CPU Sequential Computing *vs.* GPU Parallel Computing: A Comparative Study for the Solution of Complex Problems

Vázquez-Ayala, Martin U.[1]; Bauer-Mengelberg, Juan R.[1*]; Del Valle-Paniagua, David H.[1]; Velázquez-López, Noé[2]

[1] Colegio de Postgraduados - Campus Montecillo, Montecillo, Texcoco, Estado de México, México, C. P. 56264.
[2] Universidad Autónoma Chapingo, Chapingo, Texcoco, Estado de México, México, C. P. 56230.
* Correspondence: jbauer@colpos.mx

ABSTRACT

Objective: To demonstrate the advantages of parallel programming using a graphics card for the solution of issues that would otherwise require excessive execution times.

Design/Methodology/Approach: A matrix inversion method was implemented for both sequential and parallel processes. The programmed processes were executed for different dimensions of the matrix to be inverted and the resulting execution times were compared.

Results: The reduction in processing time enables the inversion of large matrices in a short time (seconds); it can be scalable for other problems whose solution requires numerous operations and/or calculations.

Findings/Conclusions: The examples indicate the need for adequate computing equipment for high-performance tasks. Specifically, GPUs (graphics processing units) will make a greater contribution to the reduction of execution times than CPUs (central processing units).

Keywords: execution times, GPU, graphic card, parallel programming.

Citation: Vázquez-Ayala, M. U., Bauer-Mengelberg, J. R., Del Valle-Paniagua, D. H., & Velázquez-López, N. (2025). CPU Sequential Computing *vs.* GPU Parallel Computing: A Comparative Study for the Solution of Complex Problems. *Agro Productividad*. https://doi.org/10.32854/2x7c5d25

Academic Editor: Jorge Cadena Iñiguez
Associate Editor: Dra. Lucero del Mar Ruiz Posadas
Guest Editor: Daniel Alejandro Cadena Zamudio

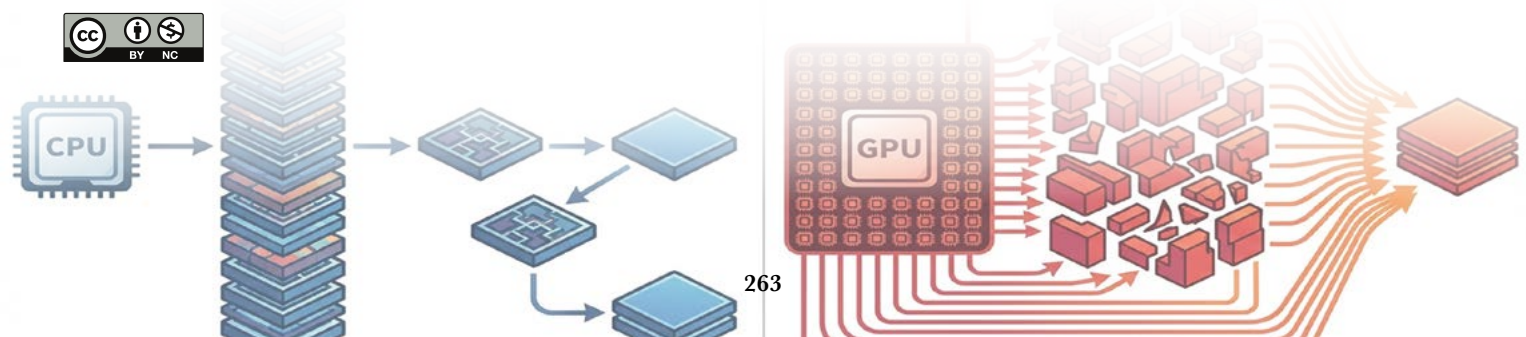
Received: November 5, 2024.
Accepted: October 13, 2025.
Published on-line: December XX, 2025.

Agro Productividad, 18(11). November. 2025. pp: 263-273.

