Tema 2-4. Programación Numérica con GPUs (II). Librerías

Métodos Numéricos para la Computación

Grado en Ingeniería Informática. Mención Computación
Escuela de Ingeniería Informática
Universidad de Las Palmas de Gran Canaria

Contenidos

- Tipos de librerías numéricas en GPU
- Descripción de MAGMA
- Utilización y ejemplos de cuBLAS
- Características de cuSPARSE. Formatos de codificación de matrices escasas.
- Utilización y ejemplos de cuSOLVER

NOTA de advertencia

Los ejemplos contenidos en este y anteriores temas se ha ejecutado en una GPU con muy pocos núcleos. Los cual, refleja una relación de prestaciones muy poco favorable hacia la GPU si se compara con la CPU (concretamente un i7.

El objetivo es ilustrar el proceso de programación y uso de las librerías, que serán software compatible con las diferentes versiones de CUDA. La velocidad de ejecución dependerá notablemente del número de multiprocesadores SM que se dispongan en cada caso.

Por ello el cronometraje debe valorarse en cuanto a la metodología de medición. Los resultados dependerán del modelo de GPU.

GPUs Utilizadas

Propiedad	GeForce GT540M	GeForce GTX 580
Compute Capability	2.1	2.0
Memoria Global	2GB	1.5GB
Clock	1.34GHz	1.54Ghz
Multiprocesadores	2	16
Cores	2*48=96	16*32=512
Ancho de Banda	6.3GB/s	6.2GB/s

Librerías Numéricas en GPUs

- Librerías básicas NVIDIA: cuBLAS y cuSolver, cuFFT, cuSPARSE, cuRAND

- Librerías avanzadas NVIDIA:

  cuDNN (Redes neuronales, Deep Learning)

  cuBLAS-XT (BLAS con varias GPUs simultáneas)

- Librería MAGMA.

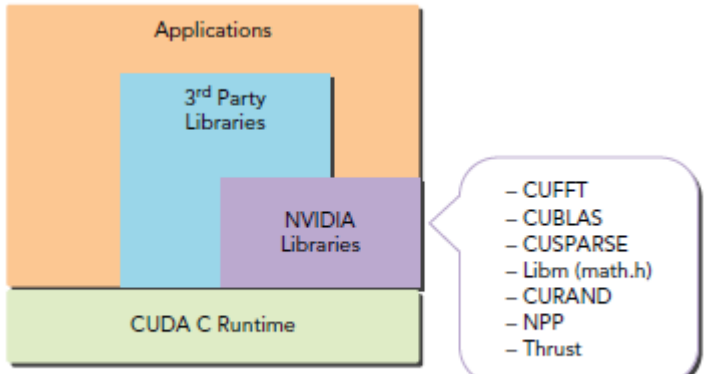


FIGURE 8-1

Librería MAGMA

- Matrix Algebra on GPU and Multicore Achitectures.
- Open Software: http://icl.cs.utk.edu/magma/index.html
- Patrocinado por: los Departamentos de Energía y Defensa de los EEUU, Intel, AMD, NVIDIA, Mathworks, …
- Incluye versiones optimizadas para la mayoría de funciones de BLAS y LAPACK.
- Se puede utilizar en CPUs con Multicores y GPUs
- Disponible para CUDA, OpenCL e Intel Xeon Phi (un procesador vectorial especializado)

Propiedades de MAGMA

MAGMA tiene algunas características destacables

1. La distinción entre CPU y GPU se diluye, dado que los procedimientos finales para el usuario permiten la computación híbrida (mezclando uso de CPU y GPU) interior y encapsulada.
2. Puede utilizar diversas GPUs simultáneamente en el computo de una misma función. La gestión y coordinación de las diversas GPUs queda encapsulada interiormente en las funciones MAGMA .

Suffix	Example	Description
none	magma_dgetrf	hybrid CPU/GPU routine where the matrix is initially in CPU host memory.
_m	magma_dgetrf_m	hybrid CPU/multiple-GPU routine where the matrix is initially in CPU host memory.
_gpu	magma_dgetrf_gpu	hybrid CPU/GPU routine where the matrix is initially in GPU device memory.
_mgpu	magma_dgetrf_mgpu	hybrid CPU/multiple-GPU routine where the matrix is distributed across multiple GPUs' device memories.

La filosofía de MAGMA es que puedas utilizar la totalidad de recursos computacionales presentes en un sistema (diversos núcleos y una o varias GPUs) de la forma más transparente al uso final.

Instalación de MAGMA

- MAGMA se distribuye en código fuente, no existe distribución binaria. Lógicamente, pues cada sistema tiene configuraciones diferentes y MAGMA trata de usar todo los recursos presentes.

- El usuario debe construir la versión binaria que se ejecuta en su sistema, o bien en uno idéntico (clon) en cuanto CPU+GPU+Librerías instaladas y versiones.

- La generación de la versión binaria requiere un proceso previo de configuración que comprueba el sistema, el software instalado y sus versiones. La cantidad de cosas que puede ir mal se bastante grande.

- EL proceso requiere la existencia de un compilador Fortran y las librerías BLAS y LAPACK, por ejemplo disponibles con MKL.

- La forma más practica para instalar MAGMA es en Linux donde es más sencillo controlar todos estos factores.

cuBLAS

Implementa los cálculos de la librería BLAS en la memoria de la GPU. El proceso para desarrollar una aplicación que utilice cuBLAS es el siguiente:

1. Inicializar la librería y contesto de cublas mediante un handle.
2. Reservar espacio en la GPU para almacenar las matrices y vectores objeto de cálculo. Se pueden usar la funciones de Runtime.
3. Copiar los datos requeridos desde la memoria central hasta la memoria reservada en la GPU.
4. Lanzar una secuencia de funciones BLAS de cálculo matricial que actúan dentro de la GPU. Para optimizar el proceso se recomienda no recuperar los resultados hacia la CPU, sino encadenar el máximo número de funciones BLAS con los datos y resultados intermedios ya presentes en la GPU.
5. Recuperar los resultados finales hacia la memoria central.

Como en casi todas las operaciones con GPUs, el uso de cuBLAS para operaciones individuales puede no ser una buena opción, por los costes de transferencia de datos. El uso eficiente reside en que los datos ya residan en la GPU, los resultados vayan a residir en la memoria de la GPU para otras operaciones.

