

uc3m

Universidad
Carlos III
de Madrid

Bachelor's degree in Computer Science and Engineering

Bachelor Thesis

“Evaluating performance and energy
impact of programming languages”

Author

Eduardo Alarcón Navarro

Advisor

Jose Daniel García Sánchez

Leganés, Madrid, Spain

September 2025



This work is licensed under Creative Commons

Attribution - Non Commercial - Non Derivatives

To reach the moon, you should aim for the stars.

—

ACKNOWLEDGEMENTS

I would like to thank my family, for their continuous support and encouragement, specially this last year, where it has been the hardest for all of us. Not only they have provided me with the best education possible, financially and emotionally, but they have also taught me the importance of hard work and dedication.

I would like to thank my parents, my father for always being there when I needed him, for listening to me rant about many university projects, or crazy ideas. Furthermore, I would also like to thank my mother, for always being there for me, and for being the best mother I could have ever asked for, and for always supporting me in everything I do, even if we don't always agree on everything.

I would not have been able to finish my degree without the help of my friends, who have always been there for me, adopting me as part of their family and being there when I needed them most. During the four years of my degree, I have met many people, some have persevered in completing their degree, while others have chosen to take a different path in life. I would like to thank all of them for being there for me, and for being part of my life.

Another person I would like to thank is my thesis advisor Jose Daniel. Thank you for giving me the opportunity to work with you, after an extreme late request for a thesis. I am grateful for the trust you placed in me from the very beginning and for guiding me when I was most lost. From the admiration, it has been a great pride to close this stage with you as my advisor.

ABSTRACT

Nowadays, the importance of energy efficiency is increasing as more computationally expensive programs are being used by more and more people. The energy impact of running these programs is directly related with the programming language used to create the program as well as the design specifics.

This thesis aims to bring a specific example of the power efficiency of three programming languages: Python, Go and C++. Each one having its differences and properties, ease of use and execution speed. Each language was chosen for its distinctive characteristics, representing its respective category of programming languages.

Compiled languages with no garbage collection and no managed runtime have usually had the best execution speed as they can reach byte-code for each specific platform, but in the last years, other methods have improved significantly, such as the use of a Just In Time Compiler (JIT).

To achieve realistic results, these languages were tested in multiple configurations, on different hardware, core count and operating systems to be able to eliminate any outliers.

Thus, this work will try showing the differences in energy consumption of different programming languages in a real world task, rendering a ray-traced image of multiple spheres with different materials and reflectivity.

Keywords: Energy Efficiency • Performance • Programming Languages • Ray-Tracing

CONTENTS

Chapter 1. Introduction	1
1.1. Motivation	1
1.2. Goals	2
1.3. Document Structure	2
Chapter 2. State of the Art	5
2.1. Energy Efficient Systems	5
2.2. System Architectures	5
2.2.1. x86 Architecture	6
2.2.2. ARM Architecture	6
2.2.3. CPU Design for Multi-core dies	7
2.2.4. Hyperthreading	8
2.2.5. GPU and Accelerators	8
2.3. Programming Languages	9
2.3.1. Go: Compiled Language with an Embedded Managed Runtime	9
2.3.2. Python: Interpreted Programming Language	13
2.3.3. C++: Directly Compiled, Unmanaged Language	16
2.3.4. Other languages not used	17
2.4. Previous Benchmarks	18
Chapter 3. Problem Statement & Analysis	21
3.1. Project Description.	21
3.2. Requirements	22
3.2.1. Functional requirements	23
3.2.2. Non-Functional requirements	28
3.3. Use Case	30
3.4. Traceability	34
3.5. System Architecture	36
Chapter 4. Design and Implementation	39
4.1. General Program Design	39
4.2. Scene	41
4.2.1. Sphere_data design.	41
4.2.2. Language Specific	42
4.3. Object	44

4.4. Renderer	45
4.4.1. Multiprocessing	46
4.5. Output	48
4.6. Running the program	49
Chapter 5. Evaluation	55
5.1. Measurement Platforms	55
5.1.1. Many-Core Platform	55
5.1.2. Desktop	62
5.1.3. Laptop	68
5.1.4. Raspberry Pi 5	72
5.2. Comment on parallelizing different languages	75
5.2.1. Go	75
5.2.2. Python	76
5.3. Most efficient language optimizations	76
Chapter 6. Planning	79
6.1. Initial Plan	79
Chapter 7. Socioeconomic Environment & Sustainable Development Goals	83
7.1. Budget	83
7.1.1. Human Resources	83
7.1.2. Material Resources	84
7.1.3. Software	84
7.1.4. Indirect Costs	85
7.1.5. Total Cost & Offer	85
7.2. Socioeconomic Impact	86
7.3. Sustainable Development Goals	86
Chapter 8. Regulatory Framework	89
8.1. Software Licenses	89
8.2. Technical Standards	90
8.3. License of the Project	90
Chapter 9. Conclusions	91
9.1. Summary of Findings	91
9.2. Discussion of Results	92
9.3. Future Research	94
Bibliography	95
Appendix A. Server Benchmarks Tables Results	
Appendix B. Desktop Tables Results	

Appendix C. Laptop Benchmarks Tables Results

Appendix D. Raspberry Pi Benchmarks Tables Results

Appendix E. Goroutines Benchmarks

Appendix F. Compiler Optimizations Tables Results

Appendix G. Number of Benchmarks for average calculation

Appendix H. Scatter plot with python

Appendix I. Declaration of Use of Generative AI

LIST OF FIGURES

1.1	Global electricity consumption by source that produced it in 2023	2
2.1	Intel Xeon Skylake-X CPU distribution of cores and caches. Figure adapted from [11]	7
2.2	AMD's ZEN architecture	8
2.3	Python's Execution loop	14
3.1	Use-case diagram for the Ray-Tracer Benchmark	31
3.2	System Architecture Diagram for the Ray Tracer.	36
4.1	General program data flow	40
4.2	File structures for Ray-Tracer implementations in different languages . .	40
4.3	Sample program output high-resolution	49
4.4	Sample program output default configuration	50
5.1	Server - Logarithmic energy (pkg) consumption	58
5.2	Server - Logarithmic energy (ram) consumption	58
5.3	Server - Logarithmic Execution Time	59
5.4	Server - Energy package relative efficiency	59
5.5	Server - Energy RAM relative efficiency	60
5.6	Server - Execution time speedup	60
5.7	Core usage per pixel renderer	62
5.8	Desktop - Logarithmic energy consumption	63
5.9	Desktop - Linear energy consumption	64
5.10	Desktop - Logarithmic package energy consumption without Hyperthreading	64
5.11	Desktop - Linear package energy consumption without Hyperthreading .	65
5.12	Desktop - Logarithmic Execution Time	65
5.13	Desktop - Linear execution Time	66
5.14	Desktop - Logarithmic Execution Time without Hyperthreading	66
5.15	Desktop - Linear execution Time without Hyperthreading	67
5.16	Desktop - Energy relative efficiency	67
5.17	Desktop - Execution time speedup	68
5.18	Laptop - Logarithmic energy consumption	69
5.19	Laptop - Linear energy consumption	70
5.20	Laptop - Logarithmic Execution Time	70

5.21	Laptop - Linear execution Time	71
5.22	Laptop - Energy relative efficiency	71
5.23	Laptop - Execution time speedup	72
5.24	Raspberry Pi - Logarithmic energy consumption	73
5.25	Raspberry Pi - Logarithmic Execution Time	73
5.26	Raspberry Pi - Energy package relative efficiency	74
5.27	Raspberry Pi - Execution time speedup	74
5.28	Relative performance improvements showing time scales better than en- ergy efficiency	75
5.29	Execution time for multiple compiler flags	77
5.30	Package consumption for multiple compiler flags	78
5.31	RAM consumption for multiple compiler flags	78
6.1	Gantt Diagram	80
9.1	Scatter plot of C++, Go, and PyPy.	92
G.1	Variation of results with respect to the number of benchmark iterations . .	
G.2	Number of benchmarks for average calculation	
H.1	All languages. Colors = language; labels = platform-cores.	

LIST OF TABLES

2.1	Comparison of C++, Python, and Go General Characteristics	10
2.2	Comparison of Language Characteristics Impacting Energy Efficiency . .	11
2.3	Alternative Python Implementations	16
2.4	Languages Excluded from the Study and Justification	18
3.1	Requirement format template	22
3.2	Requirement RF-01	23
3.3	Requirement RF-02	23
3.4	Requirement RF-03	23
3.5	Requirement RF-04	24
3.6	Requirement RF-05	24
3.7	Requirement RF-06	24
3.8	Requirement RF-07	25
3.9	Requirement RF-08	25
3.10	Requirement RF-09	25
3.11	Requirement RF-10	26
3.12	Requirement RF-11	26
3.13	Requirement RF-12	26
3.14	Requirement RF-13	27
3.15	Requirement RF-14	27
3.16	Requirement RF-15	27
3.17	Requirement RF-16	28
3.18	Requirement rf-17	28
3.19	Requirement RNF-01	28
3.20	Requirement RNF-02	29
3.21	Requirement RNF-03	29
3.22	Requirement RNF-04	29
3.23	Requirement RNF-05	30
3.24	Requirement RNF-06	30
3.25	Requirement RNF-07	30
3.26	Use Case UC-XX	31
3.27	Use Case UC-01	32
3.29	Use Case UC-03	33
3.28	Use Case UC-02	33
3.30	Use Case UC-04	34

3.31	Traceability matrix for functional requirement	35
3.32	Traceability matrix for functional requirement	35
3.33	Diagram Key for System Architecture	37
4.1	PPM file header fields	49
5.1	Language versions and compilers per platform	55
7.1	Human Costs	83
7.2	Material Resources Costs	84
7.3	Software Licensing Costs	85
7.4	Project Information	85
7.5	Cost Breakdown	86
A.1	Server - Package energy consumption	
A.2	Server - Random Acces Memory (RAM) energy consumption	
A.3	Server - Execution Time	
B.1	Desktop - Package energy consumption with Hyperthreading	
B.2	Desktop - Package energy consumption	
B.3	Desktop - Execution Time with Hyperthreading	
B.4	Desktop - Execution Time	
C.1	Laptop - Package energy consumption	
C.2	Laptop - Execution Time	
D.1	Raspberry Pi - Package energy consumption	
D.2	Raspberry Pi - Execution Time	
E.1	Go goroutines and cores	
F.1	Power/Energy and execution time for various core counts and compiler optimization flags	

CHAPTER 1

INTRODUCTION

1.1. Motivation

Energy consumption in the software industry has been raising over the years up to a point that it is now significant at a world energy consumption.

As [1] states, in 2018 an estimated 1% of total energy consumption was attributed to datacenters alone. In 2024, it is estimated that about 1.5% of the world's energy consumption is to be blamed on data centers and server farms. These numbers may not represent much, but from the total 183,230 TWh produced in 2023 [2] only 23,746 TWh come from a renewable source as it can be seen from Figure 1.1, which comes to 12.96%.

Knowing which language to use for each project is decisive not only in regard to the expertise one or their team might have on that language, but also the performance and language characteristics. If you want to develop a high-performance stock trader you would never think about using a high level language such as Python or Perl, but you would try sticking to compiled languages such as C, C++ or Rust.

Thus, the main motivation for this project lies in studying 3 different programming languages, with each one having peculiar characteristics, to test their respective speed and power consumption in different platforms and architectures.

This comes from the idea that the program efficiency does not come from the language itself, but the implementation of the algorithm that the programmer chooses. The language helps, but choosing the optimal algorithm is much more important.

My personal take in this project comes from my hesitation in choosing what programming language to use in many projects, their benefits and drawbacks, and the fact that I have been using many of these languages in multiple courses along these 4 years has made me realize the importance of choosing the correct language for each problem.

2023

in terawatt-hours

Other renewables	2,428 TWh
Modern biofuels	1,318 TWh
Solar	4,264 TWh
Wind	6,040 TWh
Hydropower	11,014 TWh
Nuclear	6,824 TWh
Natural gas	40,102 TWh
Oil	54,564 TWh
Coal	45,565 TWh
Traditional biomass	11,111 TWh
Total	183,230 TWh

Fig. 1.1. Global electricity consumption by source that produced it in 2023**1.2. Goals**

The main goal of this project is the study and analysis of three implementations of a ray-tracer program, measuring the energy consumption as well as the time each program takes to complete. It should be also noted that the platform in which the program is being run affects the energy consumption of the program.

To perform this, I have improved the code from a well-known book called Ray Tracing in One Weekend [3], translating it to go and python, updating the code so that it could handle parallel rendering.

Once the code is created, the methodology for testing the different codes need to also be created.

- x86 Intel Xeon Based (2x Intel Xeon Gold 6326)
- x86 Zen 2 (AMD Ryzen 7 3800x)
- ARM Apple Icestorm & Firestorm (M4 Pro 12 Core)
- ARM Cortex-A76 CPU - Raspberry Pi 5

1.3. Document Structure

The document contains the following chapters:

- Chapter 1, *Introduction*, details the motivation of the project.

- Chapter 2, *State of the Art*, describes the main points of interest in order to fully understand the project. Theoretical and technological issues are addressed.
- Chapter 3, *Problem Statement & Analysis*, states the project's goals, objectives and requirements.
- Chapter 4, *Design and Implementation*, describes the most relevant design decisions with the multi-language renderers and their multithreaded implementation.
- Chapter 5, *Evaluation*, analyzes and discusses the results benchmarks.
- Chapter 7, *Socioeconomic Environment & Sustainable Development Goals*, provides a comprehensive account of the project's developmental costs and its associated socio-economic implications.
- Chapter 6, *Planning*, describes the organization of the project along the development.
- Chapter 8, *Regulatory Framework*, indicates the licenses under which the project is distributed.
- Chapter 9, *Conclusions*, briefly analyzes the results obtained and states the possible future objectives of the project.

CHAPTER 2

STATE OF THE ART

This chapter describes the paradigms and characteristics of different programming languages. Thus concepts of compiled languages, interpreters optimizations and parallelism are discussed with respect of the different programming languages.

The purpose is to provide background information necessary to understand the study and present a clear justification for the decisions made

2.1. Energy Efficient Systems

An energy efficient system is defined as a system designed and optimized to performs its functions while consuming the minimum amount of energy possible, without compromising its performance, safety and reliability.

As Muralidhar et al. [4] put it, the average power a system draws is:

$$P_{avg} = P_{dynamic} + P_{leakage}$$

The dynamic power depends on the V supply, the clock frequency, the node capacitance and the switching activity. This power can be reduced by reducing the load on the chip or by manually setting a limit on how much voltage the chip can draw (known as undervoting [5])

2.2. System Architectures

While the energy efficiency of a system is significantly affected by connected devices (e.g., a graphics card or an AI Accelerator), this study excludes any external devices and expansion cards. Therefore, the processor architecture is the primary factor determining

energy consumption on the system.

2.2.1. x86 Architecture

The x86 architecture is the most widely adopted in the world of desktop and server computers, whose market share is almost entirely shared by [AMD](#) and [Intel](#) who created it in 1978. Originally called x86-16, due to the 16 bit word size, it debuted in the Intel 8086 a single core, a $3\mu m$ node processor.

Nowadays, the technology has improved, the architecture is now called x86-64, a 64 bit extension, created by AMD, and releases the full specification in August 2000. From 2006 onward, the two companies have been developing multicore processors, adding further Single Instruction Multiple Data (SIMD) Extensions such as AVX-512 [6]. Then came the integrated graphics and finally Power Efficiency Focus.

Then, a new paradigm came, instead of having a homogeneous set of cores, cores focused on performance and efficiency were added to the same package, the hybrid architecture. This set of heterogeneous cores meant the scheduler had to be changed in the operating systems, to better allocate more demanding programs on high performing cores and lower important tasks, such as background jobs to the highly efficient cores. This technology was released by Intel on the 12th generation Intel Core processors, using Intel 7 (a $7nm$ node). This approach was revolutionary for power efficiency as Padoin et al. [7] state.

2.2.2. ARM Architecture

The ARM (Advanced RISC Machine) is the newest architecture that has reached the global scale. Developed in 1986, the goal of this new 32 bit architecture was the simplicity. As Moir [8] puts it, the energy efficient came later. This allowed the ARM architecture to dominate on the mobile sector, specially on smartphones, which run on batteries.

The characteristics of this Instruction Set Architecture (ISA) are a reduced set of instructions (RISC), which allows processors to have fewer transistors than Complex Instruction Set Computer (CISC) architectures such as x86, resulting in lower cost, lower temperatures and lower power consumption.

Currently, this technology is not only used in low-power light devices, but many laptops, and even desktops are using ARM chips due to their power efficiency and performance [9].

ARM also has a hybrid technology, called big.LITTLE, as described by the [10] ARM White Paper that combines high-efficiency cores and high-performance cores. This architecture dominates the mobile device market and is increasingly found in modern laptops.

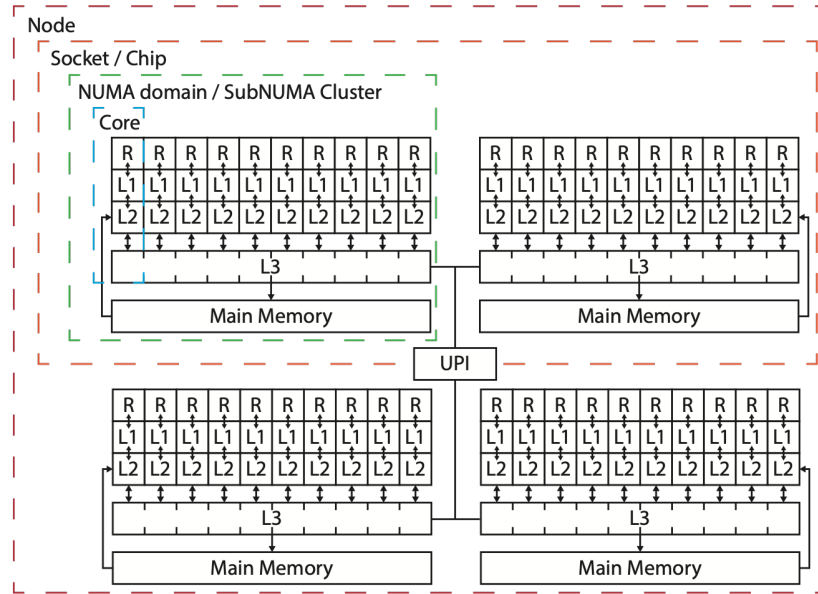


Fig. 2.1. Intel Xeon Skylake-X CPU distribution of cores and caches. Figure adapted from [11]

2.2.3. CPU Design for Multi-core dies

Is it important to note that a computer that reports having more than 32 logical cores, usually has more than one socket, thus the performance and scaling of programs on multiple sockets can affect the energy efficiency and performance. This is due to the fact that information has to move between the multiple cache levels.

From [11]’s Table 2.1 we can see there are multiple configurations, depending on the architecture, the amount of cache per core, how many cores there are per chip and the memory bandwidth.

The traditional layout of these Central Processing Units (CPUs) can be found from Figure 2.1, where each ten cores form a Non-Uniform Memory Access (NUMA) domain, two NUMA domains for each of the chips (sockets) and multiple chips (two in this case) form a NUMA node.

AMD introduced an additional hierarchy level with its Zen architecture: Core Complex (CCX). A single core complex has up to four cores, with a sliced victim L3 cache. Two CCXs are combined onto a die, sharing a local partition of main memory. Multiple dies make up a socket, as it can be seen from Figure 2.2.

Using AMD’s ZEN 2 architecture as a new CPU architecture example, we can observe there are is an additional layer, compared to Figure 2.1. As Gibbs [12] puts it in their paper, in Figure 2.2, the victim of this design is the L3 cache, which is shared between fewer cores.

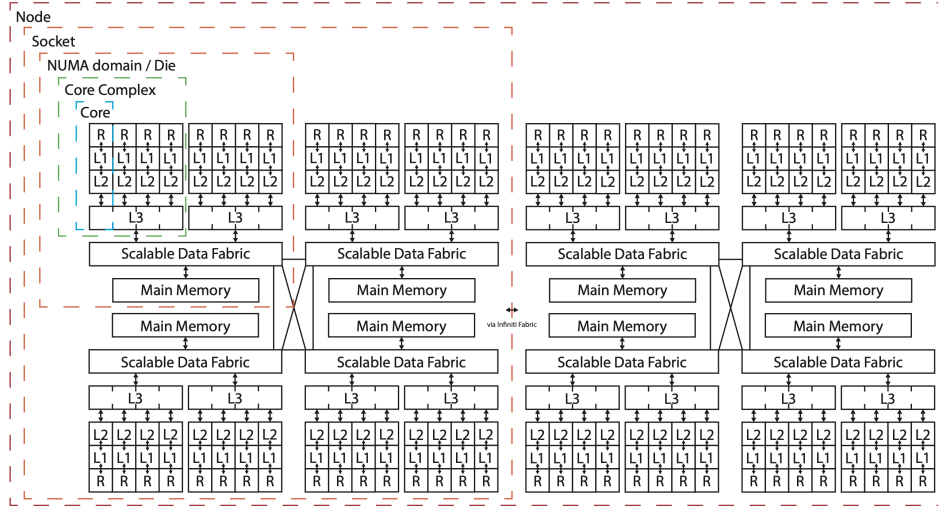


Fig. 2.2. AMD's ZEN architecture

2.2.4. Hyperthreading

Hyperthreading is a technology that allows a single physical core to appear as two logical cores to the operating system. This means that the operating system can schedule two threads on a single physical core, allowing for better utilization of the CPU resources. This usually entails a better performance, as described by Leng et al. [13], but when CPU intensive benchmarks are run, such as integer sorting, the performance can be worse than running the same program on a system that does not have hyperthreading enabled. This is due to the two threads sharing the same resources, such as the cache and the execution units, which can lead to contention and reduced performance.

2.2.5. GPU and Accelerators

The GPU is a specialized processor designed to handle complex mathematical calculations, particularly those related to graphics rendering. It is optimized for parallel processing, allowing it to perform many calculations simultaneously. This makes it ideal for tasks such as image processing, machine learning, and scientific simulations. Most games and image renderers run using a GPU acceleration and most AI workloads can also be accelerated by it.

We can see from this paper [14], the fields where GPUs are used are medical image processing, matrix-related computation, artificial intelligence, deep learning, etc. an acceleration of up to 50x can be obtained on specific kernels, compared to a CPU.

This would be the ideal architecture to implement our testing framework if what we were looking for is extreme performance, but, as the goal of this project is to analyze the energy efficiency of different programming languages, the GPU is not being considered for this project.

An accelerator is a hardware component that enhances the performance of a system by offloading specific tasks from the CPU. Accelerators can take many forms, including GPUs, Field Programmable Gate Array (fpga)s, and application-specific integrated circuits (ASICs). These devices are designed to perform particular computations more efficiently than a general-purpose CPU, making them valuable for tasks such as machine learning, data processing, and scientific simulations. They were very popular in the past, but their use has declined with the rise of more powerful CPUs.

CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows developers to use a C-like programming language to write algorithms that can be executed on the GPU, enabling significant performance improvements for compute-intensive tasks. CUDA provides a rich set of libraries and tools for optimizing performance, making it a popular choice for high-performance computing applications. However, we will not use CUDA in this project as the focus is on analyzing the energy efficiency of programming languages rather than leveraging GPU acceleration.

OpenCL

OpenCL (Open Computing Language) is an open standard for parallel programming of heterogeneous systems. It allows developers to write programs that can execute across various devices, including CPUs, GPUs, and other accelerators. OpenCL provides a C-like programming language and a set of APIs for managing parallel tasks, making it a versatile choice for high-performance computing applications. A future project could be analyzing the energy efficiency of CUDA programs vs OpenCL programs, as they are both parallel programming models for GPU acceleration.

2.3. Programming Languages

In this section, the different programming languages that have been chosen will be discussed, as well as why other similar languages were not.

2.3.1. Go: Compiled Language with an Embedded Managed Runtime

Go is a language developed by [Google](#), released in 2009, focused on concurrency. It has a runtime which manages the goroutines. As described by [15], Go has a Garbage Collector, which means while the program is running, there needs to be a thread checking for unused memory structures.

This language was designed for backend tasks, handling thousands of simultaneous

TABLE 2.1

COMPARISON OF C++, PYTHON, AND GO GENERAL CHARACTERISTICS

Characteristic	C++	Python	Go
Typing System	Statically Typed: Types are checked at compile-time, catching errors early and aiding optimization.	Dynamically Typed: Type checking occurs at runtime. Optional static type hinting (PEP 484) is available.	Statically Typed: Types are checked at compile-time, ensuring type safety and early error detection.
Compilation & Execution	Compiled: Code is compiled directly to native machine code for fast execution.	Interpreted: Typically compiled to bytecode which is then executed by a VM.	Compiled: Code is compiled directly to a self-contained native machine code executable with a runtime.
Concurrency	Provides primitives like threads and mutexes, requiring manual management.	Offers threading (limited by the GIL for CPU-bound tasks) and multiprocessing libraries.	Built-in support with lightweight goroutines and channels managed by the Go runtime.
Memory Management	Manual memory management, with modern C++ heavily relying on RAII and smart pointers.	Automatic via reference counting and a cyclic garbage collector.	Automatic via a concurrent, tri-color mark and sweep garbage collector.
Standard Library	Rich library with a focus on performance (e.g., STL containers, algorithms).	Extensive "batteries-included" library for a vast array of tasks, speeding up development.	Comprehensive library designed for modern needs like networking, I/O, and JSON handling.
Programming Paradigms	Multi-paradigm: Supports procedural, object-oriented (oop), and generic.	Multi-paradigm: Supports procedural, object-oriented, and functional styles.	Primarily procedural and concurrent. Uses composition over inheritance (no classes).

TABLE 2.2

COMPARISON OF LANGUAGE CHARACTERISTICS IMPACTING ENERGY EFFICIENCY

Characteristics Impacting Performance & Energy Efficiency			
Characteristic	C++	Python	Go
Typing System	Static typing and templates enable compile-time code specialization, avoiding runtime polymorphism overhead.	Dynamic typing limits static optimizations, as type checks and memory allocation occur at runtime.	Static typing allows compiler optimizations like virtualization and function inlining.
Execution & Compilation	Mature compilers generate highly optimized machine code, leading to shorter active CPU time and lower energy use.	Code is compiled to bytecode and run on a VM. This interpreter overhead significantly impacts performance.	Compiles to efficient native machine code but needs a runtime. The compiler performs optimizations for performance.
Concurrency Model	Low-level primitives (<code>std::thread</code>) offer fine-grained control without a GIL but require manual management.	Threading is limited by the GIL for CPU-bound tasks. Multiprocessing works but has higher overhead.	Lightweight goroutines and channels allow for high concurrency with very low overhead, managed by the runtime.
Memory Management	Manual memory control (<code>new/delete</code> , smart pointers) and RAII provide deterministic cleanup, avoiding GC overhead.	Automatic GC on mostly heap-allocated objects increases memory footprint, GC load, and access latency.	Automatic GC with a focus on stack allocation for value types, which reduces GC pressure and improves data locality.
Standard Library	The Standard Template Library (STL) provides highly optimized, performance-focused data structures and algorithms.	Performance-critical modules are often C extensions, but the call overhead from Python remains.	Many standard library functions (e.g., networking, crypto) are highly optimized, some using assembly for critical paths.
Abstractions	Aims for "zero-cost abstractions," where high-level features are compiled away and incur no runtime overhead.	High-level abstractions and dynamic features are powerful but generally incur significant runtime overhead.	Interfaces provide abstraction with a small, well-defined runtime cost. Composition is favored over inheritance.

connections, while having an easy syntax for any programmer. Some companies that use this are Uber, Docker, Twitch, previously Discord [16] and, although not mainly, Netflix.

From Table 2.1, it can be seen that Go is statically typed and compiled, which makes it have a good start as an efficient programming language. But from the Table 2.2, we can observe go has a managed runtime, which means the energy consumption will be higher than other languages that do not have this. This runtime is the section of the program in charge of running and scheduling goroutines. This is why go binaries have a bigger minimum size as the runtime has to be fitted in the binary, which is great for Cross Compilation, but not great for either energy efficiency or performance.