INTRODUCTION

The capacity to efficiently solve complex problems is crucial in various areas of knowledge. Computational optimization has become vitally importance, as a consequence of the exponential growth of data and the need to process them within a reasonable time. Traditionally, CPU sequential computing has been the prevailing approach; however, as technology advances, GPU heterogeneous computing has emerged as a powerful alternative. According to Shen (2024), as computing demands keep growing, GPUs have become a pillar of high-performance computing (HPC). For example, its application in artificial intelligence and deep learning, physics simulations, number crunching, big data analytics, and other fields indicates that they have transcended their initial rasterisation function and have become accelerators for complex computing.

Despite its effectiveness, the sequential paradigm faces significant restrictions when it deals with tasks thar require high processing capacity, particularly when parallelism is



involved. Although GPUs were initially conceived for rendering purposes, their architecture can adequately execute multiple parallel operations, which makes them ideal for scientific and engineering applications that handle big data.

Sequential Computing

The "sequential computing" concept was developed to explain the paradigm in which the instructions for a program are executed one after another, until all have been completed, always using a single thread from a processing nucleus. Tanco (2004) explains that "a computer generally consists of a single processor that can manipulate instructions and data located in its memory. The processor reads and executes the instructions in its memory one by one; this is a sequential system, where everything happens in a single deterministic sequence". Figure 1 illustrates the execution process of sequential computing, where the program downloads all the tasks into the processor and each task is executed in the order in which it was numbered, until the task list determined in the programmed code has been finished.

Parallel computing

"Parallel computing" is a way to visualize computing problems: multiple instructions are executed simultaneously in several processing nuclei, subdividing the tasks to be executed (Figure 2). Parallel computing requires greater analysis and understanding of the problem; the need to manage synchronization can increase the complexity of its implementation; and

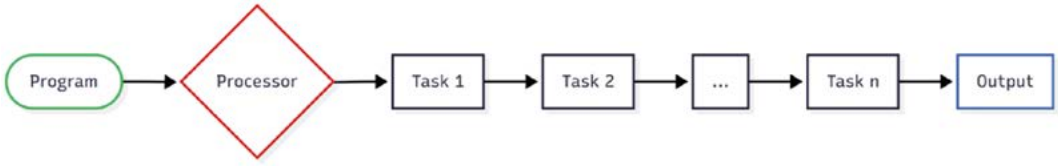


Figure 1. Execution diagram of sequential computing, developed by the authors (2024).

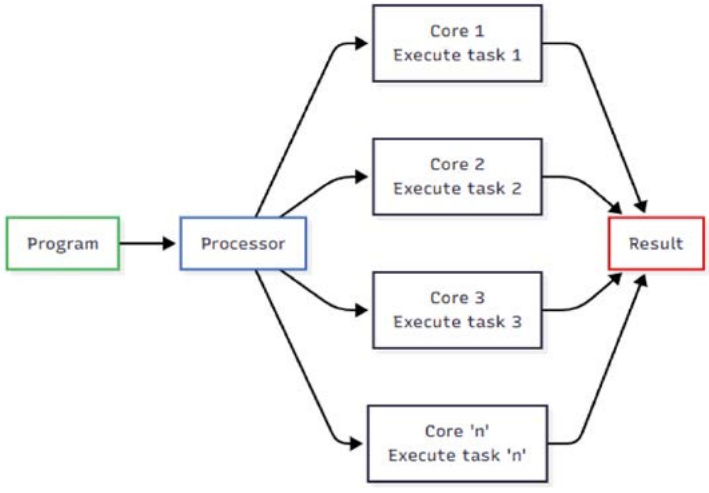


Figure 2. Execution diagram of parallel computing, developed by the authors (2024).

a balance between the load and the communication between processes must be achieved. Thus, one should determine if using parallel computing improves the solution (Rivares and Evangelista, 2000).

