Cellular Automata

TGA 2019/20-Q2

Joseba Sierra
Aina Garcia

Índice

1. Autómata Celular, especificación

1.1. Planteamiento

Por definición, un *Autómata Celular* es un modelo matemático para un sistema dinámico que evoluciona en pasos discretos. Es adecuado para modelar sistemas naturales que puedan ser descritos como una colección masiva de objetos simples que interactúen localmente unos con otros[1].

El Autómata Celular está compuesto por una rejilla o lattice de dimensión N. Cada elemento de esta rejilla, al cual llamaremos célula, puede tomar un valor entre un conjunto de valores enteros finitos. Además, cada célula está unida a sus células vecinas, las cuales van a influir en la evolución del valor de esta célula. Finalmente se definen un conjunto de normas que definen el valor de una célula respeto a sus vecinas.

Así pues, el Autómata Celular va avanzando por generaciones. Cada generación consiste básicamente en la recalculación del valor de cada una de las células acorde con el valor de sus células vecinas aplicando las normas que se han definido al inicio. En la Figura 1 se muestra un ejemplo de reglas en un Autómata celular de una dimensión. Básicamente indica que por el color de la célula y el color de sus dos células vecinas, el color que tendrá la célula central en la siguiente generación va a ser el del cuadrado de abajo.

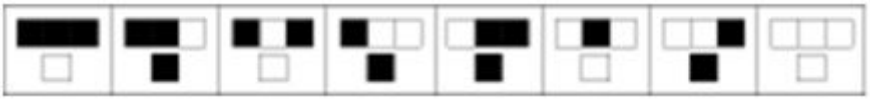

Figura 1: Ejemplo de reglas en una dimensión

1.2. El problema en las fronteras

En cuanto al cálculo del vecindario de una célula, es el control de las fronteras de nuestra lattice. Según cómo se implementan estas, el resultado de la simulación puede variar. Algunas de las formas de tratar las células frontera son las siguientes:

Frontera abierta. Se dice que un AC tiene frontera abierta cuando se le da un valor fijo a los elementos de la frontera. Estos pueden ser siempre el mismo valor (como si fuera un margen añadido a todo el AC) o un valor inicializado a mano.

Frontera periódica. Se considera a la lattice como si sus extremos se tocan, es decir, el elemento 0 tendría al elemento N como su vecino y a la inversa. En la Figura 2 se muestra un ejemplo de una lattice 2D con frontera periódica.

[1] Definición de Wikipedia

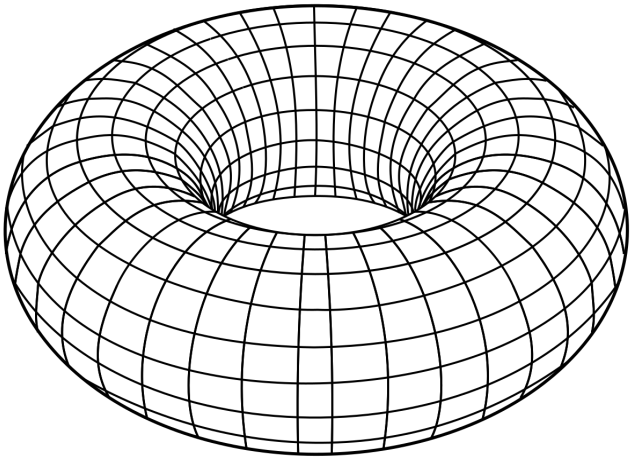

Figura 2. Lattice periódica 2D

Frontera reflectora. Las células fuera de la frontera con una copia de sus células adyacentes que están dentro de la lattice.

Sin frontera. Esta implementación es un poco diferente de las anteriores ya que estas tenían un tamaño fijo de lattice. En la implementación sin frontera, la lattice va creciendo en cada generación, añadiendo un margen en blanco al final de los bordes de la lattice.

En nuestro caso, hemos usado la frontera periódica ya que creemos que es la más simple de programar y a la vez da resultados más naturales, ya que en ningún momento estamos inicializando un valor fijo para cada generación posterior a la primera.

1.3. Definición de reglas

Como hemos mencionado antes, el AC se basa en un conjunto de reglas que sirven para calcular el valor de cada célula en la siguiente generación. Estas reglas pueden definirse de varias formas, entre las cuales las más usadas son las siguientes:

1.3.1. Reglas textuales

La primera forma de definir reglas es escribirlas de forma textual, por ejemplo:

- Si una célula tiene 3 vecinos vivos, ella vive en la siguiente generación.
- Si una célula está viva y tiene un vecino vivo, vive en la siguiente generación.

Esta forma de definir reglas resultan muy fáciles de comprender para un humano pero hacen que el AC no sea flexible, es decir, que si quieres introducirle otro tipo de reglas, deberás reprogramar toda la función que comprueba el estado de la siguiente generación. En caso de querer hacer un AC en concreto como podría ser el *Game of life*, esta aproximación ya nos serviria. Como nosotros queremos implementar un AC genérico, hemos decidido usar el

siguiente método de definición de reglas ya que como veremos a continuación nos da más libertad para definirlas.

