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

Computación en GPU

Juego de la vida

Integrantes: Joaquín Torres P.
Profesor: Nancy Hitschfeld K.
Ayudante: Sebastián González
Fecha de realización: 1 de noviembre de 2017
Fecha de entrega: 1 de noviembre de 2017
Santiago, Chile

Índice de Contenidos

Lista de Figuras

1. Introducción

El juego de la vida es un autómata celular diseñado por el matemático británico John Horton Conway en 1970. Desde un punto de vista teórico, es interesante porque es equivalente a una máquina universal de Turing, es decir, todo lo que se puede computar algorítmicamente se puede computar en el juego de la vida.

En esta tarea se implementará una variante del juego de la vida que tiene las siguientes reglas:

- Una nueva célula nace en un espacio vacío si a su alrededor existen 3 o 6 células.
- Una célula sobrevive en la próxima generación si hay 2 o 3 vecinas.
- El mundo es cíclico.

En este informe se mostrará los resultados obtenidos en la implementación en paralelo de este juego. Se compararán dos tipos de bibliotecas CUDA y OpenCL, se mostrarán resultados con respecto a la cantidad de células evaluadas y además el speedup logrado al paralelizar el juego de la vida.

Los resultados esperados son que la paralelización logre un speedup de por lo menos 5 veces lo que demora el algoritmo secuencial, además de que cada vez que se aumenta el tamaño del mundo, se analicen más células con los algoritmos paralelos. Como comparación entre los algoritmos de las distintas bibliotecas, se espera que tengan un rendimiento similar.

2. Implementación

En esta sección se describirán las soluciones secuenciales y paralelas, con respecto a la solución del juego de la vida. Todo el código con respecto a la ventana esta fuera del alcance de este informe.

Dado que el mundo es cíclico, se usó la condición de que los vecinos sean el módulo del ancho o alto dependiendo de la celda que se evalúe.

En un modo de ejemplo, en un mundo de alto 4 y ancho 4, en la celda (1,1) su casilla superior será la celda (1,4). Esto lo obtenemos con la fórmula (y + alto - 1) % alto. Para el caso de ver la casilla a la derecha se usa la fórmula (x + ancho) % ancho. Usando el módulo de ancho y alto damos "vuelta" el mundo y obtenemos las casillas que son las vecinas correspondientes en este mundo cíclico.

Para el mapeo del mundo a una estructura, se usó un arreglo de tamaño alto*ancho. El elemento (i,j) en el mundo, se accede con el índice i*alto + j en el arreglo.

2.1. Implementación secuencial

Esta implementación recorre toda la matriz que representa al mundo del juego, en cada una de las celdas suma el valor de las celdas vecinas en un acumulador, y finalmente se evalúan las reglas de este juego, con esto se decide si la celda vive o muere. Se guarda el estado obtenido en otra matriz

para no sobre escribir los valores del mundo en el estado actual.

Una vez que se recorrió toda la matriz para obtener los resultados del nuevo estado, se copian los resultados en la matriz original.

2.2. Implementación paralela en CUDA

Para esta implementación tenemos que tener el cuidado de mandar las variables necesarias para que se ejecuten en la GPU.

En CUDA primero se debe pedir memoria en la GPU para poder manejar el arreglo del juego. Por lo que se usa cudaMalloc tanto para pedir memoria del arreglo que se debe leer como el que se escribe el estado final.

Después de tener memoria en la GPU se copia el arreglo desde CPU hacia la GPU con el método cudaMemcpy y el flag cudaMemcpyHostToDevice.

Ya teniendo estos dos pasos ejecutamos el kernel, con dos parametros, el número de bloques que se correrán y el tamaño de cada bloque que simboliza la cantidad de threads en cada bloque. También se le entrega como argumento los punteros a los arreglos que fueron copiados a la GPU.

Dentro del kernel se ejecuta los mismos pasos que la implementación en CPU, solo que cada thread se encarga de solo una celda, por lo que se calculan las condiciones y se pega el resultado en el arreglo del estado final. Se tiene que tener el cuidado de que el thread no este en una celda fuera del arreglo, por lo que hay una condición al inicio del kernel. Para obtener el identificador de en que posición del arreglo se encuentra, se usa la siguiente ecuación:

```
const int k = blockIdx.x * blockDim.x + threadIdx.x;
```

Con esto se determina en que bloque esta el thread actual, lo multiplica por la dimensión actual y le suma el identificar del thread actual, lo que nos entrega la posición global en el arreglo, que es lo que usamos para calcular las posiciones de los vecinos y actualizar estado de la celda.

