# LU Decomposition Method implementation using Graphics Processing Units - GPU

Yensy Gómez, John Osorio, Lina Pérez , *Fellow, Sirius HPC , Universidad Tecnológica de Pereira*

***Abstract*—This article pretends to show how you can use GPU's to make the calculations of an LU decomposition and take advantage of the vectorial processors. At the first section we make an introduction of matrix factorization, in the second one we show some features of actual graphics cards, the third one we propose the problem to solve, in the fifth one we expose the implementation and finally the results of the research.**

***Index Terms*—GPU, LU, Matricial, CUDA, OpenCL, nVidia, ATI, HPC.**

## I. Introducción

CADA día el desarrollo de la computación es más diversa, debido a que se exploran y se explotan las capacidades de una gran variedad de arquitecturas como son: Los micro-procesadores multi-núcleo, unidades de procesamiento central (CPUs), procesadores de señal digital, hardware reconfigurable (FPGAs) y las unidades de procesamiento gráfico GPUs. Cada una de estas arquitecturas se puede encontrar sobre un equipo de cómputo formando lo que hoy se conoce como computación heterogénea.

Este tipo de dispositivos se utilizan para ejecutar aplicaciones que poseen un gran número de cargas de trabajo que van desde búsquedas, clasificación, análisis de grandes volúmenes de datos, métodos numéricos o modelos financieros, donde cada uno de ellos necesita ser ejecutado bajo arquitecturas específicas que mejoran su rendimiento obteniendo datos en menores tiempos de ejecución de los problemas a resolver.

En el presente artículo se soluciona un problema básico de algebra lineal que permite utilizarse como insumo para la solución de sistemas de ecuaciones lineales como se puede ver en la ecuación 1. La descomposición LU permite solucionar un sistema matricial sin la necesidad de calcular la inversa de la matriz A.

$$\begin{array}{ccccccccc} a_{1,1}x_1 & + & a_{1,2}x_2 & + & \dots & + & a_{1,n}x_4 & = b_1 \\ a_{2,1}x_1 & + & a_{2,2}x_2 & + & \dots & + & a_{2,n}x_4 & = b_2 \\ \vdots & & \vdots & & \ddots & & \vdots & \vdots \\ a_{m,1}x_1 & + & a_{m,2}x_2 & + & \dots & + & a_{m,n}x_n & = b_n \end{array} \tag{1}$$

El sistema de ecuaciones lineales 1 puede escribirse en forma matricial como se ve en la ecuación 2.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \ddots & a_{2n} \\ \vdots & \vdots & \dots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{2}$$

**978-1-4799-6717-9/14/$31.00 © 2014 IEEE**

La factorización LU es una variante de la eliminación Gaussiana la cual realiza una descomposición de una matriz como el producto de una matriz triangular inferior y una matriz triangular superior $A = LU$, además de ser conveniente en términos del número de operaciones de punto flotante que deben llevarse a cabo [1]. Ésta descomposición conduce a un algoritmo para hallar la solución a un sistema de ecuaciones lineales de la forma $Ax = b$, a nivel computacional la anterior representación es la más usada para resolver sistemas lineales.

La principal razón por la que el método de factorización LU es el más utilizado radica en que este ofrece una manera muy económica de resolver un sistema de ecuaciones lineales [2].

Un ejemplo de una descomposición LU puede verse en la ecuación 3.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} \alpha_{11} & 0 & 0 \\ \alpha 21 & \alpha_{22} & 0 \\ \alpha 31 & \alpha_{32} & \alpha_{33} \end{pmatrix} \begin{pmatrix} \beta_{11} & \beta_{12} & \beta_{13} \\ 0 & \beta_{22} & \beta_{23} \\ 0 & 0 & a_{33} \end{pmatrix} \tag{3}$$

Para dar solución a un sistema de la forma $Ax = b$, se debe tener en cuenta la ecuación 4.

$$Ax = (LU)x = L(Ux) = b \tag{4}$$

Ahora $y = Ux$, se puede resolver para $y$ usando forward substitution y $Ly = b$ puede resolverse usando back substitution [3].

