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

Tarea 2
Juego de la vida

Integrantes: Andrés Gallardo Cornejo
Tomás Méndez Corvalán

Profesora: Nancy Hitschfeld K.

Ayudante: Diego García
Vicente Gonzáles

Índice

1. Introducción

El juego de la vida de Conway es un modelo matemático que simula el crecimiento y evolución de células en una cuadrícula bidimensional. Donde cada célula puede estar viva o muerta. Donde cada célula cambia su estado según las siguientes reglas:

1. Una célula nace si tiene tres vecinos vivos.
2. Una célula sobrevive si tiene dos o tres vecinos vivos.

Una implementación de la lógica del juego de la vida es bastante sencilla, para cierto tamaño de cuadrícula, se crean dos arreglos. uno capaz de almacenar el estado de cada célula, y otro que almacene su siguiente estado, luego se itera por el arreglo del estado actual donde por cada célula se calcula su siguiente estado en función de sus vecinos, y se guarda el resultado en el arreglo de la siguiente iteración, por último al terminar la iteración, se cambian las referencias a los dos arreglos.

Sin embargo la implementación propuesta, aunque fácil de implementar, es bastante ineficiente cuando el tamaño de la grilla crece, ya que la cantidad de células aumenta muy rápido y las iteraciones empiezan a tomar cada vez mas tiempo. Es aquí donde vienen en juego las GPU (Unidad de procesamiento de gráficos), las cuales son muy buenas para realizar muchas tareas en paralelo, debido a su gran cantidad de núcleos en comparación a una CPU, si tan solo se pudiera dividir el problema de calcular la siguiente iteración en problemas más pequeños, sería posible crear una implementación paralela que aproveche la cantidad de núcleos de la GPU para así calcular la siguiente iteración mucho más rápido que con una CPU.

Da el caso, que si se puede paralelizar el problema, podemos dividir el arreglo en secciones mas pequeñas, donde por cada célula creamos un hilo de GPU que calcule su siguiente estado, y así obtener tiempos mucho más rápidos comparados con los tiempos de la implementación inicial en CPU.

La implementación en paralelo será escrita tanto en OpenCL como en CUDA, donde cada implementación además tendrá dos implementaciones, una con bloques de dos dimensiones, es decir, en la GPU tenemos threads en dos dimensiones corriendo la solución, y una implementación con bloques de una dimensión, para ver si hay un beneficio el utilizar dos dimensiones. Por último también se revisará la diferencia de rendimiento al usar tamaños de bloques de múltiplos de 32 y no múltiplos de 32.

Se espera que la implementación paralela en GPU supere por mucho el rendimiento de la implementación serial en CPU, sin embargo con tamaños de grilla mas pequeños, quizás sea ventajoso utilizar una CPU, ya que el tiempo de trasladar la memoria de la RAM a la VRAM supere el beneficio de paralelizar. No se espera una diferencia grande de rendimiento al usar bloques de dos dimensiones ya que, al estar utilizando en si un arreglo de una dimensión, simplemente sería un bloque mas grande, en si, quizás sea mas lento. Por ultimo se espera que sea mas rápido con bloques de tamaño de múltiplos de 32, ya que 32 es una potencia de 2.

El entorno de pruebas fue el siguiente:

- OS: Garuda Linux Broadwing x86_64
- Kernel: 6.14.10-arch1-1
- CPU: AMD Ryzen 5 3600 (12) @ 4.208GHz
- GPU: AMD ATI RX 9070 XT (PCIe 4 x16, principal)
- GPU: NVIDIA GeForce RTX 2060 SUPER (PCIe 4 x 8, secundaria)
- Memory: 32012MiB DDR4 @ 3200MHz

Sin embargo, las pruebas con GPU se realizaron **exclusivamente con el dispositivo NVIDIA GeForce RTX 2060 SUPER** para mantenerlas consistentes.

2. Implementación

2.1. Implementación serial en CPU

Se decidió utilizar vectores de `char` para almacenar los estados de las células, por lo tanto se tienen dos vectores `cells` y `next_cells`, cabe notar que se estamos representando una grilla en un vector de una dimensión, la traducción de acceder a `(x,y)` en una dimensión es `(x + y*row_size)`, donde `row_size`, es el tamaño de las filas.

El tamaño de la grilla se obtiene mediante argumentos al ejecutar el programa, enteros que se almacenan en las variables `GRID_X` y `GRID_Y`, que indican la cantidad de columnas y la cantidad de filas respectivamente. Ya con estas dimensiones se inicializan los vectores con tamaño `GRID_X*GRID_Y` con todos sus valores en falso, por último se procesan de nuevo los valores del vector `cells`, por cada célula se genera un `float` entre 0 y 1, si el valor es menor `0.1` se inicializa esa célula como viva.

Esta implementación utiliza una grilla cíclica, es decir, si una célula se encuentra en un borde, y su vecino no existe, se busca el vecino en el otro extremo de la grilla, esto se realiza mediante la operación módulo, para revisar que no se salgan de las dimensiones, por último el trabajo de calcular la cantidad de vecinos vivos de una célula se delega a la función `get_neighbours`, la cual hace uso de la grilla cíclica.

El responsable de calcular la siguiente iteración es la función `game_of_life_cpu`, la cual itera por todas las células, calculando la cantidad de vecinos por célula, y si la célula esta viva o muerta se revisa que regla aplicar y guardar el resultado en la siguiente iteración, por último al finalizar el recorrido se intercambian las referencias de `cells` y `next_cells` con la función `swap` de la librería estándar.

La función `game_of_life_cpu` es invocada en el loop principal del programa cuando la opción elegida es CPU.

