Universidade de Aveiro

DETI

Complements in Computer Architecture

Final Assignment

Performance evaluation across multiple languages

2023-2024

Teacher:

Professor Tomás Oliveira e Silva

Students:

Pedro Ramos
p.ramos@ua.pt
107348

Rodrigo Aguiar
rodrigoaguiar96@ua.pt
108969

João Andrade
joao.andrade06@ua.pt
107969

Pedro Pinho
pd.pinho@ua.pt
109986

Rafael Ferreira
rafaelcferreira@ua.pt
107340

Inês Ferreira
inesferreira02@ua.pt
104415

André Fernandes
andre.sou.fernandes@ua.pt
103252

Contents

Introduction

This course aims to make us better developers, by introducing the notion that our code has to run on hardware, which has specifications that must be taken in account when developing programs or applications.

We developed three different projects for this course, each with a given purpose, but all exploring the idea that we must take into account the hardware of the computer itself in order to fully optimize its design, functionality and performance.

This report is a compilation of all of the work done for each of these three projects by all of the students in the course and aims to provide a context, development report, testing phase, analysis and conclusion to each of the projects in question.

Finally, we wish to provide more useful code, test results and observations to anyone that is interested in bottom level of programming (closer to the hardware itself) and may not know how to best utilize the tools at hand.

We also provide this code and test results at our repository, located in Github (https://github.com/P-Ramos16/CSAC).

Project 1: Sum-Free Sets

1.1 Context

For the first project, a toy problem called "Sum-Free Sets"was given, along with a set of programs to solve the problem written in the C programming language.

The Sum-Free Sets problem consists of calculating all the sum-free sets' subsets 1,2,...,n. A set is sum-free when the sum of any two elements of the set does not belong in the set, for example, 1, 3, 5 is a sum-free set but 2, 3, 5 is not, since 2+3=5. In general, a set is not sum-free whenever elements a[i]+a[j]=a[k] for $i <= j$. This means that even the set 1, 2 is not sum-free, as $1 + 1 = 2$.

This sequence can be further explored on the On-Line Encyclopedia of Integer Sequences (OEIS for short) (https://oeis.org/) as sequence 007865 (https://oeis.org/A007865)

The objective of the project is to compare the execution times between the given algorithms and between different programming languages, such as Python, Java, Bash, etc.

1.2 Goals

As explained above, the main goal of this project is to compare the efficiency and speed of the different algorithms and programming languages when they are used to solve 'big number calculations' using heavily recursive algorithms.

So for the comparisons, the following variables were changed:

- The algorithm used (always written in C and using one thread with the -O2 compilation flag);
- The number of CPU threads used by the algorithm;
- The optimization compilation flags (more specifically, the optimizations of the single-threaded v1 algorithm written in C);
- The CPU used to execute the calculations on;
- The programming language used (using the single-threaded v1 algorithm).

All the different combinations were tested in multiple machines with different processors, including:

- AMD Ryzen 7 5700U and 5 5600x
- Intel Core i7 11370H, i7 9750H, i5 1135g7 and i5 9400f
- BCM2711 with 4x Cortex-A72 cores from a Raspberry Pi 4 (4Gb)

1.3 Algorithms

For the first 'variable' to test, we started by comparing the differences between the provided algorithms.

All of the results used on these tests were executed on the same CPU (AMD Ryzen 7 5700U), with the -O2 compiler optimization flag and all the algorithms were written in the C programming language.

We chose to run the calculations for a maximum n value of 60 since it does not require more than a few minutes of calculations and the value is high enough to make the difference in the results very apparent.

1.3.1 V1

The first version of this algorithm is quite simple, it uses a recursive function to try and append every number from the first number to try up to the maximum number to try.

If a number can be added, the global 'sum count' array is updated to reflect the new number and the function recursively calls itself to try adding the next number.

After returning the recursion, the function backtracks the changes to 'sum count'.

The results were as follows:

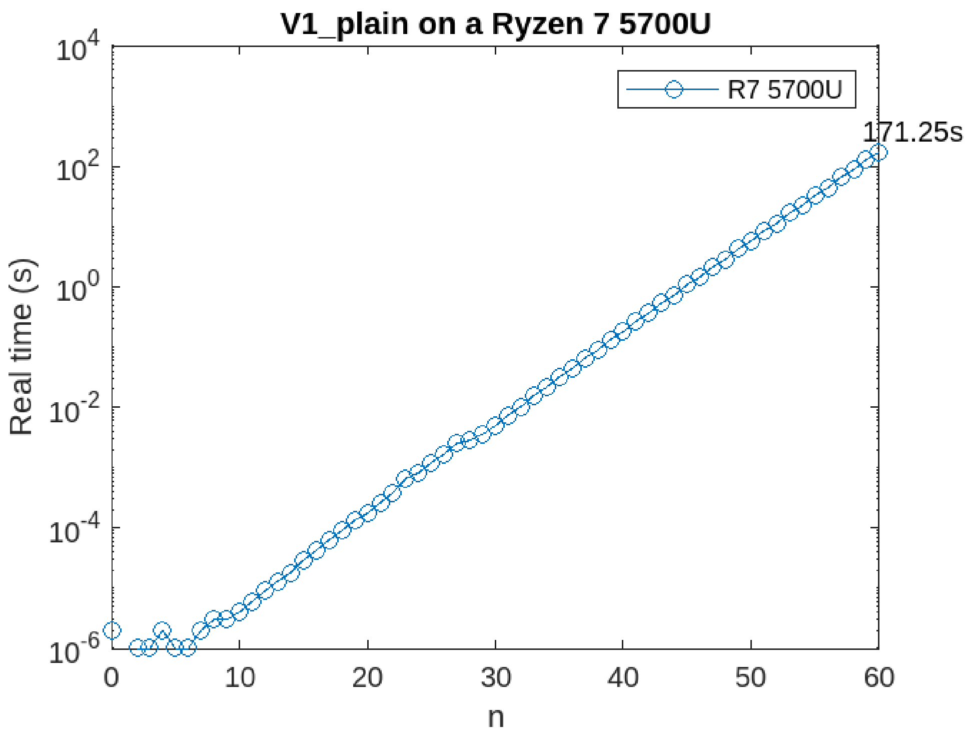


Figura 1: V1 algorithm

The final user time for the n of 60 was 171.25 seconds

1.3.2 V2

The second revision of this algorithm leverages bit manipulation to count the number of sum-free subsets more efficiently.

The processing of the subsets created utilizes bitwise algebra to improve the performance of the final calculations.

Although the algorithm's performance is greatly improved, because it remains on an $O(n^2)$ complexity, the usability for large N values is still limited.

To explain this further, although the V2 algorithm is almost 7.5 times faster than the V1 algorithm (for an N of 60), given the same amount of time, the V1 would reach, for example, a N value of 60 but the V2 would only reach a N of about 62.

So, if the objective is to calculate a specific N value, the V2 algorithm would be much faster, but if the goal is to obtain the calculate to the largest N value possible, the V2 algorithm would only calculate a few iterations more.

The results of the tests were as follows:

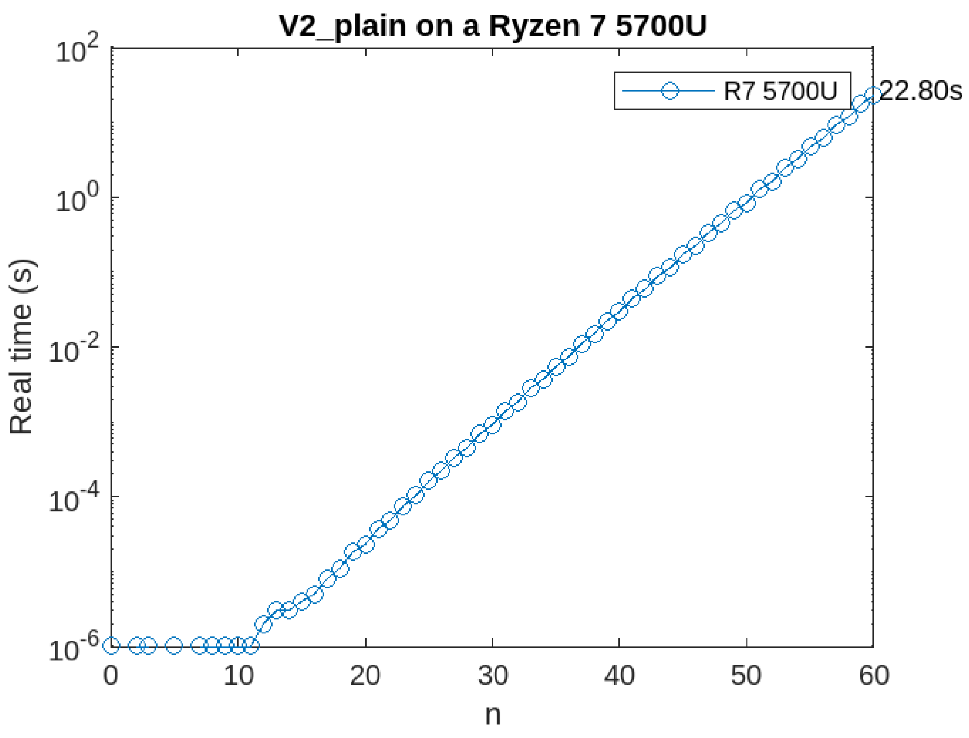


Figura 2: V2 algorithm

The final user time for the n of 60 was 22.80 seconds

1.3.3 V3

For the third version of the algorithm, bit masks were used to determine the first position in the current array of sums that do not have a sum (i.e., a position where we can add a new element without violating the sum-free property).

After finding this position, the algorithm updates the current sum state and continues to the next position.

