Singular Value Decomposition on GPU using CUDA

Abraham Nieto 51556

31 de mayo de 2018

El paper habla de hacer la descomposición SVD en GPUs a tráves de CUDA, recordar que la descomposición SVD es de la forma $A = U * \Sigma * V^t$ donde A es ua matriz de dimensión mxn , U es uma matriz de mxm ortogonal, V es una matriz de nxn ortogonal y Σ es una matriz diagonal de mxn con elementos $\sigma_{i,j} = 0$ para $i \neq j$ y $\sigma_{i,i} > 0$.

Últimamente se ha incrementado el uso de las GPU's para cómputo científico más allá de las gráficas, SVD es un ejemplo de esto, muchos algoritmos se han desarrollado usando GPU's para cómputo matemático con el objetivo de poder explotar el paralelismo de las GPU's. Se han realizado muchos esfuerzos para paralelizar el Algoritmo SVD en arquitecturas como FPGA, Cell Processors, GPU, etc., que tienen una arquitectura paralela escalable, por ejemplo Ma et al. [19] propuso la implementación de doble cara algoritmo SVD de rotación de Jacobi en un FPGA, Bobda et al. [6] propuso una implementación eficiente de la SVD para matrices grandes y la posibilidad de integrar FPGA's como parte de un Sistema Reconfigurable Distribuido, etc.

Zhang Shu presentó la implementación del método One Sided Jacobi para SVD en GPU usando CUDA. El rendimiento de su algoritmo está limitado por la disponibilidad de memoria compartida y funciona bien solo para matrices de pequeño tamaño. Bondhugula propuso un híbrido Implementación basada en GPU de descomposición de valores singulares utilizando sombreadores de fragmentos y objetos de búfer de cuadros en los que la diagonalización se realizaría en la CPU.

Se puede desarrollar el algoritmo SVD usando el método de Golub-Reinsch que consiste en 2 pasos: 1.-reducir la matriz original a una matriz bidiagonal(Bidiagonalización): La matriz se reduce primero a una matriz bidiagonal utilizando una serie de transformaciones.

En este paso dada una matriz A es descompuesta como:

$$A = QBP^T$$

Aplicando una serie de transformaciones householder donde B es unamatriz bidiagonal y Q y P son matrices de householder unitarias. Donde

$$Q^T = \prod_{i=1}^{n} H_i, \; P = \prod_{i=1}^{n-2} G_i$$

con $H_i = I - \sigma_{1,i} u^(i) u^(i)^T$ y $G_i = I - \sigma_{2,i} v^(i) v^(i)^T$.

$u^(i)$'s son los vectores de tamaño m con i-1 ceros y $v^(i)$'s son los vectores de tamaño n con i ceros.

En otras palabras la bidiagonalización puede lograrse alternándose el vector matricial multiplicado por las actualizaciones de rango uno introducidas por Golub y Kahan.

Las matrices Q y P dadas se calculan de manera similar, ya que también involucran la multiplicación por H i 's y G i' s, respectivamente, pero en orden inverso. Usamos el término parcial bidiagonalización para referirnos al cálculo de la matriz B, sin hacer un seguimiento de las matrices P y Q. Esto es computacionalmente menos costoso que completa bidiagonalización y fue la operación implementada en la GPU por Bondhugula.

Algoritmo 2

El algoritmo 2 describe el procedimiento de bidiagonalización. Cada paso se puede realizar usando las funciones CUDA BLAS. CUBLAS proporciona un alto rendimiento de matriz-vector, multiplicación matriz-matriz y función de cálculo de norma. El enfoque para la bidiagonalización se puede realizar de manera eficiente ya que CUBLAS ofrece un alto rendimiento para matriz-vector, la multiplicación matriz-matriz incluso si una de las dimensiones es pequeña. Los experimentos demuestran que CUBLAS entregan mucho más rendimiento

