Proyecto final: Algoritmo genetico en CUDA

Eduardo Tapia
Maestria en Ciencias de la Computación
Cimat
Guanajuato, Mexico
eduardo.tapia@cimat.mx

I. Introducción

II. Algoritmos Geneticos

Los algoritmos geneticos son algoritmos de optimización estocástica que están inspirados en los procesos genéticos poblacionales propuestos en la teoría de selección natural de Darwin, tomando principalmente los conceptos de reproducción, recombinación, mutación y selección.

El funcionamiento general de estos metodos se puede observar en el algoritmo 1

Algorithm 1 Algoritmo Genético

```
Generar población de manera aleatoria
for gen = 0; gen < max_gen; gen++ do
    Evaluar la población
    Selección de padres
    Recombinación
    Sustitución
    best = Seleccion del mejor de la poblacion
    if (condicion de termino) then
        break
    end if
end for
regresar best
```

Es importante mencionar que para el esquema particular de los algoritmos genéticos, existen distintas formas de realizar las distintas etapas, de manera que para cada implementación pueden existir diversas posibles combinaciones comenzando con la posibilidad de resolver problemas de variables enteras o variables reales.

III. Computo Paralelo en GPU y CUDA

El computo paralelo en GPU se remonta a principios de los 80, sin embargo no tuvo impacto hasta el año 2001 en el que NVIDIA implementó shaders programables para su arquitectura GeForce3, lo que permitió comenzar a hacer ciertas operaciones matriciales y vectoriales dad su naturaleza paralela y la similitud a las operaciones necesarias para realizar calculos de gráficos, de manera que se podía "Engaãr" al gpu para que realizara las operaciones.

Este esquema de programación en GPU se mantuvo hasta el año 2006 en el que Nvidia diseñó el primer GPU que si bien fue diseñado para gráficos, su arquitectura fue desarrollada con el objetivo de permitir que los programadores pudieran aprovechar el gpu para computo a través del lenguaje CUDA que fue presentado como una extensión de C++.

Esto ultimo dio pie a que el computo en gpu de propósito general GPGPU[1] despegara.

En la actualidad, el computo en GPU tiene aplicaciones en diversas áreas, y ha permitido que muchas otras evolucionen rápidamente como es el caso de Deep learning.
Las áreas donde el computo en GPU ha presentado mayor impacto son:

- Industria del petroleo
- Agencias aeroespaciales
- Investigacion y Supercomputo
- Finanzas
- Deep Learning

A. Paralelismo en CUDA

LA API de computo paralelo Compute Unified Device Architecture (CUDA) es una API desarrollada por Nvidia que se utiliza para desarrollar algoritmos de computo paralelo aprovechando la arquitectura de los procesadores de gráficos GPU desarrollados por la misma empresa.

Esta api contiene compiladores, utilidades y librerías que permiten desarrollar aplicaciones en C y C++ en las cuales se interactúa con el GPU, enviando y recibiendo información, así como enviando instrucciones y parámetros para la ejecución de Kernels dentro del gpu.

B. Modelo de programación en CUDA

Las funciones que se compilan para ser ejecutados dentro del GPU se llaman Kernels.

Al ejecutar los kernels, el cpu debe enviar al GPU los parámetro s con los que se debe ejecutar dicho kernel, esto es na malla de bloques, que a su vez contienen una malla de hilos dentro de si, por las relaciones que existen dentro de los kernels, son ideales para modelos de procesamiento de SIMD

[1]Por sus siglas en inglés

C. Arquitectura general de una GPU

La arquitectura de las GPU Nvidia varía de generación en generación, sin embargo manejan conceptos básicos que han sido heredados desde las primeras arquitecturas compatibles con CUDA.

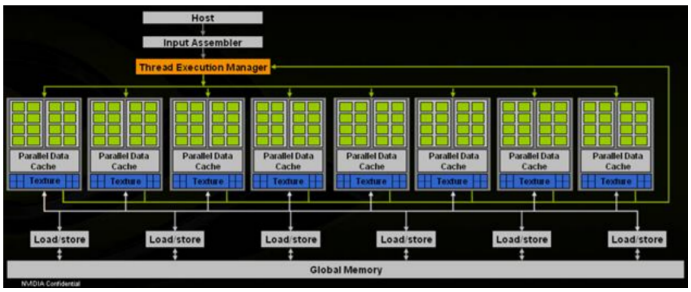

Fig. 1. Arquitectura de GPUs tesla

Como se puede observar en la imagen 1 donde se puede apreciar la arquitectura Tesla[2] de GPU los CUDA núcleos (bloques verdes) se encuentran agrupados en grupos de 32 a los cuales se les llama Stream Multi processors o SMP.

