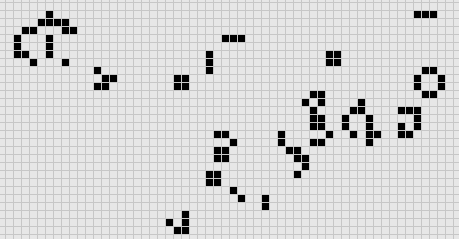
PRÁCTICA 1 PARADIGMAS DE PROGRAMACIÓN AVANZADA

# EL JUEGO DE LA VIDA, PROGRAMACIÓN USANDO GPU CUDA

# CURSO 2019/2020

## Javier García Jiménez 09099503J

## Isabel Martínez Gómez 06027983M



**INTRODUCCIÓN**

La primera práctica de Paradigmas Avanzados de Programación consiste en la implementación del “Juego de la Vida”, un algoritmo para observar cómo se comporta la ejecución paralela de hilos en la GPU.

Este juego consiste en un tablero de células que tienen dos posibles estados: vivas y muertas. Estas células pueden evolucionar cambiando de estado entre vivas y muertas en cada iteración.

Una célula que está viva se mantiene como tal si tiene otras dos células o tres células vivas a su alrededor y se muere en caso contrario.

Una célula que está muerta revive si tiene exactamente tres células vivas a su alrededor.

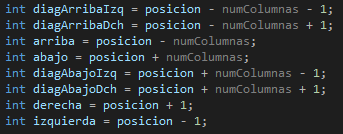
Con la expresión alrededor nos queremos referir a todas las células que son “vecinas” de la célula en cuestión, esto es, arriba, abajo, a la derecha, a la izquierda y en las cuatro diagonales.

Una vez conocido el juego y como funciona, vamos a echar un vistazo a la implementación que hemos realizado.

Se ha realizado una implementación de tres kernels, cada uno con una organización de memoria, bloques e hilos diferente.

**PATRÓN DEL JUEGO DE LA VIDA**

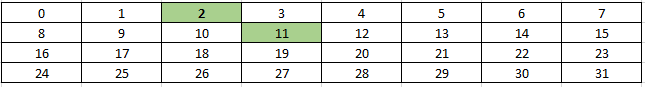
Para implementar el patrón del Juego de la Vida, hemos usado las siguientes implementaciones:

Como hemos dicho anteriormente, para mirar si una célula viva sigue viva, se muere o por el contrario, para mirar si una célula que está muerta puede revivir hay que tener en cuenta las células vecinas y, para ello vamos a tener en cuenta las siguientes posiciones: 

Por ejemplo, para sacar la posición de la diagonal arriba izquierda de una posición, habría que hacer el siguiente cálculo:

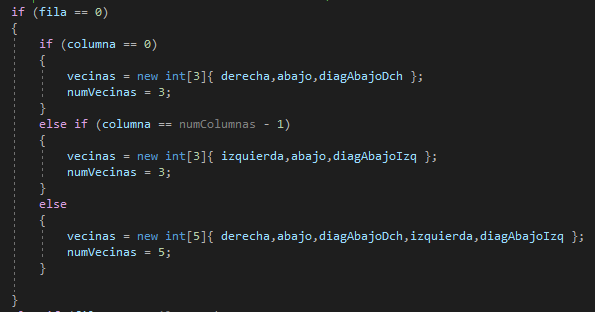
***Posición diagonal arriba izquierda*** *= posición actual - número de columnas - 1*

Esto se puede demostrar fácilmente mediante un ejemplo, supongamos que tenemos una matriz de 4\*8 y estamos en la posición 11, para calcular la posición de la diagonal arriba izquierda sería 11 - 8 - 1 = 2. Como podemos ver, se cumple.



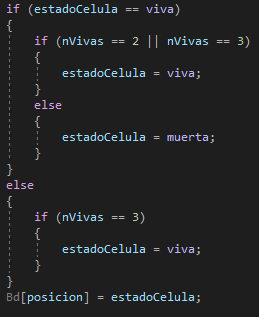
Una vez calculadas las 8 posiciones vecinas en la matriz, realizamos un array vecinas. Este array nos va a servir para saber en cada caso el número de vecinas que tiene una célula y además para almacenar las posiciones de las células vecinas.

Por ejemplo, las células que están en una esquina sólo tendrá tres células vecinas, las que están en los laterales pero no en las esquinas tendrán cinco y las que no están ni en un lateral ni en una esquina tendrán ocho células vecinas.



Una vez sacado el número de células vecinas que tiene una célula y sacado las posiciones de cada una de ellas, recorremos el array de vecinas de tal forma que contamos el número de células vivas que tiene alrededor una célula.

