CPU Performance: Benchmark Analysis and Theoretical Limits

CPU Performance
Centro de Cálculo Numérico

Autor: Juan Román Bermejo

Madrid, Septiembre de 2024

Índice

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: AVX

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.
In addition to AVX-512, processors typically include AVX or AVX2, both of which feature 256-bit registers, half the size of AVX-512 registers. To check if your processor supports AVX, AVX2, or AVX-512, you can run the following Julia code to display your processor's specifications:

```
Pkg.add("CpuId")
using CpuId

# View all processor features
cpuid = cpuinfo()
```

In your terminal, you will be able to see whether your registers are 256 bits (indicating AVX or AVX2) or 512 bits (indicating AVX-512). Additionally, if you are working from a laptop, you may also notice a Turbo Boost value, which refers to a different GHz value. This is the value that will be used in future theoretical calculations.

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.

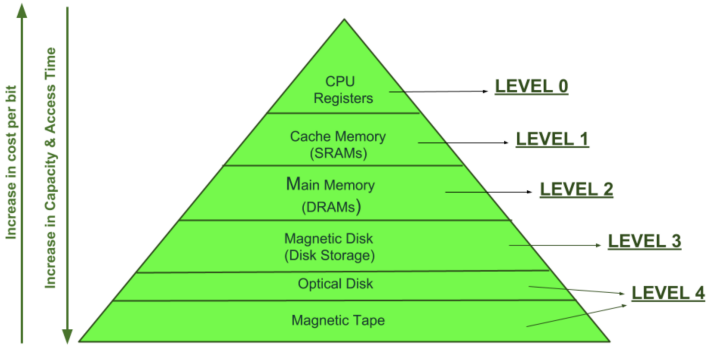

Figura 1: Representation of the hierarchy of different types of memory in a system.

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:

$$t_{CPU} = \frac{N_{ops} \times S_{ops}}{V_{vectorization} \times GH_{z_{CPU}} \times M_{micro-ops} \times C_{CPU}} \tag{1}$$

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.

The next subsection will include the theoretical runtime plotted across all graphs, using the expression 1.

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`). Additionally, it will also be compared with another basic function that is slightly optimized (`my_efficient_matrix_multiplication`).

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

# Transposing B for efficient memory access
function my_efficient_matrix_multiplication(A, B)
  (N, M) = size(A)
  (M, L) = size(B)
  BT = transpose(B)
  C = zeros(Float32, (N, L))

  for k in 1:M
      for j in 1:L, i in 1:N
          C[i, j] = C[i, j] + A[i, k] * BT[j, k]
      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) )
```

```
62
63    for (i,n) in enumerate(N)
64
65     A,B = matrix_initialization(n)
66
67     t1= time_ns()
68
69     matmul(A,B)
70
71     t2 = time_ns()
72     Time[i] = (t2-t1)/(2*n^3)
73
74     #println("N=", n, " Time per operation =", Time[i] , " nsec")
75    end
76
77    return Time
78
79  end
80
81  # settings "julia.NumThreads": "auto"
82  # en bash: $ JULIA_NUM_THREADS=4 julia
83  BLAS.set_num_threads(8)
84  N_threads = BLAS.get_num_threads()
85  N_cores = div(N_threads, 2)
86  println("Threads =", N_threads )
87  println("Cores =", N_cores )
88
89  # Precompilation: Run matrix multiplication once to warm up
90  time_matrix_multilication(2000, N_cores, matrix_multiplication)
91
92  # Set range for matrix dimensions
93  N = 0:100:2500
94
95  # Set number of threads for BLAS operations (used by matrix multiplication)
96  BLAS.set_num_threads(2*N_cores)
97  println(" threads = ", BLAS.get_num_threads(), " N_cores =", N_cores )
98
99  # Time the built-in matrix multiplication and custom multiplication
100 Time = time_matrix_multilication(N, N_cores, matrix_multiplication)
101 Time2 = time_matrix_multilication(N, N_cores, my_matrix_multiplication)
102 Time3 = time_matrix_multilication(N, N_cores, my_efficient_matrix_multiplication)
103
104 # Calculate GFLOPS (floating point operations per second) for each method
105 GFLOPS = 1 ./ Time
106 GFLOPS2 = 1 ./ Time2
107 GFLOPS3 = 1 ./ Time3
108
109 # Data for plotting
110 x = N
111 y1 = GFLOPS
112 y2 = GFLOPS2
113 y3 = GFLOPS3
114
```

