



UNIVERSIDAD
POLITÉCNICA
DE MADRID

Parallel Processing

PARALLEL PROCESSING ON CPU & GPU
GRADO EN INGENIERÍA AEROESPACIAL

Autores: Juan Román Bermejo
Álvaro Martínez Collado
Santiago
Mario Alonso Cuero
A continuar por: Pedro Rodríguez Jiménez

MADRID, 1 DE MAYO DE 2023

Contents

1	Introduction to Julia: A Comprehensive Guide	1
1.1	Installing Julia and Setting Up the Environment	1
1.2	Basic Concepts in Julia	1
1.3	Plotting, Animations, and Encapsulation	2
1.4	Functions, Lambda Expressions, and Advanced Operations	4
2	CPU Performance: Benchmark Analysis and Theoretical Limits	5
2.1	Introduction	5
2.2	About the Hardware	5
2.2.1	How CPU works	5
2.2.2	Micro-operations and pipelining	6
2.2.3	Vectorization: AVX	6
2.2.4	Memory: RAM	7
2.2.5	Memory: Cache	7
2.3	Theoretical time	8
2.4	Benchmark operation	9
2.4.1	Different Matrix-Multiplication functions	9
2.4.2	Comparison of BLAS Operations Across Different Levels	13
3	GPU Performance: Benchmark Analysis	16
3.1	Understanding the GPU	16
3.1.1	Introduction to GPUs	16
3.1.2	Vectorization and parallelism	16
3.1.3	Memory Access	17
3.2	GPU Hardware	17
3.3	GPU Software	18
3.4	Operators Used in Benchmark.	18
3.4.1	CUDA	18
3.5	Benchmarks	20
3.5.1	Simply GPU Matrix Multiplication	20
3.5.2	Matrix Multiplication Modifying N using Times	22
3.5.3	GPU Matrix Multiplication Performance GFLOPS	25
3.5.4	Performance GFLOPS NVIDIA GeForce GTX 1060 6GB	27
3.5.5	Performance in GFLOPS of the NVIDIA GeForce RTX 2060 Laptop	28
4	Solving the Convective-Diffusive Heat Equation with Numerical Methods	29
4.1	Introduction and Theoretical framework	29
4.1.1	Numerical Methods used	30
4.2	Computational Implementation	32
4.3	Results and Analysis	33
4.4	Global Interpolation solving	37
4.4.1	Graphs and tables	44
4.5	Conclusions	45
4.6	Bibliografía	45

1 Introduction to Julia: A Comprehensive Guide

1.1 Installing Julia and Setting Up the Environment

Download and Installation Julia can be downloaded from its official website [here](#). Select the appropriate version for your operating system and follow the installation instructions.

Setting Up the Development Environment For a better experience, we recommend using Julia with Visual Studio Code, which can be downloaded from [Visual Studio Code](#). After installation, add the Julia extension to enhance the development process. Also, using GitHub integration within Visual Studio Code simplifies version control.

Setting Up Julia Environment for Package Management To set up an optimal Julia development environment, especially if you're working on a project involving GitHub or GPU-Parallel computing, it is advisable to install certain packages globally:

```
1 using Pkg
2 Pkg.add("Revise")
3 Pkg.add("BenchmarkTools")
4 Pkg.add("ProfileView")
```

Next, activate your local environment and ensure all project dependencies are installed:

```
1 using Pkg
2 Pkg.activate(".")
3 Pkg.instantiate()
```

1.2 Basic Concepts in Julia

Basic Syntax and Functions Julia is a dynamic, high-performance language. Here's a simple example of how to define variables and functions:

```
1 # Variable definition
2 x = 10
3
4 # Function definition
5 function sum(a, b)
6     return a + b
7 end
8
9 # Calling the function
10 result = sum(3, 4)
11 println(result) # Prints 7
```

Explanation: - 'function sum(a, b)' defines a function called 'sum' that takes two arguments 'a' and 'b'. In Julia, you can define functions using the 'function' keyword or, for simpler functions, using the more concise arrow notation ('f(x) = x²'). - 'println(result)' prints the result to the console. In Julia, 'println' is used to print output to the screen.

The Use of ! in Function Naming In Julia, functions that modify their arguments are conventionally suffixed with an exclamation mark '!'. This is not a syntax requirement but a widely adopted convention in Julia to signal that a function performs [in-place](#) modifications (i.e., it changes the data it operates on).

For example:

```
1 function modify!(x)
2     x[1] = 100 # Modifies the first element of the vector
3 end
4
5 v = [1, 2, 3]
6 modify!(v)
7 println(v) # v is now [100, 2, 3]
```

In this example, `modify!` changes the contents of the input vector `v` directly, so it should be used carefully, especially when working with large datasets.

Matrix Multiplication: `*` vs. Element-wise Multiplication: `.*` Julia follows standard linear algebra notation where `*` is used for matrix multiplication, and element-wise operations are performed with the dot operator `.`.

```
1 A = [1 2; 3 4]
2 B = [5 6; 7 8]
3
4 # Matrix multiplication (A * B)
5 C = A * B
6 println(C) # Standard matrix multiplication
7
8 # Element-wise multiplication (A .* B)
9 D = A .* B
10 println(D) # Element-wise multiplication
```

Explanation: - `A * B`: Performs matrix multiplication. This operation is akin to the dot product and follows linear algebra rules for matrix multiplication. - `A .* B`: Performs element-wise multiplication, where corresponding elements in the matrices `A` and `B` are multiplied together.

In-place Matrix Multiplication with `mul!` Julia also provides a way to perform in-place operations, such as matrix multiplication, without allocating new memory for the result. The `mul!` function does this by storing the result directly in a preallocated array.

```
1 C = zeros(2, 2) # Preallocate result matrix
2 mul!(C, A, B)
3 println(C)
```

Explanation: - `mul!(C, A, B)` performs matrix multiplication `A * B` and stores the result in matrix `C`. This avoids the creation of a new matrix, making it more memory-efficient, especially for large matrices.

1.3 Plotting, Animations, and Encapsulation

2D Plots and Contour Maps Julia supports powerful plotting capabilities using the `Plots.jl` package.

```
1 using Plots
2
3 # Plotting y = f(x)
4 x = 0:0.1:10
5 y = sin.(x)
6 plot(x, y, label="sin(x)")
```

Explanation: - The range '0:0.1:10' defines values of 'x' from 0 to 10 with a step of 0.1. The 'sin.(x)' expression applies the sine function element-wise to the vector 'x'. This is an example of Julia's broadcasting mechanism, which allows functions to be applied to entire arrays.

For contour plots, which visualize 3D surfaces in 2D, you can use:

```
1 using Plots
2
3 # Defining the function
4 f(x, y) = sin(x) * cos(y)
5
6
7 # Range of values
8 x = y = 0:0.1:2*pi
9
10 # Contour map
11 contour(x, y, (x, y) -> f(x, y))
```

2D Animations Animations are easy to create in Julia, as shown in this example, which animates a sine wave:

```
1 using Plots
2
3 # Setting up the backend
4 gr()
5
6 # Initial data
7 x = 0:0.1:10
8 y = sin.(x)
9
10 # Create the animation
11 anim = @animate for i in 1:100
12     plot(x, sin.(x .+ 0.1*i), ylim=(-1,1))
13 end
14
15 # Save the animation
16 gif(anim, "sine_wave.gif", fps=10)
```

Explanation: - '@animate' is a macro that collects individual plot frames into an animation. - 'gif(anim, sine_wave.gif, fps=10)' saves the animation as a GIF file with 10 frames per second.

Encapsulating Plot Functions You can encapsulate plot logic into reusable functions, making your code more modular:

```
1 function plot_function(f, x_range)
2     y = f.(x_range)
3     plot(x_range, y, label="f(x)")
4 end
5
6 # Usage example
7 plot_function(x -> x^2, -10:0.1:10)
```

Explanation: - 'plot_function(f, x_range)' takes a function 'f' and a range of 'x' values and plots the corresponding 'y' values. The 'f.(x_range)' syntax applies the function 'f' to each element of 'x_range' using broadcasting. - 'x -> x²' is a lambda function (anonymous function) that squares its input.

1.4 Functions, Lambda Expressions, and Advanced Operations

Functions and Lambda Expressions Julia supports lambda functions (also called anonymous functions), which are concise one-line functions often used in operations like mapping or quick computations.

```
1 # Lambda function to square a number
2 square = x -> x^2
3 println(square(4)) # Prints 16
```

Lambda functions are particularly useful when performing element-wise operations on arrays, or for quick mathematical operations inside other functions.

Vector and Matrix Operations Basic matrix operations, such as matrix multiplication and element-wise operations, are straightforward in Julia:

```
1 A = [1 2; 3 4]
2 B = [5 6; 7 8]
3
4 # Matrix multiplication
5 C = A * B
6
7 # Element-wise multiplication
8 D = A .* B
9 println(D)
```

Julia provides multiple options for concise operations using lambdas and broadcasting (with the dot operator ‘.’).

Tensor Product Revisited For large-scale or multi-dimensional operations, the tensor product is key:

```
1 v1 = [1, 2]
2 v2 = [3, 4]
3
4 # Tensor product
5 tensor = kron(v1, v2)
```

The tensor product allows you to construct higher-dimensional arrays (or tensors) by combining smaller arrays or vectors. This is critical in advanced linear algebra and applications like quantum computing or machine learning.

2 CPU Performance: Benchmark Analysis and Theoretical Limits

2.1 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.

2.2 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.

2.2.1 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.

2.2.2 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.

2.2.3 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:

```
1 Pkg.add("CpuId")
2 using CpuId
3
4 # View all processor features
5 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.

2.2.4 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: $\frac{17}{2400} \times 1000 \simeq 7.08[ms]$

2.2.5 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.

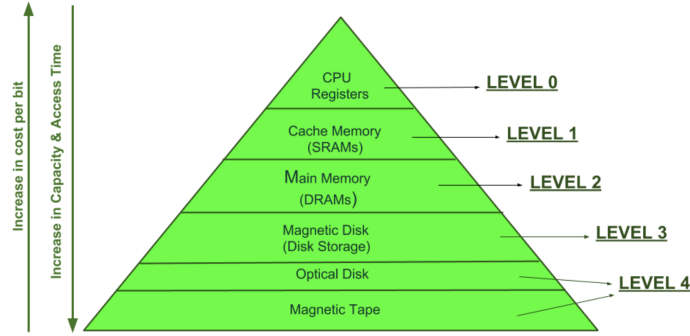


Figure 1: Representation of the hierarchy of different types of memory in a system.

2.3 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:

$$t_{CPU} = \frac{N_{ops} \times S_{ops}}{V_{vectorization} \times GH_{zCPU} \times M_{micro-ops} \times C_{CPU}} \quad (1)$$

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_{zCPU} : 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

2.4 Benchmark operation

2.4.1 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).