Finalmente, hacemos la comprobación final.

* Si la célula está viva:
  + Si el número de células vecinas vivas es de 2 o 3, la célula sigue viva.
  + En caso contrario, muere.
* Si la célula está muerta:
  + Si el número de células vecinas es exáctamente de 3, renace.
  + En caso contrario, sigue estando muerta.
  + 

Este patrón va a ser igual en todos los kernels, la única diferencia es la manera de calcular la posición. Por otra parte, en el kernel implementado con múltiples bloques se cogerá una matriz Ad, sin embargo, en el kernel implementado con memoria compartida se cogerá una matriz Ads que es un array compartido por todos los hilos de un mismo bloque.

**MEMORIA GLOBAL CON UN SOLO BLOQUE.**

Esta implementación es un kernel que se compone de un único bloque con todos los hilos de la matriz.



Como se observa en la imagen en esta implementación sólo existe un bloque que posee tantos hilos como elementos tenga la matriz.

Cabe mencionar que esta implementación sólo se ejecuta en caso de que el número de hilos máximo esté dentro de los límites del dispositivo. En nuestro caso, por ejemplo, el máximo número de hilos por bloque es de 1024, por tanto si hay una matriz que el número de elementos que contiene sobrepasa el límite de hilos del que dispone un bloque, este kernel no se ejecuta y pasaríamos a ejecutar uno de los otros dos kernels que serán explicados posteriormente.

Bien, ahora miremos como funciona el interior del kernel y como se obtienen las posiciones de los elementos en la matriz, filas, columnas, etc.



En la imagen se observa cómo se seleccionan la fila, la columna y la posición del elemento en la matriz correspondiente a un hilo.

Como estamos en un solo bloque que contiene todos los hilos ocurre lo siguiente:

* **Fila:** para coger la fila del elemento nos tenemos que fijar en el identificador correspondiente a la variable “y” del hilo que especificará la fila. Al estar trabajando con bloques de dos dimensiones, nos tendremos que referir a “y” como la fila y a “x” como la columna.
* **Columna:** en el caso de la columna haríamos lo mismo que en la fila pero seleccionando el identificador “x” del hilo, que es el que se corresponde a las columnas.
* **Posición:** para conocer la posición del elemento dentro de la matriz ya no nos sirve con fijarnos sólo en identificadores de los hilos, tenemos que recurrir a un sencillo cálculo. Este cálculo no es más que una multiplicación del identificador “y” del hilo correspondiente a la fila por el número de columnas que tenga nuestra matriz. Tras esto, le añadimos el valor del identificador “x” del hilo (correspondiente a la columna) y ya tendríamos el valor de la posición.

Una vez conocida la posición, la fila y la columna del elemento de la matriz, pasamos a calcular el siguiente estado que tendrá dicho elemento en la siguiente iteración como hemos explicado en el primer apartado de la memoria.

**MEMORIA GLOBAL CON MÚLTIPLES BLOQUES**

Ahora pasaremos a explicar el segundo kernel implementado, que ni más ni menos consiste en una ejecución en memoria global con más de un bloque.

Para definir el kernel tenemos en cuenta el tamaño de una tesela para definir el número de bloques que va a haber y cuál va a ser el tamaño de los mismos.



Para calcular la dimensión del grid, es decir el número de bloques que va a haber, multiplicamos el número de bloques por columnas y el número de bloques por filas.

Por otra parte, para calcular el dimBlock, es decir, el número de hilos en total en cada bloque, multiplicamos el tamaño de la tesela por el tamaño de la tesela de forma que tenemos el número de elementos que hay en un bloque.

En nuestro caso, hemos definido el tamaño de nuestra tesela como el número de columnas dividido entre 2, con el objetivo de tener la matriz dividida en 4 bloques.

**MEMORIA COMPARTIDA (MÚLTIPLES BLOQUES)**

En este apartado vamos a comentar la última implementación. En este caso se trata de un kernel con múltiples bloques los cuales utilizan memoria compartida para reducir los costes por acceso a memoria.

Esta implementación es algo más complicada, ya que los elementos necesitan información de sus elementos contiguos y los elementos que están en los límites de un bloque no podrían acceder a elementos que pertenecen a otro bloque (ya que se usa memoria compartida y esta pertenece sólo a los hilos de un mismo bloque). Para solucionar este problema hemos implementado un algoritmo que modifica el tamaño de los bloques del kernel.

Como bien sabemos, el tamaño de los bloques depende de la tesela. Sin embargo para solucionar el problema, lo que hemos hecho ha sido añadir una fila y una columna más a nuestra memoria compartida para que así, los elementos limítrofes del bloque, puedan tomar información de esa fila y esa columna “extras” para así poder calcular su valor.

