Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

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: 8 de Junio de 2025
Santiago, Chile

Índice

1. Introducción

1.1. Descripción del problema

El Juego de la Vida de Conway es un popular autómata 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 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) e `y2` (abajo). El cálculo de `y0` se hace con la operación `((y + GRID_ROWS - 1) % 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 `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, GRID_ROWS) luego (0 + GRID_ROWS - 1) % GRID_ROWS = 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) % 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 GRID_ROWS * GRID_COLS * totalIterations / 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 GRID_ROWS * GRID_COLS / 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 81 (9x9,

no múltiplo de 32) y 256 (16x16, producto múltiplo de 32). La cantidad de threads por bloque se calcula como `(worldSize + (blockSize + CELLS_PER_THREAD) - 1) / (blockSize + CELLS_PER_THREAD)` donde `worldSize` es la cantidad de células según la dimensionalidad de los arreglos, `blockSize` el tamaño de bloque final para la dimensión y `CELLS_PER_THREAD` es un valor que indica cuantas células debe calcular cada thread, ya que hacer que 1 thread calcule 1 célula puede causar un overhead al crear demasiados threads que hacen una operación muy simple. 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). El valor de `CELLS_PER_THREAD` se define como la raíz cuadrada del tamaño de bloque (por esa razón se escogen 9 y 16, ya que ambos son cuadrados perfectos). Las células que un thread dado revisa comienzan en el índice `idx` del arreglo:

```
idx = (blockIdx.x * blockDim.x + threadIdx.x) * cells_per_thread
```

Y terminan en el índice

```
idx + cells_per_thread - 1
```

Las coordenadas (x, y) de las células a revisar se calculan como

```
x = idx % width
```

```
y = idx / width
```

Luego los vecinos se calculan con aritmética modular, muy similar a como se hace en CPU.

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 el caso de 2 dimensiones, se opera manera similar, pero las coordenadas (x, y) de la célula por la que debe comenzar a revisar este thread se calcula de forma directa como:

```
x = blockIdx.x * blockDim.x * cells_per_thread_x + threadIdx.x
```

```
y = blockIdx.y * blockDim.y * cells_per_thread_y + threadIdx.y
```

A diferencia de la versión en 1 dimensión, esta no revisa células contiguas en cada thread, si no que hace saltos de tamaño `blockDim.x` o `blockDim.y`. En teoría esto debiese afectar negativamente el rendimiento, sin embargo, el factor que más disminuye el rendimiento de CUDA en 2 dimensiones, es el hecho de tener que copiar los datos de la grilla por filas, ya que este kernel hace uso de grillas de tipo `unsigned char**`, esto es, punteros a punteros de `unsigned char`, por lo que no hay garantía de que los datos de distintas filas estén en memoria contigua.

2.3. Implementación Paralela en OpenCL

La implementación paralela en OpenCL es completamente análoga a la de CUDA, el valor de inicio desde el cual debe revisar un work item (thread) se obtiene como `get_global_id(0) + cells_per_thread`. Todo lo demás es exactamente igual.

A diferencia de CUDA, OpenCL no permite directamente utilizar arreglos en 2 dimensiones, por lo tanto, se opta por utilizar el tipo `image2d_t` como un análogo. Este tipo representa una imagen con tantos píxeles como tamaño de grilla, para esta implementación, las células viven en el canal "rojo" de cada píxel y los píxeles tienen coordenadas (x, y) que se calculan de forma directa con `get_global_id(0) * cells_per_thread_x` y `get_global_id(1) * cells_per_thread_y`.

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 81 (9 por dimensión), 256 (16 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 resultados se usaron dos máquinas:

- **Máquina 1:** CPU AMD Ryzen 9 4900HS with Radeon Graphics 3.00 GHz (8 cores, 16 threads), GPU NVIDIA GeForce RTX 2060 with Max-Q Design (1920 cores, 6 GB de memoria), 16 GB de RAM.
- **Máquina 2:** 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.