El encadenamiento de operaciones sucesivas en la GPU es la opción que amortiza el sobre-coste de las transferencia.

NOTA: cuBLAS utiliza únicamente la codificación column-major de Fortran. Mientras las versiones en CPU como CBLAS y MKL permiten especificar el uso alternativo entre row-major o column-major.

En el uso de matrices que estén codificadas en row-major deberemos indicar la opción de transposición.

Las constantes de alpha y beta, de diversos niveles de BLAS deben pasarse por referencia a la variable, no por valor. De forma idéntica al uso de funciones Frontran dentro de C/C++

Uso de MKL y cuBLAS

El prototipo de uso de CUDA en Visual Studio está específicamente configurado para el uso de la librería Runtime pero no para las librerías especializadas como cuBLAS y restantes. Tampoco está configurado para usar MKL. Para usarlas es necesario configurar manualmente Visual Studio siguiendo los siguientes pasos para modo x64 y Release:

1. Añadir directorio include de MKL: (Raiz MKL)\include
Propiedades de configuración → Directorios de VC++ → Directorios de archivos de inclusión

2. Añadir directorio de librerías de MKL: (Raiz MKL)\lib\intel64
Propiedades de configuración → Vinculador → General → Directorio de bibliotecas adicionales

3. Añadir librerías adicionales tanto de MKL como cuBLAS o similares.
Propiedades de configuración → Vinculador → Entrada → Dependencias adicionales

mkl_core.lib
mkl_sequential.lib
mkl_intel_lp64.lib
cublas.lib
cublas_device.lib

Patrón de uso de cuBLAS

Utilizar las siguientes cabeceras. Como no es necesario ejecutar Kernels, no se incluyen todas las cabeceras de CUDA, pero si la general cuda_runtime.h para utilizar las funciones relacionadas con la memoria

```
#include <cuda_runtime.h>
#include <cublas_v2.h>
```

Patrón general

1. Crear una variable de captura de error del tipo cublasStatus_t
2. Crear un handle para el manejo de funciones cuBLAS
3. Ejecutar diversa llamadas de funciones.
4. Destruir el handle

```
cublasStatus_t statusCublas;
cublasHandle_t handle;
statusCublas = cublasCreate(&handle);

// diversas llamadas a cuBLAS con el handle
statusCublas = cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, N, N, N, &alpha, dev_A, N, dev_B, N, &beta, dev_C, N);
// ...

statusCublas = cublasDestroy(handle);
```

Transferencia de Matrices

Las transferencia y resultados desde o hacia la GPU se puede realizar con la función de la librería Runtime

cudaMemcpy();

También puede realizarse con la función de la librería de cuBLAS

cublasSetVector(…);
cublasGetVector(…);
cublasSetMatrix(…);
cublasGetMatrix(…);

Aparentemente no se aprecian diferencias.

cuBLAS es no-bloqueante

Las llamadas de algoritmos a las funciones cuBLAS al igual que la mayoría de llamadas de ese tipo en las librerías de NVIDIA son no-bloqueantes, es decir, la llamada regresa a la ejecución en CPU sin esperar a que se haya completado la operación en la GPU.

Las transferencias síncronas como cudaMemcpy son bloqueantes, es decir, esperan hasta que las transferencia de memoria se haya completado y además esperan hasta que las llamadas a funciones de algoritmo precedentes (misma cola) se hayan ejecutado.

Cuando se enlazan diversa llamadas a funciones cuBLAS y y Memcpy, se incluyen todas ellas en la misma cola Stream de la GPU, por ello la secuencialidad se conserva. Pero no es posible medir tiempos en la CPU, salvo después de los cudaMemcopy bloquenates o mediante el uso de la rutina de bloqueo de espera cudaDeviceSynchronize().

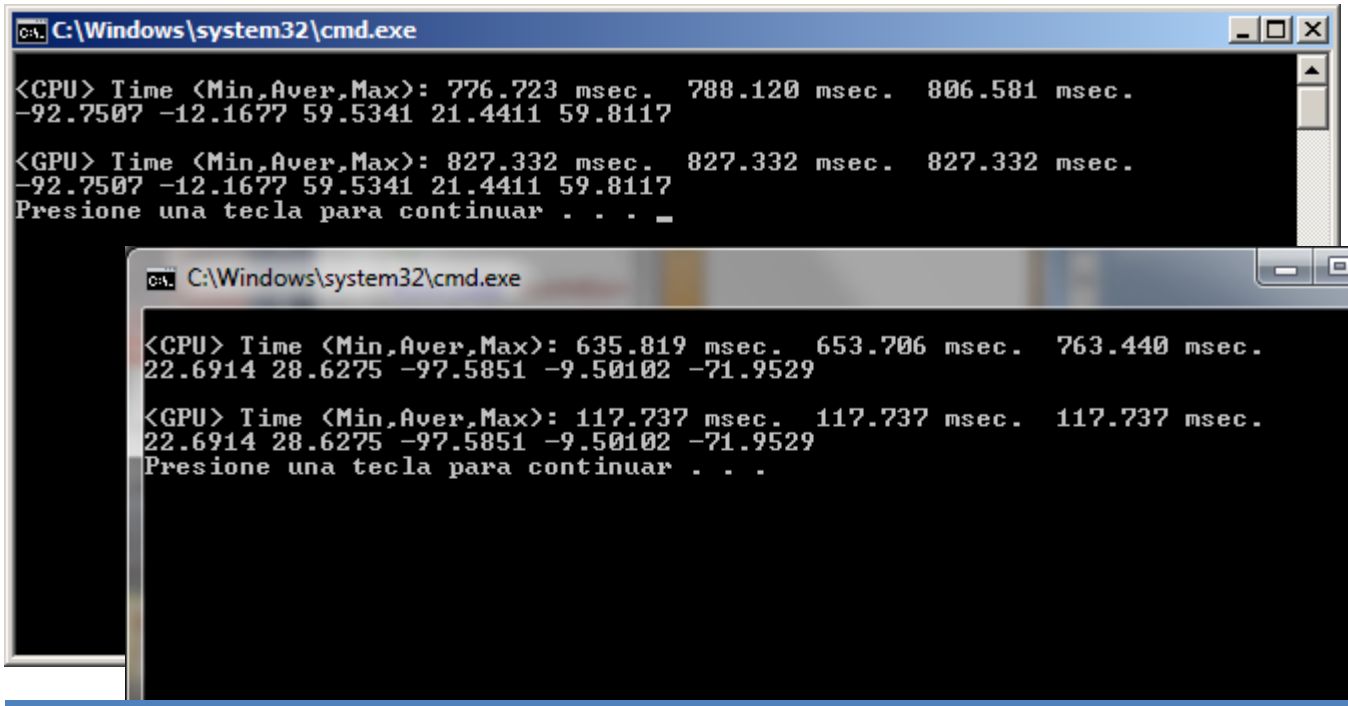
Ambos casos son correctos, pero en el de la derecha se pueden medir tiempos después de la sincronización.

