UNIVERSIDAD
POLITÉCNICA
DE MADRID

CPU Performance: Benchmark Analysis and Theoretical Limits

CPU Performance
Centro de Cálculo Numérico

Autor: Juan Román Bermejo

Madrid, Septiembre de 2024

Índice

Resumen

Listado de cambios por hacer en el documento:

- Añadir gráficos de la ejecución de los programas en otras CPU´s
- Sobre la ultima subsección (IMSL): podría ser interesante comparar el producto de matrices con un producto de kronecker, que debería de tender más rápidamente al valor teórico esperado
- Completar pies de las figuras
- Incluir conclusiones (?)
- **Nota:** Algunas de las gráficas no son coherentes, se actualizarán al cambiar algunos matices en los códigos

1. Introduction

In recent years, the Graphics Processing Unit (GPU) has gained significant attention due to its parallel processing capabilities and its role in accelerating tasks such as machine learning, scientific simulations, and graphical rendering. While the rise of the GPU has shifted focus toward its impressive computational power, it is essential not to overlook the ongoing importance of the Central Processing Unit (CPU). CPUs remain the backbone of general-purpose computing, excelling in tasks that require sequential processing and complex logic.

This paper presents an exploration of CPU performance, beginning with a brief overview of key concepts such as clock speed, memory hierarchy, and instruction processing. Following this, we introduce a theoretical expression that defines the upper bound of CPU performance based on these characteristics. This expression serves as the basis for our subsequent benchmarks, which aim to push the CPU to its theoretical limits. The benchmarks assess performance across various tasks, focusing on how well the CPU handles large-scale computations and data processing. An additional focus is placed on the usability of data—a critical factor that significantly impacts CPU efficiency.

2. About the Hardware

The performance of a CPU (Central Processing Unit) depends on more than just its raw processing power. Both its architecture—how it's designed and organized—and the surrounding hardware play critical roles in its overall efficiency. Components such as memory or storage interact closely with the CPU, influencing how well it handles tasks. Additionally, understanding key concepts related to CPU architecture, like cores, threads, and cache, is essential for grasping the full picture of system performance.

In this section, we will explore both the architectural aspects of the CPU and the related hardware that together drive the performance of modern computing systems.

2.1. How CPU works

The CPU operates by **fetching** instructions and data from memory, which it **processes** using its registers—small, fast storage locations within the CPU. These registers temporarily hold data and instructions during processing, allowing the CPU to quickly access and manipulate information. The CPU uses a cycle of **fetch, decode, and execute** to perform operations, where it retrieves the necessary data from memory, decodes the instructions, and then executes them using the registers for efficient data handling.

It is important to distinguish between the functioning at the thread level and the core level. A CPU core typically has two threads, allowing it to handle multiple instructions concurrently through simultaneous multithreading (SMT). While each thread functions independently, sharing resources such as registers and execution units within the core, the overall performance and efficiency of the CPU are significantly influenced by how these threads interact and share the core's resources. The ability to manage tasks at both the core and thread levels is crucial for optimizing CPU performance, particularly in parallel computing environments.

2.2. Micro-operations and pipelining

Micro-operations Micro-operations, are the smaller instructions into which complex CPU instructions are broken down. Modern CPUs often deal with complex instructions that are not directly executable by the CPU's hardware. To manage this complexity, these instructions are divided into simpler operations known as micro-operations. The CPU can then execute them more efficiently using its execution units.

Pipelining Pipelining is a technique used in CPUs to improve instruction throughput—the number of instructions that can be processed in a unit of time. In a pipelined CPU, a single instruction is broken down into multiple stages (like fetching, decoding, executing, etc.), and these stages are processed in parallel for different instructions. However, this doesn't mean that different threads are assigned specific stages like fetch or decode. Instead: one thread can go through all the stages of the pipeline for different instructions over time. For example, while one instruction is being executed, the next instruction might be in the decode stage, and yet another instruction might be in the fetch stage, all within the same thread and the same pipeline.

2.3. Vectorization: AVX512

Vectorization is the process of transforming operations that are performed sequentially (one by one) into operations that can be performed simultaneously on multiple data points. This is achieved by processing "vectors.ºf data instead of processing a single value at a time.

