Singular Value Decomposition on GPU using CUDA

Introducción

La descomposición de valores singulares es utilizada para factorización de una matriz rectangular real o compleja y son más robustas a errores numéricos. Comúnmente es utilizada para calcular la pseudoinversa de una matriz, resolver ecuaciones lineales homogéneas, resolver problemas de minimización por mínimos cuadrados y encontrar aproximaciones de una matriz.

Una SVD de una matriz A “m x n” es de la forma:

A = UΣVT

Donde U es una matriz ortogonal mxm, V es una matriz ortogonal nxn y Σ es una matriz diagonal mxn con elementos sij = 0 si I es distinto de j y sii ≥ 0 en orden decreciente a lo largo de la diagonal.

En la actualidad ha crecido rápidamente el performance de las unidades de procesamiento de gráficos, lo que ha facilitado el procesamiento de tareas en paralelo. En especial la marca NVIDIA ha creado muchas nuevas tarjetas con capacidades de procesamiento mucho más potentes, abriendo la posibilidad de realizar tareas para cómputo científico, pero poco trabajo para SVD, lo cual se expone en este paper.

Trabajo Relacionado

A lo largo de los años se han desarrollado muchos algoritmos en GPUs para cómputo matemático con diferentes ejemplos que van desde: operadores de álgebra lineal, reducción de descomposición matricial, hasta utilizar NVIDIA CUDA para multiplicaciones matriz-vector. Como se puede ver en el paper, muchos diferentes algoritmos han sido desarrollados para el cómputo en paralelo, lo que facilita el procesamiento de cómputo científico.

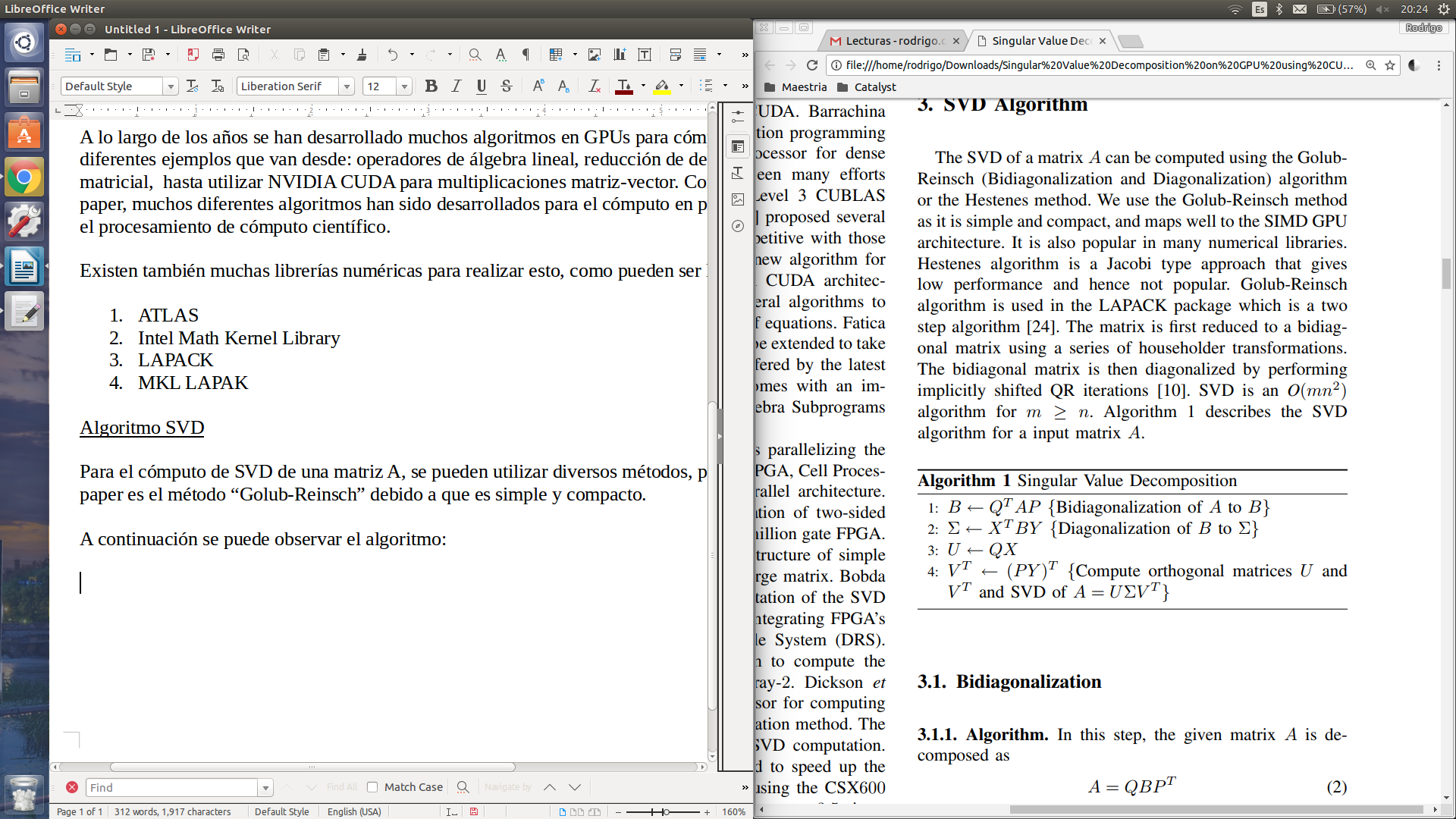
Existen también muchas librerías numéricas para realizar esto, como pueden ser lo siguientes ejemplos:

1. ATLAS
2. Intel Math Kernel Library
3. LAPACK
4. MKL LAPAK

Algoritmo SVD

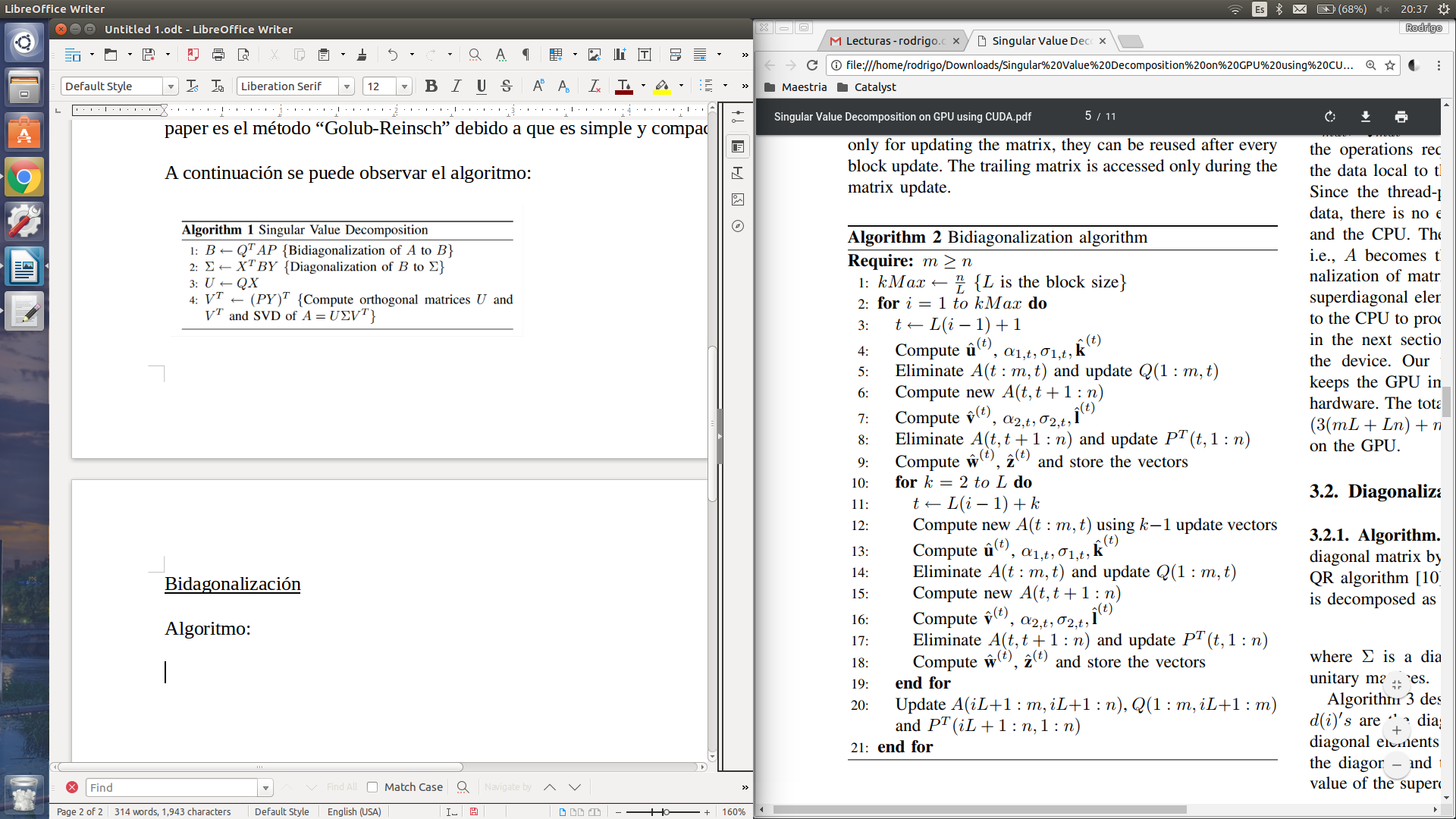
Para el cómputo de SVD de una matriz A, se pueden utilizar diversos métodos, pero el aplicado en este paper es el método “Golub-Reinsch” debido a que es simple y compacto.