La ecuación 5 muestra el proceso para hacer forward substitution.

$$\begin{aligned} y_1 &= \frac{b_1}{\alpha_{11}} \\ y_i &= \frac{1}{\alpha_{ii}} \left[ b_i - \sum_{i=1}^{i-1} \alpha_{ij} y_j \right] i = 2, 3, \dots, N \end{aligned} \tag{5}$$

La ecuación 6 muestra el proceso de cálculo de backward substitution.

$$\begin{aligned} x_N &= \frac{y_N}{\beta_{NN}} \\ x_i &= \frac{1}{\beta_{ii}} \left[ y_i - \sum_{j=i+1}^{N} \beta_{ij} x_j \right], i = N, N-2, \dots, 1 \end{aligned} \tag{6}$$

## II. Arquitecturas GPU Actuales

Desde el año 2007, cuando nvidia creó CUDA (Compute Unified Device Architecture) [4], se ha iniciado un proceso de estudio e investigación en el área de la computación científica sin precedentes, ya que gracias a la gran penetración de las tarjetas gráficas en el mercado se puede contar con dispositivos de gran capacidad de cómputo en equipos de escritorio.

La evolución de las GPUs ha pasado por la creación de distintas arquitecturas, cada una superando en recursos a su predecesor. A continuación se mencionarán algunas de las arquitecturas existentes.

### A. Architectura G80

Cuando nVidia lanzó al mercado la GeForce 8800 se conoció entonces la arquitectura G80, la cual era la primer GPU en el mercado que utilizaba una arquitectura unificada que ejecuta los vértices, la geometría, los pixeles y los programas de computación, además se basa en un modelo de instrucciones conocido como SIMT el cual ejecuta múltiples hilos de manera independiente y al mismo tiempo utilizando una sola instrucción. Esta GPU soporta lenguaje C, lo que hace que los programadores no tengan que aprender otro lenguaje y opera de forma integral con una precisión de 32 bits para operaciones de punto flotante ajustándose al estándar IEEE 754 [5] [16].

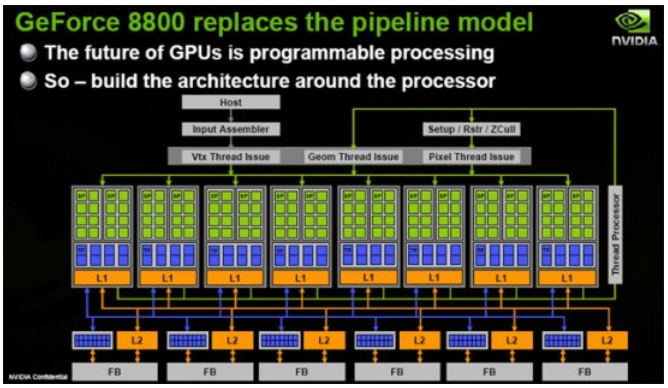


Fig. 1: Arquitectura G80

En la Figura 1 se muestra un esquema completo de la arquitectura G80; en las siguientes secciones se detalla a cada una de las partes que lo componen. La arquitectura G80 de nVidia comenzó su desarrollo a mediados de 2002, publicando su versión definitiva en los últimos meses de 2006 conocida como GT200. El objetivo básico de la mejora de las capacidades de procesamiento gráfico llevó a:

- Incremento de forma significativa de las prestaciones con respecto a la última generación de GPUs.
- Aumento de la capacidad de cálculo de punto flotante por parte de la GPU, con la vista puesta en la introducción definitiva de este tipo de procesadores en el ámbito de la computación general.
- Adición de nuevos elementos al pipeline clásico, para cumplir las características definidas por Microsoft en DirectX 10.

Una ventaja en esta arquitectura es la posibilidad de equilibrar la carga entre los distintos procesadores, los cuales se pueden asignar a una tarea u otra dependiendo del trabajo a realizar. Como desventaja se puede considerar la mayor complejidad de los procesadores y su no especificidad en un tipo de problemas.