For example, instead of adding two numbers at a time, a processor with vectorization can add several numbers simultaneously using a single instruction. This is known as SIMD (Single Instruction, Multiple Data), meaning one instruction operates on multiple data points in parallel.

AVX-512 is a technology that implements and enhances vectorization in CPUs. It works through SIMD instructions and introduces larger registers (512 bits) that allow more data to be handled at once in a single operation. For example, instead of performing an addition on just two numbers, AVX-512 can process 16 numbers of 32 bits or 8 numbers of 64 bits at the same time.

2.4. Memory: RAM

Random Access Memory (RAM) is a type of volatile memory that temporarily stores data and instructions needed by the CPU to perform tasks. RAM typically stores data in the order of gigabytes (GB), allowing the system to manage multiple active programs and processes simultaneously. This supports the smooth execution of complex operations.

However, RAM speed is slower than CPU speed, which can lead to bottlenecks. In such cases, the performance of the CPU is limited by the data transfer speed from the RAM. Some examples of RAM data transmission times, taken from the document [...], are:

1. Memory 3200 MHz, CL16: $\frac{16}{3200} \times 1000 \simeq 5[ms]$
2. Memory 4000 MHz, CL19: $\frac{19}{4000} \times 1000 \simeq 4{,}75[ms]$
3. Memory 2400 MHz, CL17: $\frac{17}{2400} \times 1000 \simeq 7{,}08[ms]$

2.5. Memory: Cache

In modern computing, the CPU is tasked with processing vast amounts of data and instructions at incredible speeds. However, retrieving this information from the main memory (RAM) can be relatively slow, creating a bottleneck that limits the CPU's performance. To bridge this gap and ensure the CPU can work as efficiently as possible, cache memory was introduced. Cache serves as a high-speed storage located closer to the CPU, designed to temporarily hold frequently accessed data and instructions. This reduces the time the CPU spends waiting for data retrieval, significantly improving system performance.

Modern CPUs use a multi-level cache hierarchy to enhance performance. Typically, there are three levels: L1, L2, and L3. L1 cache is the smallest but fastest and resides directly within the CPU cores. L2 cache is larger and slightly slower, while L3 is even bigger but still significantly faster than RAM. By utilizing these cache levels, CPUs can prioritize faster access to data that is more likely to be used again. The efficiency of this system ensures that the CPU spends less time waiting for data retrieval and more time processing information.

The following diagram illustrates the different levels of memory in a typical computer system, arranged in a pyramid to highlight the trade-offs between speed, cost, and capacity. At the top, CPU registers and cache memory (SRAM) are the fastest and most expensive per bit, but offer limited capacity. As we move down the pyramid, memory types like main memory (DRAM) and storage solutions such as magnetic disks and optical disks provide larger capacities but come with slower access times and lower costs per bit. This hierarchy demonstrates the balance between speed and storage, emphasizing why cache memory plays such a crucial role in optimizing CPU performance by acting as an intermediary between the extremely fast CPU registers and the slower but more abundant main memory and storage.

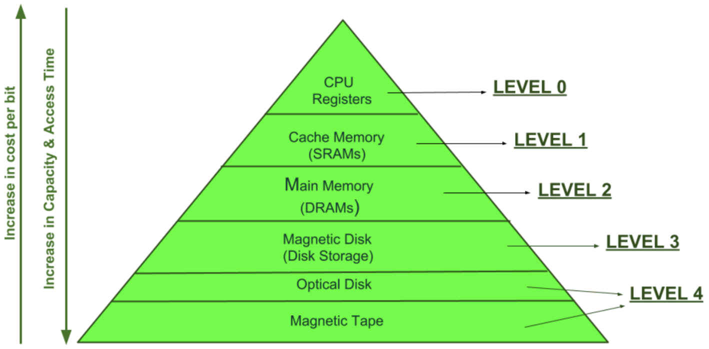


Figura 1: Representation of the hierarchy of different types of memory in a system.

3. Theoretical time

In this section, we discuss the concept of the ”theoretical time of a CPU”, which refers to the estimated time required for a CPU to complete a given task under ideal conditions. This measure assumes an optimal scenario, free from common real-world limitations such as memory latency, system bottlenecks, or the complexities introduced by parallel execution. By focusing on theoretical performance, we gain insights into the maximum potential of a CPU, providing a useful benchmark for evaluating its capabilities across various workloads.

