Reporte: Singular Value Decomposition on GPU using CUDA

Victor Quintero Marmol Gonzalez 175897

31 de mayo de 2018

Abstracto

En este artículo escrito por *Sheetal Lahabar* y *P. J. Narayanan* en el 2009, se presenta la implementación de la Descomposición en Valores Singulares (SVD, por sus siglas en inglés) para una matriz densa en GPU usando el modelo de programación CUDA. La SVD se implementó utilizando bidiagonalización (utilizando series de de transformaciones de Householder) seguido de una diagonalización. Algo muy importante es que este tipo de implementación no se hababía hecho nunca antes en GPU.

Introducción

Los autores resaltan la importancia de la SVD mencionando que se ocupa para resolver ecuaciones lineales homogeneas, resolver problema de mínimos cuadrados, reconocimiento de patrones, procesamiento de imágenes para su análisis espectral, entre otras.

La descomposición en valores singulares de una matriz A de dimensiones $m \times n$ es una factorización de la forma:

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

donde U es una matriz ortogonal $m \times m$, V es una matriz ortogonal $n \times n$ y Σ es una matriz diagonal $m \times n$.

Se hace mención al rápido incremento en el rendimiento de hardware de gráficos, lo que ha connvertido a la GPU en un candidato fuerte para realizar muchas tareas intensivas de computo, en especial tareas con paralelización de datos; y aunque se han hecho muchos trabajos y artículos científicos utilizando GPU, se menciona que ha habido muy poca investigación y trabajos para resolver problemas como la SVD que tiene muchas aplicaciones.

Trabajo relacionado.

Se hace mención a varios trabajos cuyos algotimos han sido implementados usando GPU, como computación matemática y geométrica, multiplicación de matrices y algoritmos de grafos. También se hace mención de los esfuerzos que se han hecho para optimizar y ajustar el nivel 3 de CUBLAS. Mencionan además que se han desarrollado trabajos para paralelizar el algoritmo de SVD sobre una arquitectura FPGA, Procesadores de Celdas, entre otros, los cuales tienen una arquitectura paralela y escalable.

Algoritmo SVD

En esta sección se hace mención a que la SVD de una matriz A puede calcularse usando el algoritmo de GolubReinsch (bidiagonalización y diagonalización) o el método de Hestenes. Los autores se decidieron por utilizar el algoritmo de GolubReinsch ya que es simple y compacto, además de que se adapta bien a la arquitectura de una GPU. Este algoritmo se encuentra en la paquetería de LAPACK y consta de dos pasos. El primero es reducir la matriz a una matriz bidiagonal utilizando una serie de transformaciones de Housholder. El segundo es diagonalizar la matriz bidiagonal realizando implícitamente iteraciones desplazadas QR.

El siguiente algoritmo presentado por los autores describe el algoritmo SVD para una matriz A dada:

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 $A = U\Sigma V^T$}

Bidiagonalización

En este paso, la matriz A dada es descompuesta como:

$$A = QBP^T$$

Donde B es una matriz bidiagonal y Q y P son matrices Householder unitarias.

La descomopocisión se hace usando una serie de transformaciones de Housholder de la forma:

$$H_1AG_1 = \begin{bmatrix} \alpha_1 & \beta_1 & 0 & \dots & 0 \\ 0 & x & x & \dots & x \\ \vdots & \vdots & & & \vdots \\ 0 & x & \dots & & x \end{bmatrix}$$

Las matrices Housholder H_i y G_i son de dimensiones $m \times m$ y $n \times n$ respectivamente.

Al final de todo el proceso se obtiene una matirz bidiagonal B tal que:

$$B = Q^T AP$$

Las matrices P y Q se caculan de forma similar ya que involucran multiplicaciones de las matrices Housholder, Q^T es la multiplicación de las matrices H_i y P es la multiplicación de las matrices G_i.

Los autores llaman a este paso *bidiagonalización parcial* donde calculan la matriz B sin mantener guardadas las matrices P y Q.

Se hace mención que la implementación en LAPACK utiliza aproximación por bloques, donde la matris A es dividida en bloques de tamaño L y la actualización ocurre sólo después de que L columnas y renglones son bidiagonalizados.

Bidiagonalización en GPU

En esta sección los autores muestran el algoritmo para realizar la bidiagonalización en GPU, usando en cada paso CUDA BLAS (CUBLAS), además se menciona que la aproximación por bloques para la bidiagonalización puede ser desempeñada de manera eficiente en CUBLAS.