1.3.2. Reglas codificadas

Supongamos que tenemos un AC de una dimensión con valores de célula booleanos, donde el vecindario de esta queda definido por la célula central y sus dos células adyacentes. Así pues, podríamos definir una estructura de vecindario para cada célula como la descrita en la Figura 3.

Figura 3. Vecindario completo para AC 1D (V2 célula central)

Las reglas de este vecindario se podrían codificar en binario según todas las posibilidades, con lo que nos quedaría una tabla de reglas como la que podemos ver en la Figura 4.

V1	V2	V3	NextGen
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**0**
1	0	0	**1**
1	0	1	**1**
1	1	0	**0**
1	1	1	**0**

Figura 4. Reglas codificadas

Sabiendo el orden en el que codificamos los vecinos, no nos hace falta guardar toda la tabla entera, sino que con guardar el valor de la NextGen en un vector, se puede simplemente guardar la última columna. Por ejemplo, si tenemos un vecindario de [110], sabemos que este número en decimal es el 6 en decimal, por lo que el valor de la célula central en la siguiente generación será NextGen[6] que en la tabla de ejemplo sería un 0. Podemos ir un paso más allá y para inicializar todo el array NextGen pedir al usuario un número decimal entre 0 y 2 ^ (2 ^ tamaño vecindario), que luego podemos convertir en binario y tener ya inicializadas todas las reglas.

1.4. Diseño de nuestro programa

Nuestro planteamiento del problema era el de poder crear un Autómata Celular general en dos dimensiones. Por este motivo, escogimos implementar las reglas codificadas. Así pues, al ejecutar el programa, se pasan como argumentos número de reglas junto con el tamaño de la rejilla. Además para tenerlo todo junto, un parámetro que especifica la ejecución secuencial y paralela. Una vez dentro del programa, este sigue los mismos pasos tanto para una ejecución como para la otra. En la Figura 5 se puede ver el workflow general del programa, que vamos a comentar en más detalle a continuación.

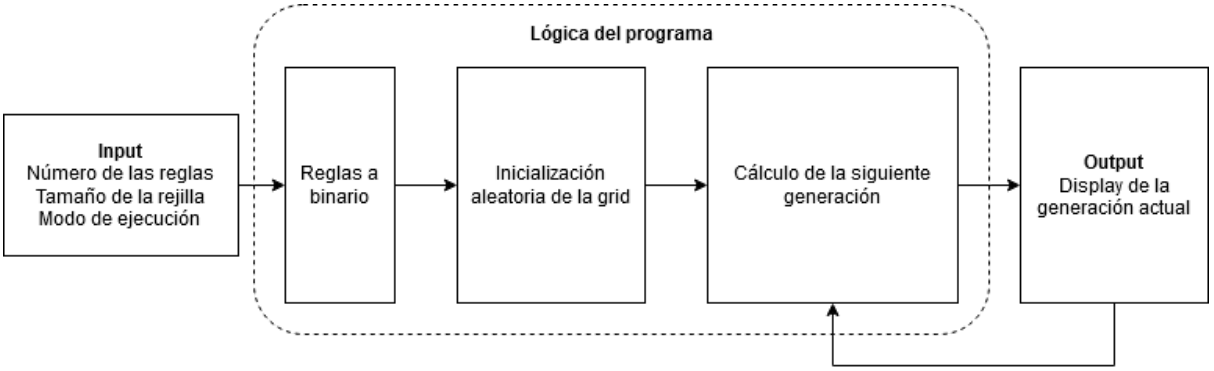

Figura 5. Workflow general del programa.

El primer paso es decodificar las reglas que se van a usar. Para ello se convierte el número decimal que ha introducido el usuario a binario y se inicializa el vector de reglas. El siguiente paso es inicializar la rejilla con valores aleatorios. A partir de ahí, cada generación se calcula iterando por cada una de las células, mirando sus vecinos actuales y calculando a partir de las reglas, el valor de la célula central en la siguiente generación.

2. Implementación

El proyecto consta de cuatro implementaciones diferentes del AC, tal y como se pueden ver en la Figura 6. La primera de ellas es la secuencial. A partir de esa, se implementó una versión con CUDA (v1) usando simplemente llamada al kernel y trabajando sobre las matrices. Luego también sale la versión de CUDA usando shared memory para mejorar la eficiencia. Al comprobar que esta segunda versión iba más rápido que la anterior (como veremos más adelante al mostrar los resultados), pero sin llegar a ser el principal cuello de botella, aplicamos otra optimización encima de ésta para reducir los datos que se tienen que pasar de gpu a cpu en cada iteración, usando cuda 'surfaces' para actualizar la textura directamente en el kernel.

