Tecnológico de Monterrey

Programación Multinucleo

Proyecto Final - Búsqueda A* en CUDA

Profesor: Octavio Navarro

Edgar Garcia - A01021730

Seung Hoon Lee - A01021720

Gualberto Casas - A00942270

26 de Noviembre, 2018

Descripción del del proyecto y compilación

Antes que nada daremos una pequeña introducción A* y distancia Manhattan. Donde en este caso A* es un algoritmo completo: en caso de existir una solución, siempre dará con ella. Donde sí para todo nodo n del grafo se cumple g(n) = 0, nos encontramos ante una búsqueda voraz. Si para todo nodo n del grafo se cumple h(n) = 0, A* pasa a ser una búsqueda de coste uniforme no informada.

Para garantizar la optimización del algoritmo, la función h(n) debe ser una heurística admisible, esto es, que no sobreestime el costo real de alcanzar el nodo objetivo. De aquí es donde viene nuestra segunda parte que es la heurística en este caso ocuparemos la heurística Manhattan donde la suma de las distancias desde la posición actual de cada ficha hasta su posición original. En este documento abordaremos 3 aspectos principales los cuales son: una pequeña introducción del problema, tiempos de ejecución con sus speedups y una pequeña conclusión.

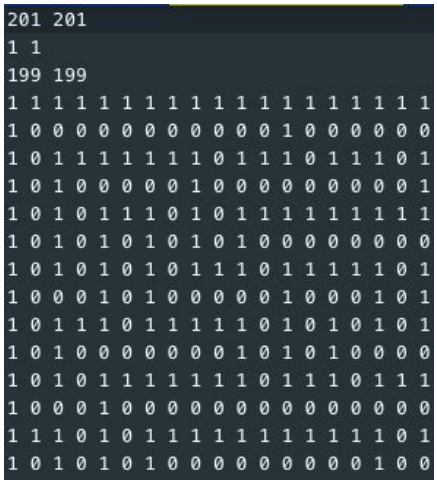
Nuestro proyecto consiste en el algoritmo A* para encontrar el camino más óptimo de un punto A a un punto B dentro de un laberinto. La razón por la cual paralelizar el algoritmo mejoraría su rendimiento, es porque podríamos calcular los diferentes caminos posibles simultáneamente en vez de calcularlo una sólo a la vez. Al estar calculando diferentes caminos al mismo tiempo, es posible reducir el tiempo de ejecución del algoritmo ya que no es necesario estar calculando la heurística en cada expansión de los nodos.

La heurística que planeamos utilizar es la distancia lineal entre el estado actual y el estado final. En otras palabras, la distancia Manhattan. Lo que queremos hacer es optimizar los cálculos para que tome aún menos tiempo.

El objetivo principal del proyecto es la creación de una matriz nueva en la que se tienen las heurísticas de todos los nodos que existirían dentro de nuestro laberinto. De esta manera es posible ahorrarnos el tiempo de cálculo de la heurística de cada nodo y solo buscarla con memoization. Con CUDA, es posible acelerar este proceso ya que cada hilo o bloque puede calcular una heurística paralelamente. Lo cual significa que podríamos calcular 4000 heurísticas en cuestión de milisegundos.

Otra manera de resolver o acelerar el algoritmo de A* con CUDA, es utilizar programación en paralelo para hacer el cálculo de los posibles vecinos de cada nodo. De esta manera, se están haciendo los cálculos de los vecinos al mismo tiempo en vez de tener que calcularlos uno por uno. Sin embargo, este método puede llegar a ser muy ineficiente ya que hay miles de hilos y bloques que se pueden llegar a utilizar pero solo estaríamos abriendo 1-3 de éstos.

A continuación se mostrará la visualización del algoritmo, donde en este caso lo que haremos es pasarle un archivo de texto que contendrá 4 puntos principales: dimensión del laberinto, punto de inicio, punto de llegada y la matriz del laberinto (constituido de 0 y 1).