Algorithm 2 Bidiagonalization algorithm

Require: $m \geq n$

1: $kMax \leftarrow \frac{n}{L}$ {L is the block size}
2: **for** $i = 1$ *to* $kMax$ **do**
3: $t \leftarrow L(i-1)+1$
4: Compute $\hat{\mathbf{u}}^{(t)}$, $\alpha_{1,t}, \sigma_{1,t}, \hat{\mathbf{k}}^{(t)}$
5: Eliminate $A(t:m,t)$ and update $Q(1:m,t)$
6: Compute new $A(t,t+1:n)$
7: Compute $\hat{\mathbf{v}}^{(t)}$, $\alpha_{2,t}, \sigma_{2,t}, \hat{\mathbf{l}}^{(t)}$
8: Eliminate $A(t,t+1:n)$ and update $P^T(t,1:n)$
9: Compute $\hat{\mathbf{w}}^{(t)}$, $\hat{\mathbf{z}}^{(t)}$ and store the vectors
10: **for** $k = 2$ *to* L **do**
11: $t \leftarrow L(i-1)+k$
12: Compute new $A(t:m,t)$ using $k-1$ update vectors
13: Compute $\hat{\mathbf{u}}^{(t)}$, $\alpha_{1,t}, \sigma_{1,t}, \hat{\mathbf{k}}^{(t)}$
14: Eliminate $A(t:m,t)$ and update $Q(1:m,t)$
15: Compute new $A(t,t+1:n)$
16: Compute $\hat{\mathbf{v}}^{(t)}$, $\alpha_{2,t}, \sigma_{2,t}, \hat{\mathbf{l}}^{(t)}$
17: Eliminate $A(t,t+1:n)$ and update $P^T(t,1:n)$
18: Compute $\hat{\mathbf{w}}^{(t)}$, $\hat{\mathbf{z}}^{(t)}$ and store the vectors
19: **end for**
20: Update $A(iL+1:m, iL+1:n)$, $Q(1:m, iL+1:m)$ and $P^T(iL+1:n, 1:n)$
21: **end for**

Figure 1:

cuando se opera en matrices con dimensiones que son un múltiplo de 32 debido a problemas de alineación de memoria. El rendimiento de las bibliotecas de GPU depende de la ubicación de los datos y de cómo se usa la biblioteca.

2.- diagonalizar la matriz encontrada en el paso 1(Diagonalización): La matriz bidiagonal se diagonaliza luego de realizar desplazamientos QR implícitamente desplazados.

SVD es un algoritmo de orden $O(mn^2)$ para $m >= n$.

La matriz bidiagonal puede ser reducida a una matriz diagonal aplicando iterativamente el algoritmo QR entonces la matriz B bidiagonal puede descomponerse como

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

donde Σ es un matriz diagonal y X y Y son matrices ortonormales, cada iteración actualiza la diagonal y los elementos de la super diagonal tales que el valor de los elementos de la super diagonal son menores que su valor anterior.

Con respecto a la diagonalización en los GPU's la diagonal y la superdiagonal de los elementos de B son copiados al CPU aplicando rotaciones de Givens en B y calculando los vectores de coeficientes se raliza secuencialmente en la CPU ya que sólo requiere acceso a los elementos de la diagonal y superdiagonal.

Se usan los threads de la GPU para procesar elementos de cada fila en paralelo. Esto proporciona un alto rendimiento en matrices grandes pero también funciona bien para matrices de tamaño mediano.

cada thread opera en un elemento de la fila, Esta división de la fila en bloques y bucle se puede hacer de manera eficiente en Arquitectura CUDA, ya que cada thread realiza de forma independiente cómputos. Los datos requeridos para el bloque se almacenan en la memoria compartida y las operaciones se pueden realizar de manera eficiente en un multiprocesador.

Algoritmo para la Matriz A.

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 $A = U\Sigma V^T$}

Figure 2:

Para la implementación del SVD se usó la multiplicación de matriz CUBLAS rutinas de curación Las matrices Q, P T, X T, Y T, U y V T son en el dispositivo. Las matrices ortogonales U y V T pueden entonces ser copiado a la CPU. d (i) 's contiene los valores singulares, es decir, elementos diagonales de Σ y está en la CPU.

Con respecto a los resultados se hicieron comparativos entre el desempeño de matlab vs.los GPUs con nvidia , para matrices cuadradas cuando la dimensión de esta es mayor a 512 el performance es es mejor con las GPUs, cuando las matrices son rectangulares a cualquier dimensión el método de las GPUs siempre es más eficiente.

En conclusión el algoritmo de descomposiciónde svd explota l paralelismo de las GPUs, la bidiagonalización de la matriz es procesada por las GPUs usando la librería CUBLAS optimizada para maximizar el peformance, se utilizó una implementación híbrida para la diagonalización de la matriz que divide el computo entre el CPU y GPU.