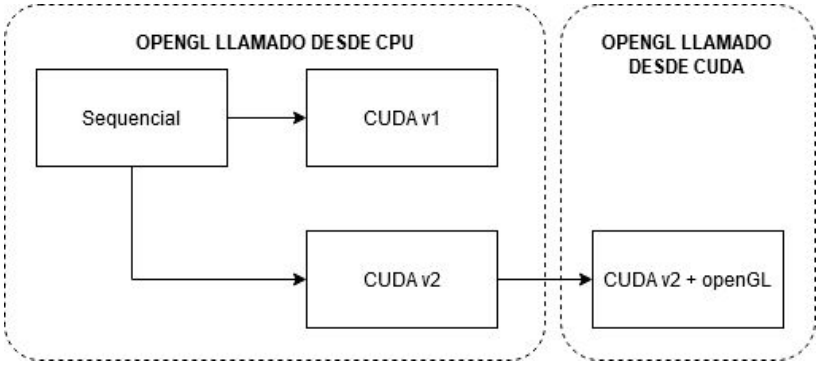

Figura 6. Versiones de código.

Tal y como hemos explicado anteriormente, un AC se basa en calcular la siguiente generación de células a partir del estado de la generación actual y un conjunto de reglas. Ahora, si recordamos la lógica del programa explicada en el apartado 1.4, tenemos que todas nuestras versiones funcionan de la misma forma hasta el bloque de cálculo de la siguiente generación (no incluido). Así pues, codificamos y decodificamos la regla a usar y inicializamos la rejilla inicial con valores aleatorios. Como a la siguiente generación tendremos que consultar la generación actual, usamos dos punteros a matrices, *old_gen* y *current_gen*, para evitar tener que crear una nueva matriz donde meter el resultado cada vez. Estos dos punteros son los que se usan en la llamada a las funciones tanto de calcular la siguiente generación como para después mostrarla por pantalla. Al final del proceso, las direcciones de memoria de los dos punteros son intercambiados, pues la *current_gen* tiene que pasar a ser la *old_gen* para calcular el siguiente step.

No entraremos en los detalles concretos de OpenGL, pero en la siguiente figura 7 podemos ver un resumen de como funciona nuestro programa desde que se actualiza el grid hasta que se crea la imagen con openGL, viendo cual es la informacion clave que necesitamos pasar a los shader de openGL para que se pueda dibujar la imagen por pantalla:

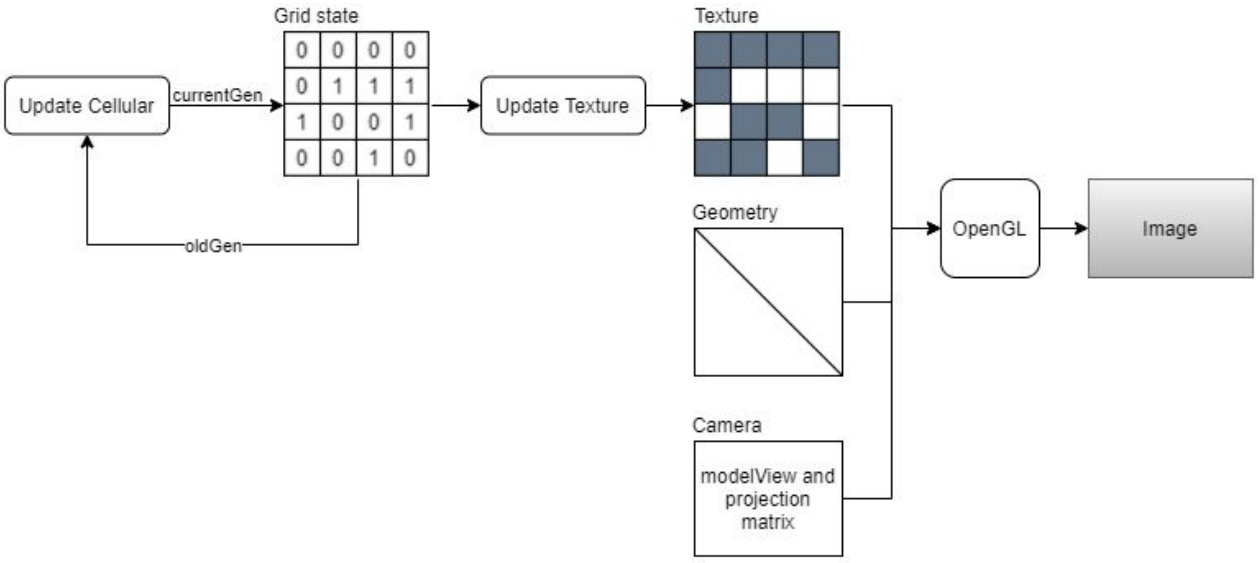

Figura 7. Workflow programa + OpenGL

2.1 Versión secuencial (cpu)

Para la versión secuencial usando la CPU, inicializamos tal y como se ha visto en el apartado anterior (convirtiendo la regla a array binario y inicializando la grid aleatoriamente). Después, en *Update Celular*, se itera por toda la grid y por cada una de las células, se mira el valor de los vecindarios para calcular el índice de la regla que corresponda. Finalmente, se mira el array de reglas y se asigna a la grid de la siguiente generación el resultado correspondiente.

Una vez está calculada la siguiente generación, pasamos a hacer update de las texturas. En este paso lo que se hace es leer el estado actual del celular, es decir una vez este ha sido actualizado y prepara los nuevos valores en la textura que va a tener que ser pintada por OpenGL. Finalmente, esta textura se carga a la GPU y se llama a la función de Draw, que muestra el estado por pantalla.