Go's scheduler performs a series of steps before starting to run the user's code. As described in [17], [18] and [19], the runtime can be divided into these steps:

1. **OS Loading:** The main function is not the actual entry point of a Go program. Rather, the starting point is an assembly level function within the runtime. You can find it in a file corresponding to your specific OS and architecture, for example, `rt0_linux_amd64.s`. This is the first function the Go's program code will have the OS execute after loading the binary, and its only responsibility is to get the environment setup for the Go runtime.
2. **Argument and Environment setup:** The runtime, after being loaded into memory, calls an internal function `runtime.args` that handles the arguments and environment. This function copies the arguments (`argc` and `argv`) and environment variables into a Go-managed memory space. This ensures that the rest of the Go program, including the main function, can access this information through standard library functions like `os.Args` and `os.Getenv`.
3. **Scheduler Initialization (M:P:G Model):** The heart of the concurrency system in Go is the "M:G:P" model. Before any go code is executed, the scheduler must be initialized, which happens inside `runtime.schedinit`.
 - **M0 and G0 Creation:** The program starts with a single Operating System (OS) thread (*M0*). Every *M* thread has a special goroutine called *g0*, which is responsible for scheduling other runtime tasks.
 - **P Initializations:** A list of Ps or processors, which is a resource required to execute Go code, is created. The limit of P is determined by the `GOMAXPROCS` environment variable or inside the code by using the `runtime.GOMAXPROCS()` function.

At this time, the scheduler limits are put in place, limiting the maximum number of OS threads to 10,000.

4. **Memory Allocator and GC initialization:** Go's runtime includes a complex memory allocator and Garbage Collector [20] and [21].

- **Memory Reservation:** The runtime reserves a large region of virtual memory, divided into 3 areas: `spans`, `bitmap` and `arena` where go objects are allocated on the heap.
 - **Allocator Structures:** Other structures such as the `mheap` (the global heap structure for Go), `mcentral` (a central cache for memory spans) and for each P a per-thread cache for allocating small objects without locking the main thread, called `mcache`.
 - **GC Pacer:** The pacer determines the optimal time to trigger a Garbage Collection cycle based on the `GOGC` environment variable. The goal of Go's collection system is to perform one as the heap doubles in size since the previous cycle.
5. **Package Initialization:** At this point, the runtime can start reading from the supplied files. It starts with importing the required dependencies and initializing package-level variables. Once all files are processed in lexical file name order, the `init()` function of each file and then subsequent functions inside each file are called.
 6. **Creating the Main Goroutine:** The runtime doesn't call the `main.main` function directly. Instead, it creates a new goroutine to execute it. This is done using the internal `runtime.newproc` function. A new goroutine (G) is created, and its instruction pointer is set to the `main.main` function. This new goroutine is then placed into the local run queue of one of the available Ps, making it runnable.
 7. **Start the Scheduler:** Finally, the `runtime.mstart` is called on the main thread, that enters into the scheduling loop. From this point on, the Go program is running, and the scheduler is fully operational, managing the execution of all goroutines on the available threads.

2.3.2. Python: Interpreted Programming Language

Python is the most popular language according to the [TIOBE Index](#) as of May 2025 and has been since October 2021. Either because of its easy to start as a simple to start with programming language or because the actual trend of Artificial Intelligence (AI) is mostly programmed with Python, its popularity has skyrocketed.

Python, on the contrary to most of the other popular languages is an interpreted language, which means instead of having a compiler turn the code into assembly and then into binary, it has an interpreter that runs the bytecode instructions written by the programmer one by one.

As [22] puts it: "Garbage collection also can have a significant impact on both execution time and memory usage, and can be fine-tuned to obtain better performance"

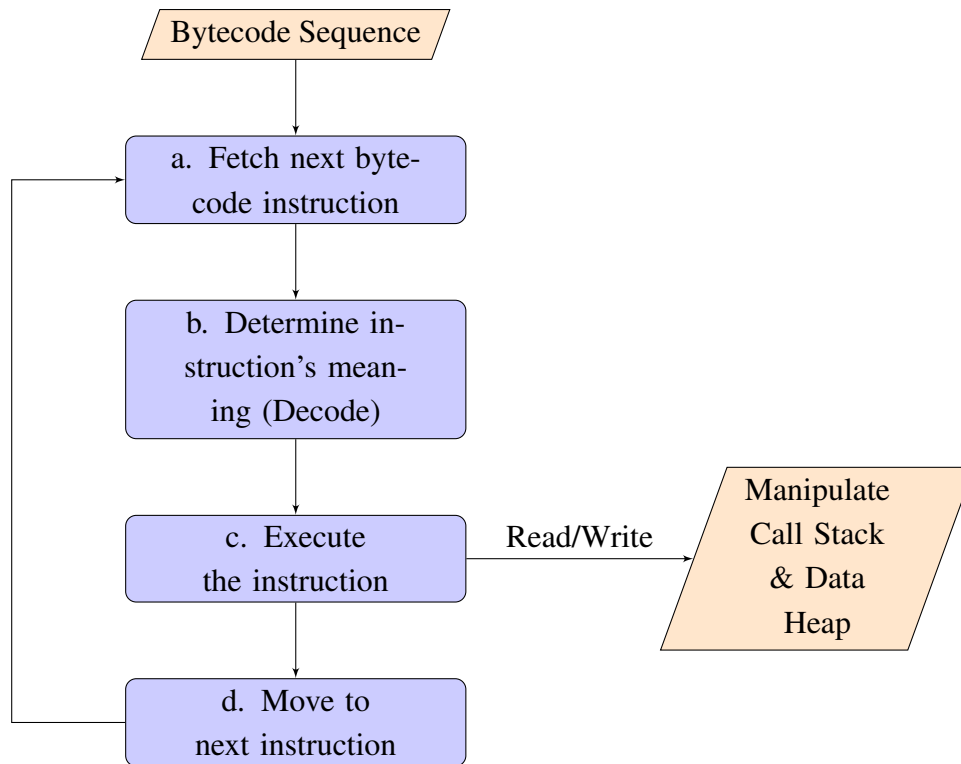


Fig. 2.3. Python's Execution loop

"Compiling" Python code to bytecode

There are multiple steps between the Python code that the user writes and its execution. While using CPython, this can be divided into two phases [23]:

1. Phase 1: The "Frontend"

- (a) **Reading the source code:** The python interpreter reads the `.py` files that were passed as an argument when invoked.
- (b) **Lexical Analysis (Lexing & Tokenizing):** In this step, the interpreter breaks down the code into a sequence of tokens, the smallest meaningful unit of the language's grammar.
- (c) **Parsing:** The stream of tokens from the previous step is fed into the parser, which checks if the sequence of tokens conforms to the rules that define Python's grammar [24].
- (d) **Compiling to bytecode:** The Abstract Syntax Tree (AST) is traversed generating bytecode. When the compilation is done, python caches the bytecode into a folder names `__pycache__` and stores `.pyc` files, representing the bytecode version of the same named file, so that in future executions, all the previous steps can be skipped.

2. Phase 2: The "Backend"

- (a) **Loading the bytecode** into the Python's Virtual Machine (PVM), either from the output of the compiler or from the `.pyc` file.
- (b) **Python's Execution loop**: As we can see from Figure 2.3, the loop is extremely simple. It starts by fetching the next instruction, decoding that instruction, executing it and moving the pointer to the next instruction.
- (c) **Stack Frame Management**: The PVM manages function execution by pushing a stack frame, containing the function's context like its variables and return address, onto the call stack upon invocation and popping it upon completion to resume the prior state.
- (d) **Memory Management**:
 - **Reference Counting**: This is the primary mechanism. It works by having all objects keep a count of how many variables or other objects refer to themselves. If this count drops to zero, the object is removed from memory and thus that section of the memory is freed.
 - **Cyclic Garbage Collector**: As there are some cases where the reference counting can not deal with cyclic references (e.g., when object α refers to β and β refers to α), a garbage collector process also has to run periodically. This means the efficiency of the interpreter is not very high as it need extra processes to clean up memory. This GC uses a generational approach, based on the idea that most objects are short-lived, and focuses its effort on newer objects.¹

A Concrete Example with `dis`

Listing 2.1. Python code demonstrating the `dis` module.

```
1 import dis
2
3 def simple_math(a):
4     x = a + 10
5     return x
6
7 # Use the disassembler to inspect the function's bytecode
8 dis.dis(simple_math)
```

Let's see this in action. The `dis` module is a 'disassembler' that shows you the bytecode for a piece of Python code. The script in Listing 2.1 defines a simple function and then uses `dis` to inspect it.

Listing 2.2. Bytecode output generated by the `dis.dis` function.

¹In some interpreters such as CPython, you can interact with the collector using the `gc` module. [25]

1	4	0 LOAD_FAST	0 (a)
2		2 LOAD_CONST	1 (10)
3		4 BINARY_ADD	
4		6 STORE_FAST	1 (x)
5			
6	5	8 LOAD_FAST	1 (x)
7		10 RETURN_VALUE	

The output of this script, shown in Listing 2.2, reveals the low-level bytecode instructions that the Python Virtual Machine will execute.

Other Interpreters

TABLE 2.3

ALTERNATIVE PYTHON IMPLEMENTATIONS

Implementation	Description
IronPython	Python running on .NET
MicroPython	Python running on microcontrollers and in the Web browser
Stackless Python	A branch of CPython supporting microthreads
Jython	Python running on the Java Virtual Machine

As python's interpreter is almost completely independent of its syntax and language development, there are multiple interpreters, each one with its features. One of the most popular alternatives to CPython is [PyPy](#), a fast implementation of Python with a JIT compiler. The problem with Just in Time compilers are that there might incur into potential warmup costs, before the functions go through the compiler. This process can optimize some hot code paths (a function or a section of the code that is run multiple times). Other examples are shown out in Table 2.3

2.3.3. C++: Directly Compiled, Unmanaged Language

Based on the programming language C, released in 1978 as a high-level language at the time, compared to assembly. C++ is one of the most famous language when it comes to high performance computing applications.

As we can see from Table 2.1, there are many characteristics on why it is one of the most used languages for high performance software, for example, [blender](#) a 3D creative software or [nuke](#) a VFX composition tool. This known examples and the multiple tests performed in multiple courses during the computer science degree have shown its capabilities.

If we take into account the energy efficiency, from Table 2.2 we can observe that being a compiled language, with multiple optimizations at the compilation level, zero-

cost abstractions, no runtime and direct memory management makes it one of the best low energy consumption language in theory. In this section, the different programming languages that have been chosen will be discussed, as well as why other similar languages were not.

Compilers

There are two main compilers for C++:

G++ is the C++ compiler for the GNU Compiler Collection (GCC). It is widely considered as a seasoned, reliable veteran; it's the default on most Linux distributions and has a long history of producing highly optimized code for final release builds. While its error messages have improved significantly over the years, they can sometimes be verbose and a bit cryptic, leaving you to decipher a long template expansion error.

Clang++ is the C++ compiler front-end for the LLVM project. It often feels like it was designed specifically to make a developer's life easier, excelling in two key areas: speed and diagnostics. Clang++ is famous for its remarkably fast compilation times and for error messages that are not only clear and color-coded but often suggest the exact fix, creating a much tighter and less frustrating coding loop. This focus on tooling is why it's also the engine behind many modern IDE features and static analysis tools.

2.3.4. Other languages not used

There are many more languages, but to reduce the scope of the project and have a good analysis on each of the languages to be analyzed, a reduced group had to be selected.

As a contender for a fast, high energy efficient language we could have used Rust, a recently new programming language, focused on performance and type-safety. As Rust is a compiled language and uses the same LLVM backend for compilation, a similar result is to be expected from this benchmark compared to the C++ implementation, thus, I excluded it from the study.

Other languages that could have been good contenders to be tested, not because of their efficiency but because of their widespread use could have been:

- **Java**
- **JavaScript**
- **Perl**
- **Fortran**

I would also like to mention that, I have not used GPU specific code, as that is not the goal of this project, as we can not really compare the energy efficiency of a CUDA

program with a C++ program, as the CUDA program is designed to run on a GPU, which has a completely different architecture and power consumption profile than a CPU.

TABLE 2.4
LANGUAGES EXCLUDED FROM THE STUDY AND JUSTIFICATION

Language	Reason for Exclusion
Java / C#	These languages primarily execute on managed runtimes (the JVM and .NET CLR, respectively). Their common Just-In-Time (JIT) compilation model is fundamentally different from the Ahead-Of-Time (AOT) native compilation of C++ and Go. Including them would be similar to go's implementation with a specific runtime .
JavaScript / TypeScript	These languages were designed for more web-centric environments, these languages run on JavaScript engines and typically use a single-threaded event loop for concurrency. This distinct execution paradigm and primary application domain fall outside the scope of this study, which focuses on general-purpose compiled languages. Some runtimes that JavaScript use are Node, Deno or bun, but all of them have to use a core, either V8 (for node and Deno) or JavaScriptCore (for bun).
Ada	While a statically typed and being Ahead-of-time compiled language, Ada is highly specialized for high-integrity, real-time, and safety-critical systems (e.g., avionics, defense). Its lower mainstream adoption and niche focus make it less representative for this study.
Zig	As a modern systems' language, Zig aligns well with the technical characteristics of C++ and Go. However, it is a relatively new language that has not yet achieved the same level of industrial adoption, ecosystem maturity, or long-term stability as the selected languages. The study's focus is on established, widely-used technologies to ensure the relevance of the findings to the current software development landscape.

2.4. Previous Benchmarks

Research in this area has intensified recently, driven by the growing global imperative to improve energy efficiency. This topic is no longer a niche concern but has garnered significant interest across industrial, economic, and policymaking sectors worldwide.

One of the first papers published that sparked the flame in the energy efficiency aspect of software development is Pereira et al. [26], that analyzed the performance of twenty-seven languages in ten different programming problems. The problem I found out with these kinds of tests is that the algorithms usually are quite simple and do not represent real-world applications.

As Kempen et al. [27] put it, "Despite the fact that these studies are statistical and only establish associations, they have nonetheless been broadly interpreted as establishing a causal relationship, that the choice of programming language has a direct effect on a system's energy consumption. This misinterpretation stems in part from the work's presentation, not only in ranking of languages by efficiency, but also from the specific claim that "it is almost always possible to choose the best language" when considering execution time and energy consumption [28]". Continuing with that idea, they also state that short benchmarks are not realistic of a real-life program being executed on a machine.

The most important aspect of comparing two languages when doing a benchmark, is making sure that the algorithm used to solve the task is the same between the multiple languages. Thus, the benchmarks that test the languages using libraries sometimes poison the results by adding another language without their knowledge. For instance, if the python program uses `numpy`, most of that library is implemented in C, or if we were to use `PyTorch`, that library is mostly implemented in C++-

Other studies such as [28], specifically on their Table 3, on the top languages, for `binary-trees`, `fannkuch-redux`, and `fasta`, are C, C++, Rust and fortran, exchanging positions depending on the test. As we see from this list, these are all compiled languages, which is not a surprise, as Abdulsalam et al. [29] comment on their paper comparing multiple compiler flags and other languages.

As Lion et al. [30] state in their publication "studies have found that V8 and CPython can be 8.01x and 29.50x slower on average than their C++ counterparts respectively" but they also state that, "choosing a language for your application simple because it is 'fast' is the ultimate form of premature optimization" [31]. They have also found out that with respect to a language with a runtime, it can provide a better performance and scalability. Although this might be counter-intuitive, Go's scheduler automatically maps user threads to kernel threads thus reducing the number of context switches.

Another challenge this area faces is measuring the energy consumption. Some CPUs have internal registers that can provide these measurements, like the intel Running Average Power Limit (RAPL), but other machines might have those registers not accessible to the user or, in case of most ARM processors, these registers do not exist. Another technique is using the measurement from an external power plug, like the TP-link HS110, used by S ndergaard et al. [32] in their measurements. The problem with these kinds of measurement devices is that the entire system is measured instead of only the CPU or the RAM and the precision it provides is much lower, being able to read only at a 1 Hz frequency.

CHAPTER 3

PROBLEM STATEMENT & ANALYSIS

This chapter contains an analysis of the functionalities provided by the renderer and the accompanying helper files. The objective of this analysis is to provide a clear understanding of the requirements and functionalities that the simulator must fulfill, as well as the constraints and non-functional requirements that must be considered during its development.

To achieve this, we will define the user requirements, functional and non-functional requirements, and restrictions that the simulator must adhere to. Then, we will present a use case diagram that illustrates the interactions between the user and the simulator, as well as traceability matrices that link the requirements to the use cases and functionalities of the simulator.

3.1. Project Description

This project aims to create a suite of programs that implement a ray-tracer engine in multiple programming languages, including C++, Python and Go. The goal is creating the pipeline to benchmark the performance and energy efficiency of each implementation, in multiple-core and single-core configurations.

Each implementation has to be able to run on macOS and Linux, and the energy consumption evaluation of each benchmark must work on each platform, as not all platforms support the same energy consumption evaluation tools.² This project is only implemented to run on CPUs as this projects' scope does not include GPUs.

²macOS uses `powermetrics` and Linux uses `perf` and a Raspberry Pi needs its power measured from the input source, as it does not have any internal counters

3.2. Requirements

The requirements for this project are divided into two main categories user requirements and program requirement. All of these requirements have been defined following the IEEE standard: [33].

To better organize the requirements, we will use the following abbreviations:

FN Functional

NF Non-Functional

To better describe these requirements, we will use the format described in Table 3.1.

TABLE 3.1
REQUIREMENT FORMAT TEMPLATE

RY-XX	
Name	Clear and concise title representing the functionality or constraint
Type	Functional / Non-Functional
Necessity	Optional / Essential / Desirable
Priority	Low / Medium / High
Complexity	Low / Medium / High
Description	Brief description of the requirement

- **Identifier:** code that identifies the requirement, following the schema *RY-XX*, where ‘Y’ is the type of requirement (‘F’ for Functional, ‘NF’ for Non-Functional) and ‘XX’ is a two-digit number that identifies the requirement.
- **Name:** clear and concise title, representing the functionality or constraint of the requirement.
- **Type:** indicates the type of the requirement, which can be functional or non-functional.
- **Necessity:**
 - *Optional:* the requirement is not essential for the system to function, but it would be nice to have.
 - *Essential:* the requirement is necessary for the system to function.
 - *Desirable:* the requirement is not essential, but it would be beneficial for the user.
- **Priority:** indicates the importance or urgency of the requirement, which can be low, medium or high.

- **Complexity:** establishes the level of effort required to implement the requirement, which can be low, medium or high.
- **Description:** a brief description of the requirement.

3.2.1. Functional requirements

TABLE 3.2
REQUIREMENT RF-01

RF-01	
Name	Cross-platform benchmark execution
Type	Functional
Necessity	Essential
Priority	Medium
Complexity	Medium
Description	The system must be able to run the language benchmarks on both macOS and Linux platforms.

TABLE 3.3
REQUIREMENT RF-02

RF-02	
Name	Output Verification
Type	Functional
Necessity	Essential
Priority	High
Complexity	Low
Description	The system must write the output of the programs.

TABLE 3.4
REQUIREMENT RF-03

RF-03	
Name	Core Restriction
Type	Functional
Necessity	Essential
Priority	High
Complexity	Low
Description	The system must be able to limit the number of cores.

TABLE 3.5
REQUIREMENT RF-04

RF-04	
Name	Per-Core Energy Consumption
Type	Functional
Necessity	Essential
Priority	High
Complexity	Medium
Description	The system must be able to show the energy consumption of the benchmarks.

TABLE 3.6
REQUIREMENT RF-05

RF-05	
Name	Custom Execution Parameters
Type	Functional
Necessity	Optional
Priority	Low
Complexity	Medium
Description	The system must be able to read the parameters from a file.

TABLE 3.7
REQUIREMENT RF-06

RF-06	
Name	Execution Time Inspection
Type	Functional
Necessity	Essential
Priority	High
Complexity	Low
Description	The system must record the time taken to run the different benchmarks.

TABLE 3.8
REQUIREMENT RF-07

RF-07	
Name	Image Visualization
Type	Functional
Necessity	Essential
Priority	High
Complexity	Low
Description	The system must output the resulting image of the program execution.

TABLE 3.9
REQUIREMENT RF-08

RF-08	
Name	Image Comparison
Type	Functional
Necessity	Essential
Priority	High
Complexity	Medium
Description	The system must output similar images with the same input parameters.

TABLE 3.10
REQUIREMENT RF-09

RF-09	
Name	Core Usage Notification
Type	Functional
Necessity	Desirable
Priority	Low
Complexity	Low
Description	The system must inform how many cores it is using.

TABLE 3.11
REQUIREMENT RF-10

RF-10	
Name	Remaining Lines Visualization
Type	Functional
Necessity	Optional
Priority	Low
Complexity	Low
Description	The system must show the remaining lines of the image to be rendered.

TABLE 3.12
REQUIREMENT RF-11

RF-11	
Name	Platform-Specific Time Measurement
Type	Functional
Necessity	Essential
Priority	Medium
Complexity	Medium
Description	The system must implement different time-measuring systems for the different platforms.

TABLE 3.13
REQUIREMENT RF-12

RF-12	
Name	Language-Agnostic Energy Measurement
Type	Functional
Necessity	Essential
Priority	High
Complexity	Medium
Description	The system must measure the energy consumption of each of the benchmarks regardless of the implementation language.

TABLE 3.14
REQUIREMENT RF-13

RF-13	
Name	Energy Result File Output
Type	Functional
Necessity	Optional
Priority	Low
Complexity	Low
Description	The system must output the file where the energy consumption result is kept at the end of the execution.

TABLE 3.15
REQUIREMENT RF-14

RF-14	
Name	Outlier Reduction via Multiple Executions
Type	Functional
Necessity	Essential
Priority	Medium
Complexity	Medium
Description	The system must reduce outlier results by executing the benchmarks multiple times.

TABLE 3.16
REQUIREMENT RF-15

RF-15	
Name	Centralized Benchmark Launcher
Type	Functional
Necessity	Essential
Priority	Medium
Complexity	Low
Description	The system must have a file from which all benchmarks can be launched.

TABLE 3.17
REQUIREMENT RF-16

RF-16	
Name	Individual Benchmark Execution
Type	Functional
Necessity	Desirable
Priority	Low
Complexity	Low
Description	The system must implement a mechanism to run each benchmark individually.

TABLE 3.18
REQUIREMENT RF-17

rf-17	
Name	Batch Benchmark Execution by Language
Type	Functional
Necessity	Desirable
Priority	Low
Complexity	Low
Description	The system must implement a mechanism to run all benchmarks for a specific language at once.

3.2.2. Non-Functional requirements

TABLE 3.19
REQUIREMENT RNF-01

RNF-01	
Name	C++ Implementation Required
Type	Non-Functional
Necessity	Essential
Priority	High
Complexity	Medium
Description	The system must include an implementation of the program in C++.

TABLE 3.20

REQUIREMENT RNF-02

RNF-02	
Name	C++ 17 Compiler Support
Type	Non-Functional
Necessity	Essential
Priority	High
Complexity	Low
Description	The user must be able to compile the code using a C++ 17 compliant or later compiler.

TABLE 3.21

REQUIREMENT RNF-03

RNF-03	
Name	Go Implementation Required
Type	Non-Functional
Necessity	Essential
Priority	High
Complexity	Medium
Description	The system must include an implementation of the program in Go.

TABLE 3.22

REQUIREMENT RNF-04

RNF-04	
Name	Go 1.20 Compiler Support
Type	Non-Functional
Necessity	Essential
Priority	High
Complexity	Low
Description	The user must be able to compile the code using a Go 1.20 compliant or later compiler.

TABLE 3.23

REQUIREMENT RNF-05

RNF-05	
Name	Python Implementation (PyPy Compatible)
Type	Non-Functional
Necessity	Essential
Priority	High
Complexity	Medium
Description	The system must include an implementation of the program in Python, with the ability to be executed using PyPy.

TABLE 3.24

REQUIREMENT RNF-06

RNF-06	
Name	Python 3.8 Support
Type	Non-Functional
Necessity	Essential
Priority	High
Complexity	Low
Description	The user must be able to compile the code using a Python 3.8 compliant or later interpreter.

TABLE 3.25

REQUIREMENT RNF-07

RNF-07	
Name	Unified Architecture Across Programs
Type	Non-Functional
Necessity	Optional
Priority	Low
Complexity	Medium
Description	The system shall implement the same architecture for every program to ensure consistency and maintainability.

3.3. Use Case

This section describes the use cases of the system. The use cases are represented in a diagram that shows the interactions between the user and the system. The ‘User’ is

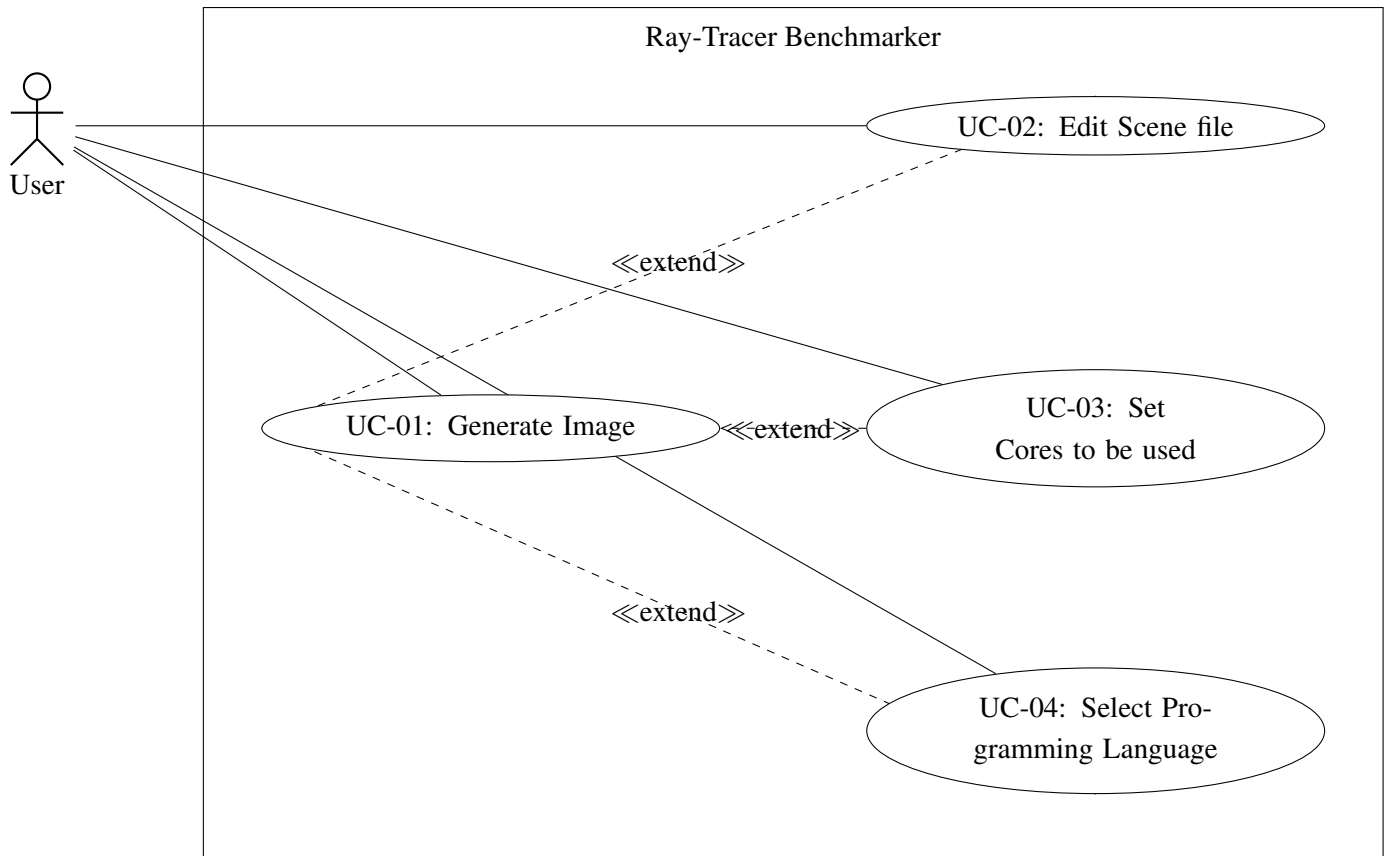


Fig. 3.1. Use-case diagram for the Ray-Tracer Benchmarker

considered to be a software engineer who will use the system to generate images using the ray-tracer engine to analyze the performance and energy efficiency of the different implementations.

Use case template:

TABLE 3.26
USE CASE UC-XX

ID: UC-XX	
Name	The name of the use case inside the diagram.
Actors	User, System
Objective	Brief description of the goal of the use case.
Description	Steps the actor has to entail.
Preconditions	The user has the system installed and configured.
Postconditions	The benchmarks are executed, and the results are stored.

- **ID:** code that identifies the use case, following the schema *UC-XX*, where ‘XX’ is a two-digit number that identifies the use case.
- **Name:** the name of the use case inside the diagram.

- **Actors:** entities that interact with the system.
- **Objective:** brief description of the goal of the use case.
- **Description:** steps the actor has to entail to achieve the objective of the use case.
- **Preconditions:** conditions that must be met before the use case can be executed.
- **Postconditions:** conditions that must be met after the use case is executed.

