VII. Código Bloque I: SOLVER

Este bloque se encarga de la resolución en CUDA del sistema AX=B donde recordemos A es una matriz Real, simétrica, cuadrada, dispersa que puede tener centenares de miles e incluso millones de elementos.

Al ser la primera tarea de codificación se decidió optar por usar bibliotecas y código previamente desarrollado y puesto a disposición de los desarrolladores por parte de NVIDIA.

La estrategia adoptada para resolver este apartado en CUDA es la representada en la figura 1.

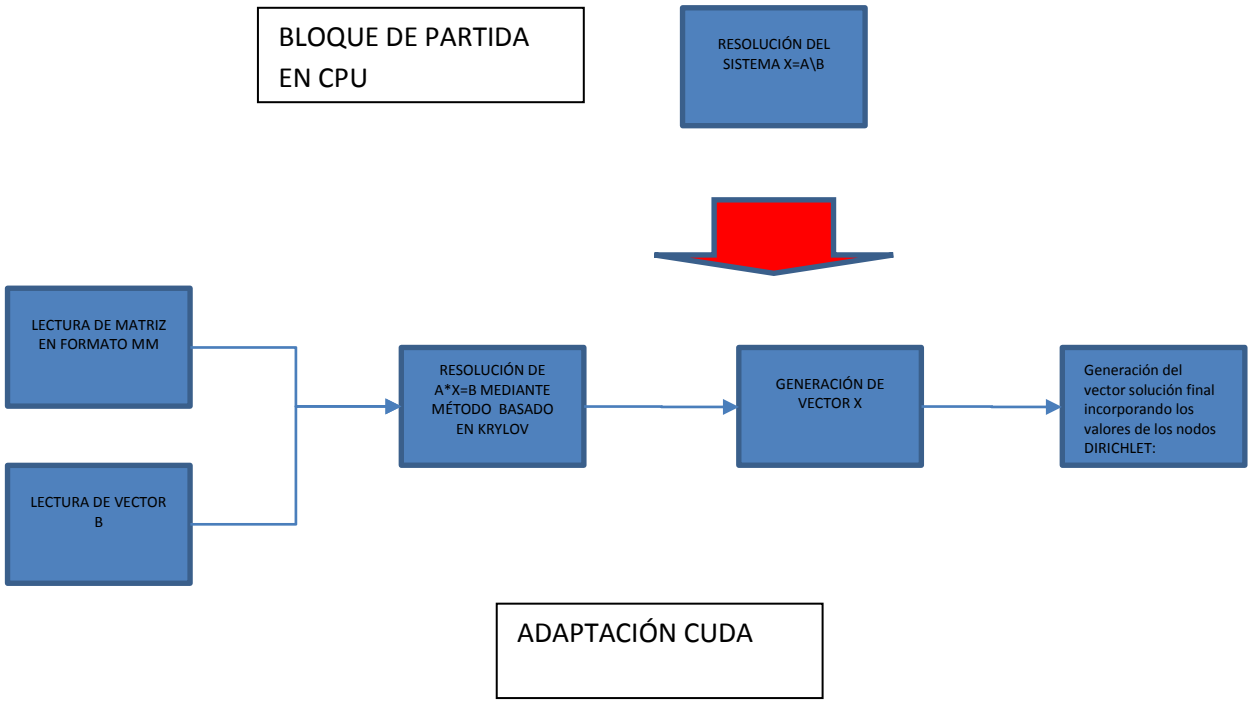

Fig 1: Diagrama de flujo de la adaptación objeto de este bloque.

En la planificación del proyecto se decidió empezar la resolución con el bloque del Solver debido a sus particularidades. Es un bloque complejo donde se plantean por un lado la elección de un algoritmo para la resolución y por otro su implementación.

1. REORDENAMIENTO DEL SISTEMA:

El sistema de partida está formado por un vector de términos independientes "B" de tamaño Número de Nodos del mallado y una matriz de rigidez "A" Simétrica, Real, Definida con diagonal principal no nula.

Sin embargo este sistema no es el que se debe resolver, ya que se le deben incorporar las condiciones de Dirichlet, en concreto las soluciones correspondientes a los Nodos Dirichlet.

Esto se realiza mediante el reordenamiento del sistema.

