Paralelización sobre GPU del Algoritmo de Eliminación de Gauss-Jordan para la Solución de Sistemas de Ecuaciones Lineales

Edgar Quispe Ccapacca
edgar.edqc@gmail.com

Escuela Profesional de Ingeniería de Sistemas UNSA, Perú
Avenida Venezuela, esquina con calle Paucarpata S/N Área de Ingenierías - Cercado
Arequipa - Perú

Resumen: *En este paper, se presenta dos modelos implementados en GPU del algoritmo de eliminación de Gauss-Jordan para la solución de sistemas de ecuaciones lineales de la forma AX=B, con la finalidad de acelerar el proceso de reducción de la matriz A, a su forma escalonada reducida, consiguiendo resultados de forma más rápida. Ambos modelos implementados muestran un buen desempeño. Los modelos implementados son probados con diferentes cantidades de soluciones a encontrar y son comparados con un modelo secuencial sobre CPU, mostrando claramente las ventajas del modelo implementado en GPU.*

Abstract: *In this paper presents two models implemented in GPU of the algorithm of Gauss-Jordan elimination to solve systems of linear equations of the form AX = B, in order to accelerate the process of reducing the matrix A, to the reduced echelon form, getting results faster. Both models implemented, achieve a good performance. The implemented models are tested with different amounts of solutions to find and are compared with a sequential model on CPU, showing clearly, the benefits of the model implemented on GPU*

Palabras clave: Eliminación hacia adelante, Eliminación hacia atrás, Gauss-Jordan, GPU, CUDA.

1. Introducción

Los sistemas de ecuaciones están presentes en diversos problemas de la ciencia e ingeniería, por ejemplo en aquellos problemas que emplean ecuaciones diferenciales parciales que modelan fenómenos físicos y que necesitan ser discretizadas para ser tratadas computacionalmente. En general, esos sistemas tienen un gran número de incógnitas. Debido al tamaño de esos sistemas es necesaria una gran velocidad de procesamiento, siendo adecuado el uso de computación de alto desempeño en la obtención de la solución de los mismos [Martinotto04].

En este paper, haremos uso del hardware gráfico para implementar el algoritmo de Gauss-Jordan para la solución de ecuaciones lineales de la forma **AX=B**, donde **A** es la matriz de coeficientes, **B** es el vector de coeficientes independientes y X es el vector de soluciones del sistema. El algoritmo realiza transformaciones elementales en las filas de la matriz aumentada **(A|B)** para llegar a convertir **A** en una matriz escalonada reducida obteniendo una matriz del tipo **(I|B')** donde **I** es una matriz identidad y **B'** es el vector de coeficientes independientes reducido. El vector solución es obtenido asignando cada elemento de **B'** a un vector solución **X**. El coste del algoritmo implementado sobre CPU es O(n3), el cual es extremadamente caro para sistemas de ecuaciones donde el número de soluciones a encontrar es muy grande.

El resto de este paper está organizado de la siguiente manera. La sección 2 muestra los trabajos previos. En la sección 3, se presenta una breve descripción del algoritmo de Gauss Jordan para la solución de sistemas de ecuaciones lineales. La sección 4 describe la programación de GPU usando CUDA. El modelo implementado es mostrado en la sección 5. Los Experimentos y Resultados obtenidos se encuentran en la sección 6. La Discusión de los Experimentos se muestra en la sección 7 y, finalmente en la sección 8, se presenta las conclusiones obtenidas en este trabajo.

2. Trabajos previos

Podemos mencionar los siguientes trabajos relevantes para el desarrollo del presente trabajo. En [Nesrin11], se presenta una implementación del algoritmo de Gauss-Jordan sobre GPU para la solución de sistemas de ecuaciones de circuitos lineales, se prueba sobre sistemas de 3x3 y 4x4. En [Buluç08], se utiliza factorización LU recursiva para realizar la eliminación gaussiana, la cual solo reduce la matriz en triangular superior, obteniendo la solución del sistema por sustituciones sucesivas y de forma secuencial. En [Yang11], se presenta una versión del algoritmo de eliminación de Gauss-Jordan en CUDA con pruebas realizadas con matrices de orden de hasta 4000.