TABLE 3.27
USE CASE UC-01

ID: UC-01	
Name	Generate Image
Actors	User
Objective	Generate a rendered image using the ray-tracer engine with the selected configuration.
Description	<ol style="list-style-type: none"> 1. The user downloads the code repository. 2. The user sets the number of cores to use for rendering (CORES=<num>). 3. The user selects the platform (SERVER / macOS / RPI). 4. The user executes the command to generate the image. 5. The system processes the scene and outputs the rendered image file.
Preconditions	The code repository has been downloaded. The number of cores and platform have been set.
Postconditions	The rendered image is generated and stored in the output directory.

TABLE 3.29
USE CASE UC-03

ID: UC-03	
Name	Set Cores to be used
Actors	User
Objective	Set the number of cores to be used for rendering the image.
Description	<ol style="list-style-type: none"> 1. The user opens the configuration file. 2. The user sets the number of cores to be used for rendering (CORES=<num>). 3. The user saves the changes to the configuration file.
Preconditions	The configuration file is accessible and editable.
Postconditions	The configuration file is updated with the new number of cores.

TABLE 3.28
USE CASE UC-02

ID: UC-02	
Name	Edit Scene file
Actors	User
Objective	Edit the scene file to change the parameters of the scene to be rendered.
Description	<ol style="list-style-type: none"> 1. The user opens the scene file in a text editor. 2. The user modifies the parameters of the scene, such as camera position, light sources, and objects. 3. The user saves the changes to the scene file. 4. The user can repeat the process to make further changes to the scene file.
Preconditions	The scene file is accessible and editable.
Postconditions	The scene file is updated with the new parameters.

TABLE 3.30
USE CASE UC-04

ID: UC-04	
Name	Select Programming Language
Actors	User
Objective	Choose the programming language (Python, C++, or Go) for the ray-tracer execution.
Description	<ol style="list-style-type: none"> 1. The user reviews the available implementations. 2. The user selects the desired programming language (Python, C++, or Go). 3. The user ensures the required dependencies for the selected language are installed. 4. The user proceeds to execute the benchmark or rendering task using the chosen implementation.
Preconditions	All language implementations are available and properly installed.
Postconditions	The selected language implementation is used for subsequent operations.

3.4. Traceability

This section provides a traceability matrix that maps the requirements to the use cases. The matrix is designed to ensure that all requirements are addressed by the use cases defined in this document. Table 3.31 shows the functional requirements, while Table 3.32 shows the non-functional requirements, both with their corresponding use cases.

TABLE 3.31

TRACEABILITY MATRIX FOR FUNCTIONAL REQUIREMENT

Requirement	UC-01	UC-02	UC-03	UC-04
RF-01				
RF-02	✓			
RF-03			✓	
RF-04			✓	
RF-05	✓		✓	
RF-06	✓		✓	
RF-07	✓	✓		
RF-08	✓	✓		✓
RF-09			✓	
RF-10				
RF-11	✓			
RF-12				✓
RF-13	✓			
RF-14	✓			
RF-15	✓			
RF-16	✓			
RF-17	✓			✓

TABLE 3.32

TRACEABILITY MATRIX FOR FUNCTIONAL REQUIREMENT

Requirement	UC-01	UC-02	UC-03	UC-04
RNF-01	✓		✓	
RNF-02	✓		✓	
RNF-03	✓		✓	
RNF-04	✓		✓	
RNF-05	✓		✓	
RNF-06	✓		✓	
RNF-07	✓	✓		✓

3.5. System Architecture

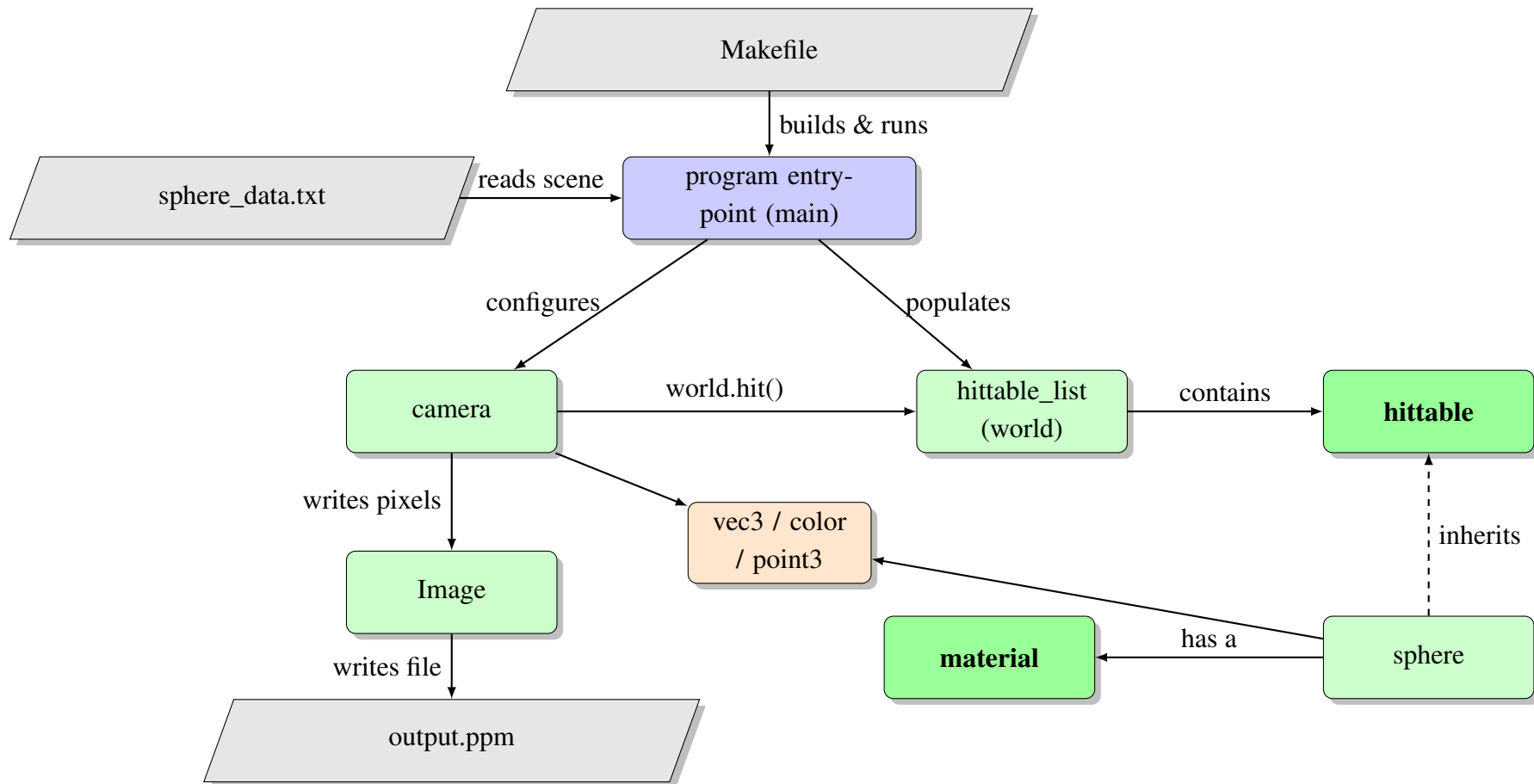


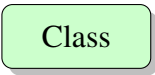
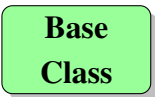
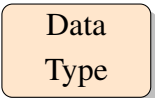




Fig. 3.2. System Architecture Diagram for the Ray Tracer.

The system architecture diagram uses the following visual conventions:

TABLE 3.33

DIAGRAM KEY FOR SYSTEM ARCHITECTURE

Symbol	Description
 Component	Program components and entry points
 File	Input/output files and build scripts
 Class	Implementation classes
 Base Class	Abstract base classes
 Data Type	Data structures and types
	Direct relationship or dependency
	Inheritance relationship

The principal components of the system, shown in Figure 3.2, are:

- **sphere_data.txt**: the file that contains the scene data, which is read by the program to generate the image. This is where the user can modify the scene parameters, adding more spheres, changing their positions, colors and materials.
- **Makefile**: the file that contains the build instructions for the program, which is used to compile and run the program. It also restricts the number of cores to be used for rendering and organizes the energy consumption and execution time measurements.
- **main**: the entry point of the program, which reads the sphere data, initializes the camera, the world and calls the rendering function.
- **camera**: the class that represents the camera used to render the scene, which is responsible for generating the rays and writing the pixels to the image.
- **hitable_list**: the class that represents the world (all the elements in the scene), which contains a list of hittable objects (spheres) and is responsible for checking if a ray hits any of the objects in the scene.

- **hittable**: the base class for all hittable objects in the scene, which defines the interface for checking if a ray hits an object.
- **sphere**: the specification of a hittable object of a sphere, which inherits from the hittable class and implements the specific behavior for spheres.
- **material**: the base class for all materials used in the scene, which defines the interface for calculating the color of a ray hit.
- **Image**: the class that represents the image to be generated, which is responsible for storing the pixels and writing the image to a file.
- **output.ppm**: the file that contains the generated image, which is written by the Image class after rendering the scene.

CHAPTER 4

DESIGN AND IMPLEMENTATION

The premise of this chapter is to describe the design, implementation and decisions taken during the development of the different programs that compose this project.

Section 4.1 describes the general design of the programs, and how they are divided. In section 4.3 and section 4.4, we will see how the objects are designed and how the rendering is done, respectively. Then, section 4.5 describes how the output is generated and how it can be used to visualize the results of the ray-tracer and finally, the last section, 4.6, describes how the program can be run and how to use the `Makefile` to run the program with different parameters.

4.1. General Program Design

The "30,000 foot view" of the programs is a ray-tracer, that has multiple spheres, with different materials, indexes of refraction and sizes. Each of the pixels is calculated individually, and then, when all pixels have been processed, they are outputted to a `ppm` file.

As it can be inferred from the Figure 4.1, the program is divided into two main sections:

- **Scene:** This section reads the input file and stores the spheres into their appropriate data-structures.
- **Renderer:** This section is the main loop of the program, processing each pixel and outputting the resulting image.

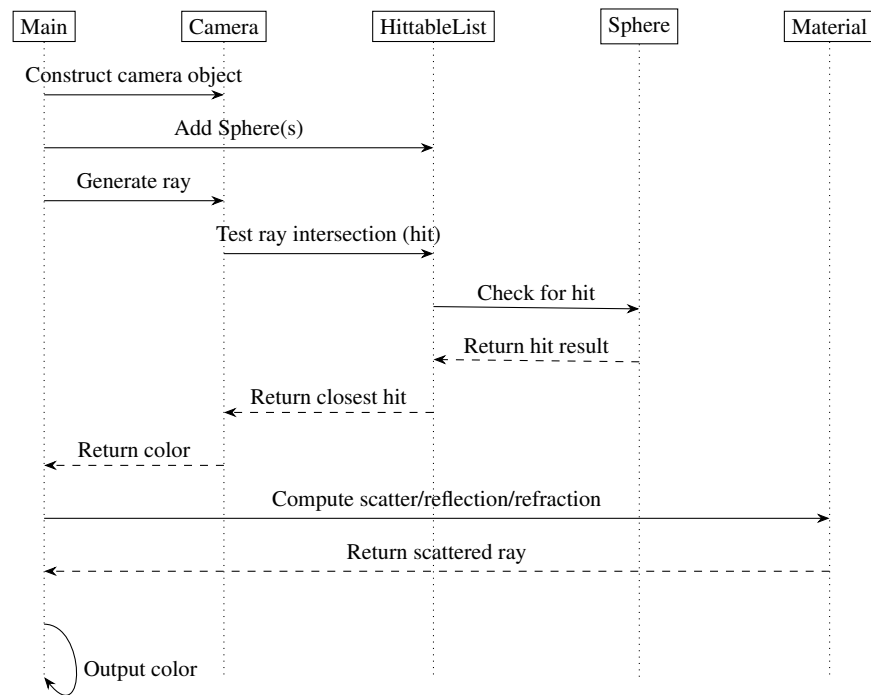


Fig. 4.1. General program data flow

C++ Implementation

```

.clang-format
.clang-tidy
CMakeLists.txt
src/
  camera.hpp
  color.hpp
  hittable_list.hpp
  hittable.hpp
  image.hpp
  interval.hpp
  main.cpp
  material.hpp
  ray.hpp
  rtweekend.hpp
  sphere.hpp
  vec3.hpp
  
```

Go Implementation

```

camera.go
color.go
go.mod
hittable_list.go
hittable.go
image.go
interval.go
main.go
material.go
ray-tracer
ray.go
rtweekend.go
sphere.go
vec3.go
  
```

Python Implementation

```

camera.py
color.py
hittable.py
image.py
main.py
material.py
ray.py
tree-py.txt
utils.py
vec3.py
  
```

Fig. 4.2. File structures for Ray-Tracer implementations in different languages

We can observe in Figure 4.2, there are the different files that compose each of the implementations of the ray-tracer, in C++, Go and Python. The file structure is similar in all three languages, as they must have an almost identical implementation.

4.2. Scene

This first section of the program reads the input file and stores the spheres into their appropriate data-structures, as we will see in section 4.3.

4.2.1. Sphere_data design

To ensure the consistency between programs and runs, I decided to create a file that would specify the layout of all spheres, and include the parameters for the camera setting, position and render settings:

- **ratio** **<width: double>** **<height: double>** \Rightarrow Aspect ratio of the output image (width / height).
- **width** **<int>** \Rightarrow The number of pixels for the width in the output image.
- **samplesPerPixel** **<int>** \Rightarrow How many times each of the pixels is processed. The higher this number is, the slower the render, but the less noise that the output image has. See Figure 4.4 and Figure 4.3 for examples of the same image with different `samplesPerPixel` values.
- **maxDepth** **<int>** \Rightarrow Specifies how many bounces a ray has to perform before getting the resulting color.
- **vfov** **<int>** \Rightarrow State the Field of Vision (FOV) of the camera.
- **lookFrom** **<x: double>** **<y: double>** **<z: double>** \Rightarrow Position of the camera in 3D space, where x is width, y is the height and z is the depth.
- **lookAt** **<x: double>** **<y: double>** **<z: double>** \Rightarrow States the relative "up" orientation of the camera.
- **vup** **<x: double>** **<y: double>** **<z: double>** \Rightarrow Vector that describes what is "up" in the scene
- **defocusAngle** **<double>** \Rightarrow This parameter represents the "aperture". A higher number will mean more objects will be in focus, and a smaller number results in a shallower dept of field.
- **focusDist** **<double>** \Rightarrow Specified the distance from camera look from point to a plane where the elements are in perfect focus

4.2.2. Language Specific

To try eliminating any possible influence of libraries created in other programming languages, all programs have been created only using their own standard library.

C++

When using C++, the intent was to use some of C++ modern features that would make development easier and adapted to new standards such as the use of smart pointers, `constexpr` and range-based loops. It was designed as an object-oriented program, with polymorphism through virtual functions (inside `material`, `hittable`).

The idea of making it header only was the benefits of inlining, easy to distribute and easily separable concepts and, as the compilation time is not crucial, the re-compilation of the headers every time there is a modification is not a drawback.

For multithreaded operations, I selected the openMP library due to its ease of use and excellent performance. openMP has been a widely adopted standard for decades, supported by most compilers, and originally emerged from the Fortran ecosystem in the 1960s.

Go

When choosing Go as to build the ray-tracer, as Go does not support inheritance like other oop languages, I had to use `interfaces`, which are the tools Go provide for polymorphism. Interfaces are a type that defines a set of method signatures. Thus, for any struct that has the signature methods described in an interface, it can be called as an object from that type.

Listing 4.1. Go interface example.

```
1  type Material interface {  
2      Scatter(rIn Ray, rec HitRecord) (bool, Color, Ray)  
3  }  
4  
5  type Hittable interface {  
6      Hit(r Ray, rayT Interval, rec *HitRecord) bool  
7  }
```

In my specific implementation, two instances of these keywords were used to denote all types of materials and all Hittable objects, that have to implement a scatter function and a hit function as described in Listing 4.1.

Python

Python is one of the most open languages, where there are many ways of developing the same program. There have been some problems with python's parameters in functions,

whether they are passed as parameters or as references. Unlike in C++ where you can add `&` to symbolize the passing the parameter by reference in the function signature, or using the `*` in Go, in Python, at first, it seems you can not specify this behavior. But if you research into the inner-workings of python functions and how they work, it seems that "Python passes function arguments by assigning to them" as [34] states at PyCon 2015:

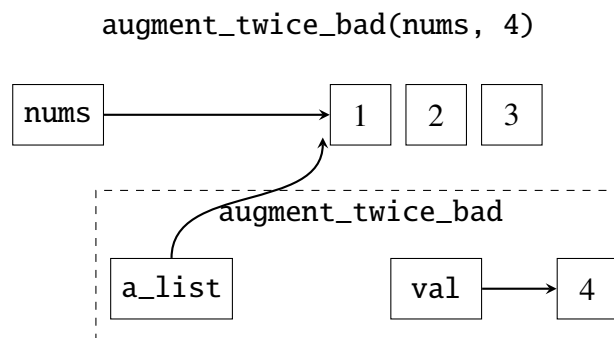
Listing 4.2. Python function assignment example.

```

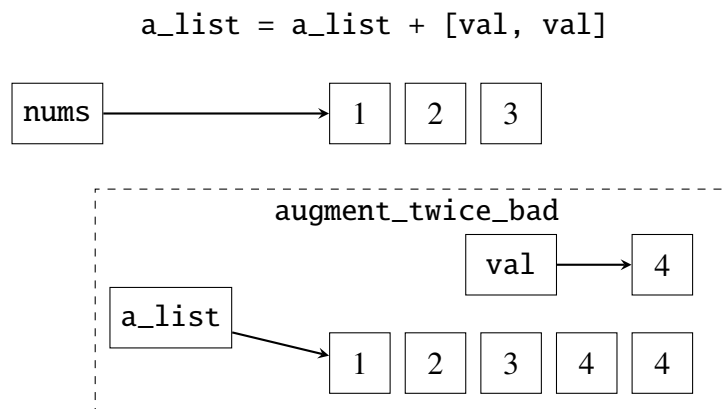
1  def augment_twice_bad(a_list, val):
2      """Put val on the end of 'a_list' twice."""
3      a_list = a_list + [val, val]
4
5  nums = [1, 2, 3]
6  augment_twice_bad(nums, 4)
7  print(nums)           # [1, 2, 3]

```

When calling the function `augment_twice_bad`, the parameters are assigned the values `nums` and `4` respectively.



The next statement is an assignment. The expression on the right-hand side makes a new list, which is then assigned to `a_list` thus any further modification is made to the local variable only:



When the function ends, its local names are destroyed, and any values no longer referenced are reclaimed, leaving us just where we started:

print(nums)



If the goal was to modify the list passed in, the `=` operator can not be used. Instead, a method that mutates the list in place should be used, such as `list.append()` or `list.extend()`. The correct implementation of the function would be:

Listing 4.3. Correct implementation of `augment_twice_good`

```

1 def augment_twice_good(a_list, val):
2     """Put val on the end of 'a_list' twice."""
3     a_list.extend([val, val]) # Mutates the list in place: [1, 2, 3,
4                               4, 4]
5
6 # Or, an in-place addition:
7 def augment_twice_good(a_list, val):
8     """Put val on the end of 'a_list' twice."""
9     a_list += [val, val] # Mutates the list in place: [1, 2, 3, 4,
10                        4]
  
```

4.3. Object

All objects in this program are spheres, even the "ground" is a sphere with a big enough radius that it seems a plane.

C++

Each of the spheres in C++ is a class called `sphere`, as all objects in this scene are spheres. `sphere` is a derived class from `hittable`, an abstract class, such that in the case that, in the future the program is modified to have more objects, it is easily implemented.

Listing 4.4. Sphere Class for C++

```

1 class sphere : public hittable {
2     public:
3         sphere(point3 const & center, double radius,
4               shared_ptr<material> mat)
5             : center(center), radius(std::fmax(0, radius)), mat(mat) { }
6
7         bool hit(ray const & r, interval ray_t,
8               hit_record & rec) const override { ... }
9
10        private:
11            point3 center;
12            double radius;
  
```

```

13     shared_ptr<material> mat;
14 };

```

Go

Each of the spheres in Go is a struct, one for each of the materials implemented (Lambertian, Metal, Dielectric). Each of these types of materials implement the Scatter method, described in Listing 4.1.

Listing 4.5. Go materials structs.

```

1  // Solid color
2  type Lambertian struct {
3      Albedo Color
4  }
5
6  // Fuzz: 0 for perfect mirror, higher for fuzzier reflection
7  type Metal struct {
8      Albedo Color
9      Fuzz    float64
10 }
11
12 // Transparent material such as water or ice
13 type Dielectric struct {
14     RefractionIndex float64
15 }

```

Python

All spheres in Python are different classes that inherit from the same Material class:

Listing 4.6. Python abstract class.

```

1 class Material(ABC):
2     @abstractmethod
3     def scatter(self, r_in: Ray, rec: 'HitRecord') -> tuple[bool,
4         Color, Ray]:
5         """Returns (scatter_happened, attenuation, scattered_ray)"""

```

4.4. Renderer

The main loop of the program is processing the object read from the file, added to the scene. This loop has two version in each of the programs designed:

- **Single-threaded loop:** The program only runs using one core. It has a double loop where it processes all the pixels in the image, one by one. This is an extremely CPU

intensive process, as there are many pixels and iterations to go through each of those pixels. After all pixels are processed, they are outputted into the `output_file`

- **multithreaded loop:** There are many ways a multithreaded renderer can work, even on different programming languages, different implementations have been chosen for specific reasons regarding their parallelization implementations. But in general, each of the pixels is processed, and then they are all joined into an array / list that is outputted to a file.

4.4.1. Multiprocessing

As previously stated, each of the programming languages, not only uses a different approach into how they have been parallelized, but even the algorithm had to be changed, as the implementation of python's interpreters makes the obvious parallelization perform surprisingly bad (this will be discussed in its section)

C++

To implement multiprocessing in C++, the openMP library has been used, as it allows implementing parallelism with a low-effort compared to the great results it provides.

Listing 4.7. OpenMP Pragma instruction.

```

1  #pragma omp parallel for schedule(dynamic, 1) default(none) \
2      shared(image, world, lines_remaining, cout_mutex, std::cout) \
3      firstprivate(samples_per_pixel, max_depth, image_width,
4          image_height)
5      for (int pixel = 0;
6          pixel < image_width * image_height;
7          pixel++) {
8          ...
          }

```

Dividing this `#pragma` directive into its components to better understand why each of the sections exist and its effects on parallelizing:

- **#pragma omp parallel for:** This construct merges a parallel region with a for-loop, enabling work-sharing. Specifically, a group of threads is created, and all the iterations of the for-loop are distributed among these threads.
- **schedule(dynamic, 1):** Uses dynamic scheduling, meaning each thread grabs one job at a time. Using 1 creates some more scheduling overhead, but it ensures fine-grained balancing.
- **default(none):** Disables all implicit data-sharing forcing the programmer to scope each variable used inside the parallel region. This helps at checking race conditions at compile time.

- **shared(image, world, lines_remaining, cout_mutex, std::cout):** The named variables refer to a single instance in shared memory, visible to all threads:
 - image: the pixel buffer, where all threads dump the processed pixel.
 - World: the scene description, with all the spheres.
 - Lines remaining: and atomic counter for progress reporting
 - cout_mutex + std::cout: Locking the cout_mutex before writing to std::cout to serialize console output.
- **firstprivate(samples_per_pixel, max_depth, image_width, image_height):** Each thread has its own copy of these variables, with the values copied from the master thread, they are constants, read-only, although you can modify them locally, but they do not copy to other threads.

Go

To parallelize in go, its standard library provides a system called goroutines. These are "Green threads" that are created by Go's runtime every time the keyword `go` comes before a function.

Listing 4.8. Goroutines.

```

1  var wg sync.WaitGroup
2  waitChan := make(chan struct{}, numThreads)
3  lines_remaining := c.imageHeight
4
5  for pixel_idx := range c.imageHeight * c.ImageWidth {
6      waitChan <- struct{}{}
7      wg.Add(1)
8      go func(pixel_idx int) {
9          defer wg.Done()
10         ...
11         <-waitChan
12     }(pixel_idx)
13 }
14 wg.Wait()

```

To be able to limit the number of threads that can be created, a channel with a size of `numThreads` is created and each time a new goroutine is going to be created, it tries to add a struct to the channel which, if there is an empty place, it adds the struct and allows the program to continue. But, if the channel is full, the program stops and does not allow any continuation of the program until the channel has a free spot, which is generated after the pixel is added to the resulting image.

To prevent the program from continue running before all the goroutines are finished, a Wait Group (`wg`) is used to prevent the main thread from continuing the main execution until all threads have finished (which is the same as the `wg` being empty).

Python

To parallelize in python, I had to use the `concurrent.futures` library to properly parallelize the python execution.

This library can create two types of "executors" which are:

- **ThreadPoolExecutor**: for I/O-bound tasks (uses threads)
- **ProcessPoolExecutor**: for CPU-bound tasks (uses processes)

To submit a task, you can use the following semantics, using python's list comprehension to create all the jobs:

Listing 4.9. Python submitting jobs ProcessPoolExecutor.

```
1 futures = [  
2     executor.submit(self.process_row, j, world)  
3     for j in range(self.image_height)  
4 ]
```

This creates a job for every pixel, running the function `self.process_row` inside the camera object, with parameters `j` and `world`.

To retrieve the results, you can simply iterate the futures object, as if it was a list of objects:

Listing 4.10. Python retrieving data from Process execution pool.

```
1 for future in futures:  
2     j, row_pixels = future.result()  
3     processed_rows += 1  
4     ...  
5     for i in range(self.image_width):  
6         img.set_pixel(i, j, row_pixels[i])
```

There is an interesting aspect on why the difference between the loops in Python and the rest of the programming languages: Compiled languages iterate over every pixel, while python iterates over every line, why is this? Because a copy of the entire python environment has to be created and the overhead of copying so much information slows down the program extremely. We will see the impact of these change in the evaluation section of the report.

4.5. Output

The output of the program was initially thought to be though the standard output of the terminal, and redirecting the output to a PPM file, but for measuring performance concerns

and stability between different programs and their implementation of interacting with the operating system, it was decided that the program would create a file and output all the data into that file.

The design of the output is a PPM file, in which the first three lines define the content, aspect and maximum values of a file.

TABLE 4.1
PPM FILE HEADER FIELDS

Field	Description
P3	Magic number (P3 = ASCII, P6 = binary)
400 225	Width and height (in pixels)
255	Maximum color value

In my specific case, I used P3, as I am outputting the Red Green Blue (RGB) values individually and a 255 maximum color value, to simplify the outputting of the resulting image.

This is an example image, of an image that this program would generate at maximum reasonable quality settings, 1920 width, 300 samples per pixel and 200 max depth.

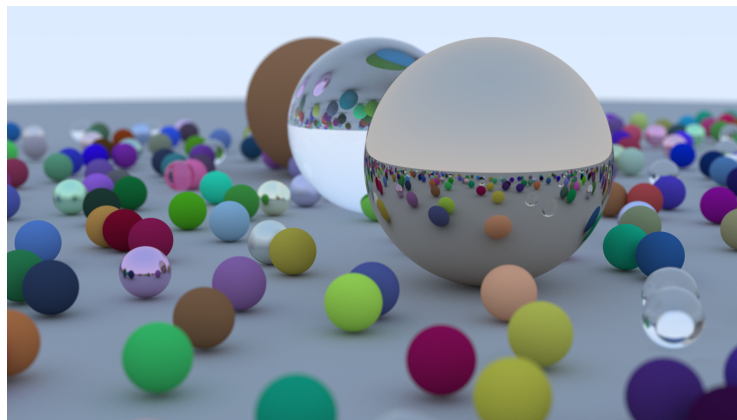


Fig. 4.3. Example of program output, 1920 width, 300 samples per pixel and 200 max depth

And this is an example of the same picture, but with the settings used to test the programs on the personal laptop and server, which takes 168x more time than Figure 4.3

4.6. Running the program

To run the program, I have designed a Makefile that allows to run the program with different parameters, such as the number of cores to be used and where the user can specify the platform the benchmark is being run on. This is specially needed for macOS, as it needs sudo privileges.