Central Processing Unit

The Central Processing Unit (CPU) is commonly known as the processor. This piece of hardware interprets the algorithms written in the software, using basic arithmetic and logical operations (Intel Corporation, 2024). As of 2024, the prevailing architecture in both desktop and portable personal computing equipment is x64 (derived from x86 architecture). Architecture is a set of instructions for the CPU, which describes how a processor works. It includes the instructions, records, buses, and operation modes that it can accept.

Graphics Processing Unit

A Graphics Processing Unit (GPU) is a chip or processor used to develop 2D or 3D graphics that will then be shown in a screen (Raya, 2008). Consequently, GPUs were designed in a different way than conventional processors, using many cores in a single chip (Guim and Rodero, 2015). Current GPUs have thousands of nuclei. For example, the RTX 4090 (Nvidia®, 2024) graphic card has 16,384 Cuda Cores, while the Radeon Rx 7900 XTX (AMD®, 2024) has 6,144 Stream Processors. These two graphic cards were used by the videogame industry in products for end users from 2022 to 2023.

Graphic Cards

Raya (2008) describes graphic cards as boards designed with the transmission of video to the computing equipment screen as their main objective. It can be visualized as the combination of the following components:

- GPU: the graphic processing unit
- Video memory (VRAM): the memory that handles all the visual information sent by the CPU.
- Heatsink: components designed with thermal conductivity materials that extract heat from the graphic chip and the memories.
- Fans: these components create a flow of air to extract heat from the chip and the memories.
- Feed and interface: the PCIe (Peripheral Component Interconnect Express) port handles the data communication with the rest of the computer (mainly the CPU); it consists of a set of data communication signal conductors (either serial, point-to-point, or "dedicated"). An additional port is required to power the graphic card.

Impact of Parallel Computing on Execution Times

To develop an example of the reduction of execution times through parallel programming, algorithms from applications that involve a high number of operations were selected. A matrix inversion algorithm was chosen, because the inversion of huge matrices is a common need in areas that require complex and accurate calculations, including:

- Simulations and modelling: used in physics and engineering to simulate and model complex systems (*e.g.*, fluid dynamics or quantum mechanics).
- Machine learning algorithms: used in deep neural networks to solve systems with linear equations and to optimize cost functions.
- Economy and finances: used in risk analysis and for the optimization of portfolios.
- Images and signal processing: used for decomposition and recomposition of signals and images (*e.g.*, the processing of medical images).

The objective of this article is to compare the sequential computing and parallel computing paradigms through a case study: a matrix inversion using lower-upper (LU) decomposition. Two implementations will be described below: one that used sequential computing and the other heterogenous computing with CuPy. The results will be analyzed based on their execution time and efficiency.

MATERIAL AND METHODS

This experimental research was focused on the comparison of two computing approaches applied to a matrix inversion: CPU sequential and GPU heterogenous.

Experimental Design

Two matrix inversion programs were developed in Python:

- Sequential implementation: based on LU decomposition, the random and time libraries were used to generate random matrices and to measure execution time, respectively.
- GPU implementation: the CuPy library was used to efficiently execute matrix operations in the GPU.

Each program was executed multiple times with matrices of various sizes to accurately compare their performance.

Scientific Devices and Equipment

- CPU: AMD Ryzen 5 7535HS
- GPU: NVIDIA GeForce RTX 4050 - 6 GB VRAM
- RAM memory: 16 GB DDR5
- Operating system: Windows 11
- Environments, Languages, and Libraries
- CUDA Toolkit: v12.6
- IDE: Spyder: 5.5.6
- Python: 3.12.4
- Libraries:
  - CuPy v13.3.0
  - Random
  - Time

Methods, Techniques, and Procedures

A method was subsequently developed to compare execution times, using parallel and sequential programming.

Matrix inversion was chosen because this algorithm involves many operations. Typically, the number of operations with the LU decomposition algorithm is O(n3). As the size of the matrix increases, execution times become exceedingly long and inversion is therefore an example that the use of parallel programming vastly diminishes execution time. This result is of major importance for processes that involve unbearably long execution times.