### B. Architectura Fermi

La arquitectura Fermi ver Figura 2 dio un salto importante en la arquitectura de las GPU. G80 era una visión general de una arquitectura unificada y paralela.GT200 extendió el rendimiento y la funcionalidad de la arquitectura de G80. Esta arquitectura emplea un enfoque completamente nuevo a la hora de diseñar la GPU. Las áreas principales en las que centraron fueron: Mejora del rendimiento de precisión doble. El rendimiento en Operaciones de punto flotante aumento 10 veces en comparación con el rendimiento de la CPU. Jerarquía de memoria Cache. Cuando un algoritmo paralelo no puede utilizar memoria compartida y los usuarios solicitan el uso de una arquitectura real de cache. Más memoria compartida. Muchos programadores CUDA solicitaron más de 16KBde memoria compartida. Cambios de Contexto.Los usuarios exigieron cambios de contexto más rápidos entre las aplicaciones, gráficos y cálculo de interoperabilidad. Operaciones más Rápidas. Los usuarios solicitaron mayor rapidez de lectura y escrituras de las operaciones de algoritmos paralelos.

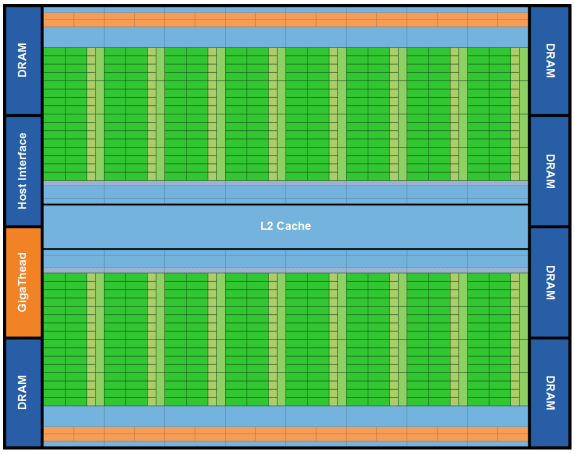


Fig. 2: Arquitectura Fermi

La arquitectura Fermi es una arquitectura modular, cuya base son los SM (Streaming Multiprocessors), los que son arrays (agrupaciones) de unidades de cómputo entre las que tenemos: procesadores de Shaders, unidades de textura, motores de teselado, entre otras. A su vez, los SM están organizados en arrays de 2 o 4 SMs, los que son denominados GPC (Graphic Processing Cluster), 1 o más GPCs agrupados conforman un GPU basado en la arquitectura Fermi.

La GPU basada en fermi, implementa 3,0 millones de transistores, la característica es que tiene 512 núcleos CUDA, cada núcleo ejecuta una instrucción de punto flotante o entero por ciclo de reloj. Los 512 núcleos están organizados en 16 SM de 32 núcleos cada uno como se muestra en la Figura 2. Núcleos de alto desempeño.. La GPU cuenta con 6 divisiones

de 64 bits, que soportan hasta un total de 6 GB de memoria GDDR5 DRAM y se conecta la GPU con la CPU a través del puerto PCIe.

Cada SM cuenta con 32 procesadores CUDA, cada procesador tiene una unidad aritmético lógica (ALU) y una unidad de punto flotante (FPU). La arquitectura Fermi implementa un nuevo estándar IEEE754-2008 de punto flotante, funcionando el Multiply-add (FMA), que permite que una instrucción sume y multiplique al tiempo ya sea de precisión simple o doble [6].

En GT200, la ALU se limita a 24-bits de precisión para multiplicar, en Fermi el nuevo diseño de la ALU soporta 32 bits de precisión para todas las instrucciones. La ALU también esta optimizada para apoyar de manera eficiente las operaciones de precisión de 64 bits. Diversas instrucciones están soportadas incluyendo cambio de Boole, mover, comparar, inserción de bits y recuento de población.

En cada bloque de Streaming Multiprocessors (SM), se encuentran otros elementos como las 4 unidades de textura, el "Raster Engine", y un elemento bien importante como el "PolyMorph Engine". El diseño general de la arquitectura se dividió en cuatro grandes grupos llamados GPC (GraphicsProcessingCluster), en lugar de uno solo como en la generación actual, metafóricamente hablando el GF100 es como una GPU Quad-Core con sus componentes organizados de tal forma que puedan tener la eficiencia y potencia necesaria para tareas de cómputo altamente demandantes, como geometría compleja en gráficos 3D y para eso incorpora nuevas unidades especializadas para el cálculo geométrico.