cublas call(…); → inserta en el Stream
cublascall(….); → inserta en el Stream

cublascall(…);
cudaDeviceSynchronize();
cublascall(…);

Ejemplo cuBLAS(1)

Multiplicar dos matrices de 2Kx2K en CPU con BLAS de MKL y en cuBLAS



GT540M

GTX 580

No existe ganancia al comparar la ejecución en CPU en un i7 y en una GPU con tan solo 2 multiprocesadores SM(GT 540M). La posibilidad de la GPU reside en utilizar modelos con más multiprocesadores (GTX 580)

De las transferencias de datos nos olvidamos ¿?

Ejemplo de cuBLAS(2)

```
/*
Ejemplo de uso de cuBLAS a Nivel 3 con dgemm()
Comparamos con la versión de MKL en la CPU.

Juan Méndez para MNC, juan.mendez@ulpgc.es
*/
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <random>

#include <mkl.h>

#include <cuda_runtime.h>
#include <cublas_v2.h>

#include "eTimer.h"

#define N 2*1024

int main(int argc, char *argv[])
{
    double *host_A, *host_B, *host_C;
    double *dev_A, *dev_B, *dev_C;
    int sizematrix = N*N*sizeof(double);
    double alpha = 1.0, beta = 0.0;

    std::random_device gen;
    std::normal_distribution<double> dist(0.0, 1.0);

    host_A = (double*)mkl_malloc(sizematrix, 64);
    host_B = (double*)mkl_malloc(sizematrix, 64);
    host_C = (double*)mkl_malloc(sizematrix, 64);

    for (int y = 0; y < N; y++){
        for (int x = 0; x < N; x++){
            host_A[y*N + x] = dist(gen);
            host_B[y*N + x] = dist(gen);
        }
    }
```

Cabeceras y declaración de variables generales.

Matrices de 2Kx2K, se debe probar en el Laboratorio y reducir o aumentar según la cantidad de memoria instalada en el sistema y en la GPU

cuBLAS requiere el paso por referencia a las constantes.

Ejemplo de cuBLAS(3)

$$C = \alpha A^{[N,T]} B^{[N,T]} + \beta C$$

Parte de la CPU usando CBLAS contenida en MKL

```
eTimer *Tcpu = new eTimer();
eTimer *Tgpu = new eTimer();

// Un nuevo uso de eTimer
// dado que dgemm en CPU no destruye sus datos y beta=0, podemos repetir el calculo muchas veces
// para obtener minimos, promedio y máximo.
for (int i = 0; i < 10; i++){
    Tcpu->start();
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, alpha, host_A, N, host_B, N, beta, host_C, N);
    Tcpu->stop();
}
Tcpu->report("CPU");

for (int x = 0; x < 5; x++)printf("%g ",host_C[x]);
printf("\n");
memset(host_C, 0, sizematrix);
```

Casos de prueba comparativa

Ejemplo de cuBLAS(4)

```
// codigo de la GPU ----------------------------------------------------------------------------
cudaError_t cudaStatus;
cublasStatus_t cublasStatus;
cublasHandle_t handle;

cudaStatus = cudaGetDevice(0);
cublasStatus = cublasCreate(&handle);

// reserva espacio en GPU
cudaStatus = cudaMalloc((void**)&dev_A, sizematrix);
cudaStatus = cudaMalloc((void**)&dev_B, sizematrix);
cudaStatus = cudaMalloc((void**)&dev_C, sizematrix);

// transfiere datos al estilo cuda o cublas
//cudaStatus = cudaMemcpy(dev_A, host_A, sizematrix, cudaMemcpyHostToDevice);
//cudaStatus = cudaMemcpy(dev_B, host_B, sizematrix, cudaMemcpyHostToDevice);
cublasStatus = cublasSetMatrix(N, N, sizeof(double), host_A, N, dev_A, N);
cublasStatus = cublasSetMatrix(N, N, sizeof(double), host_B, N, dev_B, N);

Tgpu->start();
// cublas dgemm. CuBLAS asume codificación Fortran, luego se debe usar Transpose
cublasStatus = cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, N, N, N, &alpha, dev_A, N, dev_B, N, &beta, dev_C, N);
cudaStatus = cudaDeviceSynchronize();
Tgpu->stop();
Tgpu->report("GPU");

// recupera resultados al estilo cuda o cublas
//cudaStatus = cudaMemcpy(host_C, dev_C, sizematrix, cudaMemcpyDeviceToHost);
cublasStatus = cublasGetMatrix(N, N, sizeof(double), dev_C, N, host_C, N);

// pero los datos recuperados estan traspuestos
for (int x = 0; x < 5; x++)printf("%g ", host_C[x*N]);
printf("\n");

// liberación de recursos
cudaStatus = cudaFree(dev_A);
cudaStatus = cudaFree(dev_B);
cudaStatus = cudaFree(dev_C);
cublasStatus = cublasDestroy(handle);

cudaStatus = cudaDeviceReset(); // lo ultimo referente a la GPU
// fin de la GPU -------------------------------------------------------------------------------------------------
```

Parte de la GPU. cuBLAS es tipo-Fortran. Se pasan en row-major y se deben multiplicar con T

Los resultados en dev_C, transferidos a host_C, son los traspuestos de la matriz obtenidos con CBLAS, por eso los casos de prueba debe imprimirse por columnas.

Ejemplo de cuBLAS(5)

Liberación de los recursos utilizados en la CPU

```
    mkl_free(host_A);
    mkl_free(host_B);
    mkl_free(host_C);

    delete Tcpu;
    delete Tgpu;

    return 0;
}
```

cuSPARSE

cuSPARSE es una librería que implementa la funciones de BLAS pero sobre matrices que han sido codificadas como matrices escasas.

Expondremos brevemente los diversos formatos de codificación de matrices escasas utilizados por cuSPARSE.