The process, using the "random" function of Python and CuPy, was executed for random square matrices of each of these dimensions: 100×100, 500×500, 1000×1000, 2000×2000 and 5000×5000.

Matrix inversion:

1. Sequential: a matrix inversion was implemented based on LU decomposition.

LU Decomposition Algorithm (Crout's Method)

LU decomposition requires finding two matrices (L and U), built in such a way that the comply with:

$$A = L \cdot U$$

Where: L is a lower triangular matrix; U is an upper triangular matrix with unit elements in the main diagonal (Figure 3).

Given a matrix A of order n, the coefficients of the L and U matrices are determined as follows:

For matrix L:

$$L[i][j] = a[i][j] - \sum(\text{k}=1 \text{ to } j\text{-}1)\, L[i][k] \cdot U[k][j]$$

where $j \leq i$, $i = 1, 2, 3, ..., n\text{-}1$

$$L = \begin{bmatrix} L_{11} & 0 & 0 & ... & 0 \\ L_{21} & L_{22} & 0 & ... & 0 \\ L_{31} & L_{32} & L_{33} & ... & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ L_{n1} & L_{n2} & L_{n3} & ... & L_{nn} \end{bmatrix} \qquad U = \begin{bmatrix} 1 & U_{12} & U_{13} & ... & U_{1n} \\ 0 & 1 & U_{23} & ... & U_{2n} \\ 0 & 0 & 1 & ... & U_{3}n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & ... & U_{nn} \end{bmatrix}$$

Figure 3. L and U matrices (Crout's Method), developed by Cortés-Rosas *et al.* (2019).

For matrix U:

$$U[\,i\,][\,j\,] = (\,a[\,i\,][\,j\,] - \sum(\,k=1 \text{ to } i\text{-}1\,)\,L[\,i\,][\,k\,] \cdot U[\,k\,]\,[\,j\,])\,/\,L[\,i\,][\,i\,]$$

where $i<j$, $i=1, 2, 3, ..., n\text{-}1$

Specific cases:

For the first column of L ($j=1$):

$$L[\,i\,][\,1\,] = a[\,i\,][\,1\,]$$

For the first line of U ($i=1$):

$$U[\,1\,][\,j\,] = a[\,1\,][\,j\,] \,/\, L[\,1\,][\,1\,]$$

Figure 4 illustrates de decomposition of the matrices using Crout's method for a 4×4 matrix. The lower triangular matrix L contains the elements $L[\,i\,][\,j\,]$ of L as shown above. Similarly, the upper triangular matrix U is formed by the corresponding elements $L[\,i\,][\,j\,]$ also show above. The main diagonal of the matrix consists of 1's.

The algorithm alternates the calculation of L columns with U lines to determine the direct values, avoiding uncertainties in the process. The algorithm developed in Python uses defined functions to address a matrix inversion in sequence (order $n*n$), applying the LU decomposition method (Annex 1).

2. GPU: "*cupy.linalg.inv*" was used to invert the matrices in GPU.

The cupy.linalg.inv function of CuPy was used to calculate the reverse of a matrix in the GPU. Internally, this function uses the CUDA (Compute Unified Device Architecture) library of NVIDIA to make the most of the parallel processing capacities of the GPUs. According to the official documentation (Community, Preferred Networks, Inc., 2015), its internal operation is the following:

A. Matrix preparation: The input matrix becomes a CuPy array, unless it already has that characteristic. CuPy arrays are similar to NumPy arrays, but they are stored in the GPU memory.

$$\begin{bmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{bmatrix} \cdot \begin{bmatrix} 1 & U_{12} & U_{13} & U_{14} \\ 0 & 1 & U_{23} & U_{24} \\ 0 & 0 & 1 & U_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Figure 4. $LU = A$ decomposition, developed by Cortés-Rosas *et al.* (2019).