When no valid new elements can be added, the algorithm backtracks by removing the previous state from the stack and continuing the loop.

The performance of this algorithm is similar but slightly worse, to the performance of the V2 algorithm.

This means that this version still obtains a complexity of O(n^2), invoking the previously referenced problems with the V2 algorithm, in that it could not reach a much higher N value in the same amount of time as the initial algorithm, despite being over 5.7 times faster.

The results of the tests were as follows:

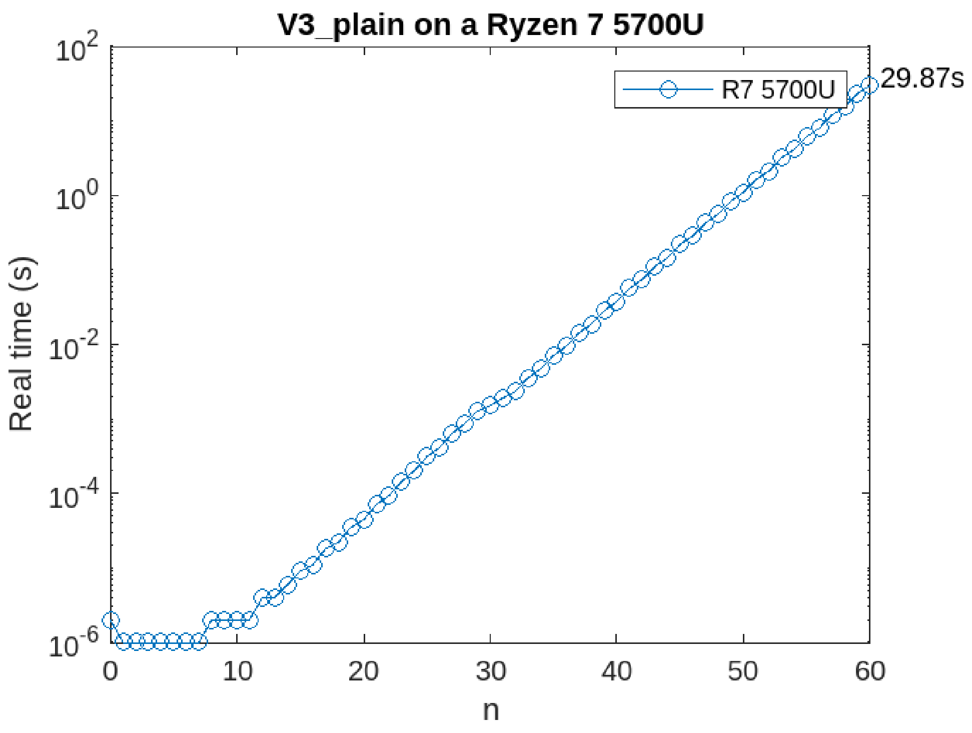


Figura 3: V3 algorithm

The final user time for the n of 60 was 29.87 seconds

1.4 Multiple Threads

For the next ”variable” to test, we chose to analyze the effects that multiple threads have on the execution time of the program.

The test parameters set were as follows:

- The algorithm used is the V1;
- The code is implemented in the C programming language;
- The optimization compilation flag used is the -O2;
- The N value calculated is of 60;
- The CPU used is the AMD Ryzen 7 5700U.

It is important to note that this CPU has 8 cores with a total of 16 threads, with a maximum frequency of 4.372GHz. The machine used also has 32Gb of 3200MHz DDR4 memory.

The results for each number of threads averaged as follows:

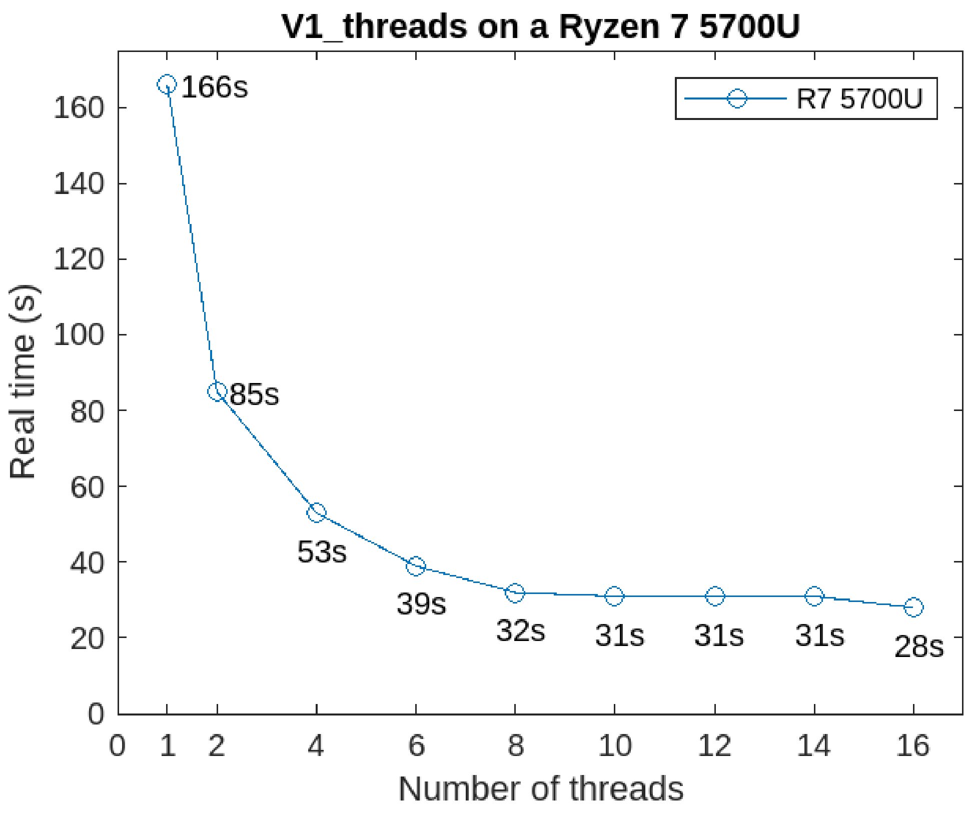


Figura 4: V1 algorithm using multiple threads

From this graph, we can deduce that the number of threads used greatly impacts the total real-time (or user time) needed to complete the execution of the program until all the sum-free sets for the total N value are found.

We can also conclude that the point of diminishing returns is located somewhere near the total number of CPU cores, and not the number of maximum CPU threads.

This means that, although the number of threads used to execute the calculations doubled, the total time to run the algorithm barely changed.

This effect is caused by multiple attributes, the most significant of which is the competition for CPU cache and resources shared between the threads. Since the extra threads provided by the multi-threading properties of this CPU are using the same cache as the "real" cores, both access the lower layers of the cache at the same time, filling this level quicker and reducing access time for both, leading to the conclusion that this algorithm's performance is not only bound to the number of threads used but to the specifications of the CPU cache itself, since it requires the persistence of a lot of results from numerical operations.

The thermal limit of the CPU and the cooling system must also be considered as a limitation of running the algorithm on multiple threads since they can cause the threads themselves to "thermal throttle", which will reduce the overall performance of each core to avoid damage caused by excessive heat. This is usually done by reducing the maximum frequency of each CPU core. In this test, it was found that the frequency of the working threads would drop from 4.4GHz (in single threading) down to 3.3GHz (for 16 threads).

Despite this, the usage of simulated cores still increased the overall performance by a small margin but made the results more inconsistent.

To test our hypothesis further, we executed the same test on multiple other CPUs.

These CPUs have many differences from the ones used in the previous tests, most notably, we have:

- Intel Core i7 9750h, with 6 cores and 12 threads, to test a processor from Intel;
- Intel Core i5 9400f, with 6 cores and 6 threads, to test a processor without multi-threading;
- BCM2711 from a Raspberry Pi 4 (4Gb), with 4 cores and 4 threads, to test an ARM processor.

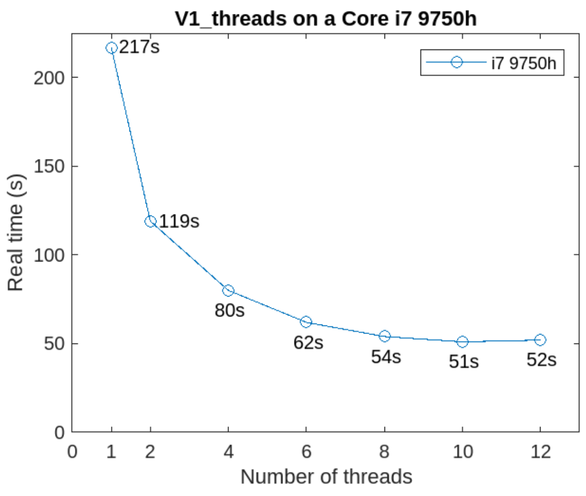


Figura 5: V1 algorithm using multiple threads on a Intel Core i7 9750h with 6 cores and 12 threads

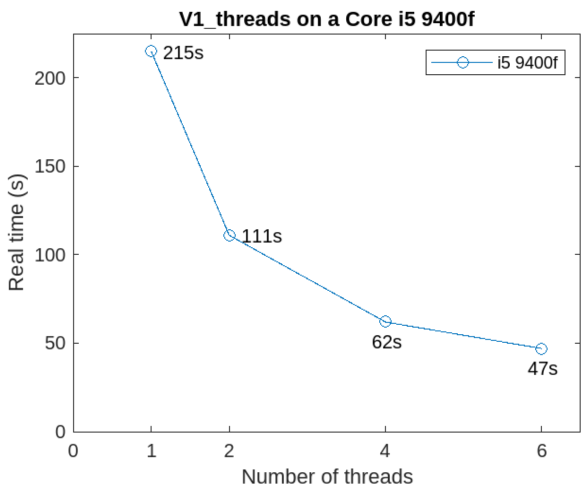


Figura 6: V1 algorithm using multiple threads on a Intel Core i5 9400f with 6 cores and 6 threads

