UNIVERSIDAD DE CHILE

Universidad de Chile

CC7515-1

Computación en GPU

Tarea 3: Juego de la Vida

Autor:
Bastián Inostroza
Cristian Tamblay

31 de diciembre de 2018

Índice

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.

Desde su publicación, ha atraído mucho interés debido a la gran variabilidad de la evolución de los patrones. Se considera que la vida es un buen ejemplo de emergencia y auto-organización. Es interesante para los científicos, matemáticos, economistas y otros observar cómo patrones complejos pueden provenir de la implementación de reglas muy sencillas.

En esta tarea, se realizan tres implementaciones de este, juego utilizando CUDA, OpenCL y serial (en CPU) con el objetivo de comparar el desempeño de estas, considerando las diferencia inherentes de utilizar una GPU, versus utilizar una CPU.

Se espera que los algoritmos empleados en CUDA y OpenCL son idénticos que los resultados sean parecidos. Además respecto a la CPU se considera que la diferencia debiese ser notable, especialmente con matrices grandes, ya que el problema es paralelo en datos, y el tamaño de la matriz al aumentar algún lado crece bastante (en celdas).

1.1. Consideraciones

Las implementaciones en GPU fueron ejecutadas en una tarjeta *NVIDIA GTX 1050 @1493MHz* con 2GiB de memoria global y direcciones de 64 bits, esta tarjeta corre en el sistema operativo *Linux Mint 19* con los drivers oficiales de NVIDIA en la versión 410.48. Como dato agregamos que este sistema posee 16GB de memoria RAM, y un procesador *Intel Core i7-7700HQ CPU @ 2.80GHz*.

La implementación serial fue ejecutada en un procesador *Intel Xeon E5620@2.40GHz* en el servidor **Anakena**, propiedad del DCC. Esta además está escrita en *C++*.

Las implementaciones en CUDA y OpenCL están escritas en *C++* y *C* respectivamente, utilizando **OpenCL 1.2** y **CUDA 10.0.141**.

2. Implementaciones

Todas las implementaciones poseen una función **main** que corre los experimentos y registra los resultados en un archivo `.csv`, y una función **run** que inicia una instancia del juego de la vida, y recibe el tamaño del *lado* del cuadrado, la cantidad de *iteraciones* y un valor que regula la cantidad de threads por bloque.

Para la explicación omitiremos por completo la función **main**, ya que realiza lo mismo en cada implementación. Además se destaca que todas las mediciones de realizaron luego de un *calentamiento* de 5 ejecuciones del mismo juego.

Además el juego de la vida se ejecuta en un "mundo" (o matriz) cuyos bordes son continuos, evitando así problemas de borde, al estilo del juego clásico Pac-Man de NAMCO.

2.1. Serial

Esta implementación escrita en *C++* del juego de la vida corre en un solo hilo de ejecución, y posee una función principal **run** que configura la ejecución del juego de la vida, pidiendo memoria para dos arreglos, el primero almacena la información actual (origen) y el otro los resultados del juego de la vida, además corre en un ciclo las iteraciones del juego de la vida, representadas por la función `computeIterationSerial()`. Cabe destacar que las variables son globales, y son *inicializadas* por la función **run**.

Dado que la matriz está representada por un arreglo, `computeIterationSerial()` tiene dos ciclos for anidados que simulan recorrer la matriz, accediendo al arreglo de la forma $array[y_1 + x]$, donde y_1 es la fila de la matriz y x corresponde a la columna de esta. Además calcula las filas y columnas correspondientes a los vecinos. Finalmente, delega el calculo de las cantidad de vecinos, para escribir el resultado en el arreglo de resultados.

`countAliveCells()` calcula la cantidad de vecinos accediendo sus posiciones en el arreglo de origen, donde la existencia de un vecino está representada por el valor 1, y la ausencia por el valor 0, y retorna la cantidad de vecinos.