La forma más sencilla de codificación escasa es la de vectores. La forma de codificación **densa** de un vector es la de posiciones sucesivas de memoria que contienen todos los elementos del mismo.

$$\text{dense} : \underbrace{1.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{3.}$$

$$\text{sparse} : \begin{cases} \text{size} : 7 \\ \text{indices} : \underbrace{0}\ \underbrace{6} \\ \text{values} : \underbrace{1.}\ \underbrace{3.} \end{cases}$$

La forma **escasa** consiste en codificar únicamente los elementos no nulos con varios datos:

1. Longitud del vector.
2. Vector de índices con la posición de los elementos no nulos.
3. Vector de los valores no nulos referidos a cada índice.

Matrices escasas/sparse

BCCS	Block Compressed Column Storage format
BCRS	Block Compressed Row Storage format
BND	Linpack Banded format
BSR	Block Sparse Row format
CCS	Compressed Column Storage format
COO	Coordinate format
CRS	Compressed Row Storage format
CSC	Compressed Sparse Column format
CSR	Compressed Sparse Row format
DIA	Diagonal format
DNS	Dense format
ELL	Ellpack-Itpack generalized diagonal format
JAD	Jagged Diagonal format
LNK	Linked list storage format
MSR	Modified Compressed Sparse Row format
NSK	Nonsymmetric Skyline format
SSK	Symmetric Skyline format
SSS	Symmetric Sparse Skyline format
USS	Unsymmetric Sparse Skyline format
VBR	Variable Block Row format

La forma de codificación densa de una matriz es la de un vector de valores consecutivos de todos los elementos. La interpretación de la matriz debe hacerse en base a las dimensiones y del criterio de codificación: por filas o por columnas.

Existen diversos formatos para codificar matrices escasas:
http://www.bu.edu/pasi/files/2011/01/NathanBell1-10-1000.pdf

http://www.cs.colostate.edu/~mcrob/toolbox/c++/sparseMatrix/sparse_matrix_compression.html

Métodos usados en cuBLAS

MATRIX DATA FORMAT	OPTIMAL USE CASE
Dense	Dense input data with very few non-zero entries. A dense data format may result in better access locality in the case of dense input data.
Coordinate (COO)	A simple and general sparse matrix format, which will represent sparse matrices more efficiently in terms of space than a dense data format, so long as less than one-third of the input matrix is non-zero.
Compressed Sparse Row (CSR)	Rather than keeping a single integer for every non-zero value to store its row coordinate, CSR keeps a single integer for every row as an offset to that row's value and column data. CSR, therefore, is the space efficient relative to COO when each row contains more than one non-zero entry. However, it does not allow O(1) lookup of a given values row.
Compressed Sparse Column (CSC)	CSC is the same as CSR except in two ways. First, the values of the input matrix are stored in column-major order. Second, the column coordinates are compressed rather than the row coordinates. CSC would be more space-efficient than CSR for input data sets with dense columns.
Ellpack-Itpack (ELL)	ELL works by compacting every row in a matrix down to only its non-zero entries. To retain column information for each value, a separate matrix stores column coordinates for each value. Programmers do not have direct access to ELL formatted matrices, but they are used in storing HYB-formatted matrices.
Hybrid (HYB)	A HYB-formatted matrix stores a regular partition of the matrix in ELL and an irregular partition in COO. This hybrid formatting scheme serves to optimize access patterns on the GPU for matrices that have partitions characterized by different sparsity.
Block Compressed Sparse Row (BSR)	BSR uses the same algorithm as CSR, but rather than storing scalar types as values, it supports storing a two-dimensional block of scalar values. The BSR format (and very similar BSRX format) optimizes the subdivision of a large matrix between multiple CUDA thread blocks.
Extended BSR (BSRX)	BSRX is identical to BSR but uses a slightly different technique for marking the locations of unique two-dimensional blocks in memory.

Determinados tipos de formatos se adaptan mejor a ciertos problemas

Formato de coordenadas (COO)

Formato de coordenadas. Coordinate Format (COO). Es el más simple:

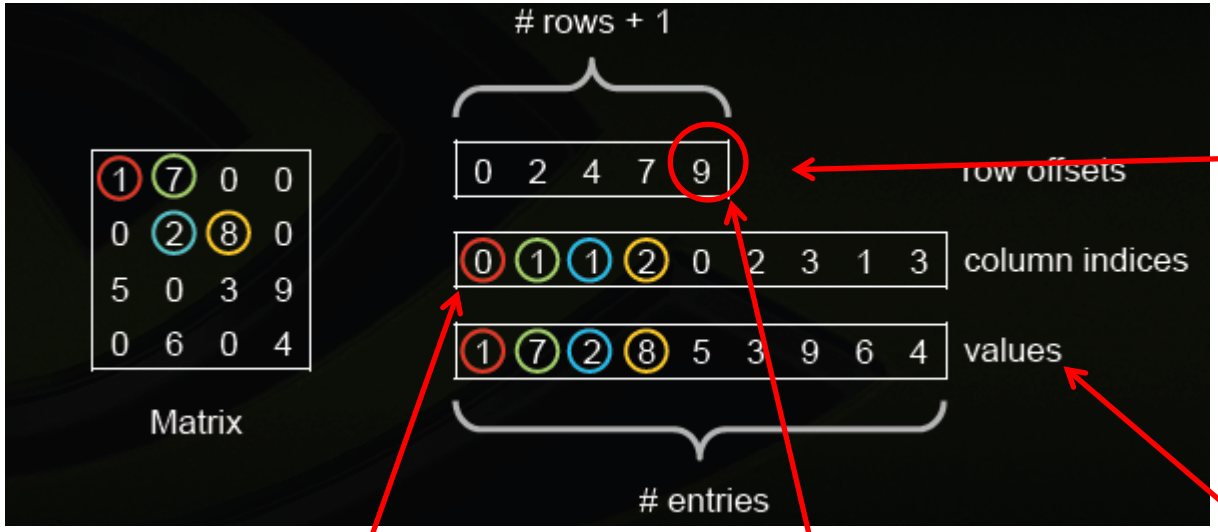
Se codifican los elementos no nulos y sendos vectores de los índices de filas y columnas de los elementos no nulos.

3.3.2. Coordinate Format (COO)

The `m×n` sparse matrix `A` is represented in COO format by the following parameters.