```
# Create the plot using PGFPlotsX
plot = @pgf Axis(
    {
        width = "10cm",
        height = "7cm",
        xlabel="Matrix dimension [N]",
        ylabel="GFLOPS",
        title="Comparison of different dot functions",
        legend="north east",
        ymax=500,
        #ymode="log",


    },
    Plot({no_marks, "blue"}, Table(x, y1)),
    Plot({no_marks, "red"}, Table(x, y2)),
    Plot({no_marks, "green"}, Table(x, y3)),
    LegendEntry("Julia dot product"),
    LegendEntry("Manual dot product"),
    LegendEntry("Optimized dot product"),
)

PGFPlotsX.save("/Users/juanroman/Desktop/manual_vs_optimized_vs_julia_dot.tex", plot,
    include_preamble=false)
```

The results of running this code are shown in the following two figures; in the first one 2 you can see the difference between the functions `my_matrix_multiplication` and `my_efficient_matrix_multiplication`. This difference lies in the transpose of the B matrix. This is because in Julia matrices are stored in column order, that is, consecutive columns are stored contiguously in memory. Therefore, when iterating over the elements of a matrix, it is more efficient to traverse it by columns than by rows.

Now, by changing the dimension of the y-axis, we can see the comparison with the function `matrix_multiplication` in Figure 3. This clearly shows the level of optimization that Julia's built-in dot product has, and therefore, this will be the function used in the remaining tests.

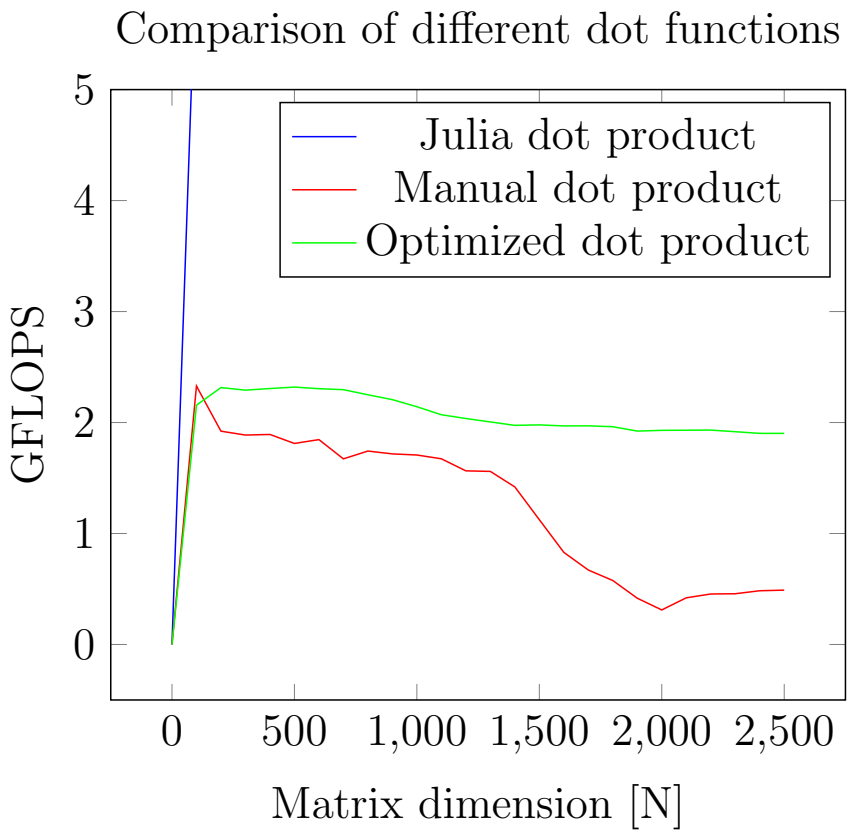

Figura 2: Matrix product efficiency, tested on a Intel(R) Core(TM) i7-8557U CPU @ 1.70GHz (1)

4.2. Comparison of Matrix-Vector Product and Matrix-Matrix Product

Before proceeding to test the CPU using matrix multiplication, it is logical to first consider the optimal shape and dimensions of these matrices. One might intuitively assume that a matrix-vector product is faster than a matrix-matrix product. To visualize the load that the CPU experiences in both cases, the following code is used to plot the figures.

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

# Ver todas las caracteristicas del procesador
cpuid = cpuinfo()
string_cpuid = string(cpuid)
```