A partir de ahí el proceso se va repitiendo para cada una de las siguientes generaciones. En la Figura 8 se puede apreciar el flow de este algoritmo una vez la inicialización ya ha sido completada.

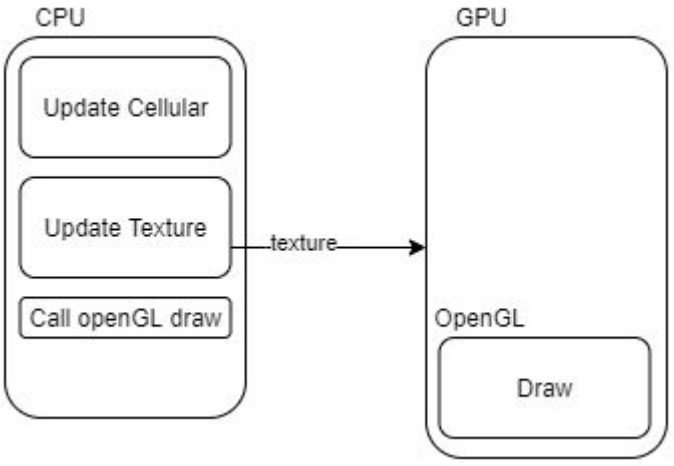

Figura 8. Versión secuencial, main loop.

2.2 Cuda (no interop)

El siguiente paso fue integrar CUDA en el código usado para calcular la siguiente generación del CA. Lo único que cambia respeto a la versión secuencial, es la llamada al Update Cellular, que en este caso, se hace en un kernel de CUDA. El Update de las texturas se sigue ejecutando en CPU, por lo que es necesario obtener el resultado del kernel para actualizar la textura, lo cual no es la mejor opción, como se verá en apartados futuros. En la Figura 9 se muestra el flow básico de esta versión del programa. Como se puede ver, se diferencia del anterior únicamente por la llamada al Update Cellular.

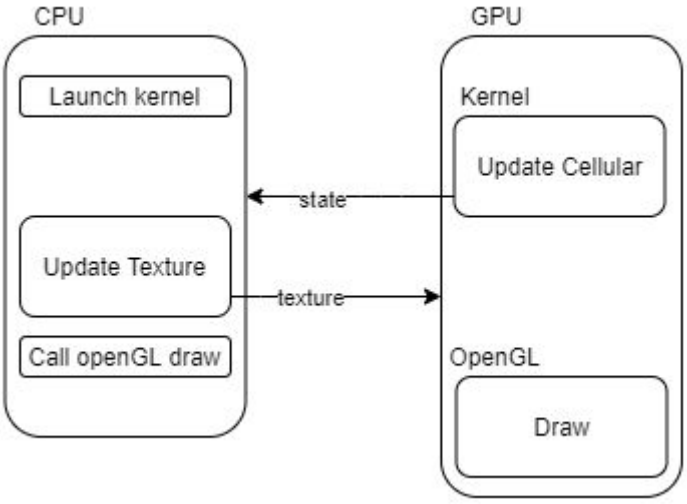

Figura 9. Versión cuda sin comunicación con OpenGL.

2.2.1 Kernel 1

Para la primera versión del kernel, usamos una traducción directa de la versión secuencial. Le pasamos al kernel una referencia a las dos matrices, la de la generación actual y la de la generación a calcular. También le pasamos referencias a las reglas a usar. Así pues, si

considerando que cada thread corresponde a una célula a actualizar, para cada thread el algoritmo básico es el siguiente:

- Saber en qué elemento de la grid nos encontramos.
- Acceder a las células/posiciones vecinas.
- Calcular el índice para el array de reglas a partir de estas.
- Actualizar en el grid de salida el nuevo valor de la célula.

Esta implementación es la más simple que se puede hacer y no conlleva mucha dificultad de programación, pero si pensamos en el rendimiento que esta puede tener nos encontramos un claro problema en el segundo punto. Para cada célula, necesitamos realizar 9 accesos de lectura a la matriz de entrada, 1 al array de reglas y finalmente uno de escritura en la matriz de salida. Realizar tantas lecturas nos hace pensar que puede haber un cuello de botella en esta parte del algoritmo. Para ello, se escribió una segunda versión del código usando memoria compartida, para intentar evitar hacer tantos accesos innecesarios a la memoria principal.

2.2.2 Kernel 2

Como se ha introducido al apartado anterior, el hecho de realizar muchas lecturas a memoria pueden ralentizar el algoritmo. Por este motivo, la segunda versión del código CUDA incorpora el uso de shared memory junto con sincronización entre threads.

El caso ideal usando shared memory sería que todos los accesos de un thread se hiciesen en esta, haciendo un único acceso por thread a memoria principal al inicio solo para cargar el dato. Esto tiene un problema con el AC ya que al estar la grid dividida por bloques, las células de una frontera no pueden acceder a la shared memory de sus vecinas en bloques adyacentes. En la figura 10 se puede ver este problema, donde la célula azul oscura sería una célula frontera del primer bloque, las azul claro las vecinas del mismo bloque y que por tanto están en shared memory y las verdes que pertenecen a otro bloque y no estarían en shared memory.