Theoretical time allows us to break down CPU performance into fundamental parameters, helping us understand how different architectural features influence the speed of computation. This model is particularly valuable for comparing CPUs across generations or architectures, as it highlights the efficiency of vectorization, micro-operations, and core usage, among other factors. While real-world performance is often constrained by a variety of external factors, theoretical models like this one offer a clear, baseline perspective on the CPU's potential.

The following equation provides a framework for estimating the theoretical time a CPU would need to complete a specific set of operations:

$$t_{CPU} = \frac{N_{ops} \times S_{ops}}{V_{vectorization} \times GH_{z_{CPU}} \times M_{micro-ops} \times C_{CPU}}$$

Where the parameters are:

1. $V_{vectorization}$: Vectorization factor: 16 (512-bit)
2. $M_{micro-ops}$: Micro-operations factor: 4, 6 (AMD Zen 3) or even 8 (Apple Silicon)
3. $GH_{z_{CPU}}$: Clock speed of the CPU
4. C_{CPU}: Number of cores in the CPU
5. S_{ops}: Sequence of operations. For example, in the case of matrix multiplication, it would have a value of 4.
6. N_{ops}: Number of operations

4. Benchmark operation

In this section, we will discuss the operator used in the benchmarks: General Matrix Multiplication (GEMM) operation.

General Matrix Multiplication (GEMM) is the operation $C = AB + C$, where A and B are input matrices, and C is a pre-existing matrix overwritten by the result. For matrices of sizes $M \times K$ (A), $K \times N$ (B), and $M \times N$ (C), the product of A and B results in $M \times N$ values, each derived from a dot product of K elements. The total number of fused multiply-add operations (FMAs) required is $M \times N \times K$, and since each FMA consists of both a multiplication and an addition, the total number of floating-point operations (FLOPs) is $2 \times M \times N \times K$.

To particularize the expression for theoretical time, we substitute the sequence of operations factor S with 4, which corresponds to the four sequences of operations required for matrix multiplication:

1. Accessing values in matrix A.
2. Accessing values in matrix B.
3. Performing the multiplication.
4. Storing the results in matrix C.

4.1. Dot product function

The first step toward evaluating the speed of our CPU is to ensure that we are using optimized operators: this means that the operator, in this case the dot product, is able to utilize all available hardware resources and does not waste them. To achieve this, the dot product operator included by Julia (`matrix_multiplication`) is compared with a function that would perform the dot product in the most basic way (`my_matrix_multiplication`):

```
import Pkg
Pkg.activate(".")
Pkg.add( "PGFPlotsX" )
using CPUTime
using Plots
using LinearAlgebra, MKL
using PGFPlotsX

#Function to initialize random matrices
function matrix_initialization(N)

    A = rand(Float32, N, N )
    B = rand(Float32, N, N )
    return A, B
```

```

end

# Function to multiply matrices using the built-in Julia method
function matrix_multiplication(A,B)

   return A * B

end

# Function to multiply matrices using a custom method (manual loop)
function my_matrix_multiplication(A,B)

  (N, M) = size(A)
  (M, L) = size(B)

  C = zeros(Float32, (N, L) )

  for i in 1:N, j in 1:L
      for k in 1:M
        C[i,j] = C[i,j] + A[i,k]*B[k,j]
      end
  end
  return C

end

# Function to time matrix multiplication and calculate performance
function time_matrix_multilication(N, N_cores, matmul)

  Time = zeros( length(N) )

  for (i,n) in enumerate(N)

   A,B = matrix_initialization(n)

   t1= time_ns()

   matmul(A,B)

   t2 = time_ns()
   Time[i] = (t2-t1)/(2*n^3)

   #println("N=", n, " Time per operation =", Time[i] , " nsec")
  end

  return Time

end

# settings "julia.NumThreads": "auto"
N_threads = Threads.nthreads()
N_cores = div(N_threads, 2)
```

```
println("Threads =", N_threads )
println("Cores =", N_cores )

# Precompilation: Run matrix multiplication once to warm up
time_matrix_multilication(2000, N_cores, matrix_multiplication)

# Set range for matrix dimensions
N = 0:100:2500

# Set number of threads for BLAS operations (used by matrix multiplication)
BLAS.set_num_threads(2*N_cores)
println(" threads = ", BLAS.get_num_threads(), " N_cores =", N_cores )

# Time the built-in matrix multiplication and custom multiplication
Time = time_matrix_multilication(N, N_cores, matrix_multiplication)
Time2 = time_matrix_multilication(N, N_cores, my_matrix_multiplication)

# Calculate GFLOPS (floating point operations per second) for each method
GFLOPS = 1 ./ Time
GFLOPS2 = 1 ./ Time2

# Data for plotting
x = N
y1 = GFLOPS
y2 = GFLOPS2

# Create the plot using PGFPlotsX
plot = @pgf Axis(
    {
        xlabel="Matrix dimension [N]",
        ylabel="GFLOPS",
        title="Comparison of different dot functions",
        #legend="north east"
    },
    Plot({no_marks, "blue"}, Table(x, y1)),
    Plot({no_marks, "red"}, Table(x, y2)),
)

PGFPlotsX.save("/Users/juanromanbermejo/Desktop/documentacion_CPU/doc_latex/code/1-
    efficiency_dot_product/grafico_dot_func_comparison.tex", plot, include_preamble=false
    )
```

