Daniel Sharp 138176

Singular Value Decomposition on GPU using CUDA

Sheetal Lahabar y P. J. Narayanan

Los autores comienzan por introducir la técnica de descomposición de matrices en valores singulares, definida de la siguiente forma:

$$A = U\Sigma V^T$$

donde A es cualquier matriz m x n, U es una matriz m x m ortogonal, V es una matriz ortogonal y sigma es una matriz de m x n diagonal, con elementos de la diagonal en orden descendiente.

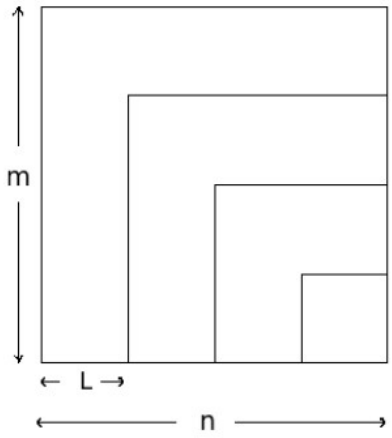
El objetivo el proyecto en el que se basa el paper es la implemetación del algorimo de SVD en CUDA para su ejecución en paralelo en GPU. Los autores afirman que son los primeros en implemetar este método en GPU utilizando la secuencia de bidiagonalización seguida de diagonalización. Además, explican que las GPU se utilizan cada vez más para cálculos numéricos para aprovechar las bondades del paralelismo masivo que se puede alcanzar con ellas. Los autores utilizan el algoritmo de Golub-Reinsh pues es simple y compacto, además de que puede ser fácilmente adaptado a la estructura SIMD (Única instrucción, múltiples datos) de la GPU.

Bidiagonalización

El primer paso consiste en descomponer la matriz A de la siguiente forma:

$$A = QBP^T$$

aprovechando las transformaciones de householder. En cada iteración se van eliminando renglones y columnas de los datos, de tal forma que al final se obtenga una matriz bidiagonal. Las operaciones que se realizan en las diferentes iteraciones se pueden expresar con las funciones de BLAS nivel 2. Sin embargo, hacer este método de manera secuencial es computacionalmente caro y lento, además de que implica mucha lectura y escritura a memoria. Una manera de contrarrestar este problema es dividiendo la matriz en bloques y actualizar la matriz cuando se hayan finalizado los cálculos. Para esto, se puede dividir la matriz A en bloques de tamaño L, y la actualización de lleva a cabo una vez que las L columnas y renglones se hayan bidiagonalizado.



El valor para L se elige de acuerdo al desempeño de las rutinas de BLAS. Por estas divisiones, el método requiere que se tenga disponible espacio de almacenamiento apra un arreglo de tamaño m x L, para almacenar los resultados que requiere.

- Ejecución en GPU

La ejecución se puede llevar a cabo de manera similar utilizando la librería de CUBLAS, que optimiza los cálculos a través de operaciones en la GPU. Un punto interesante que hacen los autores es que CUBLAS es significativamente más eficiente cuando trabaja con matrices con dimensiones múltiplos de 32. Por lo tanto, para lograr esto, se llenan los espacios remanentes con ceros. Otra consideración adicional al usar GPU es que, dado que la velocidad de transferencia de datos dentro de la GPU es muy alta, y entre CPU y GPU es mucho más baja, se debe de evitar el intercambio de datos entre CPU y GPU en medida de lo posible.

Diagonalización de la matriz bidiagonal

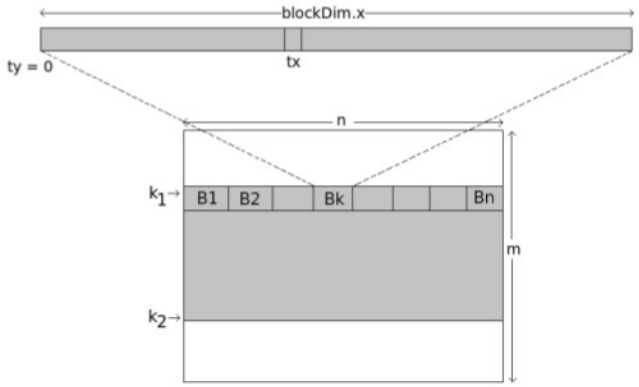
El segundo paso consiste en obtener la matriz diagnal sigma utilizando la matriz B obtenida en el primer paso, bidiagonalización.

$$\Sigma = X^T BY,$$

A través de rotaciones de Given se actualizan los elementos de la diagonal de tal forma que cada vez sean menores al anterior, encontrando así los eigenvalores de la SVD.