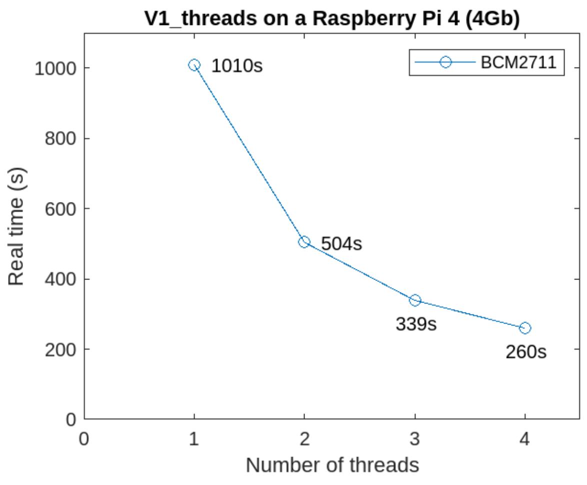


Figura 7: V1 algorithm using multiple threads on a BCM2711 from a Raspberry Pi 4 4Gb with 4 cores and 4 threads

From these results we conclude that the performance of the algorithm grows linearly depending on the number of threads used until the simulated multi-threaded cores are used

In Figure 5 we can see that the Intel processor has a similar curve to the AMD processor used before, the performance increased significantly until the maximum number of "real" cores is reached, and after that only minor performance differences are noted, so the CPU manufacture is not a limiting factor of the performance growth.

From Figure 6 we see that a processor without multi-threading also has a similar initial curve to the others, but without noticing the slowing of performance gains once the number of available "real" cores ends, since there are no more simulated cores to use. This further shows that the diminishing of the performance gains comes from the switch from "real" cores to simulated ones.

Finally, we can see in Figure 7 that even a comparatively slow ARM processor also provides a similar curve to the other CPUs, proving that the architecture itself is not a limiting factor of performance growth. The values obtained appear to be almost identical to Figure 6 but multiplied by a factor of 6 (six times slower).

Concluding this section, we can affirm that to achieve the maximum performance with this specific algorithm (V1), cache size and access speed, as well as the actual number of physical processor cores should be the highest priority in terms of hardware since core virtualization and excessive multi-threading will not significantly improve performance.

1.5 Compilation Options

The next variable to test is the compilation options provided by the compiler used, in this case, the "cc" compiler.

This option, represented by the flag -O, indicates the level of code optimization to perform on the program's compilation, allowing for some of the code itself to be altered to utilize known optimization techniques that might be too complex to implement manually in every codebase.

As per the GNU GCC manual, some drawbacks are introduced when using the flag: "Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results." [https://gcc.gnu.org/onlinedocs/gc
Options.html].

The chosen optimization levels determine if the compiler runs the before-mentioned optimization algorithms and how aggressive these will be. The levels analysed in this project are:

- Level 0: No optimization done by the compiler;
- Level 1: Implement about half of the most common optimization techniques to obtain a faster binary at the cost of compilation time and compilation memory usage;
- Level 2: Implement the more severe optimization techniques, costing even more compilation time and memory;
- Level 3: Implement all the most aggressive optimization techniques, even if they cause the generation of much larger binary files and take longer to compile.

More levels to reduce other parameters such as compilation time, binary file sizes and better debugging experience are available but were deemed mostly irrelevant in our context.

1.5.1 No Optimization

When the binary is generated without any optimization, we can expect that the general processing time for a given value of N will be much larger since these optimizations usually induce large effects on the performance of a specifically compiled binary.

The total real-time for calculating the sum free sets, using the single-threaded V1 algorithm, a maximum N value of 60 and the previous AMD Ryzen 7 5700U CPU, the results were as follows:

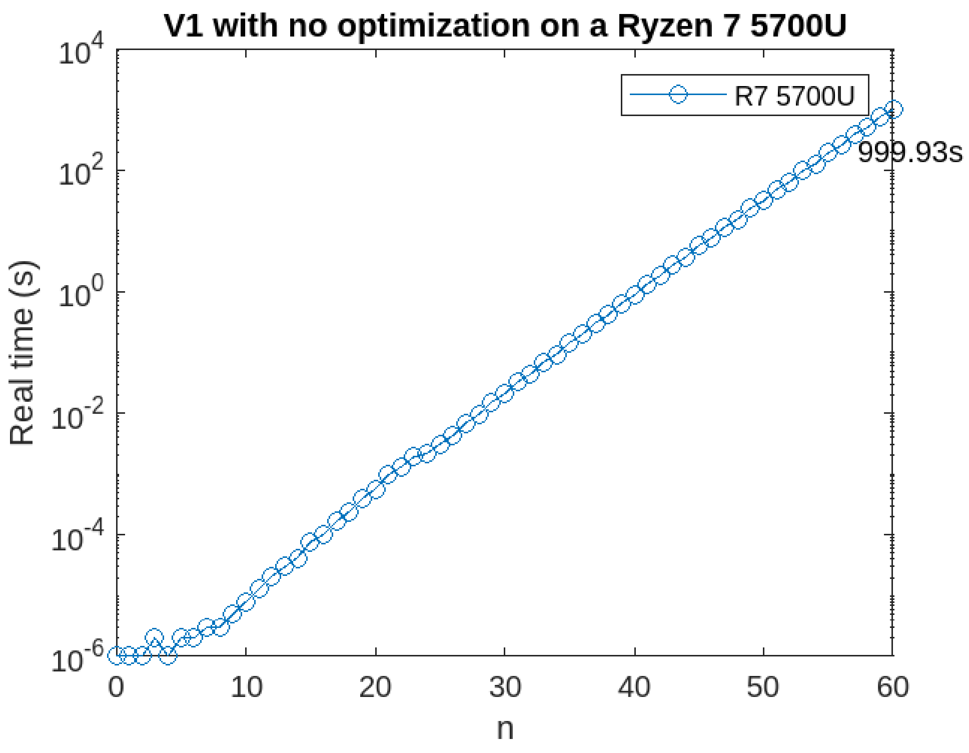


Figura 8: V1 algorithm compiled with no optimization

The final real-time for a N value of 60 was 999.93 seconds.

This result will be further analysed in the next section but we can already note that it is significantly slower than the tests performed in the previous sections of this report.

1.5.2 Optimization 1

When the first level of optimization techniques is used to compile the code, a massive performance gain is obtained, as noted by the results of the tests executed:

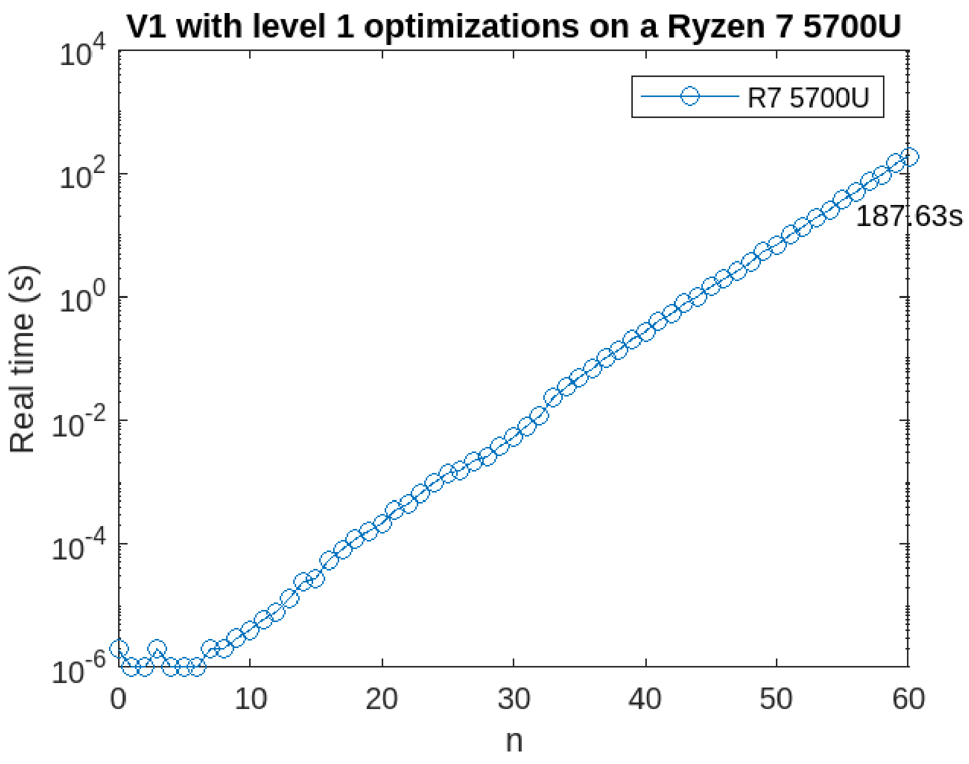


Figura 9: V1 algorithm compiled with level 1 optimizations

We can see that, for a N value of 60, the algorithm now performs about 5.3 times faster than the non-optimised version.

Despite these improvements, the final complexity of the algorithm does not change from the previous O(n^2), meaning that these optimizations are mostly focused on "static" changes such as improving memory utilization and processing, and that the algorithm's base implementation is unaltered.

1.5.3 Optimization 2

With the second level of optimisation, the performance gain is not as drastic as the gain from the non-optimised binary to level 1 of the optimization.