Dentro de cada SMP, los CUDA núcleos o SP funcionan como ALU, mientras que comparten la misma unidad de ejecución, memoria cache y memoria de textura, mientras que todos los SMP pueden acceder a la memoria global del GPU de forma independiente.

Cuando un Kernel es invocado, cada bloque es asignado a un SMP distinto, y los hilos que que envíen por bloque, se distribuirán dentro de los SP contenidos en el SMP, por este motivo se debe tener cuidado, ya que si se envían todas las ejecuciones en un único bloque gigantesco, únicamente se podrá correr al mismo tiempo el numero de SP por SMP que tenga la tarjeta, dejando todos los demás SMP completamente inactivos, lo cual resulta en una perdida de eficiencia muy fuerte, y es por esta razón que es importante tener cuidado tanto en el numero de hilos que se pide como la cantidad de bloques, para aprovechar al máximo las capacidades de paralelismo del GPU.

IV. Metodología

A. Implementacion GA

Pra este trabajo se desarrolló un algoritmo genético que está basado fuertemente en el propuesto por Arora en su articulo [1] en el cual utilizan diversas estrategias de optimización de codigo para GPU:

- Estructura del algoritmo pensada para aprovechar la máxima ejecución paralela e intensidad aritmética.
- Minimizar las transferencias de datos entre el host y device.
- Se busco lograr accesos a memoria global de forma contigua lo que permite que al realizar las lecturas por warps sea mucho mas eficiente.
- Minimizar el uso de variables locales por hilo
- Uso de memoria compartida para minimizar la posibilidad de conflictos de acceso por diversos warps
- Minimizar lo mas posible la divergencia entre warps
- Minimizar la necesidad de sincronizaciones dentro de los bloques.

En la implementación se busca mantener estos puntos sin embargo se realizaron algunas ligeras modificaciones por cuestiones de tiempo y practicad.

B. parámetro s de entrada

Los parámetro s de entrada en los que se basa el programa son:

- pop_size indica el tamaño de la población
- nvar Indica el numero de variables de la función objetivo
- lims Matriz que contiene los limites superior e inferior para cada una de las variables del problema.

C. Organización de los datos

Para maximizar la continuidad de los datos en memoria la población se almacena en un arreglo unidimensional de tamaño pobladoresxvariables, sin embargo se genero una estructura de doble apuntador para poder accesar de forma mas sencilla al contenido del mismo, tal como si fuese una matriz mas convencional, esta estructura se utilizo para almacenar las poblaciones y los limites.

D. Generacion de numeros aleatorios

En el articulo [1] propone utilizar un operador propio para generar números aleatorios y mantener un arreglo de semillas para cada hilo sin embargo este acercamiento parece un poco anticuado ya que se realizo con un GPU que no era capaz de aprovechar la librería CURAND, sin embargo en esta implementación se opto por generar los números aleatorios utilizando dicha librería lo cual permite un uso mas eficiente del GPU y de sus recursos al mismo tiempo que se tiene cierta seguridad respecto a la calidad de los números pseudoaleatorios.

E. Población inicial

Para inicializar la población se genera un conjunto de números aleatorios del tamaño de la poblacionxnvar que se quiere crear y a continuación se utiliza para generar un valor dentro de los rangos delimitados por lims utilizando la funcion:

$$pob[i][j] = lim_{min} + rand(lim_{max} - lim_{min}) \qquad (1)$$

F. Selección

Para seleccionar individuos se utilizo el método del torneo binario, en el cual para cada 2 individuos de la nueva población se seleccionan 4 individuos aleatorios, en pares de los cuales se selecciona el mejor de cada par según el criterio de fitness y se aplica el operador de recombinación para generar los nuevos 2 individuos.

[2]Se eligió la arquitectura Tesla ya que es la mas fácil de describir por el tamaño de sus componentes.