- Ejecución en GPU

Para ejecutarlo en la GPU, se copian los elementos de la diagonal y superdiagonal de B a la memoria de GPU. Las rotaciones de Givens se calculan en CPU de forma secuencial, pues solo requieren de acceso a la diagonal y a la superdiagonal. Puesto que la forma del algoritmo implica que para calcular los resultados de un renglón necesites los del anterior, no se puede paralelizar por renglones. Sin embargo, los autores mantienen la ejecución en paralelo al repartir los elementos de cada renglón a diferentes threads en la GPU.

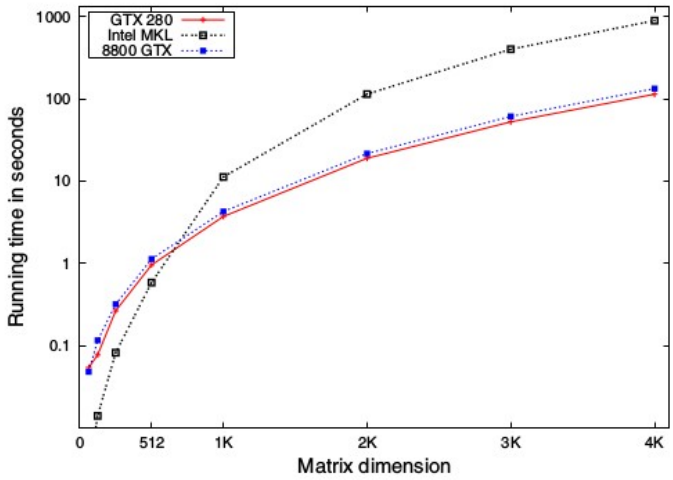


Los autores logran hacer su implementación del algoritmo eficiente, pues realizan exactamente el mismo número de operaciones en paralelo que las que tendrían que ahcer en una ejecución en sencuencial. Para concluir los calculos, se realizan multiplicaciones en CUDA para obtener las matrices ortogonales U y V.

Resultados

Se ejecutó el algortimo en diversos sistemas para comprobar su desempeño:
- Implementación optimizada en CPU de SVD en MATLAB e Intel MKL 10.0.4 LAPACK
- Probaron el algoritmo en una PC con Intel Dual Core 2.66GHz y una de las siguientes:
    - NVIDIA GeForce 8800 GTX graphics processor
    - NVIDIA GTX 280 processor
    - NVIDIA Tesla S1070

Se generaron 10 matrices diferentes y se ejecutó el algortimo 10 veces en cada matriz. Encontraron que el CPU siempre fue más veloz que GPU para matrices pequeñas. Encontraron que la ejecución del algoritmo fue más veloz para matrices cuadradas que para matrices rectangulares, y que para dimensiones más grandes siempre fue significativamente más rápida la GPU.



Además, encontraron que la optimización en tiempo se debe principalmente a mejoras en el paso que diagonalización, más que en el paso de bidiagonalización.

- Discusión con proyecto

Aunque la implementación de SVD presentada en el paper es más sofisticada que la implementación de descenso en gradiente estocástico que realizamos nosotros, enfrentamos problemas similares. Por ejemplo, buscar eliminar la transferencia de datos entre GPU y CPU. En algunos pasos no lo pudimos evitar, pero logramos que la transferencia fuera de vectores pequeños y no de matrices grandes. También aprovechamos las librerías de CUBLAS para ejecutar multiplicaciones entre matrices y matriz-vector.

Al igual que los autores, encontramos mejoras significativas en tiempo con respecto a la ejecución en GPU. Sin embargo, nos quedó pendiente probar nuestro algoritmo con otros modelos de GPU para probar si habría diferencia significativa en velocidad con tarjetas de mayor capacidad, o comprobar si hubieramos llegado al mismo resultado que ellos.