3.1. Máquina 1

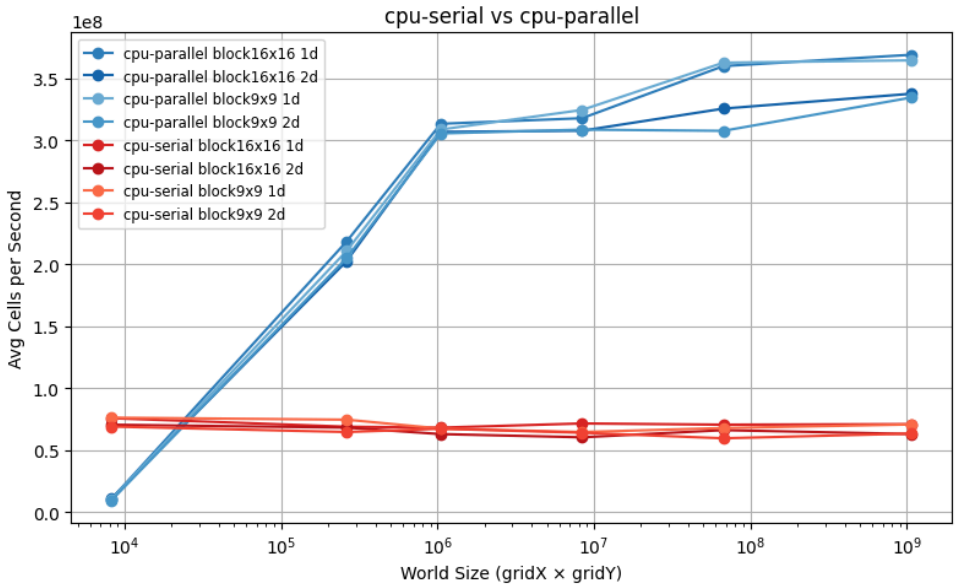

Figura 1: Velocidad de la implementación serial y paralela en CPU en Máquina 1.

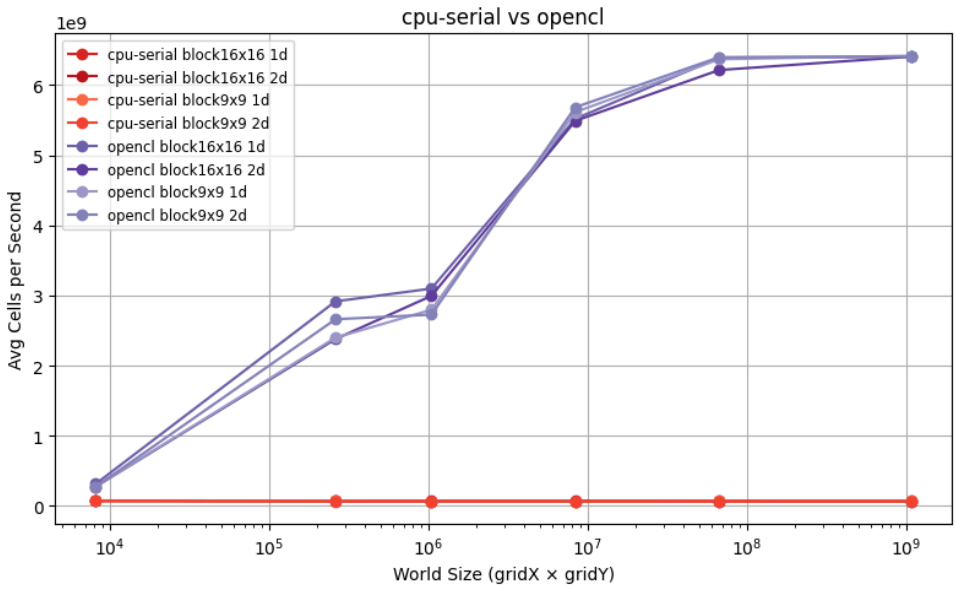

Figura 2: Velocidad de la implementación serial en CPU y paralela con OpenCL en GPU en Máquina 1.