Para ilustrarlo se expondrá un caso simple de 4 ecuaciones con cuatro condiciones de contorno.

Sea el sistema de 4 ecuaciones lineales con cuatro incógnitas siguiente:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 = b_1$$

$$a_{12}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 = b_2$$

$$a_{13}x_1 + a_{23}x_2 + a_{33}x_3 + a_{34}x_4 = b_3$$

$$a_{14}x_1 + a_{42}x_2 + a_{34}x_3 + a_{44}x_4 = b_4$$

O bien en forma matricial:

$$\begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{12} & a_{22} & a_{23} & a_{24} \\ a_{13} & a_{23} & a_{33} & a_{34} \\ a_{13} & a_{24} & a_{34} & a_{44} \end{matrix} \times \begin{matrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{matrix} = \begin{matrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{matrix}$$

Donde se definen como condiciones de contorno Dirichlet x_2 y x_4 y como condiciones de contorno Neumann b_1 y b_3. Por tanto las incógnitas serán x_1 y x_3 y b_2 y b_4.

Por lo tanto, debemos reordenar el sistema para pasar todas las incógnitas a la izquierda y mantener a la derecha un vector independiente que contenga únicamente valores conocidos.

$$a_{11}x_1 + a_{13}x_3 = b_1 - a_{12}x_2 - a_{14}x_4$$

$$a_{12}x_1 - b_2 + a_{23}x_3 = -a_{22}x_2 - a_{24}x_4$$

$$a_{13}x_1 + a_{33}x_3 = b_3 - a_{23}x_2 - a_{34}x_4$$

$$a_{14}x_1 + a_{34}x_3 - b_4 = -a_{24}x_2 - a_{44}x_4$$

Este sistema puede resolverse de dos maneras distintas:

- Manteniendo el tamaño del sistema:

Para ello construimos una nueva matriz y un nuevo vector directamente a partir del sistema de ecuaciones reordenado y que en este caso serían de la forma:

$$\begin{matrix} a_{11} & 0 & a_{13} & 0 \\ a_{12} & 1 & a_{23} & 0 \\ a_{13} & 0 & a_{33} & 0 \\ a_{14} & 0 & a_{34} & 1 \end{matrix} \times \begin{matrix} x_1 \\ -b_2 \\ x_3 \\ -b_4 \end{matrix} = \begin{matrix} b_1 - a_{12}x_2 - a_{14}x_4 \\ -a_{22}x_2 - a_{24}x_4 \\ b_3 - a_{23}x_2 - a_{34}x_4 \\ -a_{24}x_2 - a_{44}x_4 \end{matrix}$$

Como se observa, este sistema es del mismo tamaño que el original, sin embargo la matriz ha dejado de ser simétrica. El método de los subespacios de Krylov indicado para este caso es el bi-gradiente conjugado estabilizado.

En este método se hace necesario insertar en el vector ofrecido por el algoritmo los valores relativos a las posiciones Dirichlet conocidas para obtener el vector solución como queda ilustrado mediante el siguiente esquema:

$$\begin{matrix} x_1 \\ -b_2 \\ x_3 \\ -b_4 \end{matrix} \rightarrow \begin{matrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{matrix}$$

Como se observa, se realiza una sustitución de los valores Neumann del nodo por sus valores Dirichlet.

- Reduciendo el tamaño del sistema:

Para ello construimos una nueva matriz y un nuevo vector eliminando las filas y columnas correspondientes a los nodos Dirichlet cuyo valor conocemos, de tal modo que en este caso el sistema sería de la forma:

$$\begin{matrix} a_{11} & a_{13} \\ a_{13} & a_{33} \end{matrix} \times \begin{matrix} x_1 \\ x_3 \end{matrix} = \begin{matrix} b_1 - a_{12}x_2 - a_{14}x_4 \\ b_3 - a_{23}x_2 - a_{34}x_4 \end{matrix}$$

En este caso se observa que el tamaño de la matriz se reduce tanto como el número total de nodos Dirichlet. Este sistema ofrece la gran ventaja de ser simétrico en el caso que tratamos, y por lo tanto el método iterativo de los subespacios de Krylov que le es aplicable es el del Gradiente Conjugado.