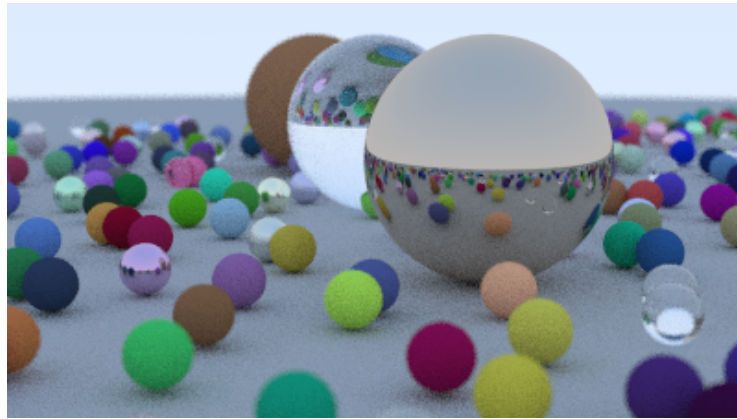


Fig. 4.4. Example of program output, 400 width, 50 samples per pixel and 50 max depth

Listing 4.11. Makefile for running the program.

```

1  # ===== Configuration =====
2  CORES                ?= 14
3  MAC_OS               ?= False
4  SERVER               ?= False
5  RESULTS_DIR          := $(CURDIR)/results-$(CORES)
6  SPHERE_DATA          := sphere_data.txt
7  POWERMETRICS_PID_FILE := $(RESULTS_DIR)/power/powermetrics.pid
8  POWER_LOG            := $(RESULTS_DIR)/power/powermetrics
9  POWER_TRIMM_LOG      := $(RESULTS_DIR)/power/powermetrics_trimmed
10 POWER_CLEANED_LOG     := $(RESULTS_DIR)/power/powermetrics_cleaned
11 POWER_INTERVAL       := 100 # Interval for powermetrics
12 PERF_COMMAND := perf stat -r 5 -e 'power/energy-pkg/,power/energy-ram/'
13
14 # ===== Directory Setup =====
15 $(RESULTS_DIR):
16     @mkdir -p $(RESULTS_DIR)
17
18 # ===== Power Management Functions =====
19 # Args: $(1)=output_name
20 define start_powermetrics
21     ...
22 endef
23
24 # Args: $(1)=output_name
25 define stop_powermetrics
26     ...
27 endef
28
29 # ===== Ray Tracer Execution Functions =====
30 # Generic function to run a ray tracer with timing
31 # Args: $(1)=directory, $(2)=description, $(3)=command, $(4)=output_name
32 define run_raytracer
33     ...

```

```
34  endif
35
36  # Single-threaded version
37  # Args: $(1)=directory, $(2)=description, $(3)=command, $(4)=
      output_name
38  define run_raytracer_single
39      ...
40  endif
41
42  # ===== Build Functions =====
43  define build_cpp
44      @echo "Building C++ ray tracer ..."
45      ...
46  endif
47
48  define build_go
49      @echo "Building Go ray tracer ..."
50      ...
51  endif
52
53  # ===== Language-Specific Targets =====
54  # Python Implementations
55  .PHONY: python python-single pypy pypy-single
56  python: $(RESULTS_DIR)
57      ...
58
59  python-single: $(RESULTS_DIR)
60      ...
61
62  pypy: $(RESULTS_DIR)
63      ...
64
65  pypy-single: $(RESULTS_DIR)
66      ...
67
68  # C++ Implementations
69  .PHONY: cpp cpp-single cpp-build
70  cpp-build:
71      ...
72
73  cpp: cpp-build $(RESULTS_DIR)
74      ...
75
76  cpp-single: cpp-build $(RESULTS_DIR)
77      ...
78
79  # Go Implementations
80  .PHONY: go go-single go-build
81  go-build:
82      ...
83
```

```
84 go: go-build $(RESULTS_DIR)
85     ...
86
87 go-single: go-build $(RESULTS_DIR)
88     ...
89
90 # ===== Batch Operations =====
91 .PHONY: all all-multi all-single benchmark
92
93 all-multi: cpp go pypy python $(RESULTS_DIR)
94     ...
95
96 all-single: cpp-single go-single pypy-single python-single $(
97     RESULTS_DIR)
98     ...
99
100 all: all-multi all-single
101     ...
102
103 benchmark: all
104     ...
105
106 # ===== Utility Targets =====
107 .PHONY: ppm-diff clean-power
108
109 ppm-diff:
110     ...
111
112 clean-power:
113     ...
114
115 # ===== Cleanup Targets =====
116 .PHONY: clean clean-builds clean-all stop-power
117 stop-power:
118     ...
119
120 clean:
121     ...
122
123 clean-builds:
124     ...
125
126 clean-all: clean clean-builds
127     ...
128
129 # ===== Help and Information =====
130 .PHONY: help info
131 help:
132     ...
133
134 info:
```

134

...

When running the macOS version, the output has to be cleaned up, as the `powermetrics` command outputs a lot of information that is not needed for the report. This is done by running the `utilities/clean_macos.sh` script inside the `results-<cores>` folder. Then, to get the power consumption, the `utilities/macos_to_jules.py` script has to be executed like this:

Listing 4.12. Running the power consumption script.

```
1 python3 macos_to_jules.py \  
2     '<output-folder>/powermetrics_cleaned_<lang>.log'
```

and the result would be printed to the terminal like this:

Listing 4.13. Power consumption output.

```
1 Power Statistics:  
2 Average: 31707.79 mW  
3 Maximum: 42631.00 mW  
4 Minimum: 2342.00 mW  
5  
6 Energy Results:  
7 Total energy: 50.605626 J  
8 Total time: 1.60 s  
9 Average power: 31707.79 mW
```

CHAPTER 5

EVALUATION

This section reflects the results of all the testing performed with the different programming languages and architectures / types of computer.

TABLE 5.1

PROGRAMMING LANGUAGES VERSIONS AND OPERATING SYSTEMS USED.

Operating System (OS)		C++	Go	Python	PyPy
Server	Ubuntu 24.04.2	GCC 14.2.0	1.24.2	3.12.3	7.3.19
Laptop	macOS 15.5	Apple Clang 17.0.0	1.24.2	3.13.5	7.3.19
Raspberry	Debian 12	GCC 14.2.0	1.24.4	3.13.5	7.3.19
Desktop	Ubuntu 24.04.2	GCC 13.3.0	1.24.5	3.13.5	7.3.19

5.1. Measurement Platforms

The benchmark is run 5 times to obtain a good average. Appendix G shows the percentage change between different runs for different numbers of iterations.

5.1.1. Many-Core Platform

This platform represents the most powerful as well as power-hungry combination of all devices in my test suite. This is a rack server with two Intel Xeon Gold 6326 processors, each having 16 cores and 32 threads, contributing to a total of 32 cores and 64 threads. It also has the largest amount of RAM from this testing, with 256GB of DDR4 memory.

As it has two sockets (one per CPU chip), there must be intercommunication between these processors if a process spreads out to more than 32 threads, or is set by the user

using the command `taskset`, which fixes the cores the process can run on.

Cache & Numa

Regarding the cache of these processors, as it can be seen from Listing 5.1, the cache at level 1 has 32 instances of 1.5 MiB of data cache and 1 MiB of instruction cache, half the total number of virtual cores, which is 64. There is also 48MB of L3 cache for each of the two processors.

Listing 5.1. Cache of the Intel Xeon Gold 6326

```

1  $ lscpu | grep -i cache
2  Caches (sum of all):
3      L1d:                1.5 MiB (32 instances)
4      L1i:                1 MiB (32 instances)
5      L2:                 40 MiB (32 instances)
6      L3:                 48 MiB (2 instances)
7  NUMA:
8      NUMA node(s):       2
9      NUMA node0 CPU(s):  0-15,32-47
10     NUMA node1 CPU(s):  16-31,48-63

```

Core Configurations

This system was the most versatile in terms of the number of tests that could be performed, as it has many processors and uses Linux on x86, providing a great advantage for forcing processes to run on specific cores.

The tests were conducted on a variety of core configurations, always setting cores in the same processor for core numbers less than 16.

- **1 Core:** Testing with one core, producing the baseline for the program's energy consumption and execution time.
- **2 Cores:** Testing with 2 cores provides the first glimpse of parallelization benefits.
- **4 Cores:** Testing with 4 cores because many computers from some time ago had four cores.
- **8 Cores:** Testing with 8 cores gives us great insight into how many processors in the market work, and it is half the amount of cores inside one chip.
- **14 Cores:** Testing with 14 cores, because it is the number of cores available on the laptop and we wanted to have an execution time comparison.
- **16 Cores:** Testing with 16 cores as it is the amount of real cores on a single chip. This should be one of the most energy-efficient and fastest tests, if there were only one CPU.

- **28 Cores (different CPUs, all real cores, no logical cores):** Testing with 28 cores distributed across two sockets is interesting because there has to be information sharing over the bus inside the motherboard to synchronize both CPUs. This won't be as energy efficient, but may be faster.
- **28 Cores (same CPU, 16 cores, 32 virtual cores):** Testing with 28 cores inside the same CPU; the performance should be slower as there are fewer real cores to tackle the work, but it has the advantage of not needing to share data with another socket.
- **32 Cores (same socket):** Testing with 32 cores in the same socket uses all available logical threads of a system: the 16 real cores and the other 16 threads the CPU has thanks to Hyper-Threading.
- **32 Cores (only real cores):** Testing with 32 real cores across two sockets should be the most powerful combination for CPU-intensive tasks, as all operations should be able to be carried out without many interruptions.
- **48 Cores:** Testing with 48 cores forces us to use all real cores and some logical cores.
- **60 Cores:** Testing with 60 cores is also interesting (not 64), as this would force the machine to interrupt the program we are benchmarking to perform routine operations, such as checking for incoming connections or logging.

Results

The results for the server are shown in the following figures and tables. The energy consumption is measured in joules, and the execution time is measured in seconds.

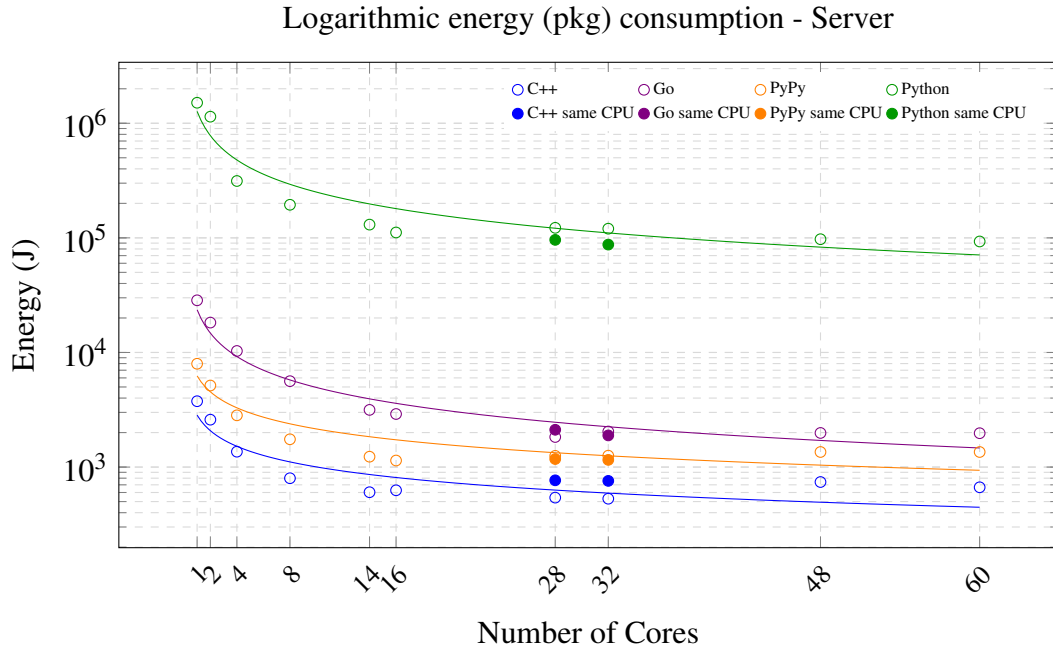


Fig. 5.1. Logarithm energy (pkg) consumption of the Server benchmark across different programming languages. source: Table A.1

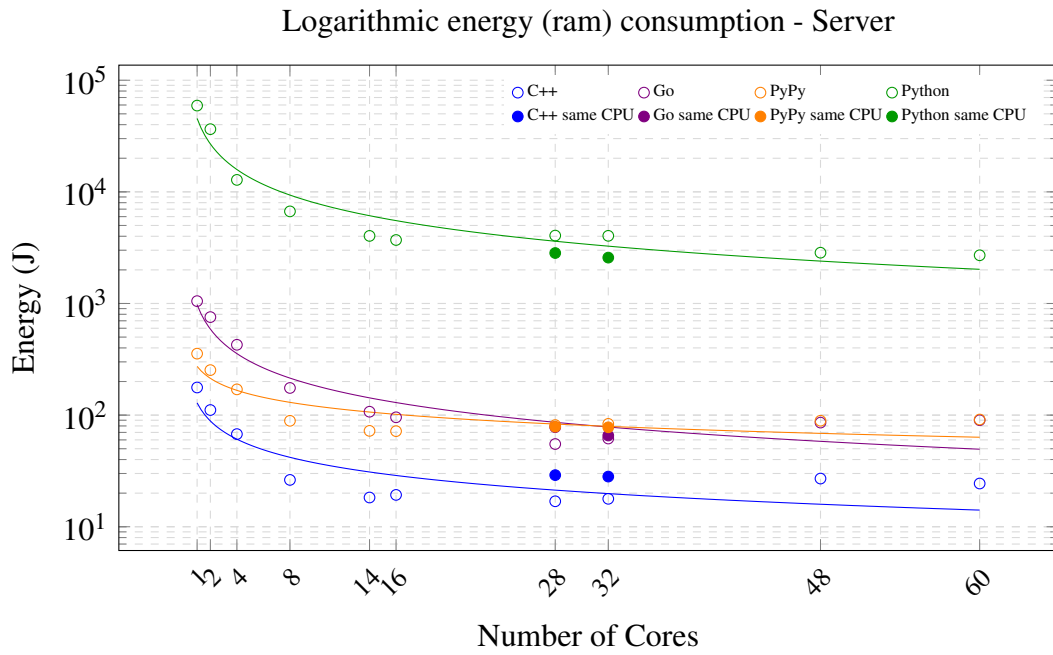


Fig. 5.2. Logarithm energy (ram) consumption of the Server benchmark across different programming languages. source: Table A.2

From Figure 5.1, we can see that the energy consumption of the server is not linear with the number of cores. It can be observed that the energy consumption decreases as the number of cores increases, but there is a point in the graph and Table A.1 where the energy consumption starts to increase slightly again, as well as the execution times in Figure 5.3, but not as much as the energy consumption.

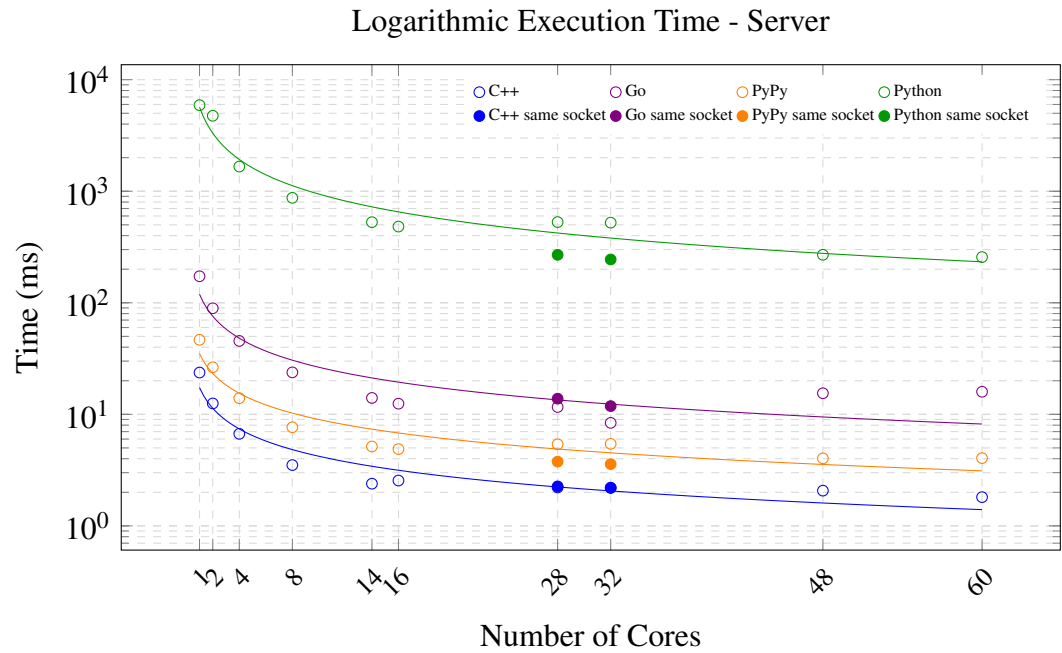


Fig. 5.3. Logarithm Execution time of the Server benchmark across different programming languages. source: Table A.3

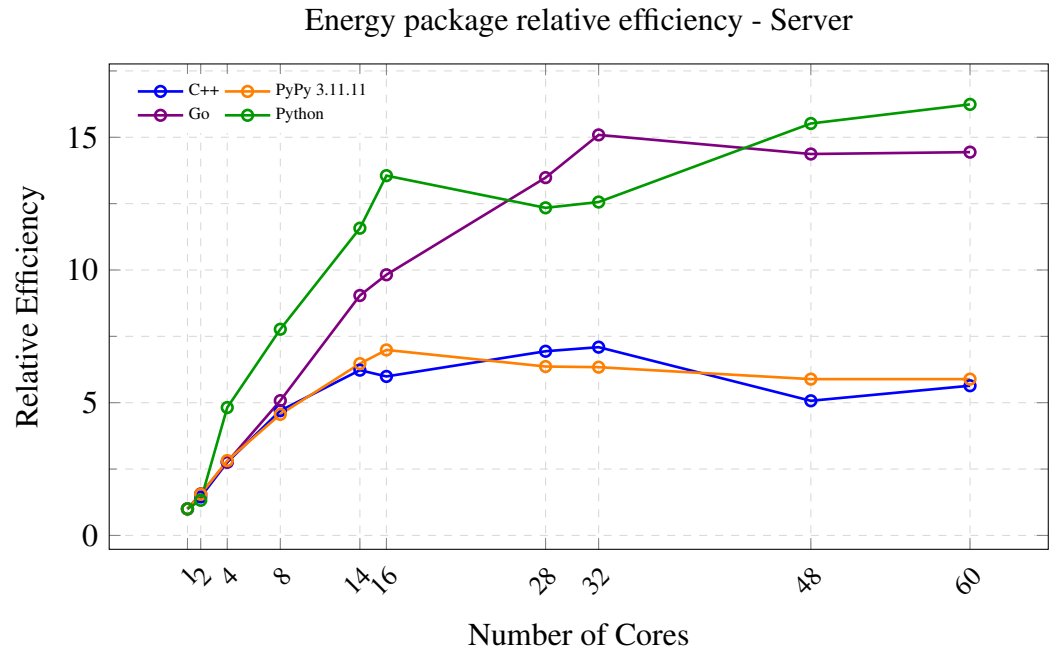


Fig. 5.4. Energy package relative efficiency of the Server benchmark across different programming languages (higher is better).

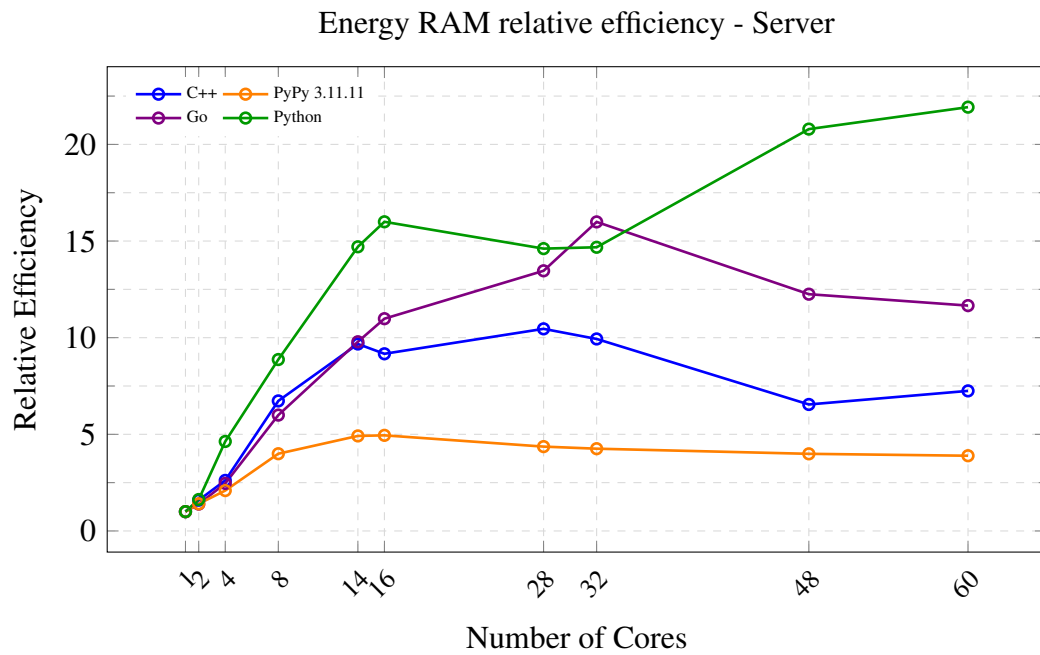


Fig. 5.5. Energy RAM relative efficiency of the Server benchmark across different programming languages (higher is better).

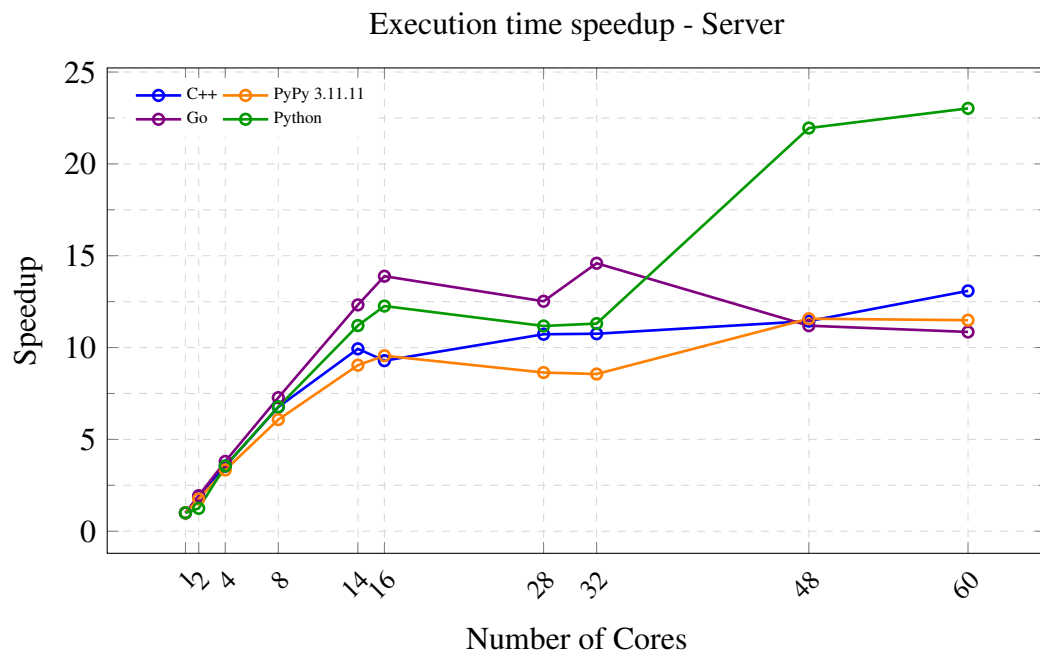


Fig. 5.6. Energy efficiency speedup of the Server benchmark across different programming languages (higher is better).

This is due to hyperthreading³ in the CPUs, which allows the CPUs to run two threads per core, but this is not as efficient as running a single thread per core, as the CPUs have to share resources between the two threads.

It is obvious from the multiple graphs and tables that the C++ implementation is the most energy-efficient and fastest by a significant margin, followed surprisingly by the PyPy execution of the Python code, which is faster than the Go implementation, and the Python implementation is the slowest and most energy-consuming by an extremely large amount.

However, when looking closely at the 28-core and 32-core tests, focusing on C++, we can see the energy consumption is lower when using cores from different CPUs rather than consuming more, as there is some energy efficiency loss when synchronizing the data between the two CPUs. What happens in this case is that the C++ parallelization algorithm makes each of the cores have a very hard CPU workload, resulting in a more efficient result. This aligns with subsection 2.2.4, where it is explained that hyperthreading is not as efficient for specific tasks.

I want to specifically discuss the 60-core test, as it is the most interesting one. In this test, the energy consumption is lower than in the 48-core test, as well as the execution time on the C++ implementation, but on the Go implementation, both energy consumption and execution time are higher than in the 48-core test. This is because the Go implementation is not as efficient as the C++ implementation, and the Go runtime has to manage more goroutines, which adds overhead.

Considering the 32-core and 48-core tests with the Python program, the energy consumption reduces significantly when the program starts using virtual cores, as the program is able to run on more cores, and the Python runtime is not very demanding, being able to use these cores efficiently. As shown in Figure 5.1 and Figure 5.3, this is an advantage to Python with respect to itself.

A usually not looked aspect is the energy consumed by the RAM in the system, which is shown in Figure 5.2. As we can see, the energy consumed by the RAM is not linear with the number of cores, and after incrementing the cores to more than 8 cores, the energy consumption does not decrease substantially. But we can see that when using the cores in the same processor, (28 same CPU & 32 same CPU), the RAM energy consumption is much higher than their counterpart using different processors when using **C++** or **Go**. But, at those same core counts, if we check the **Python** and **PyPy** implementations, the energy consumption is lower when using the same processor, as the Python runtime is not very demanding and does not require much memory bandwidth, thus being able to use the memory more efficiently.

It also must be noted that the cores during the 48 core benchmark were being used at

³Hyperthreading is enabled in this system. To set the process to a fixed CPU, I used `taskset -c [cores]`, i.e., `taskset -c 0-15,32-47` for running across multiple CPUs and `taskset -c 0-31` to force the program to only run in a single CPU.

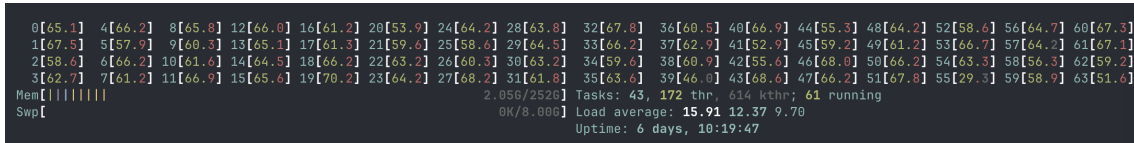


Fig. 5.7. htop showing the cores not being used at 100% when using many cores for processing in a per-pixel multi threading renderer

100% of their capacity, while in the 60 cores test, the cores were mostly being used at a lower percentage, as shown in Figure 5.7. This is because the Go runtime is not able to efficiently use all the cores when there are more than 48 cores available, and it is not able to schedule the goroutines efficiently as these routines finish so fast that the Go runtime is not able to keep all the cores busy.

If we changed the implementation to a per-row renderer, on the go-side, the Go runtime would be able to use all the cores more efficiently, as it would be able to schedule the goroutines more efficiently, and the execution time would be lower, but the energy consumption would be higher, as the cores would be used at 100% of their capacity. Thus, in this case, as we will see in other sections, having a faster execution time is not always the best option in terms of energy consumption.

All the raw data for the benchmarks can be found in the Appendix A.

5.1.2. Desktop

This platform is one of the most common in the amateur market, as it is a personal desktop computer with a Ryzen 3800x processor, which has 8 cores and 16 threads and has 32GB of RAM. It has a Zen 2 architecture, which has heterogeneous cores. These tests were performed with hyperthreading enabled, as it is the most popular configuration, and the default configuration.

Cache & Numa

The cache of the AMD Ryzen 3800x is shown in Listing 5.2. As we can see, it has 8 instances of 256 KiB of L1 data cache and 256 KiB of L1 instruction cache, which is half the number of threads available in the system (the same as the server platform). It has also two L3 caches of 32 MiB each, which is due to the processor having two CCXs.