`nnz`	(integer)	The number of nonzero elements in the matrix.
`cooValA`	(pointer)	Points to the data array of length `nnz` that holds all nonzero values of `A` in row-major format.
`cooRowIndA`	(pointer)	Points to the integer array of length `nnz` that contains the row indices of the corresponding elements in array `cooValA`.
`cooColIndA`	(pointer)	Points to the integer array of length `nnz` that contains the column indices of the corresponding elements in array `cooValA`.

Formato de filas comprimidas (CSR)



Indica en el vector de valores o columnas, donde se produce el cambio de fila, salvo el último elemento

3.3.3. Compressed Sparse Row Format (CSR)

The only way the CSR differs from the COO format is that the array containing the row indices is compressed in CSR format. The `m×n` sparse matrix `A` is represented in CSR format by the following parameters.

`nnz`	(integer)	The number of nonzero elements in the matrix.
`csrValA`	(pointer)	Points to the data array of length `nnz` that holds all nonzero values of A in row-major format.
`csrRowPtrA`	(pointer)	Points to the integer array of length `m+1` that holds indices into the arrays `csrColIndA` and `csrValA`. The first `m` entries of this array contain the indices of the first nonzero element in the `i` th row for `i=i,...,m`, while the last entry contains `nnz+csrRowPtrA(0)`. In general, `csrRowPtrA(0)` is `0` or `1` for zero- and one-based indexing, respectively.
`csrColIndA`	(pointer)	Points to the integer array of length `nnz` that contains the column indices of the corresponding elements in array `csrValA`.

Formato de columnas comprimidas (CSC)

Es un dual a CSR en el sentido de que la matriz se codifica por columnas, se codifica como en COO los elementos y las filas y posteriormente se codifican los índice de cambios de las columnas referidos al vectos de valores de filas. Justamente el dual o complementario de CSR

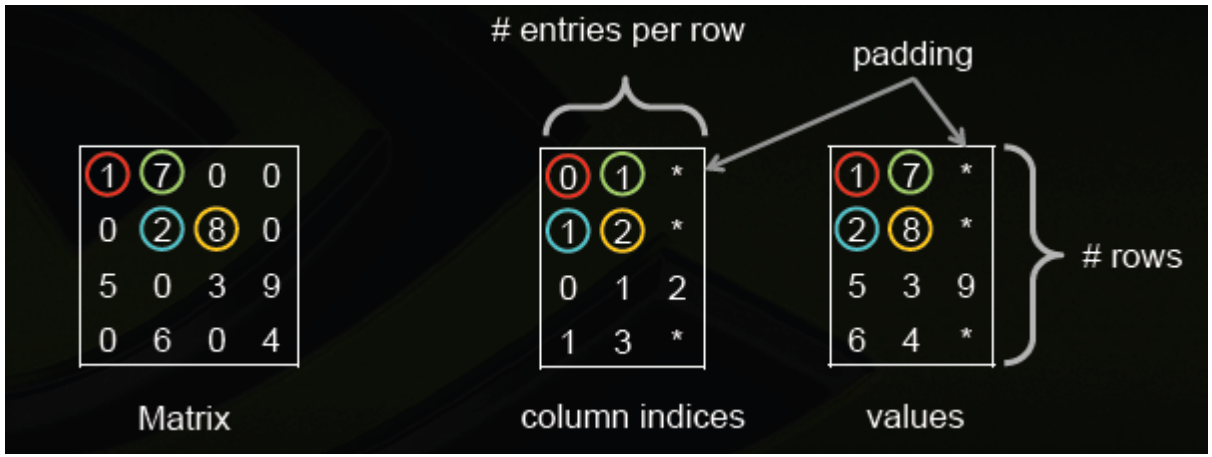
3.3.4. Compressed Sparse Column Format (CSC)

The CSC format is different from the COO format in two ways: the matrix is stored in column-major format, and the array containing the column indices is compressed in CSC format. The `mxn` matrix `A` is represented in CSC format by the following parameters.

`nnz`	(integer)	The number of nonzero elements in the matrix.
`cscValA`	(pointer)	Points to the data array of length `nnz` that holds all nonzero values of `A` in column-major format.
`cscRowIndA`	(pointer)	Points to the integer array of length `nnz` that contains the row indices of the corresponding elements in array `cscValA`.
`cscColPtrA`	(pointer)	Points to the integer array of length `n+1` that holds indices into the arrays `cscRowIndA` and `cscValA`. The first `n` entries of this array contain the indices of the first nonzero element in the `i`th row for `i=i,...,n`, while the last entry contains `nnz+cscColPtrA(0)`. In general, `cscColPtrA(0)` is `0` or `1` for zero- and one-based indexing, respectively.

ELLPACK (ELL)

Es una codificación escasa de los vectores filas. Pos tanto genera dos matrices, una de valores dentro de las filas y otras de índices dentro de cada fila de esos valores.

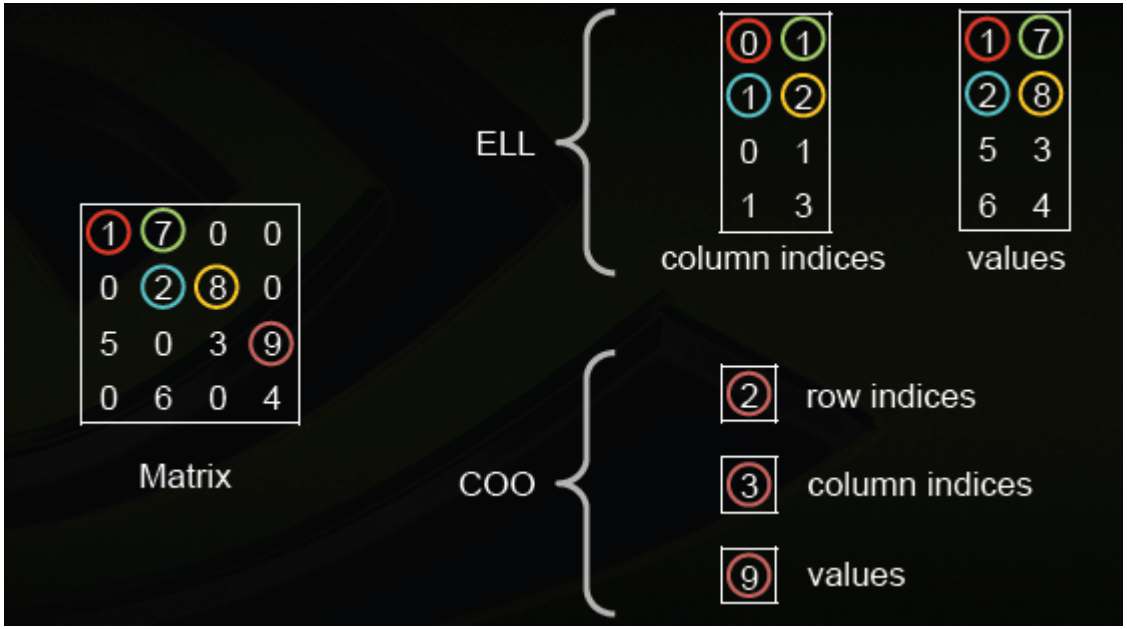


Formato de diagonales (DIA)