Además Fermi tiene en cada SM, 16 unidades de load/store [6] lo que permite direcciones de origen y destino, que se calcula 16 veces por ciclo de reloj, estas unidades cargan y almacenan los datos en cada dirección de la caché. Las unidades especiales de función (SFU) ejecutan las instrucciones transcendentales tales como el seno, el coseno, y raíz cuadrada. Cada SFU ejecuta una instrucción por hilo.

Por lo tanto la arquitectura Fermi ha sido especialmente diseñada para aplicaciones de alto desempeño como algebra lineal, simulación numérica o aplicaciones que consuman muchos recursos computacionales, ofreciendo un rendimiento tanto en precisión como en eficiencia. En [6], se muestra el alto rendimiento de la arquitectura Fermi con respecto a la GT200 obteniendo un resultado de 4.5x más rápido y preciso que la anterior arquitectura.

### C. Arquitectura Kepler

A medida que aumenta la demanda del alto rendimiento en la computación paralela desde muchos ámbitos de la ciencia como la medicina, la ingeniería y finanzas; nVidia sigue innovando para satisfacer esta demanda con nuevas arquitecturas de GPUs que son extraordinariamente poderosas. Las GPUs de nVidia van redefiniendo y acelerando sus capacidades en áreas como las simulaciones bioquímicas, procesamiento de señales, finanzas, ingeniería asistida por computador, dinámica de fluidos computacionales y análisis de datos.

La arquitectura Kepler GK110 GPU ayudará a resolver la mayor parte de problemas informáticos al ofrecer mucha más potencia de procesamiento en comparación a las arquitecturas anteriores proporcionando nuevos métodos para optimizar y aumentar la carga de trabajo de ejecución en paralelo revolucionando la computación de alto rendimiento.

El objetivo de Kepler es ser mucho mejor en rendimiento. GK110 no solo supera a la arquitectura Fermi en potencia sino que también consume menos energía y genera menos calor. Una arquitectura Kepler GK110 incluye 15 unidades y seis controladores de memoria SMX de 64-bits. Las principales características de esta arquitectura son:

- Una nueva arquitectura de procesador SMX.
- Un subsistema de memoria, que ofrece capacidades adicionales de almacenamiento en caché, más ancho banda a nivel de la jerarquía, y una DRAM totalmente rediseñada.
- Soporte de hardware en todo el diseño para permitir nuevas capacidades del modelo de programación.

## III. Método de Crout

En el método de Crout la matriz A se factoriza $A = LU$, $L$ es una matriz triangular inferior y $U$ es una matriz triangular superior donde la diagonal tiene el valor de 1 [7]. Lo primero que debe llevarse a cabo es la escritura del i,j ésimo componente de la Ecuación 3. Este componente siempre inicia con la suma:

$$\alpha_{i1}\beta_{1j} + \ldots = a_{ij} \tag{7}$$

El número de términos del sistema depende, sin embargo, de si i o j es el número más pequeño. De hecho se tienen 3 casos:

$$i < j : \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \ldots + \alpha_{ii}\beta ij = a_{ij} \tag{8}$$

$$i = j : \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \ldots + \alpha_{ii}\beta jj = a_{ij} \tag{9}$$

$$i > j : \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \ldots + \alpha_{ij}\beta jj = a_{ij} \tag{10}$$

Las ecuaciones 8, 9 y 10 dan un total de $N^2$ ecuaciones para las $N^{2+N}$ incógnitas de los $\alpha$ y $\beta$ (esto se da ya que la diagonal está representada dos veces). Como el número de incógnitas es mayor que el número de ecuaciones, se deben especificar $N$ de las incógnitas arbitrariamente y tratar de resolver para las otras, de hecho como se puede ver siempre es posible tener:

$$\alpha_{ii} = 1, i = 1, \ldots, N \tag{11}$$

Ahora el algoritmo de Crout soluciona el sistema de $N^{2+N}$ de las ecuaciones 8, 9 y 10 para todos los $\alpha$ y $\beta$ solamente reorganizando las ecuaciones como se muestra a continuación:

- Se inicializan los valores de la diagonal de acuerdo a la ecuación 11.
- Para cada $j = 1, 2, 3, \ldots, N$ hay que hacer los siguientes procedimientos. Primero, para $i = 1, 2, \ldots, j$ usar las ecuaciones 8, 9 y 10 para solucionar $\beta_{ij}$:

$$\beta_{ij} = \alpha_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj} \tag{12}$$

Segundo, para el cálculo de los $\alpha$:

$$\alpha_{ij} = \frac{1}{\beta_{jj}} \left[ \alpha_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj} \right] \tag{13}$$

En la Figura 3 se puede observar como los elementos de la matriz original son modificados en el orden indicado por las letras en minúscula de la imagen: a, b, c, entre otras. Los rectángulos sombreados muestran los elementos modificados previamente que son utilizados para calcular un nuevo elemento indicado con una "X".