En este trabajo, se presenta el estudio de dos modelos implementados sobre GPU del algoritmo de Gauss-Jordan, los cuales reducen la matriz hasta su forma escalonada reducida, para obtener directamente la solución del sistema de ecuaciones.

3. Solución de sistemas de ecuaciones por eliminación de Gauss-Jordan

El algoritmo de Gauss-Jordan es una variación del algoritmo de eliminación Gaussiana [Nesrin11]. El algoritmo de Gauss-Jordan, a diferencia del algoritmo de eliminación gaussiana, reduce la matriz **A** a su forma escalonada reducida de tal forma que la matriz **A** se convierte en matriz identidad de orden **n**, de esta forma encuentra la solución del sistema de ecuaciones de forma directa, sin realizar sustituciones sucesivas.

4. Programación de GPU usando CUDA

El modelo de programación CUDA de NVIDIA hace una distinción entre el código ejecutado en la CPU (host) con

su propia DRAM (host memory) y el ejecutado en GPU (device) sobre su DRAM (device memory). La GPU se representa como un coprocesador capaz de ejecutar funciones denominadas kernels, y provee extensiones para el lenguaje C que permiten alojar datos en la memoria de la GPU y transferir datos entre GPU y CPU. Por lo tanto, la GPU sólo puede ejecutar las funciones declaradas como kernels dentro del programa, y sólo puede usar variables alojadas en device memory, lo cual significa que todos los datos necesarios deberán ser movidos a esta memoria de forma previa a la ejecución del kernel.

Los threads son organizados en Blocks, y cada uno de éstos se ejecuta completamente en un único multiprocesador. Esto permite que todos los threads de un mismo Block compartan la misma memoria. Un kernel puede ser ejecutado por un número limitado de Blocks, y cada Block por un número de threads. Sólo los threads que pertenecen al mismo Block pueden ser sincronizados. Es decir, hilos que pertenezcan a Blocks distintos sólo pueden ser sincronizados realizando un nuevo lanzamiento de kernel. En la figura 1, mostramos la organización de los threads sobre los bloques. Asumiremos que en cada bloque tenemos solo 5 hilos entonces el id de cada thread en cada bloque varía de 0 hasta 4, para obtener diferentes identificadores para los threads multiplicamos el id del bloque por su dimensión y le agregamos el id del thread respectivo. Así tenemos diferentes identificadores para cada thread.

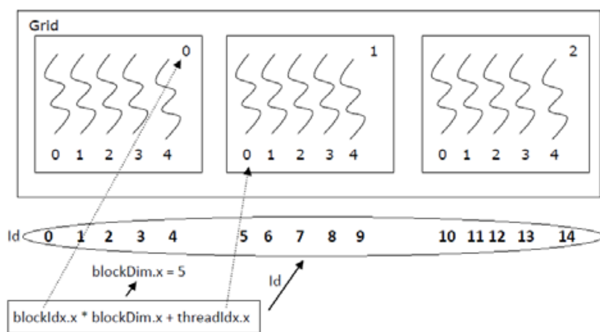

Figura 1. Obtención de un identificador diferente para cada thread.

5. Modelo implementado

Se va estudiar dos formas de convertir la matriz de coeficientes **A** en una matriz escalonada reducida. La primera es haciendo eliminación hacia adelante es decir transformándola primero en triangular superior, y luego hacer eliminación hacia atrás convirtiéndola en matriz identidad, para finalmente dar solución al sistema de ecuaciones. La segunda es hacer reducciones en las filas que están abajo y arriba del pivote actual en cada lanzamiento del kernel para finalmente dar solución al sistema.

El primer modelo implementado se divide en tres partes las cuales son implementadas como kernel y ejecutadas de forma paralela. Primero se reduce la matriz aumentada a triangular superior, lo que se conoce como eliminación hacia adelante, luego se reduce a triangular inferior (eliminación hacia atrás) y finalmente se encuentran las soluciones asignando cada elemento de la última columna de la matriz aumentada al vector de soluciones X.