```
1 import Pkg
2 Pkg.activate(".")
3 Pkg.add(["PGFPlotsX", "CPUTime", "Plots", "LinearAlgebra", "MKL"])
4 using CPUTime
5 using Plots
6 using LinearAlgebra, MKL
7 using PGFPlotsX
8 using CpuId
9
10 # CPU info
11 cpuid = cpuinfo()
12 string_cpuid = string(cpuid)
13
14 println("AVX support: ", occursin("256", string_cpuid))
15 println("AVX-512 support: ", occursin("512 bit", string_cpuid))
16
17 # Function to initialize random matrices
18 function matrix_initialization(N)
19     A = rand(Float32, N, N)
20     B = rand(Float32, N, N)
21     return A, B
22 end
23
24 # Function to multiply matrices using the built-in Julia method
25 function matrix_multiplication(A, B)
26     return A * B
27 end
28
29 # Function to multiply matrices using a custom method (manual loop)
30 function my_matrix_multiplication(A, B)
31     (N, M) = size(A)
32     (M, L) = size(B)
33     C = zeros(Float32, (N, L))
34
35     for i in 1:N, j in 1:L
36         for k in 1:M
37             C[i, j] = C[i, j] + A[i, k] * B[k, j]
38         end
39     end
40     return C
41 end
42
43 # Transposing B for efficient memory access
44 function my_efficient_matrix_multiplication(A, B)
```

```

45 (N, M) = size(A)
46 (M, L) = size(B)
47 BT = transpose(B)
48 C = zeros(Float32, (N, L))
49
50 for k in 1:M
51     for j in 1:L, i in 1:N
52         C[i, j] = C[i, j] + A[i, k] * BT[j, k]
53     end
54 end
55 return C
56 end
57
58 # Function to time matrix multiplication and calculate performance
59 function time_matrix_multilication(N, N_cores, matmul, AVX_value)
60     Theoretical_time = 1e9 / (4.5e9 * AVX_value * 2 * N_cores)
61     Time = zeros(length(N))
62
63     for (i, n) in enumerate(N)
64         A, B = matrix_initialization(n)
65         t1 = time_ns()
66         matmul(A, B)
67         t2 = time_ns()
68         Time[i] = (t2 - t1) / (2 * n^3)
69     end
70     return Time, Theoretical_time
71 end
72
73 function get_avx_value(string_cpuid)
74     AVX_value = 0
75     if occursin("256 bit", string_cpuid)
76         AVX_value = 8
77     elseif occursin("512 bit", string_cpuid)
78         AVX_value = 16
79     end
80     return AVX_value
81 end
82
83 AVX_value = get_avx_value(string_cpuid)
84 BLAS.set_num_threads(8)
85 N_threads = BLAS.get_num_threads()
86 N_cores = div(N_threads, 2)
87 println("Threads =", N_threads)
88 println("Cores =", N_cores)
89
90 # Precompilation: Run matrix multiplication once to warm up
91 time_matrix_multilication(2000, N_cores, matrix_multiplication, AVX_value)
92
93 # Set range for matrix dimensions
94 N = 10:100:2500
95 BLAS.set_num_threads(2 * N_cores)
96
97 # Time the built-in matrix multiplication and custom multiplication
98 Time, Theoretical_time = time_matrix_multilication(N, N_cores, matrix_multiplication,
99     AVX_value)
100 Time2, _ = time_matrix_multilication(N, N_cores, my_matrix_multiplication, AVX_value)
101 Time3, _ = time_matrix_multilication(N, N_cores, my_efficient_matrix_multiplication,
102     AVX_value)
103
104 # Calculate GFLOPS for each method
105 GFLOPS = 1 ./ Time
106 GFLOPS2 = 1 ./ Time2
107 GFLOPS3 = 1 ./ Time3

```

```
107 # Plotting code here (omitted for brevity)
```

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.

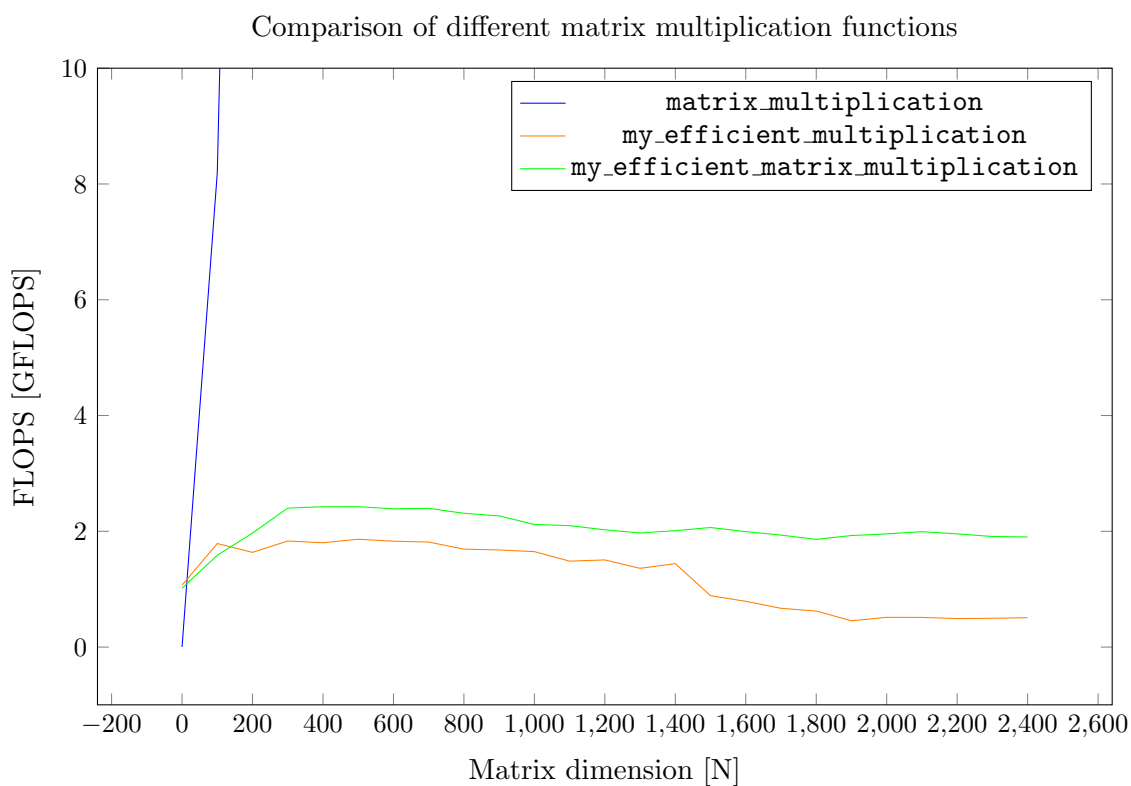


Figure 2: Matrix product efficiency, tested on a Intel(R) Core(TM) i7-8557U CPU @ 1.70GHz (1)

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, 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.

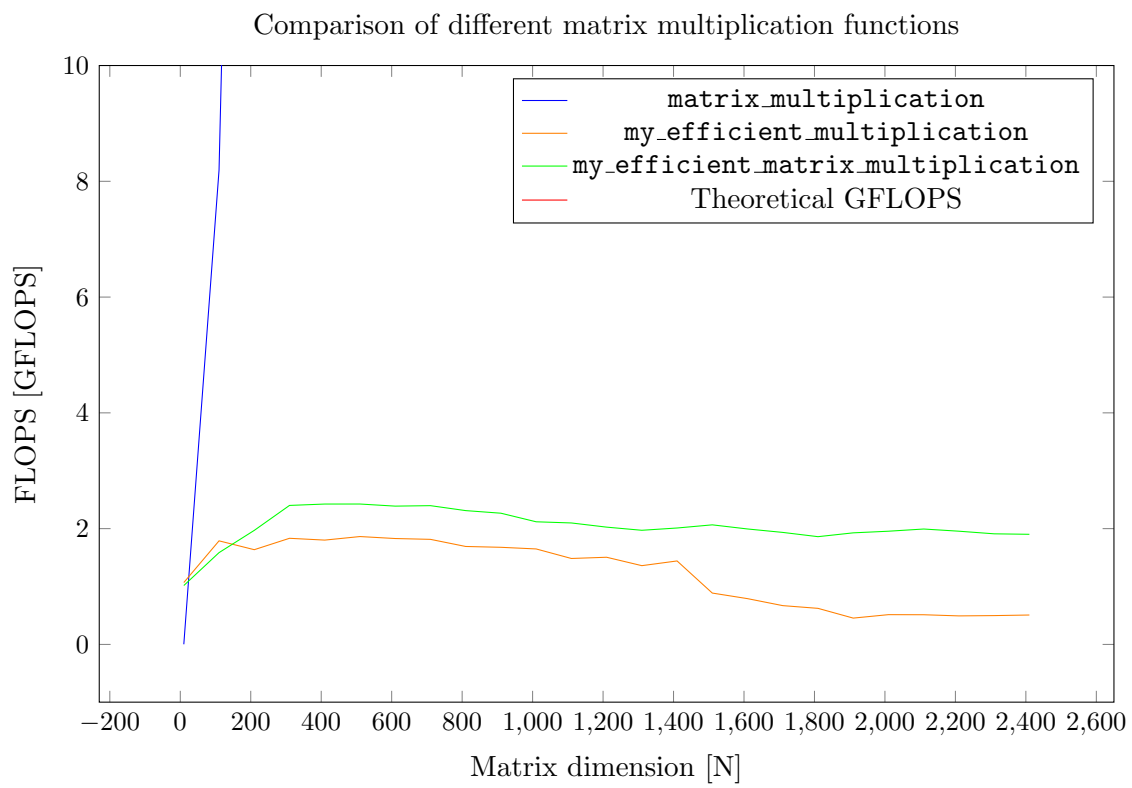


Figure 3: Matrix product efficiency, tested on a Intel(R) Core(TM) i7-8557U CPU @ 1.70GHz
(2)

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.