Sin embargo una vez obtenido el vector solución, este debe ser expandido incorporándole los valores correspondientes a los nodos Dirichlet conocidos, recuperando su tamaño original que es igual al número de nodos total del mallado. En este caso se debe tener especial cuidado para insertar los valores en las posiciones originales, como se muestra el siguiente esquema:

$$\begin{matrix} x_1 \\ x_3 \end{matrix} \rightarrow \begin{matrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{matrix}$$

REORDENAMIENTO DEL VECTOR b:

El reordenamiento del vector de términos independientes se realiza mediante la operación: B-AxU, donde B es el vector de términos independientes construido con las condiciones de Neumann, A es la matriz de rigidez del sistema y U es el vector que contiene los valores correspondientes a los nodos Dirichlet. A los nodos No Dirichlet se les asigna el valor Nulo.

REORDENAMIENTO DE LA MATRIZ A:

Se logra anulando las columnas correspondientes a los nodos Dirichlet y colocando "1" en la posición diagonal correspondiente.

Para localizar las columnas indicadas y reemplazar los valores se usó un Kernel que permitía acelerar el proceso, que en programación secuencial correspondería a un bucle anidado. Al anular las posiciones se las eliminaba antes de generar un archivo que contenía la matriz en formato Matrix Market.

REDUCCIÓN Y EXPANSIÓN DE LA MATRIZ A Y DEL VECTOR B:

Para la operación de reducción se procedió de la misma manera que en el punto anterior, eliminando también las posiciones correspondientes de la diagonal. Para el proceso especular de la expansión se usó un código basado en índices de compensación y en un vector que contenía de manera ordenada todos los nodos Dirichlet sin repetición. Lo cual permitía insertar los valor en el vector solución de manera ordenada.

2. IMPLEMENTACIÓN DEL SOLVER:

En cuanto a la implementación y debido a que era la primera aproximación al problema en CUDA, se optó por aprovechar las bibliotecas y códigos puestos a disposición para los desarrolladores por NVIDIA. En concreto se aprovechó este punto para desarrollar y comparar dos soluciones distintas basadas en la misma tecnología CUDA:

1. Solución basada en las bibliotecas del proyecto CUSP que implementa un solver para matrices dispersas basado en métodos de alto nivel BLAS.

2. Solución basada en CUBLAS. Se desarrolló usando esta biblioteca un Solver Iterativo. Sin embargo para la parte de producto de Matriz Dispersa por Vector se optó por usar la solución SpMV propuesta por NVIDIA, en lugar de la rutina de producto Matriz-Vector proporcionada por la biblioteca, porque esta rutina es indicada para matrices densas y no dispersas.

a. SOLVER BASADO EN CUSP

CUSP es una biblioteca para álgebra linear dispersa y computación gráfica sobre CUDA. CUSP proporciona una interfaz para manipular matrices dispersas y resolver sistemas lineales.

Es un proyecto abierto, en pleno desarrollo y su código es libre bajo la licencia "Apache".

El único requisito para usarla es instalar los siguientes paquetes disponibles de forma gratuita: Cusp v0.1, Thrust v1.2 y CUDA 3.0.

Thrust es una biblioteca CUDA de algoritmos paralelos con una interfaz similar a la Standard Template Library (STL) de C++. Thrust proporciona una biblioteca para la programación de GPU que permite hacer el proceso de desarrollo más sencillo. También es un proyecto abierto y su código es libre bajo la licencia "Apache".

Como Thrust y Cusp son bibliotecas "template" (plantillas) no se construye nada. Sólo hace falta descargar las últimas versiones descomprimirlas en los directorios recomendados:

- /usr/local/cuda/include/ en Linux o Mac.
- C:\CUDA\include\ en un sistema Windows.

El proyecto CUSP está diseñado para ser usado con matrices dispersas por ello tiene una amplia gama de herramientas para su manipulación; de hecho CUSP soporta los siguientes formatos de almacenamiento de matrices dispersas:

- COO - Coordinate
- CSR - Compressed Sparse Row
- DIA - Diagonal
- ELL - ELLPACK/ITPACK
- HYB - Hybrid

Que son los formatos más usuales que nos vamos a encontrar.

