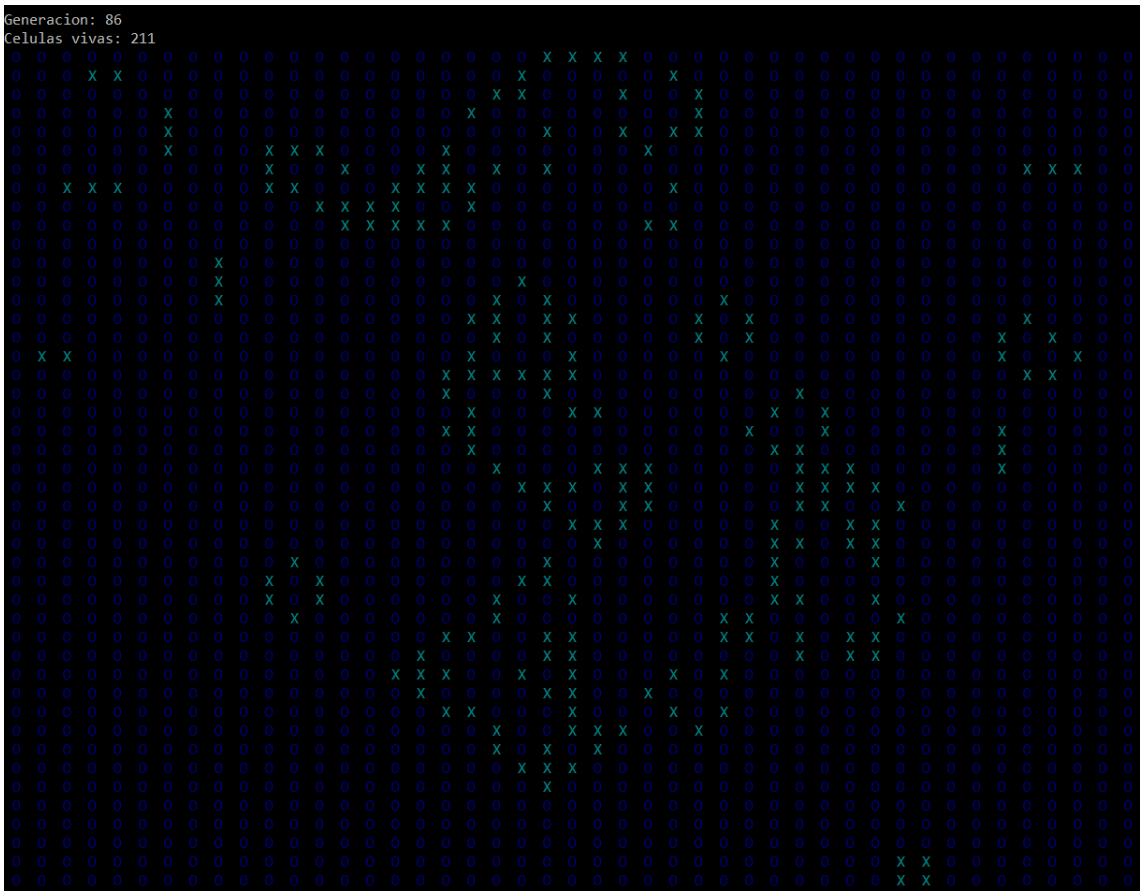
MEMORIA

GAME OF LIFE



Javier Martín Gómez 47231977-M

Alberto González Martínez 09072311-F

Contenido

1. INTRODUCCIÓN

Hemos realizado el juego de “Game of Life” en la plataforma de computación paralela “CUDA”. Para ello hemos implementado el juego en memoria global y memoria por bloques. Usamos una matriz con células que se va modificando cada instante como explicamos a continuación.

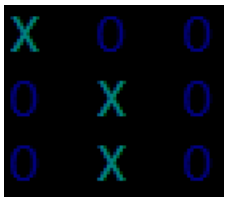
2. DINÁMICA DEL JUEGO

El juego de la vida consta de un tablero de o matriz de dimensiones variables, en donde, cada una de las posiciones del tablero es una “célula” y cada una de estas tiene células vecinas.

