REPORTE DEL ARTÍCULO: “Singular Value Decomposition on GPU using CUDA”

AUTORES: Sheetal Lahabar y P.J.Narayanan

REPORTE REALIZADO POR: Miguel Francisco de Lascurain Morhan (6068)

Resumen del Artículo

En el artículo escrito por los dos investigadores mencionados, se presenta la implementación de un algoritmo para resolver el problema de la Descomposición en Valores Singulares (SVD, por sus siglas en inglés) de una matriz densa de números reales, usando el modelo de programación CUDA. Es la primera vez que se reporta la solución de este problema en una GPU. El método utilizado para encontrar la SVD se resuelve en dos pasos: bidiagonalización seguida de diagonalización. Los resultados obtenidos son mejores que los reportados usando otros paquetes de computación.

El artículo tiene el siguiente capitulado:

1. Introducción.
2. Trabajos relacionados.
3. Algoritmo SVD.

3.1 Bidiagonalización.

3.2 Diagonalización de una matriz bidiagonal.

3.3 SVD completa.

1. Resultados.
2. Conclusiones.

1. Introducción

Se destaca la importancia de la SVD en problemas de Álgebra Lineal, como el cálculo de la pseudo inversa de una matriz, la solución de sistemas de ecuaciones lineales homogéneas y el problema de mínimos cuadrados, entre otros. Los cálculos matriciales de la SVD son robustos ante errores numéricos, por lo que se utiliza ampliamente en componentes principales, procesamiento de señales e imágenes y reconocimiento de patrones.

Se define como SVD de una matriz A, de tamaño m x n, a cualquier factorización de la forma:



En la ecuación anterior, U es una matriz ortogonal de tamaño (m x m), V es una matriz ortogonal de tamaño (n x n) y Σ es una matriz diagonal de tamaño (m x n) con elementos sii >= 0 en orden descendente a lo largo de la diagonal y todos los demás elementos iguales que cero.

Más adelante, se da una breve explicación del avance logrado por NVIDIA con sus equipos que cuentan con GPUs y el desarrollo de CUDA y CUBLAS. No se elaborará en este tema, pues ya se ha visto en el curso.

2. Trabajos Relacionados

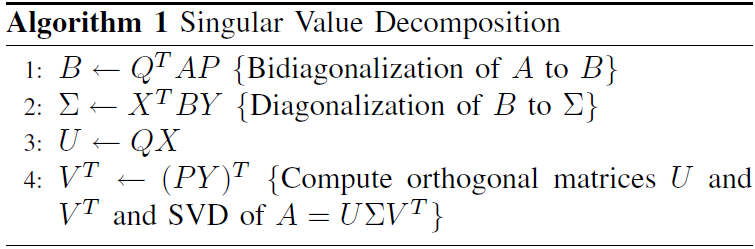
Se mencionan otros desarrollos que han trabajado en GPUs, entre ellos, computación matemática y geométrica, multiplicación de matrices, multiplicación de matrices y vectores, Transformada Rápida de Fourier (FFT), algoritmos de redes, y otros. Mencionan desarrollos enfocados a la paralelización del algoritmo SVD para matrices densas y casi vacías (sparse matrices), usando métodos diferentes al que ellos proponen. Esta información normalmente se añade a los artículos a petición expresa de los réferis. Es una lata, pues hay que revisar la literatura.

3. Algoritmo SVD

Los autores mencionan que la obtención de la SVD de una matriz se puede realizar usando el algoritmo de Golub-Reinsch (Bidiagonalización y Diagonalización) o el método de Hestenes.

Escogen el primero, pues afirman que es simple y compacto y puede mapearse a la arquitectura SIMD de un GPU. El algoritmo forma parte del paquete LAPACK (Linear Algebra Package). En la primera fase del algoritmo, la matriz es reducida a una matriz bidiagonal usando una serie de transformaciones de Householder. En la segunda fase, la matriz bidiagonal es diagonalizada usando iteraciones QR implícitamente desplazadas. Este algoritmo tiene una complejidad *O*(mn2) para m >= n.