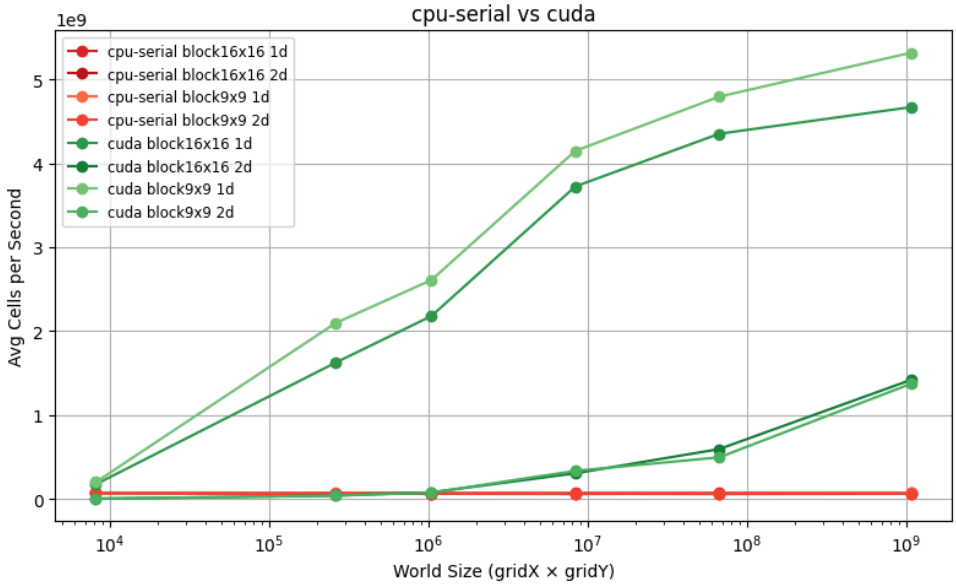

Figura 3: Velocidad de la implementación serial en CPU y paralela con CUDA en GPU en Máquina 1.

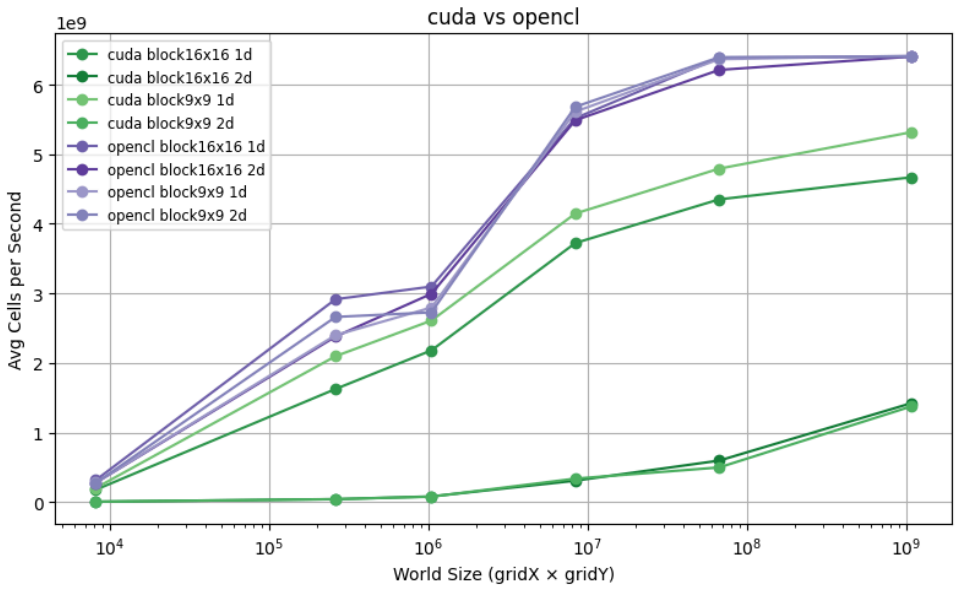

Figura 4: Velocidad de la implementación paralela con CUDA y OpenCL en GPU en Máquina 1.