G. Recombinación

El método de recombinación que se implementó es el operador de evolución diferencial ya que al buscar optimizar variables reales, resultaba mas sencillo que programar el SBX, el operador de corssover presentado por Tsoulos en [2]

$$hijo_1[j] = padre_1[j]\alpha + (1-\alpha)padre_2[j] \tag{2}$$
$$hijo_2[j] = padre_2[j]\alpha + (1-\alpha)padre_1[j] \tag{3}$$
$$\tag{4}$$

H. Mutación

Para el proceso de mutación el operador a utilizar es el de mutación polinomial la cual se desarrolló para problemas de variable real con GA. Este método depende de un parámetro η_m el cual induce un efecto de perturbación sobre el poblador de l orden de $(b-a)/\eta_m$ donde a y b los valores mínimo y maximo respectivamente para la variable sobre la que se va a realizar la perturbación.
El operador consiste en utilizar una distribución de probabilidad polinomial la cual se ajusta en ambas direcciones de manera tal que la perturbación no salga de los rangos de las variables. El operador funciona de la siguiente forma, para un padre dado $p \in [a,b]$, el padre mutado p' para una variable particular se crea a partir de un numero $u \in \mathcal{U}(0,1)$

$$p_i' = \begin{cases} p_i + \overline{\delta_L}(p_i - x_i^{(L)}), for & u \leq 0.5 \\ p_i + \overline{\delta_R}(x_i^{(U)} - p_i), for & u > 0.5 \end{cases} \tag{5}$$

Donde los parámetro s $\overline{\delta}$ se calculan como:

$$\overline{\delta_L} = (2u)^{\frac{1}{1+\eta_m}}, \quad for \quad u \leq 0.5, \tag{6}$$
$$\overline{\delta_R} = 1-(2(1-u))^{\frac{1}{1+\eta_m}}, \quad for \quad u > 0.5. \tag{7}$$
$$\tag{8}$$

En la figura 2 se puede observar la curva de densidad de probabilidad para un elemento mutado.

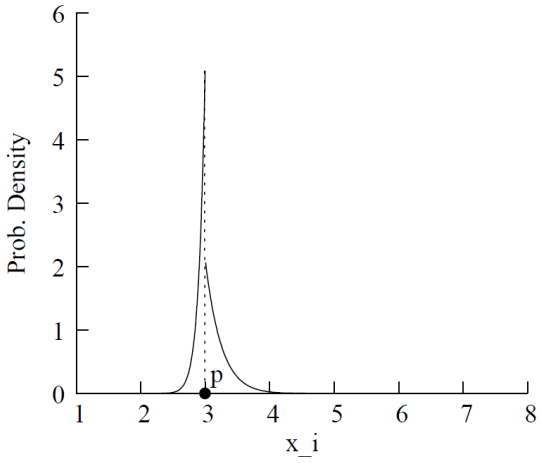

Fig. 2. Densidad de probabilidad para la mutación con el operador polinomial. de [3]

I. Funciones a optimizar

Se eligieron 2 funciones a optimizar para las pruebas, la función Rosenbrock (9) y la función esfera (10).

$$f(x) = \sum_{i=0}^{n-1} \left[100 * (x_i^2 - x_{i+1}) - (x_i - 1)^2\right] \tag{9}$$

$$f(x) = \sum_{i=0}^{n} x_i^2 \tag{10}$$

V. Resultados

Para la función rosenbrock, se encontró que el algoritmo pese a la implementación sencilla que se genero, logra optimizar la función en la mayoría de los casos, se realizaron pruebas con 10 ejecuciones, para los cuales se promediaron los maximos y mínimos, tanto en cpu figura 4 como en gpu figura 3 y se obtuvo lo siguiente:

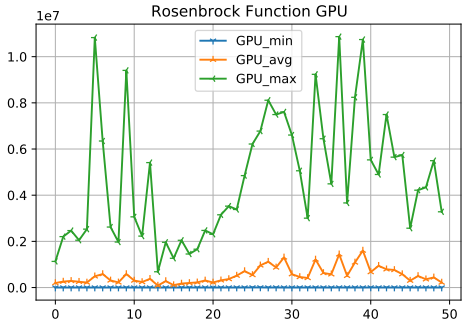

Fig. 3. Valores maximos y minimos encontrados en las ejecuciones para GPU

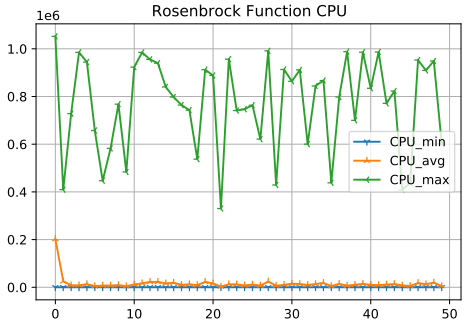

Fig. 4. Valores maximos y minimos encontrados en las ejecuciones para cpu

Asimismo se gráfico la convergencia de ambos programas con la idea de validar que estaba logrando el optimizar la función, figura 5