A continuación, se puede observar el algoritmo:



Bidagonalización de una matriz

Algoritmo:



Para realizar este procesamiento en una GPU, el algoritmo anterior se puede completar utilizando funciones CUBLAS y el acercamiento de ‘blocking’ puede realizarse eficientemente debido a que se tiene un alto desempeño. Aunque el desempeño de las librerías en GPU depende de su colocación y de cómo la librería es utilizada. Se deben tener las siguientes consideraciones:

1. El movimiento de la información es de consideración cuando se utiliza BLAS
2. El input de la matriz A se transfiere del CPU al GPU
3. Las matrices Q, PT, Umat, Vmat, Wmat, Zmat, Qmat y Pmat son inicializadas en el dispositivo
4. Todas las operaciones requeridas para bidagonalización se realizan en el GPU utilizando CUBLAS

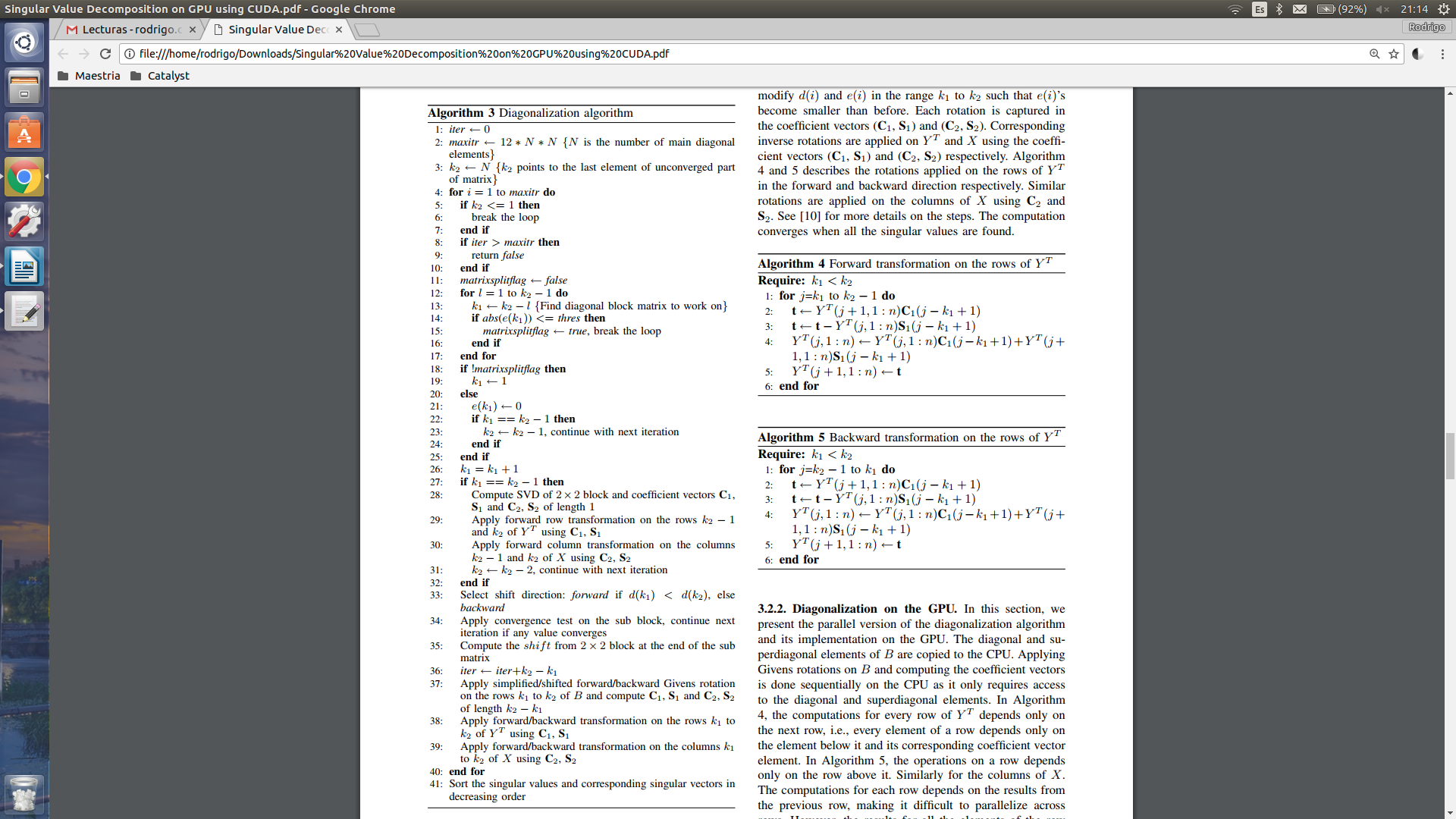
Diagonalización de una matriz bidagonal

Algoritmo:

La matriz bidiagonal puede ser reducida a una matriz diagonal aplicando iterativamente el algoritmo QR implícitamente cambiado y la matriz B obtenida en el primer paso se descompone de la siguiente manera:

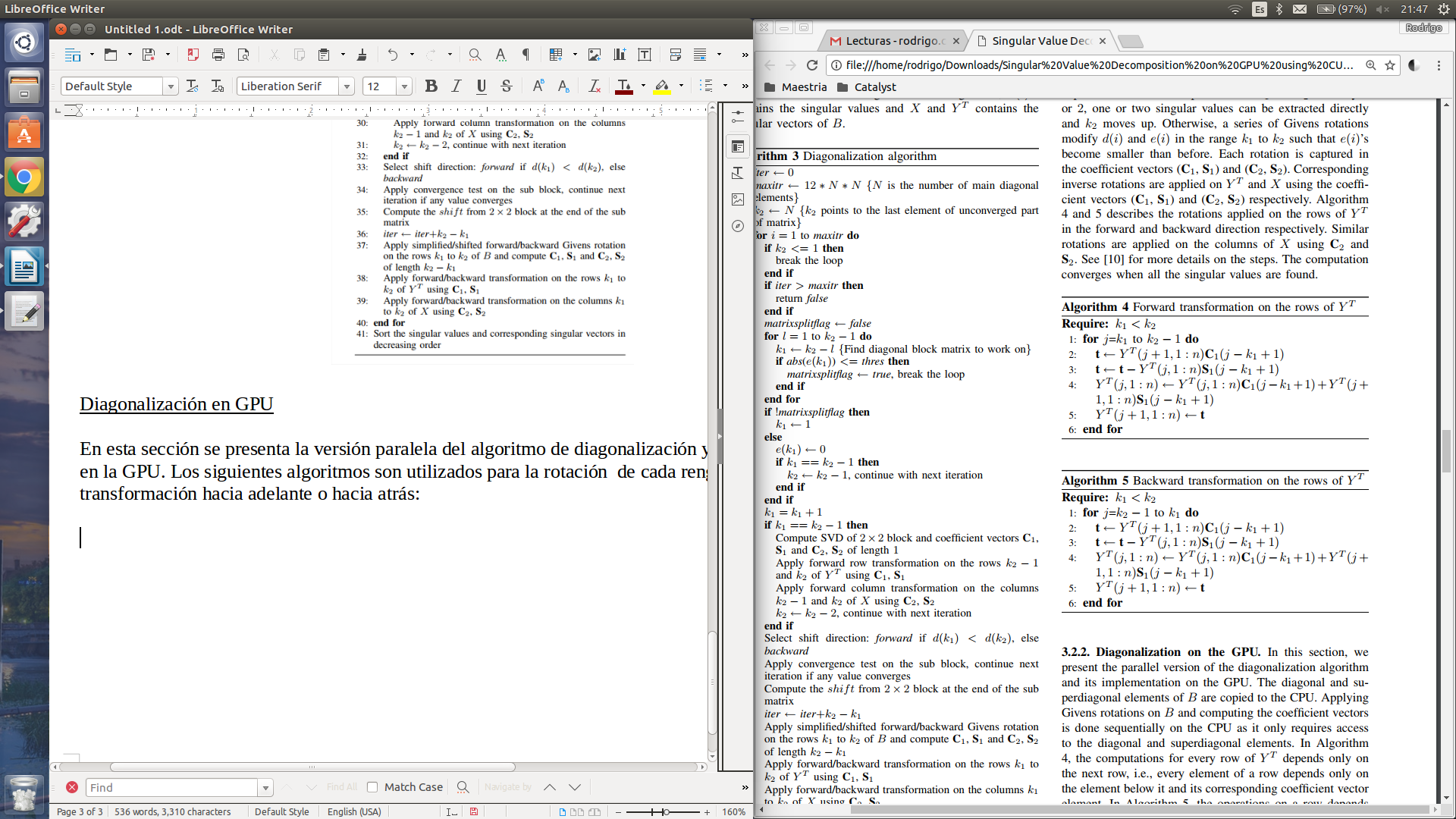
Σ = XTBY

Donde Σ es una matriz diagonal y X y Y son matrices unitarias ortogonales. A continuación se presenta el procedimiento de diagonalización.



Diagonalización en GPU

En esta sección se presenta la versión paralela del algoritmo de diagonalización y su implementación en la GPU. Los siguientes algoritmos son utilizados para la rotación de cada renglón YT, ya sea de transformación hacia adelante o hacia atrás:



El algoritmo divide cada línea de la matriz en bloques. Cada thread opera en un elemento de la fila. Esta división de líneas en bloques y formar bucles puede ser hecho eficientemente con una arquitectura CUDA debido a que cada thread realiza cómputo independiente. Los datos requeridos de cada bloque, son guardados en una memoria compartida y las operaciones pueden ser realizadas eficientemente en un multiprocesador.

Debido a que las transformaciones de columna son similares a las transformaciones de renglón, utilizamos transformación de renglón del kernel en las filas de XT en lugar de las columnas de X. Todos los threads de un bloque son usados para copiar los sectores de la memoria global a la memoria compartida que requiere que los vectores estén acomodados al bloque múltiple más cercano.

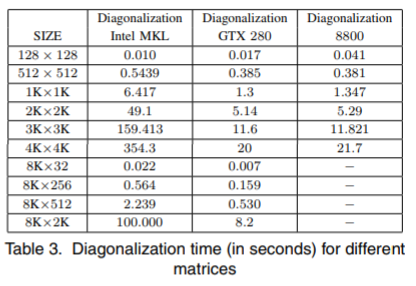
SVD Completo