B. Verification and validation: The squareness ($n \times n$) of the input matrix was confirmed, because only square matrices can be inverted.
C. LU factoring: It uses the LU factoring function (decomposition into lower and upper matrices), provided by cuSOLVER, a NVIDIA library for linear algebra in GPUs. LU factoring is a crucial step to solve linear equation systems and to calculate a matrix inversion.
D. Resolution of the linear system: Once the matrix has been decomposed into its LU components, the function uses cuSOLVER to solve the "$A \cdot X = I$" equation system, where A is the original matrix and I is the identity matrix. The result of this system is the inverse matrix of A.
E. Error control: Potential errors are verified during the process to determine, for example, if the matrix is singular (*i.e.*, it is no invertible).
F. Result return: Finally, the inverse matrix is returned as a CuPy array, which is stored in the GPU memory.

Annex 2 shows the implementation of the algorithm that solved the matrix inversion, using the CuPy library of Python (open-source code) for GPU accelerated computing.

Data Analysis

Comparison tests were carried out to determine the differences in execution times between both approaches, using the two codes shown in Annexes 1 and 2.

RESULTS AND DISCUSSION

The results of the execution of both programs are summarized in Table 1.

Both the sequential and the parallel process used the same algorithm. $O\left(\frac{n^3}{3}\right)$ floating point operations to compute the matrix U, and the same number for the matrix L. The solution of both ystems of equations to obtain the inverse of the original matrix requires an addition of $2n^2$ operations. The cardinalities indicated were obtained from Benítez and Brañas (1996).

The relation between the number of operations for matrices of two sizes (n1 and n2) is of order $n2/n1^3$. In the sequential process, this phenomenon is reflected in the length of the

Table 1. Execution times in seconds (s), inverting matrices with CPU sequential computing and GPU parallel computing.

Matrix size	CPU sequential	GPU with CuPy
100×100	0.21	0.29
500×500	23.99	0.26
1000×1000	198.81 (3 minutes)	0.28
2000×2000	1851.36 (30 minutes)	0.41
5000×5000	33231 (9 hours)	2.41

Table developed by the authors (2024).

said process; its impact on parallel computing is much lower. For n1 = 1,000 and n2 = 2,000, the increase in the sequential process is approximately nine-fold, while it increases by 1.5 in the parallel process (Figure 5).

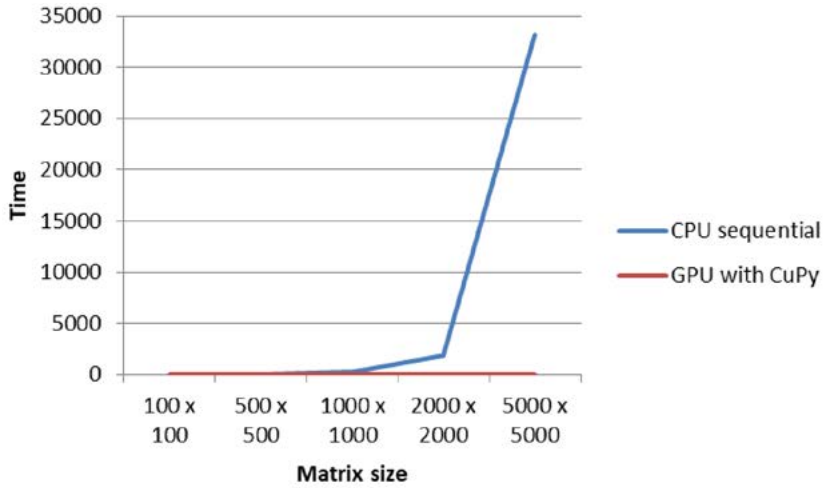


Figure 5. Execution times in seconds (s) resulting from the inversion of matrices using CPU sequential computing and GPU parallel computing. Figure developed by the authors (2024).

