Karen Esther Peña 175952

Control de lectura: Cómputo Matricial

“Singular Value Decomposition on GPU using CUDA”

Sheetal Lahabar y P.J.Narayanan

En el artículo se muestra un modelo de programación en CUDA para resolver el problema de la Descomposición en Valores Singulares de una matriz densa de números reales; lo anterior siguiendo los pasos de diagonalización y bidiagonalización.

A modo de introducción, los autores relatan la importancia de la descomposición en valores singulares pues se utiliza resolver diferentes problemas como reconocimiento de patrones, procesamiento de imágenes, mínimos cuadrados, sistemas de ecuaciones lineales.

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

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

Donde U es una matriz ortogonal de m*m , V es una matriz ortogonal de n*n Σ es una matriz diagonal de tamaño (m x n) con elementos $s_{ii} \geq 0$ en orden descendente a lo largo de la diagonal.

Después realiza una introducción sobre el cómputo en paralelo con las GPU´s, los lenguajes que se han desarrollado como Nvidia en CUDA y CTM en ATI/AMD pues estos proveen grandes anchos de banda.

Para ahondar más sobre las referencias, los autores señalan otros trabajos que se han desarrollado como la computación matemática, multiplicación de matrices, multiplicación de matrices y vectores entre otros. Estos desarrollos se han enfocado a la paralelización del algoritmo SVD para matrices densas y casi vacías.

Algoritmo SVD

A groso modo este es el algoritmo:

Algorithm 1 Singular Value Decomposition

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 $A = U\Sigma V^T$}

Para el proceso de diagonalización la matriz A es descompuesta como:

$$A = QBP^T$$

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

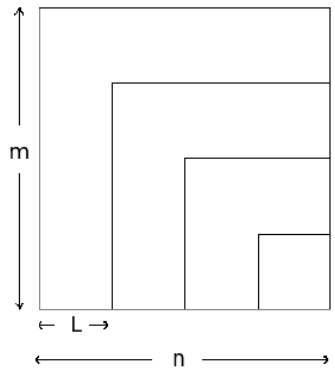
Se obtiene la matriz bidiagonal superior B:

$$B = Q^T AP$$

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.

$$H_1 A G_1 = \begin{bmatrix} \alpha_1 & \beta_1 & 0 & \dots & 0 \\ 0 & x & x & \dots & x \\ \vdots & \vdots & & & \vdots \\ 0 & x & \dots & & x \end{bmatrix}$$

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.



Para la 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.

Por otro lado, al abrir la discusión entre 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 $(3(mL + Ln) + m2 + n2 + mn + 2 \max(m, n)) \times 4$ bytes.

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

$$\Sigma = X^T BY$$

En esta ecuación, Σ es una matriz diagonal, y X y Y son matrices unitarias ortogonales.

En el algoritmo, las *d*(*i*)s son los elementos de la diagonal de B y las *e*(*i*)s son 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 Y^T son los vectores singulares de B. También se presentan los algoritmos para transformar Y^T y X.

En el rubro de diagonalización en la GPU 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 $(m^2 + n^2)$ x 4 bytes en el GPU.

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

Finalmente, se muestran los resultados sobre la implementación optimizada en CPU del SVD en MATLAB y la versión LAPACK en Intel MKL 10.0.4 a partir de ciertas pruebas 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, el 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.

Lo cual es interesante pues para matrices relativamente pequeñas un algoritmo secuencial es más rápido que uno en paralelo.