2.2. Implementación paralela en CUDA

Para la implementación en CUDA se implementaron 3 kernels, uno para cada implementación según las dimensiones de los bloques, y uno que aleatoriza los valores de las células.

El tamaño de bloque es recibido como argumento de la línea de comandos, junto a si se debe hacer uso de una o dos dimensiones.

Con una dimensión la cantidad de bloques se calcula de la siguiente forma: `grid_size = (GRID_X*GRID_Y + block_size - 1)/block_size`

De esta forma es seguro que sobran threads al contrario de que falten, ya que puede haber threads demás se debe revisar que al thread le corresponde trabajo.

Con dos dimensiones el cálculo es de la siguiente manera:
`grid_size_2d = ((GRID_X + block_size_2d.x - 1) / block_size_2d.x, (GRID_Y + block_size_2d.y - 1) / block_size_2d.y);`

Y block_size_2d es `(block_size, block_size)`

El kernel que aleatoriza las células solo es de una dimensión.

Para la memoria en la GPU, se hizo uso de dos punteros a `char`, `d_cells` y `d_next_cells`, para la generación actual y la siguiente respectivamente.

Estos punteros son inicializados mediante la función `setup_cuda_memory`, la cual recibe las dimensiones de la grilla, y reserva memoria suficiente en para cada arreglo en la GPU, en caso de error, se retorna un número negativo, y se cierra el programa.

2.2.1. Kernel aleatorio

Este kernel llamado `randomize_grid_cuda_kernel` recibe cuatro argumentos, el puntero al arreglo de células, las dimensiones de la grilla, y la semilla.

Primero se calcula el índice según la dimensión del bloque, su id, y la id del thread, si la id se sale de rango del arreglo, no se realiza trabajo. Si se encuentra en un índice válido se crea un estado curand, se inicializa y se genera un número aleatorio entre `0` y `1`, si el número es menor a `0.1`, se cambia el valor de la célula a `1`, para representar que se encuentra viva.

Por último después de invocar al kernel y espera que termine su ejecución se copia la memoria de vuelta a la RAM en el arreglo de `cells`. Esto en realidad no es necesario, pero ya si se quiere ver la interfaz gráfica es necesario, ya que el dibujado de las celulas depende del arreglo `cells`

2.2.2. Kernel 1D

De nombre `game_of_life_kernel`, este kernel recibe los argumentos, `char* d_cells, char* d_next_cells, int grid_x, int grid_y`, representando, puntero a generación actual de células en VRAM, puntero a siguiente generación en VRAM, cantidad de columnas de la grilla, cantidad de filas de la grilla.

No es necesario copiar información de la RAM a la VRAM antes de la ejecución, ya que el trabajo de generar las células fue realizado por el kernel anterior, y el arreglo de la generación actual se encuentra intacto.

Al ejecutar este kernel primero se obtiene la célula que le corresponde al thread igual que en el kernel de aleatorización, y se revisa que la célula que le corresponde exista, si existe, se realiza lo siguiente:

1. Se calculan y guardan las coordenadas de la célula en la grilla en dos enteros `x` e `y`, también se inicializa un entero que guarda la cantidad de vecinos vivos en 0.
2. Mediante un doble `for`, se exploran los vecinos de la célula, y sumando a la cantidad de vecinos, este `for` se realiza variando los valores de cada eje entre `eje -1`, `eje` y `eje+1`, por lo que es necesario revisar que no se encuentra revisando la misma célula, y saltar a la siguiente iteración cuando ocurra.
3. Se revisa si la célula se encuentra viva o muerta y se aplica la regla correspondiente según la cantidad de vecinos vivos calculada.

Después de la invocación es necesario copiar de vuelta el arreglo de `d_cells` de VRAM a `cells` en RAM, para así poder dibujar en caso de ser necesario, lo que si o si debe realizarse es cambiar los punteros de `cells` y `d_cells` para así en la siguiente invocación estar con la generación correcta.

2.2.3. Kernel 2D

Este kernel de nombre `game_of_life_kernel_2d`, recibe los mismos argumentos que el kernel anterior, más aún siguen la misma lógica pero con pocas diferencias.

Se calculan inmediatamente los valores de `x` e `y` según valores del bloque y del thread, y se revisa que estos valores se encuentren las dimensiones de la grilla, y se procede de la misma manera que en el kernel anterior.

Al igual que en el kernel anterior, es necesario copiar los datos de la VRAM a la RAM.

2.3. Implementación paralela en OpenCL

Dado que OpenCL no soporta la generación de números aleatorios de manera simple como CUDA, no se implementó un kernel para la inicialización aleatoria de las células.

La entrada del tamaño de los `work-groups` y el cálculo en base al `block-size` de entrada se calculó de manera distinta a CUDA, dado que se consulta por las limitaciones del dispositivo y en base a eso se ajustan los tamaños (ver Sección 6.6)

Para manejar la memoria se utilizaron 2 buffers del tamaño de la grilla:

cpp

```cpp
// Para el estado i
opencl_d_cells = clCreateBuffer(context,
                                CL_MEM_READ_WRITE,
                                GRID_X * GRID_Y * sizeof(char),
                                NULL,
                                &err_cl);
// Para el estado i+1
opencl_d_next_cells = clCreateBuffer(context,
                                     CL_MEM_READ_WRITE,
                                     GRID_X * GRID_Y * sizeof(char),
                                     NULL,
                                     &err_cl);
```

Los cuáles transferían sus datos a los arreglos `d_cells` y `d_next_cells` para dibujarlos con GLFW:

cpp

```cpp
clEnqueueWriteBuffer(queue, opencl_d_cells, CL_TRUE, 0,
                            GRID_X * GRID_Y * sizeof(char), cells.data(),
                            0, NULL, NULL);

clEnqueueReadBuffer(queue, opencl_d_cells, CL_TRUE, 0,
                    GRID_X * GRID_Y * sizeof(char), cells.data(), 0,
                    NULL, NULL);
```

2.3.1. Kernel 1D

El kernel de OpenCL para una dimensión se llama `game_of_life_kernel_opencl` y funciona de manera similar al de CUDA, recibiendo como parámetros `d_cells`, `d_next_cells`, `grid_x` y `grid_y`, donde `d_cells` y `d_next_cells` son los punteros a los arreglos de los estados de las células y los otros 2 corresponden a las dimensiones de la grilla.

