Reporte: “Singular Value Decomposition on GPU using CUDA”

Miguel Castañeda 175840

31 de mayo de 2018

La descomposición en valores singulares (SVD por sus siglas en inglés) es técnica importante utilizada en la factorización de una matriz rectangular real o compleja. Los cálculos matriciales usados son más robustos a errores numéricos lo que los hace ideales para determinar la pseudoinveresa de una matriz, resolver el problema de mínimos cuadrados, entre otros tales como procesamiento de señales.

Una SVD de una matriz A de mxn es una factorización de la forma:

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

Donde U es una matriz ortogonal de mxn, V es una matriz ortogonal de nxn y Σ es una matriz diagonal de mxn con los elementos $s_{ij} = 0$ si $i \neq j$ y $s_{ij} \geq 0$ en orden descendiente a lo largo de la diagonal.

El rápido incremento en el rendimiento de los procesadores gráficos han hecho que sean el candidato ideal para realizar tareas de computo intensivo especialmente en procesamiento paralelo, los GPU incluyen ahora plataformas para la programación que han incluido lenguajes de alto nivel tales como extensiones para el lenguaje C.

El artículo presenta una implementación de SVD para matrices densas en GPU usando la plataforma CUDA, la cual esta basada en las bibliotecas de CUDA CUBLAS y la programación de kernels CUDA. De acuerdo a los autores lograron una mejora con respecto a las implementaciones realizadas en MATLAB e Intel MKL y que fueron capaces de calcular SVD sobre matrices muy grandes lo que no hubiera sido posible con CPU dado las limitaciones de memoria.

Para determinar la SVD de una matriz A los autores utilizan el algoritmo de Golub-Reinsch (Bidiagonalización – Diagonalización) dado que es simple y compacto y se adapta bien a la arquitectura SIMD GPU, este algoritmo es usando en la biblioteca LAPACK.

El algoritmo consiste a grandes rasgos en realizar:

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 $A = U\Sigma V^T$}

Paso 1:

1. En este paso dada una matriz A esta se descompone en:

$$A = QBP^T$$

Aplicando transformaciones de householder donde B es una matriz bidiagonal y Q y P son matrices unitarias householder, despues de realizar las transformaciones obtenemos una matriz bidiagonal B tal que:

$$B = Q^T AP$$

Donde

$$Q^T = \prod_{i=1}^{n} H_i$$

$$P = \prod_{i=1}^{n-2} G_i$$

Las matrices householder Q y P involucran la multiplicación de H_i y G_i pero en orden inverso, se usa el término de bidiagonalización parcial al calculo de la matriz B la cual es computacionalmente menos costosa.

El algoritmo de bidiagonalización es:

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

Cada paso del algoritmo se puede ejecutar usando funciones CUDA BLAS, de acuerdo a experimentos se ha encontrado un mejor rendimiento cuando las matrices son multiplos de 32 de ahí que se pueda rellenar de ceros para que las dimensiones sean múltiplos de 32. Otro punto importante a considerar es el tiempo de mover los datos de la memoria de la CPU a la GPU por lo que las matrices Q, P^T, U_{mat} , V_{mat} y P_{mat} se deben inicializar en el device.

2. En este paso se realiza la diagonalización de una matriz bidiagonal mediante la aplicación iterativa del algoritmo QR, la matriz B obtenida en el paso 1 es descompuesta como:

$$\Sigma = X^T BY$$

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

El algoritmo es:

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

Para su implementación en la GPU se copian los elementos de la diagonal y subdiagonal a la CPU se aplacan rotaciones de Givens a B y se calcula el vector de coeficientes lo cual se puede hacer de forma secuencial directamente en la CPU, para el algoritmo 4:

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

Los cálculos para cada renglón dependes solo del siguiente renglón y para

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

Los cálculos en un renglón dependen solo del renglón superior, y esos resultados pueden ser calculados en paralelo.

Finalmente en el paso 3 y 4 se realiza la multiplicación de matrices para obtener:

$$U = QX$$

y

$$V^T = (PTY)^T$$

Para su implementación en la GPU los autores usaron la implementación de CUBLAS, donde las matrices Q, P^T, X^T, Y^T, U^T y V^T se encuentran en el device y las matrices ortogonales U y V^T se pueden copiar a la CPU.

Para obtener los resultados las pruebas se realizaron en una PC Intel Dual Core 2.66GHz PC, y tarjetas gráficas NVIDIA GeForce 8800 GTX, NVIDIA Tesla S1070. Generando 10 matrices densas con valores aleatorios single precision, el algoritmo SVD se ejecutó 10 veces para cada matriz para evitar buenas y malas muestras realizaron el promedio de los valores generados.

Realizaron el cálculo del SVD en matrices muy grandes del orden de 14K lo cual es imposible en el CPU dadas sus limitaciones de memoria. Obtuvieron una mejora de 3 a 8 sobre las implementaciones de Intel MKL y de 3 a 60 en la implementación de MATLAB, por otro lado el error fue menor al 0.001%.