El segundo modelo convierte la matriz A en escalonada reducida en una sola pasada por los pivotes, es decir para cada uno de los pivotes elimina elementos arriba y abajo del pivote, de tal forma que al final la matriz está en su forma escalonada reducida.

A continuación, describimos cada una de estas partes:

5.1. Eliminación hacia adelante

La reducción comienza desde la parte superior izquierda, a través de los elementos de la diagonal principal llamados pivotes. En total se realiza **n-1** lanzamientos del kernel, en cada lanzamiento se reduce las filas debajo de la fila del pivote de tal forma que los elementos de la columna que queda debajo del pivote actual se convierten todos en ceros. Este cálculo es realizado en paralelo con **n-i** hebras en cada iteración, para **i=1…n-1**.

En cada iteración cada hebra realiza la reducción de cada fila que queda debajo del pivote actual. En la figura 2, se muestra como ejemplo la eliminación hacia adelante para n=12.

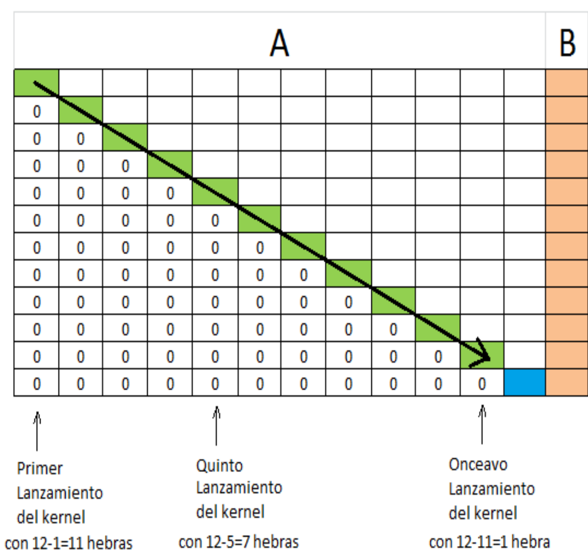

Figura 2. Eliminación hacia adelante.

A continuación, mostramos el kernel 1 que reduce la matriz **A,** a triangular superior, donde **AB** es la matriz aumentada, **n** es la cantidad de incógnitas del sistema y **poscol** es la posición de la columna del pivote actual:

kernel 1 Eliminación hacia adelante

```
__global__ void eliminacionAdelante(float * AB, int n, int
poscol) {
   int id = blockDim.x * blockIdx.x + threadIdx.x;
   if(id>=n-1-poscol){
            return;
   }
   int pospivot = (n + 2) * poscol;
   int posfinfila=(n + 1) * (poscol + 1);
   float piv=AB[pospivot];
   for (int j = pospivot; j < posfinfila; j++) {
            AB[j] =AB[j]/piv ;
   }
   int posfactor=pospivot + (n + 1) *(id+1);//posiciones bajo
el pivot
   float factor = AB[posfactor];
   for (int j = pospivot; j < posfinfila; j++) {
            int posactualelim=j + (n + 1) * (id + 1);
      AB[posactualelim] = -1*factor * AB[j] +
AB[posactualelim];
   }
}
```

5.2. Eliminación hacia atrás

La eliminación hacia atrás comienza desde la parte inferior derecha de la matriz **A**, y se va a través de cada uno de los pivotes en la diagonal principal. Después de haber realizado la reducción triangular superior todos los elementos debajo de la diagonal principal son ceros, por lo tanto, la reducción se hace solo sobre los elementos de la columna que está arriba del pivote y sobre los elementos de la columna que tiene los términos independientes, pues el resto de elementos son ceros. Este cálculo también es realizado en paralelo con **n-i** hebras en cada lanzamiento del kernel con **i=1…n**. En la figura 3, se muestra como ejemplo la eliminación hacia atrás para n=12.

