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

Contents

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:

```julia
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.

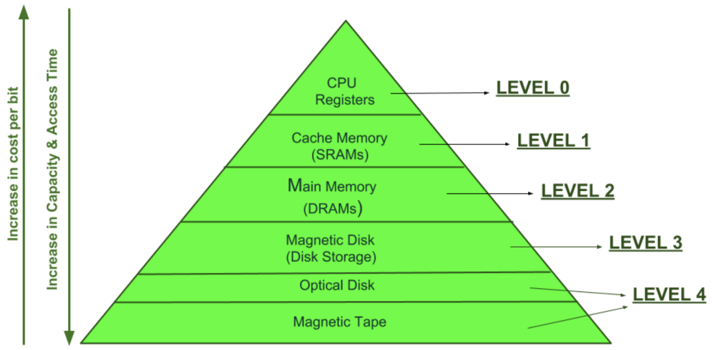

Figure 1: Representation of the hierarchy of different types of memory in a system.

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

4.1 Different Matrix-Multiplication functions

In order to achieve theoretical times (those associated with the previous expression), we need to choose which mathematical operator will be used in the benchmarks. To begin, a common operator will be used: matrix multiplication.

The next point to address is which matrix multiplication function we will use; the options are either Julia's native function, associated with the * operator (`matrix_multiplication`), or manually constructing a custom matrix multiplication function (`my_matrix_multiplication`, `my_efficient_matrix_multiplication`). To compare which of these options performs better, the number of GFLOPS (y-axis) will be plotted for different values of N, the dimension of the matrices to be multiplied (x-axis).

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

#CPU info
cpuid = cpuinfo()
string_cpuid = string(cpuid)

println("AVX support: ", occursin("256", string_cpuid))
println("AVX-512 support: ", occursin("512 bit", string_cpuid))

#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, AVX_value)

  Theoretical_time = 1e9 /(4.5e9 * AVX_value * 2 * N_cores)

  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, Theoretical_time

```

```julia
end

function get_avx_value(string_cpuid)
  AVX_value = 0

  if occursin("256 bit", string_cpuid)
      AVX_value = 8
  elseif occursin("512 bit", string_cpuid)
      AVX_value = 16
  else
      AVX_value = 0
  end

  return AVX_value
end

AVX_value = get_avx_value(string_cpuid)

# settings "julia.NumThreads": "auto"
# en bash: $ JULIA_NUM_THREADS=4 julia
BLAS.set_num_threads(8)
N_threads = BLAS.get_num_threads()
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, AVX_value)

# Set range for matrix dimensions
N = 10:100:2500

# 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, Theoretical_time = time_matrix_multilication(N, N_cores, matrix_multiplication, AVX_value)
Time2, Theoretical_time2 = time_matrix_multilication(N, N_cores, my_matrix_multiplication, AVX_value)
Time3, Theoretical_time3 = time_matrix_multilication(N, N_cores, my_efficient_matrix_multiplication, AVX_value)

# Calculate GFLOPS (floating point operations per second) for each method
GFLOPS = 1 ./ Time
GFLOPS2 = 1 ./ Time2
GFLOPS3 = 1 ./ Time3
GFLOPS_max = 1 ./ Theoretical_time

println(typeof(Time2))
println(Time2)
println(typeof(Time3))
println(Time3)

# Data for plotting
x = N
y1 = GFLOPS
y2 = GFLOPS2
y3 = GFLOPS3
y4 = GFLOPS_max
y4_vector = fill(y4, length(y1))

# Create the plot using PGFPlotsX
plot_dot = @pgf Axis(
    {
        width = "15cm",
        height = "10cm",
        xlabel="Matrix dimension [N]",
        ylabel="GFLOPS",
        title="Comparison of different matrix multiplication functions",
        legend="north east",
        ymax=500,
        #ymode="log",

    },
    Plot({no_marks, "blue"}, Table(x, y1)),
    Plot({no_marks, "orange"}, Table(x, y2)),
    Plot({no_marks, "green"}, Table(x, y3)),
    Plot({no_marks, "red"}, Table(x, y4_vector)),
    LegendEntry("matrix_multiplication"),
    LegendEntry("my_matrix_multiplication"),
    LegendEntry("my_efficient_matrix_multiplication"),
    LegendEntry("Theoretical GFLOPS"),
)

display(plot_dot)

#PGFPlotsX.save("code/dot_func_comparison_dyn.tex", plot_dot, include_preamble=false)
```