Listing 5.2. Cache of the AMD Ryzen 3800x

```

1  $ lscpu | grep -i cache
2  Caches (sum of all):
3      L1d:                256 KiB (8 instances)
4      L1i:                256 KiB (8 instances)
5      L2:                  4 MiB (8 instances)
6      L3:                  32 MiB (2 instances)
7  NUMA:

```

8	NUMA node(s) :	1
9	NUMA node0 CPU(s) :	0-7

Results

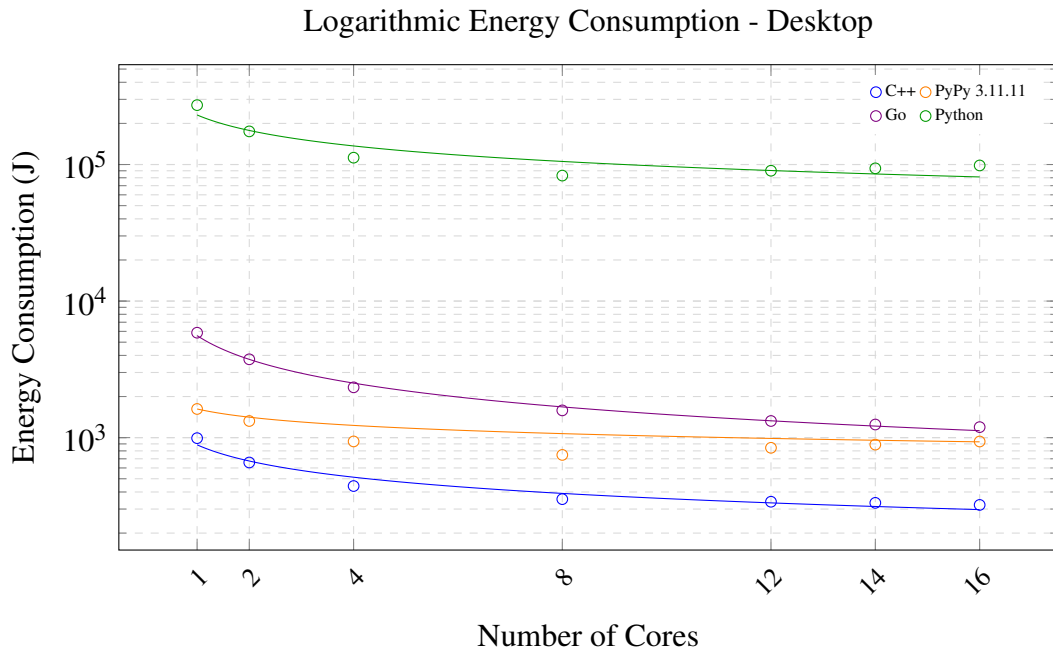


Fig. 5.8. Logarithm energy consumption of the Desktop benchmark across different programming languages. source: Table B.1

These results show a clear trend in the energy consumption and execution time of the different programming languages. The C++ implementation is the most energy-efficient, as expected, but again, PyPy surprises with its results, being the second most energy-efficient and being much closer to the C++ implementation than the Go implementation, which is the third most energy-efficient. Specific numbers for energy consumption and execution time can be found in Table B.1 and Table B.3.

We can also observe from Figure 5.9 and Figure 5.13 that the difference between the 8-core and 16-core tests is less pronounced compared to the server. This is because the processor has only 8 physical cores, while the remaining 8 are hyperthreaded cores, which are less powerful than the physical ones.

This can also be seen very clearly in the energy consumption of the PyPy and Python tests in Figure 5.9 (Table B.1) that when using more than 8 cores makes the energy consumption as the number of threads use separates from 8 (the number of real cores). This also affects the execution time but not as significantly, as we can see in Figure 5.13 (Table B.3), where the execution time does not change much when using more than 8 cores, but the energy consumption does.

All the raw data for the benchmarks can be found in the Appendix B.

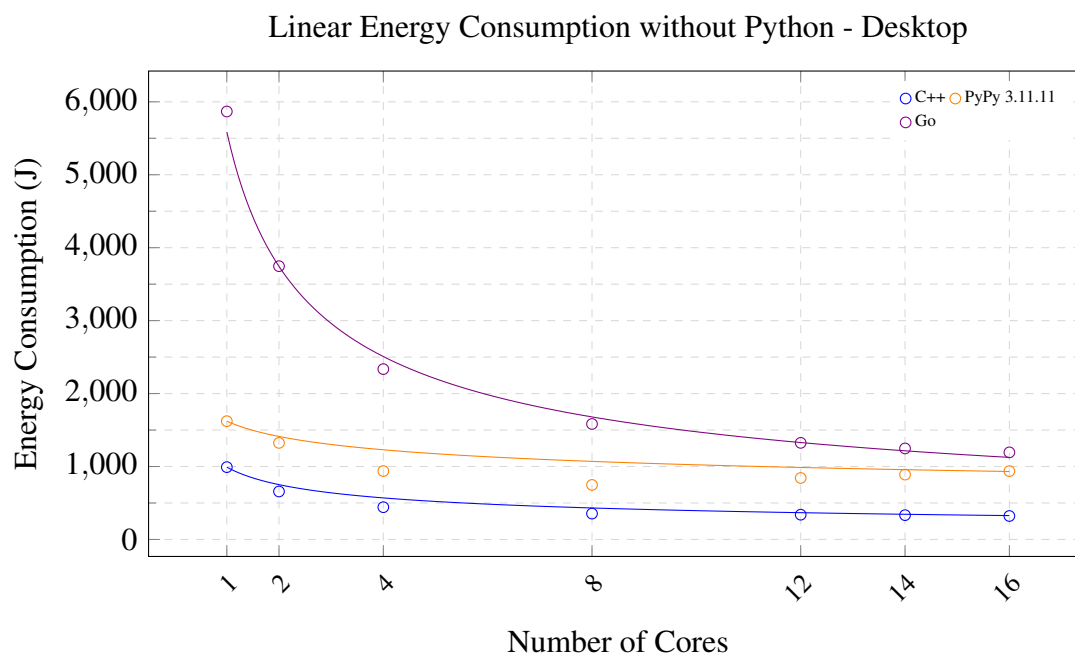


Fig. 5.9. Linear energy consumption of the Desktop benchmark across different programming languages. source: Table B.1

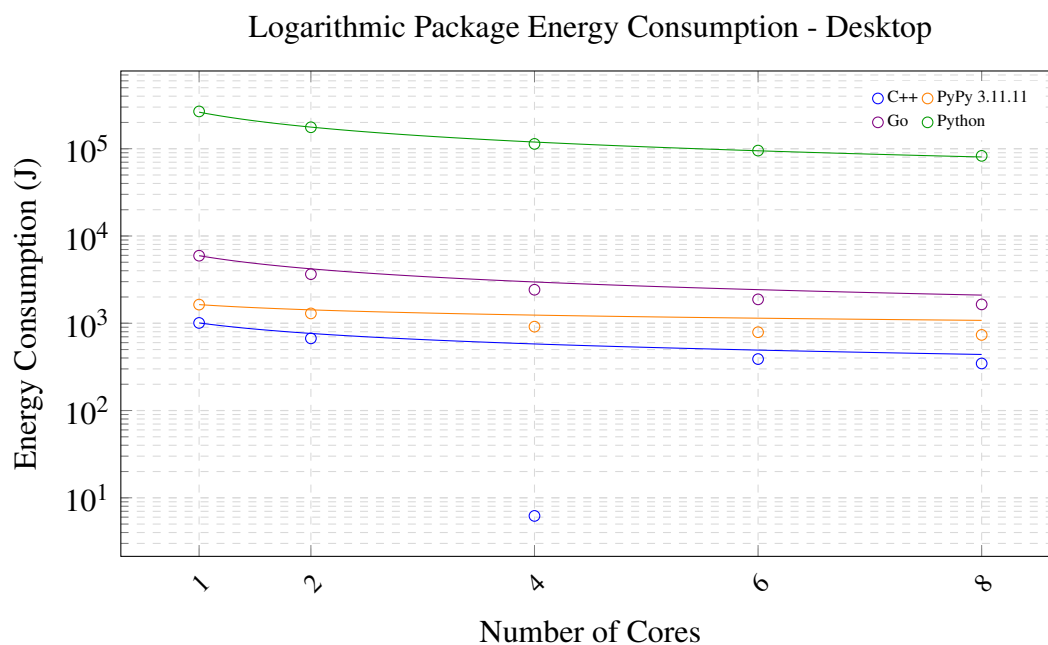


Fig. 5.10. Logarithmic package energy consumption of the Desktop benchmark without Hyper-threading across different programming languages. source: Table B.2

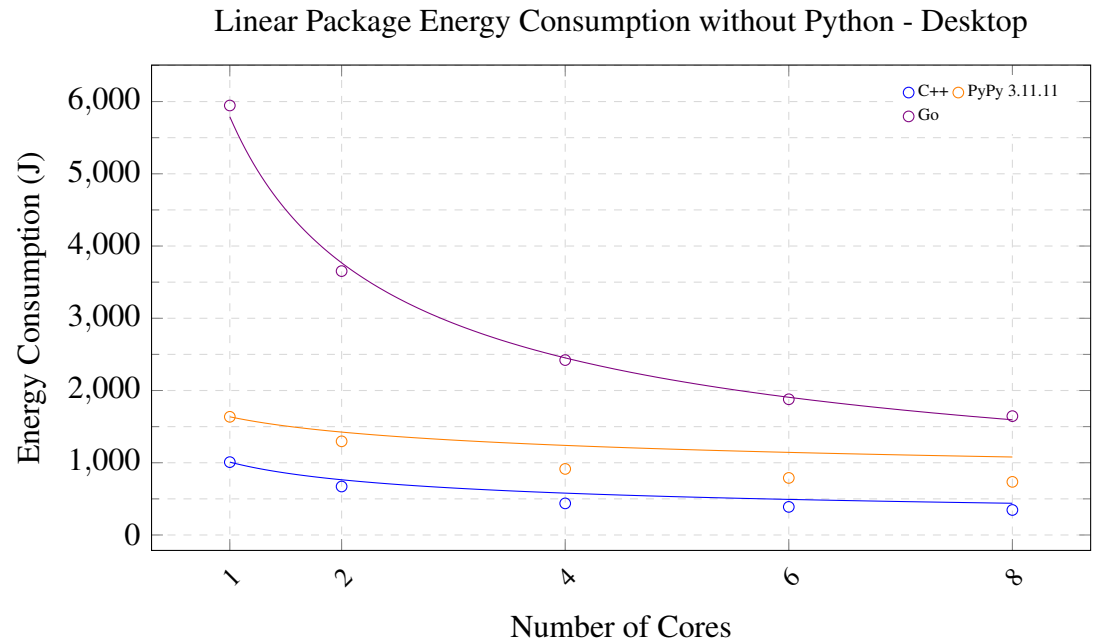


Fig. 5.11. Linear package energy consumption of the Desktop benchmark without Hyperthreading across different programming languages. source: Table B.2

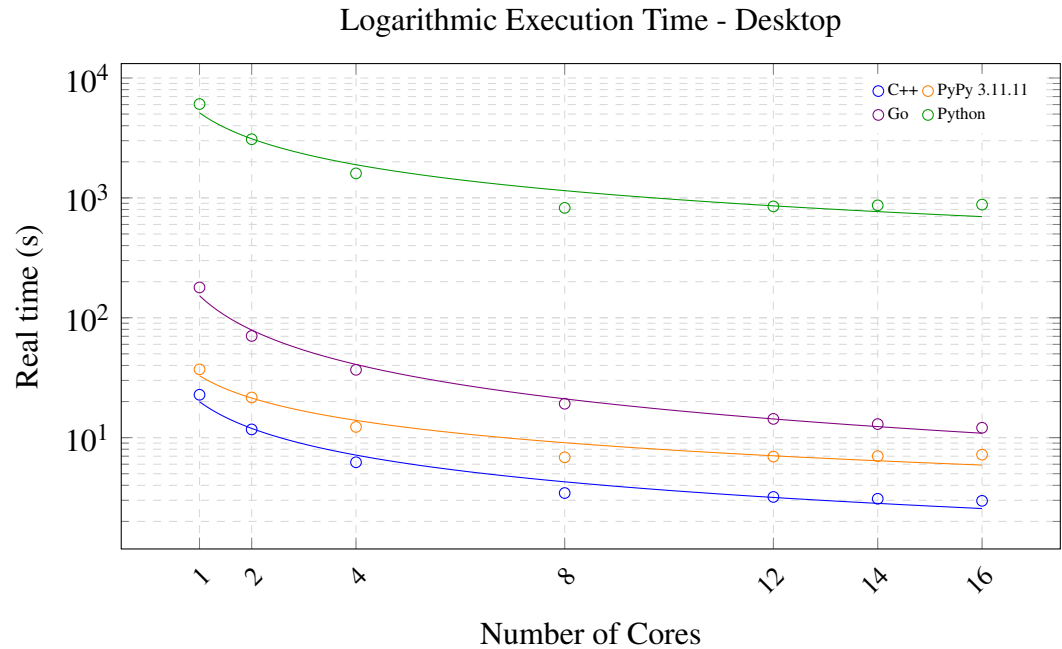


Fig. 5.12. Logarithm Execution time of the Desktop benchmark across different programming languages. source: Table B.3

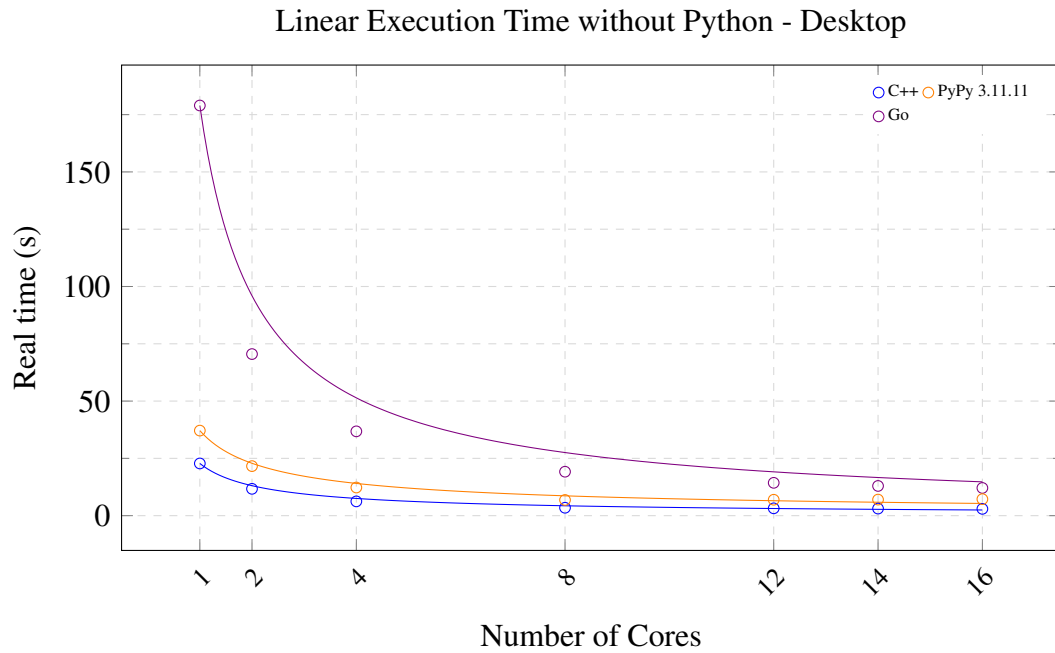


Fig. 5.13. Linear execution time of the laptop benchmark across different programming languages without Python. source: Table B.3

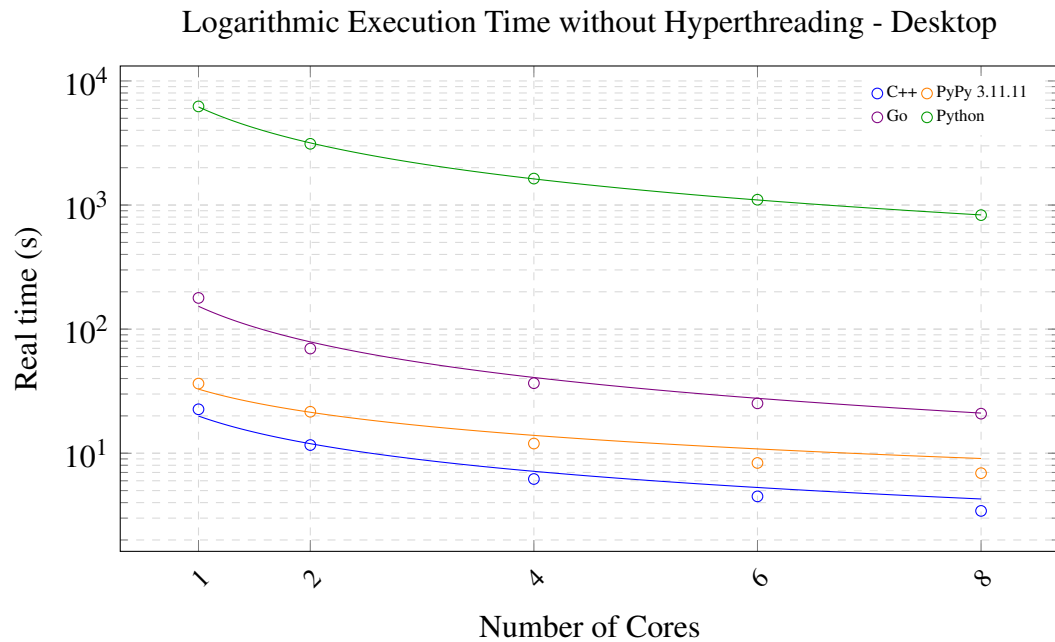


Fig. 5.14. Logarithm Execution time of the Desktop benchmark without Hyperthreading across different programming languages. source: Table B.4

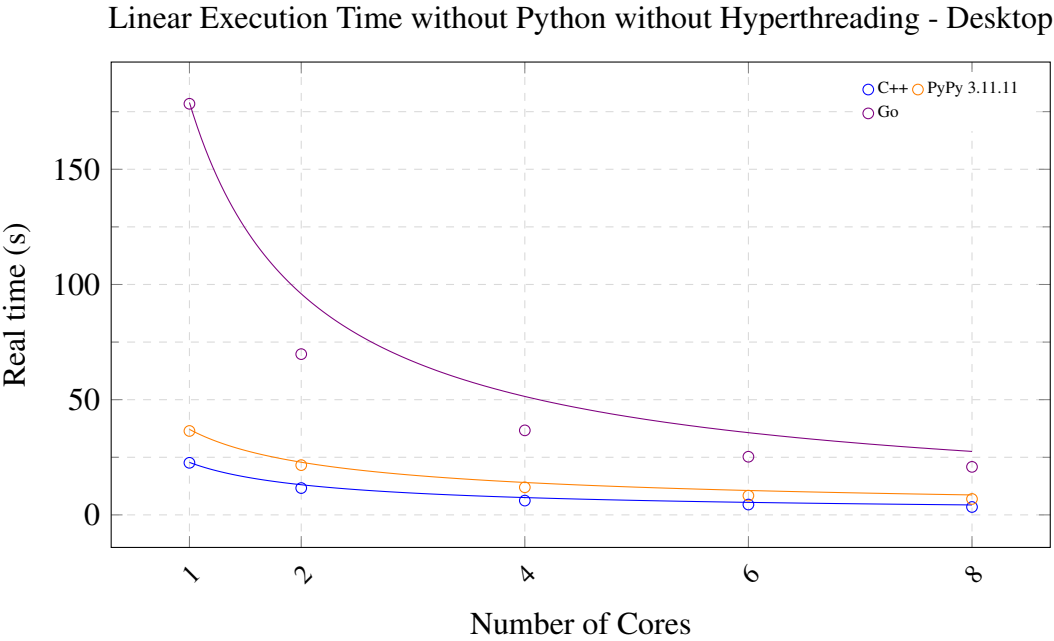


Fig. 5.15. Linear execution time of the laptop benchmark without Hyperthreading across different programming languages without Python. source: Table B.4

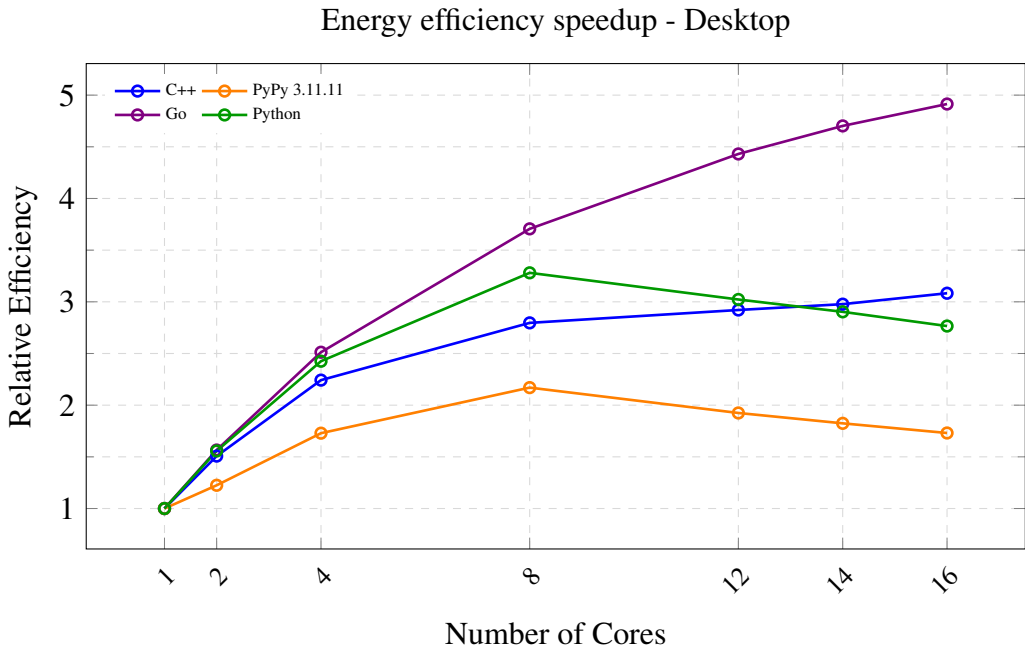


Fig. 5.16. Energy relative efficiency of the Desktop benchmark across different programming languages.

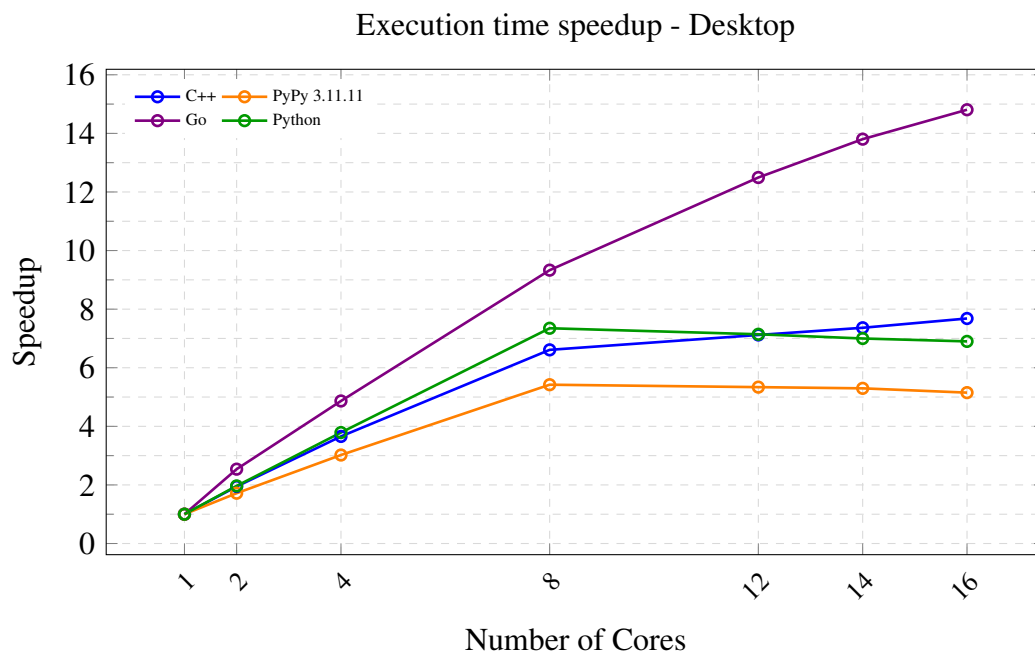


Fig. 5.17. Execution time speedup of the Desktop benchmark across different programming languages.

5.1.3. Laptop

This laptop is said to have one of the fastest single-core performance in the market. It has a 14 core ARM processor, using the big.LITTLE architecture, with 10 high performance cores and 4 high efficiency cores. It has 48GB of RAM, which is enough to run any of the tests.

This platform is a personal laptop, with a 14 core processor, the Apple M4 Pro, which has 10 high performance cores and 4 high efficiency cores, which is a big.LITTLE architecture. This means that the high performance cores are used for CPU intensive tasks, while the high efficiency cores are used for less demanding tasks, such as web browsing or watching videos. But as Apple does not allow the user setting the cores to be used by a specific process, like it happens on Linux, we can not test the high efficiency cores isolated from the high performance cores, as the operating system will decide for us which cores to use for each process.

Cache & Numa

Listing 5.3. Cache of the Apple M4 Pro by performance level

1	Performance Level 0 (High-Performance Cores (10x)):		
2	Level	Type	Size
3	L1d	Data	128 KB
4	L1i	Inst	192 KB
5	L2	Uni	16 MB
6			

7	Performance Level 1 (Efficiency Cores (4x)):		
8	Level	Type	Size
9	L1d	Data	64 KB
10	L1i	Inst	128 KB
11	L2	Uni	4 MB

Results

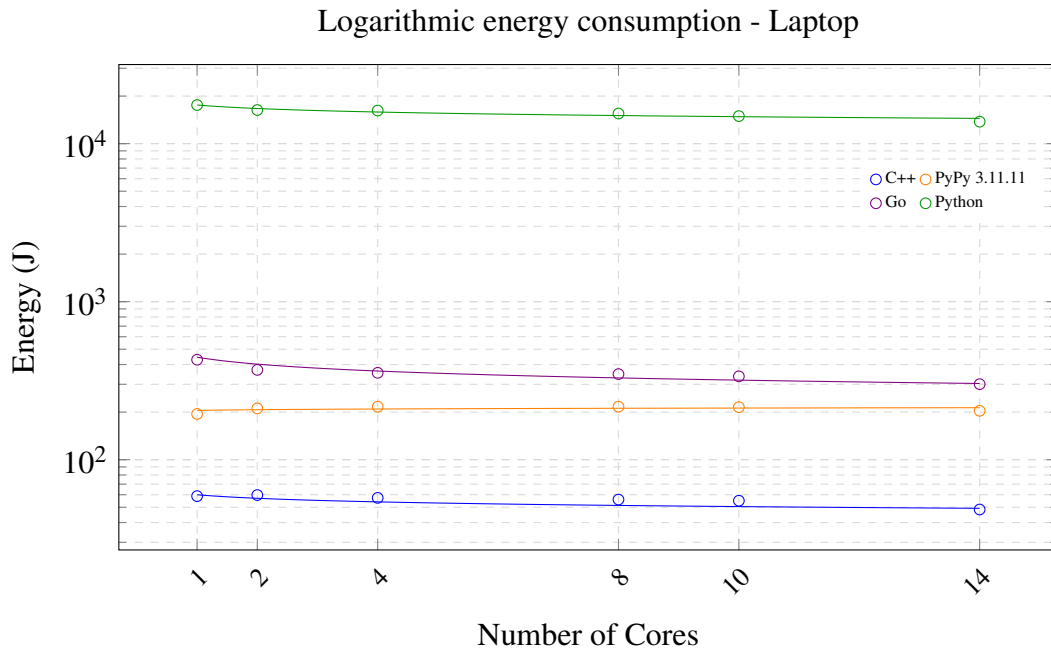


Fig. 5.18. Logarithm energy consumption of the Laptop benchmark across different programming languages. source: Table C.1

We can see from Figure 5.18 there does not seem to be a big difference between the execution with one or multiple cores, but this is due to the fact that the table has a logarithmic scale. If we take a look at Figure 5.19, we can see that the energy consumption decreases substantially with the Go implementation. But, strangely, both the C++ and PyPy implementation do not seem to reduce the energy consumption with multiple cores. This seems to be due to the fact that the Apple M4 Pro has a big.LITTLE architecture, and the operating system is not able to efficiently use the high performance cores when there are more than 10 cores available, as it is not able to schedule the tasks efficiently. It could also be that the CPU is drawing its maximum power (around 45 – 50W) when running the C++ and PyPy implementations, and thus the energy consumption does not change much with the number of cores as these are quite fast.⁴

Even though the energy consumption does not decrease, the execution time does, as we can see from Figure 5.20. The C++ implementation is the fastest, followed by the

⁴This is one of the points that should be studied forward, more testing with ARM chips and a better measurement of the power consumption, with a desktop computer such as the Mac Mini.