Cuando se manejan matrices es importante conocer las ventajas y desventajas de cada sistema. En general los formatos DIA y ELL son los más eficiente para el cálculo de productos Matriz Dispersa por Vector, y por ello los más rápidos para la resolución de sistemas lineales usando métodos iterativos. Los formatos COO y CSR son más flexibles y más sencillos de manejar. El formato HYB es una mezcla de ELL que le confiere rapidez y de COO que le confiere flexibilidad, por ello es una buena elección.

CUSP permite la Conversión entre formatos soportados de tal modo que se puede leer una matriz en un formato, convertirla en otro para operar y presentar la salida en un tercero, por ejemplo.

CUSP permite la transferencia de datos entre el host (CPU) y device (GPU) y la conversión de formatos de matrices. Por ejemplo, la orden:

```
cusp::csr_matrix<int,float,cusp::host_memory> A(5,8,12);
```

Reserva una matriz CSR en el host de 5 filas, 8 columnas y 12 valores no-nulos. Tras inicializar las entradas podemos copiarla de golpe al device mediante una sola orden:

```
cusp::csr_matrix<int,float,cusp::device_memory> B = A;
```

O bien convertirla en otro formato medinate la orden:

```
cusp::hyb_matrix<int,float,cusp::device_memory> C = A;
```

antes de copiarla al device.

CUSP proporciona soporte para la lectura y escritura de matrices en formato Matrix Market.

La versión de CUSP instalada proporcionaba las siguientes herramientas:

Algoritmos

- cusp::multiply – Multiplicación dispersa de MatrizxVector. CUSP proporciona una implementación de SpMV para todos los formatos de matrices dispersas tanto para host como para device.

- cusp::transpose – Transpuesta de matrices.

Solvers iterativos:

- cusp::krylov::cg – Gradiente conjugado

- cusp::krylov::bicgstab Gradiente bi-conjugado estabilizado

Monitores

Determinan los criterios de convergencia de los solvers e informan acerca de las iteraciones. Los monitores disponibles son:

- cusp::default_monitor – Criterios de convergencia estándar.

- cusp::verbose_monitor – imprime a la salida la información de convergencia para cada iteración.

Precondicionadores

Son un método para mejorar la velocidad de convergencia en los solvers iterativos. El precondicionador debe ser de cálculo rápido y que se aproxime, lo máximo posible, a la inversa de la matriz. En la versión usada sólo se disponía del precodicionador diagonal:

- cusp::precond::diagonal – precondicionador diagonal que corresponde a la inversa de la matriz que se obtiene construyendo una matriz del mismo tamaño que de la de partida y almacenando exclusivamente los valores que forman la diagonal principal, anulando todos los demás.

Operadores definidos por el usuario

Para poder resolver sistemas del tipo A*x = b sin convertir "A" a uno de los formatos de almacenamiento; CUSP proporciona soporte para operadores definidos por el usuario que toman un vector x y calculan y = A * x.

- cusp::linear_operator - interface for user-defined linear operators

Utilidades

- cusp::gallery::* - rutinas para generar ejemplos de matrices
- cusp::blas::* - colección de rutinas BLAS nivel 1 usadas por los solvers iterativos.
- cusp::print_matrix – soporte de salida de contenidos de matrices.
- cusp::is_valid_matrix – comprueba la validez de formato de una matriz.

Formatos de Matrices

Además de los formatos de matrices dispersa CUSP soporta también formatos densos en concreto:

array1d – Array lineal unidimensional

array2d – Array bi-dimensional con formatos de columna o fila mayor.

Conversión de formato de matrices

Cusp proporciona rutinas para los siguientes cambios de formato.

Funciones de Conversión Host

- De COO ->
  - a CSR
  - a Dense
- De CSR ->
  - a COO
  - a DIA
  - a ELL
  - a HYB
  - a Dense
- De DIA ->
  - a CSR
- De ELL ->
  - a CSR

- De HYB ->
  - a CSR
- De Dense ->
  - a COO
  - a CSR

Funciones de Conversión Device

- De COO ->
  - a CSR
- De CSR ->
  - a COO

Gracias a estas herramientas se desarrolló un primer SOLVER para nuestro problema basado enteramente en CUSP. Este solver se llamó: Programa CUSP_version_3. En el Anexo I encontraremos el contenido de su archivo principal.