Con el arreglo ya calculado se copia el resultado desde la GPU hacia la CPU con el mismo método que en la parte que se envían los datos, pero con el flag cudaMemcpyDeviceToHost. Ya teniendo los datos actualizamos el estado de la matriz actual obteniendo el resultado de la iteración.

2.3. Implementación paralela en OpenCL

En esta implementación a diferencia de la anterior en CUDA, se debe setear en qué plataforma y que dispositivo se usarán. En esta implementación se usará la plataforma más reciente de OpenCL, y se usará el dispositivo Nvidia como primer lugar y si no esta usará el que esté disponible.

Una vez cargado el dispoitivo se carga el archivo que contiene el kernel que se debe ejecutar, en OpenCL se pide entregar un string con toda la implementación, por lo que se lee el archivo línea a línea y se entrega un string con la implementación del kernel. Ya teniendo las configuraciones

iniciales se crea el programa que se ejecutará en cada iteración.

En cada iteración se copiarán a un buffer las diferentes estructuras que se usarán en la GPU para correr el kernel. Después de guardar lo que se va a usar se carga el kernel en el programa y se setea que buffer es que argumento en el kernel que se esta ejecutando. Al tener todo bien configurado se ejecuta finish(), lo que inicia la iteración en la GPU.

En el kernel cada thread se encargará de un elemento del arreglo, cada uno toma el indice del arreglo con el método get_global_id(0), y se procede a accumular los valores de los vecinos para finalmente con una operación lógica de acuerdo a las reglas del juego obtener el estado que tendrá esa celda.

Una vez que se tiene el resultado se copia en una variable local en CPU y se copia el estado a la matriz que tiene el mundo actualmente.

2.4. Variaciones en configuración

Para las variaciones en configuración se eligieron las siguientes configuraciones:

- Usando ifs para preguntar si la vecindad está viva.
- Usando tamaños de bloque tanto múltiplos de 32 como no múltiplo de 32.

Para la primera configuración se cambia la lógica de como se ven los vecinos y se pregunta en cada uno de los vecinos si es que estan vivos, con esto se cuenta la cantidad de vecinos vivos.

```
Sin If
sum += A[right*H + j];
```

```
Con If
if (A[right*H + j])
        sum++;
```

Para la segunda configuración se eligió como no múltiplo de 32, tamaño de bloque de 8. Para así comparar si es mejor tener un mayor tamaño con respecto a potencias de 2. Además de que OpenCL tiene la condición de que el tamaño del bloque debe dividir el tamaño del arreglo que se paraleliza.

En OpenCL se cambia el tamaño de los bloques en la línea:

```
queue.enqueueNDRangeKernel(kernel_add,
        cl::NullRange, cl::NDRange(n), cl::NDRange(blockSize));
```

Donde n es el tamaño del arreglo a procesar en paralelo y blockSize es el tamaño del bloque. Por lo que en OpenCL cambiamos blockSize para cambiar la configuración.

En CUDA se entrega como parametro al kernel la cantidad de bloques a correr y el tamaño de los bloques, donde el tamaño de los bloques indica la cantidad de threads en cada bloque. Esto se ve en la línea:

```
gridSize = (int)ceil((float)n/blockSize);
deviceIterationNotIf<<< gridSize, blockSize >>>(d_c, d_a, H, W);
```

Donde n es el tamaño del arreglo, blockSize el tamaño del bloque y gridSize es la cantidad de bloques que caben en el arreglo. Dado que se pide el ceil de la división, se debe tener precaución de preguntar si el índice no esta en el arreglo, es decir, es mayor al tamaño del arreglo.

3. Resultados experimentales

En esta sección se mostrarán los resultados de dos distintas mediciones, speedup y para el número de células evaluadas por segundo.

El computador utilizado para correr los experimentos tiene las siguientes características:

- Procesador: Intel Core i5-7200U, 2.50 GHz
- GPU: Nvidia GeForce 940MX

Para la configuración de ifs se usaron los siguientes tamaños para N: 1024 (32*32), 4096 (64*64), 6561 (81*81), 22500 (150*150), 90000 (300*300), 250000 (500*500), 1000000 (1000*1000). Para la configuración de bloques, dada la restricción de que el arreglo sea divisible por el tamaño del bloque, se usaron los siguientes tamaños para N: 1024 (32*32), 4096 (64*64), 16384 (128*128), 65536 (256*256), 262144 (512*512), 1056784 (1028*1028).