```

1 import Pkg
2 Pkg.activate(".")
3 Pkg.add(["CPUTime", "Plots", "LinearAlgebra", "MKL", "PGFPlotsX", "CpuId"])
4 using CPUTime, Plots, LinearAlgebra, MKL, PGFPlotsX, CpuId
5
6 cpuid = cpuinfo() # CPU Features
7 string_cpuid = string(cpuid)
8 println("AVX support: ", occursin("256", string_cpuid))
9 println("AVX-512 support: ", occursin("512 bit", string_cpuid))
10
11 function matrix_initialization(N)
12     A = rand(Float32, N, N)
13     B = rand(Float32, N, N)
14     return A, B
15 end
16
17 function matrix_vector_initialization(N)
18     A = rand(Float32, N, N)
19     B = rand(Float32, N, 1)
20     return A, B
21 end
22
23 function vector_vector_initialization(N)
24     A = rand(Float32, N, 1)
25     B = rand(Float32, N, 1)
26     return A, B
27 end
28
29 function vector_multiplication(A, B)
30     return dot(A, B)
31 end
32
33 function matrix_multiplication(A, B)
34     return A * B
35 end
36
37 function time_matrix_multiplication(N, N_cores, matinit, matmul, AVX_value)
38     Time = zeros(length(N))
39     Theoretical_time = 1e9 / (4.5e9 * AVX_value * 2 * N_cores)
40
41     for (i, n) in enumerate(N)
42         A, B = matinit(n)
43         t1 = time_ns()
44         matmul(A, B)
45         t2 = time_ns()
46         dt = t2 - t1
47         Time[i] = dt / (2 * n^3)
48         println("N=", n, " Time per operation =", Time[i], " nsec")
49         println("N=", n, " Theoretical time per operation =", Theoretical_time, " nsec")
50     end
51
52     return Time, Theoretical_time
53 end
54
55 function get_avx_value(string_cpuid)

```



```

56     AVX_value = 0
57     if occursin("256 bit", string_cpuid)
58         AVX_value = 8
59     elseif occursin("512 bit", string_cpuid)
60         AVX_value = 16
61     else
62         AVX_value = 0
63     end
64     return AVX_value
65 end
66
67 AVX_value = get_avx_value(string_cpuid)
68
69 N_cores = 4
70 N = Vector([10:25:2500; 2500:100:5000])
71 BLAS.set_num_threads(2 * N_cores)
72 println("Threads = ", BLAS.get_num_threads(), " N_cores =", N_cores)
73
74 Time, Theoretical_time = time_matrix_multiplication(N, N_cores, matrix_initialization,
    matrix_multiplication, AVX_value)

```

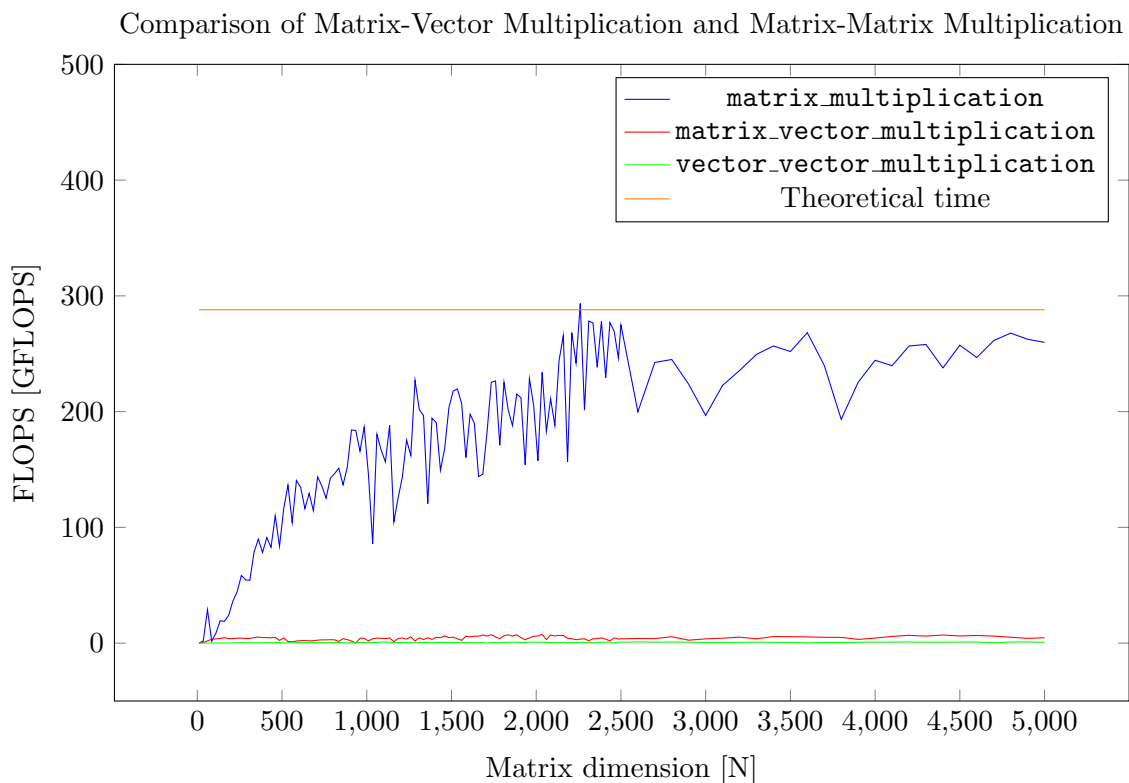


Figure 4: Representation of GFLOPS for the different levels of BLAS: matrix multiplication (Level 3 BLAS), matrix-vector multiplication (Level 2 BLAS), and vector multiplication (dot product, Level 1 BLAS).

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 :

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

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

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:

$$U = \frac{N_{ops}}{N_{data}} \quad (4)$$

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.**

3 GPU Performance: Benchmark Analysis

3.1 Understanding the GPU

(Before starting, it is important to mention that during the study of this project, an NVIDIA GeForce RTX 3050 was used as the base graphics card. Therefore, unless explicitly mentioned in specific subsections, this graphics card will be assumed as the default.)

3.1.1 Introduction to GPUs

A GPU is a specialized electronic circuit designed to accelerate the creation and rendering of images, animations, and videos for display output. GPUs are highly efficient at performing parallel calculations on large blocks of data, making them essential not only for graphics tasks but also for computational workloads like machine learning and scientific simulations.

How is a GPU different from a CPU?

GPUs differ from CPUs in their architecture and processing capabilities. GPUs have many small, simple cores designed for parallel processing, making them highly effective for tasks like image processing and deep learning, which require handling many operations simultaneously. In contrast, CPUs have fewer, more complex cores optimized for general-purpose tasks, including complex decision-making and single-threaded applications.

How is a GPU similar to a CPU?

Despite their differences, GPUs and CPUs both execute instructions and process data using semiconductor technology. They share fundamental principles of binary logic and data management. Additionally, both have evolved to handle a broader range of tasks: GPUs now support general-purpose computing, and CPUs include more cores and specialized instructions for parallel processing.

3.1.2 Vectorization and parallelism

As previously mentioned, GPUs support both vectorization and parallelization, though differently from CPUs. While CPUs have a few powerful cores optimized for sequential processing and use vectorization, GPUs consist of thousands of smaller cores designed for massive parallelism. This makes GPUs exceptionally efficient for tasks like matrix multiplication, where many operations can be performed concurrently.

Vectorization:

GPUs achieve vectorization by executing the same instruction across multiple cores organized in grids and blocks (or threads in CUDA). Each core processes a different data set simultaneously, enabling efficient large-scale data processing.

Parallelization:

GPUs excel in parallelization by leveraging their many cores to handle thousands of threads concurrently, making them ideal for tasks that can be divided into numerous parallel operations.

3.1.3 Memory Access

Memory access in GPUs is optimized for high throughput and parallel processing. GPUs use several types of memory with different characteristics:

- **Global Memory:** This is the largest and slowest memory on the GPU, accessible by all cores. It's used for storing large datasets and is suitable for tasks that involve significant data exchange between the CPU and GPU. However, access to global memory can be slow if not managed efficiently.
- **Shared Memory:** Shared memory is much faster than global memory and is used for communication between threads within the same block. It allows for efficient data sharing and reduces the need to repeatedly access global memory, improving performance for tasks with high inter-thread communication.
- **Local Memory:** Each thread has its own local memory, which is private and used for storing data specific to that thread. Local memory helps in storing intermediate results and temporary data.
- **Texture and Constant Memory:** These are specialized types of memory. Texture memory is optimized for spatial locality and is used for image and graphical data, while constant memory is used for read-only data that remains constant across all threads during a kernel execution.

Efficient memory access is crucial for performance, and optimizing memory use involves minimizing global memory accesses, maximizing shared memory usage, and ensuring coalesced memory accesses to reduce latency and improve throughput.

3.2 GPU Hardware

The hardware architecture of GPUs is designed to facilitate massive parallel processing, making them ideal for tasks that can be parallelized. Key components include:

- **Cores:** GPUs are equipped with thousands of small, simple cores designed for parallel execution. Unlike CPU cores, which are few in number but powerful, GPU cores are numerous and less complex. These cores work together to perform many operations simultaneously, which is crucial for tasks like rendering images and processing large datasets.
- **Streaming Multiprocessors (SMs):** The GPU cores are organized into Streaming Multiprocessors (SMs). Each SM contains multiple cores, a portion of shared memory, and scheduling hardware. SMs manage the execution of threads within a block, handling the scheduling and synchronization necessary for efficient parallel processing.
- **Memory Hierarchy:** The GPU has a complex memory hierarchy designed to balance speed and capacity.
- **Interconnects:** High-speed interconnects, such as NVIDIA's NVLink and AMD's Infinity Fabric, are used to connect multiple GPUs or to facilitate rapid communication between different parts of a GPU. These interconnects enhance performance by allowing for efficient data transfer and coordination among multiple GPUs in a system.

3.3 GPU Software

GPU software encompasses various tools and frameworks that facilitate the development and execution of parallel programs on GPUs. Major components include:

- **CUDA:** NVIDIA's Compute Unified Device Architecture (CUDA) is a parallel computing platform and API that allows developers to leverage NVIDIA GPUs for general-purpose computing. CUDA provides extensions to standard programming languages such as C, C++, and Fortran, and includes a comprehensive toolkit for developing GPU-accelerated applications. It supports features like kernel programming, memory management, and inter-thread synchronization.
- **OpenCL:** The Open Computing Language (OpenCL) is an open standard for parallel programming across heterogeneous systems, including GPUs from various vendors. OpenCL provides a unified programming model and API that enables developers to write code that can run on different types of processors (CPUs, GPUs, FPGAs) and is suitable for a wide range of applications.
- **Libraries and Frameworks:** Several libraries and frameworks provide pre-built functions and abstractions for GPU programming, making it easier to develop high-performance applications:
 1. **cuBLAS:** A library for performing basic linear algebra operations on NVIDIA GPUs, optimized for performance.
 2. **cuDNN:** A library for deep neural network operations, providing highly optimized routines for deep learning tasks.
 3. **TensorFlow and PyTorch:** Popular machine learning frameworks that utilize GPU acceleration to improve training and inference performance.
- **Driver Software:** GPU drivers are essential for enabling communication between the operating system and GPU hardware. They handle the execution of GPU-accelerated applications, manage resources, and provide support for various APIs and frameworks. Regular updates to drivers ensure compatibility with new software and hardware features.
- **Development Tools:** Various development tools and IDEs (Integrated Development Environments) support GPU programming by providing debugging, profiling, and performance analysis capabilities. Examples include NVIDIA Nsight and AMD ROCm tools.

Understanding these software components and their interaction with GPU hardware is crucial for developing efficient GPU-accelerated applications and achieving optimal performance.

3.4 Operators Used in Benchmark.

3.4.1 CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA. It enables developers to leverage NVIDIA GPUs for general-purpose computing tasks beyond traditional graphics rendering. CUDA provides a set of extensions to standard programming languages such as C, C++, and Fortran, allowing developers to write programs that execute on the GPU. CUDA facilitates efficient parallel processing by

exposing low-level GPU architecture features, including fine-grained control over memory and execution.

Key Features of CUDA:

- **Parallel Computing Model:** CUDA allows for the execution of multiple threads in parallel on the GPU. Threads are organized into blocks, and blocks are organized into grids, providing a flexible and scalable way to manage parallelism.
- **Memory Hierarchy:** CUDA provides access to different types of memory, including global, shared, and local memory, enabling optimization of memory access patterns and efficient data management.
- **Kernel Functions:** In CUDA, functions executed on the GPU are called kernels. Kernels are written in CUDA C/C++ and are launched from the host (CPU) to run on the device (GPU).
- **Thrust Library:** CUDA includes the Thrust library, which provides high-level abstractions for common parallel algorithms, such as sorting and reduction, simplifying development.

To use CUDA with Julia, you need to install the CUDA toolkit and the necessary Julia packages.

Once the CUDA toolkit and Julia packages are installed, you can start utilizing CUDA features in your Julia code. Here are some common tasks:

- **Checking GPU Availability:** Use the ‘CUDA’ package to check if your GPU is available and properly configured:

```
using CUDA
println(CUDA.has_cuda())
```

- **Allocating and Transferring Data:** Allocate memory on the GPU and transfer data between the host and device:

```
a = CUDA.rand(10) # Allocate an array on the GPU
b = CUDA.fill(2.0, 10) # Fill an array with a constant value on the GPU
c = a .+ b # Perform element-wise addition on the GPU
```

CUDA Utilities: In addition to core CUDA functionalities, several utilities and tools are available to help with development and performance optimization:

- **NVIDIA Nsight:** A suite of tools for debugging and profiling CUDA applications, providing insights into performance and helping identify bottlenecks.
- **CUDA Profiler:** Built into the CUDA toolkit, this tool allows for detailed performance analysis of GPU applications, including memory usage and execution time.

- **CUDA Samples:** The CUDA toolkit includes sample code and examples that demonstrate various features and best practices for CUDA programming.

Understanding and utilizing CUDA effectively can significantly enhance the performance of computational tasks by leveraging the parallel processing power of NVIDIA GPUs.

3.5 Benchmarks

In this subsection, we present benchmarks with the GPU that compare the performance of various computational tasks. The goal is to provide a detailed analysis of the time processes involved and to draw meaningful conclusions about the performance characteristics of the GPU.

Benchmark Methodology

The benchmarks conducted are designed to evaluate the performance of GPU computations to see the differences against the CPU performance. To ensure a fair comparison, the same set of algorithms and operations are executed on both types of hardware, so in this subsection it's going to be analyzed similar benchmarks to the CPU ones. The benchmarks focus on evaluating:

- **Execution Time:** The time taken to complete the computations on each platform.
- **Scalability:** How performance scales with increasing problem size and complexity.
- **Efficiency:** The computational efficiency of the operations, including memory usage and processing power.

3.5.1 Simply GPU Matrix Multiplication

This subsection presents a simple benchmark for evaluating the performance of matrix multiplication on the GPU. The objective is to measure the time taken for matrix multiplication using GFlops with varying matrix sizes. The benchmark uses the Julia programming language and the CUDA library.

The code below performs matrix multiplications for matrices of different sizes and records the time taken for each operation. The results are printed to the console, providing an overview of how the execution time scales with matrix size.

Description of the Benchmark

1. **Matrix Size Variation:** The benchmark tests matrix sizes ranging from 100 to 10,000 in increments of 100.
2. **Matrix Initialization:** For each size N , two matrices A and B are initialized with random floating-point values using the 'CUDA.rand' function.
3. **Timing Measurement:** The time required for matrix multiplication $A \times B$ is measured using the 'CUDA.@elapsed' macro. This provides the duration of the multiplication operation.
4. **Results Output:** The matrix size N and the corresponding duration t are printed to the console. The results can be saved to a CSV file, although the line for writing to the file is commented out in this example.

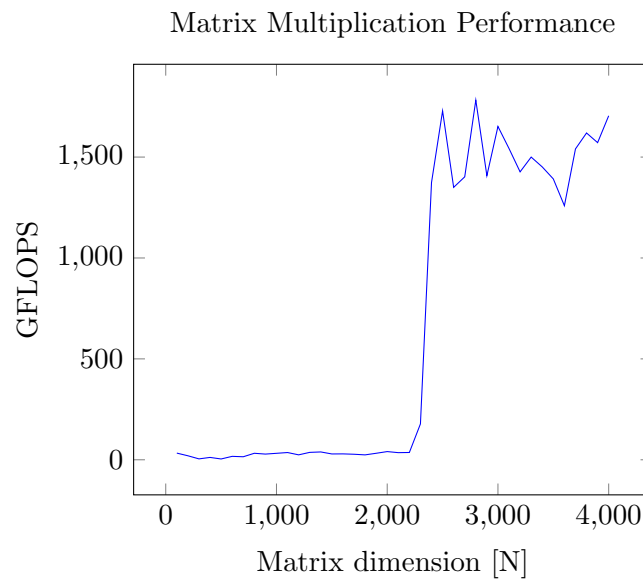


Figure 5: GPU efficiency with matrix multiplications.

Code Here is the Julia code used for this benchmark:

```

1 using LinearAlgebra
2
3 # Function to initialize random matrices
4 function matrix_initialization(N)
5     A = rand(Float32, N, N)
6     B = rand(Float32, N, N)
7     return A, B
8 end
9
10 # Function to perform matrix multiplication using the built-in Julia method
11 function matrix_multiplication(A, B)
12     return A * B
13 end
14
15 # Custom matrix multiplication method with a manual loop
16 function custom_matrix_multiplication(A, B)
17     (N, M) = size(A)
18     (M, L) = size(B)
19     C = zeros(Float32, (N, L))
20
21     for i in 1:N, j in 1:L
22         for k in 1:M
23             C[i, j] += A[i, k] * B[k, j]
24         end
25     end
26     return C
27 end
28
29 # Transpose optimization for more efficient memory access
30 function optimized_matrix_multiplication(A, B)
31     (N, M) = size(A)
32     (M, L) = size(B)
33     BT = transpose(B)
34     C = zeros(Float32, (N, L))

```



```
35 |
36 |     for k in 1:M
37 |         for j in 1:L, i in 1:N
38 |             C[i, j] += A[i, k] * BT[j, k]
39 |         end
40 |     end
41 |     return C
42 | end
43 |
44 | # Example usage
45 | N = 500
46 | A, B = matrix_initialization(N)
47 |
48 | # Built-in matrix multiplication
49 | C_builtin = matrix_multiplication(A, B)
50 |
51 | # Custom matrix multiplication
52 | C_custom = custom_matrix_multiplication(A, B)
53 |
54 | # Optimized matrix multiplication
55 | C_optimized = optimized_matrix_multiplication(A, B)
56 |
57 | println("Matrix multiplication complete.")
```

3.5.2 Matrix Multiplication Modifying N using Times

In this subsection, we present a benchmark that measures the performance of matrix multiplication on the GPU. Specifically, we analyze the duration required for matrix multiplication as the size of the matrices varies. The benchmark is conducted using the Julia programming language with the CUDA library.

The benchmark procedure involves varying the size of the matrices and recording the time taken for a series of matrix multiplications. The results are then saved to a CSV file for further analysis.

Description of the Benchmark

1. **Matrix Size Variation:** The benchmark varies the size of the matrices from 300 to 2499 with a step size of 25.
2. **Number of Operations:** For each matrix size N , the number of operations ‘TIMES’ is calculated to ensure that the total number of operations is constant for different matrix sizes. The formula used is:

$$\text{TIMES} = \frac{N_{\text{ops}}}{2 \times N^3}$$

where N_{ops} is set to 2×10000^3 .

3. **Timing Measurement:** The time taken to perform the matrix multiplications is measured using the ‘@elapsed’ macro. The benchmark performs ‘TIMES’ multiplications and records the total duration.
4. **Data Recording:** The results, including matrix size, number of operations, and duration, are written to a CSV file for analysis.

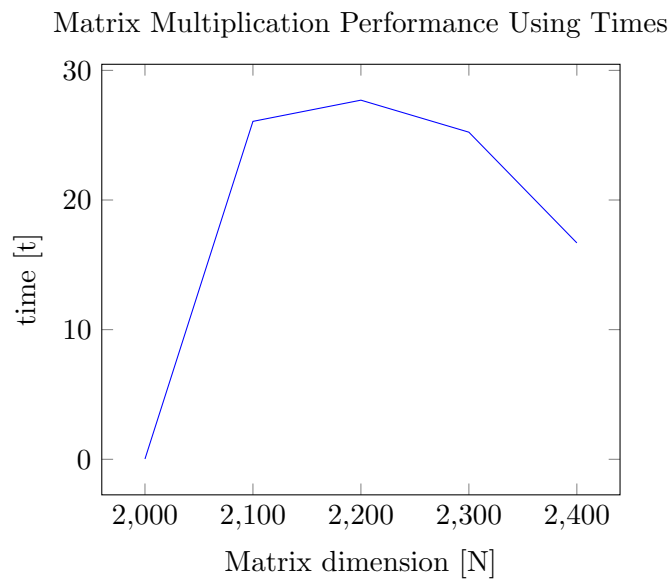


Figure 6: GPU efficiency with matrix multiplications.

Code Below is the Julia code used to perform this benchmark:

```

1 using LinearAlgebra, CPUTime
2
3 # Function to initialize random matrices
4 function matrix_initialization(N)
5     A = rand(Float32, N, N)
6     B = rand(Float32, N, N)
7     return A, B
8 end
9
10 # Function to perform matrix multiplication using the built-in Julia method
11 function matrix_multiplication(A, B)
12     return A * B
13 end
14
15 # Custom matrix multiplication method with a manual loop
16 function custom_matrix_multiplication(A, B)
17     (N, M) = size(A)
18     (M, L) = size(B)
19     C = zeros(Float32, (N, L))
20
21     for i in 1:N, j in 1:L
22         for k in 1:M
23             C[i, j] += A[i, k] * B[k, j]
24         end
25     end
26     return C
27 end
28
29 # Function to measure execution time of matrix multiplication
30 function measure_execution_time(N, matmul_func)
31     A, B = matrix_initialization(N)
32     start_time = time_ns()
33     matmul_func(A, B)
34     elapsed_time = (time_ns() - start_time) / 1e9 # Convert from nanoseconds to

```

```
        seconds
35     return elapsed_time
36 end
37
38 # Example usage
39 N = 500
40 time_builtin = measure_execution_time(N, matrix_multiplication)
41 time_custom = measure_execution_time(N, custom_matrix_multiplication)
42
43 println("Execution time for built-in matrix multiplication: ", time_builtin, " seconds
44 ")
45 println("Execution time for custom matrix multiplication: ", time_custom, " seconds")
```

3.5.3 GPU Matrix Multiplication Performance GFLOPS

This subsection describes a benchmark that measures the performance of matrix multiplication on the GPU using an NVIDIA GeForce GTX 1060 6GB. The objective is to evaluate the time taken for matrix multiplication across various matrix sizes and compare it to theoretical performance estimates. The benchmark uses the Julia programming language along with the CUDA and Plots libraries.

The following Julia code performs the matrix multiplication and measures the execution time. It also plots the performance results in GFLOPS (Giga Floating Point Operations per Second) to visualize how the GPU handles different matrix sizes.

Description of the Benchmark

1. **Matrix Initialization:** The matrix initialization GPU function initializes matrices A and B with random floating-point values and transfers them to the GPU as ‘CuArray’.
2. **Matrix Multiplication:** The matrix multiplication function performs matrix multiplication on the CPU, while the matrix multiplication GPU function measures the time required for GPU-based matrix multiplication using the ‘CUDA.@elapsed’ macro.
3. **Timing Measurement:** The time matrix multiplication function measures the duration of matrix multiplications for different matrix sizes N . It calculates the time per operation and compares it to theoretical performance estimates.
4. **Results Plotting:** The plot results function generates a plot of the GPU performance in GFLOPS versus matrix size N . It also includes a label indicating the maximum GFLOPS achieved.
5. **Data Analysis:** The benchmark results are printed to the console and visualized using the ‘Plots’ library. The maximum GFLOPS achieved is highlighted in the plot for comparison with theoretical performance.

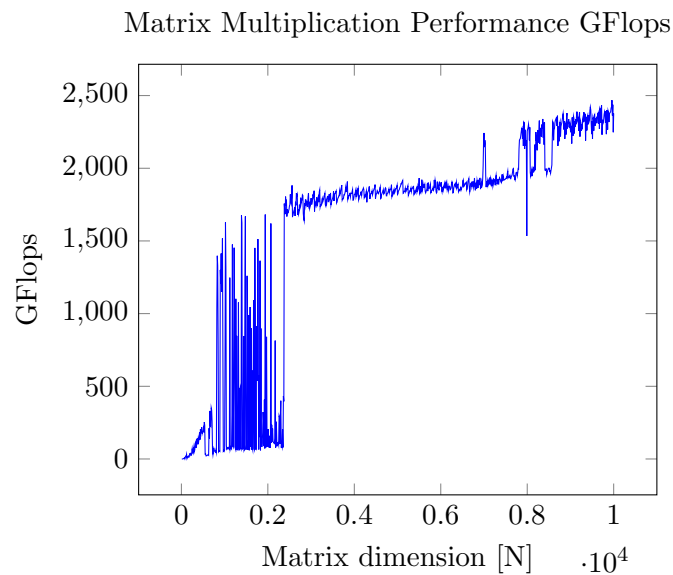


Figure 7: GPU efficiency with matrix multiplications.

Code Here is the Julia code used for this benchmark:

```

1 using LinearAlgebra, CUDA
2
3 # Function to initialize random matrices on the GPU
4 function gpu_matrix_initialization(N)
5     A = CUDA.rand(Float32, N, N)
6     B = CUDA.rand(Float32, N, N)
7     return A, B
8 end
9
10 # Function to perform matrix multiplication using the GPU
11 function gpu_matrix_multiplication(A, B)
12     return A * B
13 end
14
15 # Function to measure the performance of GPU matrix multiplication in GFLOPS
16 function measure_gflops_gpu(N)
17     A, B = gpu_matrix_initialization(N)
18     start_time = time_ns()
19     C = gpu_matrix_multiplication(A, B)
20     CUDA.synchronize() # Ensure all operations are complete
21     elapsed_time = (time_ns() - start_time) / 1e9 # Convert from nanoseconds to
                seconds
22     gflops = (2 * N^3) / (elapsed_time * 1e9) # Compute GFLOPS
23     return gflops
24 end
25
26 # Example usage
27 N = 1024
28 gflops_gpu = measure_gflops_gpu(N)
29 println("GPU matrix multiplication performance: ", gflops_gpu, " GFLOPS")

```

3.5.4 Performance GFLOPS NVIDIA GeForce GTX 1060 6GB

LINK GTX 1060. <https://www.techpowerup.com/gpu-specs/geforce-gtx-1060-6gb.c2862>

The NVIDIA GeForce GTX 1060 6GB is a graphics card from the GTX 10 series, released in 2016, offering significant performance for parallel processing and intensive computation tasks. To evaluate its processing capability, the GFLOPS (Giga Floating Point Operations Per Second) metric is used, which measures the number of floating-point operations the GPU can perform in one second.

Key Specifications To calculate the GFLOPS performance of the GTX 1060 6GB, we consider the following specifications:

- **Number of CUDA Cores:** 1,280
- **Base GPU Clock:** 1,506 MHz
- **Boost GPU Clock:** 1,708 MHz
- **Number of Floating Point Operations per Cycle per Core:** 2 (each CUDA core can perform 2 floating-point operations per cycle)

GFLOPS Calculation GFLOPS performance is calculated using the formula:

$$\text{GFLOPS} = \text{Number of CUDA Cores} \times \text{GPU Clock} \times \text{Operations per Cycle} \times \frac{1}{10^6}$$

where the conversion factor changes the clock frequency from MHz to GHz. Using the boost clock frequency of 1,708 MHz for the calculation, we get:

$$\text{GFLOPS} = 1,280 \times 1,708 \text{ MHz} \times 2 \times \frac{1}{10^3}$$

$$\text{GFLOPS} = 1,280 \times 1.708 \times 2$$

$$\text{GFLOPS} = 4,371.84 \text{ GFLOPS}$$

Summary The theoretical maximum performance of the NVIDIA GeForce GTX 1060 6GB is approximately **4.37 TFLOPS** (teraflops) for single-precision (FP32) floating-point calculations. This figure provides an indication of the GPU's potential to handle floating-point operations in high-performance applications such as scientific simulations, machine learning, and graphics processing.

It is important to note that actual performance may vary depending on the specific workload, software implementation efficiency, and operational conditions of the GPU.

3.5.5 Performance in GFLOPS of the NVIDIA GeForce RTX 2060 Laptop

LINK GTX 1060. <https://www.techpowerup.com/gpu-specs/geforce-rtx-2060.c3310>

The NVIDIA GeForce RTX 2060 Laptop is a high-performance graphics card designed for laptops, offering significant computational power for various demanding tasks. To assess its performance, we use the GFLOPS (Giga Floating Point Operations Per Second) metric, which measures how many floating-point operations the GPU can perform in one second.

Key Specifications For calculating the GFLOPS performance of the RTX 2060 Laptop, we consider the following specifications:

- **Number of CUDA Cores:** 1,920
- **Base GPU Clock:** 1,050 MHz
- **Boost GPU Clock:** 1,440 MHz
- **Number of Floating Point Operations per Cycle per Core:** 2 (each CUDA core can perform 2 floating-point operations per cycle)

GFLOPS Calculation The GFLOPS performance can be calculated using the formula:

$$\text{GFLOPS} = \text{Number of CUDA Cores} \times \text{GPU Clock} \times \text{Operations per Cycle} \times \frac{1}{10^6}$$

where the conversion factor adjusts the clock frequency from MHz to GHz. Using the boost clock frequency of 1,440 MHz for the calculation, we get:

$$\text{GFLOPS} = 1,920 \times 1,440 \text{ MHz} \times 2 \times \frac{1}{10^3}$$

$$\text{GFLOPS} = 1,920 \times 1.44 \times 2$$

$$\text{GFLOPS} = 5,529.60 \text{ GFLOPS}$$

Summary The theoretical maximum performance of the NVIDIA GeForce RTX 2060 Laptop is approximately **5.53 TFLOPS** (teraflops) for single-precision (FP32) floating-point calculations. This figure provides a measure of the GPU's capability to handle floating-point operations efficiently, which is crucial for tasks such as high-resolution gaming, complex simulations, and professional creative applications.

It is important to note that the actual performance may vary based on the specific workload, the efficiency of the software implementation, and the operating conditions of the GPU.

4 Solving the Convective-Diffusive Heat Equation with Numerical Methods

4.1 Introduction and Theoretical framework

The heat equation is a PDE that describes the heat distribution in a region over time. In its most general form, diffusion and convection effects appear, making it a complex problem to simulate numerically. This document aims to address the solution of the heat equation with convective and diffusive terms using different numerical techniques to determine which one achieves greater speed.

In particular, two approaches will be employed: the three-point finite difference method and global interpolation, the latter in two forms, using matrix-vector and matrix-matrix operations. These methods will be implemented on CPU and GPU, evaluating their computational performance in execution times and floating-point operations (flops). The comparative analysis will allow for an evaluation of the speed of each method.

The two-dimensional heat equation, with convection and diffusion terms, is expressed as:

$$\frac{\partial T}{\partial t} + u \frac{\partial T}{\partial x} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right)$$

where $T(x, y, t)$ is the Temperature at point (x, y) at time t , u is the flow velocity in the x -direction, and α is the thermal diffusivity.

A hypothesis was thought about the expected results in performance between writing a code that performs matrix-matrix operations vs matrix-vector operations.

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 more 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 subsection delves into why this disparity occurs, explaining the memory bandwidth limitations and their impact on overall computational performance.

4.1.1 Numerical Methods used

Three-Point Finite Differences The three-point finite differences are based on the discretization of the heat equation on a rectangular grid. The second-order approximation for the second derivative is expressed as follows:

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2}$$

$$\frac{\partial^2 T}{\partial y^2} \approx \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}$$

This approach is simpler to implement. El código empleado para ir a continuación.

Global Interpolantion Global interpolation using Lagrange polynomials is a method used to find a polynomial that passes through a set of points. Given $n+1$ distinct points $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$, where $y_i = f(x_i)$, the goal is to construct a polynomial $P(x)$ of degree n such that:

$$P(x_i) = y_i \quad \text{for } i = 0, 1, \dots, n.$$

Lagrange Interpolating Polynomial

The Lagrange interpolating polynomial $P(x)$ is given by:

$$P(x) = \sum_{i=0}^n y_i \ell_i(x),$$

where $\ell_i(x)$ are the Lagrange basis polynomials, defined as:

$$\ell_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j}.$$

These basis polynomials satisfy two important properties:

- $\ell_i(x_j) = 0$ for $j \neq i$,
- $\ell_i(x_i) = 1$.

Thus, the interpolating polynomial $P(x)$ passes through all the points (x_i, y_i) .

Error of Interpolation

The error of the Lagrange interpolating polynomial is given by:

$$f(x) - P(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^n (x - x_i),$$

where $\xi \in [x_0, x_n]$ is some point in the interval. This shows that the interpolation's accuracy depends on the function's smoothness and the placement of the points x_i .

Example

For three points $(x_0, y_0), (x_1, y_1), (x_2, y_2)$, the Lagrange interpolating polynomial is:

$$P(x) = y_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + y_1 \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + y_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}.$$

This is the polynomial of second degree that passes through $(x_0, y_0), (x_1, y_1), (x_2, y_2)$.

4.2 Computational Implementation

All the code is written in Julia in the IDE Visual Studio Code. Julia is a language designed for numerical analysis and high-performance computing (HPC). Its ease of use, combined with its ability to execute code at speeds comparable to low-level languages like C or Fortran, makes it an interesting choice for scientific computing and large-scale simulations.



Figure 8: Julia and VScode logos

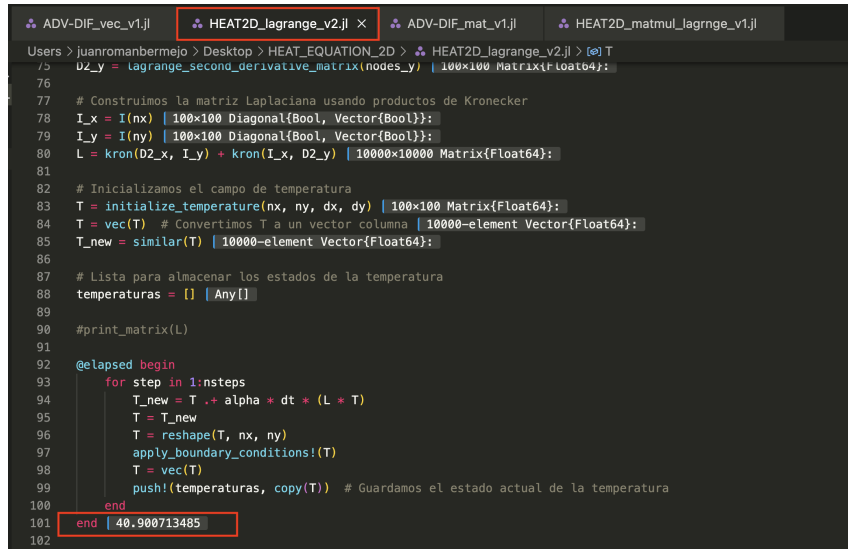
CPU Implementation: The CPU implementation was carried out using BLAS (Basic Linear Algebra Subprograms), which is a highly optimized library for performing common linear algebra operations such as matrix multiplication, vector addition, and matrix factorizations. BLAS is widely used in numerical computing for its efficiency and reliability, and its optimized routines are specifically tailored for performance on modern processors. By leveraging BLAS, the CPU implementation benefits from both multi-threading and memory optimizations, ensuring that the computations are performed as efficiently as possible.

GPU Implementation: For the GPU implementation, CUDA (Compute Unified Device Architecture) will be used, which allows for the parallel execution of tasks on NVIDIA GPUs. CUDA is well-suited for highly parallelizable workloads, such as matrix operations, where thousands of cores on the GPU can work simultaneously to accelerate computations. In the context of this project, CUDA might enable performance gains by offloading the computationally intensive parts of the code, such as matrix-matrix multiplications and vector operations, to the GPU. The use of CUDA could allow the simulation to run faster and handle larger datasets more efficiently compared to a CPU-only implementation.

By utilizing both BLAS on the CPU and CUDA on the GPU, we can compare the performance of each architecture and analyze the trade-offs in terms of speed, and computational efficiency. This approach demonstrates how Julia can seamlessly integrate with these powerful computing libraries to optimize numerical algorithms on both CPUs and GPUs.

4.3 Results and Analysis

The first test was carried out considering only the diffusive term, in a square plate at a certain temperature, where a peak with a higher temperature is introduced. In this case, it was tested by writing it in two ways: using matrix-vector operations (1) and matrix-matrix operations (2).

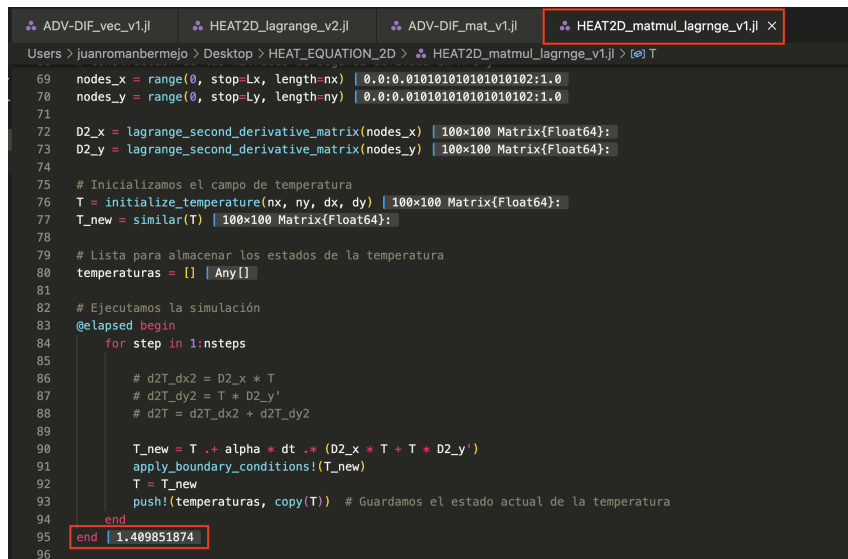


```

ADV-DIF_vec_v1.jl | HEAT2D_lagrange_v2.jl | ADV-DIF_mat_v1.jl | HEAT2D_matmul_lagrange_v1.jl
Users > juanromanbermejo > Desktop > HEAT_EQUATION_2D > HEAT2D_lagrange_v2.jl > T
75 D2_y = lagrange_second_derivative_matrix(nodes_y) | 100x100 Matrix{Float64}:
76
77 # Construimos la matriz Laplaciana usando productos de Kronecker
78 I_x = I(nx) | 100x100 Diagonal{Bool, Vector{Bool}}:
79 I_y = I(ny) | 100x100 Diagonal{Bool, Vector{Bool}}:
80 L = kron(D2_x, I_y) + kron(I_x, D2_y) | 10000x10000 Matrix{Float64}:
81
82 # Inicializamos el campo de temperatura
83 T = initialize_temperature(nx, ny, dx, dy) | 100x100 Matrix{Float64}:
84 T = vec(T) # Convertimos T a un vector columna | 10000-element Vector{Float64}:
85 T_new = similar(T) | 10000-element Vector{Float64}:
86
87 # Lista para almacenar los estados de la temperatura
88 temperaturas = [] | Any[]
89
90 #print_matrix(L)
91
92 @elapsed begin
93   for step in 1:nsteps
94     T_new = T .+ alpha * dt * (L * T)
95     T = T_new
96     T = reshape(T, nx, ny)
97     apply_boundary_conditions!(T)
98     T = vec(T)
99     push!(temperaturas, copy(T)) # Guardamos el estado actual de la temperatura
100   end
101 end | 40.900713485
102

```

Figure 9: Matrix-vector