Finalmente, los arreglos de origen y destino son intercambiados, dejando la información actualizada en el arreglo de origen, y dejando todo listo para realizar una iteración nueva.

2.2. CUDA

En este caso **run** solicita memoria tanto en *host* como en *device* e inicializa las variables que fueron declaradas globalmente, genera de forma aleatoria las vida en las celdas iniciales y pasa la tanto la configuración como los datos relevantes como argumentos a `runSimpleLifeKernel`, un ejemplo de dato relevante es la cantidad de threads que se correrán, información que luego es utilizada para determinar cuantos bloques se utilizarán, en caso de ser menos que el máximo disponible.

`runSimpleLifeKernel` calcula la cantidad de bloques a utilizar, y ejecuta el *kernel* la cantidad de veces indicada, intercambiando los buffers en cada iteración para actualizar los datos.

El *kernel*, llamado `simpleLifeKernel`, para cada celda calcula la cantidad de vecinos mediante la suma de estos, escribiendo en el arreglo de resultados, un 1 si hay vida en la celda, un 0 si no.

2.3. OpenCL

En esta implementación **run** solicita memoria en *host*, crea los buffers necesarios para el dispositivo, además de ~~pasar por el infierno de~~ cargar la información del contexto, crear y construir el programa, crear el kernel, y liberar todos los elementos de memoria una vez terminado el programa. Una vez configurado y creado todo lo necesario para correr el kernel, se entra en el *loop* (bucle) principal, correspondiente a cada iteración dentro del juego, donde se pasa la información al dispositivo escribiendo en los *buffer* correspondientes, pasando los argumentos al dispositivo y ejecutando el kernel con la cantidad de procesadores indicada. Finalmente intercambia los arreglos, para actualizar la información y permitir reiniciar el ciclo.

El *kernel*, ubicado en el archivo `lifeKernell.cl`, para cada celda calcula la cantidad de vecinos mediante la suma de estos, escribiendo en el arreglo de resultados, un 1 si hay vida en la celda, un 0 si no.