El fichero del programa contiene los siguientes documentos:

- A_matlab.mtx: archivo generado desde matlab con la orden mmwrite('A_matlab.mtx',A); contiene la matriz A almacenada en formato matriz market. Para ser generado se necesita la función mmwrite que se almacena en la carpeta del programa Matlab.
- temp.dat: archivo generado desde Matlab con la orden save temp.dat b –ascii contiene el vector B en formato ASCII. Es una función propia de MATLAB.
- AxB_sencillo_GPU_3.cu: Código basado en CUSP para ejecutar en GPU. El código está en C++. Aprovecha las propiedades de CUSP y THRUST para una ejecución optimizada sobre CUDA.
- GPU_codigo_AxB_3: ejecutable copilado con "nvcc".
- salida.dat: Contiene el resultado del vector solución X.
- norma.dat: Archivo que contiene el resultado de comprobación B-AX. Cuanto mejor sea X, más cerca estará del vector nulo.
- timer.h: Función auxiliar en C++ para la medida del tiempo de ejecución.

El código expuesto se basa en el uso del algoritmo de gradiente bi-conjugado estabilizado con precondicionador (la inversa de la diagonal principal de "A") con monitor fijado en un límite de 2000 iteraciones; y una Tolerancia = 1e-7.

b. SOLVER BASADO EN CUBLAS

Debido a la disponibilidad de la biblioteca CUBLAS descrita anteriormente y para comprobar los resultados de CUSP. Se ha desarrollado un nuevo código, esta vez basado en las bibliotecas optimizadas CUBLAS de NVIDIA. A diferencia de CUSP, ésta biblioteca no es un Solver. Y lo que se ha hecho es codificar el mismo algoritmo directamente en CUDA usando las bibliotecas CUBLAS que hacen transparentes los accesos y gestión de los elementos de la GPU.

En un principio se desarrolló un código CUBLAS exclusivo. Este código trataba la matriz "A" como matriz densa y no dispersa. El objetivo de este código era comprobar las limitaciones de un solver normal frente a uno adaptado para matrices dispersas. De hecho al ejecutar este con matrices de tamaño modesto empieza a dar problemas de falta de memoria; ya que almacena todos los elementos de la matriz en la memoria, incluyendo los valores nulos.

Se debe tener presente que CUBLAS contiene rutinas que tratan los elementos como densos y no como dispersos.

No obstante el algoritmo es válido y la implantación del método iterativo basado en Krylov es correcta. En un paso siguiente y observando que la matriz A interviene en determinados puntos concretos del código, en concreto en productos Matriz-Vector se optó por reemplazar en estos puntos la orden de producto de CUBLAS por el código SpMV de NVIDIA. Logrando así que el solver sea para matrices dispersas.

i. SpMV (Sparse Product Matrix by Vector) de NVIDIA:

Una de las herramientas puestas a disposición por NVIDIA para los desarrolladores es un código que realiza de manera eficiente el producto Metriz dispersa-Vector.

Esta operación es esencial en el desarrollo e implementación de muchas operaciones y rutinas; en especial de un SOLVER donde en algunos de los pasos del algoritmo se recurre a este tipo de productos.

La idea fundamental de partida es realizar el producto de manera eficiente. Por tanto se usa la matriz dispersa almacenada en un formato apropiado. Esto se traduce en una disminución de los requisitos de almacenamiento. Por otro lado, para la operación de producto sólo se realizan aquellos en los que intervienen elementos no-nulos ahorrando capacidad.

Esta diferencia se aprecia comparando códigos convencionales "C". Para matriz densa tendríamos para la operación A*B=C donde A es una matriz NxN que trataremos como matriz densa y "B" y "C" vectores de tamaño N que:

```
for (i=0; i<N; i++){

            for(j=0; j<N; j++){

                  C[i]+=A[i,j]*B[j];

                  }

}
```

Se observa que se recorren todas las filas y todas las columnas. "A" está almacenada completamente en la memoria y todos los posibles productos se llevan a cabo.