```

ADV-DIF_vec_v1.jl | HEAT2D_lagrange_v2.jl | ADV-DIF_mat_v1.jl | HEAT2D_matmul_lagrange_v1.jl | X
Users > juanromanbermejo > Desktop > HEAT_EQUATION_2D > HEAT2D_matmul_lagrange_v1.jl > T
69 nodes_x = range(0, stop=Lx, length=nx) | 0.0:0.0101010101010102:1.0
70 nodes_y = range(0, stop=Ly, length=ny) | 0.0:0.0101010101010102:1.0
71
72 D2_x = lagrange_second_derivative_matrix(nodes_x) | 100x100 Matrix{Float64}:
73 D2_y = lagrange_second_derivative_matrix(nodes_y) | 100x100 Matrix{Float64}:
74
75 # Inicializamos el campo de temperatura
76 T = initialize_temperature(nx, ny, dx, dy) | 100x100 Matrix{Float64}:
77 T_new = similar(T) | 100x100 Matrix{Float64}:
78
79 # Lista para almacenar los estados de la temperatura
80 temperaturas = [] | Any[]
81
82 # Ejecutamos la simulación
83 @elapsed begin
84   for step in 1:nsteps
85     # d2T_dx2 = D2_x * T
86     # d2T_dy2 = T * D2_y'
87     # d2T = d2T_dx2 + d2T_dy2
88
89     T_new = T .+ alpha * dt .* (D2_x * T + T * D2_y')
90     apply_boundary_conditions!(T_new)
91     T = T_new
92     push!(temperaturas, copy(T)) # Guardamos el estado actual de la temperatura
93   end
94 end | 1.409851874
95
96

```

Figure 10: Matriz por Matriz

The results are the following:

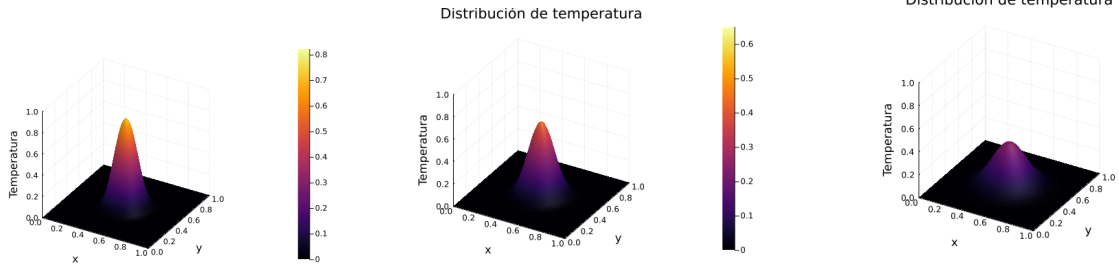


Figure 11: Image 1

Figure 12: Image 2

Figure 13: Image 3

After these results, we attempted a more complex simulation, where we observed the evolution of the temperature around a rectangle at a constant temperature T_{const} , immersed in a flow with a different ambient temperature T_{∞} and a uniform velocity. The first test is done using finite differences, as shown in the figure below($t=0.416$):

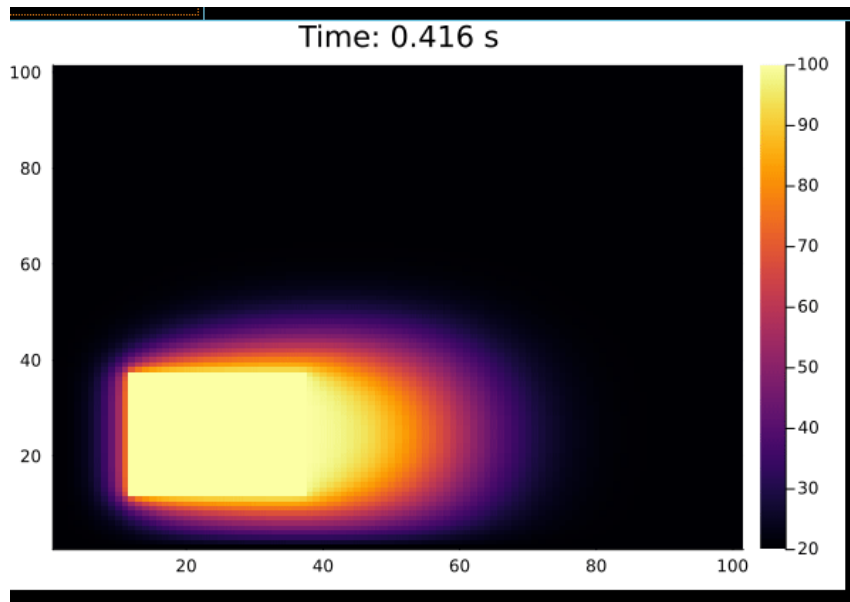


Figure 14: Heat distribution around rectangle ($t=0.416$)

The code used for this is the following:

```

1 using Plots
2 using Printf
3
4 # Parameters
5 nx, ny = 100, 100 # numero de puntos en x e y
6 Lx, Ly = 1.0, 1.0 # dimensiones del dominio
7 dx, dy = Lx/nx, Ly/ny # espaciado de la malla
8 alpha = 0.01 # difusividad termica
9 T_inf = 20.0 # temperatura del aire entrante
10 T_rect = 100.0 # temperatura del rectangulo
11 u = 0.1 # velocidad del flujo de aire (izquierda a derecha)
12
13 # Esabilidad usando CFL
14 dt_diff = (dx^2) / (2 * alpha)
15 dt_conv = dx / u
16 dt_max = min(dt_diff, dt_conv)
17 dt = min(dt_max, 0.0001) # elegir dt estable
18
19 println("Paso temporal: $dt")
20 println("Maximo paso estable: $dt_max")
21
22 # Numero de pasos
23 t_end = 1.0 # tiempo final
24 nsteps = Int(t_end / dt)
25
26 # Inicializacion del campo de temperatura
27 T = fill(T_inf, nx+1, ny+1)
28
29 # Indices del rectangulo
30 rect_x = Int(floor(nx/8)):Int(floor(3*nx/8))
31 rect_y = Int(floor(ny/8)):Int(floor(3*ny/8))
32 T[rect_x, rect_y] .= T_rect
33
34 # Condiciones de contorno
35 function apply_boundary_conditions!(T, T_inf, u, dt, dx)
36     nx, ny = size(T)
37
38     # Inflow (left side)
39     T[1, :] .= T_inf
40
41     # Outflow (right side) - COND CONT Convectiva outflow
42     # for j in 1:ny
43     # T[end, j] = T[end, j] - u * dt / dx * (T[end, j] - T[end-1, j])
44     #end
45
46
47
48
49
50     # Arriba
51     T[:, end] .= T_inf
52
53     # Abajo
54     T[:, 1] .= T_inf
55 end
56
57 # Actualizar el campo de temperatura
58 function update_temperature!(T, alpha, u, dx, dy, dt)
59     nx, ny = size(T)
60     T_new = copy(T)
61     for i in 2:nx-1
62         for j in 2:ny-1

```

```

63         # Conveccion
64         convection = -u * (T[i,j] - T[i-1,j]) / dx
65     #upwinding
66
67     # Tx = Dx T Incluye todas las derivadas (cualquier orden) ver Matvect/ matmat// para
        ver comparacion(tiempo y resultados) u*gradT
68         # Difusion
69
70         # Txx = Dxx T igual evaluar que implementacion va mas rapido Dxx la
        calculas una vez
71
72         diffusion = alpha * (
73             (T[i+1,j] - 2*T[i,j] + T[i-1,j]) / dx^2 +
74             (T[i,j+1] - 2*T[i,j] + T[i,j-1]) / dy^2 )
75
76         T_new[i,j] = T[i,j] + dt * (convection + diffusion)
77     end
78 end
79 return T_new
80 end
81
82 # Temperatura del rectangulo
83 function set_rectangle_temperature!(T, rect_x, rect_y, T_rect)
84     T[rect_x, rect_y] .= T_rect
85 end
86
87 # Configuracion de la visualizacion
88 anim = Animation()
89
90 # Bucle de integracion en el tiempo
91 for step in 1:nsteps
92     # Aplicar condiciones de contorno
93     apply_boundary_conditions!(T, T_inf, u, dt, dx)
94
95     # Actualizar el campo de temperatura
96     global T = update_temperature!(T, alpha, u, dx, dy, dt)
97
98     # Fijar la temperatura del rectangulo
99     set_rectangle_temperature!(T, rect_x, rect_y, T_rect)
100
101     # Visualizacion
102     if step % 10 == 0
103         heatmap(T', c=:inferno, clim=(T_inf, T_rect), title=@sprintf("Time: %.3f s",
            step*dt))
104         frame(anim)
105     end
106 end
107
108 # Guardar la animacion como GIF
109 gif(anim, "heat_transfer_simulation1.gif", fps=10)

```

4.4 Global Interpolation solving

The next step is to solve the simulation using other methods that will allow us to compare implementations that perform matrix-vector operations to matrix-matrix operations. The choice is to solve the equation using a Global Interpolation based on Lagrange polynomials, The Code used is listed below and performs matrix-matrix operations to solve the problem:

```

1 #using Pkg
2 #Pkg.add("Plots")
3 #Pkg.add("Kronecker")
4 using LinearAlgebra, SparseArrays, Kronecker, BenchmarkTools
5 using Plots
6
7 function print_matrix(m)
8     for row in 1:size(m, 1)
9         println(join(m[row, :], " "))
10    end
11 end
12
13 # Parametros del problema
14 nx = 25 # Numero de puntos en la direccion x
15 ny = 25 # Numero de puntos en la direccion y
16 Lx = 1.0 # Longitud del dominio en la direccion x
17 Ly = 1.0 # Longitud del dominio en la direccion y
18 alpha = 0.001 # Difusividad termica
19 v = [0, 0.01] # Vector de velocidad (vx, vy)
20 dt = 0.01 # Paso de tiempo
21 tfinal = 15 # Tiempo final de la simulacion
22 nsteps = Int(tfinal / dt) # Numero de pasos de tiempo
23
24 # Definimos el tamano de la malla
25 dx = Lx / (nx - 1)
26 dy = Ly / (ny - 1)
27
28 # Inicializacion continua de temperaturas
29 function initialize_temperature(nx, ny, dx, dy)
30     T = zeros(nx, ny)
31     for i in 1:nx
32         for j in 1:ny
33             x = (i-1) * dx
34             y = (j-1) * dy
35             T[i,j] = exp(-(25*(x-0.5)^2 + 25*(y-0.5)^2))
36             if T[i,j] < 0
37                 T[i,j] = 0
38             end
39         end
40     end
41     return T
42 end
43
44
45
46 # Funcion para aplicar las condiciones de frontera y mantener la fuente termica fija
47 function apply_boundary_conditions!(T, nx, ny, dx, dy)
48     # Mantener la temperatura fija en los bordes
49     T[1, :] .= exp(-(25*(-0.5)^2))
50     T[end, :] .= exp(-(25*(0.5)^2))
51     T[:, 1] .= exp(-(25*(-0.5)^2))
52     T[:, end] .= exp(-(25*(0.5)^2))
53
54     # Mantener la fuente termica fija en el centro con la condicion inicial
55     # for i in 1:nx
56     #     for j in 1:ny

```



```

57     # x = (i-1) * dx
58     # y = (j-1) * dy
59     # distance = sqrt((x - 0.5)^2 + (y - 0.5)^2)
60     # if distance <= 0.05
61     # #T[i,j] = exp(-(25*(x-0.5)^2 + 25*(y-0.5)^2))
62     # T[i,j] = 1
63     # end
64     # end
65     # end
66 end
67
68
69
70 # Definimos los nodos en x e y
71 nodes_x = range(0, stop=Lx, length=nx)
72 nodes_y = range(0, stop=Ly, length=ny)
73
74
75
76 #
=====
77
78 # Calculo de los polinomios de Lagrange para los x
79 function lagrange_basis(x_nodes, i, x)
80     l_i = 1.0
81     for j in 1:length(x_nodes)
82         if j != i
83             l_i *= (x - x_nodes[j]) / (x_nodes[i] - x_nodes[j])
84         end
85     end
86     return l_i
87 end
88
89 # Funcion para calcular la derivada del polinomio de Lagrange
90 function lagrange_derivative(x_nodes, i, x)
91     dl_i = 0.0
92     for m in 1:length(x_nodes)
93         if m != i
94             term = 1.0 / (x_nodes[i] - x_nodes[m])
95             for j in 1:length(x_nodes)
96                 if j != i && j != m
97                     term *= (x - x_nodes[j]) / (x_nodes[i] - x_nodes[j])
98                 end
99             end
100             dl_i += term
101         end
102     end
103     return dl_i
104 end
105
106 # Funcion para calcular la matriz de derivadas usando polinomios de Lagrange
107 function lagrange_derivative_matrix(x_nodes)
108     n = length(x_nodes)
109     D = zeros(n, n)
110     for i in 1:n
111         for j in 1:n
112             D[i, j] = lagrange_derivative(x_nodes, j, x_nodes[i])
113         end
114     end
115     return D
116 end
117
118

```