3. Resultados Experimentales

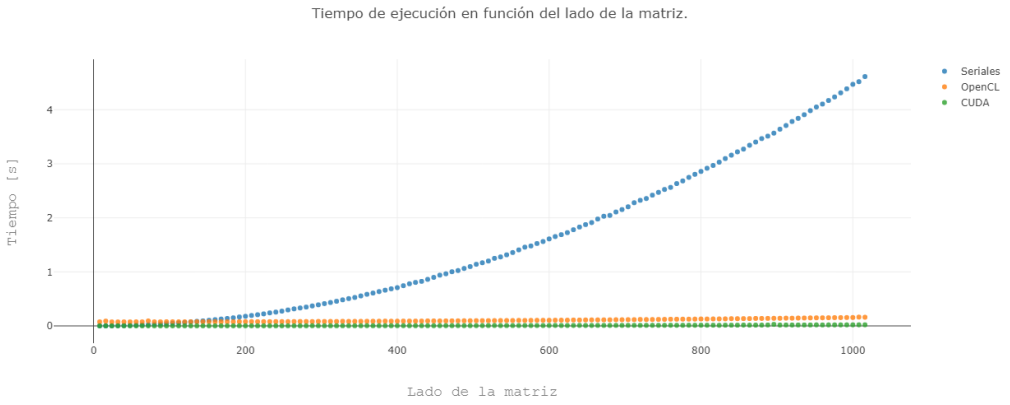

Figura 1: Tiempos de ejecución para cada una de las implementaciones

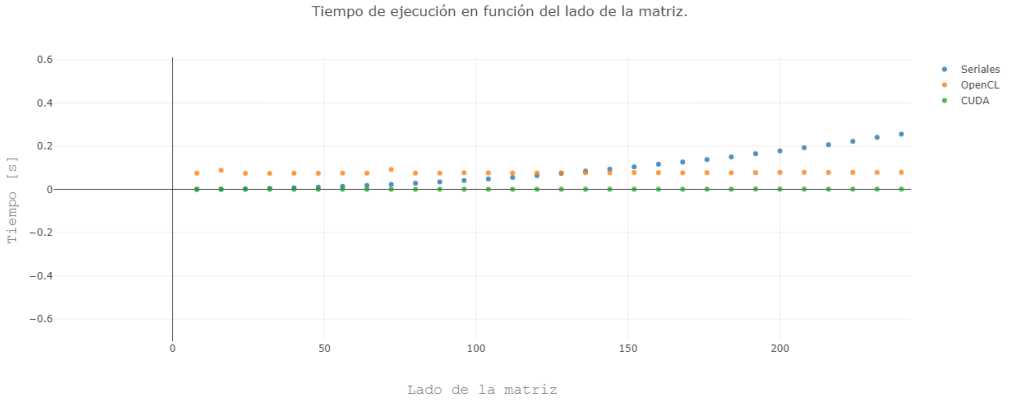

Figura 2: Tiempos de ejecución, detallando cuando serial deja de ser eficiente

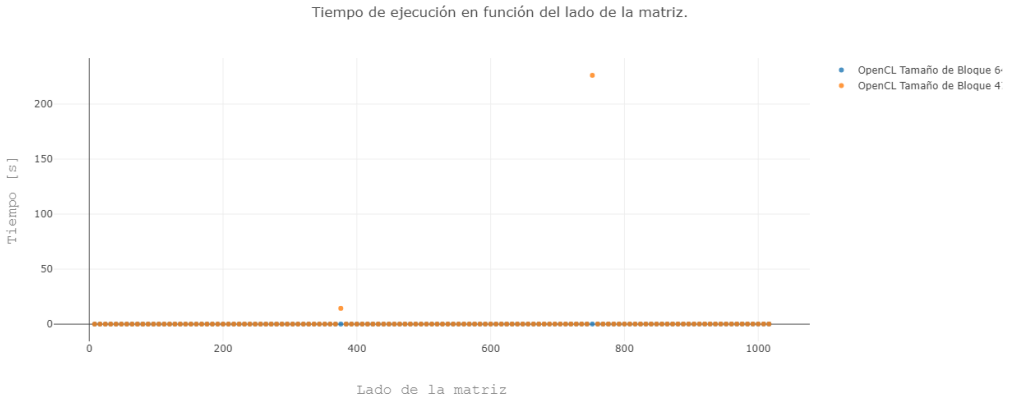

Figura 3: Tiempos de ejecución para bloques de tamaño 47 (no óptimo) y 64 en OpenCL

OpenCL presenta dos outliers, que se repiten en distintas realizaciones del experimento, para el mismo lado de matriz, descartando así que sean aleatorios.

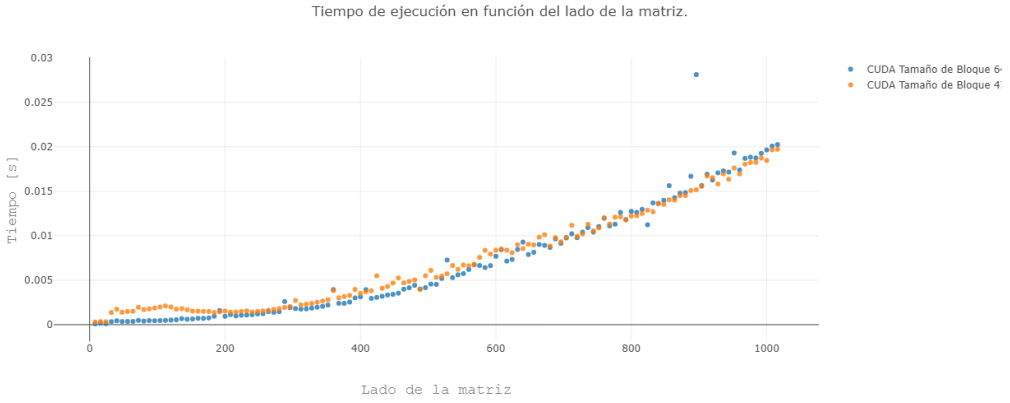

Figura 4: Tiempos de ejecución para bloques de tamaño 47 (no óptimo) y 64 en CUDA