The results of running this code are shown in the following two figures; in the first one (Figure 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. The theoretical GFLOPS value is also represented in this graph,

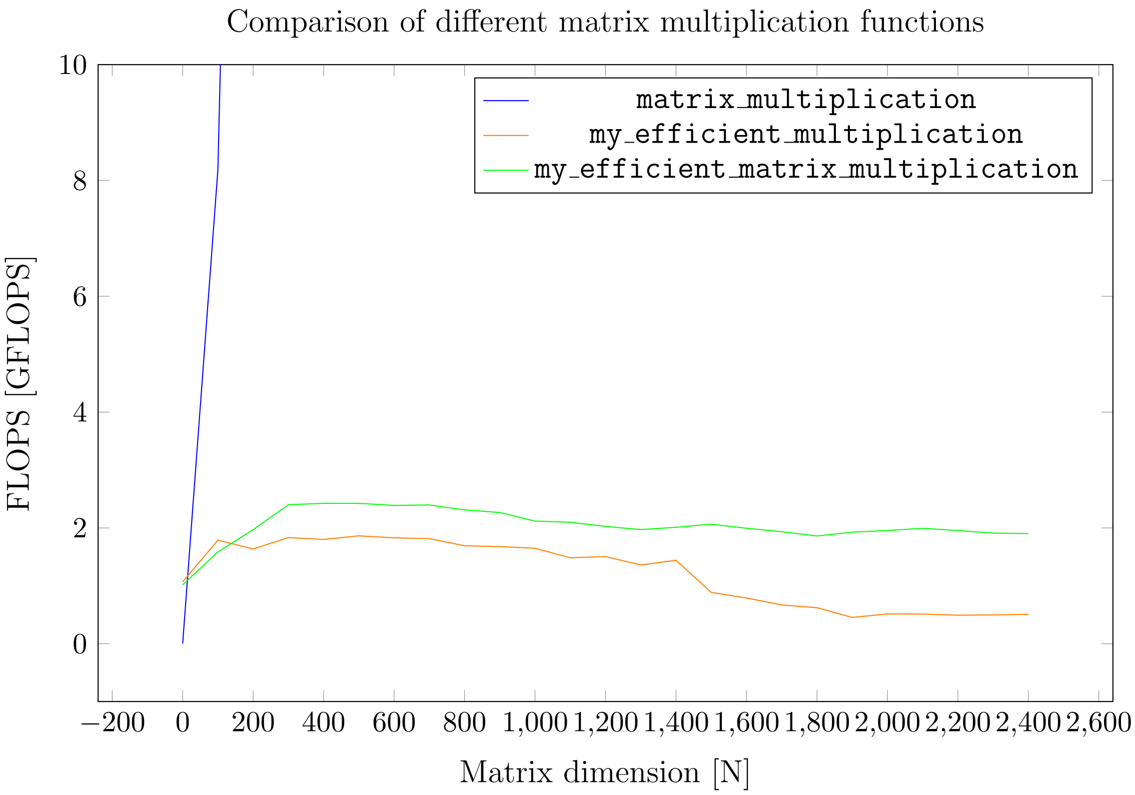

Figure 2: Matrix product efficiency, tested on a Intel(R) Core(TM) i7-8557U CPU @ 1.70GHz (1)

and the convergence of the `matrix_multiplication` function to this value can be observed. It can therefore be said that, seemingly quickly, we have achieved our objective: to observe convergence to theoretical values in experimental tests.