Despite this, the performance still improved by a factor of about 1.1 (10 percent improvement), which is noticeable in the results below:

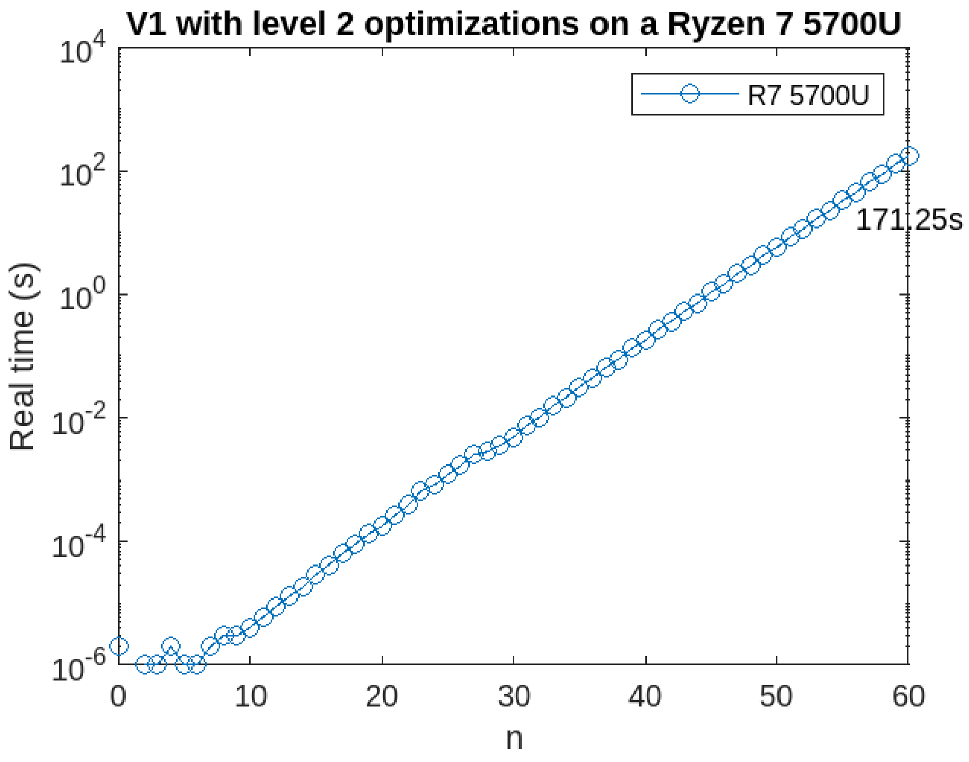


Figura 10: V1 algorithm compiled with level 2 optimizations

We can see that, for a N value of 60, the algorithm now performs about 5.8 times faster than the non-optimised version, and 1.1 times faster than the level 1 optimised version.

Once again, the complexity of the algorithm remains the same.

1.5.4 Optimization 3

Finally, we tested the third level of optimization.

Accounting for unexpected variables and system utilization, we can note that the overall performance remained the same, as shown by the results below:

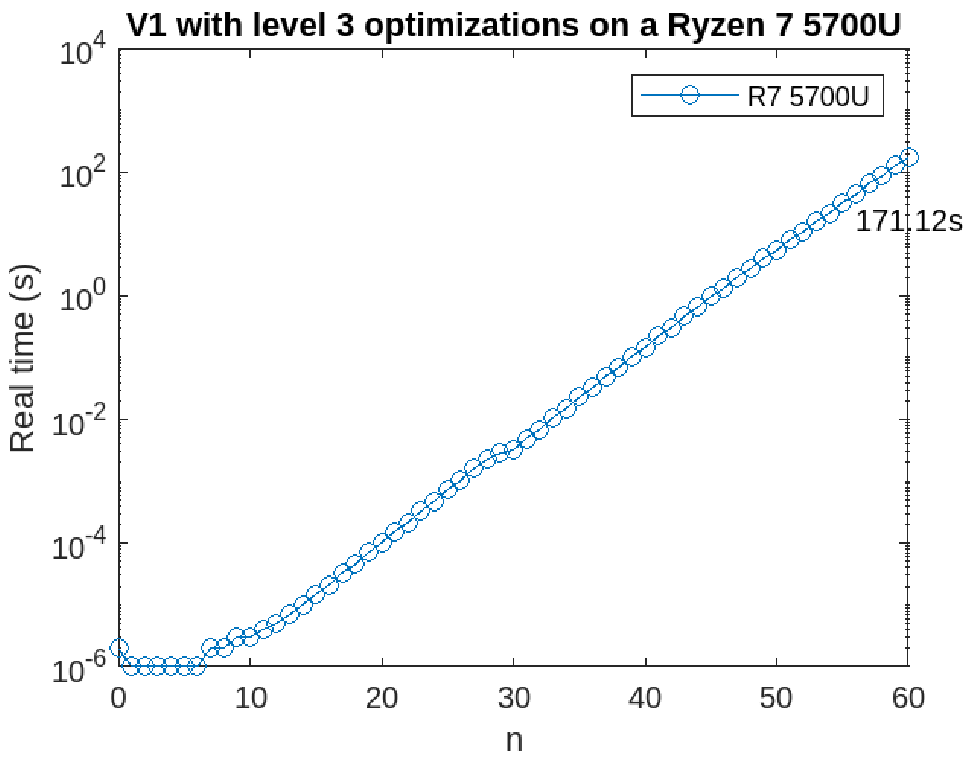


Figura 11: V1 algorithm compiled with level 3 optimizations

To note that the third level of optimization did result in a more consistent set of results, which varied less across the test runs executed, but this can happen due to other external factors.

This level of optimization did not create any noticeable difference in the final real-time results because this level is usually only applicable to larger and more specific algorithm techniques, so probably only a few of the optimizations that this level provides have been applied to the final binary, meaning that this level is very similar to the previous.

Concluding, this level does not provide a significant performance increase for the final binary since almost none of the improvements provided by this level can be applied.

1.5.5 Results

Comparing the four levels tested, we can see that the compiler optimization flags play a big role in the performance of the algorithm.

Although the level 3 flag did not provide any performance advantage compared to the previous level, the difference between the non-optimised binary and the level 2 optimized binary is massive, as can be seen in the next graphs:

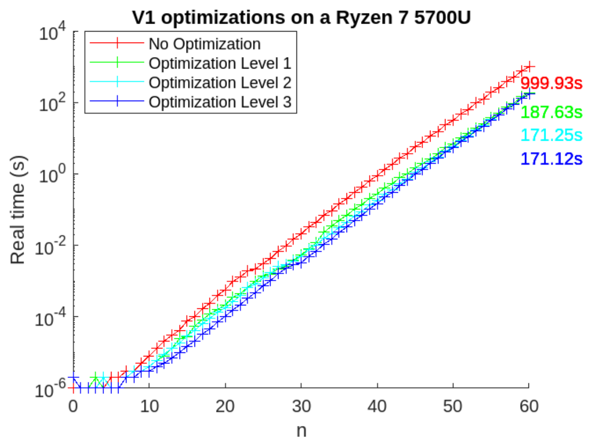


Figura 12: V1 algorithm tested using multiple optimization flags on an AMD Ryzen 7 5700U (logarithmic scale)

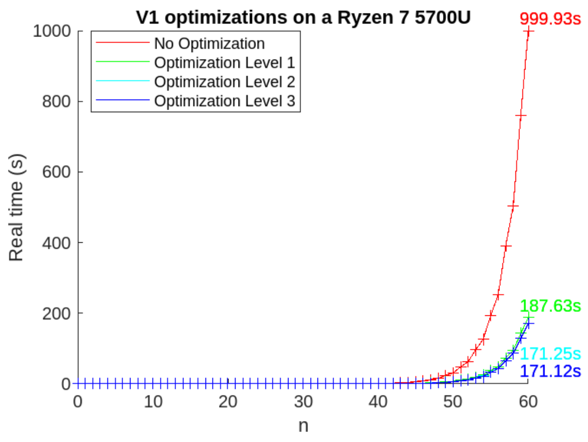


Figura 13: V1 algorithm tested using multiple optimization flags on an AMD Ryzen 7 5700U (linear scale)

In Figure 12, we can see that the overall algorithmic complexity of the code still remains at O(n^2), despite the usage of these optimization techniques, but from Figure 13 we can see that the total real-time used to compute the result of a specific value of N was reduced drastically, with the second level providing almost 5.8 times the performance of the non-optimised code.

In conclusion, we found that the second level of optimization is the most appropriate for the compilation of this specific implementation of the algorithm, providing a big performance improvement for a specific value of N.

1.6 Different CPUs

The next "variable" that we will test is the same code with the same algorithm, optimization level, number of threads and programming language, but executed on different processor units.

The parameters of the test include a final N value of 60, a compilation optimization flag level of 2, the V1 algorithm and a single-threaded configuration.

The usage of multi-threading by different processors would also be interesting to test, since, for example, most of Intel's processors focus on better single-core performance while AMD manufactured processors usually focus on better multi-core performance. But since these tests were already performed on the previous "Multiple Threads" section of this report, we only tested single-core performance on this section.