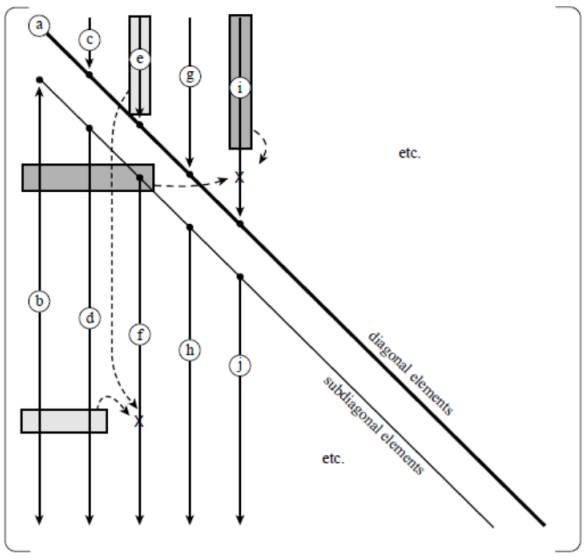


Fig. 3: Orden de Modificación Matriz

Al realizar algunas iteraciones del método de Crout se podrá observar que cada $\alpha_{ij}$ es utilizado solo una vez. Esto significa que para optimizar el uso de la memoria los valores correspondientes a $\alpha_{ij}$ y $\beta_{ij}$ se pueden almacenar en las posiciones correspondientes de la matriz A. La diagonal de elementos unitarios de $\alpha_{ii}$ de la ecuación 11 no se almacenan. Por lo tanto el método de Crout entrega como resultado una matriz con los $\alpha$ y los $\beta$ combinados de la siguiente forma.

$$\begin{pmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ \alpha_{21} & \beta_{22} & \beta_{23} & \beta_{24} \\ \alpha_{31} & \alpha_{32} & \beta_{33} & \beta_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \beta_{44} \end{pmatrix} \tag{14}$$

### A. Pivote Parcial

La estrategia de pivote parcial está restringida al intercambio de filas, la factorización $LU$ debe entonces satisfacer la siguiente ecuación:

$$LU = PA \tag{15}$$

Donde P es una matriz de permutación y se define de la siguiente manera:

- $P$ se inicializa en $I$.
- Para cada intercambio que se dé durante la descomposición de $A$ se debe efectuar el mismo intercambio en $P$.

Retomando la definición del sistema lineal de ecuaciones se tiene que:

$$Ax = b \tag{16}$$

Y multiplicando en ambos lados por P se tiene:

$$PAx = Pb \tag{17}$$

Usando la ecuación 17 para sustituir $PA$:

$$LUx = Pb \tag{18}$$

Se tiene entonces que:

$$y = Pb \tag{19}$$

$$Lc = y \tag{20}$$

$$Ux = c \tag{21}$$

El producto de $Pb$ se desarrolla antes de la sustitución hacia adelante (forward substitution).

## IV. Implementación

En el presente capítulo se hará la descripción de la factorización LU (Método de Crout) y cómo se llegó a la paralelización del método. Para la implementación secuencial se utilizó C, ésta implementación puede consultarse en [8], para la implementación en paralelo se utilizó OpenCL [9] y se planteó un ejemplo para una matriz de $4x4$.

### A. Proceso Paralelización

Como se puede observar en [8], los cálculos se llevan a cabo dentro de un loop que permite hallar la respuesta primero para los $\beta$ y luego para los $\alpha$ correspondientes en cada iteración, además es posible observar la existencia de la dependencia de datos para el cálculo de los elementos dentro de la factorización LU.