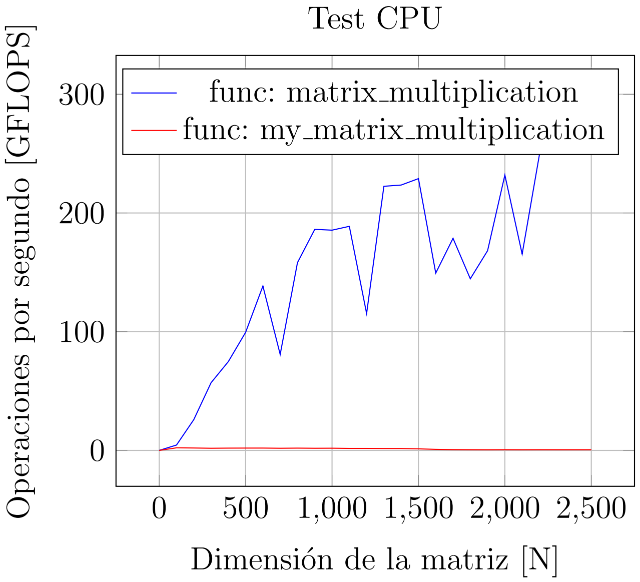


Figura 2: Eficiencia del producto de matrices, probado en una CPU 1,7 GHz Intel Core i7 de 4 núcleos