After executing the tests, these were the results:

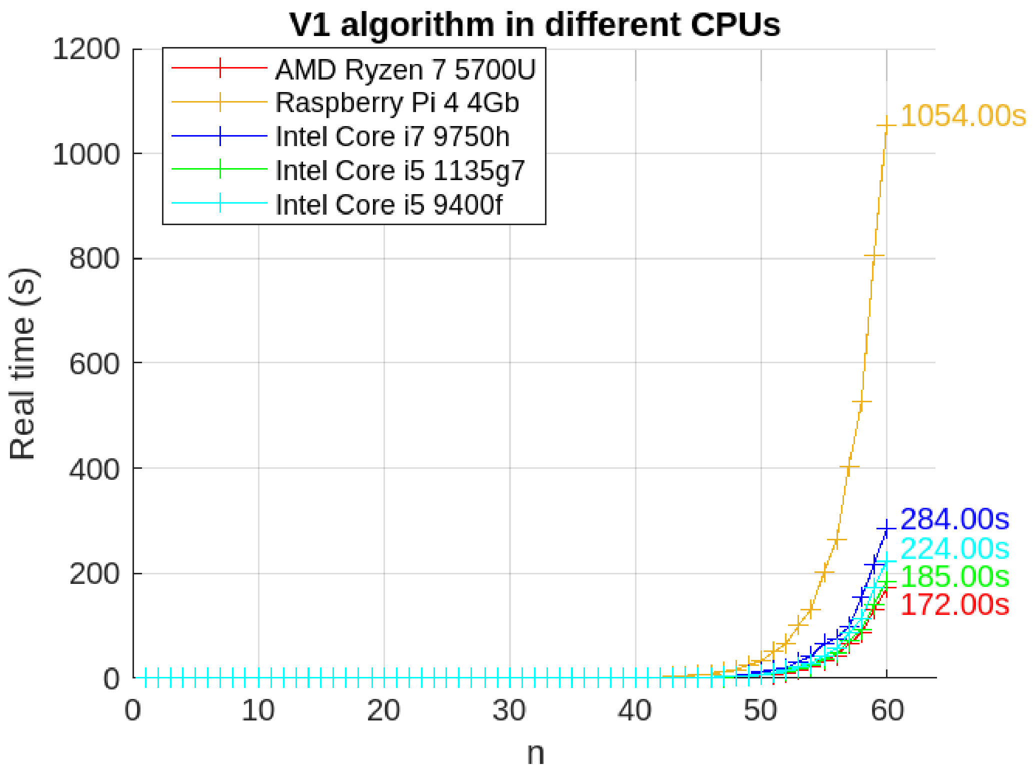


Figura 14: V1 algorithm executed on multiple different CPUs

From these, we can conclude that the performance difference between similarly classed processors, such as the AMD Ryzen 7 5700U and the Intel Core i7 9750h, is quite different. This is mostly because the Intel processor is about two years older, showing that the performance gains over a couple of generations of processors are still growing significantly (despite the slowdown of Moore's law).

We should also note that the AMD processor has a maximum frequency of 4.3GHz, and the Intel processor has a maximum frequency of 4.5GHz, meaning the optimizations made to the efficiency of the processor and cache access mainly drove the improvement noticed.

It is important to note that the machine with the Core i7 is executing the binary through the Windows Subsystem for Linux, which may cause some of the performance loss that we see. If the machine was running a proper UNIX system, it might have fared much closer to the Intel Core i5 1135g7 tested.

We can also see that the ARM processor, tested on a Raspberry Pi 4 4Gb, despite being considerably slower than the rest of the x86 processors, still shows a similar arithmetic complexity as the rest, which means that this complexity is not bound to the architecture of the processor.

This specific ARM processor, named the BCM2711, is built on the ARMv7 Pro architecture and has 4 Cortex-A72 cores with a maximum frequency of 1.5GHz.

From the tests we can see that despite the ARM processor having, for example, 2.8 times lower clock speed than the AMD Ryzen 7 5700U, the final compute time was over 18 times slower. This can be caused by numerous factors, such as the lower size of the cache for this processor, the thermal limits of the Raspberry Pi 4 (which is designed with small packaging in mind, not cooling performance) and the inefficiencies brought by compiling the code to another CPU architecture such as ARM.

Finally, we can also see the differences between two intel CPUs of the same category, but manufactured almost two years apart, the Core i5 9400f and Core i5 1135g7.

These CPUs, despite having similar features and characteristics, have quite different performances, with the newer processor (1135g7) achieving significantly better single-core performance despite the maximum core frequencies of both being similar (4.2GHz for the newer i5 and 4.1GHz for the older model).

The machines used for testing both these CPUs are also quite similar, so the difference in performance can be mostly attributed to the advancements in processor and memory efficiency shown in the two years that space these processors apart.

Concluding this section, the difference in the results obtained from each processor shows that the CPU choice is also massively important if the end user requires the algorithm to complete its work as fast as possible.

This conclusion is to be expected, as it is obvious that a newer and more powerful processor will provide better performance in CPU demanding tasks such as the execution of this algorithm, but if we analyse the bottom end of the graph, we can see that the Intel Core i5 1135g7 is remarcably close to the AMD Ryzen 7 5700U, despite being a year older and having a lower maximum frequency, which leads us to confirm our suspicions that AMD processors might favour multi-threaded performance, while Intel performs better in single-threaded tasks, something we can also see in the previous 1.4 section, titled "Multiple Threads", where the AMD processor outperforms the Intel processors by a larger margin than in this single-threaded test.

1.7 Programming languages

To start with, the professor-provided algorithm was implemented in the C programming language, which would then later be translated into different programming languages to be tested.

These languages were chosen based on the most prominent features that they provide, whether it is the usage of interpretation instead of compilation, efficiency claims, ease of use and relevancy to the overall project.

Some languages, such as Bash script and Scratch were also tested to analyse how their limited features would impact overall performance.

To automate the execution of these programs across multiple computers, a bash file was created which executes and stores all the results in files, taking two optional arguments, the "max_n"number and the file's name.

1.7.1 C

The C implementation of this algorithm is taken as the baseline for the main part of this report, as it would be the most sensible choice for implementing such an algorithm since it provides one of the highest levels of efficiency without compromising readability, portability and development time.

For this programming language, the final results were as follows:

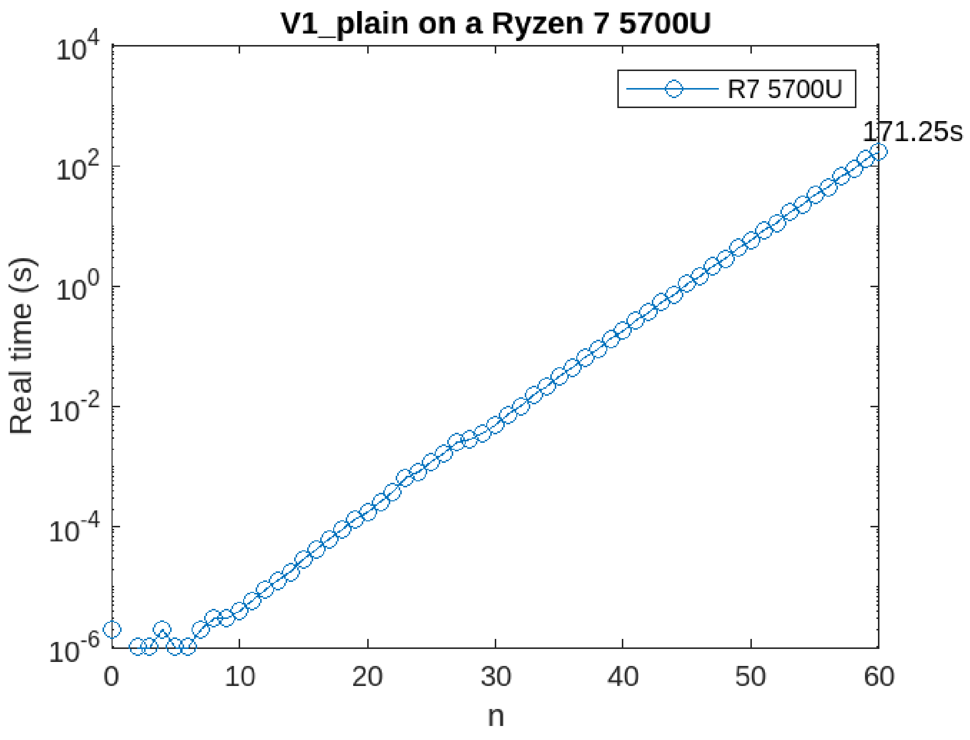


Figura 15: V1 algorithm implemented on the C programming language

As seen previously, this implementation completes the calculations for a N value of 60 after 171.25 seconds.

1.7.2 Python

The Python implementation of this algorithm provides useful information on how an interpreted programming language can affect the performance of the algorithm.

Python is written on top of the C language, but despite this, features such as interpretation, automatic garbage collection and non-static type checking severely affect the final performance, as noted by the results below:

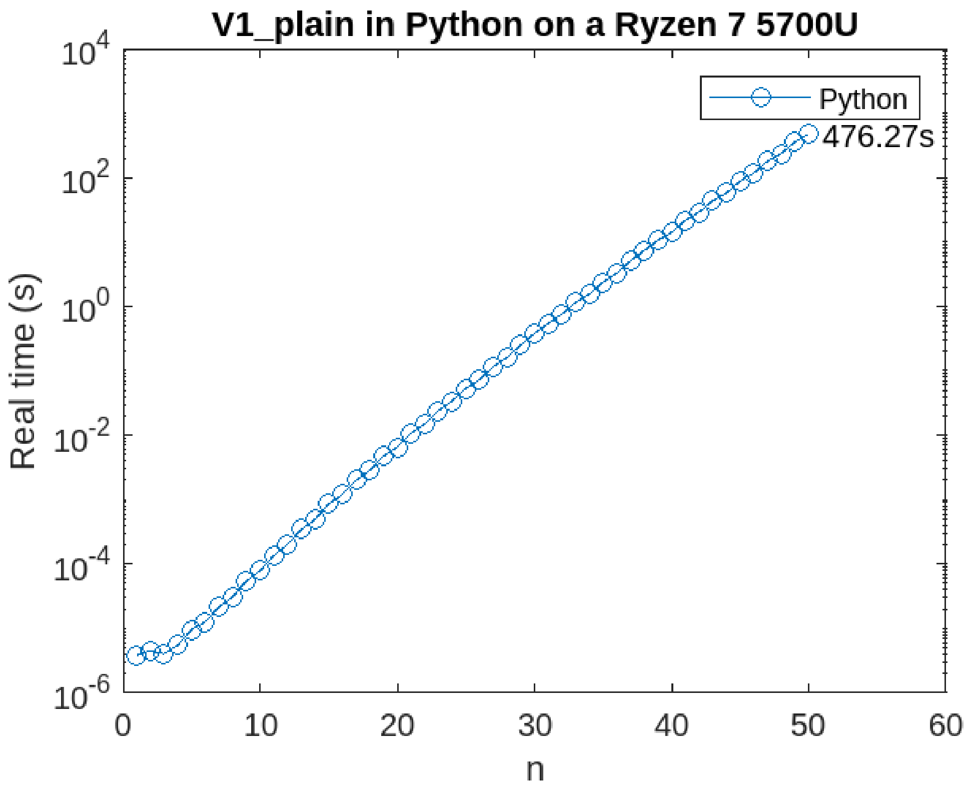


Figura 16: V1 algorithm implemented on the Python programming language

This implementation, despite using the same basic algorithm, completes the calculations for an N value of 50 after 476.27 seconds, which is 83.5 times slower than the C implementation.