Cada *work item* se encarga de procesar una célula y calcular su estado siguiente, obteniendo las coordenadas de 2 dimensiones que le corresponde usando el índice del *work item*. Se recorren los vecinos alrededor de la célula usando un doble for y en caso de llegar a salirse de la grilla, se vuelve por el lado opuesto, para luego aplicar las reglas del juego de la vida.

Al terminar cada ejecución del kernel, primero se lee el buffer `opencl_d_cells` y se copia en `cells` y después se intercambian los punteros de `opencl_d_cells` y `opencl_d_next_cells`.

2.3.2. Kernel 2D

La firma del kernel `game_of_life_kernel_2d_opencl` es idéntica a la anterior pues necesita las mismas variables para trabajar. Dado que se trabaja en 2 dimensiones, se calcula la posición de la célula usando los índices globales 0 y 1 del *work item*:

OpenCL

```opencl
int block_x = get_global_id(0); // coordenada X de este work-item
int block_y = get_global_id(1); // coordenada Y de este work-item
```

Luego, se aplican de manera similar al kernel 1D las reglas del juego de la vida.

2.4. Diferencias en configuración

La entrada del programa quedó definida como:

```
GameOfLife <UI> <GRID_SIZE_X> <GRID_SIZE_Y> <MODE> <2D> <DIM>
```

- `UI`: 1 si se desea visualización, 0 si no.
- `GRID_SIZE_X` y `GRID_SIZE_Y`: Los tamaños de la grilla del juego de la vida.
- `MODE`: El modo de ejecución, puede ser: CPU, OPENCL o CUDA.
- `2D`: 1 si se desea la optimización de 2 dimensiones, 0 si no.
- `DIM`: La dimensionalidad de los grupos de trabajo de OpenCL y grilla de CUDA.

Se realizaron experimentos en modo *headless* (sin visualización) midiendo la cantidad de células/segundo analizadas alcanzadas. Se probaron primero los 3 modos variando los tamaños de grilla entre 100 y 8000. Luego se compararon los resultados de CUDA y OpenCL con potencias de 2 como tamaños y dimensiones entre 1 y 16 (el máximo soportado por el dispositivo al ejecutar OpenCL). También se comparó el uso de 1 vs 2 dimensiones.

Al desarrollar con CUDA y OpenCL se pudieron observar las siguientes diferencias:

2.4.1. Dimensiones de trabajo

La más notoria fue la definición de los tamaños para los *work groups* y *work items*, pues el límite máximo que les da OpenCL a estas variables difiere del *grid size* y *block size* de CUDA, por lo que se optó por hacer "clamping" con respecto a **CL_DEVICE_MAX_WORK_GROUP_SIZE**. Esto implicó que al trabajar con 2 dimensiones, el tamaño máximo de *work group* fuese 16 x 16, a diferencia de CUDA donde se permitía una grilla de hasta 32 x 32.

Los kernels se lanzaron con estas dimensiones ajustadas para evitar el `error: -54` relacionado al tamaño incorrecto de los *work groups*.

2.4.2. Manejo de memoria

Otra diferencia fue cómo se manejó la memoria en OpenCL. En este caso, se debieron utilizar 2 buffers *cl_mem* para guardar los punteros a los arreglos de células, mientras que en CUDA no era necesario crear estos buffers adicionales pues `cudaMemcpy` trabaja directamente con el arreglo.

3. Resultados

Hardware/ API	# Columnas	# Filas	¿Doble dimensión?	Tamaño de bloque	Células por segundo
CPU	4	4	No	32	5548717
CPU	8	8	No	32	11880551
CPU	16	16	No	32	14019518
CPU	32	32	No	32	19000114
CPU	64	64	No	32	19479026
CPU	128	128	No	32	19229076
CUDA	4	4	No	32	447952
CUDA	8	8	No	32	1811712
CUDA	16	16	No	32	2685947
CUDA	32	32	No	32	27201542
CUDA	64	64	No	32	96732552
CUDA	128	128	No	32	273052864
OPENCL	4	4	No	32	653950
OPENCL	8	8	No	32	2642769
OPENCL	16	16	No	32	10954399
OPENCL	32	32	No	32	41444760
OPENCL	64	64	No	32	142408016
OPENCL	128	128	No	32	371788192

Tabla 1: Resultados del experimento 1, comparación de rendimiento entre CPU, y GPU utilizando Cuda y OpenCL para grillas pequeñas.