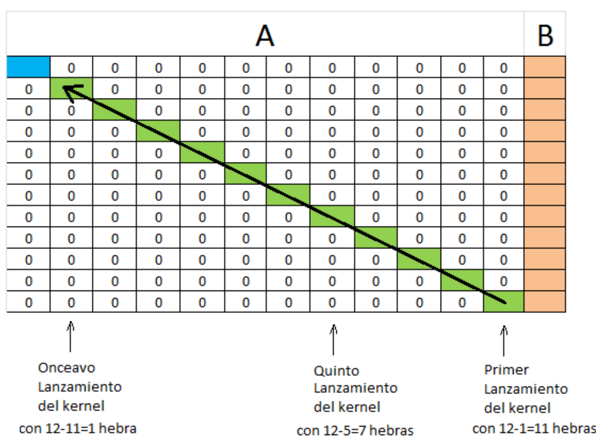

Figura 3. Eliminación hacia atrás.

A continuación mostramos el kernel 2 que reduce la matriz a triangular superior, donde AB es la matriz aumentada, n es la cantidad de incógnitas del sistema y poscol es la posición de la columna del pivote actual:

kernel 2 Eliminación hacia atrás

```
__global__ void eliminacionAtras(float * AB, int n, int
poscol) {
   int id = blockDim.x * blockIdx.x + threadIdx.x;
   if(id>=n-1-poscol){
            return;
   }
   int pospivot = (n + 2) * (n - 1 - poscol);
   if(poscol==0){
            float pivot=AB[pospivot];
            AB[pospivot]=AB[pospivot]/pivot;
            AB[pospivot+1]=AB[pospivot+1]/pivot;
   }
   float factor = AB[pospivot - (n + 1)*(id + 1)];
   int posactualelim1=pospivot - (n + 1)*(id + 1);
   int posactualelim2=pospivot - (n + 1)*(id + 1) + 1 +
poscol;
   AB[posactualelim1] = -1*factor * AB[pospivot] +
AB[posactualelim1];
   AB[posactualelim2] = -1*factor * AB[pospivot + 1 +
poscol] + AB[posactualelim2];
}
```

5.3. Reducción a triangular superior e inferior

En este caso avanzamos de izquierda a derecha como haciendo una eliminación hacia adelante, en total se realiza **n** lanzamientos del kernel, en cada lanzamiento se reduce las filas que queden por arriba y abajo del pivote actual, de tal forma que todos los elementos arriba y debajo de la columna del pivote se convierten en ceros, ese cálculo es realizado en paralelo con **n** hebras en cada iteración, en este caso se está manteniendo constante el número de hebras en cada lanzamiento del kernel.

En cada iteración cada hebra realiza la reducción de cada fila que queda debajo y arriba del pivote actual. En la figura 4 se muestra como ejemplo la reducción para n=12.

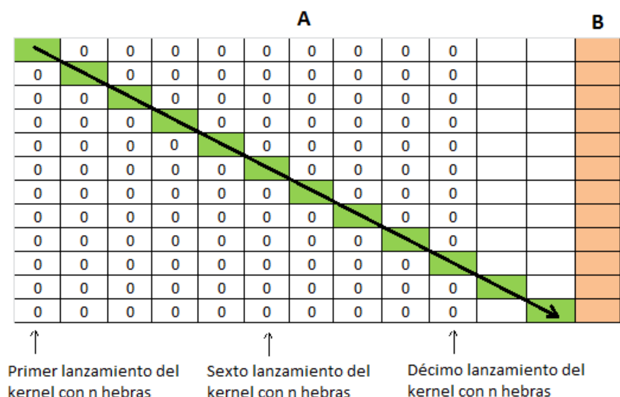

Figura 4. Reducción Triangular Inferior.