3.1. SpeedUp

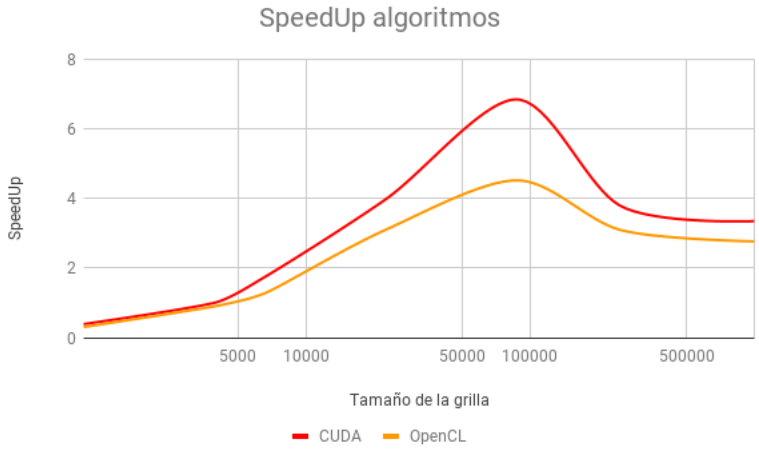

Figura 3.1: Speedup en el experimento sin if

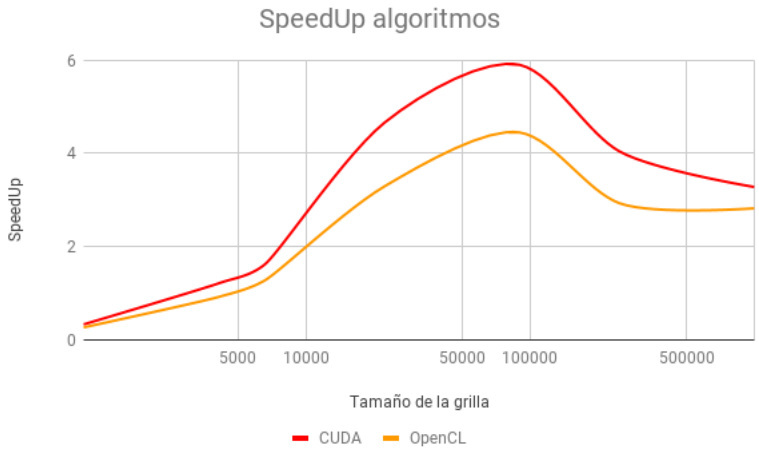

Figura 3.2: Speedup en el experimento con if

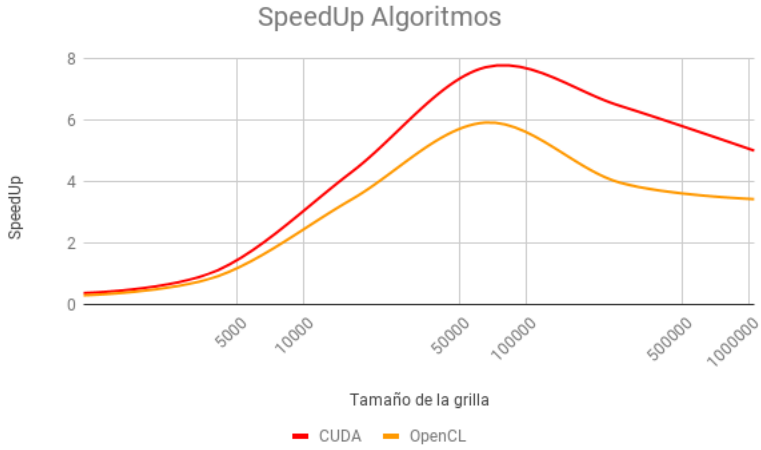

Figura 3.3: Speedup en el experimento con bloque tamaño 32

Figura 3.4: Speedup en el experimento con bloque tamaño 8

3.2. Células evaluadas por segundo

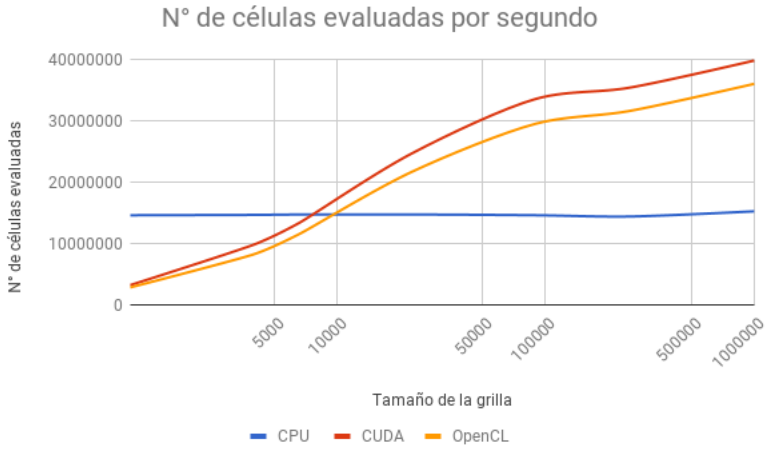

Figura 3.5: Células evaluadas por segundo en el experimento sin if

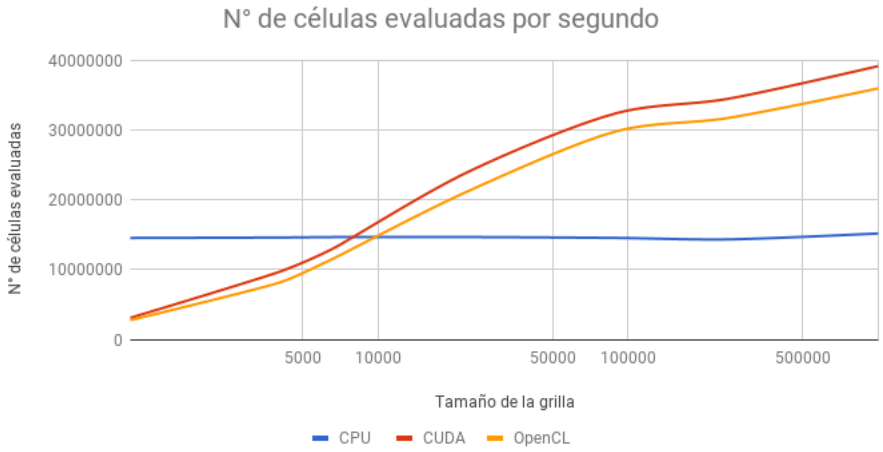

Figura 3.6: Células evaluadas por segundo en el experimento con if

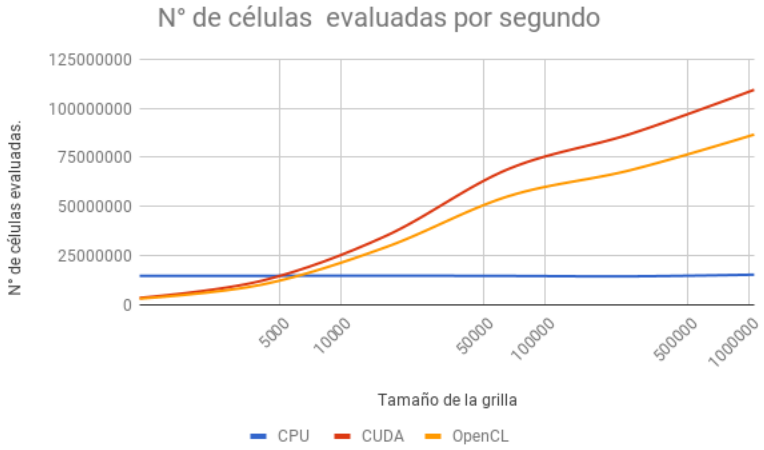

Figura 3.7: Células evaluadas por segundo en el experimento con bloque tamaño 32