Hardware/ API	# Columnas	# Filas	¿Doble dimensión?	Tamaño de bloque	Células por segundo
CPU	100	1500	No	32	18518520
CPU	128	1500	No	32	19119392
CPU	500	500	No	32	18801234
CPU	512	512	No	32	19109984
CPU	1024	1024	No	32	18908446
CPU	1500	1500	No	32	18181964
CPU	2048	2048	No	32	18588966
CPU	4096	4096	No	32	18503784
CPU	5000	5000	No	32	18687634
CPU	8192	8192	No	32	18431384
CPU	15000	15000	No	32	18603040
CUDA	100	100	No	32	185910160
CUDA	128	128	No	32	271407680
CUDA	500	500	No	32	669576000
CUDA	512	512	No	32	684278656
CUDA	1024	1024	No	32	737541440
CUDA	1500	1500	No	32	793107840
CUDA	2048	2048	No	32	785619776
CUDA	4096	4096	No	32	798431488
CUDA	5000	5000	No	32	798208448
CUDA	8192	8192	No	32	790354304
CUDA	15000	15000	No	32	748057600
OPENCL	100	100	No	32	278097952
OPENCL	128	128	No	32	375047424
OPENCL	500	500	No	32	706491136
OPENCL	512	512	No	32	711076928
OPENCL	1024	1024	No	32	748140544
OPENCL	1500	1500	No	32	795296832
OPENCL	2048	2048	No	32	788808256
OPENCL	4096	4096	No	32	804141568
OPENCL	5000	5000	No	32	806557376
OPENCL	8192	8192	No	32	808297792
OPENCL	15000	15000	No	32	809612544

Tabla 2: Resultados del experimento 2, comparación de rendimiento entre CPU, y GPU utilizando Cuda y OpenCL.

Hardware/ API	# Columnas	# Filas	¿Doble dimensión?	Tamaño de bloque	Células por segundo
CUDA	8192	8192	No	30	786927040
CUDA	8192	8192	No	32	790300928
CUDA	8192	8192	No	60	796186112
CUDA	8192	8192	No	64	798239360
CUDA	8192	8192	No	90	795495424
CUDA	8192	8192	No	96	798135616
CUDA	8192	8192	No	120	795236672
CUDA	8192	8192	No	128	798403904
CUDA	8192	8192	No	150	794290432
CUDA	8192	8192	No	160	797824320
CUDA	8192	8192	No	190	796472000
CUDA	8192	8192	No	192	797447936
OPENCL	8192	8192	No	30	806514368
OPENCL	8192	8192	No	32	808240576
OPENCL	8192	8192	No	60	816437504
OPENCL	8192	8192	No	64	817374976
OPENCL	8192	8192	No	90	816093696
OPENCL	8192	8192	No	96	817234048
OPENCL	8192	8192	No	120	816357632
OPENCL	8192	8192	No	128	817465344
OPENCL	8192	8192	No	150	815603840
OPENCL	8192	8192	No	160	816801280
OPENCL	8192	8192	No	190	816252416
OPENCL	8192	8192	No	192	816581888

Tabla 3: Resultados experimento 3, comparación de performance entre OpenCl y Cuda utilizando distintos tamaños de bloque, en una dimensión.

Hardware/API	#Columnas	#Filas	¿Doble dimensión?	Tamaño de bloque	Celulas por segundo
CUDA	8192	8192	No	1	171243728
CUDA	8192	8192	No	2	383936960
CUDA	8192	8192	No	4	557412480
CUDA	8192	8192	No	8	678071104
CUDA	8192	8192	No	16	750645504
CUDA	8192	8192	Sí	1	176134640
CUDA	8192	8192	Sí	2	465664704
CUDA	8192	8192	Sí	4	654740544
CUDA	8192	8192	Sí	8	739804544
CUDA	8192	8192	Sí	16	781216512
OPENCL	8192	8192	No	1	421549280
OPENCL	8192	8192	No	2	547957952
OPENCL	8192	8192	No	4	663837824
OPENCL	8192	8192	No	8	739611840
OPENCL	8192	8192	No	16	784339328
OPENCL	8192	8192	Sí	1	431701440
OPENCL	8192	8192	Sí	2	664625216
OPENCL	8192	8192	Sí	4	777048512
OPENCL	8192	8192	Sí	8	804414592
OPENCL	8192	8192	Sí	16	816254912

Tabla 4: Resultados experimento 4, comparación de performance entre OpenCL y Cuda, con y sin utilizar doble dimensión.

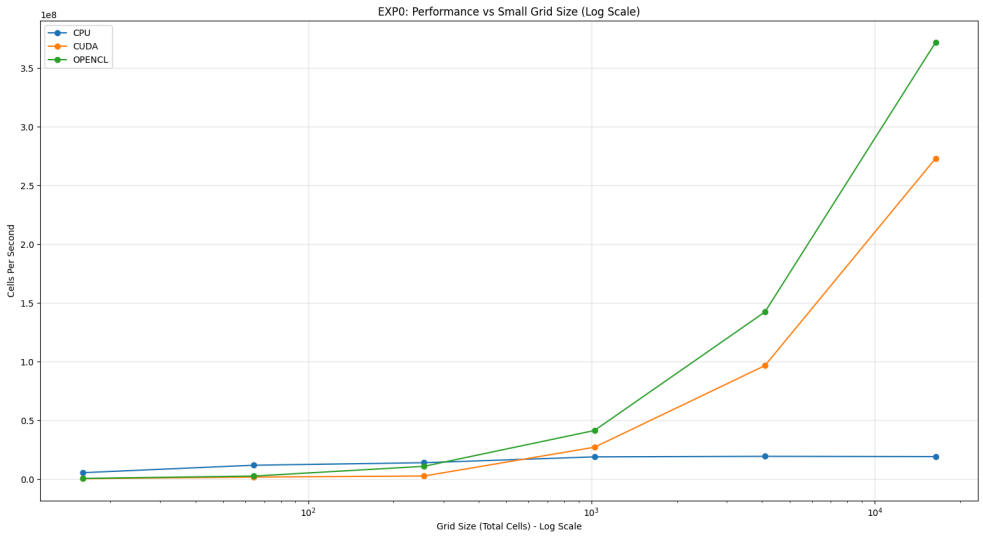

Figura 1: Cantidad de celulas procesadas por segundo VS logaritmo del tamaño de la grilla, para CPU, y GPU con Cuda y OpenCl para grillas pequeñas.