With these results we conclude that the Python language is unfit to run such a heavily algebraic algorithm since it is affected by too many inefficiencies to become a viable alternative at higher values of N.

In fact, the Python implementation was so slow that the algorithm only completed the calculations for an N value of 50 in a reasonable amount of time, since it quickly overwhelmed the resources of the machine tested and would take about 6 hours to compute the results for an N value of 60.

1.7.3 Rust

The implementation of this algorithm provides useful information on how a language that focuses heavily on safety, performance, and concurrency fairs in terms of heavy computational performance.

Rust implementation is written in Rust, utilizing the Rust compiler and following Rust's project structure, with the algorithm divided between multiple modules and adhering to Rust's design principles. The results were as follows:

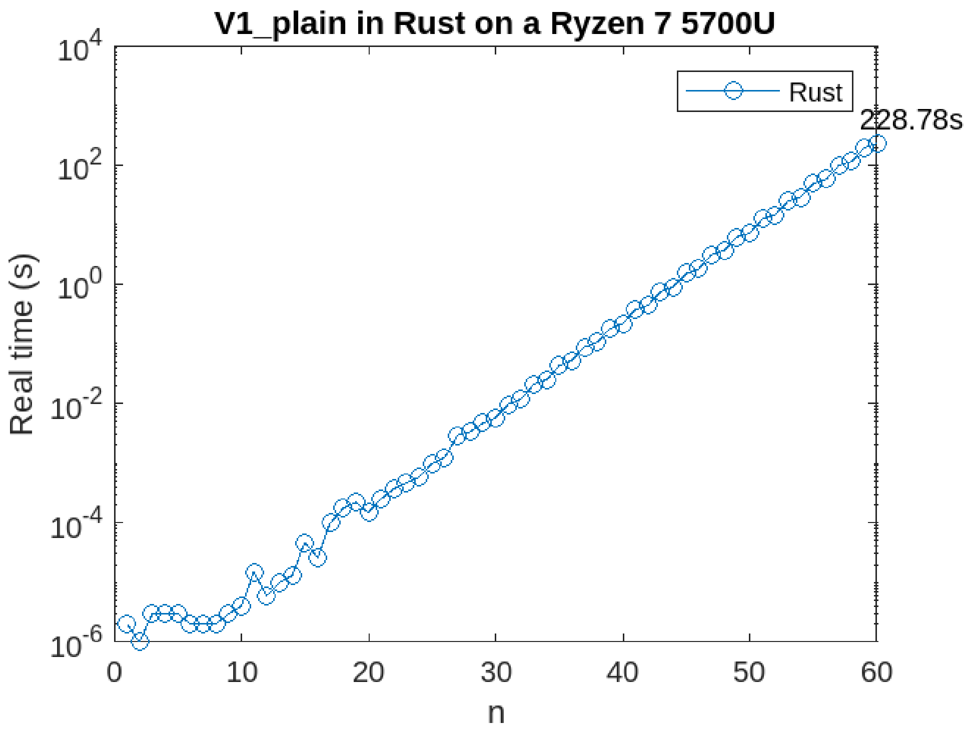


Figura 17: V1 algorithm implemented on the C programming language

These results demonstrate that Rust also offers a good balance between performance and ease of writing, as the result from the test (228.78 seconds) only makes it about 1.3 times slower than C.

Rust was built to improve upon the C++ language, so it retains a lot of the simplicity of C but with modern functions such as a large library of built-in and optimized data structures and other useful functions. This leads us to conclude that if the algorithm tested required a lot of complex data structures and interactions with other libraries, Rust would allow for a much quicker and bug-free implementation without compromising too much of the performance provided by the C language.

1.7.4 Java

The Java implementation of this algorithm provides useful information on how a language that is heavily focused on portability and that runs in the Java Virtual Machine fairs in terms of heavy computational performance.

The Java implementation is written as a normal Java project would, having the algorithm divided between multiple classes and following basic Java design patterns. The tests lead to the following results:

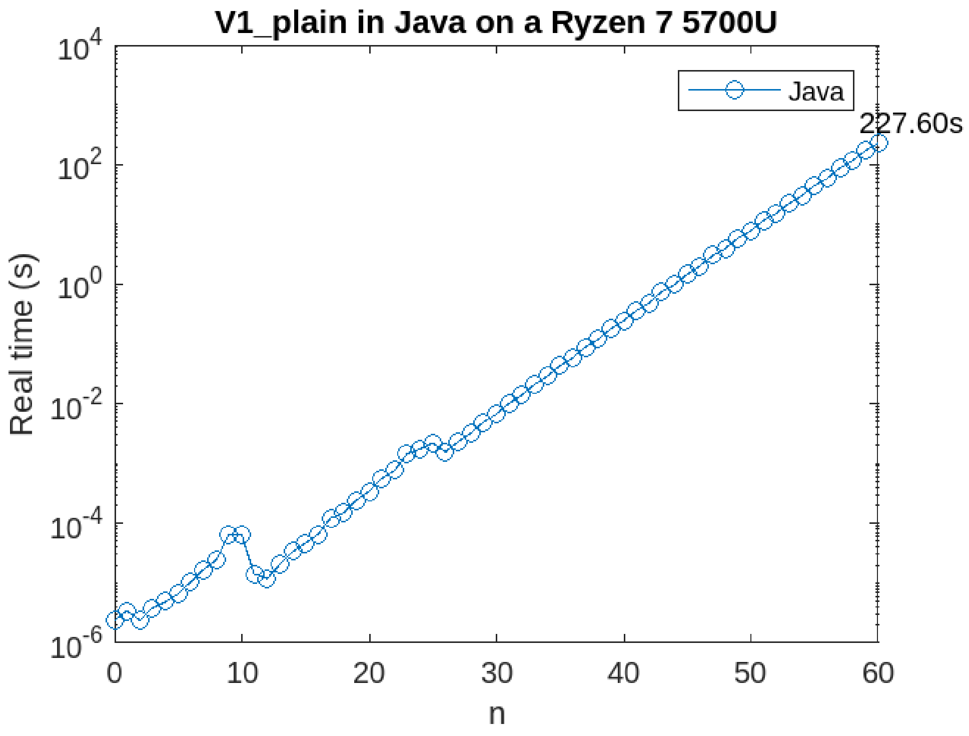


Figura 18: V1 algorithm implemented on the Java programming language

This implementation provided results which are very easily comparable with the base C implementation, since the final real-time necessary to compute the results of the N with a value of 60 was only 227.60 seconds, only 1.3 times slower than the C implementation.

One of the advantages of the Java language for these types of algorithms is the heavy optimization of the code that can be achieved due to the usage of the Java Virtual Machine, which allows the code to be executed in the same way regardless of CPU architecture or the host machine's specifications, leading to optimized execution and memory access, despite the need to launch and maintain the Virtual Machine.

This means that the Java language could be a viable alternative to the C implementation of this project if the requirements necessitate a higher level of code comprehension or portability and the small performance impact is deemed low enough to not affect the choice.

1.7.5 JavaScript

As for the JavaScript implementation, the main focus is once again to test if the portability of the language has a drastic effect on performance, but unlike the Java implementation, JavaScript is meant to be integrated on web, not on a separated Virtual Machine.