El algoritmo dado por los autores es el siguiente:

Algorithm 2 Bidiagonalization algorithm

Require: $m \geq n$

$kMax \leftarrow \frac{n}{L}$ {L is the block size}
for $i = 1$ *to* $kMax$ **do**
  $t \leftarrow L(i-1)+1$
  Compute $\hat{\mathbf{u}}^{(t)}$, $\alpha_{1,t}, \sigma_{1,t}, \hat{\mathbf{k}}^{(t)}$
  Eliminate $A(t:m,t)$ and update $Q(1:m,t)$
  Compute new $A(t, t+1:n)$
  Compute $\hat{\mathbf{v}}^{(t)}$, $\alpha_{2,t}, \sigma_{2,t}, \hat{\mathbf{l}}^{(t)}$
  Eliminate $A(t, t+1:n)$ and update $P^T(t, 1:n)$
  Compute $\hat{\mathbf{w}}^{(t)}$, $\hat{\mathbf{z}}^{(t)}$ and store the vectors
  for $k = 2$ *to* L **do**
    $t \leftarrow L(i-1)+k$
    Compute new $A(t:m,t)$ using $k-1$ update vectors
    Compute $\hat{\mathbf{u}}^{(t)}$, $\alpha_{1,t}, \sigma_{1,t}, \hat{\mathbf{k}}^{(t)}$
    Eliminate $A(t:m,t)$ and update $Q(1:m,t)$
    Compute new $A(t, t+1:n)$
    Compute $\hat{\mathbf{v}}^{(t)}$, $\alpha_{2,t}, \sigma_{2,t}, \hat{\mathbf{l}}^{(t)}$
    Eliminate $A(t, t+1:n)$ and update $P^T(t, 1:n)$
    Compute $\hat{\mathbf{w}}^{(t)}$, $\hat{\mathbf{z}}^{(t)}$ and store the vectors
  end for
  Update $A(iL+1:m, iL+1:n)$, $Q(1:m, iL+1:m)$ and $P^T(iL+1:n, 1:n)$
end for

Otro aspecto que se trata en esta parte es que la bidiagonalización se hace *inplace*, donde A se vuelve la matriz bidiagonal.

Diagonalización de una matriz bidiagonal

La matriz bidiagonal puede reducirse a una matriz diagonal aplicando iterativamente el algoritmo QR. La matriz B obtenida en el paso anterior se descompe como:

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

Donde Σ es una matriz diagonal y X y Y son matrices ortogonales unitarias.

El algoritmo dado por los autores es el siguiente. Además se dan los algoritmos para las transformaciones *Forward* y *Backward* para los renglones de Y^T.

Algorithm 3 Diagonalization algorithm

iter $\leftarrow 0$
maxitr $\leftarrow 12 * N * N$ {N is the number of main diagonal elements}
$k_2 \leftarrow N$ {k_2 points to the last element of unconverged part of matrix}
for $i = 1$ to *maxitr* **do**
  if $k_2 <= 1$ **then**
    break the loop
  end if
  if *iter* $>$ *maxitr* **then**
    return *false*
  end if
  matrixsplitflag $\leftarrow$ *false*
  for $l = 1$ to $k_2 - 1$ **do**
    $k_1 \leftarrow k_2 - l$ {Find diagonal block matrix to work on}
    if *abs*$(e(k_1)) <=$ *thres* **then**
      matrixsplitflag $\leftarrow$ *true*, break the loop
    end if
  end for
  if !*matrixsplitflag* **then**
    $k_1 \leftarrow 1$
  else
    $e(k_1) \leftarrow 0$
    if $k_1 == k_2 - 1$ **then**
      $k_2 \leftarrow k_2 - 1$, continue with next iteration
    end if
  end if
  $k_1 = k_1 + 1$
  if $k_1 == k_2 - 1$ **then**
    Compute SVD of 2×2 block and coefficient vectors $\mathbf{C}_1$, $\mathbf{S}_1$ and $\mathbf{C}_2$, $\mathbf{S}_2$ of length 1
    Apply forward row transformation on the rows $k_2 - 1$ and k_2 of Y^T using $\mathbf{C}_1$, $\mathbf{S}_1$
    Apply forward column transformation on the columns $k_2 - 1$ and k_2 of X using $\mathbf{C}_2$, $\mathbf{S}_2$
    $k_2 \leftarrow k_2 - 2$, continue with next iteration
  end if
  Select shift direction: *forward* if $d(k_1) < d(k_2)$, else *backward*
  Apply convergence test on the sub block, continue next iteration if any value converges
  Compute the $shift$ from 2×2 block at the end of the sub matrix
  iter $\leftarrow$ *iter*$+k_2 - k_1$
  Apply simplified/shifted forward/backward Givens rotation on the rows k_1 to k_2 of B and compute $\mathbf{C}_1$, $\mathbf{S}_1$ and $\mathbf{C}_2$, $\mathbf{S}_2$ of length $k_2 - k_1$
  Apply forward/backward transformation on the rows k_1 to k_2 of Y^T using $\mathbf{C}_1$, $\mathbf{S}_1$
  Apply forward/backward transformation on the columns k_1 to k_2 of X using $\mathbf{C}_2$, $\mathbf{S}_2$
end for
Sort the singular values and corresponding singular vectors in decreasing order