CONCLUSIONS

The results confirm that the use of parallel computing reduces the length of a given process from hours to seconds. In many cases, it will determine if the process is useful for the situation in question. For small matrices (100×100), the difference in time is minimal, as a consequence of the initial overload of the GPU. Evidently, as the size of the matrix to be inverted increases, the time reduction will be much more important.

Based on the comparison of the processes, the capacity of the GPU to execute multiple operations at the same time results in a highly significant reduction of the time required to process big data.

The comparison of CPU sequential computing with GPU heterogeneous computing for the matrix inversion in Python reveals that the use of GPU (specifically, the CuPy library), significantly improves performance. This optimization is particularly remarkable in the case of big matrices, where the reduction of the execution time can be critical.

Regardless of the type of computing employed, problems that require processing large amounts of data must take into account the size of the random access memory (RAM) and the video random access memory (VRAM), in the case of CPU and GPU, respectively, since these sizes could limit the resolution of complex problems, because the processed data and the outputs will be stored in the temporary memory. If the storage capability of the computing equipment is exceeded, its operations will stop, causing the applications and/or the operating system to "crash".

Reducing execution times is essential to address even more complex problems in several areas. The findings of this study stress the usefulness of heterogeneous computing for high performance computing in scientific and engineering contexts.

ACKNOWLEDGMENTS

The authors would like to thank the Colegio de Postgraduados and the CONAHCyT for the access to the resources necessary for this study, which was part of the Msc in Applied Computing of the Colegio de Postgraduados.

REFERENCES

Advanced Micro Devices, Inc. (2023). "RDNA3" Instruction Set Architecture. Advanced Micro Devices, Inc. Recuperado de https://www.amd.com/content/dam/amd/en/documents/radeon-tech-docs/instruction-set-architectures/rdna3-shader-instruction-set-architecture-feb-2023_0.pdf

Benítez, P. R. A., & Brañas, J. R. F. (1996). Eliminación gaussiana para sistemas de ecuaciones lineales. Recuperado de https://www.revista-educacion-matematica.org.mx/descargas/Vol10/1/08Almeida.pdf

Cortés Rosas, J. J., González Cárdenas, M. E., et. al. (2019). Métodos de descomposición LU: Crout y Doolittle. En Plataforma educativa para Análisis Numérico. UNAM-DGAPA-PAPIME PE105717. Recuperado de https://www.ingenieria.unam.mx/pinilla/PE105117/pdfs/tema3/3-2_descomposicion_lu.pdf

Community, Preferred Networks, Inc. (2015). CuPy – NumPy & SciPy for GPU. Cupy. Recuperado de https://docs.cupy.dev/en/stable/

Guim, F., & Rodero, I. (2019). Arquitecturas basadas en computación gráfica (GPU). Recuperado de: https://dspace.itsjapon.edu.ec/jspui/bitstream/123456789/400/1/Arquitecturas-basadas-en-computacion-grafica.pdf

Intel Corporation. (2024). CPU o GPU: opciones interesantes para sus necesidades informáticas. Intel Corporation Recuperado de: https://www.intel.la/content/www/xl/es/products/docs/processors/cpu-vs-gpu.html

Nvidia Corporation. (2023). NVIDIA ADA GPU ARCHITECTURE. Nvidia Corporation Recuperado de: https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.1.pdf

Piccoli, M. F.. (2011). Computación de alto desempeño en GPU. Argentina: Editorial de la Universidad Nacional de La Plata (Edulp). Recuperado de https://sedici.unlp.edu.ar/bitstream/handle/10915/18404/Documento_completo__.pdf?sequence=1&isAllowed=y

Raya L, (2008). El mundo de las tarjetas gráficas. *Manual formativo de ACTA* 50: pp 31-38. Recuperado de: https://www.acta.es/medios/articulos/informatica_y_computacion/050031.pdf

Rivares, C. O., & Evangelista, A. E. (2000). Procesamiento paralelo. Doctoral dissertation, Universidad Nacional de La Plata. https://sedici.unlp.edu.ar/bitstream/handle/10915/3867/Documento_completo__.pdf?sequence=1&isAllowed=y