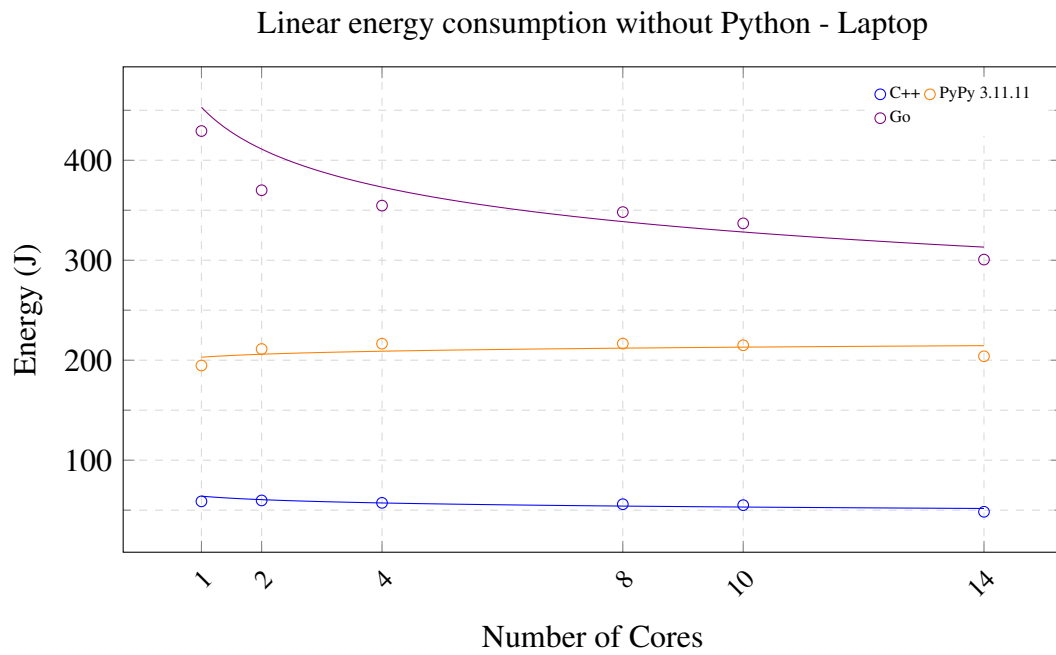


Fig. 5.19. Linear energy consumption of the laptop benchmark across different programming languages without Python. source: Table C.1

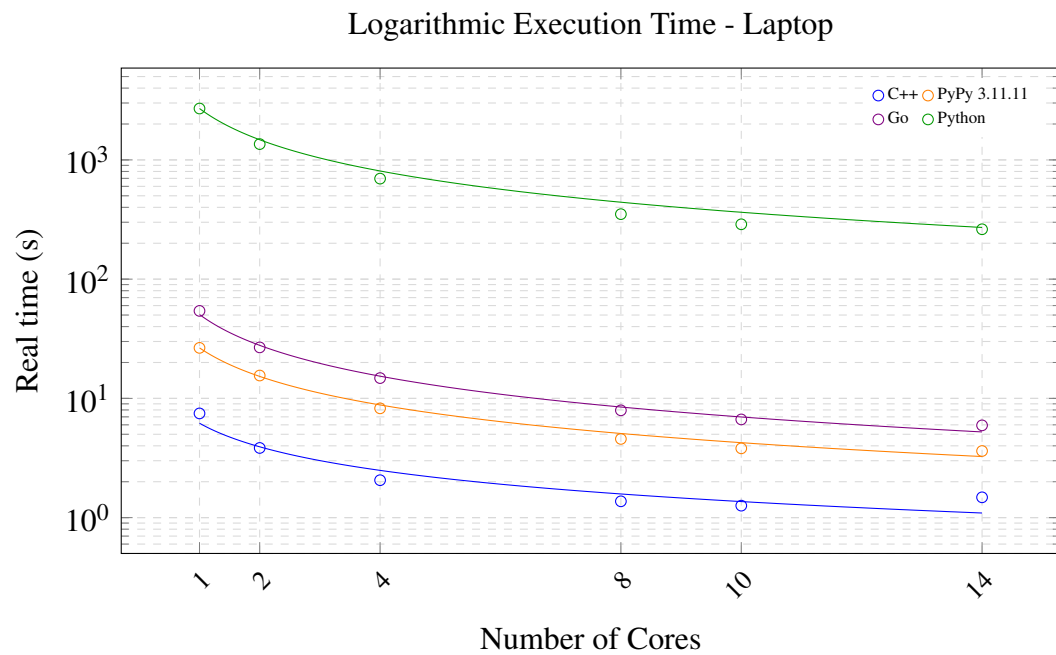


Fig. 5.20. Logarithm Execution time of the Laptop benchmark across different programming languages. source: Table C.2

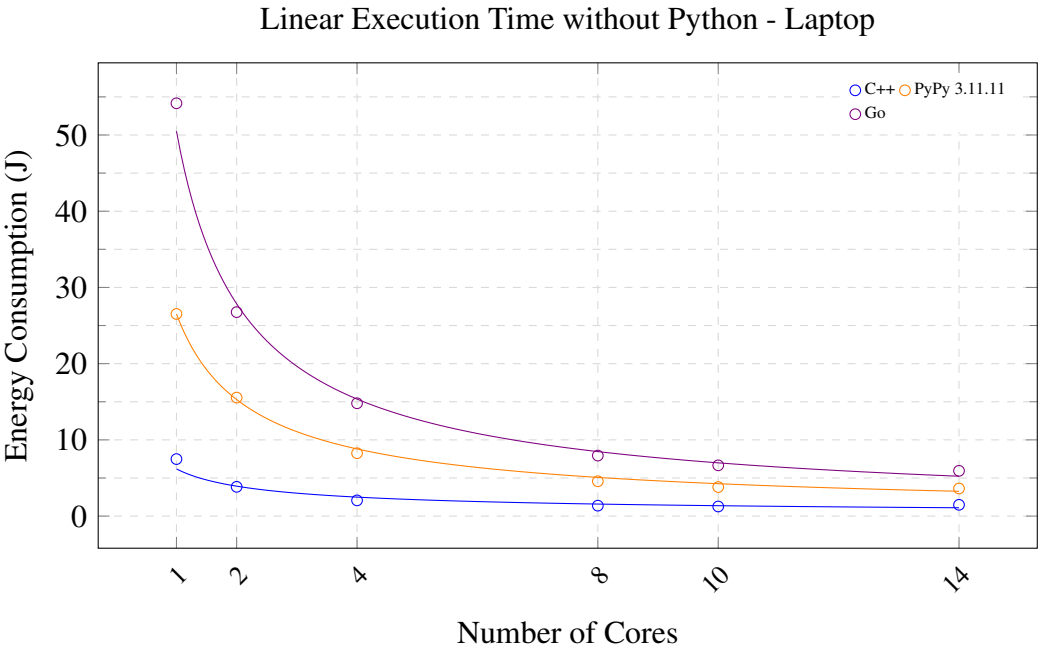


Fig. 5.21. Linear execution time of the laptop benchmark across different programming languages without Python. source: Table C.2

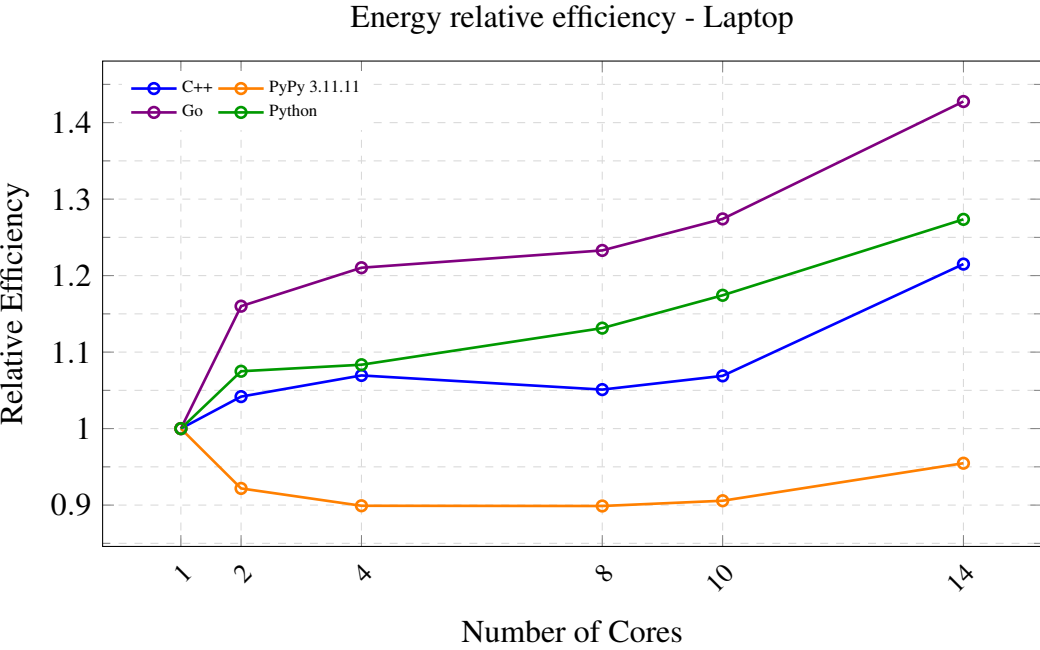


Fig. 5.22. Energy relative efficiency of the Laptop benchmark across different programming languages.

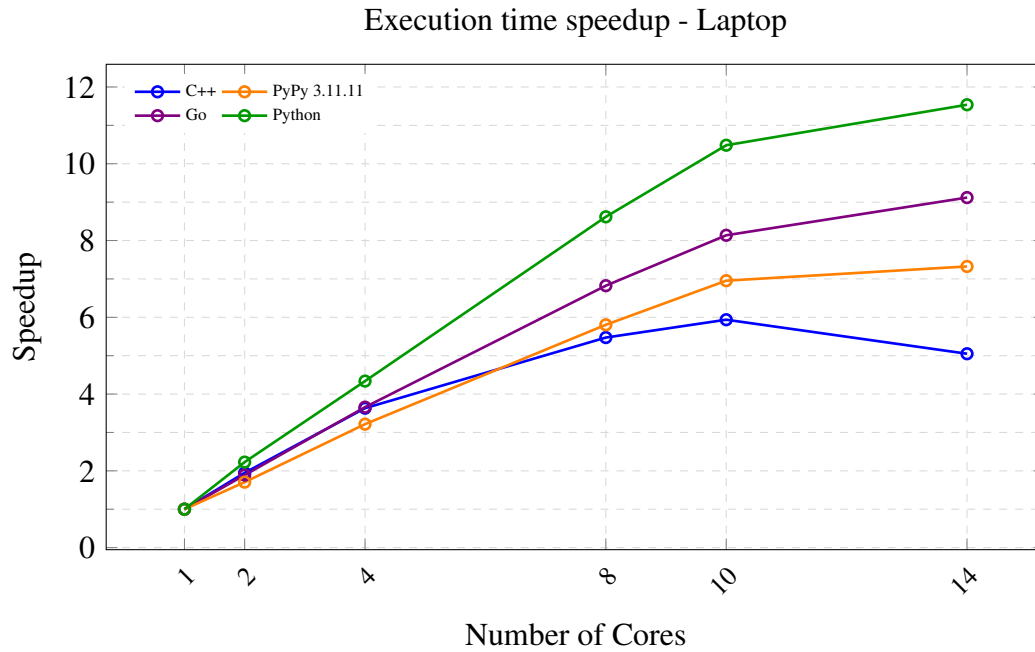


Fig. 5.23. Execution time speedup of the Laptop benchmark across different programming languages.

PyPy implementation, then the Go implementation, and finally the Python implementation, which is the slowest by a large margin. I also created this Figure 5.21 to better visualize the execution time of each implementation without Python, as it distorts the results due to its slow performance. Raw data for both energy consumption and execution time can be found in Table C.1 and Table C.2 respectively.

All the raw data for the benchmarks can be found in the Appendix C.

5.1.4. Raspberry Pi 5

This SBC (Single Board Computer) is one of the most popular in the market, and though is not a great platform for testing the performance of different programming languages its popularity has made it an interesting option as it has much less power requirements than the other alternatives. It has a 4 core ARM processor, 8GB of RAM, which is enough to run any of the tests.

Results

From both Figure 5.24 and Figure 5.25 we can see that the trend lines, in this case intersect, meaning that passing from 2 to 4 cores, the energy increases for the PyPy implementation and decreases for the rest of the implementation. This is due to the fact that the Raspberry Pi 5 has a much less powerful processor and thus, compiled languages such as C++ and Go can obtain better power efficiency results when using more cores, as the processor is able to handle the load better.

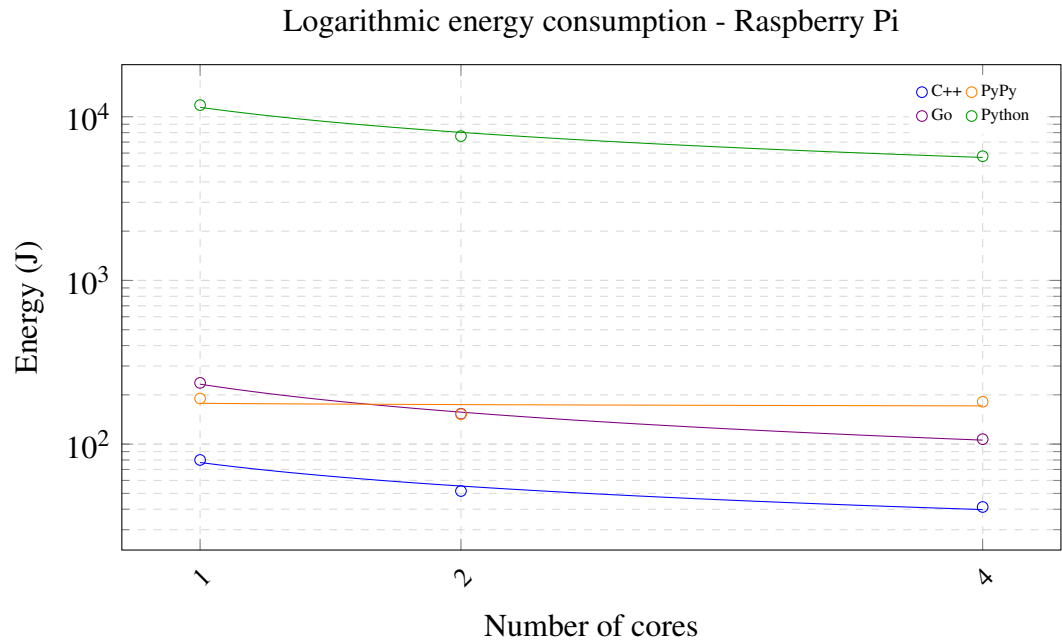


Fig. 5.24. Logarithm energy consumption of the Raspberry Pi benchmark across different programming languages. source: Table D.1

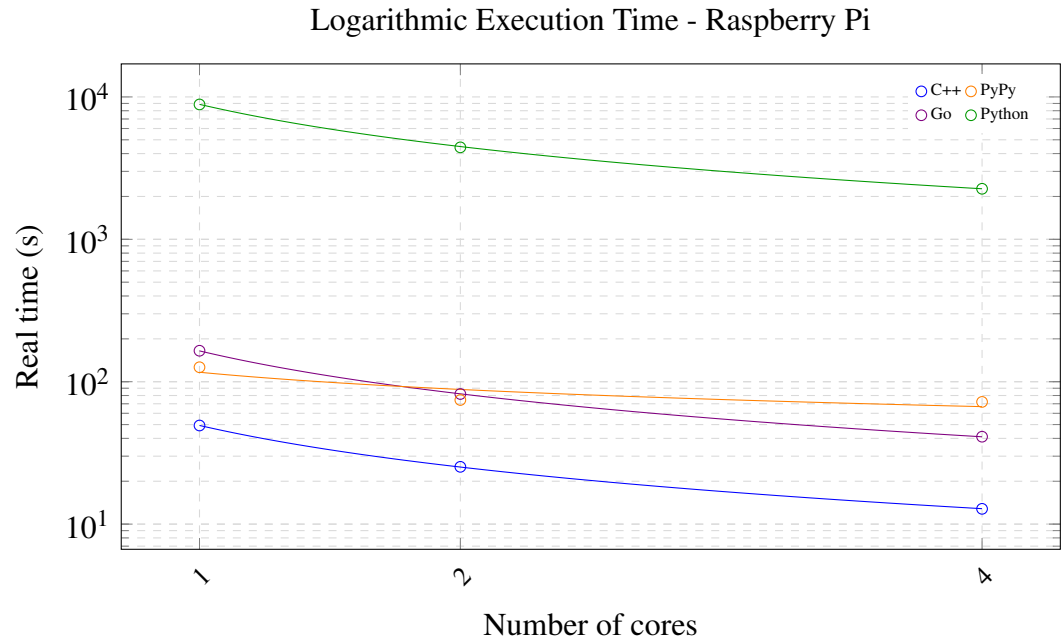


Fig. 5.25. Logarithm Execution time of the Raspberry Pi benchmark across different programming languages. source: Table D.2

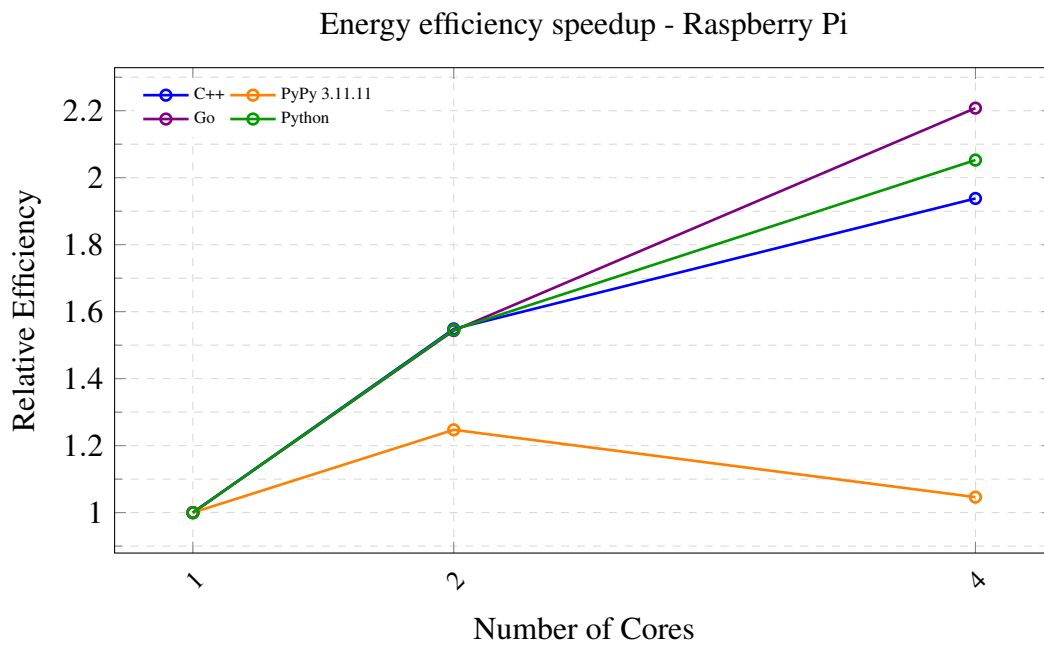


Fig. 5.26. Energy package relative efficiency of the Raspberry Pi benchmark across different programming languages.

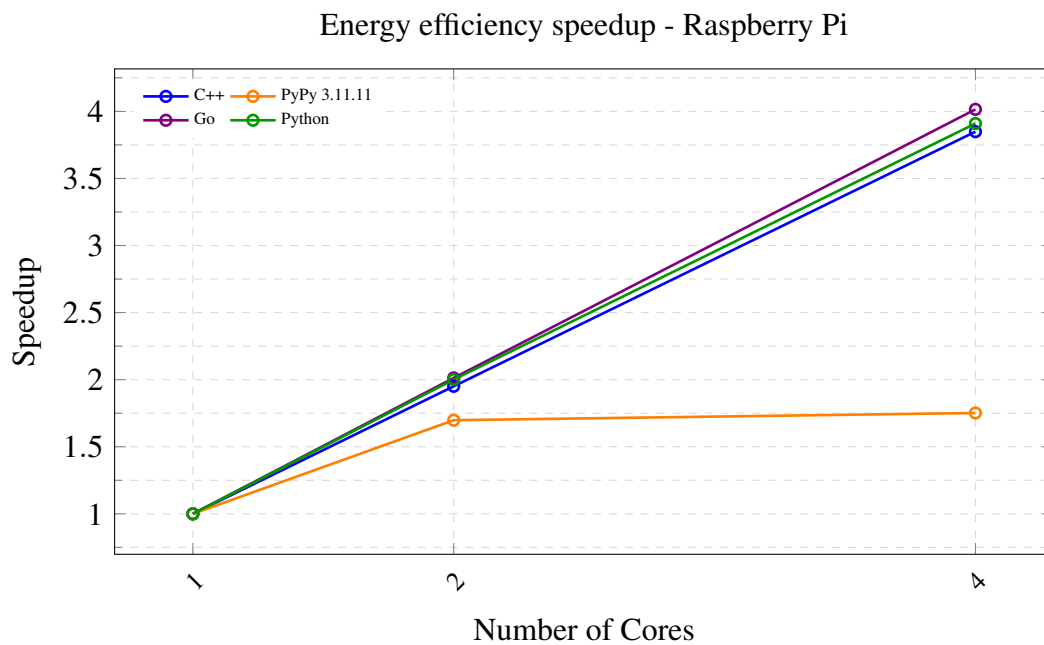


Fig. 5.27. Energy efficiency speedup of the Raspberry Pi benchmark across different programming languages.

It is impressive the slowness that python shows in this platform, 62x less power efficient than PyPy and consuming more than 147x the energy of the C++ implementation.

If we turn to the execution times, we can see that the C++ implementation is the fastest, followed by the Go implementation, then PyPy and lastly Python when looking at the 4 cores run of the benchmark. But, when the program was run with 2 cores, the PyPy implementation does not reduce much the execution time, compared to the Go result, interchanging the position of the Go and PyPy implementations, as it can be seen in clearer in the raw data in ?? and ??.

All the raw data for the benchmarks can be found in the ??.

5.2. Comment on parallelizing different languages

In this section, I would like to make some comment on the parallelization on different languages, and why some might experience a different behavior.

5.2.1. Go

When choosing how many ‘cores’ the tests are using, for the Go implementation, I used the size of the `waitChan` channel. This number can be changed to be more than the total number of threads in the system, which sometimes increases the performance.

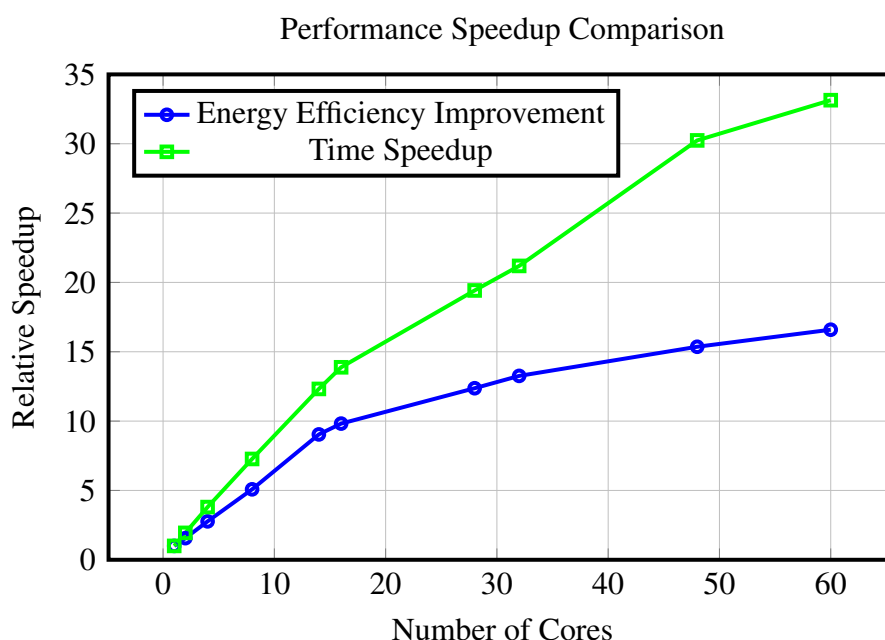


Fig. 5.28. Relative performance improvements showing time scales better than energy efficiency

As it can be seen from Table E.1, the Go implementation is able to use more than the total number of threads in the system, and it is able to use them efficiently, as the Go runtime is able to schedule the goroutines efficiently. We can also observe from the

table, that the results that are run in the same CPU chip versus different CPU chips, have similar energy consumptions, but the execution times are significantly lower as there are no context switches happening between the two CPUs. This can be seen in the 28 cores same CPU and 32 cores same CPU tests, marked in Table E.1 with a *.

5.2.2. Python

When iterating through every pixel in Python, as the environment has to be copied for every single pixel, the cores are not being used at 100% of their capacity, specifically, while testing I saw that the cores were being used at around 5%-15% of their capacity. Meaning the creation of too many threads is not beneficial, as the overhead of creating the threads is larger than the actual work being done by each thread. Another factor that Python, each time a task is submitted to a process, Python needs to serialize (pickle) the entire world object and other parameters, then deserialize them in the worker process, which means that, if this has to happen for every pixel, the serializing and deserializing tasks run for much longer than the actual pixel processing.

5.3. Most efficient language optimizations

As we can see from these results, the most efficient language in terms of energy consumption and execution time is C++.

But, out of the box, does C++ always provide the best performance? The answer is no, as the compiler plays an extremely important role in the performance of the code, and the compiler optimizations can make a huge difference in the performance of the code. For these tests, I tested many optimization flags, such as `-O3` and `-march=native`, which allows the compiler to optimize the code for the specific architecture of the machine it is being compiled on. But what would happen if we used different compiler flags, would the results change? Would another language be more efficient?

As to not leave the reader with the intrigue of what would happen if we used different compiler flags, I have compiled and tested all the programs with the following flags:

- `-O0`: No optimizations, the compiler will not optimize the code at all.
- `-O1`: Basic optimizations; the compiler will optimize the code. It performs a basic cleanup, removing dead code and some simple inlining.
- `-O2`: More optimizations; this is the recommended optimization level for most use cases.
- `-O3`: Maximum optimizations, very aggressive optimizations:

- Loop transformations: Unrolling loops even more than `-O2`, changing the distribution of loops and interchanging them.
 - Speculative optimizations
 - Vectorization: SIMD instructions (AVX, SSE, etc.)
 - Predictive commoning (reusing computations from previous loop iterations)
- **`-O3` with `-fast-math`**: Maximum optimizations; the compiler will optimize the code as much as possible, but operations will not be as precise.

As [35] explains, the `-fast-math` flag allows the compiler to perform optimizations that may not be mathematically correct, but will result in faster code.

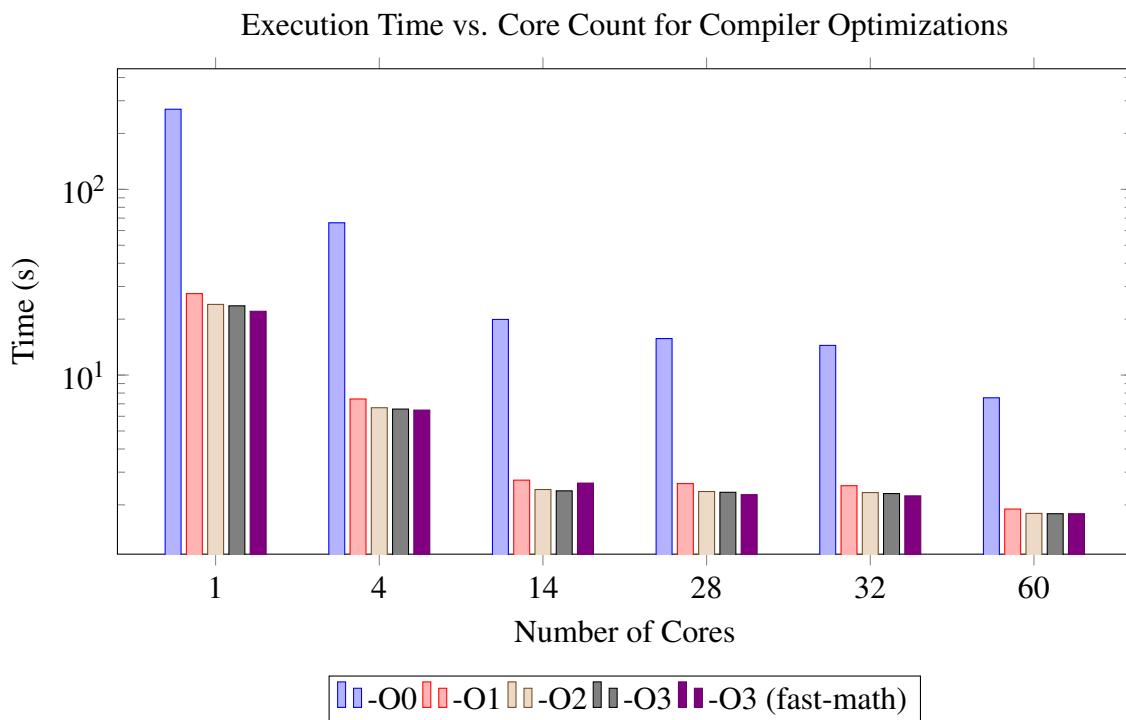


Fig. 5.29. Execution time (log scale) for various core counts and compiler flags.