The results of the test, when executed using the node.js command in a terminal, lead to the following results:

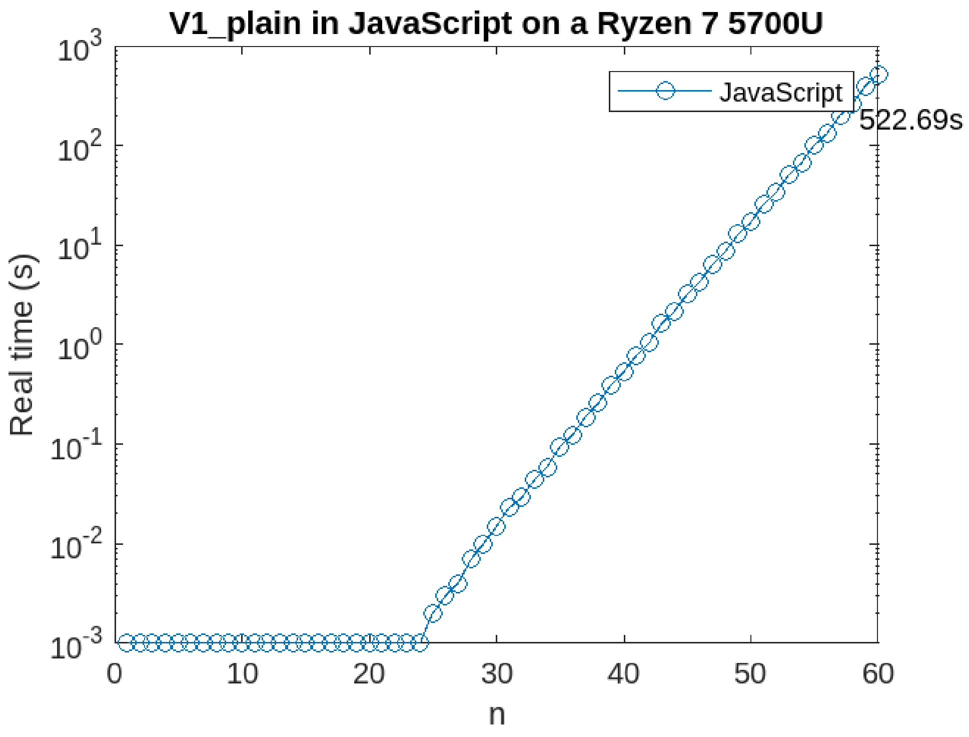


Figura 19: V1 algorithm implemented on the JavaScript programming language

This implementation provided results which are also comparable to the base C implementation, despite being considerably slower than both C and Java. The final real-time for an N value of 60 was 522.69 seconds, which is 3 times slower than the C implementation and 3 times slower than the Java implementation.

This means that the JavaScript language should only be considered if the calculations need to be done, for example, in cases where a client-server architecture needs to execute the algorithm on the client side of the application since it could still provide a relatively moderate performance while being even more portable than the Java implementation itself.

1.7.6 Crystal

The Crystal implementation of this algorithm provides useful information on how a language that is designed for speed, efficiency, and ease of use fairs in terms of heavy computational performance.

The Crystal implementation is written following the typical Crystal syntax, which is completely different to C's and closer to the more expressive syntax of Ruby, making the internal code structure appear much more different than it is. The tests performed led to the following results:

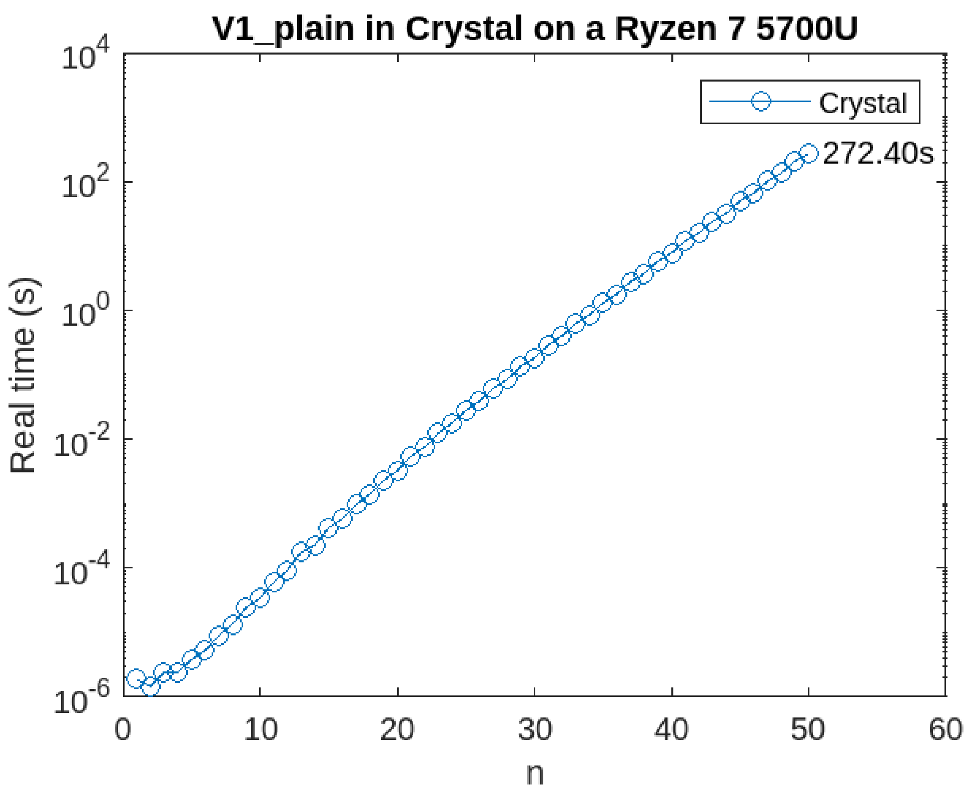


Figura 20: V1 algorithm implemented on the Crystal programming language

From the results obtained, we can see that Crystal's implementation is disappointingly slower than the original C implementation, as it did not even complete the calculations for a N value above 50 in a reasonable time. The final real-time for calculating the result of the n value of 50 was 272.4 seconds, which is 47.8 times slower than the C implementation.

Crystal's main goal is to provide a language that achieves performance close to C's but with a syntax similar to Ruby's.

These results may be caused by a poorly transcribed program, but our Crystal implementation follows the same basic structure as the rest, so more testing needs to be done.

Despite this, we conclude that the Crystal language should not be considered for these types of algorithms.

1.7.7 GO

The Go implementation aimed to test how simple the conversion between C and GO truly was, along with how efficient it is compared to C and other languages when executing this algorithm.

Go is a statically typed, compiled high-level programming language designed at Google, highly influenced by C but with emphasis on greater simplicity and safety, therefore its performance is expected to be quite decent, as we can clearly see from the following results:

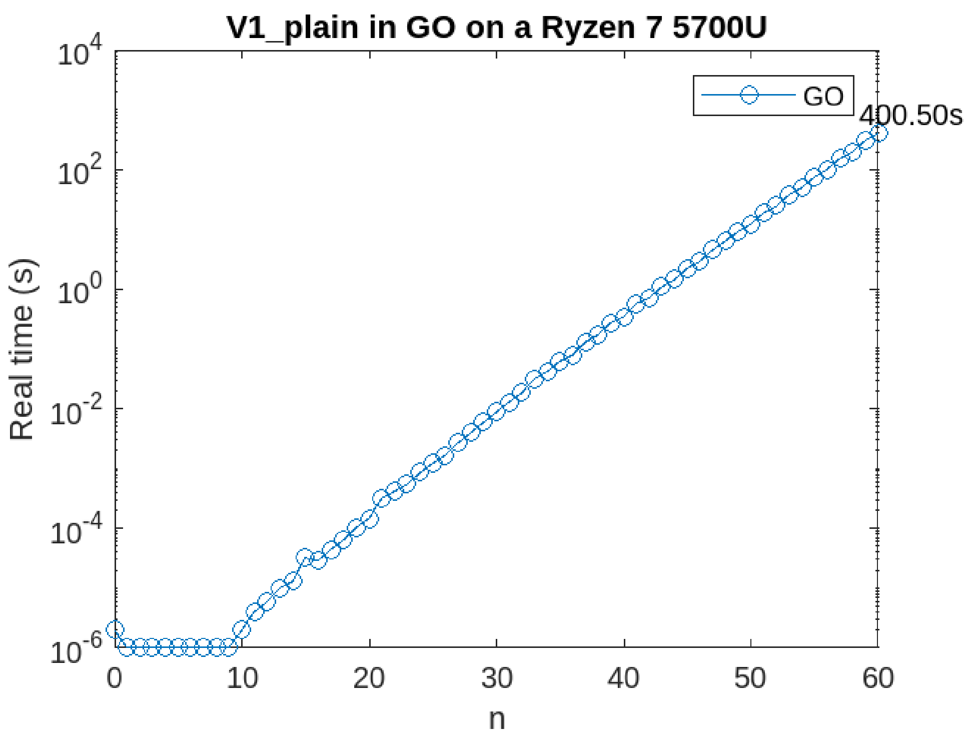


Figura 21: V1 algorithm implemented on the GO programming language

From these results, we can conclude that Golang is not as competitive as C, normally being half the time slower. However, when comparing this with other languages, such as Javascript and Python, it wins by a big margin. This implementation reaches an N value of 60 at 400.5 seconds, which when compared to C, is 2.3 times slower.

If simplicity is a must but efficiency is also required, Golang is an effective language for these problems.

1.7.8 Bash

The Bash implementation was created to test how a scripting language that is only meant to chain commands would be fair when tasked with executing an algorithm with high computational complexity.

