Alan Salinas González

A00806101

Reporte Proyecto Final (Buscador ciego secuencial local)

Programación Concurrente y paralela

Introducción.

Para este proyecto, en materia de programación paralela, se utiliza la arquitectura de GPU’s de Nvidia con la tecnología Cuda para procesamiento de alto rendimiento. El propósito practico del proyecto es emplear la capacidad de cómputo que provee esta tecnología para evaluar la misma función, la cual llamaremos función objetivo, con múltiples diferentes valores de entrada que corresponden un espacio de búsqueda.

El objetivo es buscar dentro de ese espacio, valores de entrada que produzcan resultados óptimos en la evaluación de la función objetivo (ej. Mínimos y máximos). En el caso particular de este proyecto se busca maximizar la transferencia de potencia en un circuito simple (minimizando la diferencia entre la resistencia equivalente y la resistencia de carga), y tratar de que el valor del voltaje de carga sea lo más cercano al deseado provisto por el usuario. Esto considerando que solo pueden usarse resistencias con valores comerciales.

Llamaremos al sistema evaluador de la función objetivo y que regresa los valores óptimos, un buscador ciego. Lo llamamos ciego porque puede no conocer el comportamiento de la función objetivo, aun así busca evaluando ciegamente los puntos del espacio.

El código de la solución se provee adjunto al texto como un proyecto de Visual Studio Express 2013, es necesario notar que para compilar el proyecto debe usarse la plataforma (32, 64bit) acorde a la versión del driver para Cuda de Nvidia(32 o 64bit). La versión más reciente se mantiene en GitHub: <http://github.com/alansalinas/BuscadorSecuencial>

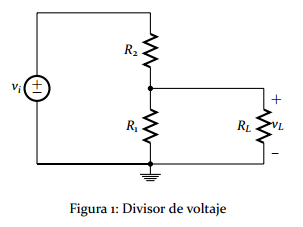
Implementación

A continuación la descripción del circuito a resolver, el buscador intentara encontrar los valores de R1 y R2 en valores comerciales, de forma que el voltaje en la carga sea Vd (VL ~= Vd), y que la transferencia de potencia sea máxima,

Req(R1, R2) ~= RL

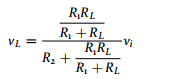
**Circuito**

Considérese el voltaje deseado sobre la carga como Vd.



**Modelo matemático del circuito**

Voltaje en la carga RL



Resistencia equivalente



Buscamos reducir las diferencias entre Req y RL para maximizar la transferencia de potencia, también se busca reducir la diferencia entre el voltaje real en la carga y el voltaje deseado en ella, para esto reducimos los errores cuadráticos buscando los valores que minimicen la siguiente función objetivo.



En donde le asignamos una importancia o peso de 0.2 al error de la resistencia equivalente y un peso de 1 al error en el voltaje en la carga.

**CUDA**

La rutina lanzadora **evaluarCuda(float \*c, float vi, int rl, float vd, size\_t size)** recibe los parámetros de valores del circuito y el vector para resultados así como el tamaño del mismo, esta se encarga de inicializar el kernel decidiendo si la carga es suficiente para un bloque de 512 *threads* y un solo *grid*, o de otra forma generando tantos *grids* de bloques de 512 *threads* en la dimensión en “*x*” para cubrir el espacio de búsqueda requerido por el parámetro “**size**”. Se encarga también de reservar el espacio de memoria necesario en el GPU suficiente para acomodar al vector de resultados. La misma rutina los copiara de regreso hacia la memoria del CPU cuando su ejecución haya terminado. Esta rutina lanzara el kernel en el primer GPU Cuda encontrado en el sistema, es decir utilizara el *Device 0* del listado de dispositivos de Cuda.

El kernel **objetivo** es el programa que cada núcleo Cuda ejecutara, los parámetros que recibe el kernel es un vector de flotantes para resultados, recibe además los valores para **Vi**, **Vd** y **RL**. Cada nodo Cuda reconocerá que punto del espacio de búsqueda tiene que evaluar mediante su coordenada en bloque y *grid* en *“x”.* Los valores en 3 bandas de **R1** y **R2** estarán codificados como 3 dígitos en ese orden correspondiente para cada resistencia, esto quiere decir que las dos resistencias ocuparan 6 dígitos y se obtendrán con el número que provee el id en la coordenada “*x*”.