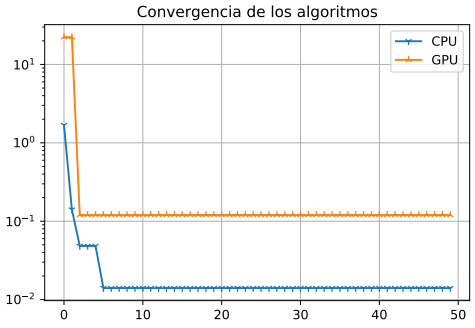

Fig. 5. Convergencias de las implementaciones, gpu y cpu

La función esfera se utilizo como parámetro de confirmación de que el algoritmo estaba realizando la tarea de forma correcta, ya que es mucho mas sencilla de optimizar que la función rosenbrok, la curva de convergencia obtenida para la Ejecución en GPU es la siguiente:

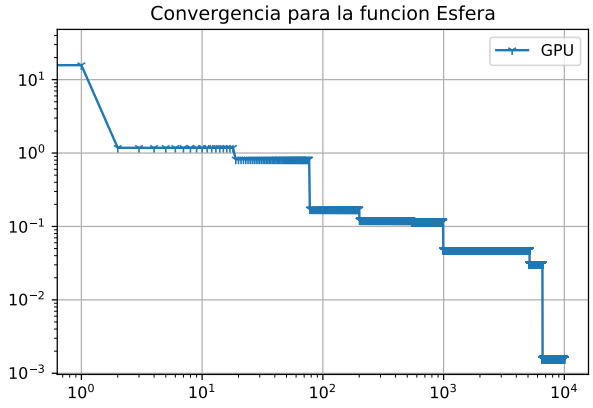

Fig. 6. Convergencia función esfera GPU

El speedup obtenido de diversas configuraciones de individuos y numero de variables se puede observaren la figura 6

Tiempos de ejecucion y Speedup

	GPU			CPU			Speedup		
	2	8	16	2	8	16	2	8	16
128	6.56E-03	6.71E-03	6.89E-03	7.00E-05	8.12E-04	1.17E-03	0.01	0.12	0.17
256	7.32E-03	7.52E-03	7.77E-03	1.17E-03	1.68E-03	2.39E-03	0.16	0.22	0.31
512	1.04E-02	9.10E-03	9.67E-03	2.62E-03	3.50E-03	4.89E-03	0.25	0.38	0.51
1024	1.21E-02	1.24E-02	1.41E-02	5.08E-03	7.10E-03	1.00E-02	0.42	0.57	0.71
2048	2.00E-02	2.07E-02	1.93E-02	1.08E-02	1.46E-02	2.10E-02	0.54	0.71	1.09
4096	3.13E-02	3.19E-02	3.30E-02	2.27E-02	3.14E-02	4.32E-02	0.73	0.98	1.31
8192	5.62E-02	5.72E-02	6.09E-02	4.91E-02	6.48E-02	9.03E-02	0.87	1.13	1.48

Fig. 7. Speedup entre ejecuciones, tiempos en segundos

VI. Discusion y recomendaciones

Si bien se logro programar un algoritmo que funciona para optimizar las funciones de prueba, el speedup que se obtuvo dista mucho de lo esperado, eso puwde estar asociado a que la implementación no aproveche de la mejor manera la arquitectura del gpu, por esta razón se considera que es posible mejorar el rendimiento si se realizan estas consideraciones, así como buscar la forma de utilizar de forma mas eficiente la memoria compartida dentro de los SM.

En las gráficas se puede observar también que la calidad de los resultados del algoritmo no es muy buena, esto puede estar asociado a que el algoritmo implementado es muy sencillo, comenzando por que tiene muy poco elitismo y la calidad del operador de crossover, que idealmente debió ser el SBX.

References

[1] R. Arora, R. Tulshyan, and K. Deb, "Parallelization of binary and real-coded genetic algorithms on gpu using cuda," in *IEEE Congress on Evolutionary Computation*, pp. 1–8, IEEE, 2010.

[2] I. G. Tsoulos, "Modifications of real code genetic algorithm for global optimization," *Applied Mathematics and Computation*, vol. 203, no. 2, pp. 598–607, 2008.

[3] K. Deb and D. Deb, "Analysing mutation schemes for real-parameter genetic algorithms," *International Journal of Artificial Intelligence and Soft Computing*, vol. 4, no. 1, pp. 1–28, 2014.

[4] J. R. Cheng and M. Gen, "Accelerating genetic algorithms with gpu computing: A selective overview," *Computers & Industrial Engineering*, vol. 128, pp. 514–525, 2019.