```

119 # Definimos los nodos en x e y
120 nodes_x = range(0, stop=Lx, length=nx)
121 nodes_y = range(0, stop=Ly, length=ny)
122
123 # Construccion de las matrices de derivadas con POLINOMIOS DE LAGRANGE
124 D_x = lagrange_derivative_matrix(nodes_x)
125 D_y = lagrange_derivative_matrix(nodes_y)
126
127 # Las matrices de segunda derivada son simplemente el producto de la matriz de primera
    derivada con ella misma
128 D2_x = D_x * D_x
129 D2_y = D_y * D_y
130
131 #
    =====
132
133
134
135
136 # Inicializamos el campo de temperatura
137 T = initialize_temperature(nx, ny, dx, dy) # Convertimos T a un vector columna
138 T_new = similar(T)
139
140 # Lista para almacenar los estados de la temperatura
141 temperaturas = []
142
143
144 # Medir el tiempo del bucle
145 @elapsed begin
146     for step in 1:nsteps
147
148         # difusion = alpha * (Laplacian * T)
149         # adveccion = -(Gradient * T)
150
151         T_new = T .+ dt .* (alpha * (D2_x * T + T * D2_y') - (v[1] * (D_x * T) + v[2]
            * (T * D_y')))
152
153         global T = T_new
154         apply_boundary_conditions!(T, nx, ny, dx, dy)
155         push!(temperaturas, copy(T)) # Guardamos el estado actual de la temperatura
156     end
157 end
158
159
160 ## === REPRESENTACION GRAFICA === ##
161
162 # Convertimos el resultado a una matriz para mostrarlo
163 T = reshape(T, nx, ny)
164
165 # Determinamos los limites de los ejes
166 x = range(0, stop=Lx, length=nx)
167 y = range(0, stop=Ly, length=ny)
168 z_min, z_max = 0, 1.0 # Limites del eje z (temperatura)
169
170 # # Creamos la animacion 3D
171 # intervalo_animacion = 30
172 # animation = @animate for i in 1:intervalo_animacion:length(temperaturas)
173 # t = temperaturas[i]
174 # surface(x, y, reshape(t, nx, ny), title="Distribucion de temperatura [M][M]", xlabel
    ="x", ylabel="y", zlabel="Temperatura", c=:inferno, xlims=(0, Lx), ylims=(0, Ly),
    zlims=(z_min, z_max))
175 # end
176

```

```

177 # # Guardamos la animacion como un gif
178 # gif(animation, "adveccion_difusion_diferencias_finitas_matxmat_3d.gif", fps=30)
179
180
181 # Creamos la animacion 2D con lineas de contorno sin letras
182 intervalo_animacion = 30
183 animation = @animate for i in 1:intervalo_animacion:length(temperaturas)
184     t = temperaturas[i]
185     contourf(x, y, reshape(t, nx, ny), c=:inferno, xlims=(0, Lx), ylims=(0, Ly), zlims
186               =(z_min, z_max), xlabel="", ylabel="", title="", clabels=false)
187 end
188 # Guardamos la animacion como un gif
189 gif(animation, "adveccion_difusion_contornos2d.gif", fps=10)

```

Now with a matrix-vector approach:

1. Heat Diffusion

The heat diffusion equation describes how heat spreads through the medium over time due to thermal diffusivity. Mathematically, it is expressed as:

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right)$$

where:

- $T(x, y, t)$ is the temperature field,
- α is the thermal diffusivity, which determines the rate at which heat diffuses through the material,
- $\frac{\partial^2 T}{\partial x^2}$ and $\frac{\partial^2 T}{\partial y^2}$ are second-order spatial derivatives representing the diffusion of heat in the x and y directions.

In this simulation, the diffusion term is discretized using **Lagrange polynomial interpolation** to approximate the second-order derivatives in both directions.

2. Advection

Advection describes the transport of heat due to fluid motion, which can be represented as:

$$-\mathbf{v} \cdot \nabla T = -v_x \frac{\partial T}{\partial x} - v_y \frac{\partial T}{\partial y}$$

where:

- $\mathbf{v} = (v_x, v_y)$ is the velocity field, representing the movement of the heat due to fluid flow,
- ∇T is the gradient of the temperature, representing the directional change in temperature.

In this simulation, the velocity vector $\mathbf{v} = [0, 0.01]$ causes heat to be advected primarily in the y -direction (upward).

3. Initial and Boundary Conditions

- **Initial Condition:** The temperature field is initialized with a Gaussian-like heat distribution centered at the middle of the domain. This represents a concentrated heat source that spreads out over time.
- **Boundary Conditions:** The domain has fixed temperature values at its boundaries (Dirichlet conditions). These remain constant throughout the simulation, enforcing a specific temperature on the edges of the domain.

4. Numerical Method

The simulation uses **Lagrange polynomials** to discretize the spatial derivatives in both the x - and y -directions. This allows us to approximate the spatial derivatives necessary for solving the heat equation.

At each time step, the temperature field is updated using the following equation:

$$T_{\text{new}} = T + \Delta t \left(\alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) - v_x \frac{\partial T}{\partial x} - v_y \frac{\partial T}{\partial y} \right)$$

where Δt is the time step size, α controls the rate of heat diffusion, and v_x and v_y represent the velocities in the x - and y -directions (advection).

5. Visualization

The simulation results are visualized as contour plots, which show the evolving temperature field over time. The animation demonstrates how the heat diffuses across the domain while being transported upward by the advection process.

Key Physical Concepts

- **Heat Diffusion:** The process by which heat spreads from areas of high temperature to low temperature.
- **Advection:** The transport of heat due to the fluid motion, represented by the velocity vector \mathbf{v} .
- **Lagrange Polynomial Interpolation:** Used to discretize the spatial derivatives of the temperature field.
- **Boundary Conditions:** Fixed temperature values are enforced on the domain boundaries.

This combination of heat diffusion and advection captures the behavior of heat flow in systems with fluid motion and thermal effects. The numerical method used here allows for the accurate simulation of temperature evolution in the domain.

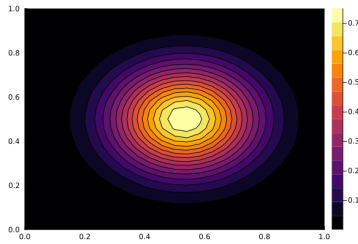


Figure 15: $t = 0$ s

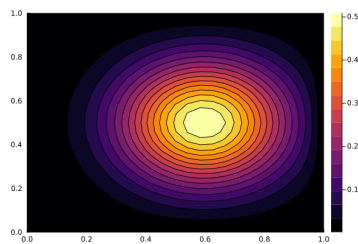


Figure 16: $t = 7.5$ s

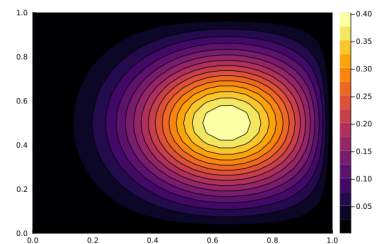


Figure 17: $t = 15$ s

Extra comparison As an extra step, this code written by Javier solves the equation in 1D, 2D, or 3D , we will compare it to the other results

```

1 using LinearAlgebra, GLMakie
2
3
4 function heatpropagate(u::AbstractArray{T,N}, timesteps::Integer,  $\alpha\Delta t\Delta x^{-2}$ ::Real,
5     uprev::AbstractArray{T,N}) where {T<:Number,N}
6
7      $\alpha\Delta t\Delta x^{-2} \leq 0.5 / N$  || throw(ArgumentError(" $\alpha\Delta t\Delta x^{-2} = \$\alpha\Delta t\Delta x^{-2}$  violates Fourier
8         stability condition"))
9
10    unitvecs = ntuple(i -> CartesianIndex(ntuple(==(i), Val(N))), Val(N))
11
12    I = CartesianIndices(u)
13    Ifirst, Ilast = first(I), last(I)
14    I1 = oneunit(Ifirst)
15
16    for t = 1:timesteps
17        @inbounds @simd for i in Ifirst+I1:Ilast-I1
18
19             $\nabla^2 u = -2N * u[i]$ 
20            for uvec in unitvecs
21                 $\nabla^2 u += u[i+uvec] + u[i-uvec]$ 
22            end
23
24            u[i] = uprev[i] +  $\alpha\Delta t\Delta x^{-2} * \nabla^2 u$ 
25
26        end
27
28        # Here Boundary conditions would be imposed (except if they are Dirichlet at
29        # boundary)
30
31        uprev = u
32    end
33    return u, uprev
34 end
35
36 begin
37     x = range(-1.0, 1.0, length=100)
38     u = @. exp(-(x / 0.2)^2)
39     u[1] = u[end] = 0.0
40
41     set_publication_theme!()
42
43     lines(x, u)
44
45      $\alpha = 1.0$ 
46      $\Delta x = x[2] - x[1]$ 
47      $Fo = 1 / 2$ 
48      $\Delta t = Fo * \Delta x^2 / \alpha$ 
49
50     uprev = copy(u)
51 end
52
53 # @time heatpropagate(u, 20, Fo, uprev);
54
55 for i = 1:50
56     u, uprev = heatpropagate(u, 20, Fo, uprev)
57     lines!(x, u)
58 end

```

4.4.1 Graphs and tables

Graphs and tables summarizing the results obtained, allowing a comparison between the different methods and/or hardware platforms.

Method	Simulation Time (seconds)
Global Interpolant (matrix-matrix)	0.127
Global Interpolant (matrix-vector)	2.789
Finite Differences	0.646
Javier's Code	??

Table 1: Comparison of methods and expected simulation times

4.5 Conclusions

This study researches different numerical methods to solve the heat equation with convective and diffusive terms. The aim is to compare the different approaches and verify that the theoretical hypothesis are approximately proven.

clearpage

4.6 Bibliografía

- Juan A. Hernandez Ramos ´ Mario A. Zamecnik Barros. *Interpolación Polinómica de Alto orden, Métodos Espectrales*
- Smith, G. D. (1985). *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press.
- LeVeque, R. J. (2007). *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. Society for Industrial and Applied Mathematics.
- Kirk, D. B., & Hwu, W.-m. W. (2016). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann.
- Román Bermejo, J. (2024). *CPU Performance: Benchmark Analysis and Theoretical Limits*.
- Boyd2001 J. P. Boyd. (2001) *Chebyshev and Fourier Spectral Methods*, 2nd edition, Dover Publications.