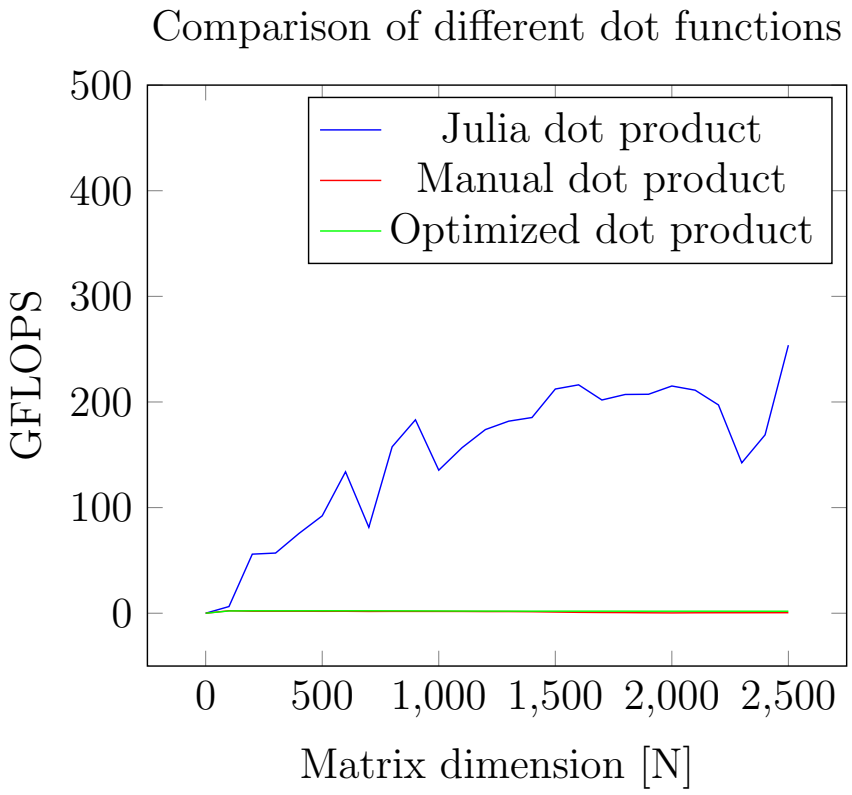

Figura 3: Matrix product efficiency, tested on a Intel(R) Core(TM) i7-8557U CPU @ 1.70GHz (2)

```
13
14 # Comprobar si AVX, AVX2 o AVX-512 estan soportados
15 println("AVX support: ", occursin("256", string_cpuid))
16 println("AVX-512 support: ", occursin("512 bit", string_cpuid))
17
18 # Function to initialize random matrices of size N x N
19 function matrix_initialization(N)
20
21   A = rand(Float32, N, N )
22   B = rand(Float32, N, N )
23
24   return A, B
25
26 end
27
28 # Function to initialize a random matrix and a vector (N x 1)
29 function matrix_vector_initialization(N)
30
31
32     A = rand(Float32, N, N )
33     B = rand(Float32, N, 1 )
34
```

```
35      return A, B
36
37  end
38
39  # Function to initialize a random matrix and a vector (N x 1)
40  function vector_vector_initialization(N)
41
42
43      A = rand(Float32, N, 1 )
44      B = rand(Float32, N, 1 )
45
46      return A, B
47
48  end
49
50  # Function for vector multiplication (dot product)
51  function vector_multiplication(A,B)
52
53      return dot(A, B)
54
55  end
56
57  # Function for matrix multiplication
58  function matrix_multiplication(A,B)
59
60      return A * B
61
62  end
63
64  # Function for matrix multiplication
65  function vector_multiplication(A,B)
66
67      return transpose(A) * B
68
69  end
70
71  # Function to time matrix multiplication operations
72  function time_matrix_multiplication(N, N_cores, matinit, matmul, AVX_value)
73
74      Time = zeros( length(N) )
75      Theoretical_time = 1e9/(4e9 * 512/32 * N_cores)
76      #Se considera que solo se necesita 1 instruccion para FMA
77      Theoretical_time = 1e9 /(4.5e9 * AVX_value * 2 * N_cores)
78      #Theoretical_time = 2e9/(1.7e9 * 512/32 * 2 * N_cores)
79
80      for (i,n) in enumerate(N)
81
82       A,B = matinit(n)
83
84       t1 = time_ns()
85       matmul(A,B)
86       t2 = time_ns()
87       dt = t2-t1
```

```
88
89      Time[i] = dt/(2*n^3)
90
91      println("N=", n, " Time per operation =", Time[i] , " nsec")
92      println("N=", n, " Theoretical time per operation =", Theoretical_time, " nsec")
93
94     end
95
96     return Time, Theoretical_time
97
98   end
99
100
101 function get_avx_value(string_cpuid)
102     # Inicializar la variable AVX_Value
103     AVX_value = 0
104
105     # Buscar el size del vector SIMD en la cadena y asignar el valor correspondiente
106     if occursin("256 bit", string_cpuid)
107         AVX_value = 8
108     elseif occursin("512 bit", string_cpuid)
109         AVX_value = 16
110     else
111         AVX_value = 0
112     end
113
114     return AVX_value
115 end
116
117 AVX_value = get_avx_value(string_cpuid)
118
119
120 # Function to time matrix-vector multiplication operations
121 function time_matrix_vector_multiplication(N, N_cores, matinit, matmul)
122
123     Time2 = zeros( length(N) )
124
125     for (i,n) in enumerate(N)
126
127      A,B = matinit(n)
128
129      t1 = time_ns()