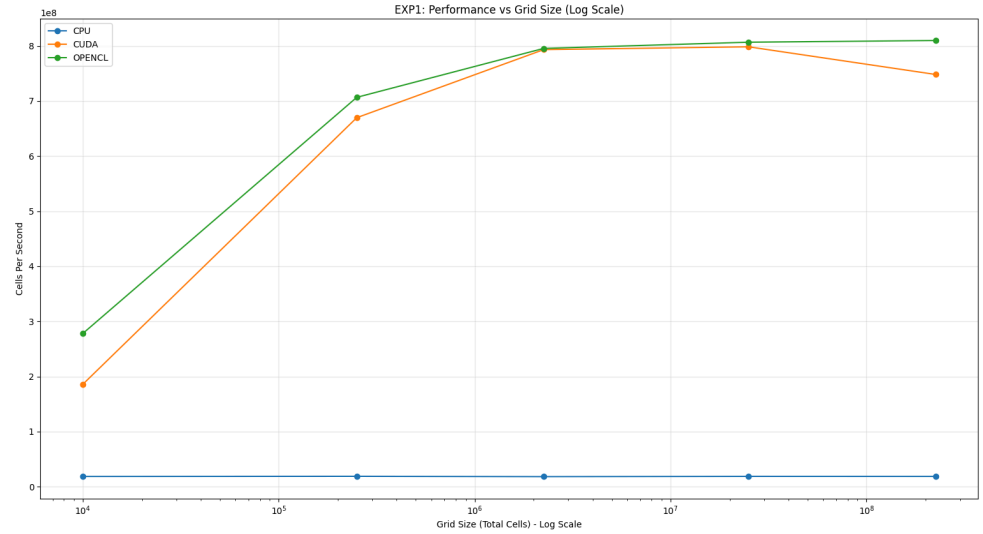

Figura 2: Cantidad de celulas procesadas por segundo VS logaritmo del tamaño de la grilla, para CPU, y GPU con Cuda y OpenCl.

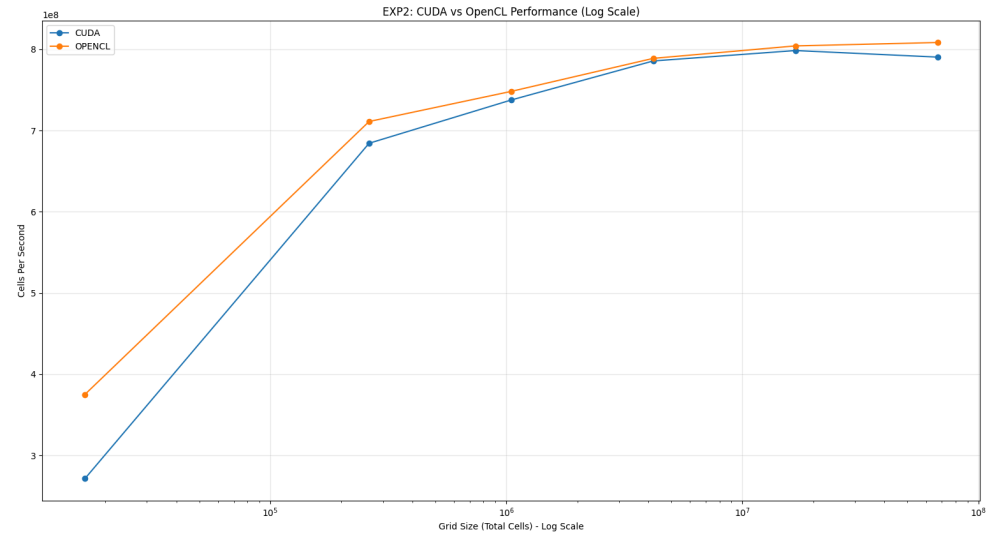

Figura 3: Cantidad de celulas procesadas por segundo VS logaritmo del tamaño de la grilla, para GPU con Cuda y OpenCl.

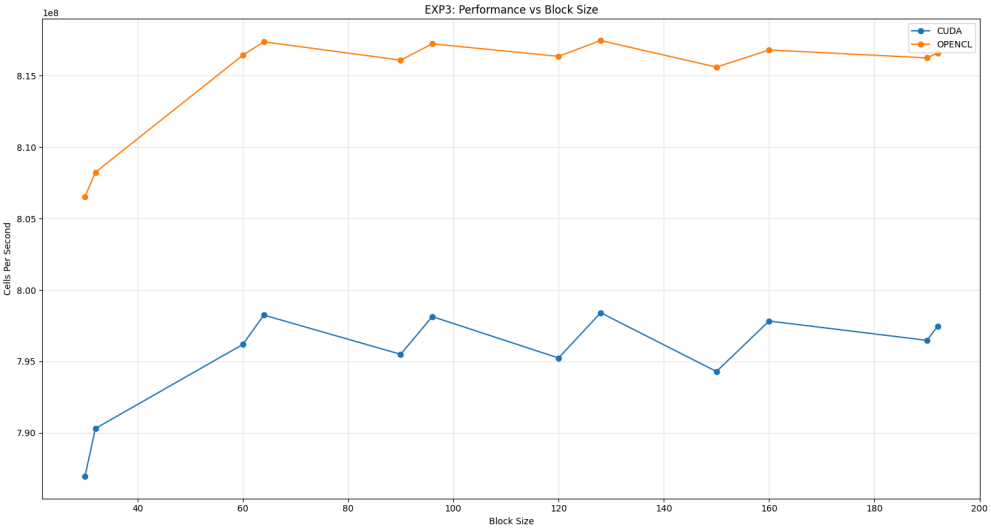

Figura 4: Cantidad de celulas procesadas por segundo VS tamaño de bloque, para GPU con Cuda y OpenCl.

