**Reporte de “Singular Value Decomposition on GPU using CUDA”**

**Cristian Ignacio Challú 120652**

**Introducción**

La descomposición en valores singulares (SVD) es una de las descomposiciones de matrices más utilizadas en la práctica. Su aplicación varía desde resolver solución de sistemas de ecuaciones lineales, mínimos cuadrados, componentes principales, entre otras. Poder realizar esta descomposición en menor tiempo es entonces de gran utilidad ya que reduciría el tiempo de ejecución de todos los algoritmos en los que se utiliza.

La descomposición en valores singulares de una matriz A se define como:



donde U es una matriz ortogonal de m x m, V es una matriz ortogonal de n x n y S una matriz diagonal de m x n con entradas no negativas.

Los autores exponen que las GPU’s están siendo cada vez mas utilizadas para el computo científico, pero que todavía no existían muchos trabajos sobre esta descomposición en particular. El objetivo de este articulo entonces es el de presentar una implementación de SVD para matrices densas en la GPU utilizando CUDA.

**Trabajos relacionados**

Se mencionan otros trabajos relacionados con implementaciones de algoritmos en GPU’s y en particular de CUDA. En ellos se presentan algoritmos para resolver diversos problemas comunes como multiplicación de matrices y ordenamiento. En particular hace especial énfasis en artículos relacionados con algoritmos para SVD en GPU’s. Muchos de estos algoritmos logran obtener buenos resultados sólo para para matrices grandes, pero no para matrices pequeñas. Por último, se menciona que existen varias librerías optimizadas para el computo en CPU, como la Intel Math Kernel Library.

**Algoritmo SVD**

Si bien existen muchas formas de obtener la descomposición de valores singulares de una matriz, los autores proponen para su algoritmo utilizar el algoritmo de Golub-Reinsh ya que puede mapearse a la arquitectura de una GPU. Este consta fundamentalmente de realizar primero una biodiagonalización y luego una diagonalización.

Con respecto al primer paso, la bidiagonalización, consta de descomponer la matriz en:



utilizando una serie de transformaciones householder donde B es una matriz bidiagonal y Q y P matrices unitarias de householder. En el articulo se detalla a continuación esta descomposición y los pasos necesarios para obtenerla. Las actualizaciones se pueden obtener utilizando la librera BLAS nivel 2. Es importante el señalamiento de que este método es computacionalmente caro e involucra múltiples lecturas y escrituras a memoria. Es por lo anterior que se divide la matriz A en bloques de tamaño L.

Los autores luego describen como se puede llevar a cabo este algoritmo en una GPU. En esta implementación destaca el uso de la librería CUBLAS, ya que contiene operaciones matriciales optimizadas, como multiplicaciones de matrices y computo de normas. Es particularmente interesante el hecho de que las operaciones de matrices son mas eficientes cuando las dimensiones de las matrices son múltiplos de 32. Se menciona además que la transferencia entre CPU y GPU se debería reducir lo mas posible para obtener un mejor rendimiento, por lo que todas las operaciones necesarias se realizan con las matrices alojadas en la GPU y utilizando CUBLAS.

Con respecto al segundo paso del algoritmo, la diagonalización, esta consta en diagonalizar la matriz obtenida en el paso anterior:



Aplicando iterativamente el algoritmo QR. A continuación, los autores detallan el algoritmo de diagonalización. En cada iteración se actualizan los elementos de la diagonal y de la superdiagonal de tal forma que los elementos de la superdiagonal se achiquen en cada iteración.

Al igual que para el paso anterior, se explica la implementación de este paso en la GPU. Los elementos de la diagonal y de la superdiagonal son copiados a la CPU. Se menciona que para parte del algoritmo se dificulta la paralelización, ya que las operaciones para cada fila dependen de la fila anterior. Es posible sin embargo paralelizar las operaciones necesarias para cada fila. Se utilizan los threads processsors de la GPU para procesar los elementos de cada fila en paralelo. Esta paralelización funciona bien para matrices medianas y grandes, y puede ser implementada eficientemente con CUDA.

**Resultados**

En la sección de resultados se compara el tiempo de ejecución del algoritmo antes mencionado en GPU contra implementaciones en CPU en MATLAB e Intel MKL 10.0.4 LAPACK con diferentes tamaños de matrices. Para realizar esta comparación generaron 10 matrices densas de forma aleatoria para cada tamaño de matriz analizado y promedian los tiempos de ejecución para las 10 matrices.

El principal resultado expuesto es que la implementación de su algoritmo en GPU logra mejoras sustanciales en los tiempos de ejecución, que varia entre 3.04 y 8.2 veces más rápido que la implementación en MKL y 3.32 a 59.3 que la de MATLAB. Esta mejora en el tiempo depende sobre todo del tamaño de la matriz a descomponer. Para matrices pequeñas el algoritmo en CPU es más rápido, pero al aumentar el tamaño de la matriz la implementación en GPU supera ampliamente la implementación en CPU.

Los autores además muestran los tiempos de ejecución para los pasos del algoritmo (biodiagonalización y diagonalización) por separado. Encuentran que la mayor parte de la ganancia en desempeño se logra en el paso de la diagonalización. También menciona que la implementación en GPU’s más potentes no incrementa considerablemente el desempeño, pero pueden lidiar con matrices más grandes por tener más memoria. Su algoritmo puede además se implementado para matrices extremadamente grandes (del orden de 14K). Para este tamaño de matrices no se puede realizar la descomposición en la CPU.

Por último, los autores hacen referencia al hecho de que en la GPU las implementaciones realizadas tienen precisión simple (se puede utilizar más precisión, pero disminuye el desempeño considerablemente). Encuentran que a pesar de lo anterior el error del algoritmo en GPU es menor al 0.001%.

**Conclusiones**

En este artículo se presenta un algoritmo para realizar descomposición de matrices en valores singulares en la GPU. Se comprobó que el desempeño de este es considerablemente superior a la implementación en CPU al aumentar el tamaño de la matriz. Además, este algoritmo permite descomponer matrices mas grandes que la implementación en CPU. El artículo, más allá de tratarse sobre un algoritmo en particular, ejemplifica las mejoras potenciales en desempeño que se pueden alcanzar utilizando las GPU’s (y CUDA en particular).

**Relación con nuestro trabajo final**

Este articula además se relaciona con nuestro trabajo final ya que también utilizamos CUDA y las librerías de CUBLAS. En nuestro caso, que implementamos descenso en gradiente estocástico para el problema de mínimos cuadrados, también observamos mejoras sustanciales con el uso de la GPU.

Es interesante además que en este articulo se compara la implementación en diferentes GPU’s. Ellos observan que utilizar GPU’s más potentes no mejora significativamente el desempeño, lo que es posible que ocurra también con nuestro algoritmo.

Por último, en nuestro caso también tuvimos que lidiar con el problema de minimizar la transferencia entre CPU y GPU. Nuestra implementación final sólo transfiere un vector de tamaño igual al número de coeficientes de la regresión lineal, por lo que no perjudicó el desempeño del algoritmo considerablemente,