From the Table F.1, we can see that `-O3` can be up to 13.13x more efficient and 11.44x faster than `-O0`, and `-O3` and 13.22x more efficient and 12.25x faster if we add the `-fast-math` flag on a single core task. But when we change and add more cores, the improvement decreases up to 4.24x faster more energy efficient and 4.16x faster with `-O3` and consume 4.22x less energy and take 4.11x less time with `-O3 -fast-math`.

Thus, we can see that the compiler optimizations play a very important role in the performance of the code, and that C++ is not always the most efficient language, as it depends on the compiler optimizations used. In conclusions, for optimization flags, even though `-O3` with `-fast-math` is the most efficient, I would recommend using `-O3` for high-performance computing, as it provides a good balance between performance and accuracy. If the program is just a regular program, I would recommend using `-O2`, as it

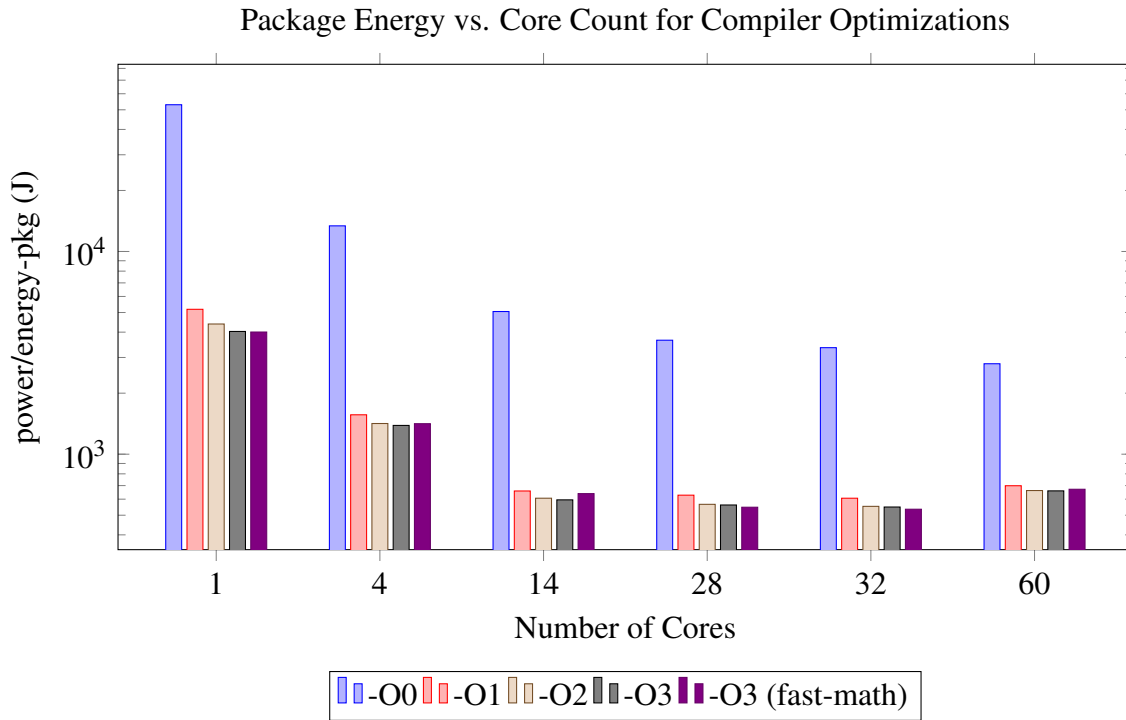


Fig. 5.30. Package energy consumption (log scale) for various core counts and compiler flags.

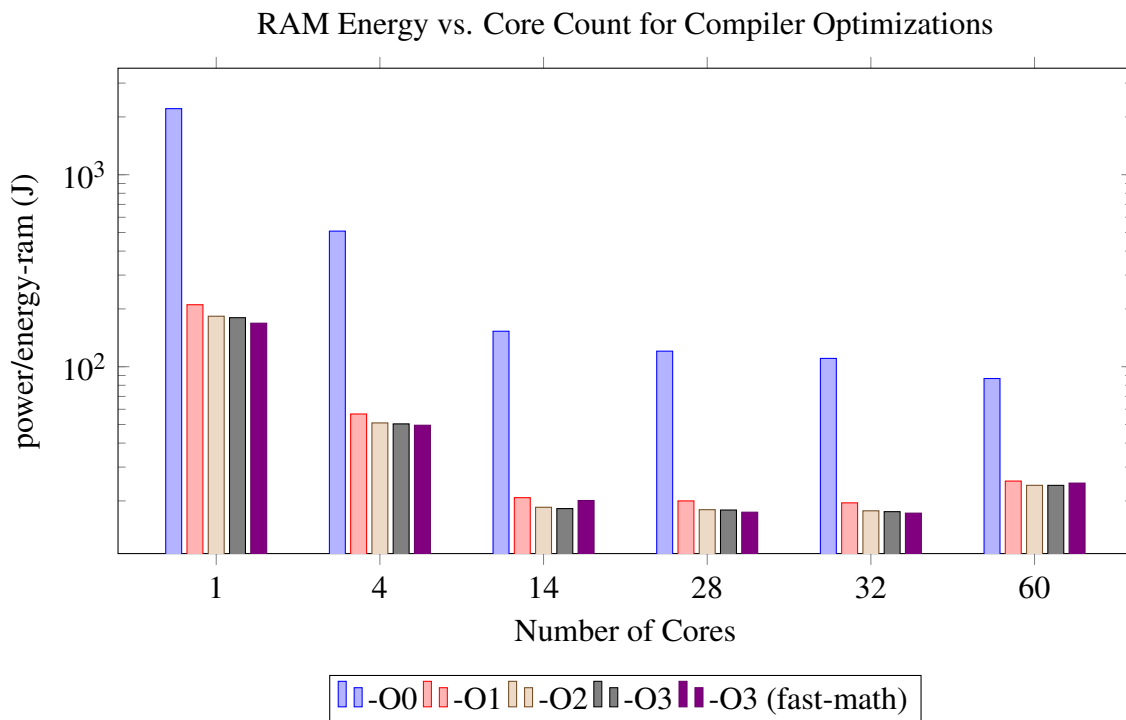


Fig. 5.31. RAM energy consumption (log scale) for various core counts and compiler flags.

provides a good balance between performance and code size as well as compilation time, which we have not taken into account as the program is just compiled once and may be executed thousands of times.

CHAPTER 6

PLANNING

This chapter will cover the plan followed for the development of this project. It will include the initial plan, the changes made during the project and the final plan.

6.1. Initial Plan

The initial plan for the development of this project was to create a suite of benchmarks, and a framework for evaluating the performance of different programming models on multiple computing systems.

The benchmarks were to be implemented in three different programming languages: C++, Go, and Python. The framework was to be designed to allow for easy integration of new benchmarks and to provide a way to measure the performance and energy consumption of the benchmarks on different hardware platforms.

The development of the project started on February 19th 2025 and was completed on August of the same year. The Gantt diagram (Figure 6.1) illustrates the project plan shows the different phases of the project, including the initial research, the development of the benchmarks, the implementation of the framework, and the evaluation of the results.

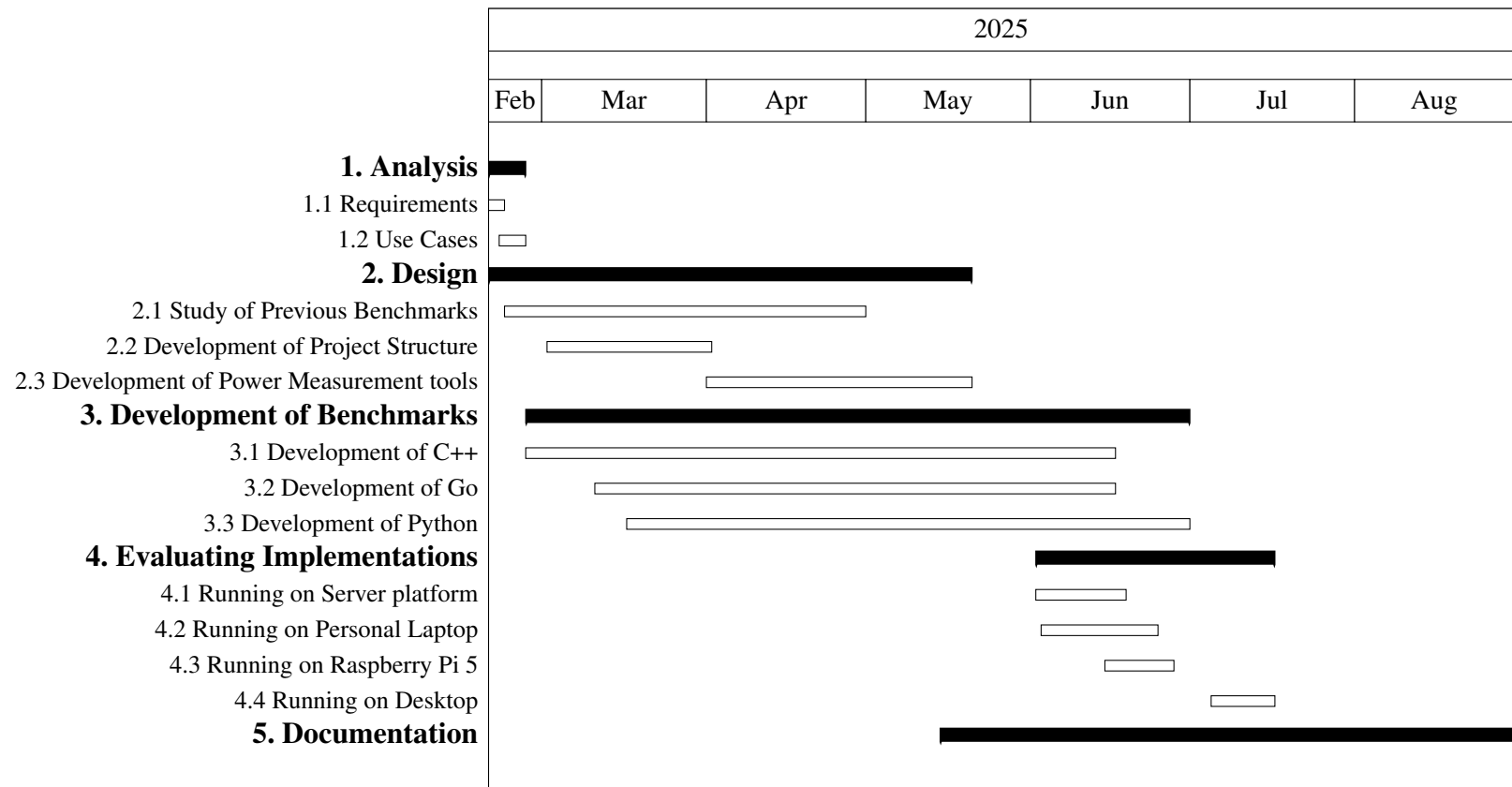


Fig. 6.1. Gantt Diagram of the project plan.

1. Analysis: Define requirements and use cases.
 - Gather project requirements.
 - Identify and document use cases.
2. Design: Study previous benchmarks and develop the project structure along with power measurement tools.
 - Review and analyze existing benchmark suites.
 - Design the architecture of the benchmarking framework.
 - Develop tools for power measurement.
3. Development of Benchmarks: Implement benchmarks in C++, Go, and Python.
 - Implement C++ benchmarks.
 - Implement Go benchmarks.
 - Implement Python benchmarks.
 - Validate and test each benchmark implementation.
4. Evaluation of Implementations: Run benchmarks on server, personal laptop, Raspberry Pi 5, and personal desktop.
 - Set up each hardware platform.
 - Execute benchmarks on each platform.
 - Collect and analyze performance and power data.
5. Documentation: Compile all findings and prepare the final documentation.
 - Document methodology and results.
 - Prepare diagrams and charts.
 - Write and review the final report.

CHAPTER 7

SOCIOECONOMIC ENVIRONMENT & SUSTAINABLE DEVELOPMENT GOALS

7.1. Budget

In this section of the report, an analysis of the budget for the project is presented. The budget includes the costs associated with the development of the benchmarks, the framework, and the evaluation of the results. The budget is divided into different categories, including human resources, material resources, software, and indirect costs.

7.1.1. Human Resources

The human resources budget includes the costs associated with the personnel involved in the project. This includes salaries, benefits, and any other costs related to the project team. The total cost for human resources is estimated to be €9,000. This budget is based on the assumption that the project will require a total of 300 hours of work, with an hourly rate of €30, which is a reasonable rate for software development and project management in Spain. If we were to consider a different region, such as the United States, the hourly rate could be significantly higher, potentially reaching €100/hour or more, which would increase the total cost to €30,000 or more.

TABLE 7.1
HUMAN COSTS

Total Hours	Hourly Rate	Total Cost
300 hours	€30/hour	€9,000

7.1.2. Material Resources

The material resources budget includes the costs associated with the physical resources needed for the project. This includes hardware, software, and any other materials required for the development and evaluation of the benchmarks.

TABLE 7.2
MATERIAL RESOURCES COSTS

Category	Description	Cost	Lifetime (months)	Usage (months)	Cost (month)	Amortized Cost
Personal Laptop	Development & testing	3,444€	48	7	71.75€	502.25€
Professional Server	Development & testing	4,500€	24	1	187.50€	187.50€
Raspberry Pi 5	Development & testing	85€	12	2	7.08€	14.17€
Desktop	Development & testing	2,000€	72	2	27.78€	55.56€
Hardware	Power measurement	20€	12	1	1.67€	1.67€
Software	Licenses, tools, etc.	0€	∞	3	0€	0.00€
Total						761.14€

Hardware

The hardware budget includes the costs associated with the physical machines used for testing and evaluation. This includes the server used for testing, a personal computer, one laptop and a Raspberry Pi 5 as compute nodes. I also acquired a power measurement device. The total cost for hardware is estimated to be 761.14€.

7.1.3. Software

The software budget would include the costs associated with the software tools and licenses needed for the project. As this project has been developed using open-source software or free tools, the software budget is estimated to be €0. The free for everyone software used includes:

- **VS Code:** Served as the primary Integrated Development Environment.
- **CMake:** Managed the build process in a cross-platform environment.
- **Ubuntu:** Operating System for most of the testing.
- **MacOS:** Supported development and compatibility checks on an alternative OS.
- **L^AT_EX** Used for formatting and compiling the project documentation.

But there are some software licenses that were used during the project, but they were not paid for, as they were provided by the fact that these tools have an education license. The following table summarizes the software licensing costs:

TABLE 7.3
SOFTWARE LICENSING COSTS

Product	Monthly Cost (€)	Usage (Months)	Amortized Cost (€)
CLion	0.0	7	0.0
GitHub Pro	0.0	7	0.0
GitHub Copilot	0.0	7	0.0
Total	0.00 €		0.00 €

7.1.4. Indirect Costs

The indirect costs budget includes any costs that are not directly attributable to the project but are necessary for its completion. This includes overhead costs, administrative expenses, and any other indirect costs associated with the project. Some of these costs may include office space, utilities, and other general expenses that are necessary for the project but not directly related to the development or evaluation of the benchmarks:

- Office space: €200/month
- Utilities (electricity, internet, etc.): €100/month
- Administrative expenses: €50/month

Assuming the project runs for 6 months, and the work is performed remotely, eliminating the need for office space, the total indirect costs can be estimated to be: 900€.

7.1.5. Total Cost & Offer

The total cost for the project is the sum of all the individual budget categories. This includes human resources, material resources, software, and indirect costs. The total estimated cost for the project is 10,661.14€.

Before stating the offer, a summary of the project is provided in Table 7.4. And finally, the cost breakdown is presented in Table 7.5.

TABLE 7.4
PROJECT INFORMATION

Title	Evaluating performance and energy impact of programming languages
Author	Eduardo Alarcón Navarro
Start Date	2025-02-19
End Date	2025-08-30
Duration	7 months
Project Offer	10,661.14€

TABLE 7.5
COST BREAKDOWN

Concept	Increase	Partial Cost	Aggregated Cost
Total Cost	-	10,661.14€	10,661.14€
Risk	20%	2,132.23€	12,793.37€
Profit	15%	1,919.01€	14,712.38€
Taxes	21%	3,089.60€	17,801.98€
Total (56%)			17,801.98€

7.2. Socioeconomic Impact

The following project was mainly done for the research purposes, prototyping, and establishing a framework to measure the impact of programming languages on energy consumption, contributing to open-source software.

The project has the potential to impact the software development community by providing a framework that can be used to evaluate the performance and energy consumption of different programming languages. This can lead to more efficient software development practices and contribute to the overall sustainability of software engineering.

Further development of the programs can be implemented, but not many companies will use this framework, as it is not a common practice to measure the energy consumption of programs in most companies.

7.3. Sustainable Development Goals

As the UN states, the Sustainable Development Goals (SDGs) [36] are a universal call to action to end poverty, protect the planet, and ensure prosperity for all by 2030. This project aligns with several of the SDGs, particularly:

- **Goal 7: Affordable and Clean Energy** - By evaluating the energy consumption of different programming languages, this project contributes to the understanding of how software can be optimized for energy efficiency, supporting the transition to more sustainable energy practices.
- **Goal 9: Industry, Innovation, and Infrastructure** - The project promotes innovation in software development by providing a framework for evaluating programming languages, which can lead to more efficient and sustainable software solutions.
- **Goal 12: Responsible Consumption and Production** - By focusing on the energy consumption of software, this project encourages responsible consumption of

resources in the software industry, promoting sustainability in technology development.

CHAPTER 8

REGULATORY FRAMEWORK

This chapter provides an overview of the regulatory framework that governs the development and deployment of the benchmarks, framework, and evaluation processes. It outlines the relevant laws, regulations, and standards that must be adhered to in order to ensure compliance and ethical considerations throughout the project lifecycle.

In Europe, the General Data Protection Regulation (GDPR) [37] is widely recognized as a key regulatory standard in the software domain. However, because this work neither collects, processes, nor uses any user data, the GDPR does not apply. Instead, the project makes extensive use of third-party software and established technical standards.

8.1. Software Licenses

Several software licenses govern the use of the tools and libraries employed in this project. The following tools are particularly relevant:

- **CMake:** CMake is an open-source, cross-platform build system designed for several programming languages. It is distributed under the 3-Clause BSD License [38], which allows use, modification and distribution of the code as long as the copyright notice file is retained. Other restrictions regarding the copyright holders apply.
- **LLVM:** The LLVM project is a collection of modular and reusable compiler and toolchain technologies. It was distributed under the University of Illinois/NCSA Open Source License [39], which allowed for use, modification, and distribution with certain conditions regarding the copyright notice and disclaimer of warranties. However, the license has since been updated to the Apache License 2.0 [40], which is more permissive and widely accepted in the open-source community.
- **CLion:** CLion is a commercial Integrated Development Environment (IDE) for C

and C++ programming languages developed by JetBrains. It is distributed under a proprietary license, which allows for use and distribution under specific conditions set by the copyright holder. [41]

- **Python:** The programming language has a *PSF* license [42] which is a permissive open-source license that allows for use, modification, and distribution of the Python programming language and its libraries.
- **Go:** The Go programming language is distributed under a BSD-style license [43], which allows for use, modification, and distribution of the language and its libraries with certain conditions regarding the copyright notice and disclaimer of warranties. The license allows anyone to freely use, modify, and distribute the software for any purpose, including commercially, as long as the individual provides proper attribution and do not use the original authors' names for endorsement.

8.2. Technical Standards

- **C++ Standard:** The C++ programming language is governed by the standard [44], which defines the syntax, semantics, and libraries of the language. This standard ensures that C++ code is portable and consistent across different compilers and platforms. For this project, the C++20 standard [45] has been used.

8.3. License of the Project

The project is licensed under the MIT License [46], which is a permissive open-source license that allows for use, modification, and distribution of the software. This license ensures that the project can be freely used and adapted by anyone, including for commercial purposes, as long as the original copyright notice and license terms are retained.

CHAPTER 9

CONCLUSIONS

After analyzing the data collected from the various experiments, I clearly confirmed the election of a programming language does affect the performance and energy consumption.

I found that the most efficient programming language, by a great margin is C++, outperforming in every case. The results indicate that it outperforms both in energy consumption and execution time when compared to Go, a compiled language, and interpretations of languages, such as CPython or PyPy, the other languages and implementations of languages used in this project. This was to be expected, as C++ is a compiled language and is known for its performance and energy efficiency, as it is an evolution of C.

The program used to perform the benchmarks is an image renderer, that processes each pixel of an image, casting multiple rays per pixel, and calculating the color of each one based on the intersection of the rays with the objects in the scene. This is a computationally intensive task that uses the Arithmetic Logic Unit (ALU), that can be parallelized.

This benchmark has been performed on the same component, the CPU, even when performing the benchmarks on different platforms, such as a desktop computer, a laptop, and a Single Board Computer (SBC).

9.1. Summary of Findings

The main findings of this project are that C++ is up to 440 times faster and 300 times more energy efficient on the Server than CPython, and up to 180 times faster and 150 times more energy efficient on the Raspberry Pi (the two extremes of the platforms used in this project).

On every test case, C++ outperformed the other programming languages, given the same number of cores and platform. An interesting result is that the lowest consuming

board has been the Raspberry Pi, due to its design for IoT low-consumption devices. For each of the languages tested, the performance on up to 4 cores on each platform is detailed on this graph:

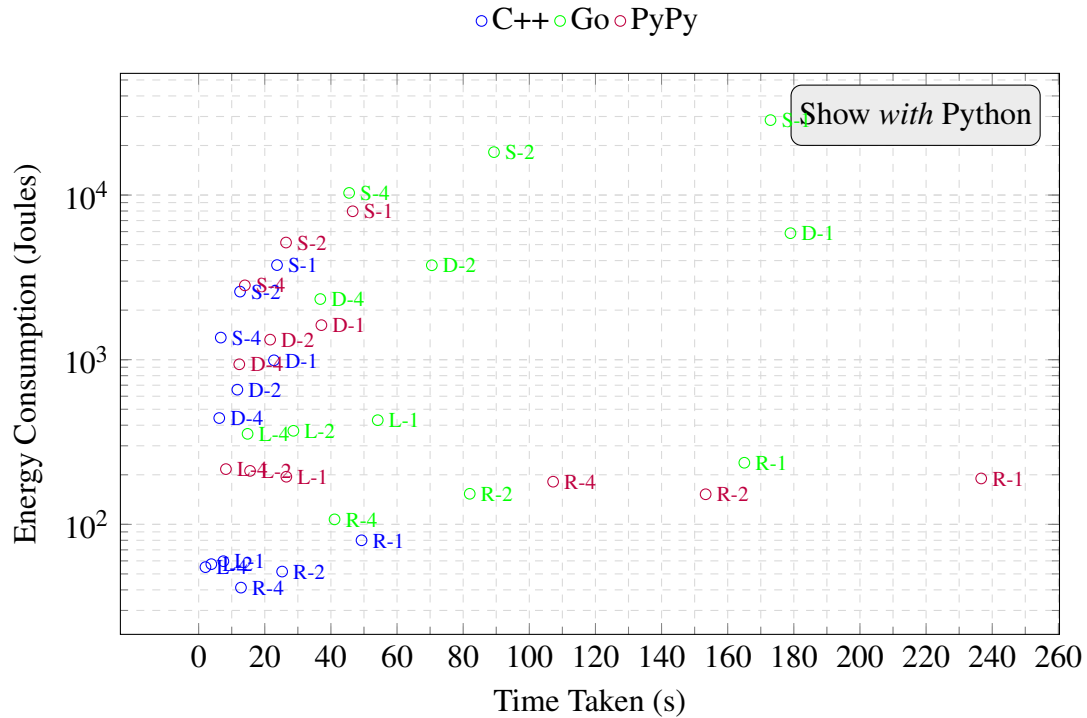


Fig. 9.1. The plot shows the relationship between time taken and energy consumption for each programming language. The data points represent different implementations, where the label has the form X-Y, with X being the platform and Y being the number of cores used.

9.2. Discussion of Results

The most impressive and surprising result for many tests has been the performance of PyPy compared to the other programming languages. PyPy is just an interpreter for Python (that can run a limited subset of what CPython can run), but it uses a JIT compilation technique that allows it to execute Python code much faster than the standard interpreter. This JIT is not compatible with some python libraries such as Numpy [47] as these libraries use C extension to speed up CPython's speed. This means that PyPy can only be used with pure Python code, which limits its applicability in some cases.

The implementation of multiple threads in the different programming languages was also a key factor in the performance of the programs. Depending on the technique used, parallelizing for each line or for each pixel, the performance deferred on the different languages. The easiest to implement, and the one that showed the best results, was C++, followed by Go, with its goroutines and channels. Python's default implementation, CPython, does not support true multithreading due to the GIL, which limits the

performance of multi-threaded programs, thus, the need of the library multithreading is needed.

If we were to use libraries such as Numpy, the performance of CPython would be much better, as these libraries are optimized for performance and can take advantage of the underlying hardware. However, this would not be a fair comparison, as we would not be using pure Python code.

If I was to deploy a render-farm, with this program, I could be paying 300x less energy costs if I used C++ instead of CPython or Go, and renders could take up to 400x less time. This is a significant difference, and it is clear that C++ is the best choice for these types of application.

There is an interesting outlier in this analysis, which are the results of PyPy on the laptop. Specifically, if we look at the results of the relative energy efficiency (Figure 5.22), we can see that PyPy, as the number of cores increase, the relative energy efficiency decreases, while the other programming languages increase. This seems to indicate that PyPy is not able to take advantage of the multiple cores in the laptop as well as other platforms. I have some possible ideas on why this might happen; My first hypothesis was that the implementation being used on the laptop was an emulation of the x86 version of PyPy using Rosetta 2 ([48]), but when checking the ‘Activity Monitor’, it is clear that this process is native to ARM. My second hypothesis is that this is due to the fact that, as this is one of the platforms that does not use x86 architecture, the JIT compilation is not as effective.

In conclusion, if the program is to be run only a couple of times, and the performance is not a key factor, CPython is a good enough choice. However, if the program is to be run multiple times, or if performance is a key factor, C++ is the best choice. Go is also a good choice, as it has a good balance between performance and ease of use, specially for hundreds of threads working light tasks, for example a web server. PyPy is a good choice for pure Python code, but it has limitations when it comes to libraries that use C extensions.

This comes as an interesting conclusion, as when I think about a program or function that has to be run multiple times, I think about a web serverless function, for example AWS Lambda ([49]), but in the [examples](#) they provide, there is no mention either of PyPy or C++. The main focus they have is Node.js (a JavaScript runtime), Python, Ruby, Java, and, in the fifth place, Go. This is an interesting choice, as it seems that the focus is on ease of use rather than performance. I think this is also due to the fact that these services charge by execution time, so the slower the execution time, the more they can charge the customer.

9.3. Future Research

Future research could focus on the following areas:

- Investigating the performance of other programming languages, such as Rust, Javascript, or Java, in similar scenarios.
- Exploring the impact of different libraries and frameworks on the performance of the programming languages.
- Analyzing the performance external accelerators such as GPU or fpga in conjunction with the programming languages.
- Conducting a more extensive analysis of the performance of the programming languages in different scenarios, such as web applications, data processing, or machine learning.
- Analyzing more workloads, such a web-server or a database, to see how the programming languages perform in these scenarios.
- Investigating the impact of different hardware configurations on the performance of the programming languages such as the RAM speed, size and SSD storage speed.
- Exploring the impact of different operating systems on the performance of the programming languages, such as Linux, Windows, or macOS.