5. Benchmarks

5.1. Single-core, multi-core

In this section, we will explore the results of restricting CPU usage to a defined number of threads, comparing the performance of single-core versus multi-core execution. Ideally, one might expect a near doubling in performance when the number of threads is doubled, as the workload is theoretically spread across more processing units. However, in practice, the scaling is far from perfect. Various factors such as overhead from managing threads, memory access bottlenecks, or CPU architecture limitations can prevent the linear scaling we might anticipate. This experiment demonstrates these real-world constraints by showing the actual performance improvement as the number of threads increases.

```
import Pkg
Pkg.activate(".") # environment in this folder
Pkg.add( "Plots" )
Pkg.add( "CPUTime" )
#Pkg.add( "BLIS" )
Pkg.add( "MKL" )
Pkg.add( "PGFPlotsX" )
using CPUTime
using Plots
using LinearAlgebra
using PGFPlotsX


# Function to initialize two random matrices A and B of size N x N
function matrix_initialization(N)


  A = rand(Float32, N, N )
  B = rand(Float32, N, N )

  return A, B

end


# Function to perform matrix multiplication without timing (basic operation)
function matrix_multiplication(A,B)

  return A * B

end


# Function to time the matrix multiplication process and calculate time per operation
function time_matrix_multilication(N, N_cores, matinit, matmul)

  Time = zeros( length(N) )
  #Theoretical_time = 4e9/(4e9 * 512/32 * 4 * N_cores)
  Theoretical_time = 1e9/(4e9 * 512/32 * N_cores) # jahr

  for (i,n) in enumerate(N) # variables inside loop have local scope

   A,B = matinit(n)
   m = length(B)

   # dt = 1e9 * matmul(A,B)

   t1 = time_ns()
   matmul(A,B)
   t2 = time_ns()
   dt = t2-t1

   Time[i] = dt/(2*n*m)


   println("N=", n, " Time per operation =", Time[i] , " nsec")
   println("N=", n, " Theoretical time per operation =", Theoretical_time, " nsec")

  end

  return Time, Theoretical_time

end












function plot_combined(N_threads_range)
  # Initialize the vectors to store GFLOPS for each thread count
  y1 = Float32[]
  y2 = Float32[]
  y3 = Float32[]
  y4 = Float32[]
  N = Vector{Int}[] # To store the values of N (matrix dimensions)

  # Loop over the number of threads (N_threads) from 1 to 4
  for N_threads in N_threads_range
      N_threads = N_threads
      N_cores = N_threads
      println("Threads =", N_threads )
      println("Cores =", N_cores )

      # Define the matrix size range N
      N_matrix = Vector([10:10:2500; 2500:100:5000])
      BLAS.set_num_threads(N_cores) # Set the number of threads for BLAS operations
      println(" threads = ", BLAS.get_num_threads(), " N_cores =", N_cores )

      # Measure the time for matrix multiplication for each size N
      Time, Theoretical_time = time_matrix_multilication(N_matrix, N_cores, matrix_initialization, matrix_multiplication)
      GFLOPS = 1 ./ Time # Calculate GFLOPS (Giga Floating Point Operations Per Second)

      # Store the GFLOPS results in the respective vectors
      if N_threads == 1
          y1 = GFLOPS # Store in y1 if N_threads is 1
      elseif N_threads == 2
          y2 = GFLOPS # Store in y2 if N_threads is 2
      elseif N_threads == 3
          y3 = GFLOPS # Store in y3 if N_threads is 3
      elseif N_threads == 4
          y4 = GFLOPS # Store in y4 if N_threads is 4
      end

      # Ensure the N vector is stored only once
      if isempty(N)
          N = N_matrix
      end
  end

  # Return the data: N (matrix dimensions), y1, y2, y3, y4
  return N, y1, y2, y3, y4
end

# Extract the GFLOPS data for plotting
N, y1, y2, y3, y4 = plot_combined(1:4)

# Create the plot using PGFPlotsX
plot = @pgf Axis(
    {
        xlabel="Matrix dimension [N]",
        ylabel="GFLOPS",
        title="Comparison of different dot functions",
        legend_pos="north west", # Position of the legend
        legend_entries={"Threads = 1", "Threads = 2", "Threads = 3", "Threads = 4"}, # Add legend entries here
        ymin=0, # Set the minimum value for the y-axis
        ymax=400 # Set the maximum value for the y-axis (increase height)
    },
    Plot({no_marks, "blue"}, Table(N, y1)), # Plot for Threads = 1
    Plot({no_marks, "red"}, Table(N, y2)), # Plot for Threads = 2
    Plot({no_marks, "green"}, Table(N, y3)), # Plot for Threads = 3
    Plot({no_marks, "black"}, Table(N, y4)) # Plot for Threads = 4
)

# Save the plot in .tex format
PGFPlotsX.save("/Users/juanromanbermejo/Desktop/documentacion_CPU/doc_latex/code/2-
    singlecore_vs_multicore/n_threads.tex", plot, include_preamble=false)
```

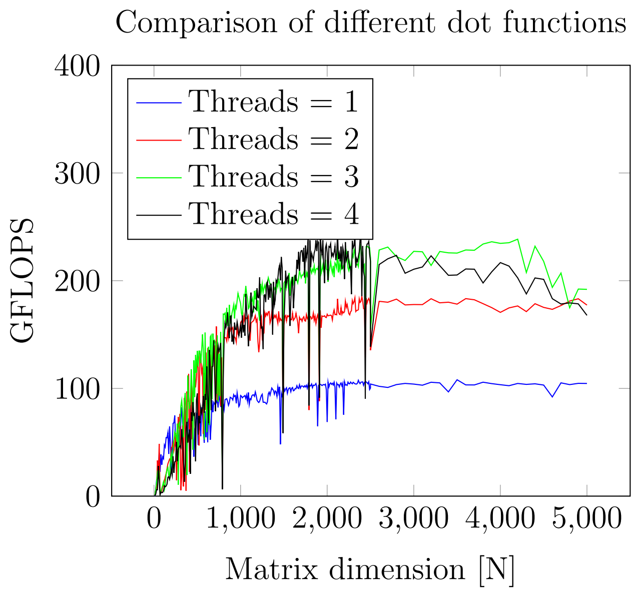


Figura 3:

Speed-up The following code investigates how performance scales as more threads are added, referred to as ”speed-up.”The goal here is to observe how performance increases with parallelism, plotted as a curve representing the speed-up as more cores are utilized. The expected behavior is not a linear speed-up—where the slope of the curve remains constant—but rather a diminishing rate of performance gain. This is because of the law of diminishing returns in parallel computing: as the number of threads increases, factors such as communication between threads, memory access contention, and overheads from parallelization begin to outweigh the benefits of adding more threads. Hence, while performance improves, the slope of the curve flattens, though it should never turn negative.

```
import Pkg
Pkg.activate(".")
Pkg.add( "PGFPlotsX" )
using PGFPlotsX
using LinearAlgebra, MKL
using Plots

N = 10_000
A = rand(Float32, N, N)
B = rand(Float32, N, N)

function matrix_multiplication(A, B)
    return A * B
```

```julia
end

N_threads = [1, 2, 3, 4, 5, 6]
times = Float64[]

matrix_multiplication(A, B)

# Calculation of the reference for speed-up
BLAS.set_num_threads(1)
reference_time = @elapsed matrix_multiplication(A, B)

# Calculation of speed-up for different numbers of threads
speedups = Float64[]
for (i, threads) in enumerate(N_threads)
    BLAS.set_num_threads(threads)

    t = @elapsed matrix_multiplication(A, B)
    push!(times, t)
    speedup = reference_time / t
    push!(speedups, speedup)
    println("Threads: $threads, Time: $t, Speedup: $speedup")
end


x = N_threads
y = speedups

plot = @pgf Axis(
    {
        xlabel="Number of threads in use",
        ylabel="Speedup factor",
        title="Speedup",
        #legend="north east"
    },
    Plot({no_marks, "blue"}, Table(x, y)),
)

PGFPlotsX.save("/Users/juanromanbermejo/Desktop/documentacion_CPU/doc_latex/code/3-speed-
    up/speed-up.tex", plot, include_preamble=false)
```

5.2. Theoretical time

In this part, we display the trend towards the theoretically expected execution time as the size of the matrices increases. This code shows how, as the dimensions of the matrices grow, the observed execution times begin to approach these theoretical predictions.

```julia
import Pkg
Pkg.activate(".")
Pkg.add( "PGFPlotsX" )
using CPUTime
using Plots
```

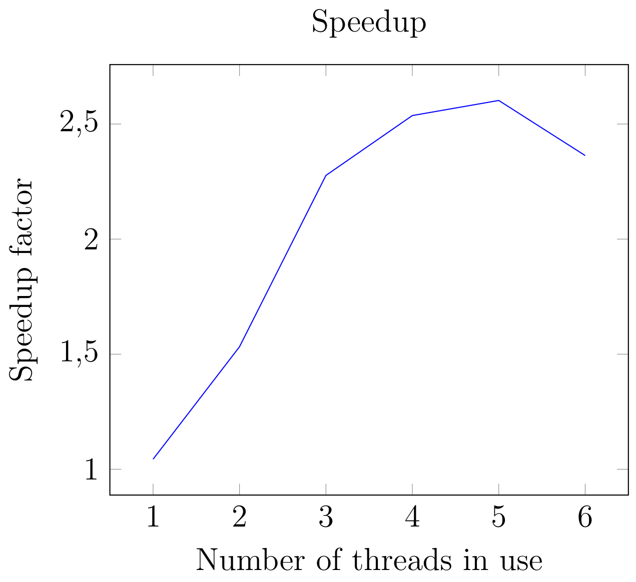
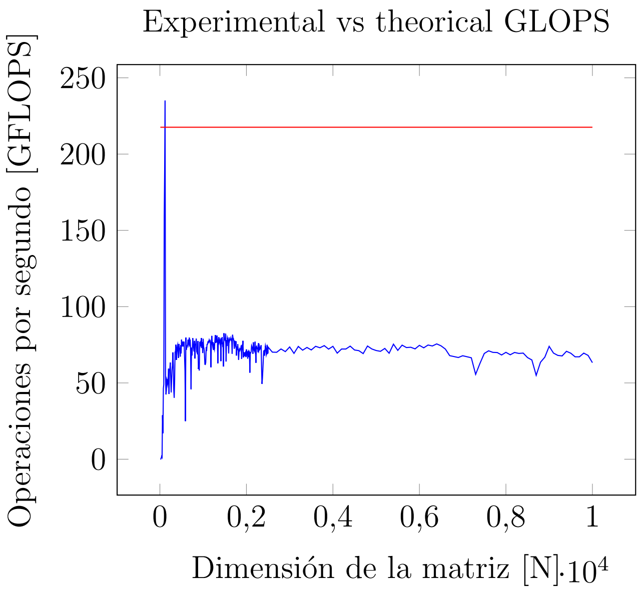