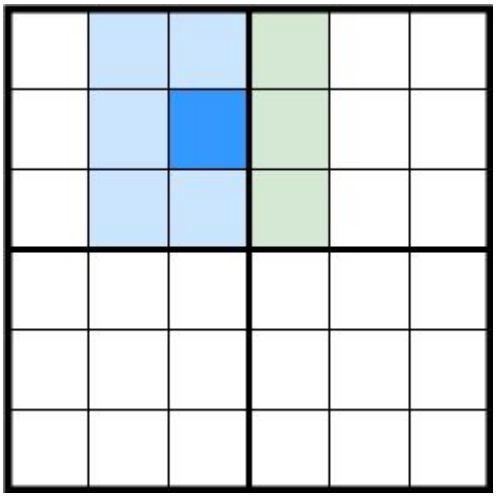

Figura 10. Problemática de adyacencia.

Para solucionar este problema, tuvimos dos ideas en el diseño de esta segunda versión CUDA. La primera idea que tuvimos fue hacer que la shared memory de cada bloque tuviese los vecinos adyacentes de otros bloques, eso es, las células fronteras tendrían que encargarse de cargar a shared memory sus vecinos de otros bloques también. Esto nos daba en el peor de los casos poco más de 1 acceso a memoria principal por thread, pero programarlo daba muchísimos problemas ya que teníamos que poner muchas condiciones para saber a que acceder en función de la célula en la que estemos.

Nuestra segunda opción fue simplemente comprobar si estábamos en una célula frontera o no. En caso de estar en una célula intermedia, se leen todos los vecinos directamente de la shared memory mientras que si nos encontramos en una célula frontera, se leen todos los vecinos de la memoria principal. Al final nos decidimos por la 2a opción, ya que consideramos que la complejidad que añadiría la 1a opción al algoritmo en estos casos no vale la pena si tenemos en cuenta que usamos bloques de 32x32 threads, y de estos 124 threads son frontera, un 12,1% de todos los threads tendrán que acceder a memoria global. Consideramos que conseguir que el 87,9 % de los threads hagan uso exclusivo de shared memory ya es una mejora justificable como veremos en el apartado 3.

2.3 Cuda (interop OpenGL)

La última mejora que hicimos, y la más significativa, como veremos cuando analizemos el rendimiento, fue la de implementar el interop entre Cuda y OpenGL. Esto significa que estos tengan la capacidad de comunicarse/compartir recursos de la propia GPU, sin necesitar al host de por medio.

En nuestro caso, el problema principal es que estamos actualizando la textura en la CPU, eso requiere que en cada update del cellular automata, la GPU tenga que pasar el grid/state hacia la CPU, la CPU actualiza la textura a partir de este, y entonces los datos de esta textura se tiene que volver a pasar a GPU, como hemos visto en la figura 9.

Entonces, nuestro objetivo es actualizar la propia textura en un kernel, en la misma GPU, a partir de los datos del state del C.A que ya estaría en la misma GPU. El problema es que las **texturas** son **read-only**, para solucionar esto tenemos que crear un objeto '**surface**' de cuda. Este es un tipo de textura que puede ser actualizada en run-time por un kernel de cuda. Hay un pequeño overhead necesario al reservar el array de memoria requerido por la surface en cada iteración, pero es irrelevante a medida que aumenta el tamaño del problema.

Con el uso de surfaces ganamos 2 ventajas:

1. **Reducimos prácticamente a 0 el ancho de banda CPU-GPU** , ya que no hace falta mover ni la textura ni el estado del celular entre host y device (solo en la inicialización del programa)
2. **Actualización en paralelo de la textura/surface** en un kernel de cuda, reduciendo el tiempo drásticamente respecto la actualización secuencial de esta en CPU.

En la figura 11 podemos ver la forma que tiene nuestra versión final del programa.

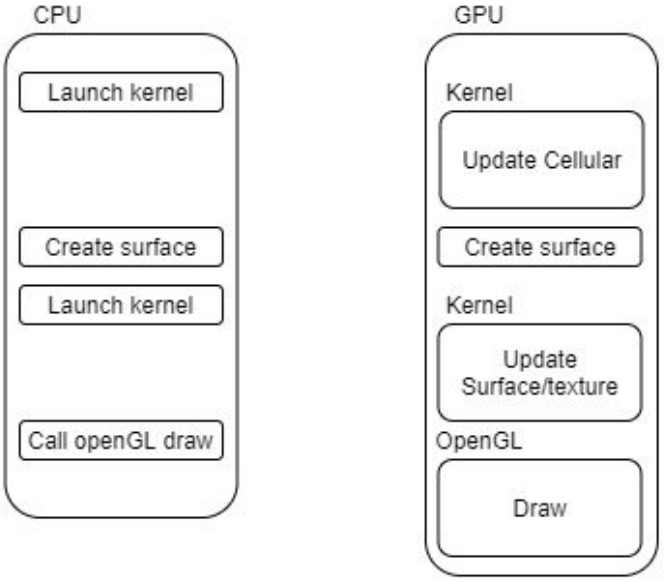

Figura 11. Versión final haciendo uso de surfaces

3. Análisis de rendimiento

Las siguientes pruebas de rendimiento se han hecho en una 1660 ti Max-Q(versión de bajo consumo para laptops) y CUDA 10.2.