BIBLIOGRAPHY

- [1] E. Masanet, A. Shehabi, N. Lei, S. Smith, and J. Koomey, “Recalibrating global data center energy-use estimates,” *Science*, vol. 367, no. 6481, pp. 984–986, 2020. DOI: [10.1126/science.aba3758](https://doi.org/10.1126/science.aba3758). eprint: <https://www.science.org/doi/pdf/10.1126/science.aba3758>. [Online]. Available: <https://www.science.org/doi/abs/10.1126/science.aba3758>.
- [2] H. Ritchie, P. Rosado, and M. Roser. “Energy production and consumption.” [Online]. Available: <https://ourworldindata.org/energy-production-consumption>.
- [3] P. Shirley, T. D. Black, and S. Hollasch. “Ray tracing in one weekend.” [Online]. Available: <https://raytracing.github.io/books/RayTracingInOneWeekend.html>.
- [4] R. Muralidhar, R. Borovica-Gajic, and R. Buyya, “Energy efficient computing systems: Architectures, abstractions and modeling to techniques and standards,” *CoRR*, vol. abs/2007.09976, 2020. arXiv: [2007.09976](https://arxiv.org/abs/2007.09976). [Online]. Available: <https://arxiv.org/abs/2007.09976>.
- [5] A. Muc, T. Muchowski, M. Kluczyk2, and A. Szeleziński, “Analysis of the use of undervolting to reduce electricity consumption and environmental impact of computers,” *Rocznik Ochrona Środowiska*, 2020.
- [6] Intel. “Maximize roi and performance for demanding workloads with intel avx-512.” [Online]. Available: <https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/what-is-intel-avx-512.html>.
- [7] E. Padoin, L. L. Pilla, M. Castro, F. Z. Boito, P. Navaux, and J.-F. Méhaut, “Performance/energy trade-off in scientific computing: The case of arm big.little and intel sandy bridge,” *IET Computers & Digital Techniques*, vol. 9, pp. 27–35, Dec. 2014. DOI: [10.1049/iet-cdt.2014.0074](https://doi.org/10.1049/iet-cdt.2014.0074).
- [8] C. Moir. “The remarkable story of arm.” [Online]. Available: <https://medium.com/swlh/the-remarkable-story-of-arm-85760399c38d>.
- [9] W. Wolff and B. Porter, “Performance optimization on big.little architectures: A memory-latency aware approach,” in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES ’20, London, United Kingdom: Association for Computing Machinery, 2020, pp. 51–61. DOI: [10.1145/3372799.3394370](https://doi.org/10.1145/3372799.3394370). [Online]. Available: <https://doi.org/10.1145/3372799.3394370>.

- [10] A. Holdings, “Big.little technology: The future of mobile,” ARM Limited, White Paper, Sep. 2013. Accessed: Jun. 7, 2025. [Online]. Available: <https://armkei1.blob.core.windows.net/developer/Files/pdf/white-paper/big-little-technology-the-future-of-mobile.pdf>.
- [11] J. Hammer, “Design and implementation of an automated performance modeling toolkit for regular loop kernels,” Ph.D. dissertation, Jan. 2023. DOI: [10.25593/opus4-fau-21514](https://doi.org/10.25593/opus4-fau-21514). Accessed: Aug. 28, 2025.
- [12] M. Gibbs, “Design and implementation of pay for performance,” *SSRN Electronic Journal*, pp. 35–36, Feb. 2012. DOI: [10.2139/ssrn.2003655](https://doi.org/10.2139/ssrn.2003655).
- [13] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini, “An empirical study of hyper-threading in high performance computing clusters,” pp. 6–11, Jan. 2002.
- [14] X. Yi, *A study of performance programming of cpu, gpu accelerated computers and simd architecture*, 2024. arXiv: [2409.10661](https://arxiv.org/abs/2409.10661) [cs.DC]. [Online]. Available: <https://arxiv.org/abs/2409.10661>.
- [15] M. Rosecrance. “The garbage collector,” Accessed: Jun. 5, 2025. [Online]. Available: <https://www.youtube.com/watch?v=gPxFOmuhnUU>.
- [16] J. Howarth. “Why discord is switching from go to rust,” Accessed: Mar. 15, 2025. [Online]. Available: <https://discord.com/blog/why-discord-is-switching-from-go-to-rust>.
- [17] J. Espino, *Understanding the go runtime*, 2023. Accessed: Jun. 7, 2025. [Online]. Available: <https://golab.io/talks/understanding-the-go-runtime>.
- [18] W. Kennedy. “Scheduling in go : Part ii - go scheduler,” Accessed: Jun. 7, 2025. [Online]. Available: <https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part2.html>.
- [19] T. Liu. “Memory allocations,” Accessed: Jun. 7, 2025. [Online]. Available: <https://go101.org/optimizations/0.3-memory-allocations.html>.
- [20] SoByte. “Principle of memory allocator implementation in go language,” Accessed: Jun. 8, 2025. [Online]. Available: <https://www.sobyte.net/post/2021-12/golang-memory-allocator/>.
- [21] S. Matsiukevich. “Golang internals, part 6: Bootstrapping and memory allocator initialization,” Accessed: Jun. 8, 2025. [Online]. Available: <https://www.altoros.com/blog/golang-internals-part-6-bootstrapping-and-memory-allocator-initialization/>.
- [22] M. Hertz and E. D. Berger, “Quantifying the performance of garbage collection vs. explicit memory management,” *SIGPLAN Not.*, vol. 40, no. 10, pp. 313–326, Oct. 2005. DOI: [10.1145/1103845.1094836](https://doi.org/10.1145/1103845.1094836). [Online]. Available: <https://doi.org/10.1145/1103845.1094836>.

- [23] P. S. Foundation. “Cpython internal documentation,” Accessed: Jun. 7, 2025. [Online]. Available: <https://github.com/python/cpython/tree/main/InternalDocs>.
- [24] P. S. Foundation. “Full grammar specification.” Part of the Python Language Reference for Python 3, Accessed: Jun. 7, 2025. [Online]. Available: <https://docs.python.org/3/reference/grammar.html>.
- [25] P. S. Foundation. “Gc - garbage collector interface,” Accessed: Jun. 7, 2025. [Online]. Available: <https://docs.python.org/3/library/gc.html>.
- [26] R. Pereira et al., “Energy efficiency across programming languages: How do energy, time, and memory relate?” In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017, Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 256–267. DOI: [10.1145/3136014.3136031](https://doi.org/10.1145/3136014.3136031). [Online]. Available: <https://doi.org/10.1145/3136014.3136031>.
- [27] N. van Kempen, H.-J. Kwon, D. T. Nguyen, and E. D. Berger, *It’s not easy being green: On the energy efficiency of programming languages*, 2025. DOI: [10.48550/arXiv.2410.05460](https://arxiv.org/abs/2410.05460). arXiv: [2410.05460 \[cs.PL\]](https://arxiv.org/abs/2410.05460). [Online]. Available: <https://arxiv.org/abs/2410.05460>.
- [28] R. Pereira et al., “Ranking programming languages by energy efficiency,” *Science of Computer Programming*, vol. 205, p. 102 609, 2021. DOI: <https://doi.org/10.1016/j.scico.2021.102609>. [Online]. Available: <https://www.science-direct.com/science/article/pii/S0167642321000022>.
- [29] S. Abdulsalam, D. Lakowski, Q. Gu, T. Jin, and Z. Zong, “Program energy efficiency: The impact of language, compiler and implementation choices,” in *International Green Computing Conference*, 2014, pp. 1–6. DOI: [10.1109/IGCC.2014.7039169](https://doi.org/10.1109/IGCC.2014.7039169).
- [30] D. Lion, A. Chiu, M. Stumm, and D. Yuan, “Investigating managed language runtime performance: Why JavaScript and python are 8x and 29x slower than c++, yet java and go can be faster?” In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 835–852. [Online]. Available: <https://www.usenix.org/conference/atc22/presentation/lion>.
- [31] N. Humrich. “Yes, python is slow, and i don’t care,” Accessed: Mar. 15, 2025. [Online]. Available: <https://medium.com/pyslackers/yes-python-is-slow-and-i-dont-care-13763980b5a1>.
- [32] J. Z. Søndergaard, M. C. B. Nielsen, S. A. B. Jensen, and V. F. J. and, “Measuring energy efficiency of jit compiled programming languages with micro-benchmarks,” Aalborg University, Tech. Rep., 2025.

- [33] IEEE, “Iso/iec/ieee international standard - systems and software engineering – life cycle processes – requirements engineering,” *ISO/IEC/IEEE 29148:2018(E)*, pp. 1–104, 2018. DOI: [10.1109/IEEESTD.2018.8559686](https://doi.org/10.1109/IEEESTD.2018.8559686).
- [34] N. Batchelder, *Facts and myths about names and values*, 2015. Accessed: Jun. 16, 2025. [Online]. Available: <https://nedbatchelder.com/text/names1.html>.
- [35] LLVM. “Fast math flags,” Accessed: Jun. 16, 2025. [Online]. Available: <https://llvm.org/devmtg/2024-10/slides/techtalk/Kaylor-Towards-Useful-Fast-Math.pdf>.
- [36] U. Nations. “Sustainable development goals,” Accessed: Jul. 14, 2025. [Online]. Available: <https://sdgs.un.org/goals>.
- [37] E. Union. “General data protection regulation (gdpr),” Accessed: Jun. 26, 2025. [Online]. Available: <https://gdpr.eu/tag/gdpr/>.
- [38] O. S. Initiative. “The 3-clause bsd license,” Accessed: Jun. 26, 2025. [Online]. Available: <https://opensource.org/license/BSD-3-Clause>.
- [39] LLVM. “Llvm license (legacy),” Accessed: Jun. 26, 2025. [Online]. Available: <https://llvm.org/docs/DeveloperPolicy.html#legacy>.
- [40] LLVM. “Llvm license (new),” Accessed: Jun. 26, 2025. [Online]. Available: <https://llvm.org/LICENSE.txt>.
- [41] JetBrains. “Toolbox subscription agreement for students and teachers,” Accessed: Jun. 26, 2025. [Online]. Available: https://www.jetbrains.com/legal/docs/toolbox/license_educational.
- [42] P. S. Foundation. “History and license,” Accessed: Jun. 26, 2025. [Online]. Available: <https://docs.python.org/3/license.html>.
- [43] Google. “License,” Accessed: Jun. 26, 2025. [Online]. Available: <https://go.dev/LICENSE>.
- [44] ISO/IEC, “Iso/iec 14882:2024 programming languages,” Tech. Rep., 2024. Accessed: Jun. 26, 2025. [Online]. Available: <https://www.iso.org/standard/83626.html>.
- [45] ISO/IEC, “Iso/iec 14882:2020(e) - programming languages, environments and system software interfaces,” Tech. Rep., 2020. Accessed: Jun. 26, 2025. [Online]. Available: <https://www.iso.org/standard/79358.html>.
- [46] O. S. Initiative. “Mit license,” Accessed: Jun. 26, 2025. [Online]. Available: <https://opensource.org/license/mit/>.
- [47] N. O. S. Team. “Numpy,” Accessed: Jun. 26, 2025. [Online]. Available: <https://numpy.org/>.
- [48] Apple. “Rosetta 2,” Accessed: Jul. 25, 2025. [Online]. Available: <https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment>.

- [49] A. W. Services. “Aws lambda,” Accessed: Jul. 25, 2025. [Online]. Available: <https://aws.amazon.com/lambda/>.

APPENDIX A

SERVER BENCHMARKS TABLES

RESULTS

This section provides the reader with the detailed results of the server benchmarks:

TABLE A.1

ENERGY USAGE BY THE PACKAGE ON THE SERVER BENCHMARK BY IMPLEMENTATION AND
CORE COUNT

energy-pkg	C++	Go	PyPy	Python
1	3,756.26	28,522.69	7,972.65	1,510,534.76
2	2,591.91	18,231.97	5,147.60	1,141,490.15
4	1,362.61	10,304.27	2,828.21	313,537.85
8	799.23	5,617.27	1,747.96	194,458.98
14	603.37	3,155.30	1,232.03	130,506.94
16	627.16	2,904.52	1,140.80	111,438.01
28	1,830.15	2,306.35	1,252.93	122,384.40
28 same CPU	2,116.22	2,271.71	1,175.52	96,049.86
32	2,043.31	2,151.74	1,257.81	120,259.78
32 same CPU	1,889.55	2,109.37	1,155.74	87,343.61
48	1,985.14	1,856.93	1,354.03	91,185.55
60	1,975.05	1,744.76	1,354.03	93,015.31

TABLE A.2

ENERGY USAGE BY THE RAM ON THE SERVER BENCHMARK BY IMPLEMENTATION AND CORE COUNT

energy-ram	C++	Go	PyPy	Python
1	176.83	1,049.92	355.42	59,269.48
2	111.08	755.43	253.06	36,442.43
4	67.60	426.69	170.08	12,799.44
8	26.28	175.15	88.94	6,683.08
14	18.28	107.26	72.28	4,031.07
16	19.29	95.60	71.84	3,705.32
28	55.09	55.07	81.44	4,057.46
28 same CPU	78.01	83.98	78.81	2,831.52
32	61.70	66.51	83.50	4,037.98
32 same CPU	65.66	66.54	77.79	2,576.33
48	85.73	70.64	89.06	2,657.54
60	90.01	90.23	91.31	2,703.02

TABLE A.3

EXECUTION TIME ON THE SERVER BENCHMARK BY IMPLEMENTATION AND CORE COUNT

time	C++	Go	PyPy	Python
1	23.70	172.952	46.58	5,913.41
2	12.52	89.32	26.43	4,749.55
4	6.69	45.51	13.97	1,665.41
8	3.51	23.78	7.66	872.89
14	2.39	14.03	5.15	528.12
16	2.55	12.46	4.88	482.30
28	11.63	8.91	5.39	529.22
28 same CPU	13.82	7.61	3.77	269.13
32	8.39	8.16	5.44	523.04
32 same CPU	11.85	6.82	3.58	244.57
48	15.46	6.82	4.03	252.17
60	15.94	5.36	4.05	256.87

APPENDIX B

DESKTOP TABLES RESULTS

This section provides the reader with the detailed results of the personal desktop benchmarks, including energy consumption, RAM power consumption and execution times for the benchmarks ran on a personal desktop with different core counts and programming languages.

TABLE B.1

ENERGY USAGE BY THE PACKAGE ON THE DESKTOP BENCHMARK BY IMPLEMENTATION
AND CORE COUNT WITH HYPERTHREADING ENABLED

energy-pkg	C++	Go	PyPy	Python
1	991.49	5,867.26	1,621.77	272,235.58
2	657.79	3,746.84	1,324.05	174,822.21
4	442.17	2,335.54	938.02	112,247.92
8	354.57	1,583.17	747.41	82,983.52
12	339.44	1,324.22	842.68	90,069.44
14	333.01	1,247.80	889.03	93,771.09
16	321.60	1,194.03	937.02	98,436.51

TABLE B.2

ENERGY USAGE BY THE PACKAGE ON THE DESKTOP BENCHMARK BY IMPLEMENTATION
AND CORE COUNT

energy-pkg	C++	Go	PyPy	Python
1	1,008.55	5,945.33	1,635.95	
2	671.13	3,652.85	1,296.42	176,092.08
4	437.34	2,421.40	914.60	113,582.17
6	388.96	1,878.72	789.89	95,214.00
8	346.15	1,645.18	735.35	82,869.55

TABLE B.3

EXECUTION TIME ON THE DESKTOP BENCHMARK BY IMPLEMENTATION AND CORE COUNT
WITH HYPERTHREADING ENABLED

execution-time	C++	Go	PyPy	Python
1	22.78	178.98	37.12	6,063.96
2	11.71	70.51	21.62	3,084.07
4	6.23	36.79	12.28	1,600.72
8	3.45	19.18	6.85	825.24
12	3.20	14.33	6.95	848.99
14	3.09	12.97	7.01	866.47
16	2.97	12.09	7.21	878.90

TABLE B.4

EXECUTION TIME ON THE DESKTOP BENCHMARK BY IMPLEMENTATION AND CORE COUNT

execution-time	C++	Go	PyPy	Python
1	22.60	178.43	36.42	
2	11.64	69.77	21.59	3,115.45
4	6.20	36.68	11.97	1,634.00
6	4.49	25.24	8.35	1,104.04
8	3.43	20.84	6.91	829.50

APPENDIX C

LAPTOP BENCHMARKS TABLES RESULTS

This section provides the reader with the detailed results of the laptop benchmarks, including energy consumption and execution times for the benchmarks ran on a MacBook Pro with different core counts and programming languages.

TABLE C.1

ENERGY USAGE BY THE PACKAGE ON THE LAPTOP BENCHMARK BY IMPLEMENTATION AND CORE COUNT

Cores	C++	Go	PyPy 3.11.11	Python
1	58.79	429.23	194.69	17,546.05
2	68.93	370.01	211.21	16,322.57
4	54.97	354.64	216.53	16,193.99
8	55.94	348.14	216.60	15,509.12
10	55.00	336.88	214.96	14,942.09
14	48.39	300.66	203.92	13,777.34

TABLE C.2

EXECUTION TIME ON THE LAPTOP BENCHMARK BY IMPLEMENTATION AND CORE COUNT

Cores	C++	Go	PyPy 3.11.11	Python
1	7.48	54.16	26.52	2,697.08
2	3.63	26.76	15.55	1,356.76
4	2.06	14.80	8.25	696.75
8	1.37	7.94	4.57	350.76
10	1.26	6.66	3.81	288.37
14	1.48	5.94	3.62	262.00

APPENDIX D

RASPBERRY PI BENCHMARKS TABLES RESULTS

This section provides the reader with the detailed results of the Raspberry Pi benchmarks, including energy consumption and execution times for the benchmarks ran on a Raspberry Pi 5 with different core counts and programming languages.

TABLE D.1

ENERGY USAGE BY THE PACKAGE ON THE RASPBERRY PI BENCHMARK BY
IMPLEMENTATION AND CORE COUNT

Cores	C++	Go	PyPy	Python
1	80.00	236.67	190.00	11 780.00
2	51.67	153.33	152.33	7 620.67
4	41.28	107.20	181.60	5 739.00

TABLE D.2

EXECUTION TIME ON THE RASPBERRY PI BENCHMARK BY IMPLEMENTATION AND CORE
COUNT

Cores	C++	Go	PyPy	Python
1	49.26	165.03	126.36	8 853.08
2	25.25	81.98	74.42	4 423.91
4	12.80	41.10	72.14	2 264.50

APPENDIX E

GOROUTINES BENCHMARKS

TABLE E.1

GO GOROUTINES AND CORES USED, * MEANS THE EXECUTION IS FIXED TO ONE CPU

Cores	Goroutines	Energy (J)	Relative Energy	Execution time (s)	Relative Time
1	1	28,522.69	1x	172.952	1x
1	2	28,919.31	0.99x	175.381	0.99x
2	2	18,231.97	1.56x	89.319	1.94x
2	4	18,224.50	1.57x	89.275	1.94x
4	4	10,304.27	2.77x	45.508	3.8x
4	8	10,299.06	2.77x	45.482	3.8x
8	8	5,617.27	5.08x	23.781	7.27x
8	16	5,580.52	5.11x	23.594	7.33x
14	14	3,155.30	9.04x	14.034	12.32x
14	28	3,151.93	9.05x	14.001	12.35x
16	16	2,904.52	9.82x	12.456	13.88x
16	32	3,018.54	9.45x	12.435	13.91x
28	28	2,306.35	12.37x	8.906	19.42x
28 *	28	2,271.71	12.56x	7.613	22.72x
28	56	2,314.29	12.32x	7.791	22.2x
28 *	56	2,290.85	12.45x	8.815	19.62x
32	32	2,151.74	13.26x	8.163	21.19x
32 *	32	2,109.37	13.52x	6.822	25.35x
32	64	2,121.88	13.44x	8.101	21.35x
32 *	64	2,142.21	13.31x	6.896	25.08x
48	48	1,856.93	15.36x	5.718	30.24x
48	96	1,848.47	15.43x	5.673	30.48x
60	60	1,744.76	16.35x	5.357	32.28x
60	120	1,737.80	16.41x	5.320	32.5x
60	200	1,724.73	16.54x	5.255	32.91x
60	250	1,719.49	16.59x	5.218	33.14x

APPENDIX F

COMPILER OPTIMIZATIONS TABLES RESULTS

TABLE F.1

POWER/ENERGY AND EXECUTION TIME FOR VARIOUS CORE COUNTS AND COMPILER
OPTIMIZATION FLAGS

Metric / Flags	-O0	-O1	-O2	-O3	-O3 (fast-math)
1 core					
power/energy-pkg (J)	52,957.47	5,185.28	4,388.12	4,031.83	4,005.70
power/energy-ram (J)	2,208.08	210.28	183.21	180.00	168.52
time (s)	270.06	27.453	24.02	23.5993	22.054
4 cores					
power/energy-pkg (J)	13,379.59	1,564.51	1,417.01	1,386.60	1,414.08
power/energy-ram (J)	508.40	56.72	50.96	50.37	49.56
time (s)	66.066	7.4351	6.67	6.5605	6.478
14 cores					
power/energy-pkg (J)	5,056.15	658.46	606.58	595.29	639.00
power/energy-ram (J)	153.03	20.79	18.53	18.24	20.10
time (s)	19.9344	2.7184	2.42	2.3807	2.62
28 cores					
power/energy-pkg (J)	3,650.43	627.89	565.83	561.59	547.56
power/energy-ram (J)	120.49	20.00	18.00	17.93	17.47
time (s)	15.729	2.6043	2.36	2.3382	2.2701
32 cores					
power/energy-pkg (J)	3,353.53	606.52	552.98	548.83	535.22
power/energy-ram (J)	110.47	19.54	17.76	17.58	17.27
time (s)	14.4472	2.5369	2.33	2.3015	2.2368
60 cores					
power/energy-pkg (J)	2,794.03	698.96	661.76	659.31	671.51
power/energy-ram (J)	86.80	25.37	24.11	24.10	24.76
time (s)	7.5463	1.89955	1.80	1.79288	1.79288

APPENDIX G

NUMBER OF BENCHMARKS FOR AVERAGE CALCULATION

The number of benchmarks that are run for each of the programs has to be 5 as if it is run with less iterations, the results vary more than 0.5% between runs. I chose 5 times, as it can be seen in Figure G.1, that after 5 iterations, the results start to stabilize. This is a good compromise between the time it takes to run the benchmarks and the stability of the results.

It can be observed from this chart that, by using the different number of benchmarks to average from, the results can vary significantly.

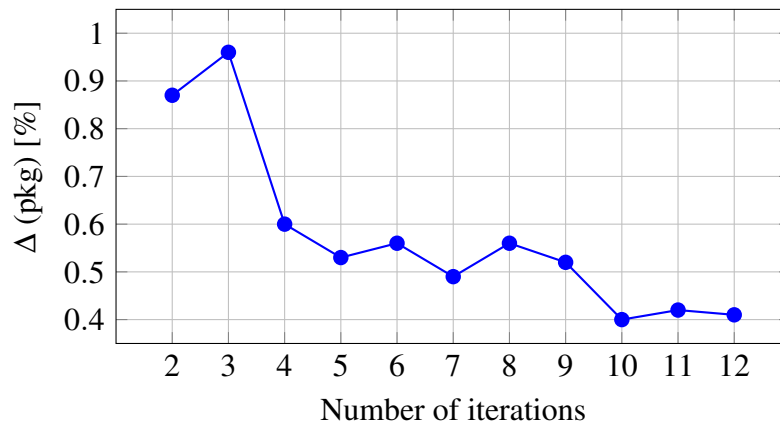


Fig. G.1. Variation of results with respect to the number of benchmark iterations

Number of iterations	Δ (pkg)
2	0.87%
3	0.96%
4	0.60%
5	0.50%
6	0.56%
7	0.49%
8	0.56%
9	0.52%
10	0.40%
11	0.42%
12	0.41%

Fig. G.2. Number of benchmarks for average calculation

APPENDIX H

SCATTER PLOT WITH PYTHON

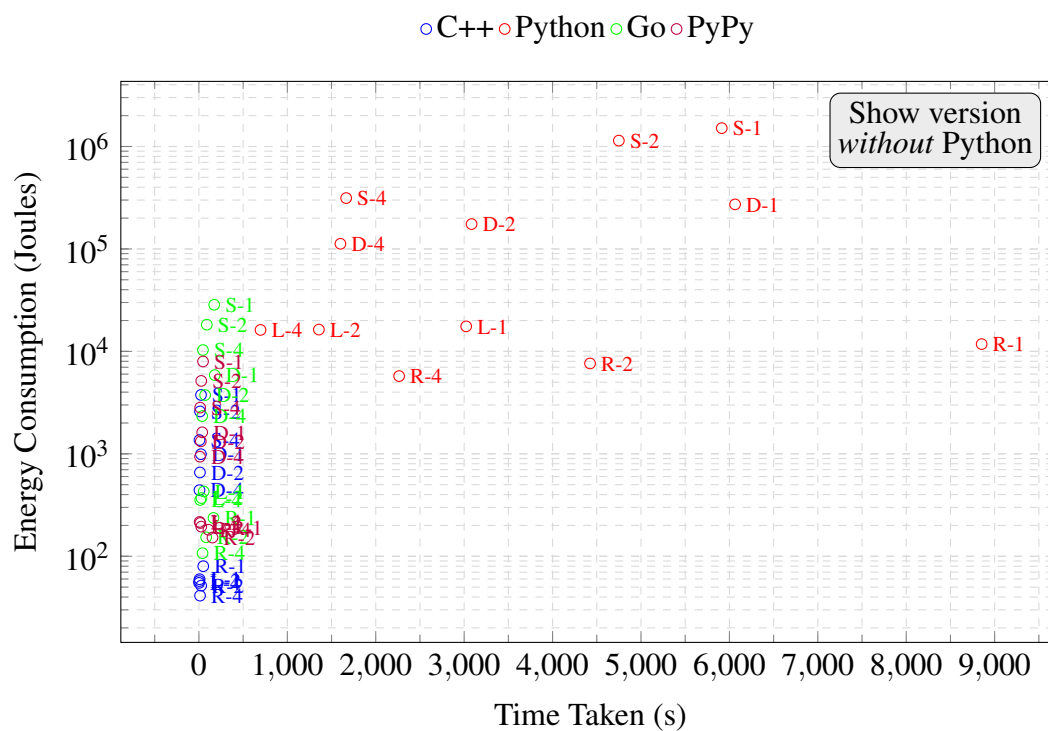


Fig. H.1. All languages. Colors = language; labels = platform-cores.

APPENDIX I

DECLARATION OF USE OF GENERATIVE AI

I have used Generative AI in this work ☐ YES

Part 1: Reflection on ethical and responsible behavior

Keep in mind that the use of Generative AI entails risks and can have consequences that affect the moral integrity of your actions with it. Therefore, we ask you to answer the following questions honestly (mark what applies):

1. In my interaction with Generative AI tools, I have submitted sensitive data with the proper authorization of the interested parties.

☐ NO, I have not used sensitive data

2. I have submitted copyrighted materials with the proper authorization.

☐ NO, I have not used protected materials

3. I have submitted personal data with the proper authorization of the interested parties.

☐ NO, I have not used personal data

4. My use of the tool has respected the terms of use and ethical principles.

☐ YES

Part 2: Technical Use Declaration

Documentation and writing

I have used Generative AI for:

- Search of bibliography to [Google Scholar](#), [GPT o3](#), [o4](#).
- Generation of tables based on provided data.

Development of specific content

- Programming assistance.

I have asked for explanations of Python, Go and C++ code to [GPT 4o Copilot](#), [Claude \(3.5, 3.7, 4\)](#), [Gemini 2.5 Pro](#) and [Qwen 2.5](#).

- Generation of elements.

I have asked for generation of diagrams and tables based on benchmark data to [OpenAI's GPT 4.1, 4o, o4](#), [Gemini 2.5 Pro](#), [Qwen3](#) and [Claude \(3.5, 4\)](#).

- Read proofing and correction of texts.

I have asked for corrections of the text in the appendix and the main body of the work to [Gemini 2.5 Pro](#).

Part 3: Reflection on Utility

Please provide a personal assessment of the strengths and weaknesses you have identified in the use of Generative AI tools. Comment on whether it has helped you in the process of learning, development, or conclusions of the work.

The use of Generative AI tools has been beneficial in several ways. It has allowed me to quickly generate tables and diagrams based on benchmark data, which would have taken significantly more time to create manually and style consistently. Additionally, the ability to ask for explanations of code and receive immediate feedback has been invaluable in using new programming techniques.

I have not used AI to debug code as I have found out that by using AI to debug errors, it can lead to a lack of understanding of the underlying issues, and it can lead to an erroneous fix.

In the process, I have learned also how LLMs work and what is the best way to use them, never accepting changes blindly, always checking the output.