En términos amplios, el algoritmo SVD se puede visualizar como:



3.1 Bidiagonalización

3.1.1 Algoritmo

En este paso la matriz A es descompuesta como:

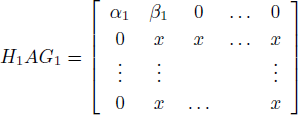


mediante la aplicación de transformaciones de Householder; B es una matriz bidiagonal y Q y P son matrices Householder unitarias.

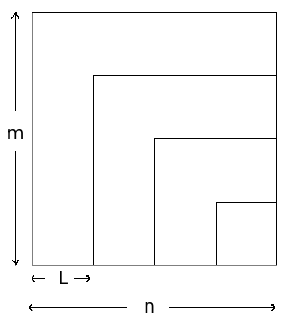
Lo que sigue en el artículo es una descripción pormenorizada del algoritmo, en la cual no me detendré. En términos muy amplios, se obtiene la matriz bidiagonal superior B:



mediante la eliminación iterativa de columnas debajo de la diagonal principal de la matriz A y de las filas a un lado de la super diagonal de esta matriz.

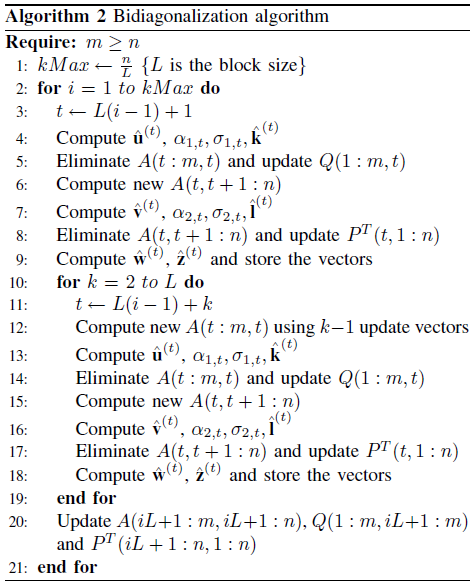


De forma similar, se obtienen las matrices Q y P. Es importante mencionar que se requiere de la actualización continua de matrices, lo cual utiliza muchas lecturas y escrituras en la memoria, que son caras computacionalmente. Para minimizar los accesos, se usa una técnica de división de la matriz A en bloques de tamaño L, y la actualización se hace después de que L filas y columnas se han bidiagonalizado.



3.1.2 Bidiagonalización en el GPU

Se aprovechan las funciones de CUDA BLAS de multiplicación de matriz-vector, matriz-matriz y de obtención de normas. Se menciona que CUBLAS es más eficiente cuando las matrices tienen dimensiones que son múltiplos de 32. También se discute el tema de cuál es la mejor estrategia para pasar información entre el CPU y el GPU. La bidiagonalización de la matriz A se hace en la GPU. La memoria total utilizada en el GPU en esta primera fase es del orden de (3(*mL* + *Ln*) + *m*2 + *n*2 + *mn* + 2 max(*m, n*)) *×* 4 bytes. El algoritmo de bidiagonalización es el siguiente:



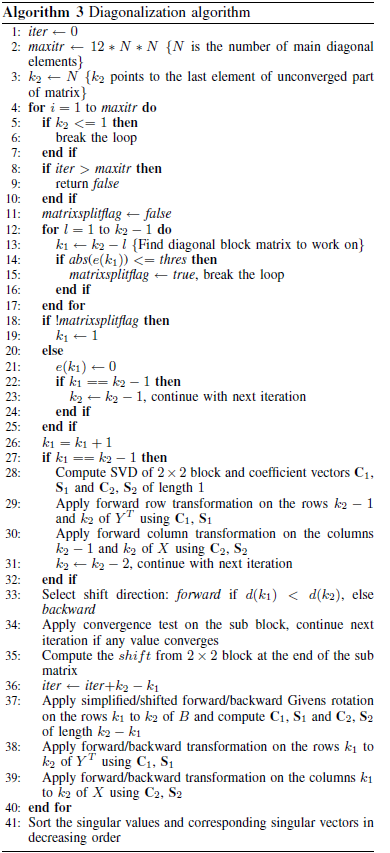
3.2 Diagonalización de la matriz bidiagonal