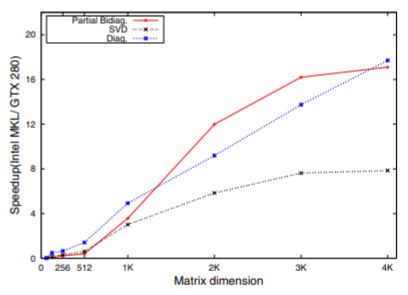
Se realizan dos multiplicaciones matriz-matriz al final para para calcular las matrices ortogonales U = QX and VT = (P Y )T. Se utilizan las rutinas de multiplicación matricial CUBLAS. Las matrices ortogonales son después copiadas al CPU.

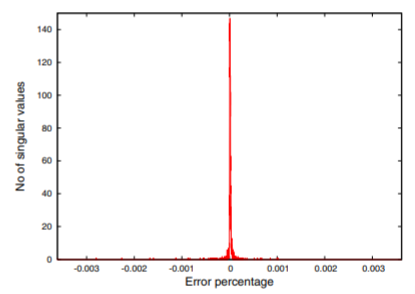
Resultados

En esta sección se analiza el desempeño del algoritmo con implementación CPU optimizada de SVD en MATLAB e Intel MKL 10.0.4 LAPACK.

Se generaron 10 matrices densas aleatorias de números de precisión singular para cada tamaño. El algoritmo de SVD fue ejecutado 10 veces. Para evitar una muestra mala o buena se realizó un promedio de matrices aleatorias para cada tamaño. A continuación, se pueden observar algunas gráficas y tablas para visualizar los resultados.







Conclusión

En este paper se presentó la implementación de SVD en GPUs. El algoritmo explota el paralelismo en la arquitectura GPU y alcanza un desempeño computacional alto. La bidagonalización de la matriz se realiza completamente en GPU usando la librería de CUBLAS optimizada para alcanzar el más alto desempeño. Se utilizó una implementación para diagonalización que divide el cómputo entre el CPU y el GPU. El error derivado de las limitaciones de los números de precisión única fue menor al 0.001% en matrices aleatorias con las que se experimentaron.