Es el formato utilizado en LAPACK para matrices en bandas diagonales. El número de columnas de la matriz codificada es el mismo que de la matriz original, mientras que el número de filas es el del número de diagonales no nulas

Formato híbrido (HYB) ELL+COO

Consiste en codificar mediante el formato de coordenadas (COO) las dos matrices del formato ELL



Ruta de conversión de formatos

Las funciones para convertir formatos son las siguientes:

TABLE 8-4: Supported Sparse Matrix Format Conversions

		SOURCE FORMAT					
		DENSE	COO	CSR	CSC	HYB	BSR
DESTINATION FORMAT	DENSE			csr2dense	csc2dense	hyb2dense	
	COO			csr2coo			
	CSR	dense2csr	coo2csr			hyb2csr	
	CSC	dense2csc		csr2csc		hyb2csc	bsr2csr
	HYB	dense2hyb		csr2hyb	csc2hyb		
	BSR			csr2bsr			

El formato que convierte a todos los demás es CSR, de forma que la ruta para convertir, por ejemplo matrices densa a cualquier tipos sería la siguiente, salvo que exista forma directa de conversión:

Densa → CSR → Todos los tipos

Pasos en cuSPARSE

1. Definir una variable de status del tipo cusparseStatus_t para todas las funciones de la librería, un handle y un descriptor de matriz
2. Crear un handle con
   status = cusparseCreate(&handle);
3. Crear un descriptor de matriz con
   status = cusparseCreateMatDescr(&descr);
4. Definir los atributos de la matriz con (son por defecto):
   status = cusparseSetMatType(descr,CUSPARSE_MATRIX_TYPE_GENERAL);
   status = cusparseSetMatIndexBase(descr,CUSPARSE_INDEX_BASE_ZERO);
5. Convertir matrices de tipo coordenadas, COO, a tipo CSR
   status = cusparseXcoo2csr(….);
6. Realizar un preanálisis de las dimensiones que tendrá la matriz de resultados con:
   status = cusparseXcsrgrmmNnz(…)
7. Realizar una operación, por ejemplo multiplicación matricial:
   status = csparseDcsrgemm(…);
8. Convertir la matriz resultado a tipo coordenadas, que es más fácilmente interpretable.
   status = cusparseXcst2coo(…)

Utilización de cuSPARSE

Las matrices escasas en formato de coordenadas no se crean explícitamente en GPU, sino que se transfieren sus tres vectores.

Se pueden crear matrices escasas en formato csr desde matrices densa en el propio host o en el dispositivo, utilizando una combinación de pre-análisis+ conversión con las funciones:

cusparseDnnz(…); determina las dimensiones de los vectores de datos escasos.
cusparseDdense2csr(…); realiza la conversión

En la fase de configuración de Visual Studio, seleccionar x64, dado que las librerias especiales de CUDA, cuSPARSE es una, no se distribuyen en versión 32 bits en la release 7.0 instalada.

En esta fase de configuración, añadir manualmente las librerías:
cublas.lib
cublas_device.lib
cusparse.lib

cuSOLVER

La librería cuSOLVER es la contrapartida a LAPACK en GPU. Pero no implementa toda LAPACK, solo el subconjunto de mayor uso. Esta librería utiliza, para sus procesos internos, las ya referidas cuBLAS y cuSPARSE.

Tiene dos versione:

cuSolverDN para matrices densas. Se implementan las funciones de solución de sistemas lineales y de determinación de valores singulares.
cuSolverSP para matrices escasas. Se implementan los siguientes problemas que usan formato CSR

cuSolverSP API

routine	data format	operation	output format	based on
csrlsvlu	csr	linear solver (ls)	vector (v)	LU (lu) with partial pivoting
csrlsvqr	csr	linear solver (ls)	vector (v)	QR factorization (qr)
csrlsvchol	csr	linear solver (ls)	vector (v)	Cholesky factorization (chol)
csrlsqvqr	csr	least-square solver (lsq)	vector (v)	QR factorization (qr)
csreigvsi	csr	eigenvalue solver (eig)	vector (v)	shift-inverse
csreigs	csr	number of eigenvalues in a box (eigs)		
csrsymrcm	csr	Symmetric Reverse Cuthill-McKee (symrcm)		

Problemas densos

Resuelve dos tipos de problemas: la solución de sistemas lineales, y la obtención de valores singulares. Se emplean las factorizaciones o descomposiciones matriciales LU, QR, Cholesky, SVD y LDLT

Es necesario crean previamente un handle para utilizarlos posteriormente en las funciones

```
cusolverStatus_t
cusolverDnCreate(cusolverDnHandle_t *handle);
```

```
cusolverDnDestroy(cusolverDnHandle_t handle);
```

El patrón de nombre de las funciones es:

`cusolverDn<t><operation>`

where `<t>` can be `S`, `D`, `C`, `Z`, or `X`, corresponding to the data types `float`, `double`, `cuComplex`, `cuDoubleComplex`, and the generic type, respectively. `<operation>` can be Cholesky factorization (`potrf`), LU with partial pivoting (`getrf`), QR factorization (`geqrf`) and Bunch-Kaufman factorization (`sytrf`).