En cambio, para los mismos elementos de partida, tratando "A" como matriz dispersa almacenada en formato COO, mediante los arrays I, J y V y siendo NZ el número de elementos no nulos, tendríamos:

```
      for (i=0; i<NZ; i++){

            C[I[i]]+=V[i]*B[J[i]];

}
```

Observamos que este código es mucho más eficiente. De hecho contiene un bucle simple y no anidado como el anterior. Además el índice NZ es más corto que el N, ya que hemos eliminado todos los valore nulos. Finalmente esto se traduce en menos operaciones de producto.

La adaptación de éste código a Tecnología CUDA parte básicamente de las mismas ideas. Lo que se hace es un aprovechamiento mayor del almacenamiento, desarrollando ejecuciones en paralelo que optimizan los accesos de memoria según los warps.

El equipo de NVIDIA desarrolló de hecho kernels adaptados a los formatos de almacenamiento principales. Se pueden comparar los distintos rendimientos que de ello resultan y compararlos con ejecuciones en CPU y en GPU.

Se reproduce a continuación el código del kernel para el producto de una matriz CSR:

```
__global__ void spmv_csr_scalar_kernel(const IndexType num_rows,
                        const IndexType * Ap,
                        const IndexType * Aj,
                        const ValueType * Ax,
                        const ValueType * x,
                              ValueType * y)
{
    // row index
    const IndexType row = large_grid_thread_id();


    if(row < num_rows){
        ValueType sum = y[row];
        const IndexType row_start = Ap[row];
        const IndexType row_end   = Ap[row+1];
        for (IndexType jj = row_start; jj < row_end; jj++){
            sum += Ax[jj] * fetch_x<UseCache>(Aj[jj], x);
        }
        y[row] = sum;
    }
}
```

En esta versión denominada “scalar”, se observa que el núcleo principal es similar al expuesto más arriba para el caso de código secuencial C. La aceleración se logra asignando las filas a hilos.

Una versión más sofisticada se basa en asignar cada fila a un warp logrando una ejecución más eficiente y rápida. Es la versión “vector”, contenido en el archivo spmv_csr_vector_device.cu.

ii. Algoritmo basado en CUBLAS:

El código se divide en 4 funciones o bloques mayores:

1- El main para la gestión de los datos de entrada y salida. Son los archivos generados por MATLAB igual que en el caso anterior. Toma como argumento la matriz A en formato Matriz Market.

2- Rutina de identificación y lectura de datos desde una archivo en formato Matrix Market y su almacenamiento en una matriz.

1. Realizando llamadas a la función mmio.c facilitada por los desarrolladores del formato, se almacena "A" en una matriz NxN.

3- Almacenamiento de los datos de Btilde.dat correspondientes al vector de términos independientes.

4- La función solver propiamente dicho. "krylov" donde se codifica el algoritmo Gradiente bi-conjugado estabilizado (bi-cgstab) ó "Dcg" deonde se codifica el algoritmo del Gradiente Conjugado (CG); ambos con precisión "double". Este módulo devuelve el vector solución y la norma lograda.

El módulo de salida alimenta un bloque adicional donde se insertan los valores correspondientes a las condiciones de Dirichlet formando el vector solución.

En el caso del método bi-cgstab, se realiza una sustitución; mientras que para el método CG se realiza una expansión.