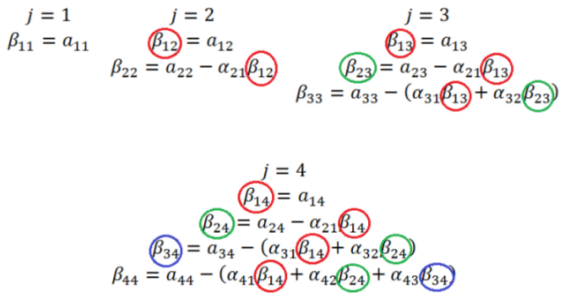


Fig. 4: Dependencia de datos cálculo $\beta$

Como se observa en la Figura 4 y haciendo el estudio para el caso de $j = 4$, se define que no se puede llevar a cabo el cálculo de $\beta_{14}$, $\beta_{24}$, $\beta_{34}$ y $\beta_{44}$ de manera simultánea, ya que para calcular el siguiente valor de $\beta$ es necesario los $\beta$ anteriores, es decir, si se quiere hacer el cálculo de $\beta_{44}$ es

necesario conocer $\beta_{14}$, $\beta_{24}$ y $\beta_{34}$, por lo tanto la paralelización en el cálculo de los $\beta$ por columnas no se puede hacer.

De la misma manera es posible observar que cuando se hace el cálculo de $\beta_{24}$ no hay dependencia de los datos $\beta_{23}$, ni de $\beta_{22}$, por lo tanto estos valores se pueden calcular de manera paralela, y es allí donde se encuentra el punto en el cálculo de los $\beta$ que se puede hacer en simultáneo, los $\beta$ se pueden calcular por filas.

$$j = 1 \qquad j = 2$$

$$\alpha_{21} = \frac{1}{\beta_{11}} a_{21}$$

$$\alpha_{31} = \frac{1}{\beta_{11}} a_{31} \qquad \alpha_{32} = \frac{1}{\beta_{22}} (a_{32} - \alpha_{31}\beta_{12})$$

$$\alpha_{41} = \frac{1}{\beta_{11}} a_{41} \qquad \alpha_{42} = \frac{1}{\beta_{22}} (a_{42} - \alpha_{41}\beta_{12})$$

$$j = 3 \qquad j = 4$$

$$\alpha_{43} = \frac{1}{\beta_{33}} (a_{43} - \alpha_{41}\beta_{13} + \alpha_{42}\beta_{23}) \quad \textit{Para esta iteracion no se calculan } \alpha\textit{'s}$$

Fig. 5: Dependencia de datos cálculo $\alpha$

Para los $\alpha$, se tiene que la dependencia de datos no se evidencia en la iteración, es decir, para $j = 1$ los datos se pueden calcular de manera independiente, lo único que se necesita son los cálculos previos de $\beta$. En [10] se puede ver la implementación en OpenCL.

## V. Resultados Obtenidos

A continuación se mostrarán las características técnicas del equipo utilizado para la ejecución del algoritmo secuencial y paralelo.

| Características del Equipo de Cómputo |
|---|
| Procesador Intel Core i7 3770k, 3,5 GHz |
| 16 GB Memoria RAM DDR3, con bus de 1600 MHz |
| Tarjeta Gráfica ATI 7950 Direct CU2, Manufacturada por ASUS |

Tabla I: Características Equipo

En la figura 6 se puede ver la aceleración obtenida por la versión paralelizada en OpenCL al compararla con la versión secuencial, inicialmente no se consigue ninguna aceleración, sin embargo, a partir de 65025 datos los tiempos empiezan a verse mejorados, hasta obtener una aceleración de 34x.

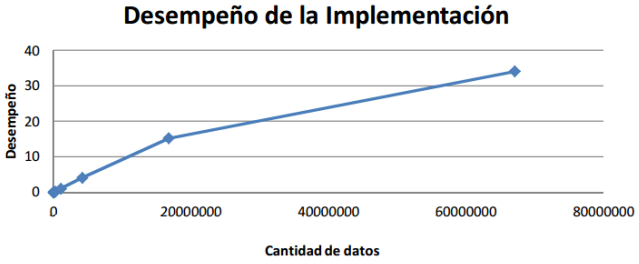


Fig. 6: Aceleración GPU

La figura 7 por su parte muestra una comparación entre los tiempos de ejecución de cada una de las implementaciones.