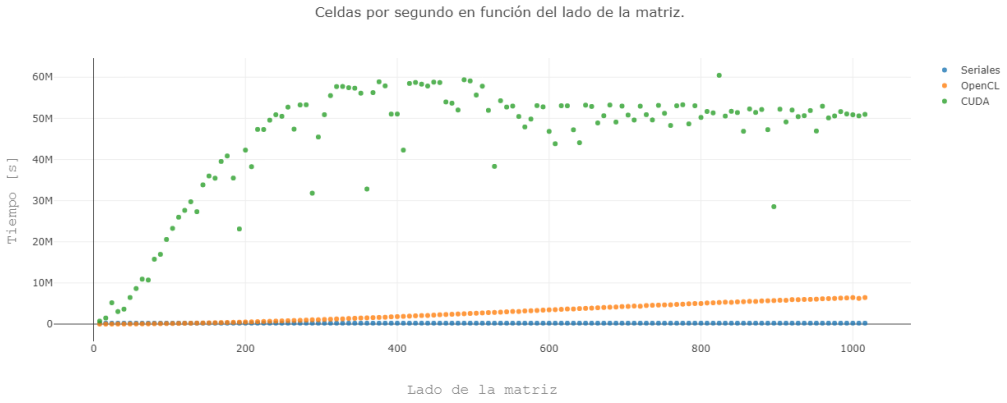

Figura 5: Celdas procesadas por segundo para la configuración óptima en cada una de las implementaciones

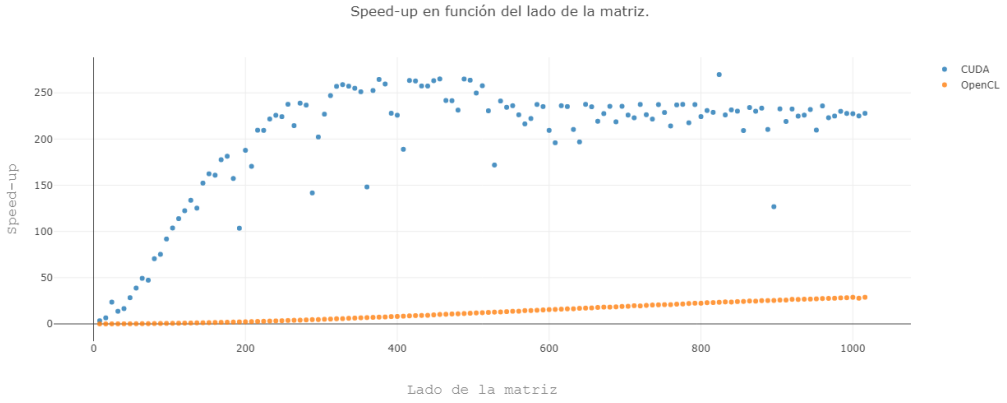

Figura 6: Speed-up para la configuración óptima en cada una de las implementaciones

4. Análisis

OpenCL se vuelve mejor que serial en $n = 130$, CUDA se vuelve mejor que serial en $n = 25$ (Figura 2) (en tiempos de ejecución, tiempos incluyen traspaso de memoria a GPU). Se ve que CUDA tiene un desempeño excepcional en todas las pruebas, logrando Speed-ups de hasta 250x. En cambio nuestra implementación de OpenCL se queda bastante atrás, logrando Speed-ups de sólo 35x (figura 6).