Figura 4:

```
using LinearAlgebra, MKL
using PGFPlotsX


function matrix_initialization(N)

    A = rand(Float32, N, N )
    B = rand(Float32, N, N )
    return A, B

end


function matrix_multiplication(A,B)

   return A * B

end


function time_matrix_multilication(N, N_cores, matmul)

  Time = zeros( length(N) )
  #Theoretical_time = 4e9/(4e9 * 512/32 * 4 * N_cores)
  Theoretical_time = 2e9/(1.7e9 * 512/32 * 2 * N_cores)

  for (i,n) in enumerate(N) # variables inside loop have local scope

   A,B = matrix_initialization(n)

```

```
    t1= time_ns()

    matmul(A,B)

    t2 = time_ns()
    Time[i] = (t2-t1)/(2*n^3)

    println("N=", n, " Time per operation =", Time[i] , " nsec")
    println("N=", n, " Theoretical time per operation =", Theoretical_time, " nsec")

  end

  return Time, Theoretical_time

end

# settings "julia.NumThreads": "auto"
N_threads = Threads.nthreads()
N_cores = div(N_threads, 2)
println("Threads =", N_threads )
println("Cores =", N_cores )
time_matrix_multilication(2000, N_cores, matrix_multiplication)

N = Vector([10:10:2500; 2500:250:10000])
BLAS.set_num_threads(2*N_cores)
println(" threads = ", BLAS.get_num_threads(), " N_cores =", N_cores )
Time, Theoretical_time = time_matrix_multilication(N, N_cores, matrix_multiplication)
GFLOPS = 1 ./ Time
GFLOPS_max = 1 / Theoretical_time

x = N
y1 = GFLOPS
y2 = GFLOPS_max
y2_vector= y2*ones(281)

plot = @pgf Axis(
    {
        xlabel="Matrix dimension [N]",
        ylabel="Operations per second [GFLOPS]",
        title="Experimental vs theorical GLOPS",
        #legend="north east"
    },
    Plot({no_marks, "blue"}, Table(x, y1)),
    Plot({no_marks, "red"}, Table(x, y2_vector)),
)

PGFPlotsX.save("/Users/juanromanbermejo/Desktop/documentacion_CPU/doc_latex/code/3-matmul_vs_theoretical-time/matmul_vs_theoretical-time.tex", plot, include_preamble=false)

#display( plot(N, GFLOPS, ylims=(0, 5000), title="GFLOPS", minorgrid=true ) )
#display( plot!(N, GFLOPS_max *ones( length(N) ), minorgrid=true ) )
```