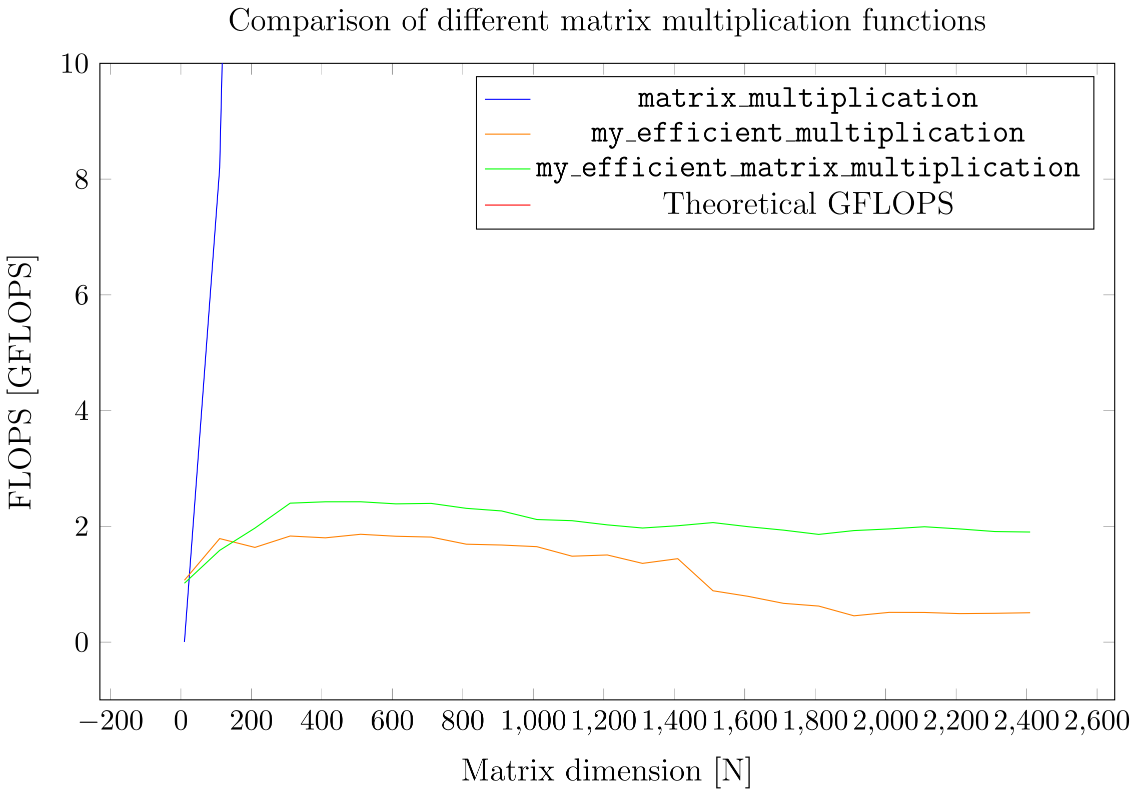

Figure 3: Matrix product efficiency, tested on a Intel(R) Core(TM) i7-8557U CPU @ 1.70GHz (2)

4.2 Comparison of BLAS Operations Across Different Levels

But what about matrix-vector multiplications? It is logical to consider the optimal shape and dimensions of these matrices. One might intuitively assume that a matrix-vector multiplication is faster than a matrix-matrix multiplication. To visualize the load that the CPU experiences in both cases, the following code is used to plot the figures.

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

cpuid = cpuinfo() # CPU Features
string_cpuid = string(cpuid)
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_vector_initialization(N)

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

    return A, B
end

function vector_vector_initialization(N)

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

    return A, B
end

function vector_multiplication(A,B)

    return dot(A, B)
end

function matrix_multiplication(A,B)

    return A * B
end

function vector_multiplication(A,B)
```

```

    return transpose(A) * B
end

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

    Time = zeros( length(N) )
    #Se considera que solo se necesita 1 instruccion para FMA
    Theoretical_time = 1e9 /(4.5e9 * AVX_value * 2 * N_cores)
    #Theoretical_time = 2e9/(1.7e9 * 512/32 * 2 * 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 get_avx_value(string_cpuid)
    # Inicializar la variable AVX_Value
    AVX_value = 0

    # Buscar el size del vector SIMD en la cadena y asignar el valor correspondiente
    if occursin("256 bit", string_cpuid)
        AVX_value = 8
    elseif occursin("512 bit", string_cpuid)
        AVX_value = 16
    else
        AVX_value = 0
    end

    return AVX_value
end

AVX_value = get_avx_value(string_cpuid)

# Function to time matrix-vector multiplication operations
function time_matrix_vector_multiplication(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

# Function to time matrix-vector multiplication operations
function time_vector_vector_multiplication(N, N_cores, matinit, matmul)