130      matmul(A,B)
131      t2 = time_ns()
132      dt = t2-t1
133
134      Time2[i] = dt/(2*n^2)
135
136      println("N=", n, " Time per operation =", Time2[i] , " nsec")
137      println("N=", n, " Theoretical time per operation =", Theoretical_time, " nsec")
138
139     end
140
```

```
141     return Time2
142 
143 end
144 
145 # Function to time matrix-vector multiplication operations
146 function time_vector_vector_multiplication(N, N_cores, matinit, matmul)
147 
148     Time3 = zeros( length(N) )
149 
150     for (i,n) in enumerate(N)
151 
152      A,B = matinit(n)
153 
154      t1 = time_ns()
155      matmul(A,B)
156      t2 = time_ns()
157      dt = t2-t1
158 
159      Time3[i] = dt/(2*n)
160 
161      println("N=", n, " Time per operation =", Time3[i] , " nsec")
162      println("N=", n, " Theoretical time per operation =", Theoretical_time, " nsec")
163 
164     end
165 
166     return Time3
167 
168 end
169 
170 # Number of cores
171 N_cores = 4
172 
173 # Range of matrix dimensions to test
174 N = Vector([10:25:2500; 2500:100:5000])
175 
176 # Set the number of BLAS threads based on the number of cores
177 BLAS.set_num_threads(2*N_cores)
178 println(" threads = ", BLAS.get_num_threads(), " N_cores =", N_cores )
179 
180 # Time the matrix multiplication and matrix-vector multiplication operations
181 Time, Theoretical_time = time_matrix_multiplication(N, N_cores, matrix_initialization,
        matrix_multiplication, AVX_value)
182 Time2 = time_matrix_vector_multiplication(N, N_cores, matrix_vector_initialization,
        matrix_multiplication)
183 Time3 = time_vector_vector_multiplication(N, N_cores, vector_vector_initialization,
        vector_multiplication)
184 # Calculate GFLOPS (floating-point operations per second)
185 GFLOPS = 1 ./ Time
186 GFLOPS2 = 1 ./ Time2
187 GFLOPS3 = 1 ./ Time3
188 GFLOPS_max = 1 / Theoretical_time
189 
190 # Data for plotting
```

```
x = N
y1 = GFLOPS
y2 = GFLOPS2
y3 = GFLOPS3
y4 = fill(GFLOPS_max, length(y1))


plot = @pgf Axis(
    {
        xlabel="Matrix dimension",
        ylabel="FLOPS [GFLOPS]",
        title="[M]x[M] vs [M]x[v]",
        legend="north east",
        ymax=500
    },
    Plot({no_marks, "blue"}, Table(x, y1)),
    Plot({no_marks, "red"}, Table(x, y2)),
    Plot({no_marks, "green"}, Table(x, y3)),
    Plot({no_marks, "orange"}, Table(x, y4)),
    LegendEntry("Matmul"),
    LegendEntry("MatVec"),
    LegendEntry("VecVec"),
    LegendEntry("Theoretical"),
)

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

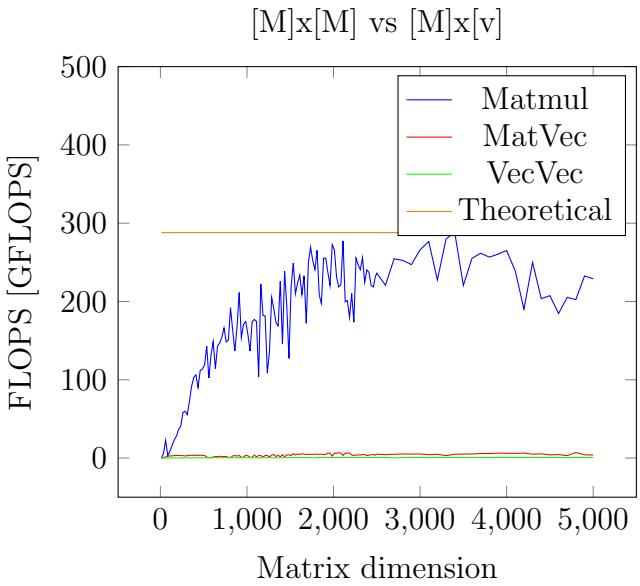

Figura 4: ..

Figure 4 illustrates the inherent limitation in matrix-vector multiplication (which is not due to CPU capacity but rather a bottleneck issue). This limitation arises because the "usability" of data in a matrix-matrix operation is higher than in a matrix-vector operation. Consider the following example with N:

$$\begin{bmatrix} a_{11} & \dots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \dots & a_{NN} \end{bmatrix} \begin{bmatrix} b_{11} & \dots & b_{1N} \\ \vdots & \ddots & \vdots \\ b_{N1} & \dots & b_{NN} \end{bmatrix} = \begin{bmatrix} c_{11} & \dots & c_{1N} \\ \vdots & \ddots & \vdots \\ c_{N1} & \dots & c_{NN} \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} \alpha_{11} & \dots & \alpha_{1N} \\ \vdots & \ddots & \vdots \\ \alpha_{N1} & \dots & \alpha_{NN} \end{bmatrix} \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_N \end{bmatrix} = \begin{bmatrix} \gamma_1 \\ \vdots \\ \gamma_N \end{bmatrix} \tag{3}$$

In this example, vector $\vec{a}_1 = \sum_{i=1}^{N} a_{1i}\vec{e}_i$ is used N times to compute N values ($\sum_{i=1}^{N} c_{i1}\vec{e}_i$). In contrast, the vector of elements $\vec{\alpha}_1 = \sum_{i=1}^{N} \alpha_{1i}\vec{e}_i$ is only used once (to compute γ_1).

We can define the term üsability.ªs the ratio between the number of operations performed by the CPU and the number of data elements (in this case, Float32) used during the process. This can be expressed as:

$$U = \frac{N_{ops}}{N_{data}} \tag{4}$$

where N_{ops} represents the number of operations executed by the CPU, and N_{data} denotes the number of data elements involved in the process.

For matrix multiplication of dimension N, considering the use of Fused Multiply-Add (FMA), we have $N_{ops} = N^3$ and $N_{data} = 2N^2$. This yields a usability value greater than 1.

In the case of matrix-vector multiplication, again with dimension N (as shown in expression 3), $N_{ops} = N^2$ and $N_{data} = N^2 + N$. Here, the usability value is approximately 1.

As the usability value tends towards infinity, the CPU's performance approaches its theoretical maximum.

Desfasado This difference in "usability" means that the values in matrix-matrix multiplication are recycled more efficiently. We can reuse the same data—which were obtained through slower methods, such as memory access—more times. As N approaches infinity, the CPU performance tends to converge toward its theoretical maximum.

Scalar product It is worth noting that the graph 4 also includes the vector-vector product. As expected, the results are even worse. The value of **U** is less than 1 ($N_{ops} = N$ and $N_{data} = 2N$)

Therefore, the tests conducted in the following section will focus on matrix multiplication.

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) )
```

```
38    #Theoretical_time = 4e9/(4e9 * 512/32 * 4 * N_cores)
39    Theoretical_time = 1e9/(4e9 * 512/32 * N_cores) # jahr
40
41    for (i,n) in enumerate(N) # variables inside loop have local scope
42
43     A,B = matinit(n)
44     m = length(B)
45
46     # dt = 1e9 * matmul(A,B)
47
48     t1 = time_ns()
49     matmul(A,B)
50     t2 = time_ns()
51     dt = t2-t1
52
53     Time[i] = dt/(2*n*m)
54
55
56     println("N=", n, " Time per operation =", Time[i] , " nsec")
57     println("N=", n, " Theoretical time per operation =", Theoretical_time, " nsec")
58