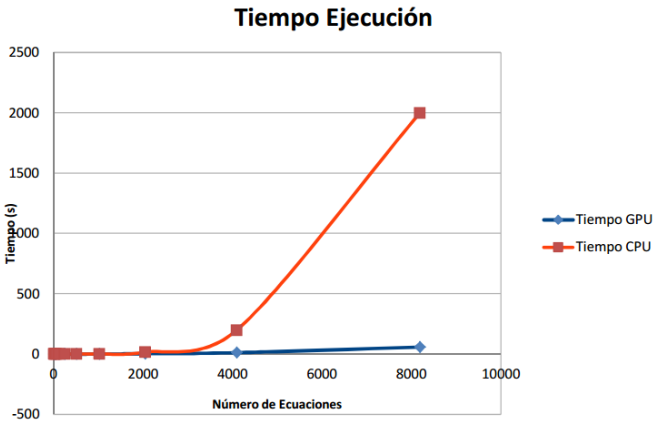


Fig. 7: Tiempo ejecución GPU vs CPU

## VI. Conclusiones

Se crearon dos algoritmos para realizar el cálculo de la factorización LU con pivote parcial, la primera implementación fue hecha en lenguaje C y su ejecución es absolutamente secuencial, la segunda implementación se construyó aprovechando el framework de trabajo OpenCL y de manera paralela.

Se evaluó el tiempo de ejecución de cada una de las implementaciones para diferentes conjuntos de datos como se puede ver en la figura 7. Se puede notar claramente que cuando el número de datos está entre 0 y 2000 ambas ejecuciones entregan tiempos similares, es decir no se logra ningún tipo de aceleración; sin embargo cuando se supera el umbral de 2000 datos, el algoritmo secuencial empieza a tardar más tiempo que el paralelo.

La implementación en paralelo necesitó de un cambio para el cálculo de los betas dentro del algoritmo de Crout el cuál consistió en realizar los cálculos resolviendo primero cada una de las filas, ya que entre ellas no hay dependencia de datos.

La figura 6 muestra el desempeño obtenido de la implementación en paralelo en comparación con la implementación secuencial, inicialmente se puede notar una mejora del desempeño que aumenta a medida que se realiza el cálculo con más datos hasta obtener un desempeño de 34X, lo cual muestra entonces que la implementación en paralelo si logra mejorar los tiempos de ejecución del algoritmo.

Cabe aclarar que la implementación en paralelo puede ser mejorada si se aprovecha la memoria local de cada uno de los procesadores vectoriales con los que cuenta la GPU; sin embargo esta implementación no se tuvo en cuenta para el presente trabajo y se plantea como un proceso investigativo a futuro.

## References

[1] D. Lay, *Algebra Lineal y sus Aplicaciones*, Mexico: Pearson Educacion, 2007.

[2] B. Kolman y D. Hill, *Algebra Lineal*, Mexico: Pearson Education, 2006.

[3] W. Press, S. Teukolsky, W. Vetterling y B. Flannery, *Numerical Recipes in C. The art of Scientific Computation*, Ney York: Cambridge University Press, 2002

[4] nVidia, *About CUDA*, https://developer.nvidia.com/about-cuda, 2014

[5] D. Goldberg, *What Every Computer Scientish Should Know About Floating-Point Arithmetic*, ACM Computer Surveys, vol. 23, nº 1, pp. 5-48, 1991
[6] nVidia, *NVIDIA*, http://www.nvidia.es/object/gpu-computing-es.html
[7] J. L. De la Fuente, *Tecnicas de Calculo para Sistemas de Ecuaciones, Programación Lineal y programacion Entera*, Barcelona: Editorial REVERTÉ, 1998.
[8] J. Osorio, L. Pérez, *Implementación Secuencial Descomposición LU*, https://github.com/kala855/luSequential
[9] A. Munshi, B. Gaster, T. Mattson, J. Fung y E. All, *OpenCL Programming Guide*, Pearson Education, Inc., 2012.
[10] J. Osorio, L. Pérez, *Implementación Paralela Descomposición LU*, https://github.com/kala855/luDecomposition