**R1 = idx / 1000**

**R2 = idx % 1000**

De esta forma separamos los valores de cada resistencia en dos variables. El espacio de búsqueda será entonces, codificado como (R1:R2):

000:000 hasta 999:999

La función objetivo para minimizar los errores cuadráticos es evaluada entonces en el punto del espacio correspondiente a cada núcleo, obtenido mediante su coordenada de *thread* y el resultado flotante se escribe en la coordenada correspondiente (que también representa **R1:R2**), de esta forma el buscador puede reconocer que valores de resistencias pertenecen a cada evaluación, puesto que el índice del vector es también la concatenación de dígitos **R1:R2**. Existe un poco de perdida de precisión por utilizar flotantes en vez de dobles, esto se pretendía con el fin de hacer más rápidas las operaciones de punto flotante pero en este punto parece ser despreciable la diferencia en tiempo comparado con la ganancia que se tendría al utilizar precisión doble.

**Buscador.c**

El programa principal recibe la entrada del usuario para escoger los valores de **Vi** (float), **Vd** (float) y **RL**(int).

Se encarga de llamar a la función lanzadora **evaluaObjetivo** con los estos valores y el vector de resultados, también se reserva la memoria para estos en la computadora. Una vez evaluados los objetivos, el procesador se encarga de reportar el valor mínimo y máximo encontrado de evaluación y cuáles fueron los valores de resistencias **R1** y **R2** que los generaron. En este caso el que nos interesara será el mínimo.

Nótese en este problema la reducción del espacio por el simple conocimiento de que la búsqueda se hará solo en valores comerciales de resistencias, lo cual ingeniosamente se modela en 3 bandas con posibles valores entre 0 y 9, esto quiere decir que por cada resistencia habrá 1000 posibles combinaciones de bandas, y en dos resistencias tendremos 1,000,000 de posibles combinaciones como espacio de búsqueda. Este espacio es tan pequeño que la persona que utiliza el buscador puede darse el lujo de recorrerlo todo, asegurando encontrar el óptimo, lo que convierte a nuestro buscador en un buscador secuencial y local, siendo su localidad todo el espacio posible.

Conclusiones

Aun cuando la tecnología de multiprocesamiento se vuelve una herramienta poderosa para los buscadores, estos se vuelven inútiles si no se tiene un modelo suficientemente aproximado del problema, de modo que se conozca el espacio de búsqueda. Es necesario notar que entre más conocimiento se tenga del modelo o del espacio de búsqueda, menor es el trabajo que tendrá que realizar el buscador.

Este problema es sencillo y el espacio de búsqueda es tan pequeño que le resulta muy fácil al buscador aun iterando por fuerza bruta secuencialmente, evaluando cada punto posible de todo el espacio. Sin embargo existen problemas de mayor cantidad de complejidad y mayor número de entradas y salidas a optimizar dentro de la función objetivo. La ventaja es que estos buscadores parecen no topar con limitaciones de concepto, complejidades lógicas o matemáticas. Si no que simplemente el espacio de búsqueda se vuelve más grande o más lento de evaluar, pero aun así es posible trabajarse y encontrar hasta tal vez cierto grado, un óptimo. El buscador no es afectado por la complejidad o no linealidad del proceso, sino que es afectado por el tamaño del espacio en donde buscar.

Esto se traduce en que en un problema mayor el buscador secuencial empezara a presentar dificultades o le tomara tiempos demasiado encontrar un óptimo decente. Para resolver esto se emplearan algoritmos de búsqueda ciegos, que nos dan una cierta idea o guía de hacia donde debemos buscar, los algoritmos intentaran predecir que evaluaciones serán optimas y las usaran para generar nuevas evaluaciones, ejemplos de estos algoritmos de optimización ciega son los algoritmos genéticos o el recocido simulado, de este modo podemos volver esta búsqueda más eficiente en espacios de mayor tamaño y dimensiones. La implementación de estos optimizadores podría beneficiarse de la rapidez que provee el cómputo paralelo en arquitecturas de GPUs.

Bibliografía

Se toma como ejemplo el ejercicio del profesor Manuel Valenzuela para implementación de un recocido simulado, cuya descripción puede encontrarse en su sitio. Para la implementación de este proyecto se utiliza un formato de resistencias de tres bandas.

[Link directo al pdf del ejercicio.](https://b0aeedd8-a-62cb3a1a-s-sites.googlegroups.com/site/tc3023/proyectos/divisor_voltaje.pdf?attachauth=ANoY7cpAIKdLbM8Xn99wEblRXSKGv-KC4fqw8yRHtf-bkmU7Y7RoYzlUGaPrPlbcmg3e8T58ROHZ_JoyXmSd5Ngeq-tiAoJ6KtkYutDKSyUkt8F73c5UPYg_9bxEB_XzOpbbHJlreGny1PrycBSL_wrAfLkT9EPEWBHRBQzTnsRnKHZ4IVhwfHsJci2XeRdOPgab4d9-RESsV0euitm9iCwnev54IHao_ABJ1qikMpUuBbE5UnTrsTU%3D&attredirects=0)

Página del curso:

https://sites.google.com/site/tc3023/proyectos