y`, las cuales tienen por defecto el valor 32, pero pueden ser redefinidas en tiempo de compilación con flags de CMake. En las pruebas se utilizaron tamaños de bloque de tamaño 30, 32 y 35.

La cantidad de threads por bloque se calcula como  $(worldSize + blockSize - 1) / blockSize$  donde `worldSize` es la cantidad de células según la dimensionalidad de los arreglos y `blockSize` el tamaño de bloque final para la dimensión, es decir, si se trabaja con arreglos de 1 dimensión, entonces `worldSize` es `GRID_ROWS * GRID_COLS` y `blockSize` es `BLOCK_SIZE_X * BLOCK_SIZE_Y`, en caso contrario, si se usan arreglos de 2 dimensiones, `worldSize` es `GRID_ROWS` o `GRID_COLS` y `blockSize` es `BLOCK_SIZE_X` o `BLOCK_SIZE_Y` dependiendo de si son los bloques en  $X$  (las columnas `[i][k]` con  $k$  variable) o en  $Y$  (las filas `[k][j]` con  $k$  variable).

Dado que CUDA corre en la GPU, es necesario tener 4 arreglos (o en este caso 2 arreglos y 2 vectores), un par para el anfitrión (CPU) y otro par para el dispositivo (GPU).

En esta implementación con arreglos de 1 dimensión, cada thread se encarga de procesar  $(GRID\_ROWS * GRID\_COLS) / (blockDim.x * gridDim.x)$ , donde `blockDim.x` es la cantidad de threads que hay por bloque y `gridDim.x` es la cantidad total de bloques.

En el caso de 2 dimensiones, se deja que cada thread revise 1 célula, lo que es ineficiente a propósito para notar la diferencia (no se aplica *thread coarsening*<sup>1</sup>).

### 2.3. Implementación Paralela en OpenCL

La implementación paralela en OpenCL es casi completamente análoga a la de CUDA, pero en este caso tampoco se aplicó *thread coarsening* al kernel en 1 dimensión. De aplicarse *coarsening* este seguiría una lógica similar haciendo que cada thread procese una cantidad `worldSize / get_global_size(0)` de células.<sup>2</sup>

A diferencia de CUDA, OpenCL no permite directamente utilizar arreglos en 2 dimensiones, sin embargo, se opta por utilizar el tipo `image2d_t` como un análogo ya que este tipo es copiable de vuelta al anfitrión en un arreglo en 2 dimensiones.

### 2.4. Variaciones de configuración

Las opciones escogidas para hacer medición de su desempeño fueron:

- Usar tamaños de bloque tanto múltiplos de 32 como no múltiplos de 32. Específicamente 30, 32 y 35 (por dimensión).
- Usar arreglos de dos dimensiones en vez de un mapeo a arreglo de una dimensión

Los tamaños de grilla (producto de filas por columnas) probados variaban desde  $2^{13}$  hasta  $2^{31}$ .

## 3. Resultados

Para los tests se utilizó la siguiente máquina:

CPU Intel Core i5 10400 (6 cores, 12 threads), GPU NVIDIA GeForce RTX 4060Ti (4352 stream processors, 8 GB de memoria), 16 GB RAM.

Además de los resultados de cada implementación, las variaciones previamente mencionadas se ven en los gráficos siguientes identificadas por la leyenda, block  $n \times m$ , y 1d o 2d según corresponda.

---

<sup>1</sup>Solo por temas de tiempo, idealmente se debió aplicar *thread coarsening* en todas las implementaciones y el repo final tendrá esto. Cabe mencionar que al momento de correr las pruebas, el código de CUDA tenía un error que copiaba los datos del anfitrión al dispositivo 2 veces, reduciendo su rendimiento considerablemente