Estas células tienen dos posibles estados “muertas” o “vivas”, pudiendo cambiar su estado a lo largo de unidades de tiempo (en nuestro caso un contador llamado generaciones), el estado de estas células viene dado por un patrón:

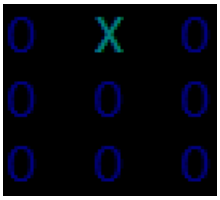
- Si una célula viva tiene a su alrededor dos o tres células vivas, en la siguiente generación esta célula seguirá viva, si no, en la siguiente generación la célula pasará a estar muerta.

- Si una célula muerta, tiene a su alrededor tres células vivas, en la siguiente generación pasará a estar viva, si no, en la siguiente generación seguirá muerta.


Viva -> Viva


Viva -> Muerta


Muerta -> Viva


Muerta -> Muerta

3. EXPLICACIÓN DEL PROGRAMA

Para ejecutar el programa, tenemos que introducir por consola si queremos ejecutarlo automática (cada cierto tiempo hace una iteración) o manualmente (hay que pulsar una tecla para pasar de iteración), el número de filas y el número de bloques, un ejemplo sería así:

```
C:\Users\Alberto\Documents\GitHub\Game_of_Life_CUDA\GameOfLife\x64\Debug>GameOfLife.exe -m 25 32
```

Donde -m significa manual, 25 las filas y el 32 las columnas.

Para inicializar la matriz, lo hacemos de manera aleatoria de modo que haya pocas células ‘X’ (vivas) y bastantes células ‘O’ (muertas) para que el juego tenga más sentido, ya que si hay muchas vivas desde el principio el juego durará muy poco.

Una vez inicializada la matriz, utilizamos el Kernel para controlar qué células estarán vivas o muertas en la siguiente iteración. Para ello controlamos cada posición de la matriz y usamos un contador que se sume 1 cuando una célula tenga otra célula en las posiciones de su alrededor y, dependiendo del valor del contador, la célula vivirá o morirá en la siguiente iteración. Los cambios en cada iteración se irán guardando en otra matriz.

También, vamos indicando en cada iteración el número de generación y las células vivas. El programa se irá ejecutando hasta que el número de células vivas sea igual a la dimensión de la matriz (todas vivas). Por lo tanto, se irá lanzando el Kernel hasta que todas estén vivas, por lo que, en cada iteración, la matriz resultado será la matriz de inicio de la siguiente.

Además, el programa, comprueba las características de su tarjeta, por si los datos introducidos superan el número de hilos, bloques…

4. PARTES REALIZADAS

Se ha realizado el programa en memoria global, utilizando un solo bloque.

También se ha realizado el programa en memoria por bloques, donde, si el número de hilos es mayor que el número máximos de hilos por bloque que admite la tarjeta (habitualmente 32), divide el programa en más de un bloque.

No se ha implementado el programa mediante memoria compartida y por tanto no se ha podido implementar la interfaz gráfica, aunque mediante la función imprimirMatriz() se muestra la ejecución del programa lo más clara y visual posible.

5. FUNCIONES DEL PROGRAMA

cudaError_t lanzarKernel(char* matriz, char* matrizResultado, int fila, int columna):

Este método, primero controla todos los errores posibles del programa y después, una vez comprobados los errores, establece el número de bloques e hilos del programa y lanza el Kernel con sus respectivas fases (reservar memoria, intercambiar los datos entre el device y el host, liberar memoria…).

En el caso de memoria por bloques, si se necesita más de un bloque, establece la condición y llama al método numeroBloques() para comprobar el número de bloques necesarios para el programa.

int contarVivas(char* matriz, int dimension):

Este método recorre la matriz y cuando una casilla sea igual a ‘X’ suma 1 al contador, por lo que devolverá el número de células vivas que hay en el tablero.

void rellenarMatriz(char* matriz, int dimension):

Este método crea la matriz inicial con las ‘X’ y las ‘O’ de forma aleatoria utilizando el método srand() que genera un seed aleatoria en función de la hora del sistema la cual se consigue con time(0).