A continuación, mostramos el kernel 3 que reduce la matriz a triangular superior e inferior, donde **AB** es la matriz aumentada, n es la cantidad de incógnitas del sistema y **poscol** es la posición de la columna del pivote actual:

kernel 3 Reducción a triangular superior e inferior

```
__global__ void triangSupInf(float * AB, int n, int poscol) {
   int idx = blockDim.x * blockIdx.x + threadIdx.x;
   if(idx>=n){
            return;
   }
   int pospivot = (n + 2) * poscol;
   int posfinfila=(n + 1) * (poscol + 1);
   float piv=AB[pospivot];
   for (int j = pospivot; j < posfinfila; j++) {
      AB[j] =AB[j]/piv ;
   }
   int posfactor=pospivot%(n+1)+idx*(n+1);
   if(posfactor!=pospivot){
            float factor=AB[posfactor];
            for (int j = pospivot; j < posfinfila; j++) {
               int posactualelim=j%(n+1)+idx*(n+1);
               AB[posactualelim] =-
1*factor*AB[j]+AB[posactualelim] ;
            }
   }
}
```

5.4. Solución del sistema de ecuaciones

Una vez que la matriz A quedó en su forma escalonada reducida con cualquiera de los dos modelos descritos anteriormente, el resultado del sistema de ecuaciones lo obtenemos, lanzando un kernel con **n** hebras que hacen la asignación de cada uno de los elementos de la última columna de la matriz aumentada **AB** al vector solución **X**.

Finalmente, mostramos el kernel 4 que obtiene el vector solución del sistema de ecuaciones.

kernel 4 Solución del sistema de ecuaciones

```
__global__ void resultado(float *AB, int n, float* X) {
   int id = blockDim.x * blockIdx.x + threadIdx.x;
   if (id < n) {
            int posultimacol=(id + 1)*(n + 1) - 1;
                     X[id] = AB[posultimacol];
   }
}
```

Ahora, mostramos el lanzamiento del kernel para cada uno de los modelos implementados, el algoritmo 1 muestra el lanzamiento del kernel para el modelo 1:

Algoritmo 1 Lanzamiento del kernel para el modelo 1

```
   int blocksize = 1024;
   for (int i = 0; i < n-1 ; i++) {
      int numBlocks = ceil((n-1-i)/((float)blocksize));
      eliminacionAdelante << <numBlocks, blocksize >> >
(d_AB, n, i);
   }
   for (int i = 0; i < n - 1; i++) {
      int numBlocks = ceil((n - 1 - i)/((float)blocksize));
      eliminacionAtras << <numBlocks, blocksize >> > (d_AB,
n, i);
   }
   int nbloques=ceil(n/((float)blocksize));
   resultado << <nbloques, blocksize >> >(d_AB, n, d_X);
```

A continuación el algoritmo 2 muestra el lanzamiento del kernel para el modelo 2:

Algoritmo 2 Lanzamiento del kernel para el modelo 2

```
int blocksize = 1024;
int numBlocks = ceil((n)/((float)blocksize));
for (int i = 0; i < n ; i++) {
   gaussJordan << <numBlocks, blocksize >> > (d_AB, n, i);
}
resultado << <numBlocks, blocksize >> >(d_AB, n, d_X);
```

6. Experimentos y Resultados

Para la realización de los experimentos, se ha trabajado con un procesador AMD Phenom(tm) II X4 945 Processor 3.00 GHz. Memoria RAM de 1.0 GB. A continuación, mostramos la información de la GPU utilizada para el desarrollo del presente trabajo:

Device Name: GeForce GT 520
Total global memory size: 1073741824 bytes
Total Shared memory per block: 49152 bytes
Total constant memory: 65536 bytes
Max mem pitch: 2147483647
Texture Alignment: 512
Multiprocessor count: 1
Registers per block: 32768
Threads in warp: 32
Max threads per block: 1024
Max thread dimensions: (1024, 1024, 64)
Max grid dimensions: (65535, 65535, 65535)

Los experimentos fueron realizados con datos generados de forma aleatoria, las cuales fueron asignadas a una matriz que representa la matriz aumentada (A|B) la cual

contiene n*(n+1) datos. Estos datos fueron generados en la CPU y fueron pasados al GPU para la ejecución de los kernels.

En la tabla 1 mostramos los tiempos de ejecución sobre CPU y GPU así como el speed up para diferentes cantidades de datos. El tiempo de ejecución en GPU considera el tiempo que demora en transferir los datos del host al dispositivo así como la transferencia del vector solución desde el dispositivo al host.