3.1. Comparación de kernels

Vamos a empezar observando las diferencias de rendimiento entre los 2 kernels implementados, para actualizar el estado del Cellular Automata, para así justificar la elección del segundo, haciendo uso de shared memory.

En la figura 12 podemos observar el rendimiento de los 2 kernels en función del tamaño de textura (equivalente al tamaño del grid del cellular). Por ejemplo, en el caso de un tamaño de textura 4096x4096 (2^24 células a actualizar por iteración), el primer kernel tarda unos 9.1 ms en media, mientras que haciendo uso de shared memory conseguimos un tiempo de 5.2 ms, ofreciendo un **speedup de 1.75**.

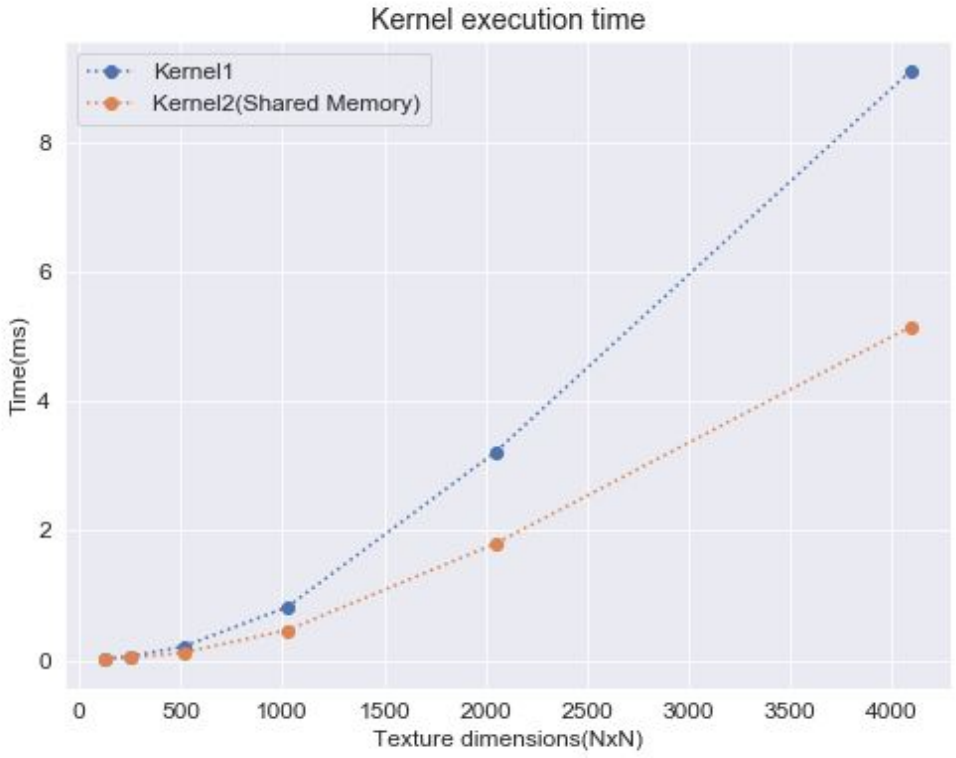

Figura 12. Comparación de tiempos de los Kernels 'Update Cellular'.

Para ganar un poco más de contexto, vamos a calcular los flops y ancho de banda del kernel. Para calcular los flops hemos contado todas las operaciones de coma flotante (*,/,+,-) más los 4 módulos, que los hemos contado com 3 operaciones, teniendo en cuenta que:

$$\% = a - (n * int(a/n)$$

Una vez sabemos las operaciones por threads ejecutado, ya podemos calcular el total de flops y lo dividimos entre el tiempo de ejecución del kernel. Para el ancho de banda los mismo pero contamos accesos de lectura y escritura, en este caso 9 lecturas (vecinos y propia célula) + 1 escritura del resultado final de la célula. Multiplicamos este valor por el total de células y los dividimos por el tiempo de ejecución. Los resultados se pueden ver en la figura 13.