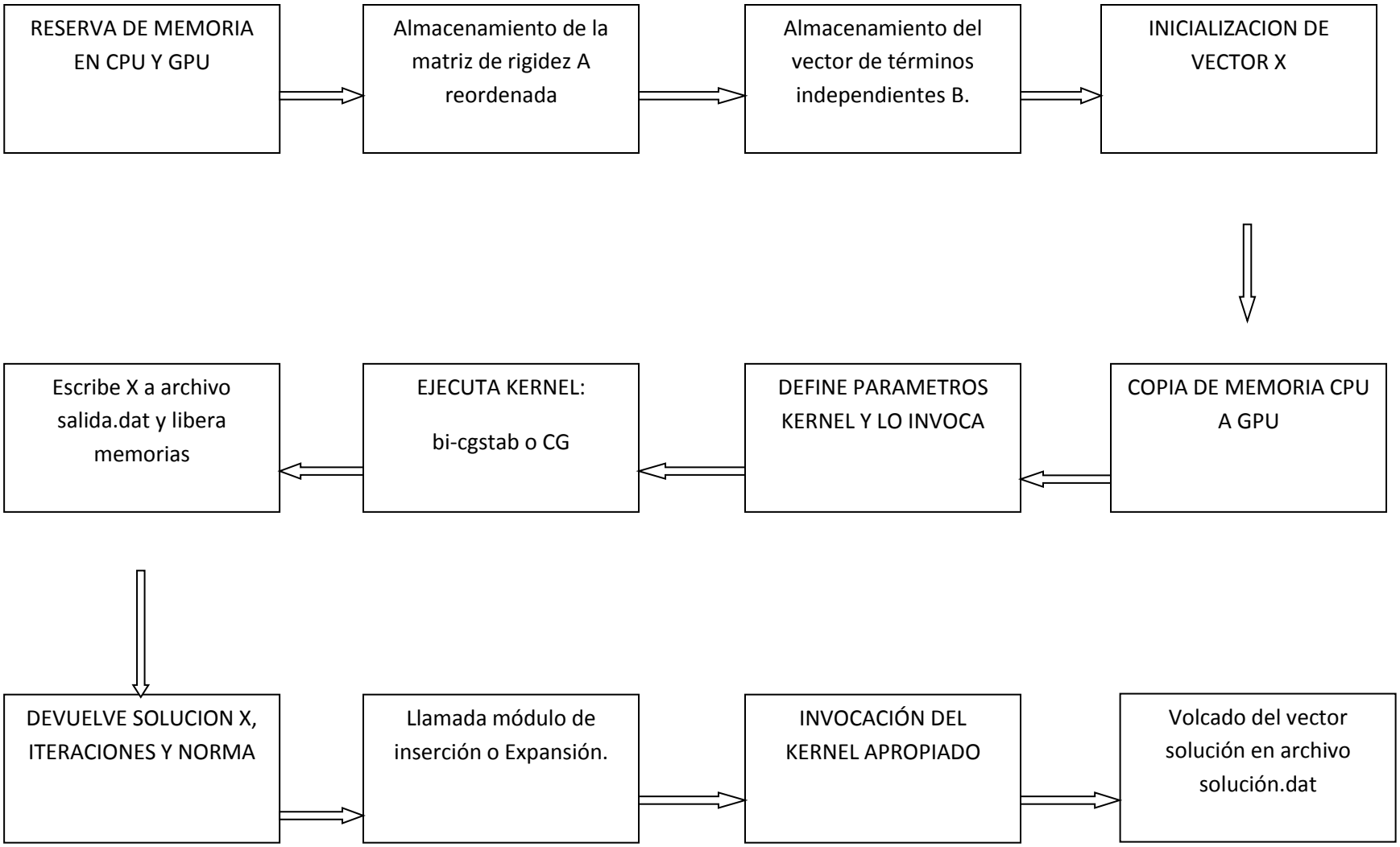

Fig 2: Diagrama de flujo del Bloque solver en la versión ensamblada.

3. CONCLUSIONES:

Para el código SOLVER se desarrollaron versiones con y sin precondicionador diagonal. También, se incluyeron unas líneas de código que permiten en el caso de no llegar a la precisión deseada ofrecer en la salida la mejor precisión lograda y el número de iteraciones necesarias para lograrla.

Se modificó mínimamente el código Matlab porque era la referencia con respecto a la cual se iban a comparar los demás.

- Se observó que la tarea de generar el precondicionador es muy exigente y lenta y retrasa mucho el programa. Una vez creado el precondicionador no se aprecia una gran diferencia en cuanto al tiempo de convergencia ni en precisión máxima que compense el tiempo invertido en su generación.

  Esta aparente paradoja se explica porque el Solver Precondicionado incorpora código adicional para generar el Precondicionador Diagonal, por otro lado los cálculos en cada iteración son también más complejos que en el Solver No-Precondicionado; ya que incrporan productos adicionales de Matriz-Vector. Como resultado se obtenía que el Solver Precondiconado, como es de esperar, convergía en menos iteraciones que el Solver No Precondicionado, pero las iteraciones resultaron ser más costosas y por tanto en su conjunto era más lento

- El almacenamiento de matrices en formato CSR ahorra mucha memoria y permite la ejecución con matrices grandes.