NOTA: Todas las matrices densa se suponen almacenadas en formato de column-major

Factorización LU. getrf

Se necesita un espacio de trabajo para el algoritmo. Este espacio de trabajo debe ser determinado y reservado previamente en lamemoria de la GPU. Los pasos a seguir son:

1. Determinar el tamaño del espacio de trabajo con la función cusolverDnDgetrf_bufferSize(…)
2. Reservar espacio en la GPU para el espacio de trabajo con cudaMalloc(…).
3. Obtener la descomposición con la función: cusolverDnDgetrf(…)

Recordemos que realiza la descomposición LU de la matriz A con pivotamiento P

$$PA = LU$$

Sistema Lineal. getrs

Resuelve el sistema lineal siguiente, donde la matriz A puede utilizarse traspuesta o no-traspuesta y B puede tener múltiples columnas, para diversos sistemas simultáneos.

$$A^{[N/T]}X = B$$

La función que lo resuelve utiliza la descomposición LU obtenida anteriormente. La función es:

cusolverDnDgetrs(…)

Descomposición de Valores Singulares (SVD). gesvd

Obtiene la descomposición:

$$A = USV^T$$

Donde U y V son matrices unitarias y S contiene los valores

Realiza la operación en dos etapas, como la descomposición LU . Una primera determina el tamaño de un espacio de trabajo auxiliar. Se debe reservar ese espacio y luego utilizarlo en la obtención de la descomposición.

cusolverDnDgesvd_bufferSize(…);
cusolverDnDgesvd(…)

Uso de MKL y cuSOLVER

Configurar manualmente Visual Studio siguiendo los siguientes pasos para modo x64 y Release:

1. Añadir directorio include de MKL: (Raiz MKL)\include
Propiedades de configuración → Directorios de VC++ → Directorios de archivos de inclusión

2. Añadir directorio de librerías de MKL: (Raiz MKL)\lib\intel64
Propiedades de configuración → Vinculador → General → Directorio de bibliotecas adicionales

3. Añadir librerías adicionales tanto de MKL como cuBLAS o similares.
Propiedades de configuración → Vinculador → Entrada → Dependencias adicionales

mkl_core.lib
mkl_sequential.lib
mkl_intel_lp64.lib
cusolver.lib

Ejemplo Sistema Lineal Denso(1)

Resolveremos un sistema lineal denso, con valores aleatorios pero con una diagonal destacada para reducir la posibilidad de que sea singular.

Dados algunos problemas de malfuncionamiento, se ha optado por tamaños más reducidos de la matriz en este caso 3*512.

La resolución se realizará con MKL en la CPU con dgesv()

Generar un nuevo proyecto CUDA y eliminar el contenido. Renombrar el fichero a uno adecuado a la aplicación, pero con extensión .cpp, dado que se usan librerías y no kernels.

Configurar el uso de MKL en versión x64

Sistema Lineal Denso(2)

```
/*
Resolver sun sistema lineal con LAPACK en CPU y cuSOLVER en GPU
*/

#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <random>
#include <chrono>
#include <assert.h>
#include <mkl.h>

#include <cuda_runtime.h>
#include <cusolverDn.h>

#include "eTimer.h"

#define N 3*512

int main(int argc, char *argv[]){

    std::random_device gen; // mas aleatorio
    std::normal_distribution<double> dist(0.0, 1.0);

    double *A1, *A2, *B1, *B2;
    double inicio, fin;

    A1 = (double*)mkl_malloc(N*N*sizeof(double), 64); // LAPACK destruye las matrices originales
    A2 = (double*)mkl_malloc(N*N*sizeof(double), 64); // debenos tener copias
    B1 = (double*)mkl_malloc(N*sizeof(double), 64);
    B2 = (double*)mkl_malloc(N*sizeof(double), 64);

    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            A1[i*N + j] = dist(gen);
            A2[i*N + j] = A1[i*N + j];
        }
        B1[i] = dist(gen);
        B2[i] = B1[i]; // una copia
        A1[i*N + i] += 10.0;
        A2[i*N + i] = A1[i*N + i];
    }
```

Sistema Lineal Denso(3)

Solución en la CPU con LAPACK de MKL

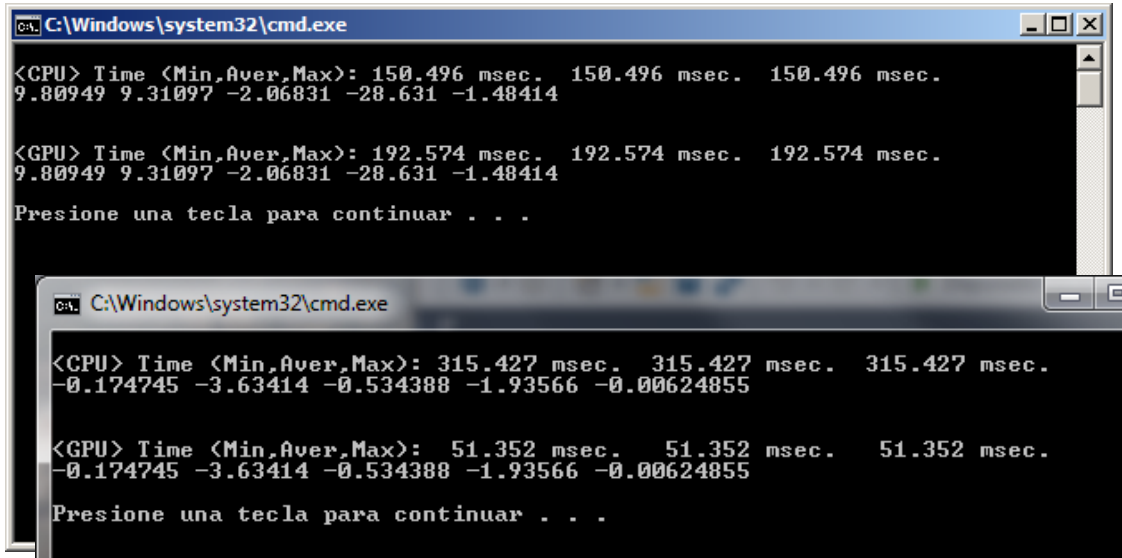
```
int info;
int *ipiv = (int *)mkl_malloc(N*sizeof(double), 64);

eTimer *Tcpu = new eTimer();
eTimer *Tgpu = new eTimer();

Tcpu->start();
info = LAPACKE_dgesv(LAPACK_ROW_MAJOR, N, 1, A1, N, ipiv, B1, 1); // en B1 la solución en A1 la dec. LU
Tcpu->stop();
Tcpu->report("CPU");

for(int i=0;i<5;i++) printf("%g ", B1[i]);
printf("\n\n");
```

```
// Codigo de la GPU ------------------------------------------------------------------------
cudaError_t cudaStatus;
cusolverStatus_t cusolverStatus;
cusolverDnHandle_t handle;

double *dev_A, *dev_B, *Work;
int *dev_pivot, *dev_info, Lwork;

cudaStatus = cudaSetDevice(0);
cusolverStatus = cusolverDnCreate(&handle);

cudaStatus = cudaMalloc((void**)&dev_A, N*N*sizeof(double));
cudaStatus = cudaMalloc((void**)&dev_B, N*sizeof(double));
cudaStatus = cudaMalloc((void**)&dev_pivot, N*sizeof(int));
cudaStatus = cudaMalloc((void**)&dev_info, 1*sizeof(int));
cudaStatus = cudaMemcpy(dev_A, A2, N*N*sizeof(double), cudaMemcpyHostToDevice);
cudaStatus = cudaMemcpy(dev_B, B2, N*sizeof(double), cudaMemcpyHostToDevice);

Tgpu->start();
// primero, calculamos el espacio de trabajo auxiliar
cusolverStatus = cusolverDnDgetrf_bufferSize(handle, N, N, dev_A, N, &Lwork);
// segundo, reservamos espacio de trabajo auxiliar
cudaStatus = cudaMalloc((void**)&Work, Lwork*sizeof(double));
// tercero, obtenemos la descomposicion LU en A misma
cusolverStatus = cusolverDnDgetrf(handle,N,N,dev_A,N,Work,dev_pivot,dev_info);
// cuarto, resolvemos el sistema
cusolverStatus = cusolverDnDgetrs(handle, CUBLAS_OP_T, N, 1, dev_A, N, dev_pivot, dev_B, N, dev_info);
cudaStatus = cudaDeviceSynchronize();
Tgpu->stop();
Tgpu->report("GPU");

cudaStatus = cudaMemcpy(B2,dev_B,N*sizeof(double), cudaMemcpyDeviceToHost);

for (int i = 0; i<5; i++) printf("%g ", B2[i]);
printf("\n\n");

cudaStatus = cudaFree(dev_A);
cudaStatus = cudaFree(dev_B);
cudaStatus = cudaFree(dev_pivot);
cudaStatus = cudaFree(dev_info);
cudaStatus = cudaFree(Work);

cusolverStatus = cusolverDnDestroy(handle);
cudaStatus = cudaDeviceReset();
// fin de la GPU ---------------------------------------------------------------------------
```