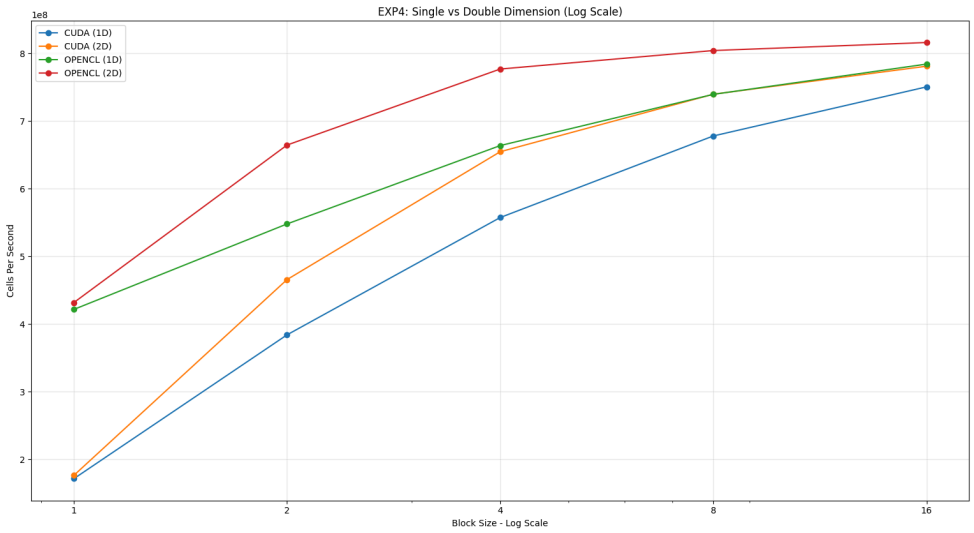

Figura 5: Cantidad de celulas procesadas por segundo vs tamaño de grilla en escala logaritmica, para GPU con Cuda y OpenCl, utilizando bloques de una y dos dimensiones.

4. Análisis de resultados

4.1. Experimento 1: CPU VS GPU/CUDA VS GPU/OPENCL

Primero se estudió el comportamiento de cada método (CPU, CUDA, OPENCL) en tamaños de grillas pequeños para determinar en qué momento el rendimiento en tarjetas de video supera al de la CPU. Se estimó que entorno a las 250 células (grilla 16x16) es el punto de inflexión pues luego la diferencia crece exponencialmente a favor de la GPU.

4.2. Experimento 2: CUDA VS OPENCL

El segundo experimento destaca la diferencia entre utilizar una CPU y una GPU para calcular las siguientes generaciones dentro de un rango más amplio de dimensiones. Se puede observar que la diferencia para tamaños pequeños de grilla es de un aumento de rendimiento de diez veces en la GPU, pero con unos tamaños mas grandes se puede alcanzar un aumento de 50x veces la cantidad de células.

4.3. Experimento 3: Tamaño de bloque

El tercer experimento consiste en observar como el tamaño de los bloques afecta el rendimiento (manteniendo constante el tamaño de la grilla en 8192x8192 y trabajando en una sola dimensión), se puede observar un comportamiento curioso: al llegar a un tamaño de bloque de 32, aumentar el tamaño del bloque no resulta en un incremento de rendimiento, al contrario, baja el rendimiento, excepto cuando el tamaño es un múltiplo de 32, obteniendo un rendimiento similar a cuando el tamaño es 32, y en el gráfico correspondiente se puede observar como esto genera una sierra en el gráfico. Es importante notar que este fenómeno sucede también cuando se ejecuta en CPU.

4.4. Experimento 4: Dimensión de bloques

El cuarto experimento muestra la diferencia entre utilizar o no bloques de doble dimensión al trabajar con distintos tamaños de grillas: se puede observar que en general utilizar doble dimensión es una mejora con respecto a bloques de una dimensión (es decir, para cualquier tamaño), y se mantiene la siguiente invariante: CUDA1D < CUDA2D <= OpenCL 1D < OpenCL 2D.

5. Conclusiones

En la simulación del juego de la vida, es claro que al aumentar las dimensiones de la grilla la cantidad total de células aumenta de manera exponencial, por lo cual la CPU cada vez se queda máS atrás en la cantidad de generaciones que puede simular por segundo, y alcanza rápidamente su cantidad total de células procesadas, y aquí es donde destacan de manera excepcional la GPU, pues, tanto en Cuda como OpenCl, se obtuvieron resultados que superan por mucho los obtenidos por CPU en cuanto a rendimiento en grillas pequeñas como en el máximo de células por seugndo que logran, lo cual está **en línea con la hipótesis propuesta**.

Además, entre los distintos tipos de ejecución por GPU y los parámetros de simulación, se puede obtener información valiosa:

- La CPU puede entregar un mejor rendimiento para grillas pequeñas menores a 256 células (16x16).
- Con tamaños de bloque que sean múltiplos de 32, es donde el rendimiento tanto en Cuda como OpenCL alcanzan su máximo, **validando la hipótesis inicial** y nos presenta un comportamiento que refleja la naturaleza de la arquitectura de la tarjeta utilizada.
- Hay incremento de rendimiento si utilizamos bloques de doble dimensión, y lo posiciona como una optimización importante, **refutando lo que planteamos en un principio**.
- Una constante que se mantiene es que el rendimiento de OpenCL supera al de CUDA, pero si consideramos la facilidad que es programar en Cuda en comparación con OpenCL, es un buen tradeoff.