Tabla 1. Comparación del Tiempo de ejecución en CPU y GPU considerando el tiempo de enviar el vector de soluciones desde el dispositivo al host.

N	Tiempo CPU (seg)	Tiempo GPU Modelo 1 (seg)	Speed up	Tiempo GPU Modelo 2 (seg)	Speed up
1000	2.169	8.891	0.24	11.321	0.19
2000	17.862	12.259	1.46	12.383	1.44
3000	60.252	13.556	4.44	13.665	4.41
4000	145.015	14.681	9.88	14.711	9.86
5000	279.771	15.086	18.55	16.071	17.41
6000		16.569		16.837	
7000		17.488		18.382	
8000		18.876		19.959	
10000		22.149		24.785	

En la figura 5, mostramos una comparación de los tiempos sobre CPU y GPU considerando la transferencia del vector de soluciones desde el device al host.

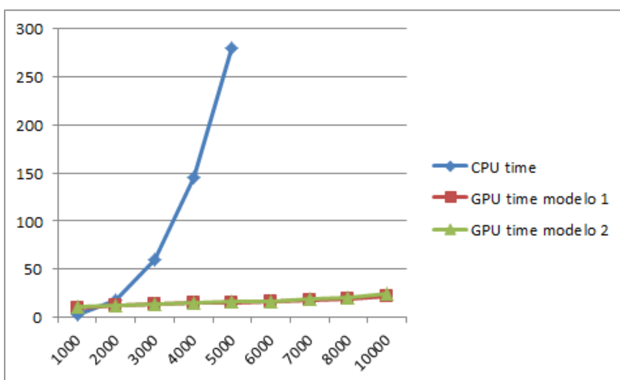

Figura 5. Comparación del Tiempo de ejecución en CPU y GPU considerando el tiempo de enviar el vector de soluciones desde el dispositivo al host.

En la tabla 2, mostramos los tiempos de ejecución sobre CPU y GPU, así como el speed up para diferentes cantidades de datos. En este caso el tiempo de ejecución en GPU considera el tiempo que demora en transferir los datos del host al dispositivo, pero no la transferencia del vector de soluciones desde el dispositivo al host.

Tabla 2 Comparación del tiempo de ejecución en CPU y GPU sin enviar el vector de soluciones desde el dispositivo al host.

N	Tiempo CPU (seg)	Tiempo GPU Modelo 1 (seg)	Speed up	Tiempo GPU Modelo 2 (seg)	Speed up
1000	2.254	0.215	10.48	0.282	7.99
2000	18.162	1.262	14.39	1.364	13.32
3000	60.696	2.215	27.40	2.676	22.68
4000	145.007	4.361	33.25	4.718	30.73
5000	280.315	5.762	48.65	5.753	48.73
6000		6.601		6.786	
7000		6.923		7.791	
8000		8.753		9.856	
10000		10.192		12.959	

En la figura 6, mostramos una comparación de los tiempos de ejecución sobre CPU y GPU.

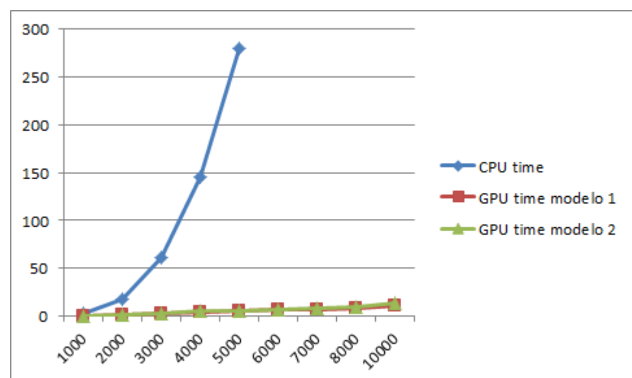

Figura 6. Comparación del tiempo de ejecución en CPU y GPU sin enviar el vector de soluciones desde el dispositivo al host.