Además, notamos que en configuraciones no óptimas (como numeros que no fueran múltiplos de la arquitectura, como 32 o 64 en este caso) CUDA de comportó prácticamente de igual manera (figura 4), sin embargo, OpenCL presentó consistentemente en las pruebas dos ejecuciones **muy ineficientes**, por motivos que se desconocen (figura 3).

obre el rendimiento de CUDA respecto a OpenCL, destacamos que el *speedup* de CUDA es muy explosivo, alcanzando rápidamente la estabilidad, con un comportamiento logarítmico, lo que le permite ser consistentemente más raápido que OpenCL, que crece de una forma que pareciera ser una cuadrática muy cercana a la linealidad (en la figura 1 podemos notar como al aumentar el lado, OpenCL se va haciendo notablemente más lento).

5. Conclusiones

CUDA mostró tener un rendimiento muy superior al de OpenCL, aún considerando que el primero solo funciona en GPUs de la marca NVIDIA el rendimiento es notablemente superior (figuras 1 y 5). Además, CUDA tiene mucho menos azúcar sintáctico y es más ameno para trabajar, ahorrando tiempo del programador. OpenCL demostró su portabilidad ya que fue probado en un computador sin tarjeta de video dedicada y funcionó sin problemas.

Destacamos que en algún momento cercano a la matriz de $580x580$ la cantidad de celulas procesadas por segundo baja respecto al máximo (figura 5), y se mantiene constante al aumentar el lado de la matriz, a nuestro juicio esto se debe a que la cantidad de bloques alcanzó su máximo, y a partir de ese momento utiliza un solo thread por celda. Diferenciandose de la sección anterior (lado entre 350 y 580) en que usa más de un thread por celda para algunas celdas, acelerando un poco las sumas (con los 8 vecinos) realizadas por bloque con memoria local (L2).

Dado que el algoritmo implementado para las dos versiones es análogo, Donde se modificaron solo las cosas que son propias de cada API, concluimos que la diferencia

en *performance* se debe a una limitación de OpenCL, sin descartar que nuestra implementación de esta no tenga la configuración óptima, el comportamiento que posee es tan inferior que nos permite descartar que se deba *solo* a problemas en la configuración en la cantidad de bloques o hilos de ejecución.

En este fenómeno cabe destacar que al tener OpenCL con los drivers de NVIDIA, este no reconoce a la CPU como dispositivo (no se encuentra al consultar la información del dispositivo, entregando este siempre la GPU dedicada), por lo que descartamos en primera instancia que la ejecución se haya hecho en la CPU. A pesar de esto, la *performance* de OpenCL en nuestra opinión se parece mucho más a la ejecución serial que a la de CUDA. A pesar de esto, los tiempos de ejecución son definitivamente más parecidos a los de CUDA que al serial, lo que nos ayuda a descartar que OpenCL corra en CPU. Sin embargo, nuestros conocimientos específicos sobre OpenCL no nos permiten explicar este comportamiento.

Nuestra experiencia tanto en instalación, programación y ejecución de OpenCL y CUDA nos permite desincentivar la utilización del primero en beneficio del segundo, dada su fácil instalación, su facilidad de configuración, tiempo de programación (coding), y desempeño relativo.

Para complementar esta discusión, destacamos una discusión relacionada, que nace a partir de los resultados en este paper con simulaciones de fotones en GPUs, en el cual se obtienen tambien diferencias significativas de desempeño entre CUDA y OpenCL. Uno de los autores del paper, inicia un hilo consultando cual podría ser una causa de esta diferencia en el foro de desarrolladores de NVIDIA, del que se puede aventurar que la causal de esta diferencia serían optimizaciones del compilador de NVIDIA NVCC respecto al compilador ocupado con OpenCL, tanto en el uso de registros, y memorias L2 y L3 de forma óptima, como también optimizaciones sobre los *if*, en las cuales al parecer CUDA estaría varios años más adelante en desarrollo.

6. Fuentes

Las implementaciones en GPU y CPU están basadas en la implementación dada en el enunciado hechas por Marek Fiser, donde en el caso de las implementaciones de GPU solo está el kernel en CUDA, y el de OpenCL fue realizado utilizando el mismo algoritmo, adaptado a la API.