Con respecto al último punto, es necesario agregar que, en los kernels implementados se realizó un manejo básico de memoria, sin optimizaciones como memoria compartida, por lo que se abre la posibilidad de estudiar qué tanto afecta aprovechar al máximo el modelo de memoria de la GPU, que se puede complementar con mejores divisiones del espacio y/o estructuras de datos más avanzadas.

6. Anexo: Código

6.1. CUDA - Kernel 1D

CUDA

```cuda
__global__ void game_of_life_kernel(char *d_cells, char *d_next_cells,
                                    int grid_x, int grid_y) {

  int idx = blockIdx.x * blockDim.x + threadIdx.x;

  if (idx < grid_x * grid_y) {

    int x = idx % grid_x;
    int y = idx / grid_x;

    int neighbors = 0;

    for (int dy = -1; dy <= 1; ++dy) {
      for (int dx = -1; dx <= 1; ++dx) {
        if (dx == 0 && dy == 0)
          continue;

        int nx = (x + dx + grid_x) % grid_x;
        int ny = (y + dy + grid_y) % grid_y;

                neighbors += d_cells[nx + ny * grid_x];
      }
    }

    if (d_cells[idx]) {
            d_next_cells[idx] = (neighbors == 2 || neighbors == 3);
    } else {
            d_next_cells[idx] = (neighbors == 3);
    }
  }
}
```

6.2. CUDA - Kernel 2D

CUDA

```cuda
__global__ void game_of_life_kernel_2d(char *d_cells, char *d_next_cells,
                                       int grid_x, int grid_y) {

  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  if (x < grid_x && y < grid_y) {

    int neighbors = 0;

    for (int dy = -1; dy <= 1; ++dy) {
      for (int dx = -1; dx <= 1; ++dx) {

        if (dx == 0 && dy == 0)
          continue;

        int nx = (x + dx + grid_x) % grid_x;
        int ny = (y + dy + grid_y) % grid_y;

                neighbors += d_cells[nx + ny * grid_x];
      }
    }

    int idx = x + y * grid_x;
    if (d_cells[idx]) {
            d_next_cells[idx] = (neighbors == 2 || neighbors == 3);
    } else {
            d_next_cells[idx] = (neighbors == 3);
    }
  }
}
```

6.3. CUDA - Kernel RandomInit

CUDA

```cuda
__global__ void randomize_grid_cuda_kernel(char *d_cells, int grid_x,
                                           int grid_y,
                                           unsigned long long seed) {

  int idx = blockIdx.x * blockDim.x + threadIdx.x;

  if (idx < grid_x * grid_y) {

        curandState state;
    curand_init(seed, idx, 0, &state);

    float random_value = curand_uniform(&state);

    if (random_value < 0.1f) {
            d_cells[idx] = 1;
    } else {
            d_cells[idx] = 0;
    }
  }
}
```

6.4. OpenCL - Kernel 1D

OpenCL

```
__kernel void game_of_life_kernel_opencl(
        __global const char* d_cells,      // current state array
        __global char*       d_next_cells, // next state array
    const int            grid_x,       // number of columns
    const int            grid_y        // number of rows
) {
    // Compute 1D global index for this work-item
    int idx = get_global_id(0);

    // Total number of cells
    int total = grid_x * grid_y;

    // Bounds check: exit if beyond the grid
    if (idx >= total) return;

    // Compute 2D coordinates from 1D index
    int x = idx % grid_x;
    int y = idx / grid_x;

    // Count live neighbors with toroidal wrapping
    int neighbors = 0;
    for (int dy = -1; dy <= 1; ++dy) {
        for (int dx = -1; dx <= 1; ++dx) {
            if (dx == 0 && dy == 0) continue;
            int nx = (x + dx + grid_x) % grid_x;
            int ny = (y + dy + grid_y) % grid_y;
                        neighbors += d_cells[nx + ny * grid_x];
        }
    }

    // Apply Conway's rules
    if (d_cells[idx]) {
                d_next_cells[idx] = (neighbors == 2 || neighbors == 3);
    } else {
                d_next_cells[idx] = (neighbors == 3);
    }
}

```

6.5. OpenCL - Kernel 2D

OpenCL

```
__kernel void game_of_life_kernel_2d_opencl(
        __global const char* d_cells,      // current cell states
        __global       char* d_next_cells, // next generation buffer
    const int      grid_x,             // number of columns
    const int      grid_y              // number of rows
) {
    // 2D global indices
    int block_x = get_global_id(0);    // X coordinate of this work-item
    int block_y = get_global_id(1);    // Y coordinate of this work-item

    // Bounds check: exit if outside the grid
    if (block_x >= grid_x || block_y >= grid_y) {
        return;
    }

    // Count live neighbors with toroidal wrap
    int neighbors = 0;
    for (int dy = -1; dy <= 1; ++dy) {
        for (int dx = -1; dx <= 1; ++dx) {
            if (dx == 0 && dy == 0) continue;
            int nx = (block_x + dx + grid_x) % grid_x;
            int ny = (block_y + dy + grid_y) % grid_y;
                        neighbors += d_cells[nx + ny * grid_x];
        }
    }

    // Compute flat index and apply Conway's rules
    int idx = block_x + block_y * grid_x;
    if (d_cells[idx]) {
                d_next_cells[idx] = (neighbors == 2 || neighbors == 3);
    } else {
                d_next_cells[idx] = (neighbors == 3);
    }
}