7. Discusión de los Experimentos

Como se puede observar en la sección anterior, a medida que aumentan la cantidad de datos la CPU se vuelve más ineficiente, ya que para n=5000 la CPU demora cerca de 5 minutos en responder, sin embargo ambos modelos implementados en GPU dan resultados en menores tiempos, aunque ligeramente resulta ser mejor el modelo 1. En el caso de la ejecución en GPU, la transferencia del vector solución desde el dispositivo al host es un proceso que toma bastante tiempo, sin embargo, aun así, los modelos implementados en GPU toman ventajas sobre el CPU. El tiempo de GPU va incrementándose muy poco a medida que crecen el tamaño de datos, sin embargo la CPU va tomando cada vez mayor tiempo.

8. Conclusiones y trabajo futuro

El proceso de enviar el vector de soluciones desde el dispositivo a la CPU es bastante comparado con el tiempo que le toma a la GPU realizar el proceso de reducción hacia adelante y hacia atrás.

Sin considerar el tiempo de enviar el vector de soluciones desde el dispositivo al host, el modelo implementado logra obtener mejores speed up, sin embargo aun considerando ese tiempo se sigue obteniendo mejores resultados que la CPU.

Para pequeñas cantidades de datos, la CPU puede tener mejor desempeño debido a que enviar los datos desde el host al dispositivo y viceversa toma bastante tiempo.

El modelo 1 en GPU logra ligeramente tomar ventaja sobre el modelo 2 debido a que la eliminación hacia atrás, en el modelo 1, solo se realiza sobre los elementos que están en la columna arriba del pivote y sobre el vector de coeficientes independientes B. Sin embargo, el modelo 2 implementado realiza el equivalente de 2 eliminaciones hacia adelante, tanto sobre las filas abajo del pivote como las filas arriba del pivote.

En este paper, en cada lanzamiento del kernel, cada hebra es asignada para realizar la reducción de una fila. Se propone como trabajo futuro, que en cada lanzamiento del kernel, cada fila sea reducida por tantas hebras como elementos queden por reducir en la fila, a partir de la columna donde está el pivote actual.

Referencias bibliográficas

[Buluç08] Buluç, A. (2008). Gaussian Elimination Based Algorithms on the GPU. Computer Science Department, University of California

[Ezzatti10] Ezzatti, P. (2010) Resolución de sistemas triangulares en tarjetas Gráficas (gpu), Centro de Cálculo–Instituto de Computación, Universidad de la República, 11.300–Montevideo.

[Kalinová11] Kalinová, P. (2011) Solving Large Sparse Systems of Linear Equations on GPU, Department of Computer Graphics and Interaction, Czech Technical University in Prague.

[Kindelán08] Kindelán, U. (2008). Resolución de sistemas lineales de ecuaciones: Método del gradiente conjugado. Departamento de Matemática Aplicada y Métodos Informáticos, Universidad Politécnica de Madrid.

[Martinotto04] Martinotto, A. Resolução de Sistemas de Equações Lineares através de Métodos de Decomposição de Domínio. Dissertação de Mestrado, Universidade Federal do Rio Grande do Sul. Instituto de informática, 2004.

[Michels11] Michels, D. (2011) Sparse-Matrix-CG-Solver in CUDA. Institute of Computer Science II, Universitat Bonn

[Nesrin11]Nesrin, A.(2011) Using gauss - Jordan elimination method with CUDA for linear circuit equation systems, Engineering Faculty, Karabuk University.

[Remón08] Remón, A. (2008) Resolución de Sistemas de Ecuaciones Lineales Banda Sobre Procesadores Actuales y Arquitecturas Multihebra. Aplicaciones en Control. E.S. de Tecnología y Ciencias Experimentales. Universidad Jaime I de Castellón.

[Soriano11] Soriano, J. (2011). Matrix inversion speed up with cuda. Electrical and computer Engineering Department, Illinois Institute of Technology.

[Yang11]Yang M. (2011). Parallel algorithm for solving large-scale dense linear system on CUDA. Department of Electrical Engineering↵ Harbin Institute of Technology↵ Harbin 150001↵ China.