Singular Value Decomposition on GPU using CUDA

Elizabeth Solis Díaz

113032

SVD (Descomposición en Valores Singulares por sus siglas en inglés) es una importante técnica empleada en la factorización de matrices. La ventaja
de dicha técnica es que es más robusta ante los posibles errores numéricos. Una de las principales aplicaciones es para obtener la pseudoinversa de una matriz y el anáilisis de componentes principales, procesamiento de señales, problemas de mínimos cuadrados, reconocimiento de patrones, entre otras.

La factorización SVD es de la forma

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

donde

- A es una matriz de m x n
- U es una matriz ortogonal de m x m
- V es una matriz ortogonal de n x m
- Σ es una matriz diagonal con $s_{ij} = 0$ si $i \neq j$ y $s_{ij} \geq 0$ en orden descendente a lo largo de la matriz

Por otro lado, GPU ha resultado ser un candidato bastante fuerte para el procesamiento de tareas en paralelo. Desafortunadamente, la descomposición SVD que tiene gran cantidad de aplicaciones ha sido desarrollada muy poco en GPU.

Existen distintos algoritmos relacionados que han sido desarrollados en GPUs por ejemeplo ordenamiento, multiplicación de matrices, FFT y algoritmos gráficos, entre otros. El paper menciona algunas personas que han realizado mejoras tanto en CUDA, Level 3 BLAS y CUBLAS.

Ahora bien, la descomposicón en valores singulares de una matriz A puede ser calculada mediante el algoritmo de Golub-Reinsch que consiste en bidiagonalización y diagonalización o también mediante el método de Hestenes. El método de Golub-Reinsch es simple y compacto por lo que es popular. Por otra parte, el método de Hestnes tiene menor rendimiento por lo que es menos popular.

El método de Golub-Reinsch se encuentra en el paquete de LAPACK y primero la matriz es biodiagonalizada usando transformaciones de householder y después es diagonalizada. Cabe mencionar que el algoritmo de SVD es del orden $O(mn^2)$ y consiste en 4 pasos mostrados en la Figura 1.

En el primer paso que hace referencia a la Bidiagonalización, se realiza la descomposición de la matriz:

$$A = QBP^T$$

Como hemos mencionado es mediante transformaciones de Householder y donde B es una matriz bidiagonal, Q y P son matrices unitarias householder. Después se procede a obtener la matriz B mediante iteraciones

1: $B \leftarrow Q^T AP$ {Bidiagonalization of A to B}
2: $\Sigma \leftarrow X^T BY$ {Diagonalization of B to Σ}
3: $U \leftarrow QX$
4: $V^T \leftarrow (PY)^T$ {Compute orthogonal matrices U and V^T and SVD of $A = U\Sigma V^T$}

Figure 1:

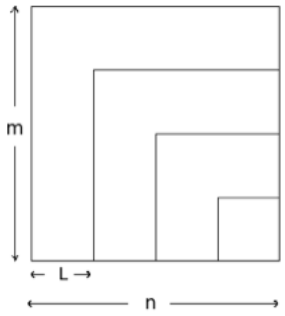

Figure 2:

que consiste en eliminar aquellos elementos debajo de la diagonal por columna y por renglones los que se encuentran por encima de la diagonal. Una vez obtenida la matriz B se procede a obtener la Q y P; destacamos que la matriz se subdivide en bloques de tamaño L con la finalidad de disminuir el gran costo computacional como se muestra en la Figura 2.

Lo anterior se realiza mediante el uso de cuBLAS y se habla de la ventaja notable de trabajar con matrices cuya domensión es en múltiplos de 32. Posteriormente, se procede a la diagonalización de una matriz bidiagonal por medio de la factorización QR de manera iterativa y descomponiendo la matriz B previamente obtenida como

$$\Sigma = X^T B Y$$

donde

- Σ es una matriz
- X y Y son matrices ortogonales y unitarias

El algoritmo consiste en actualizaciones de la diagonal y la super diagonal por lo que se van haciendo más pequeños con cada iteración. Lo que resulta es que los elementos de la diagonal son los valores singulares y X y Y^T contienen los vectore ssingulares de la matriz B.

Después de presentar la diagonalización en GPU se detalla como completar la SVD que consiste en calcular la multiplicación de 2 matrices y al final calcular las matrices ortogonales $U = QX$ y $V^T = (PY)^T$. Para esto se hace uso de las rutinas de multiplicación cuBLAS; las matrices Q, P^T, X^T, Y^T, U, V^T se en encuentran en el *device* y las matrices U, V^T pueden ser copiadas al CPU. Los elementos de la diagonal, los cuales contienen a los valores singulares se encuentran en el CPU.

Finalmente se presentan los resultados de comparar el algoritmo implementado en CUDA y la implementación en MATLAB de SVD y con una versión LAPACK en Intel MKL 10.0.4. Para esto generaron de manera aleatoria 10 matrices densas y cada prueba es replicada 10 veces. Los resultados mostraron que su algoritmo era más rápido que el implementado en MATLAB y en Intel MKL. Se hace mención de que se presentan problemas de memoria si se desea correrlo en CPU por lo que esto no es viable.