3.2. Máquina 2

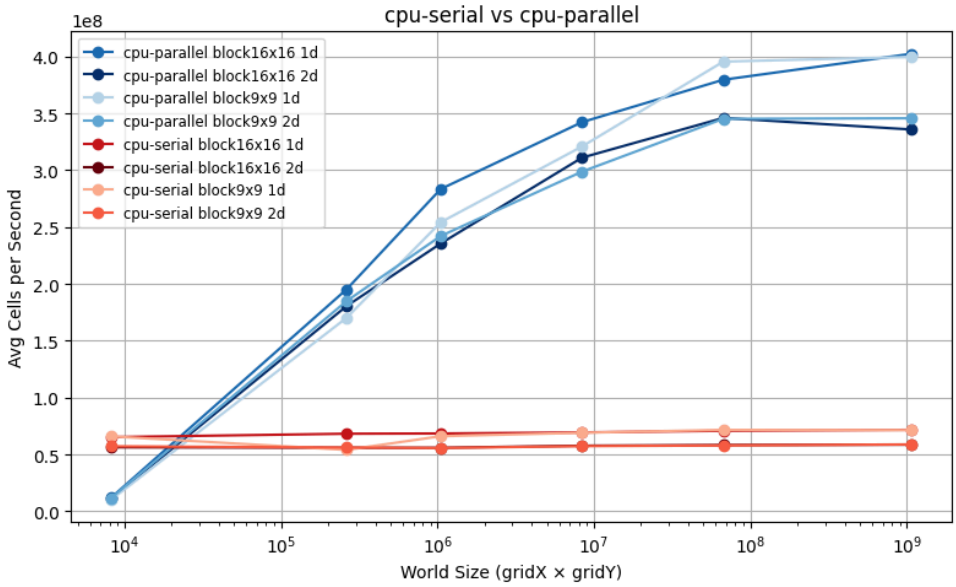

Figura 5: Velocidad de la implementación serial y paralela en CPU.

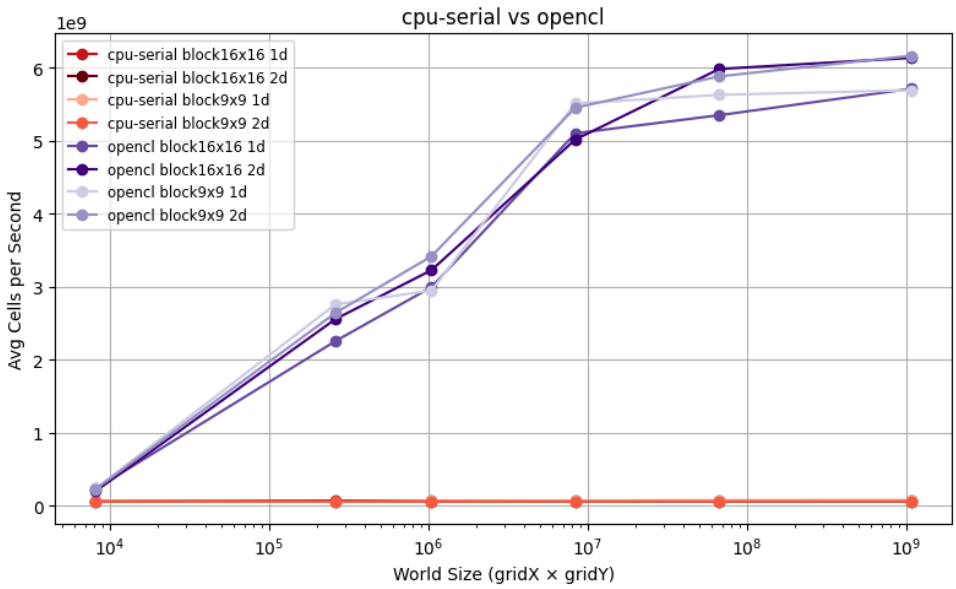

Figura 6: Velocidad de la implementación serial en CPU y paralela con OpenCL en GPU.