- Entre las soluciones CUSP y CUBLAS se aprecia que CUSP es más preciso. Aparentemente las rutinas CUBLAS son menos precisas que las rutinas BLAS. Sin embargo el diseño de CUBLAS es más flexible.

4. ANEXOS:

ANEXO I: Código Cusp

```
#include <iostream>
#include <cstdlib>
#include <fstream>
using namespace std;

#include <cstdlib>
#include <cusp/io/matrix_market.h>
#include <cusp/print.h>
#include <cusp/array2d.h>
#include <cusp/array1d.h>
#include <cusp/krylov/bicgstab.h>
#include <cusp/krylov/cg.h>
#include <cusp/transpose.h>
#include <timer.h>
#include <cusp/hyb_matrix.h>
#include <cusp/coo_matrix.h>
#include <cusp/csr_matrix.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/copy.h>
#include <thrust/transform.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/functional.h>
#include <iterator>
#include <algorithm>
#include <cusp/precond/diagonal.h>

//Los calculos los ejecutamos en la GPU
typedef cusp::device_memory MemorySpace;
```

```
      // Precisión float
      typedef float ValueType;


      int main()
      {
          // crea una matriz dispersa vacía en format CSR
          cusp::csr_matrix<int, ValueType, MemorySpace> A;


          // lee datos almacenados en matriz tipo Matrix Market
          cusp::io::read_matrix_market_file(A,
"A_matlabSBig.mtx");


          // Asigna memoria a x y lo inicializa a cero para
evitar problemas
          cusp::array1d<ValueType, MemorySpace> x(A.num_rows, 0);


      // codigo para leer b desde temp.dat y almacenarlo en array


            int i=0;
        float array[A.num_rows];
        int max_read = A.num_rows;
        int amountRead = 0;


        std::ifstream in("tempSBig.dat",std::ios::in
|std::ios::binary);
        if(!in)
        {
          std::cout<<"Error"<<std::endl;
          return 1;
        }
```

```
        //lectura y almacenamiento
        while(in>>array[amountRead]&& amountRead < max_read)
        {
          amountRead++;
        }

      in.close();
          // parametro el array b como archivo

          cusp::array1d<float, cusp::device_memory>
b(A.num_rows);
          thrust::copy(array, array + A.num_rows, b.begin());

          // define los parámetros de parada:
          int iteration_limit    = 2000;
          float relative_tolerance = 0.00024;
          cusp::verbose_monitor<ValueType> monitor(b,
iteration_limit, relative_tolerance);

          // precondicionador diagonal
          //cusp::identity_operator<ValueType, MemorySpace>
M(At.num_rows, At.num_rows);
       cusp::precond::diagonal<ValueType, MemorySpace> M(A);

      //Codigo para comparar tiempos de ejecución
          timer t;
          // Resuelve el sistema A * x = b usando el método del
bi-gradiente conjugado estabilizado
          cusp::krylov::bicgstab(A, x, b, monitor, M);

         float time = t.seconds_elapsed();

         cudaThreadSynchronize();
```

```
          if (monitor.converged())

              std::cout << "  Successfully converged";

          else

              std::cout << "  Failed to converge";

          std::cout << " after " << monitor.iteration_count() <<
" iterations." << std::endl;


          std::cout << "  Solver time " << time << " seconds ("
<< (1e3 * time / monitor.iteration_count()) << "ms per
iteration)" << std::endl;


      //alamaceno resultado en el archivo "salida"

       ofstream outdata;

        outdata.open("salida.dat");

         if( !outdata ) {

            cerr << "Error" << endl;

            exit(1);

         }


        for(i = 0; i < A.num_rows; i++)

            outdata << x[i] << endl;

         outdata.close();


      // comprueba norma del residuo

          cusp::array1d<float, MemorySpace> residual(A.num_rows,
0.0f);

          cusp::multiply(A, x, residual);

          cusp::blas::axpby(residual, b, residual, -1.0f, 1.0f);
```

```
  ofstream norma;
  outdata.open("norma.dat");
   if( !norma ) {
      cerr << "Error" << endl;
      exit(1);
   }

  for(i = 0; i < A.num_rows; i++)
      outdata << residual[i] << endl;
   norma.close();
    return 0;
}
```