```

6.6. Dimensiones de trabajo

6.6.1. OpenCL

cpp

```cpp
size_t max_work_group_size;
clGetDeviceInfo(device, CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(size_t),
                &max_work_group_size, NULL);

size_t max_work_item_dimensions;
clGetDeviceInfo(device, CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, sizeof(size_t),
                &max_work_item_dimensions, NULL);

size_t max_work_item_sizes[3];
clGetDeviceInfo(device, CL_DEVICE_MAX_WORK_ITEM_SIZES,
                sizeof(max_work_item_sizes), max_work_item_sizes, NULL);

// Adjust work sizes based on device capabilities
size_t adjusted_block_size = block_size;

// For 1D: ensure local work size doesn't exceed max work group size
if (adjusted_block_size > max_work_group_size) {
    adjusted_block_size = max_work_group_size;
    std::cout << "Warning: Adjusted 1D block size from " << block_size
              << " to " << adjusted_block_size << std::endl;
}

// For 2D: ensure each dimension and total don't exceed limits
size_t adjusted_block_size_2d = block_size;
if (adjusted_block_size_2d > max_work_item_sizes[0] ||
        adjusted_block_size_2d > max_work_item_sizes[1]) {
    adjusted_block_size_2d =
      std::min(max_work_item_sizes[0], max_work_item_sizes[1]);
    std::cout << "Warning: Adjusted 2D block size per dimension from "
              << block_size << " to " << adjusted_block_size_2d << std::endl;
}

// Check total work items in 2D work group
if (adjusted_block_size_2d > max_work_group_size) {
    adjusted_block_size_2d = max_work_group_size * max_work_group_size;
    std::cout << "Warning: Adjusted 2D block size for total work group from "
              << block_size << " to " << adjusted_block_size_2d << std::endl;
}

// Recalculate work sizes with adjusted values
const size_t adjusted_local_work_size[1] = {adjusted_block_size};
const size_t adjusted_local_work_size_2d[2] = {adjusted_block_size_2d,

adjusted_block_size_2d};

const size_t adjusted_total_cells = GRID_X * GRID_Y;
const size_t adjusted_global_work_size[1] = {
    ((adjusted_total_cells + adjusted_block_size - 1) /
          adjusted_block_size) *
```

```
49         adjusted_block_size};
50
51 size_t adjusted_num_groups_x =
52     (GRID_X + adjusted_block_size_2d - 1) / adjusted_block_size_2d;
53 size_t adjusted_num_groups_y =
54     (GRID_Y + adjusted_block_size_2d - 1) / adjusted_block_size_2d;
55
56 const size_t adjusted_global_work_size_2d[2] = {
57         adjusted_num_groups_x * adjusted_local_work_size_2d[0],
58         adjusted_num_groups_y * adjusted_local_work_size_2d[1]};
```

6.6.2. CUDA

cpp

```
1 double_dim = std::atoi(argv[5]);
2 block_size = std::atoi(argv[6]);
3
4 dim3 block_size_2d(block_size, block_size);
5 dim3 grid_size_2d((GRID_X + block_size_2d.x - 1) / block_size_2d.x,
6                   (GRID_Y + block_size_2d.y - 1) / block_size_2d.y);
```

6.7. Manejo de memoria

6.7.1. OpenCL

cpp

```cpp
// Inicializar arreglos
char *d_cells, *d_next_cells, *d_cells_double, *d_next_cells_double;

// Inicializar variables de OpenCL para los buffers
cl_mem opencl_d_cells, opencl_d_next_cells;

// Inicializar buffer para guardar el estado actual de las células
opencl_d_cells = clCreateBuffer(context, CL_MEM_READ_WRITE,
                                GRID_X * GRID_Y * sizeof(char),
                                NULL, &err_cl);

// Inicializar buffer para guardar el estado de la próxima generación de células
opencl_d_next_cells = clCreateBuffer(context, CL_MEM_READ_WRITE,
                                     GRID_X * GRID_Y * sizeof(char),
                                     NULL, &err_cl);

// Escribir en el buffer el estado inicial (randomizado en CPU)
err_cl = clEnqueueWriteBuffer(queue, opencl_d_cells, CL_TRUE, 0,
                              GRID_X * GRID_Y * sizeof(char), cells.data(),
                              0, NULL, NULL);

// Leer del buffer que tiene los resultados (después del swap) hacia cells
clEnqueueReadBuffer(queue, opencl_d_cells, CL_TRUE, 0,
                    GRID_X * GRID_Y * sizeof(char), cells.data(),
                    0, NULL, NULL);

// Liberar buffers
clReleaseMemObject(opencl_d_cells);
clReleaseMemObject(opencl_d_next_cells);
```

6.7.2. CUDA

cpp

```cpp
// Inicializar arreglos
char *d_cells, *d_next_cells, *d_cells_double, *d_next_cells_double;

// Inicializar buffer para el estado actual
cudaMalloc((void **)&d_cells, GRID_X * GRID_Y * sizeof(char));

// Inicializar buffer para el estado siguiente
cudaMalloc((void **)&d_next_cells, GRID_X * GRID_Y * sizeof(char));

// Llenar el arreglo incial de células, para randomizarlo en la GPU
cudaMemcpy(d_cells, cells.data(), GRID_X * GRID_Y * sizeof(char),
           cudaMemcpyHostToDevice);

// Copiar datos de la siguiente generación hacia el host
cudaMemcpy(cells.data(), d_next_cells, GRID_X * GRID_Y * sizeof(char),
           cudaMemcpyDeviceToHost);
```

```
// Liberar recursos
cudaFree(d_cells);
cudaFree(d_next_cells);
```