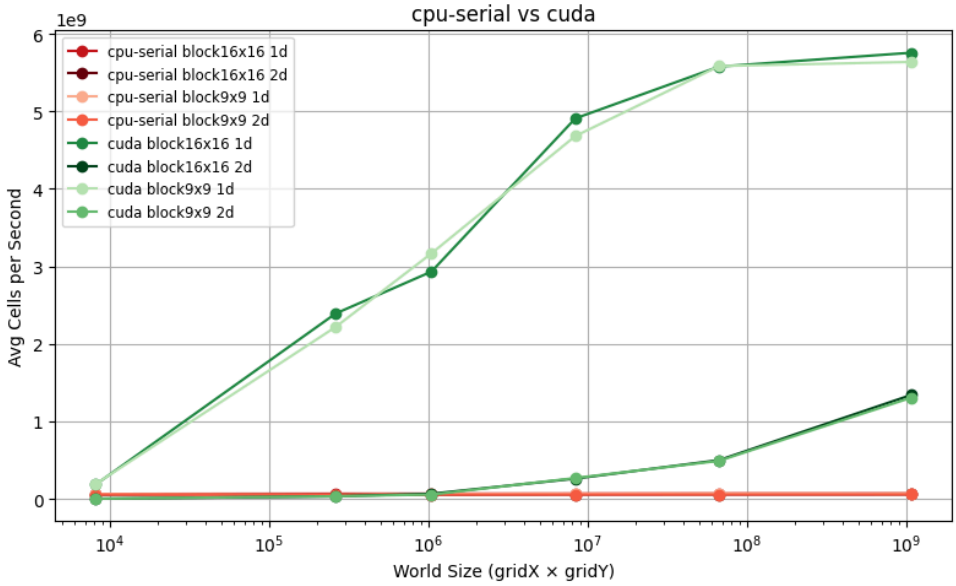

Figura 7: Velocidad de la implementación serial en CPU y paralela con CUDA en GPU.

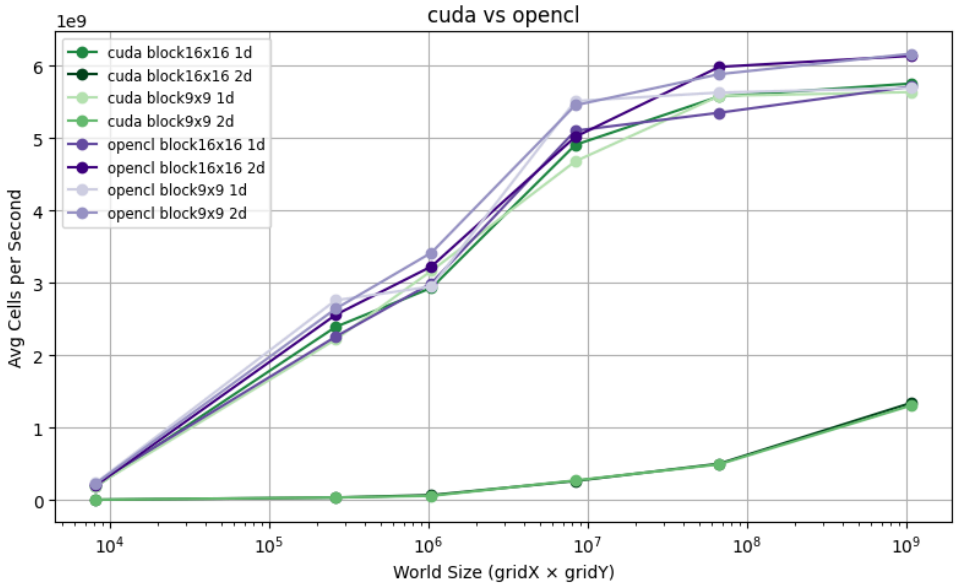

Figura 8: Velocidad de la implementación paralela con CUDA y OpenCL en GPU.

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× y 40× 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 en 2 dimensiones, pero esto fue debido a que la forma de copiar no es la ideal, ya que como fue mencionado previamente, se debe copiar el arreglo de arreglos por filas.

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.

Algo que no estaba en las expectativas fue que OpenCL tuviera un speedup superior a CUDA, pero esto puede deberse a que OpenCL fue compilado desde el código fuente y podría tener optimizaciones específicas para la configuración de hardware concreto en contraste con CUDA que es una librería compilada ya en código máquina.

En estos resultados también se evidencia que, en general, los tamaños de bloque 256 (16x16) que son múltiplos de 32, tienden a funcionar mejor, sobre todo en CUDA donde la diferencia es notoria a medida que crece la grilla, pero no tanto en OpenCL.

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 (el tamaño de bloque no tiene importancia para esta implementación en especifico). 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 pierde mucho tiempo en la copia de vuelta al anfitrión.

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. La dimensionalidad de los arreglos puede tener un impacto significativo en el rendimiento de la simulación y el tamaño de bloque influye levemente, sobre todo en CUDA, pero no de manera significativa.