Reporte: Singular Value Descomposition on GPU using CUDA

Pablo Soria Garcia 111969

La descomposición en valores singulares o SVD por sus siglas en inglés es una aplicación de álgebra lineal importante para varias ramas de la ciencia, sirve principalmente para encontrar la pseudoinversa de una matriz rectangular (real o compleja) mediante la factorización en tres matrices distintas de la misma. Sus aplicaciones son muy variadas frecuentemente se usa para el desarrollo de algoritmos de Análisis de componentes principales, procesamiento de señales, reconocimiento de patrones y procesamiento o compresión de imágenes.

La descomposición SVD de una matriz de A de mxn es cualquier factorización de la forma:

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

Dónde:

U: es una matriz ortogonal de mxm, V es una matriz ortogonal de m x n y Σ es una matriz diagonal de m x n con elementos distintos de cero en la diagonal y ceros fuera de esta.

El rápido crecimiento del poder de procesamiento de las GPU's hace lógico que la solución optimizada de este tipo de problemas este orientada a este componente de Hardware, el paper defiende que aun cuando hace todo el sentido desde el punto de vista lógico, la descomposición SVD no ha sido ampliamente explorada en su implementación sobre arquitecturas de GPU para matrices particularmente grandes y que las propuestas existentes dejan mucho que desear en términos de rendimiento.

La propuesta de algoritmo del autor utiliza el enfoque de Golub-reinsh que consiste en una Bidiagonalización y posteriormente una Diagonalización para obtener la descomposición SVD. En primer lugar se reduce la matriz A a una matriz bidiagonal que posteriormente es diagonalizada empleando un factorización QR iterativa.

Pseudocódigo:

1.- $B \xleftarrow{¿} Q^T AP$ Bidiagonalización de A a B

2.- $\Sigma \xleftarrow{¿} X^T BY$ Bidiagonalización de B a Σ

3.- $U \xleftarrow{¿} QX$

4.- $V^T \xleftarrow{¿} (PY)^T$ Calcular la matrices ortogonales U y V^T

Bidiagonalización

En el paso de Bidiagonalización, la matriz se descompone como:

$$A = Q B P^T$$

Aplicando una serie de transformaciones de Householder y dónde B es una matriz Bidiagonal y Q y P son matrices unitarias de householder, despejando B de la ecuación anterior nos deja con el primer paso del pseudocódigo resuelto.

El algoritmo de bidiagonalización propuesto por el trabajo, utiliza funciones de CUDA – BLAS (CUBLAS) para calcular de manera eficiente operaciones vector-matriz, matriz-matriz y normas dividiendo en sub-bloques de tamaño L a la matriz original de m x n. Algo particularmente interesante presentado en el trabajo es que todas las matrices cuyas dimensiones no son múltiplos de 32, son re-escaladas con ceros hasta el siguiente múltiplo de 32 para que CUBLAS realice las operaciones de forma más rápido utilizando principios de alineación de bloques de memoria. Adicionalmente menciona que otro enfoque para minimizar el tiempo de ejecución al momento de realizar la bidiagonalización implica minimizar las transferencias de datos entre el CPU y el GPU ya que el ancho de banda de la memoria en estos casos es muy inferior al que se encuentra dentro de la GPU una vez que los datos se encuentran de forma local.

De esta forma una vez terminada la bidiagonalización solamente los elementos de la diagonal y la súper-diagonal son transferidos al CPU para proceder con la siguiente diagonalización de dichos elementos.

Diagonalización de una matriz bidiagonal

Durante este paso, el trabajo propone diagonalizar la matriz resultante aplicando de forma iterativa la factorización QR de esta forma la matriz B encontrada en el paso anterior se descompone de la siguiente forma:

$$\Sigma = X^T BY$$

Dónde Σ es una matriz diagonal, X e Y son matrices unitarias ortogonales. Cada iteración de la factorización QR, actualiza los elementos de la diagonal y de la súper-diagonal de tal forma que los elementos de esta última se van haciendo más pequeño que en la iteración anterior de tal forma que en la convergencia (lograda en este trabajo tras 10 iteraciones) nos encontramos que los elementos de la diagonal con los eigenvalores de B y X e Y^T contienen los eigenvectores de B. El trabajo propone usar una serie de “threads” en el GPU para computar de forma paralela cada renglón, la cantidad de threads lanzados dependerá del tamaño de la matriz pero oscila entre 64 a 256.

Completar la SVD

Finalmente se realizan 2 multiplicaciones matriz-matriz para calcular las matrices ortogonales $U = QX$ y $V^T = (PY)^T$ mediante las rutinas estándar de CUBLAS en la GPU. Estas últimas matrices son copiadas directamente al CPU y los elementos de la diagonal contienen los eigenvalores de la matriz A que implícitamente forman a la matriz Σ.

Enfoque de comparación

El autor compara su algoritmo implementado en CUDA contra una versión optimizada del algoritmo SVD en MATLAB y contra la implementación de INTEL MKL 10.0.4 LAPACK en este último se permitió threading dinámico para mejorar el rendimiento. Para probar

los resultados se utilizó un PC Intel Dual Core @ 2.66GHz, una tarjeta gráfica NVIDIA GeForce 8800 GTX con 180 procesadores (stream) y 768 MB de memoria, un procesador NVIDIA GTX 280 con 240 procesadores (stream) 1 GB de memoria y finalmente una NVIDIA Tesla S1070 con 240 cores y 4 GB de memoria.

En términos generales, los resultados del trabajo muestran que la implementación en CUDA es hasta 60 veces más rápida que el enfoque resuelto por el CPU en MATLAB y hasta 8 veces más rápida que la implementación de INTEL MKL, no solamente esto además el autor muestra que matrices que ni siquiera podrían ser resueltas en el CPU debido a un problema de memoria (m,n > 14k) pueden ser resueltas por el algoritmo sobre una tarjeta gráfica de mediano poder, sin embargo menciona que las GPUs están limitadas números de precisión simple pero que esto implica un error de apenas 0.001% en matrices aleatorias.