Singular Value Decomposition on GPU using CUDA

Introducción

Los autores comienzan describiendo la importancia de la descomposición de valores singulares (SVD) que se utiliza para la factorización de matrices reales o complejas. La metodología es usada, entre otras cosas, para temas relacionados a componentes principales, para resolver ecuaciones lineales homogéneas y que se utiliza en la práctica para problemas como reconocimiento de patrones y procesamiento de imágenes.

La SVD de una matriz de tamaño m x n es cualquier factorización de la forma

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

donde:

U es una matriz ortogonal de m x m, V es una matriz ortogonal de n x n, y Σ es una matriz diagonal de m x n con sij = 0 si i $\neq$ j y sii >=0 en orden descendente en la diagonal.

Dada el rendimiento de las tarjetas gráficas, el GPU se ha vuelto un candidato fundamental para desarrollar tareas con grandes cantidades de datos. Éste desarrollo ha venido de la mano con el crecimiento y soporte de lenguajes de programación de alto nivel que permiten programar bajo una interfaz “C like“. Estos desarrollos han provocado que el GPU sea utilizado de por la counidad científica para muchos problemas, sin embargo los autores mencionan que se ha hecho muy poco para resolver problemas como SVD. Por esta razón, en este paper los autores presentan una implementación de la descomposición de valores singulares (SVD) de una matriz densa en GPU utilizando CUDA. La implementación se realizó con los pasos gemelos de bi-diagonalización y diagonalización, la primera se implementa utilizando series de transformación de householder mediante operaciones BLAS y la segunda se implementa aplicando implícitamente el algoritmo QR.

Proyectos relacionados

Los autores hacen una revisión de algoritmos desarrollados en GPU para distintos problemas computacionales como algoritmos de sorteo, multiplicación de matrices, factorización, etc. Posteriormente hacen una recapitulación de lso distintos esfuerzos que existían antes de su implementación por paralelizar el algoritmo SVD como es el caso de Yamamoto et al. donde su propuesta de SVD para matrices rectangulares grandes logra una velocidad 3.5 veces más rápida a intel mkl.

Algoritmo

El autor propone una implementación de solución para SVD en GPU usando la paquetería CUBLAS (implementación de BLAS en GPU) de NVIDIA y kernels de CUDA, propone utilizar el enfoque de Golub-reinsh por su simplicidad y porque mapea bien con la arquitectura de gpu, el algoritmo está implementado en LAPACK mediante un método de dos etapas que consiste en 1) la matriz se reduce a la matriz Bidiagonal y 2) la matriz se diagonaliza implementando una factorización iterativa QR para obtener la descomposición SVD.

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 $A = U\Sigma V^T$}

1.Bidiagonalización de A a B

En la primera etapa la matriz A es descompuesta aplicando transformaciones de householder.

$$A = QBP^T$$

Siendo Q y P matrices householder que se obtienen mediante la elección de vectores u tales que

$A(1:m,1)$ and $\mathbf{v}^{(1)}$ of length n for $A(1,2:n)$ such that

$$\begin{aligned} \hat{A}_1 &= (I - \sigma_{1,1}\mathbf{u}^{(1)}\mathbf{u}^{(1)^T})A(I - \sigma_{2,1}\mathbf{v}^{(1)}\mathbf{v}^{(1)^T}) \quad (3) \\ &= H_1 A G_1 = \begin{bmatrix} \alpha_1 & \beta_1 & 0 & \dots & 0 \\ 0 & x & x & \dots & x \\ \vdots & \vdots & & & \vdots \\ 0 & x & \dots & & x \end{bmatrix}. \end{aligned}$$

La bidiagonalización es una descomposición de matriz en donde se cumple que **U*** **A V = B**, donde **U** y **V** son matrices ortogonales. Al terminar las iteraciones y las columnas y los renglones son eliminados se obtiene la matriz bidiagonal B. Ésto se puede obtener alternando multiplicaciones matriz-vector con actualizaciones rank-uno.

La Matriz A es dividida en bloques de tamaño L y actualizada solamente cuando L columnas y filas han sido bidiagonalizadas. El valor de L se elige dependiendo del rendimiento de las rutinas de BLAS.

Los autores describen el proceso de comunicación y traspaso de información entre el CPU y el GPU que utilizaron para aplicar la bidiaginalización en GPU. Inicialmente la matriz A se debe de encontrar en el CPU y sería transmitida al GPU (es importante buscar minimizar este tipo de transferencia porque el ancho de banda de la memoria no es tan grande) toda las operaciones requeridas se realizaron en el GPU con la información local utilizando rutinas de CUBLAS. Una vez que se ha obtenido a bidiagonalización se copia la diagonal y la superdiagonal al CPU.

2. diagonalización de la matriz bidiagonal

La matriz bidiagonal b obtenida en el primer paso se puede descomponer en diagonal aplicando el algoritmo QR. Se puede descomponer de la siguiente forma

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

Siendo sigma una matriz diagonal y x y y matrices unitarias ortogonales. Cada iteración actualiza el elemento diagonal y superdiagonal de la matriz B de forma tal que la superdiagonal se vuelve menor que su valor anterior. Al converger d(i) contendrá los valores singulares de X y de Y^T

Uno de los problemas con la propuesta del algoritmo secuencial es que las operaciones de fila dependen de las filas anteriores. Para hacer este proceso en versión paralelizada en el GPU los autores describen que una vez que se obtuvo la bidiagonalización los elementos diagonal y superdiagonal de B se copian al CPU, y utilizan los procesadores, el algoritmo divide las filas de las matrices en bloques para hacer cada fila en paralelo y de forma independiente. La información necesaria para el cómputo por bloque se almacena en la memoria compartida. Cuando se obtienen los coeficientes estos se copian del CPU a la memoria del device. Como el proceso por renglón es similar al de columna se aplica lo mismo.

3.Calcular la matrices ortogonales

Finalmente se calculan dos multiplicaciones de matriz por matriz al final para obtener las las matrices ortogonales U y V^T que serán copiadas al CPU.

Evaluación y conclusiones

Para evaluar el desempeño de su método los autores generaron 10 matrices densas aleatorias para cada uno de los tamaños definidos. Se encontró que el promedio no cambió mucho si se usaban 10 o más matrices pero cuando se tienen matrices cuadradas muy grandes la velocidad incrementa con el tamaño. Los autores concluyen que el algoritmo explota de forma eficiente la paralelización de la arquitectura de GPU para mejorar el performance. La implementación híbrida de la diagonalización logra separar el proceso entre el CPU y el GPU por lo que logra que esta implementación sea más rápida que la de SVD de Intel MKL y la de MATLAB.