Tanco, F., Verrastro, C., Grinberg, D., & Roitman, J. (2004). Implementación de redes neuronales artificiales en hardware para aplicación en detección automática de fulguraciones solares. Buenos Aires. Recuperado de: https://www.academia.edu/download/43957027/IMPLEMENTACIN_DE_REDES_NEURONALES_ARTIFI20160321-29197-7tdr47.pdf

Shen, H. (2024). Enhancing GPU performance and energy efficiency: Innovative strategies for sustainable computing. *Applied and Computational Engineering*, (49), 242-246, DOI: 10.54254/2755-2721/49/20241253 Recuperado de: https://www.researchgate.net/publication/379180571_Enhancing_GPU_performance_and_energy_efficiency_Innovative_strategies_for_sustainable_computing

ANNEXES

Annex 1. Matrix inversion using sequential programming with LU decomposition algorithm (Python).

```
def descomposicion_lu(A):
    n = len(A)
    L = [[0.0] * n for _ in range(n)]
    U = [[0.0] * n for _ in range(n)]

    for i in range(n):
        # Cálculo de L
        for j in range(i+1):
            suma = sum(L[i][k] * U[k][j] for k in
range(j))
            L[i][j] = A[i][j] - suma

        # Cálculo de U
        for j in range(i, n):
            if i == j:
                U[i][i] = 1.0  # Diagonal de U es 1
            else:
                suma = sum(L[i][k] * U[k][j] for k in
range(i))
                U[i][j] = (A[i][j] - suma) / L[i][i]

    return L, U

def sustitucion_adelante(L, b):
    n = len(L)
    y = [0.0] * n
    for i in range(n):
        suma = sum(L[i][j] * y[j] for j in range(i))
        y[i] = (b[i] - suma) / L[i][i]  # Dividimos
por L[i][i]
    return y

def sustitucion_atras(U, y):
    n = len(U)
    x = [0.0] * n
    for i in range(n-1, -1, -1):
        suma = sum(U[i][j] * x[j] for j in range(i+1,
n))
        x[i] = y[i] - suma  # U[i][i] es 1, así que
no dividimos
    return x

def invertir_matriz(A):
    n = len(A)
    L, U = descomposicion_lu(A)

    # Crear matriz identidad I
    I = [[float(i == j) for i in range(n)] for j in
range(n)]

    # Resolver para cada columna de la inversa
    A_inv = []
    for i in range(n):
        e = [fila[i] for fila in I]
        y = sustitucion_adelante(L, e)
        x = sustitucion_atras(U, y)
        A_inv.append(x)

    # Transponer el resultado para obtener el formato
correcto
    A_inv = [[A_inv[j][i] for j in range(n)] for i in
range(n)]

    return A_inv
```

Source: Annex developed by the authors (2024).

Annex 2. Matrix inversion using GPU parallel computing with LU decomposition algorithm from CuPy (Python).

```
import time
import cupy as cp

#Crea una matriz aleatoria en la GPU
dim=10000
print(f'la matriz creada sera de {dim} x {dim}')
a = cp.random.rand(dim, dim)
print("Matriz A:")
print(a)

#Función que permite contar el tiempo de ejecución
empieza_acontar=time.time()

#Calcular la inversa de la matriz en la GPU
a_inv = cp.linalg.inv(a)
print("Inversa de A:")
print(a_inv)

#Permite la sincronización de la gpu
#Se termina de contar el tiempo
cp.cuda.Stream.null.synchronize()
fin_contar=time.time()
tiempo=fin_contar-empieza_acontar

#Verificar la multiplicación de la matriz original con
su #inversa

identity = cp.dot(a, a_inv)
print("A * A_inv (debe ser la matriz identidad):")
print(identity)


print(f'la inversion y verificacion de la matriz se
realizo en : {tiempo} segundos')
```

Annex developed by the authors (2024).