Pongamos un ejemplo para comprender mejor esta solución. Si tenemos una matriz de 32 x 32, como hemos comentado anteriormente tendremos una tesela que se corresponderá al número de columnas dividido entre dos para dividir la matriz en 4 bloques. Con esto tendríamos unos bloques de *tesela \* tesela* dimensiones. En este caso, los bloques serían de 16 x 16 (256 hilos). Por tanto pongamos en situación que ocurriría si en la memoria compartida no aplicásemos la solución.

En caso de que la solución no fuese aplicada, si nos encontramos por ejemplo en el bloque (0, 0) que es el primer bloque, en la última columna del mismo (columna 15), si ese elemento quiere calcular su próximo valor deberá coger información de sus elementos vecinos, los cuales algunos de ellos se corresponden con la columna 16 de la matriz. Esto sería imposible, ya que nuestra memoria compartida es de 16 x 16 y estaríamos intentando acceder a un elemento que se encuentra en la columna 16 (que realmente sería la columna 17 ya que los números comienzan en 0 ). Por tanto, realizar esta implementación sin aplicar la solución sería imposible.

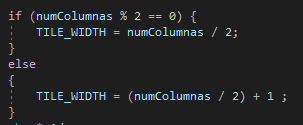
Sin embargo, si aplicamos la solución propuesta, cuando cargásemos en memoria compartida nuestra sub-matriz de 16 x 16, añadiríamos a la misma una fila y una columna extras para que nuestros elementos limítrofes no tengan problemas a la hora de calcular su próximo valor a través de sus células vecinas.

Pero esto no es todo, ya que también debemos tener en cuenta en qué bloque estamos situados para saber exactamente cuáles son la fila y la columna extra que nos interesan, ya que no siempre es una fila más abajo o una columna más a la derecha, esto varía dependiendo del bloque en el que nos encontremos.

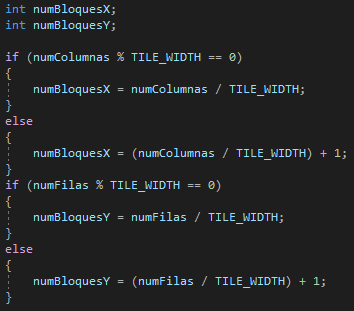
Por tanto, como hemos comentado, tenemos nuestra matriz siempre dividida en 4 bloques, lo que nos indica lo siguiente:

* Si nos encontramos en el bloque (0,0), tenemos que añadir la columna de la derecha y la fila de abajo.
* Si nos encontramos en el bloque (0,1), tenemos que añadir la columna de la izquierda y la fila de abajo.
* Si nos encontramos en el bloque (1,0), tenemos que añadir la columna de la derecha y la fila de arriba.
* Si nos encontramos en el bloque (1,1), tenemos que añadir la columna de la izquierda y la fila de arriba.

En definitiva, con las siguientes imágenes mostraremos las distintas definiciones y variables que componen esta implementación.



Aquí vemos la definición de la tesela. Como se ha comentado anteriormente, esto se realiza dividiendo entre dos para tener la matriz dividida en 4 bloques.



En la imagen anterior vemos como se define el número de bloques que van a existir. Para ello dividimos el número de columnas y de filas que tiene nuestra matriz por la tesela, teniendo en cuenta que si no es divisor la tesela del número de columnas o de filas que tenga nuestra matriz, hay que añadir un bloque más para que no haya elementos de la matriz que se queden sin computar.

En cuanto a el número de bloques y de hilos, lo calcularemos según lo siguiente:



Como se puede observar, los bloques tienen *tesela \* tesela* hilos, y el grid tiene los números de bloques que se definen como hemos comentado anteriormente.



Por último, tenemos la definición del kernel. En este caso tiene una peculiaridad con respecto a las otras definiciones. En este caso vemos que el kernel tiene un tercer argumento. Este corresponde al asignamiento de memoria compartida a cada bloque. Como hemos mencionado anteriormente con la solución propuesta, nuestras matrices en memoria compartida poseen una fila y columna extras, por eso mismo, se le suma uno al tamaño de *tesela \* tesela*, para que la matriz con los elementos extras quepa perfectamente en la memoria compartida.

**EXPLICACIÓN DEL MENÚ CREADO**

En la implementación se ha creado un breve menú para elegir la ejecución del kernel. Gracias a este menú se podrá ejecutar el programa de cualquiera de las tres formas introduciendo un número:

* 1 para memoria global con un bloque
* 2 para memoria global con múltiples bloques
* 3 para memoria compartida