También según la dimensión de la matriz, habrá más o menos probabilidad de que aparezca o no una célula viva en una posición del tablero.

void imprimirMatriz(char* matriz, int dimension, int columna):

Este método muestra la matriz como una matriz dividida en filas y columnas y espacios de manera que se vea con claridad.

Además, añade color a la matriz que se muestra por consola, teniendo las células vivas color cian y las células muertas color azul para facilitar la visualización del movimiento y comportamiento de las células a lo largo del tiempo

void numeroBloques(int dimension, int width):

Este método se usa para saber el número de bloques necesarios para el programa. Si la división de la fila o columna de la matriz entre el ancho de la tesela es una división exacta, el número de bloques será igual a la división, y si no es exacta, la división se redondea a la alta para usar los bloques necesarios (si se redondease hacia abajo no cabrían los hilos en los bloques).

__global__ movimientoCelular(char* matriz, char* matrizResultado, int fila, int columna):

Es el Kernel del programa de memoria global, en él, primero accedemos a la posición a través de la fórmula threadIdx.x * columna + threadIdx.y.

Una vez hemos calculado la posición del hilo en nuestra matriz unidimensional, se aplican algunas restricciones a la matriz, en donde, en función del id del hilo (threadIdx.x y threadIdx.y) limitaremos el movimiento de las células fuera de los márgenes de la matriz.

Para cada uno de los casos correspondientes que controlan los límites del tablero incrementaremos el contador, en función de si alguna de las células vecinas a la posición del tablero está viva.

Una vez acabadas las comprobaciones y teniendo el número de células vivas vecinas, dependiendo de si la célula está viva o muerta, introduciremos un valor u otro dentro de la matriz resultado, la cual tendrá la información respecto a las células de la siguiente generación.

__global__ movimientoCelularBloques(char* matriz, char* matrizResultado, int fila, int columna):

Es el Kernel del programa de memoria global por bloques, en él, para acceder a la posición de nuestra matriz unidimensional primero hallamos la fila y columna en la que estamos mediante la siguiente fórmula:

- fila = blockIdx.x * blockDim.x + threadIdx.x
- columna = blockIdx. * blockDim.x + threadIdx.x

Una vez halladas la fila y columna, hallamos la posición que será igual a fila * columna + columna.

El patrón para comprobar las células vivas o muertas es el mismo que en el anterior, con la ligera diferencia de que en vez de hacer las comparaciones con el mismo id del hilo, utilizamos las variables fila y columna, ya que, si usáramos el id del hilo, cuando hubiera más de un bloque esas posiciones se repetirían y el patrón sería incorrecto.

6. MAIN

El main de nuestro programa, antes de empezar con la ejecución de la lógica del programa, comprueba que todos los parámetros introducidos por la terminal sean correctos para evitar errores.

Primero, comprueba que el número de argumentos introducidos por la terminal sean 4 (tal y como hemos visto anteriormente) <.exe modo fila columna>, luego convierte los parámetros fila y columna a un entero el cual pueda ser usado por el programa e inicializar las matrices correspondientes y comprueba que esos valores sean válidos y que el argumento de ejecución automática o manual sea el correcto.

Para el caso del programa de memoria global sin bloques, comprueba que la dimensión de la matriz no sea mayor al máximo número de hilos que puede ejecutar nuestra tarjeta.

A continuación, rellenamos la matriz y, mediante un bucle while con condición de parada en los dos casos en los cuales se termina el juego (ninguna viva o todas vivas) llama a la función lanzarKernel, mostramos por pantalla la nueva matriz junto con el contador de células vivas del tablero más la generación del juego en la que estamos.

En el caso de que el usuario introduzca ejecución manual -m, el programa esperará que el usuario pulse una tecla para mostrar la siguiente generación, en el caso de que el usuario introduzca ejecución automática -a, programa esperará 1 segundo para mostrar la siguiente generación.