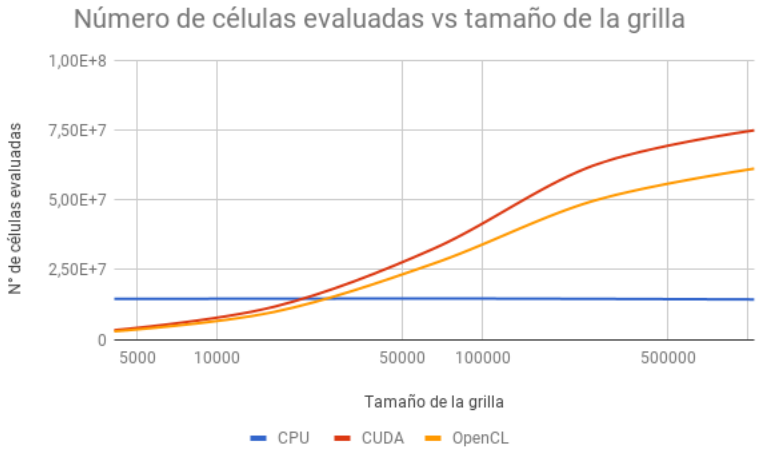

Figura 3.8: Células evaluadas por segundo en el experimento con bloque tamaño 8

3.3. Comparación entre implementación con If y sin If

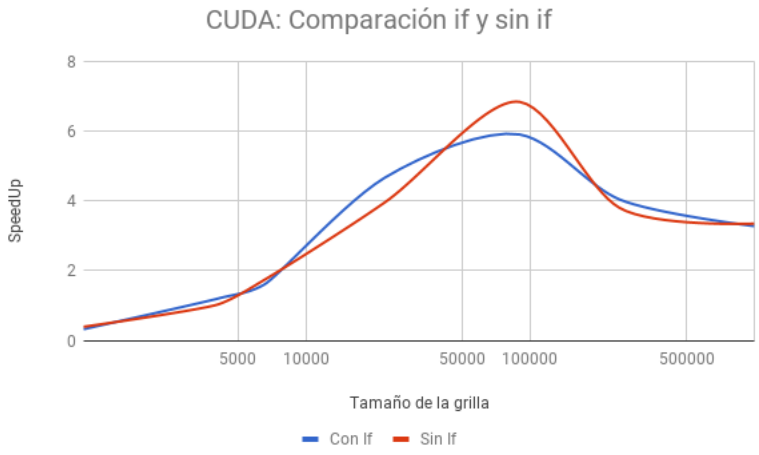

Figura 3.9: Speedup con CUDA

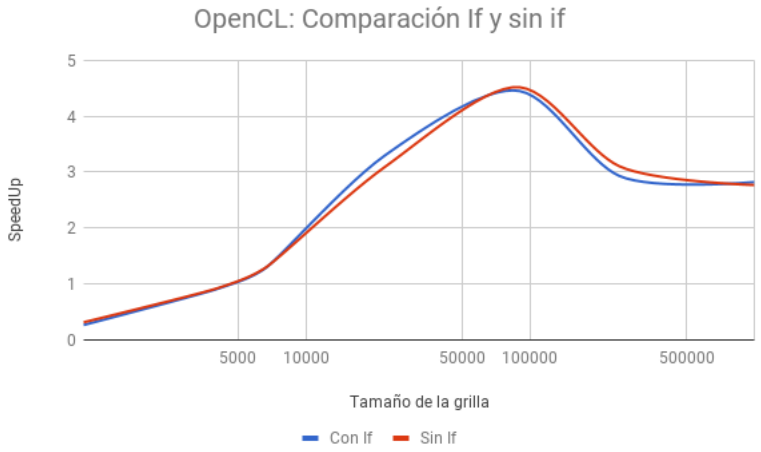

Figura 3.10: Speedup con OpenCL

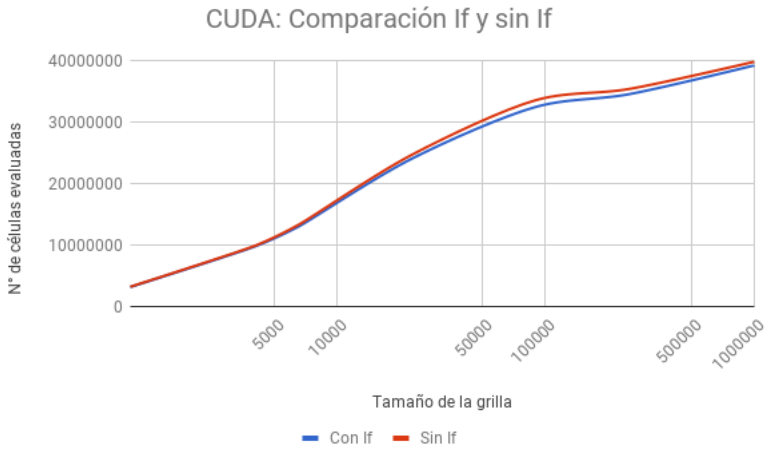

Figura 3.11: Células evaluadas por segundo, CUDA

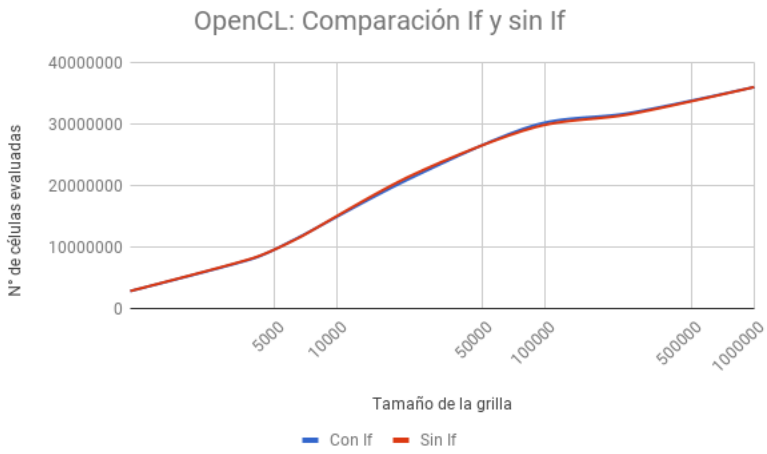

Figura 3.12: Células evaluadas por segundo, OpenCL

3.4. Comparación entre implementación bloque tamaño 8 y bloque tamaño 32

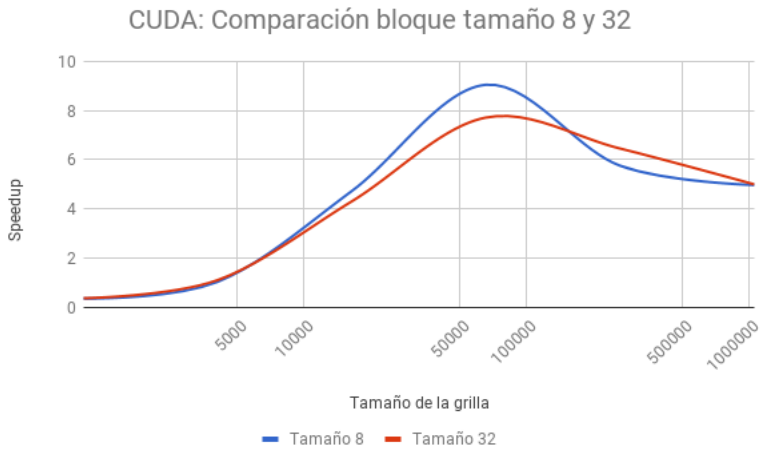

Figura 3.13: Speedup con CUDA

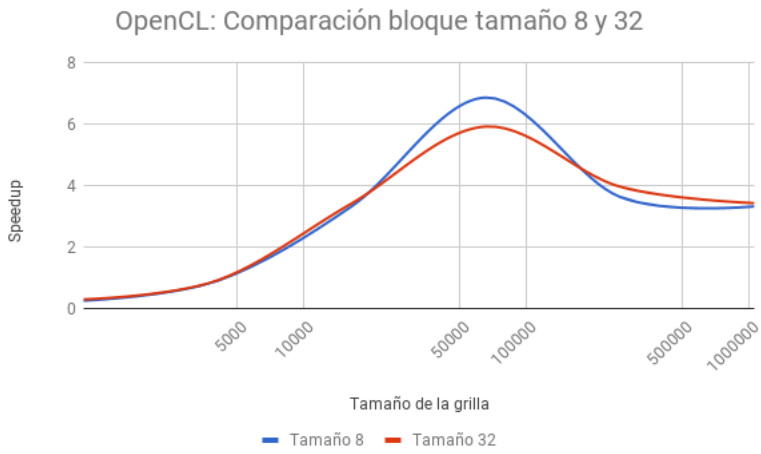

Figura 3.14: Speedup con OpenCL