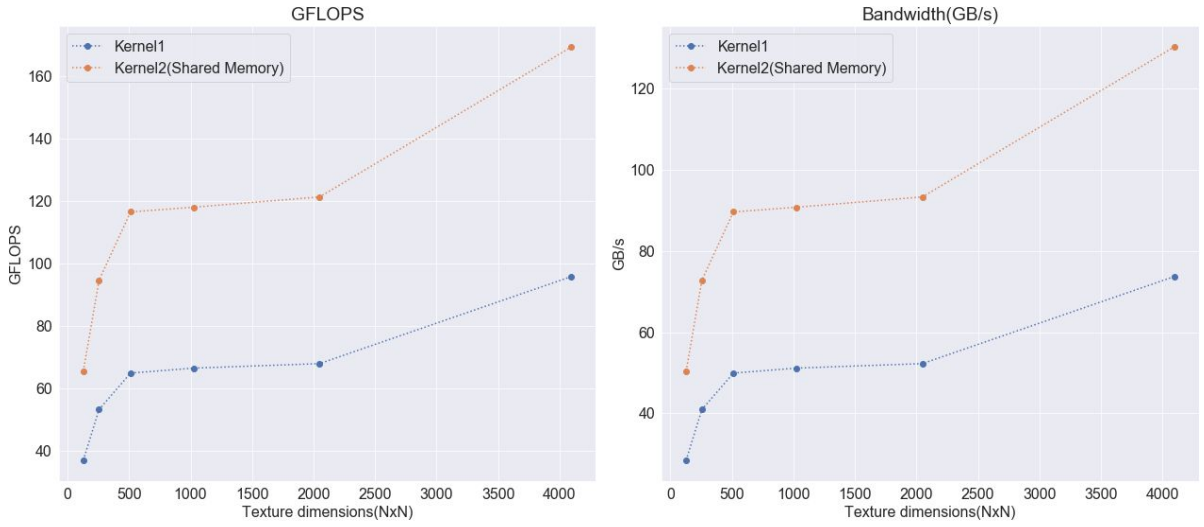

Figura 13. Flops y ancho de banda de los 2 kernels

3.2 Rendimiento general del programa

A continuación vamos a evaluar el rendimiento del programa entero, desde que se actualiza el estado del celular hasta que se pinta por pantalla. En la figura 14 podemos ver el rendimiento del programa con el Kernel 2 del apartado anterior integrado, respecto a la versión secuencial ejecutada ejecutada en CPU.

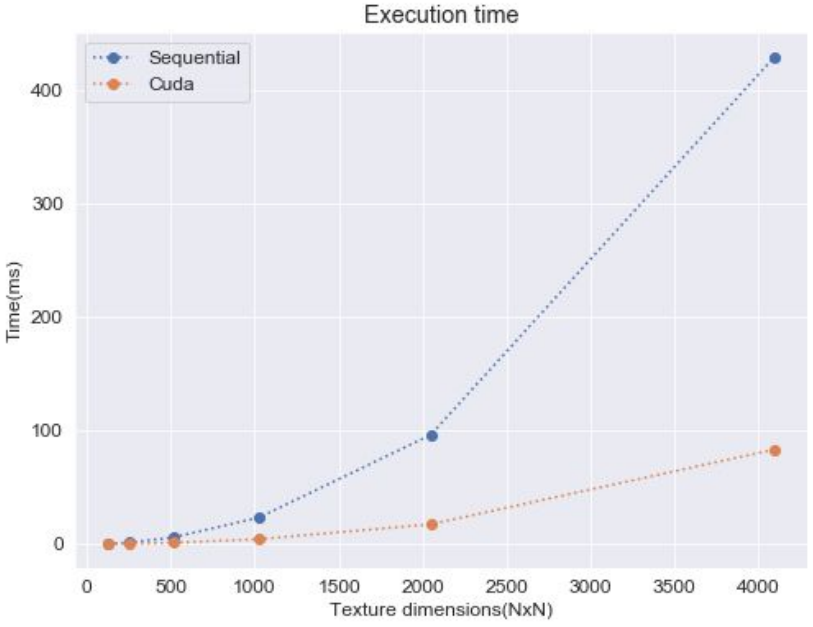

Figura 14. Versión secuencial vs cuda(no interop).

Podemos ver como el aumento de rendimiento es considerable, con un **speedup global del 5.15** en el caso de un tamaño de 4096x4096. Antes de plantearnos hacer más optimizaciones al kernel de 'Update Cellular', es interesante observar la figura 15.

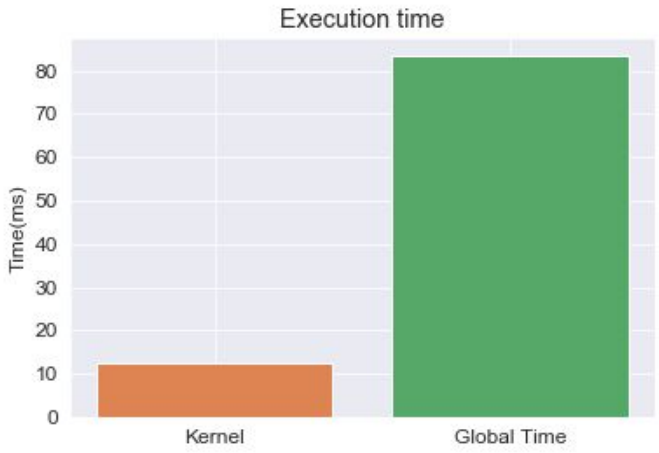

Figura 15. UpdateCellular kernel vs global time.

Vemos cualquier mejora que hagamos a partir de este punto no será muy significativa en el rendimiento total del programa. Esto justifica nuestra decisión para que la siguiente mejora sea implementar el interop cuda-opengl mediante el uso de surfaces. En la figura 16 se puede observar el rendimiento de la versión final.