59    end
60
61    return Time, Theoretical_time
62
63  end
64
65
66
67
68
69
70
71
72
73
74
75  function plot_combined(N_threads_range)
76    # Initialize the vectors to store GFLOPS for each thread count
77    y1 = Float32[]
78    y2 = Float32[]
79    y3 = Float32[]
80    y4 = Float32[]
81    N = Vector{Int}[] # To store the values of N (matrix dimensions)
82
83    # Loop over the number of threads (N_threads) from 1 to 4
84    for N_threads in N_threads_range
85        N_threads = N_threads
86        N_cores = N_threads
87        println("Threads =", N_threads )
88        println("Cores =", N_cores )
89
90        # Define the matrix size range N
```

```
91      N_matrix = Vector([10:10:2500; 2500:100:5000])
92      BLAS.set_num_threads(N_cores) # Set the number of threads for BLAS operations
93      println(" threads = ", BLAS.get_num_threads(), " N_cores =", N_cores )
94
95      # Measure the time for matrix multiplication for each size N
96      Time, Theoretical_time = time_matrix_multilication(N_matrix, N_cores,
            matrix_initialization, matrix_multiplication)
97      GFLOPS = 1 ./ Time # Calculate GFLOPS (Giga Floating Point Operations Per Second)
98
99      # Store the GFLOPS results in the respective vectors
100     if N_threads == 1
101         y1 = GFLOPS # Store in y1 if N_threads is 1
102     elseif N_threads == 2
103         y2 = GFLOPS # Store in y2 if N_threads is 2
104     elseif N_threads == 3
105         y3 = GFLOPS # Store in y3 if N_threads is 3
106     elseif N_threads == 4
107         y4 = GFLOPS # Store in y4 if N_threads is 4
108     end
109
110     # Ensure the N vector is stored only once
111     if isempty(N)
112         N = N_matrix
113     end
114   end
115
116   # Return the data: N (matrix dimensions), y1, y2, y3, y4
117   return N, y1, y2, y3, y4
118 end
119
120 # Extract the GFLOPS data for plotting
121 N, y1, y2, y3, y4 = plot_combined(1:4)
122
123 # Create the plot using PGFPlotsX
124 plot = @pgf Axis(
125     {
126         xlabel="Matrix dimension [N]",
127         ylabel="GFLOPS",
128         title="Comparison of different dot functions",
129         legend_pos="north west", # Position of the legend
130         legend_entries={"Threads = 1", "Threads = 2", "Threads = 3", "Threads = 4"}, # Add
                legend entries here
131         ymin=0, # Set the minimum value for the y-axis
132         ymax=400 # Set the maximum value for the y-axis (increase height)
133     },
134     Plot({no_marks, "blue"}, Table(N, y1)), # Plot for Threads = 1
135     Plot({no_marks, "red"}, Table(N, y2)), # Plot for Threads = 2
136     Plot({no_marks, "green"}, Table(N, y3)), # Plot for Threads = 3
137     Plot({no_marks, "black"}, Table(N, y4)) # Plot for Threads = 4
138 )
139
140 # Save the plot in .tex format
141 PGFPlotsX.save("/Users/juanromanbermejo/Desktop/documentacion_CPU/doc_latex/code/2-
```

```
    singlecore_vs_multicore/n_threads.tex", plot, include_preamble=false)
```