<sup>2</sup>Esta optimización está en el repositorio

### 3.1. Máquina

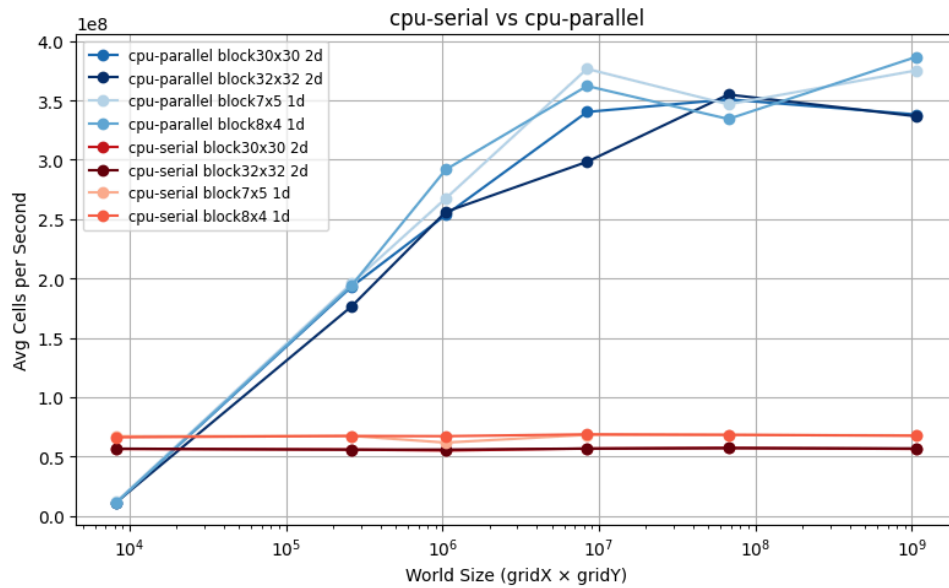


Figura 1: Velocidad de la implementación serial y paralela en CPU.

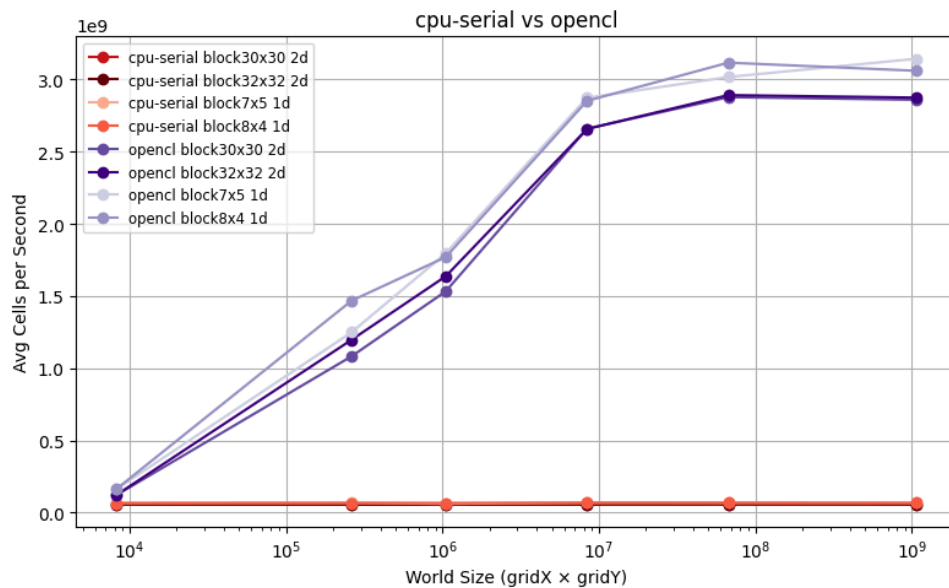


Figura 2: Velocidad de la implementación serial en CPU y paralela con OpenCL en GPU.

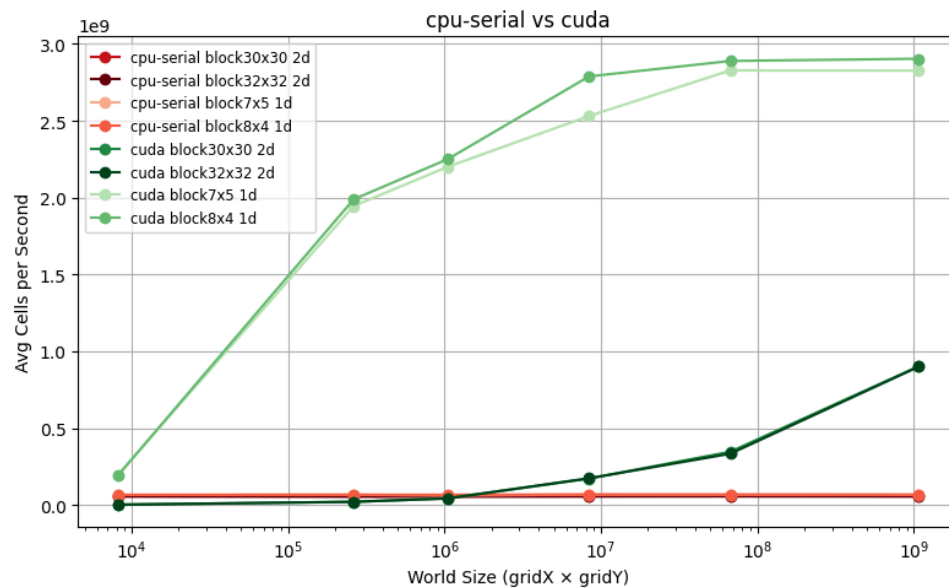


Figura 3: Velocidad de la implementación serial en CPU y paralela con CUDA en GPU.