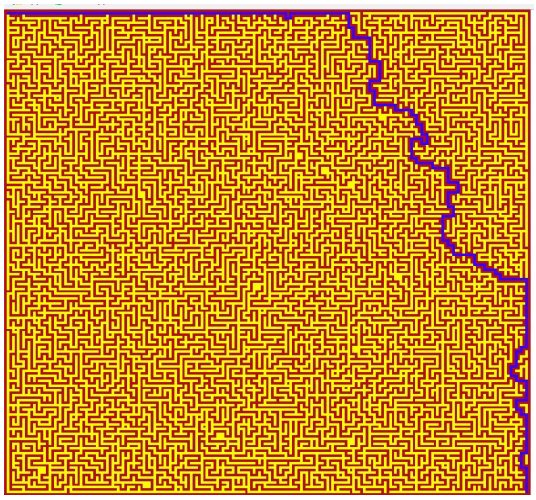
Img. 1: Lectura del archivo donde se leen los parámetros dichos anteriormente

Instrucciones de Ejecución

Para ejecutar el programa completo es necesario encontrarse en la carpeta de /react y correr el siguiente comando:

- yarn generate
- yarn start

Resultados:



Explicación:

El comando de yarn generate correrá el codigo de generator.py (el cual genera un laberinto con el algoritmo de Prim) de N x M tamaño. Si se desea cambiar el tamaño es necesario irse a este archivo y cambiar los parámetros del mazeGenerator. Después de correr generator.py, se correrá el comando para correr el solver en cpp utilizando el maze que se puso en la carpeta de /react/public y ahí mismo se tendrá la solución también. Para finalizar, el yarn start arrancará el servidor para poder ver en React cómo es que se ve el laberinto. (Imagen del resultado)

Desarrollo: Tiempos y Speedups

En esta sección correremos los programas aproximadamente 10 veces, donde compararemos los tiempos en tanto en GPU o CPU. Para ello compraremos y sacaremos unas gráficas de los tiempos que tardaron. A continuación mostraremos los comandos para correr los siguientes programas:

CPU:

```
A01021730@alien1-lab:~/.../react$ g++ -o gualSolver gualSolver.cpp -std=c++11
gualSolver.cpp: In function 'int main(int, char**)':
gualSolver.cpp:194:23: warning: ISO C++ forbids converting a string constant to 'char*' [-Wwrite-strings]
     char * mazeText = "public/python-generated-maze.txt";
                       ^
```

Tiempo de ejecución

```
A01021730@alien1-lab:~/.../react$ ./gualSolver
solution found
End of Search
Time for Astar Search: 1697.82
```

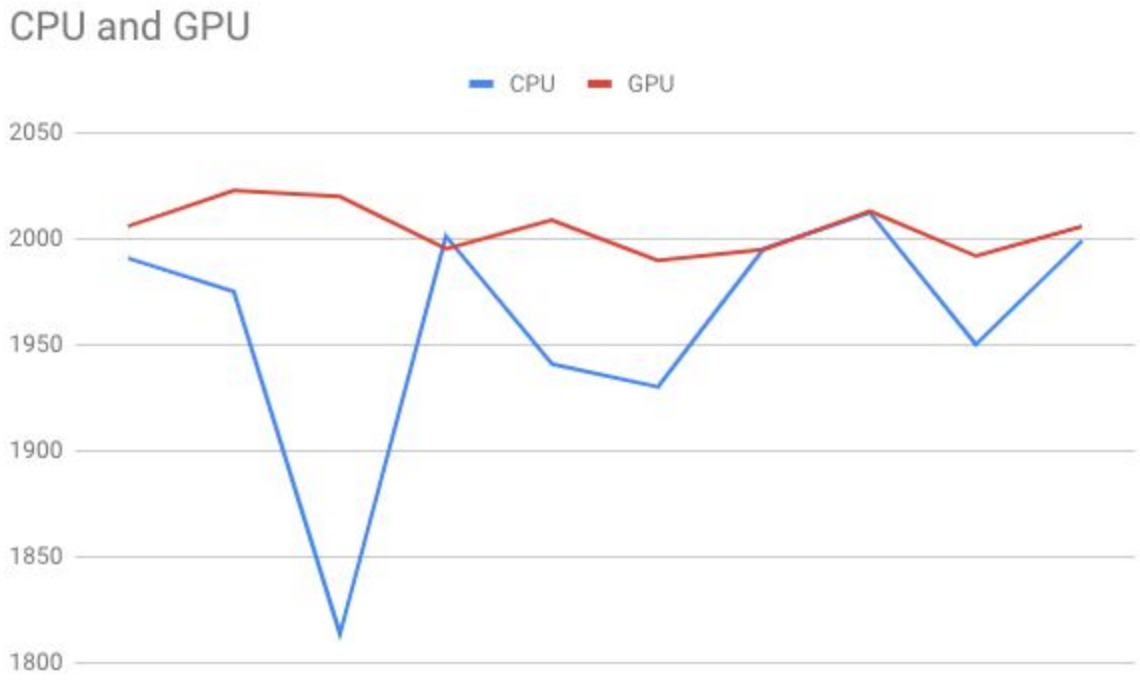
GPU:

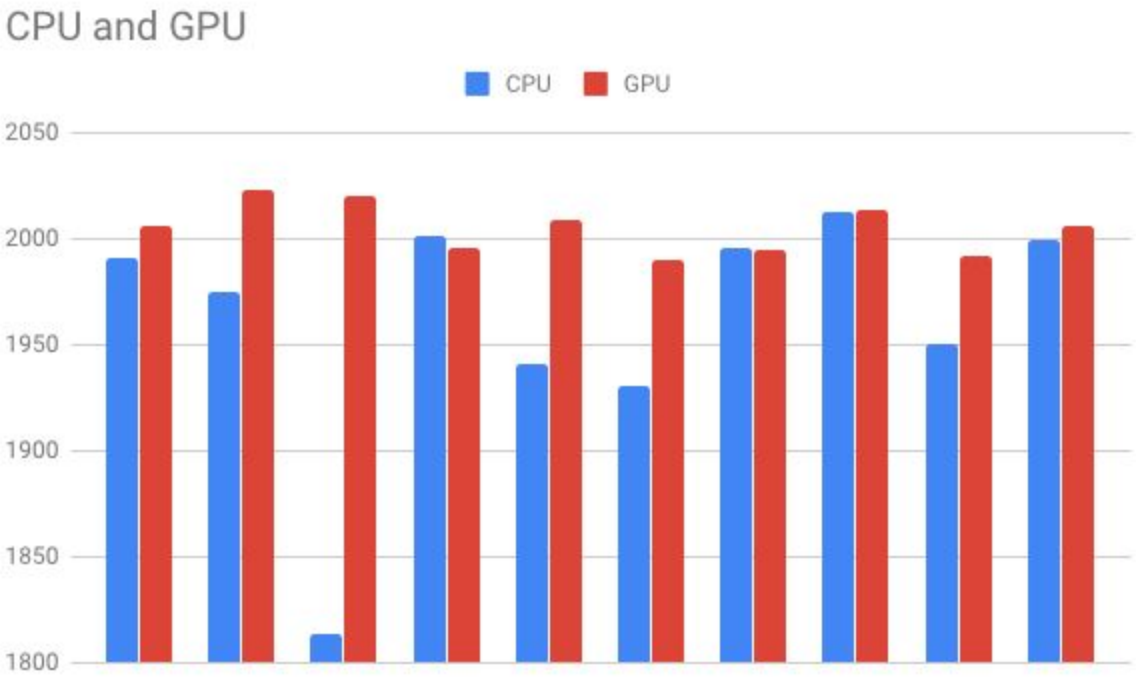
```
A01021730@alien1-lab:~/.../react$ nvcc -o aStar aStar.cu -std=c++11
nvcc warning : The 'compute_20', 'sm_20', and 'sm_21' architectures are d
aStar.cu(184): warning: variable "foundSolution" was set but never used
```

Tiempo de ejecución

```
A01021730@alien1-lab:~/.../react$ ./aStar
Using Device 0: GeForce GTX 670
solution found
End of Search
Time for Astar Search: 1990.71
```

	CPU	GPU
Tiempo(ms)	1990.71	2005.87
Tiempo(ms)	1975.01	2022.67
Tiempo(ms)	1813.56	2019.81
Tiempo(ms)	2001.14	1995.14
Tiempo(ms)	1940.90	2008.71
Tiempo(ms)	1930.10	1989.69
Tiempo(ms)	1995.19	1994.89
Tiempo(ms)	2012.16	2013.04
Tiempo(ms)	1950.18	1991.8
Tiempo(ms)	1999.01	2005.69
Promedio	1,960.796	2,004.731


CPU and GPU
CPU
GPU
2050
2000
1950
1900
1850
1800


CPU and GPU
CPU
GPU
2050
2000
1950
1900
1850
1800

Conclusión

Es evidente que hubo un error con los tiempos de speedup y la solución que utilizamos para tratar de acelerar el algoritmo de A* no funcionó. El problema principal es que los tiempos de ejecución en la GPU deberían de ser infinitamente más rápidas ya que estamos haciendo uso de los varios hilos y bloques que existen dentro de la GPU para poder paralelizar el algoritmo de A*. Sin embargo, nuestra solución para utilizar la GPU con CUDA para calcular paralelamente las heurísticas de todos los nodos y solo tener que leerlas desde una matriz no fue la solución óptima y tampoco brindó el resultado que esperábamos.