Algorithm 4 Forward transformation on the rows of Y^T

Require: $k_1 < k_2$
for j=k_1 to $k_2 - 1$ **do**
  $\mathbf{t} \leftarrow Y^T(j+1, 1:n)\mathbf{C}_1(j - k_1 + 1)$
  $\mathbf{t} \leftarrow \mathbf{t} - Y^T(j, 1:n)\mathbf{S}_1(j - k_1 + 1)$
  $Y^T(j, 1:n) \leftarrow Y^T(j, 1:n)\mathbf{C}_1(j - k_1 + 1) + Y^T(j + 1, 1:n)\mathbf{S}_1(j - k_1 + 1)$
  $Y^T(j+1, 1:n) \leftarrow \mathbf{t}$
end for

Algorithm 5 Backward transformation on the rows of Y^T

Require: $k_1 < k_2$
for j=$k_2 - 1$ to k_1 **do**
  $\mathbf{t} \leftarrow Y^T(j+1, 1:n)\mathbf{C}_1(j - k_1 + 1)$
  $\mathbf{t} \leftarrow \mathbf{t} - Y^T(j, 1:n)\mathbf{S}_1(j - k_1 + 1)$
  $Y^T(j, 1:n) \leftarrow Y^T(j, 1:n)\mathbf{C}_1(j - k_1 + 1) + Y^T(j + 1, 1:n)\mathbf{S}_1(j - k_1 + 1)$
  $Y^T(j+1, 1:n) \leftarrow \mathbf{t}$
end for

Diagonalización en GPU

En esta sección los autores presentan la versión en paralelo del algoritmo de diagonalización y su implementación en GPU. La diagonal y superdiagonal de B son copiados a CPU. Se aplica rotaciones de Givens a B y el cálculo de los coeficientes de los vectores son hechos de manera secuencial en el CPU. Esta parte es muy compleja para poder resumirla de manera efectiva en este reporte, por lo que se invita a leer el artículo original para un mayor entendimiento.

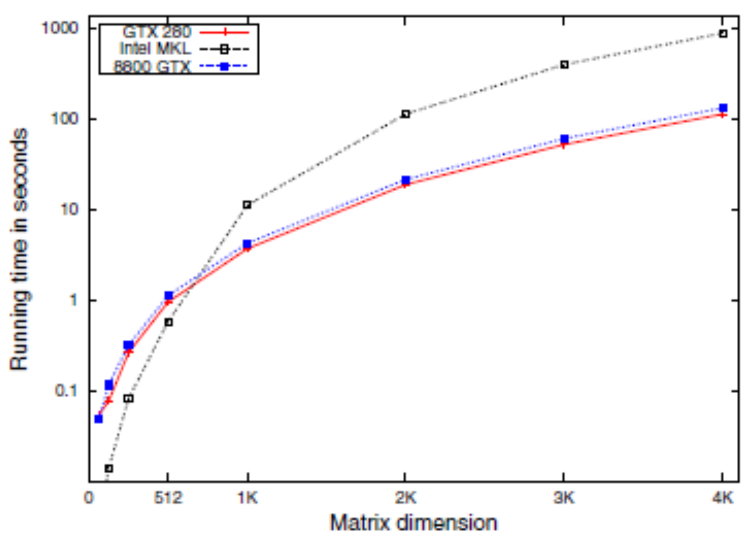
SVD Completo

Los autores nos hablan de que realizan la multiplicación de dos matrices al final para calcular las matrices ortogonales $U = QX$ y $V^T = (PY)^T$. Se usó las rutinas de multiplicación de matrices de CUBLAS.

Resultados

En esta sección los autores analizán sus resultados obtenidos con implementación CPU optimizada de SVD en MATLAB e Intel MKL 10.0.4 LAPACK. Para esto generaron de manera aleatoria 10 matrices densas. El algoritmo SVD se ejecutó a cada matriz 10 veces.

Entre sus resultados muestran que su algortimo fue entre 3.04 y 8.2 veces más rápido que la Intel MKL y entre 3.32 y 59.3 veces más rápido que MATLAB.



Conclusiones

La conclusión de los autores es que se explotó de manera adecuada la paralelización en GPU, logrando un gran desempeño computacional. Además, lograron implementar la bidiagonalización completamente en paralelo. otra mención importante que hacen es que se logró calcular SVD para matrices del orden 14K, lo que es imposible lograr en CPU debido a limitaciones de memoria.

Como conclusión personal pienso que el paper demuestra que aun queda muchas cosas por implementar en paralelo y que son de gran ayuda en diferentes campos, desde el científico hasta el empresarial, por lo que siempre es bueno seguir explorando nuevas tecnologías para mejorar en la resolución de estos problemas.