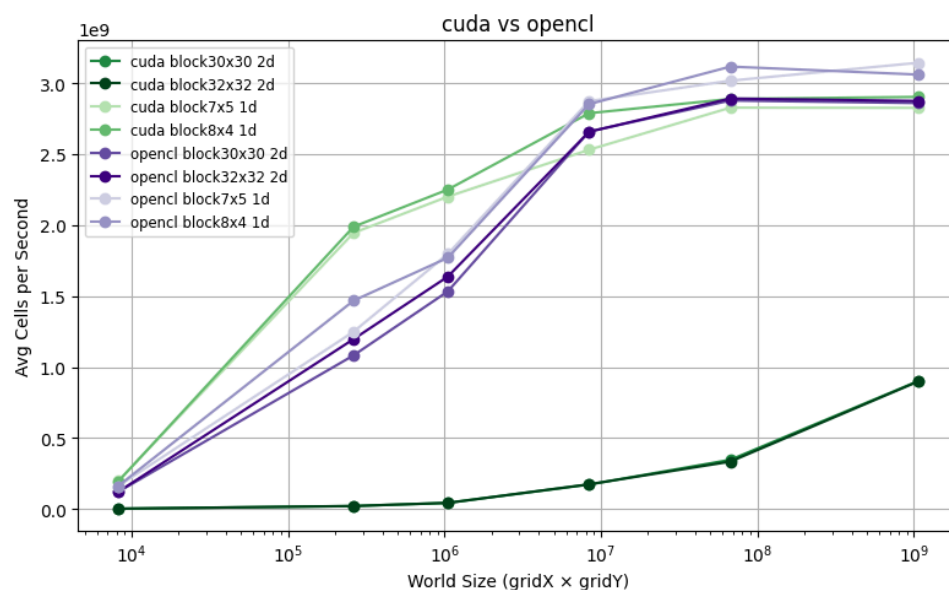


Figura 4: Velocidad de la implementación paralela con CUDA y OpenCL en GPU.

## 4. Análisis de Resultados

### 4.1. *Speed-up* de las implementaciones paralelas

La versión paralela en CPU es  $\sim 5 \times$  más rápida que el CPU serial, siempre en el rango de  $10^6$  celdas en adelante.

OpenCL en GPU rinde entre  $15 \times$  y  $40 \times$  frente al algoritmo serial de CPU, mejorando cuanto más grande es la grilla (sobre todo a partir de  $10^6$  celdas).

CUDA tiene una tendencia similar a OpenCL salvo en el caso 2d, pero esto fue debido a un error de implementación detectado después y no a CUDA en si. También la forma de copiar no es la ideal.

Estos resultados se encuentran dentro de lo esperado ya que al ser un algoritmo donde no hay tantas operaciones complejas, pero si múltiples accesos a datos, este se beneficia enormemente de la paralelización.

#### 4.2. Estudio de tamaño de grillas

En casi todos los casos la versión paralela es más rápida, cuando se compara la versión en CPU serial vs la paralela se puede observar que con un tamaño de alrededor de  $10^4$  y  $10^5$  celdas la versión serial es más rápida, probablemente debido al overhead de la creación de los threads o lectura y escritura que no supera los beneficios del cálculo en paralelo.

A medida que la grilla crece, el algoritmo paralelo se vuelve más eficiente, siguiendo una tendencia sublineal.

Para los tamaños de grilla probados, es decir, el rango entre  $2^{13}$  y  $2^{31}$ , siempre conviene usar GPU por sobre CPU, es posible que con tamaños más pequeños esto no se cumpla, y se dé un caso similar al que se ve en el gráfico comparativo de CPU serial vs paralelo, donde el overhead de copiar datos e iniciar threads sea mayor al beneficio de paralelizar.

#### 4.3. Estudio del tipo de arreglo

Para CPU paralelo, en realidad las cuatro series (dos bloques 2D y dos bloques 1D) casi se solapan, pero los arreglos de 1 dimensión suelen ir levemente mejor. Esto también se cumple en GPU como se puede notar con el rendimiento de OpenCL, exceptuando el caso de CUDA que como se mencionó anteriormente contenía un error al momento de correr las pruebas.

### 5. Conclusiones

El juego de la vida de Conway es un problema altamente paralelo en los datos que casi no tiene partes secuenciales, por lo que es idóneo para ser simulado en GPU llegando a speedups de casi  $\sim 50 \times$  respecto a la implementación serial en CPU.

El juego de la vida sigue una tendencia de *speedup* sublineal y en general sigue la ley de Gustafson al ser un problema con trabajo por thread fijo y un tamaño variable.