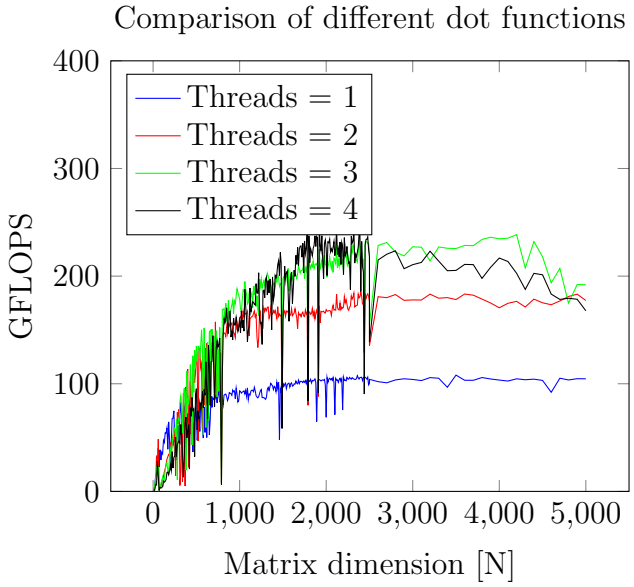

Figura 5:

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.

```
1  import Pkg
2  Pkg.activate(".")
3  Pkg.add( "PGFPlotsX" )
4  using PGFPlotsX
5  using LinearAlgebra, MKL
6  using Plots
7
8  N = 10_000
9  A = rand(Float32, N, N)
10 B = rand(Float32, N, N)
11
12 function matrix_multiplication(A, B)
13     return A * B
```

```
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.

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

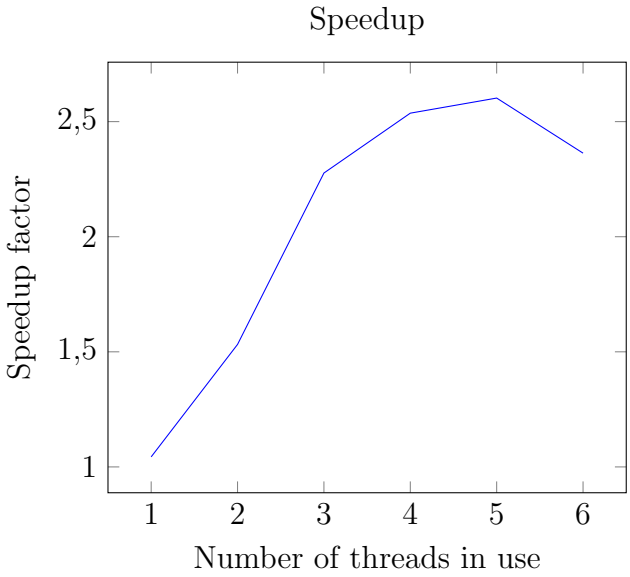

Figura 6:

```
using LinearAlgebra, MKL
using PGFPlotsX
using CpuId

# Ver todas las caracteristicas del procesador
cpuid = cpuinfo()
string_cpuid = string(cpuid)

# Comprobar si AVX, AVX2 o AVX-512 estan soportados
println("AVX support: ", occursin("256", string_cpuid))
println("AVX-512 support: ", occursin("512 bit", string_cpuid))


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_multiplication(N, N_cores, matmul, AVX_value)
```

```
36
37    Time = zeros( length(N) )
38    #Theoretical_time = 4e9/(4e9 * 512/32 * 4 * N_cores)
39
40    #Se considera que solo se necesita 1 instruccion para FMA
41    Theoretical_time = 1e9 /(4.5e9 * AVX_value * 2 * N_cores)
42    #Theoretical_time = 2e9/(1.7e9 * 512/32 * 2 * N_cores)
43
44    for (i,n) in enumerate(N) # variables inside loop have local scope
45
46     A,B = matrix_initialization(n)
47
48     t1= time_ns()
49
50     matmul(A,B)
51
52     t2 = time_ns()
53     Time[i] = (t2-t1)/(2*n^3)
54
55     println("N=", n, " Time per operation =", Time[i] , " nsec")
56     println("N=", n, " Theoretical time per operation =", Theoretical_time, " nsec")
57
58    end
59
60    return Time, Theoretical_time
61
62  end
63
64  function get_avx_value(string_cpuid)
65      # Inicializar la variable AVX_Value
66      AVX_value = 0
67
68      # Buscar el size del vector SIMD en la cadena y asignar el valor correspondiente
69      if occursin("256 bit", string_cpuid)
70          AVX_value = 8
71      elseif occursin("512 bit", string_cpuid)
72          AVX_value = 16
73      else
74          AVX_value = 0
75      end
76
77      return AVX_value
78  end
79
80  AVX_value = get_avx_value(string_cpuid)
81
82  # settings "julia.NumThreads": "auto"
83  #N_threads = Threads.nthreads()
84  N_threads = BLAS.get_num_threads()
85  N_cores = div(N_threads, 2)
86  println("Threads =", N_threads )
87  println("Cores =", N_cores )
88  time_matrix_multiplication(2000, N_cores, matrix_multiplication)
```

```
89
90  N = Vector([10:25:2500; 2500:100:10000])
91  BLAS.set_num_threads(2*N_cores)
92  # type: BLAS.set_num_threads(n) for n threads
93  println(" threads = ", BLAS.get_num_threads(), " N_cores =", N_cores )
94  Time, Theoretical_time = time_matrix_multiplication(N, N_cores, matrix_multiplication)
95  GFLOPS = 1 ./ Time
96  GFLOPS_max = 1 / Theoretical_time
97
98  x = N
99  y1 = GFLOPS
100 y2 = GFLOPS_max
101 y2_vector = fill(y2, length(y1))
102
103 plot = @pgf Axis(
104     {
105         xlabel="Matrix dimension [N]",
106         ylabel="Operations per second [GFLOPS]",
107         title="Experimental vs theorical GLOPS",
108         legend_pos = "north east",
109         ymax = 500,
110     },
111     Plot({no_marks, "blue"}, Table(x, y1)),
112     Plot({no_marks, "red"}, Table(x, y2_vector)),
113     LegendEntry("Experimental results"),
114     LegendEntry("Theoretical results"),
115 )
116
117 PGFPlotsX.save("/Users/juanroman/Desktop/documentacion_CPU/doc_latex/code/4-
        matmul_vs_theoretical-time/matmul_vs_theoretical-time.tex", plot, include_preamble=
        false)
```