Hemos usado Memcpy al estilo cuda

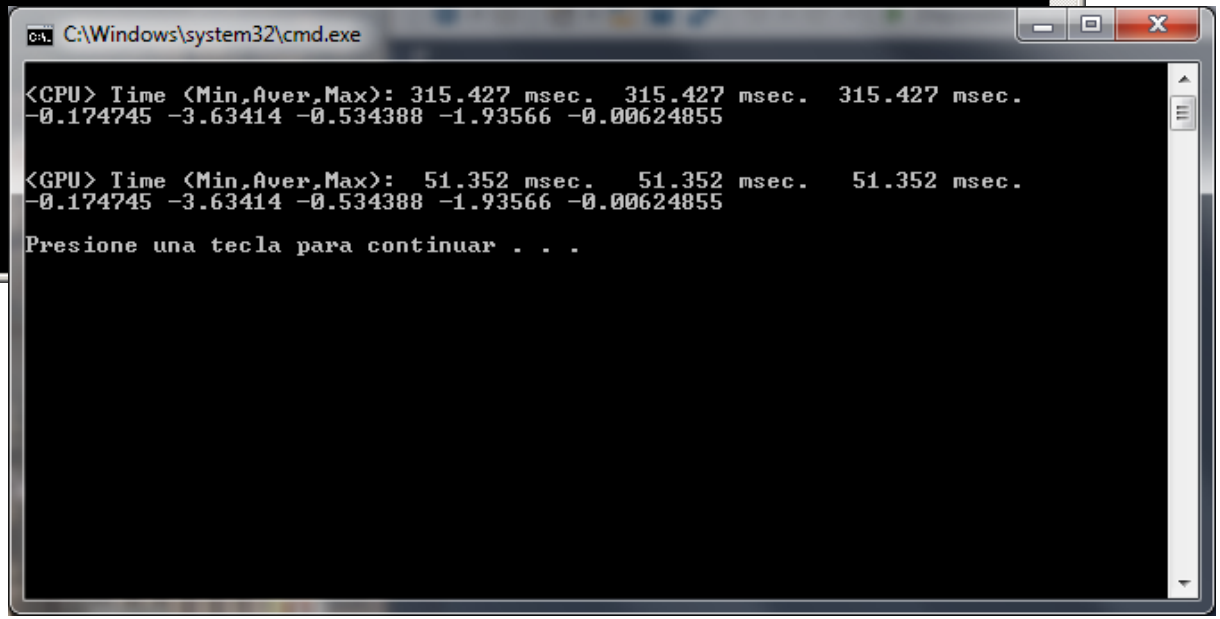
Resolución en la GPU, El proceso en laborioso dado que hay que concatenar varias tareas sucesivas.

Sistema Lineal Denso. Resultados



GT 540M

```
    mkl_free(A1);
    mkl_free(A2);
    mkl_free(B1);
    mkl_free(B2);
    mkl_free(ipiv);

    delete Tcpu;
    delete Tgpu;

    return 0;
}
```



GTX 580

Se vuelve a evidenciar la influencia del numero de cores en la velocidad de la resolución.

Bibliografía

CUDA C Programming Guide, NVIDIA Corporation.
https://docs.nvidia.com/cuda/cuda-c-programming-guide/#abstract

MAGMA User Guide. http://icl.cs.utk.edu/projectsfiles/magma/doxygen/

J. Cheng, M. Grossman, T. MacKercher, Profesional CUDA C Programming, Wrox/Wiley, 2014

cuBLAS User Guide: http://docs.nvidia.com/cuda/cublas/#axzz3ovtzHUfy

cuSPARSE User Guide: http://docs.nvidia.com/cuda/cusparse/index.html#axzz3ovtzHUfy

cuSOLVER User Guide: http://docs.nvidia.com/cuda/cusolver/index.html#axzz3ovtzHUfy

Comparación MAGMA vs cuBLAS:
https://developer.nvidia.com/sites/default/files/akamai/cuda/files/Misc/mygpu.pdf