    Time3 = 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

     Time3[i] = dt/(2*n)

     println("N=", n, " Time per operation =", Time3[i] , " nsec")
     println("N=", n, " Theoretical time per operation =", Theoretical_time, " nsec")

    end

    return Time3
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_multiplication(N, N_cores, matrix_initialization, matrix_multiplication, AVX_value)
Time2 = time_matrix_vector_multiplication(N, N_cores, matrix_vector_initialization, matrix_multiplication)
Time3 = time_vector_vector_multiplication(N, N_cores, vector_vector_initialization, vector_multiplication)
# Calculate GFLOPS (floating-point operations per second)
GFLOPS = 1 ./ Time
GFLOPS2 = 1 ./ Time2
GFLOPS3 = 1 ./ Time3
GFLOPS_max = 1 / Theoretical_time

# Data for plotting
x = N
y1 = GFLOPS
y2 = GFLOPS2
y3 = GFLOPS3
y4 = fill(GFLOPS_max, length(y1))


plot = @pgf Axis(
    {
        width = "15cm",
        height = "10cm",
        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"),
)

display(plot)
#PGFPlotsX.save("code/BLAS_levels_dyn.tex", plot, include_preamble=false)
```

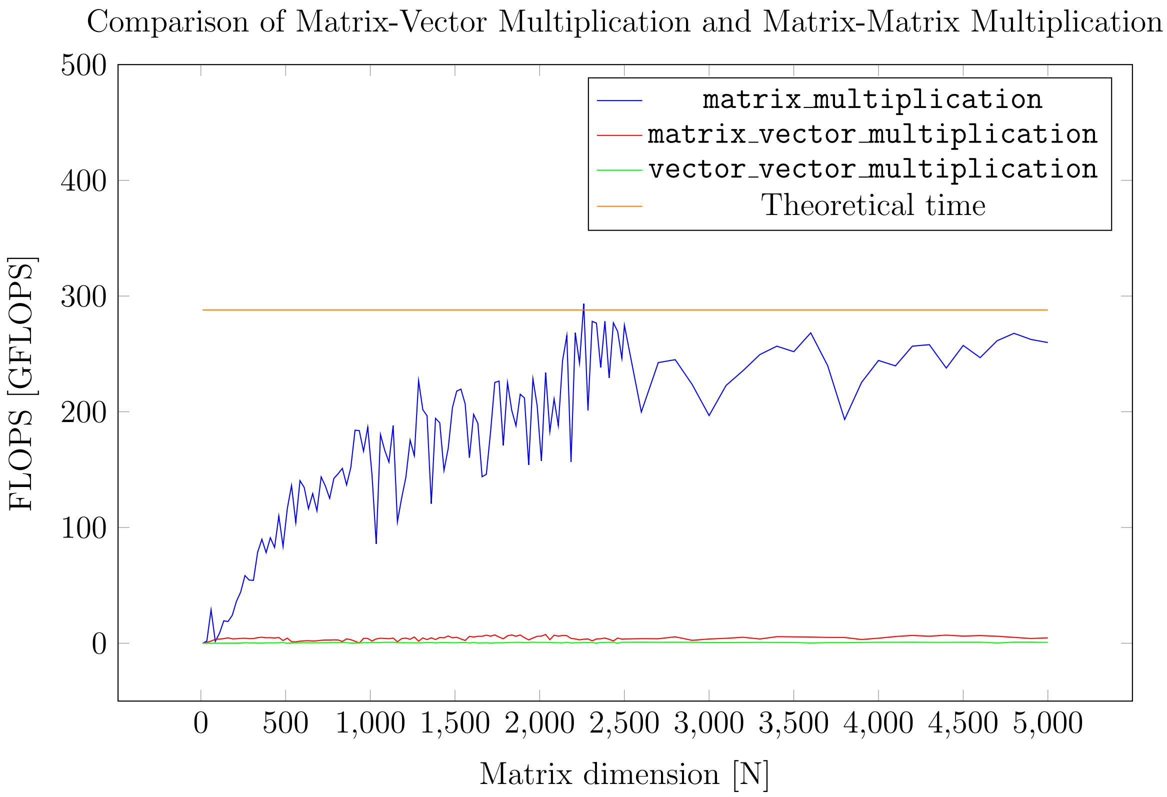

Figure 4: Representation of GFLOPS for the different levels of BLAS: matrix multiplication (Level 3 BLAS), matrix-vector multiplication (Level 2 BLAS), and vector multiplication (dot product, Level 1 BLAS).

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 `usability` 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.

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 $N_{data} = 2N$)

In conclusion, **as the usability value tends towards infinity, and with sufficiently large values of N, the CPU's performance approaches its theoretical maximum.**