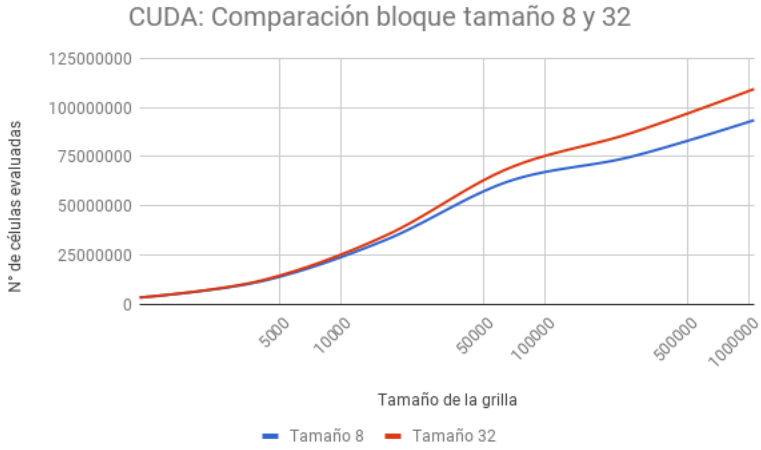

Figura 3.15: Células evaluadas por segundo, CUDA

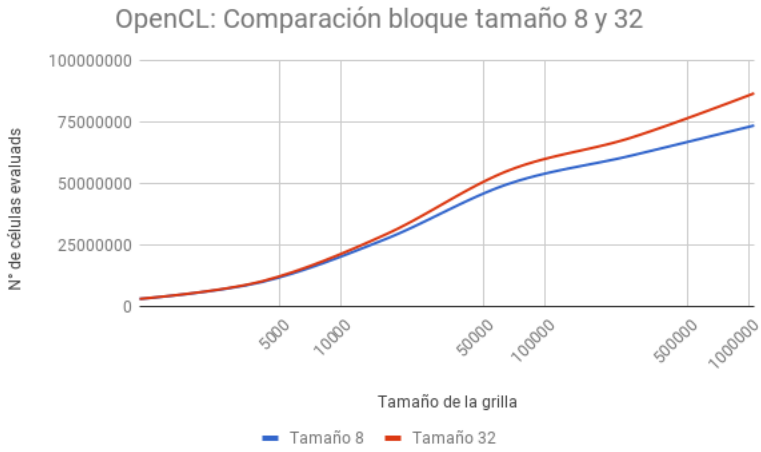

Figura 3.16: Células evaluadas por segundo, OpenCL

4. Análisis de resultados

De los resultados obtenidos, se puede observar que el speedup conseguido por las bibliotecas llega a un valor de 6x y hasta 7x. Comparando las dos bibliotecas, se puede ver una notable superioridad de CUDA frenta a OpenCL.

Viendo las gráficas de células evaluadas por segundo, se puede observar que a partir de un umbral (7000) las implementaciones en GPU superan a las de CPU. Aproximadamente el tamaño necesario para que las implementaciones paralelas superen a la secuencial es de 7000 células.