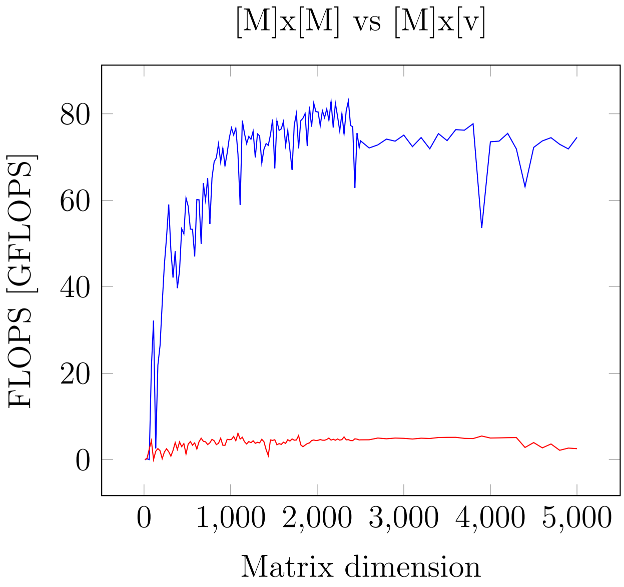
5.3. IMSL levels

At first glance, one might expect that matrix-vector multiplication would behave similarly to matrix-matrix multiplication in terms of performance trends, given that both involve the multiplication of matrix elements. However, the relationship between input data and the number of operations is more significant in the matrix-vector case. The ratio of memory accesses to computational operations is higher for matrix-vector multiplication compared to matrix-matrix multiplication. This means that a greater number of memory accesses are required for the same amount of CPU work, leading to longer computation times and a decrease in FLOPS (floating-point operations per second). This section delves into why this disparity occurs, explaining the memory bandwidth limitations and their impact on overall computational performance.

```
import Pkg
Pkg.activate(".")
Pkg.add( "PGFPlotsX" )
using CPUTime
using Plots
using LinearAlgebra, MKL
using PGFPlotsX

# Function to initialize random matrices of size N x N
function matrix_initialization(N)


  A = rand(Float32, N, N )
  B = rand(Float32, N, N )

  return A, B

end
```

```

# Function to initialize a random matrix and a vector (N x 1)
function matrix_vector_initialization(N)


    A = rand(Float32, N, N )
    B = rand(Float32, N, 1 )

    return A, B

end

# Function for vector multiplication (dot product)
function vector_multiplication(A,B)

    return dot(A, B)

end

# Function for matrix multiplication
function matrix_multiplication(A,B)

    return A * B

end

# Function to time matrix multiplication operations
function time_matrix_multilication(N, N_cores, matinit, matmul)

    Time = zeros( length(N) )
    Theoretical_time = 1e9/(4e9 * 512/32 * N_cores)

    for (i,n) in enumerate(N)

     A,B = matinit(n)

     t1 = time_ns()
     matmul(A,B)
     t2 = time_ns()
     dt = t2-t1

     Time[i] = dt/(2*n^3)

     println("N=", n, " Time per operation =", Time[i] , " nsec")
     println("N=", n, " Theoretical time per operation =", Theoretical_time, " nsec")

    end

    return Time, Theoretical_time

  end

# Function to time matrix-vector multiplication operations
```

```
function time_matrix_vector_multilication(N, N_cores, matinit, matmul)

    Time2 = zeros( length(N) )

    for (i,n) in enumerate(N)

     A,B = matinit(n)

     t1 = time_ns()
     matmul(A,B)
     t2 = time_ns()
     dt = t2-t1

     Time2[i] = dt/(2*n^2)

     println("N=", n, " Time per operation =", Time2[i] , " nsec")
     println("N=", n, " Theoretical time per operation =", Theoretical_time, " nsec")

    end

    return Time2

end

# Number of cores
N_cores = 4

# Range of matrix dimensions to test
N = Vector([10:25:2500; 2500:100:5000])

# Set the number of BLAS threads based on the number of cores
BLAS.set_num_threads(2*N_cores)
println(" threads = ", BLAS.get_num_threads(), " N_cores =", N_cores )

# Time the matrix multiplication and matrix-vector multiplication operations
Time, Theoretical_time = time_matrix_multilication(N, N_cores, matrix_initialization, matrix_multiplication)
Time2 = time_matrix_vector_multilication(N, N_cores, matrix_vector_initialization, matrix_multiplication)

# Calculate GFLOPS (floating-point operations per second)
GFLOPS = 1 ./ Time
GFLOPS2 = 1 ./ Time2
GFLOPS_max = 1 / Theoretical_time

# Data for plotting
x = N
y1 = GFLOPS
y2 = GFLOPS2


plot = @pgf Axis(
    {
        xlabel="Matrix dimension",
        ylabel="FLOPS [GFLOPS]",
        title="[M]x[M] vs [M]x[v]",
        #legend="north east"
    },
    Plot({no_marks, "blue"}, Table(x, y1)),
    Plot({no_marks, "red"}, Table(x, y2)),
)

PGFPlotsX.save("/Users/juanromanbermejo/Desktop/documentacion_CPU/doc_latex/code/5-IMSL_levels/1_IMSL_levels.tex", plot, include_preamble=false)
```


[M]x[M] vs [M]x[v]
FLOPS [GFLOPS]
Matrix dimension