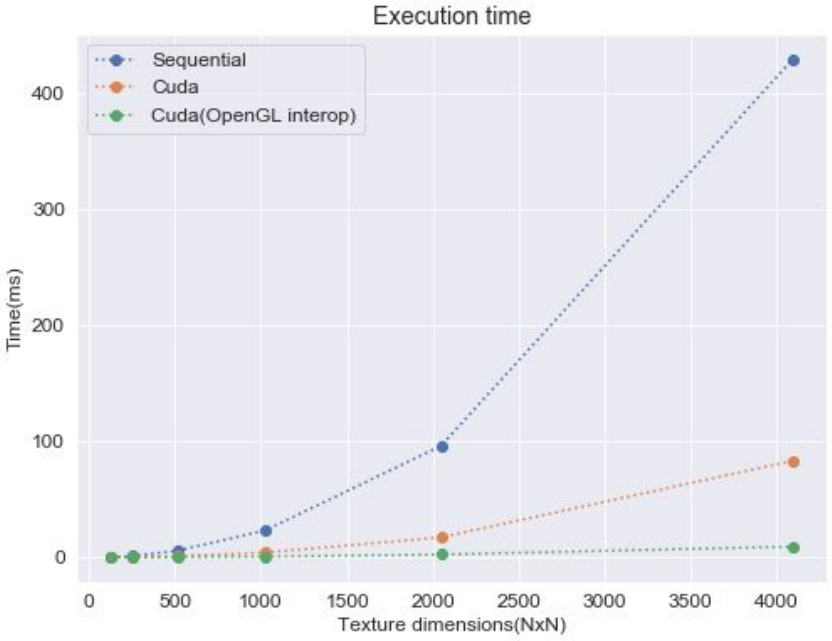

Figura 16. Comparación de los tiempos globales de la ejecución del programa.

En este caso, para un tamaño 4096x4096 el **speedup** es de **46.24** respecto la versión secuencial, y de 8.79 respecto la versión cuda sin interop.

Para finalizar, ahora mismo en el programa tenemos configurado que el C.A se actualice cada 50 ms, para que se pueda visualizar a una velocidad adecuada, pero que tampoco vaya muy lento. Entonces, vamos a ver cual es el tamaño máximo de células posibles antes de que la velocidad de la simulación se vea afectada.

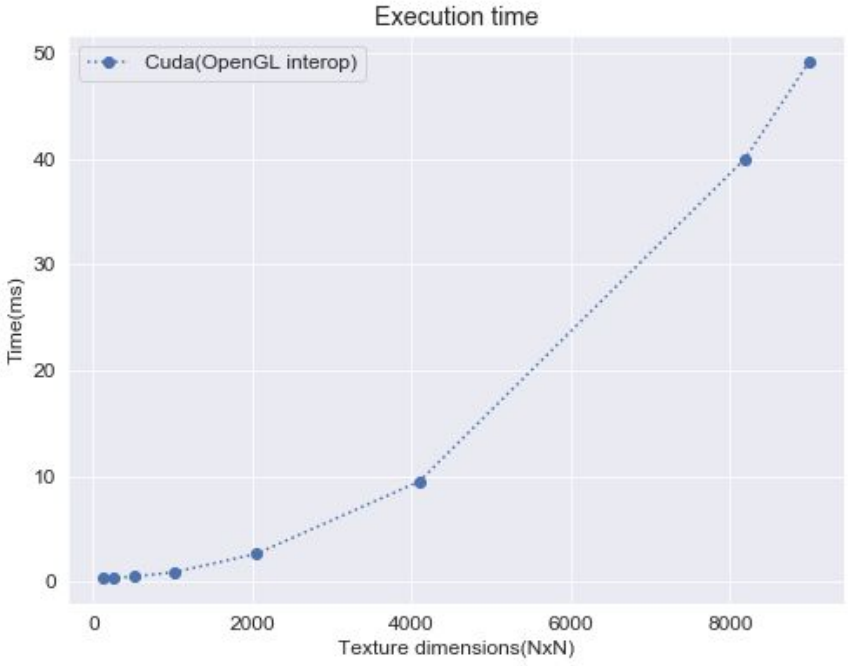

Figura 17. Tiempos de la versión final del programa.

En la figura 17 podemos observar el rendimiento final obtenido, con un tamaño de textura de 9000x9000 obtenemos un tiempo aproximado por iteración de 49.2 ms. El programa es capaz de procesar y renderizar por pantalla sin afectar la velocidad de simulación un total de $81 \cdot 10^6$ células.

Bibliografía

“Cellular Automata problem definition”
https://plato.stanford.edu/entries/cellular-automata/

“Cellular Automata fronteras y variaciones”
https://es.wikipedia.org/wiki/Aut%C3%B3mata_celular

“Two dimensions and beyond cellular automata”
https://www.wolframscience.com/nks/notes-5-2--numbers-of-possible-2d-cellular-automaton-rules/

“Rule integer: How to express cellular automata rules”
https://www.conwaylife.com/wiki/Rule_integer

“Life-Like cellular automatas”
https://www.conwaylife.com/wiki/Cellular_automaton#Life-like_cellular_automata

“CUDA guide: surface objects and memory”
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#surface-memory

“OpenGL textures, parameter settings examples, etc”
https://learnopengl.com/Getting-started/Textures