Para las configuraciones que se puso en estos experimentos, la configuración de ifs y no ifs no se ven cambios notables en el speedup o en la cantidad de células evaluadas por segundo.

Para el experimento con respecto al tamaño del bloque. Se puede observar que el tamaño 32 supera en speedup a la implementación con bloque tamaño 8, pero la implementación con bloque tamaño 8 logra un mejor speedup máximo, pero evalúa menor cantidad de células por segundo.

5. Conclusiones

De los resultados obtenidos se puede concluir que la biblioteca CUDA es mejor en términos de cálculos y speedup que OpenCL, esto se observa en los gráficos que comparan las bibliotecas mencionadas.

Para las configuraciones, entre usar if y no usar if no implica mucha diferencia en rendimiento. Para los tamaños de bloques es mejor usar bloques de tamaño 32 o multiplo de ella, y con lo analizado dependiendo del tamaño de la matriz sería mejor usar bloques de tamaño 8.

Se aprendió a programar en GPU, tanto en CUDA como OpenCL, lo que nos da una nueva herramienta para el futuro.

6. Fuentes

[1] Conway's Game of Life
https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

[2] Tutorial simple OpenCL
http://simpleopencl.blogspot.cl/2013/06/tutorial-simple-start-with-opencl-and-c.html

[3] Repositorio del proyecto
https://github.com/juakotorres/JuegoDeLaVida

[4] Excel con los resultados
https://docs.google.com/spreadsheets/d/1Bp_KM5xz_hd0LYFYkb2AZQckZg70UKrl16cngTu-gsM/edit?usp=sharing