3.2.1 Algoritmo

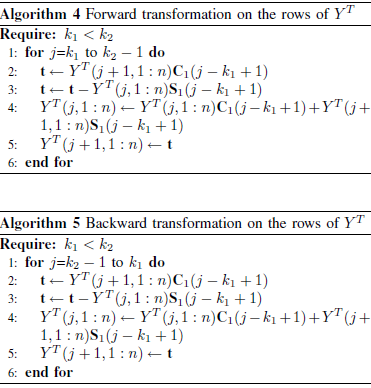
Como se mencionó, la matriz bidiagonal es diagonalizada usando iteraciones QR implícitamente desplazadas. La matriz B obtenida en la fase anterior se descompone en:



En esta ecuación, Σ es una matriz diagonal, y X y Y son matrices unitarias ortogonales. El algoritmo empleado es el siguiente:



En el algoritmo, las *d*(*i*)s son los elementos de la diagonal de B y las *e*(*i*)sson los elementos de la superdiagonal de la misma matriz. En cada iteración, se actualizan estos elementos de tal forma que los valores de la superdiagonal se reducen. Al lograr la convergencia, las *d*(*i*)s contienen los valores singulares y X y YT son los vectores singulares de B. También se presentan los algoritmos para transformar YT y X.



3.2.2 Diagonalización en la GPU

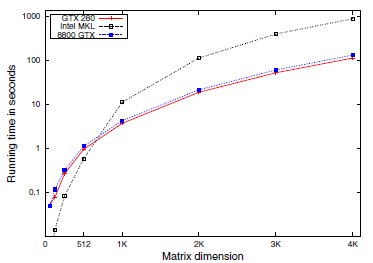
En esta sección se explica cómo se efectúan los cálculos de los tres algoritmos anteriores en la NVIDIA. El algoritmo 3 se lleva a cabo en el CPU y partes de los algoritmos 4 y 5 se paralelizan en la GPU. Los detalles –demasiado complejos para el alcance de este reporte– se pueden encontrar en el artículo. La memoria utilizada en esta fase es de *(6 min(m, n))* x 4 bytes en el CPU y de (m2 + n2) x4 bytes en el GPU.

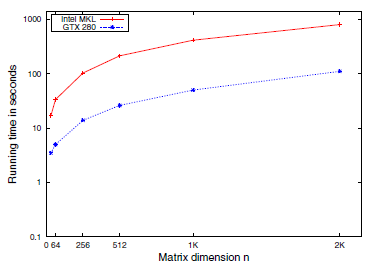
3.3 SVD completo

Para finalizar, usando CUBLAS, se efectúan dos multiplicaciones matriz-matriz para obtener las matrices ortogonales U = QX y VT = (PY)T. Los elementos *d*(*i*)s son los valores singulares, esto es, la diagonal de Σ.

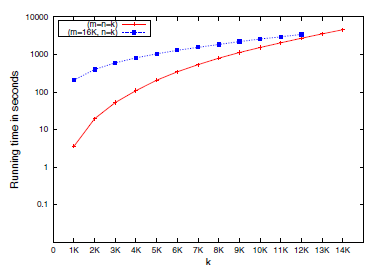
4. Resultados

En esta sección se comparan los resultados de los autores de este artículo frente a la implementación optimizada en CPU del SVD en MATLAB y la versión LAPACK en Intel MKL 10.0.4. Su algoritmo lo prueban en diferentes equipos. Para efectuar las pruebas, generan aleatoriamente 10 matrices densas de diferentes tamaños. Cada corrida de su algoritmo es replicada 10 veces. En promedio, su algoritmo resultó ser entre 3.04 y 8.2 veces más rápido que la Intel MKL y entre 3.32 y 59.3 veces que MATLAB. Es interesante mencionar que para matrices relativamente pequeñas un algoritmo secuencial es más rápido que uno en paralelo. Este mismo efecto lo hemos observado en nuestra investigación sobre el SALBP-2. Agrego las gráficas siguientes que se explican solas.





También efectúan experimentos con matrices muy grandes, del orden de 14K, para diferentes tamaños de m y de n. La figura siguiente muestra los resultados.



5. Conclusiones

No me cabe ninguna duda que la paralelización produce resultados fenomenales. En este artículo se demuestra fehacientemente este hecho utilizando un problema clásico de Álgebra Lineal y los resultados son espectaculares.

Hay mucho por hacer en este campo.