Bash does not provide many of the required functionalities by itself, since it is more of an ”orchestration language” for binding the synchronized execution of other programs. Still, our implementation only required a couple of these programs to be used, mainly to time the algorithm itself, so they do not affect the final results by more than a few milliseconds. The results from the tests performed were as follows:

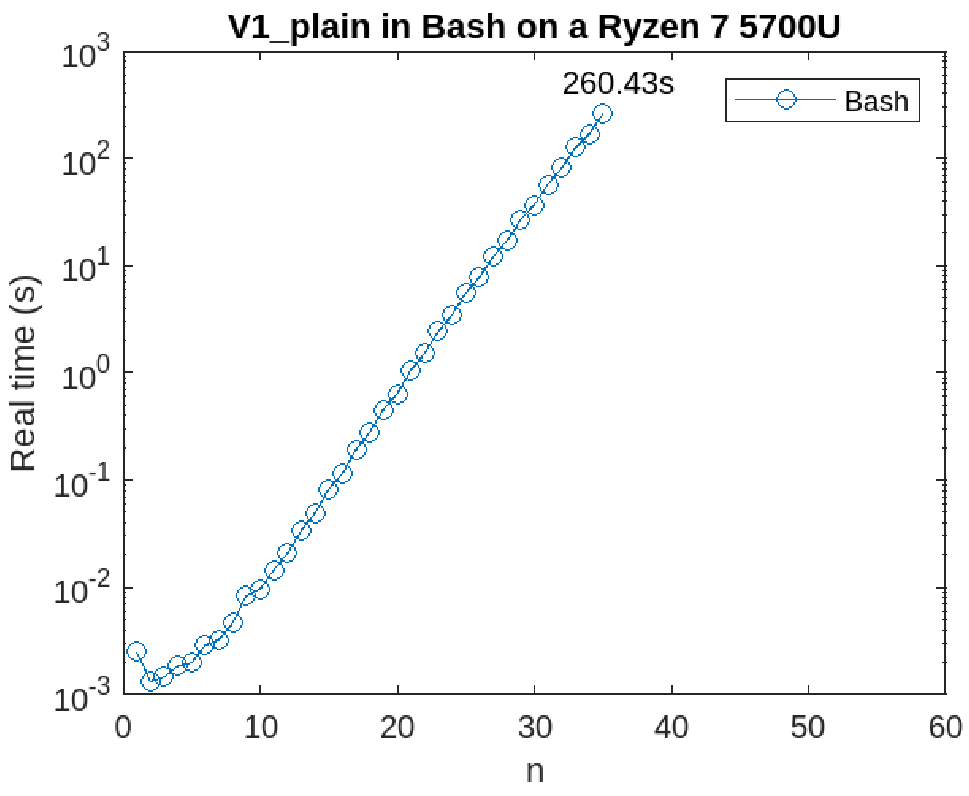
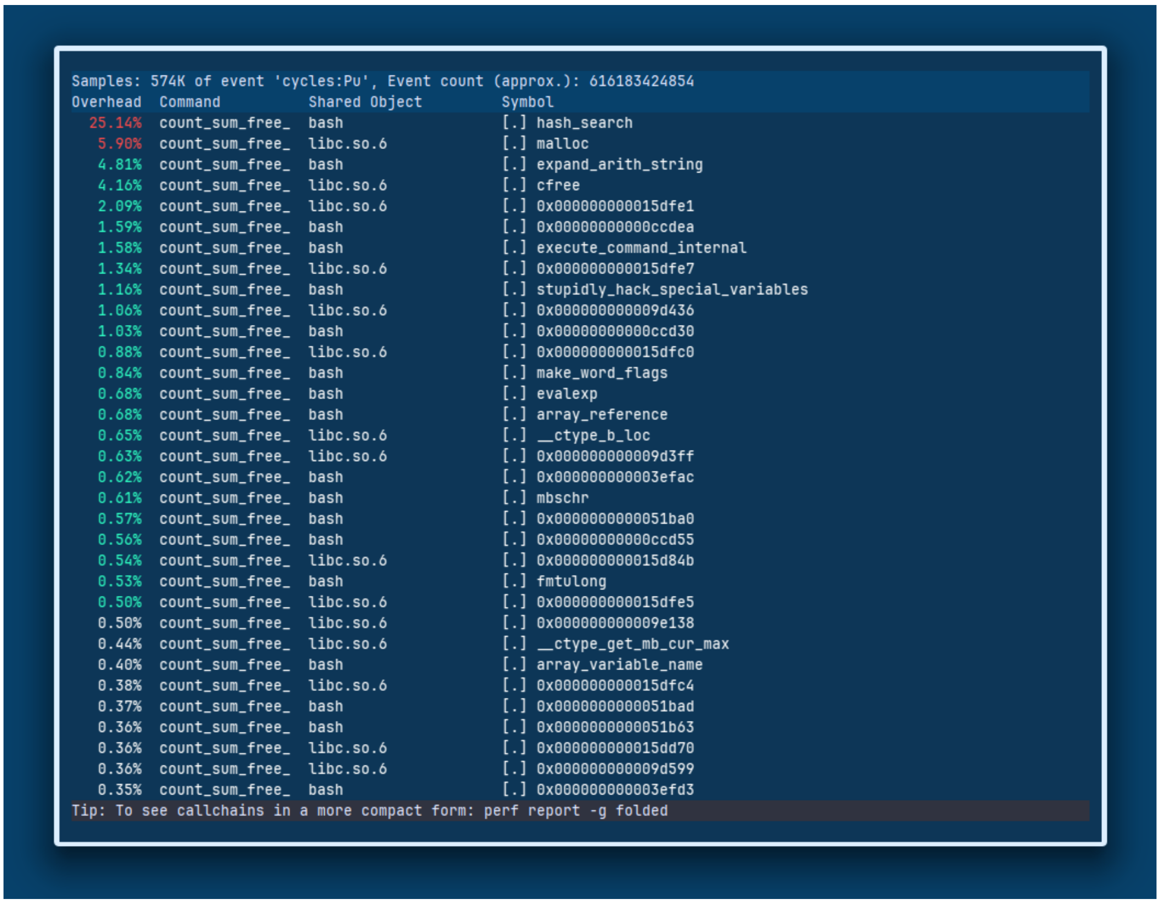


Figura 22: V1 algorithm implemented on the Bash language

As expected, the results were terrible when compared even to the slowest of the tested languages, with our Bash implementation only achieving the results for an N value of 35 in a reasonable amount of time. It took 261.0 seconds to calculate the results for the N value of 35, which is 8 233,4 times slower than the C implementation.

To further explain why this performance is so low, we executed the program through the ”perf” performance measuring tool.



```
Samples: 574K of event 'cycles:Pu', Event count (approx.): 616183424854
Overhead  Command          Shared Object          Symbol
  25.14%  count_sum_free_  bash                   [.] hash_search
   5.90%  count_sum_free_  libc.so.6              [.] malloc
   4.81%  count_sum_free_  bash                   [.] expand_arith_string
   4.16%  count_sum_free_  libc.so.6              [.] cfree
   2.09%  count_sum_free_  libc.so.6              [.] 0x000000000015dfe1
   1.59%  count_sum_free_  bash                   [.] 0x00000000000ccdea
   1.58%  count_sum_free_  bash                   [.] execute_command_internal
   1.34%  count_sum_free_  libc.so.6              [.] 0x000000000015dfe7
   1.16%  count_sum_free_  bash                   [.] stupidly_hack_special_variables
   1.06%  count_sum_free_  libc.so.6              [.] 0x000000000009d436
   1.03%  count_sum_free_  bash                   [.] 0x00000000000ccd30
   0.88%  count_sum_free_  libc.so.6              [.] 0x000000000015dfc0
   0.84%  count_sum_free_  bash                   [.] make_word_flags
   0.68%  count_sum_free_  bash                   [.] evalexp
   0.68%  count_sum_free_  bash                   [.] array_reference
   0.65%  count_sum_free_  libc.so.6              [.] __ctype_b_loc
   0.63%  count_sum_free_  libc.so.6              [.] 0x000000000009d3ff
   0.62%  count_sum_free_  bash                   [.] 0x000000000003efac
   0.61%  count_sum_free_  bash                   [.] mbschr
   0.57%  count_sum_free_  bash                   [.] 0x0000000000051ba0
   0.56%  count_sum_free_  bash                   [.] 0x00000000000ccd55
   0.54%  count_sum_free_  libc.so.6              [.] 0x000000000015d84b
   0.53%  count_sum_free_  bash                   [.] fmtulong
   0.50%  count_sum_free_  libc.so.6              [.] 0x000000000015dfe5
   0.50%  count_sum_free_  libc.so.6              [.] 0x000000000009e138
   0.44%  count_sum_free_  libc.so.6              [.] __ctype_get_mb_cur_max
   0.40%  count_sum_free_  bash                   [.] array_variable_name
   0.38%  count_sum_free_  libc.so.6              [.] 0x000000000015dfc4
   0.37%  count_sum_free_  bash                   [.] 0x0000000000051bad
   0.36%  count_sum_free_  bash                   [.] 0x0000000000051b63
   0.36%  count_sum_free_  libc.so.6              [.] 0x000000000015dd70
   0.36%  count_sum_free_  libc.so.6              [.] 0x000000000009d599
   0.35%  count_sum_free_  bash                   [.] 0x000000000003efd3
Tip: To see callchains in a more compact form: perf report -g folded
```

Figura 23: Performance analysis of the Bash implementation

With this report, we can speculate as to why this implementation is so slow.

At first we notice a large amount of hash map-related functions and memory manipulation functions regarding arrays.

This is because Bash does not provide the array data structure that we are familiar with, but instead, it used hash maps with variable size. This means that despite the algorithm not needing any sort of hash map, the Bash implementation has to convert all the lists and arrays into these structures, which are immensely more complex to access, store and manage. The advantages of hash maps are from the search of entries by their values, and this function is not used in our algorithm.

Furthermore, it is impossible to set a fixed size for these types of structures (at least in Bash), which leads to many memory allocation operations in order to transfer the values once the original data structure is full. These are costly operations that cause even more efficiency loss.

Concluding, the Bash scripting language should not be used to implement any sort of large algorithms, since the language is not designed to handle such complex data structure management and arithmetic.

1.7.9 Web Assembly

Web Assembly is a binary instruction format designed to allow easy porting of other languages to a web-based application. Web Assembly is not a language by itself, but rather a ”compilation target” which aims to be fast, simple and safe.

In this project, Web Assembly was used to compile the original C implementation of the V1 algorithm to a web application made with JavaScript.

This allows us to compare the results of the tests performed, not only to the C version but also to the previously tested ”manual” JavaScript version.

Another advantage of this tool is to allow the algorithm to be executed efficiently in client-server architectures, as the client would not need to compile or even manually download the code itself, but rather access it through a web server with his browser, severely simplifying the distribution of the algorithm between machines.

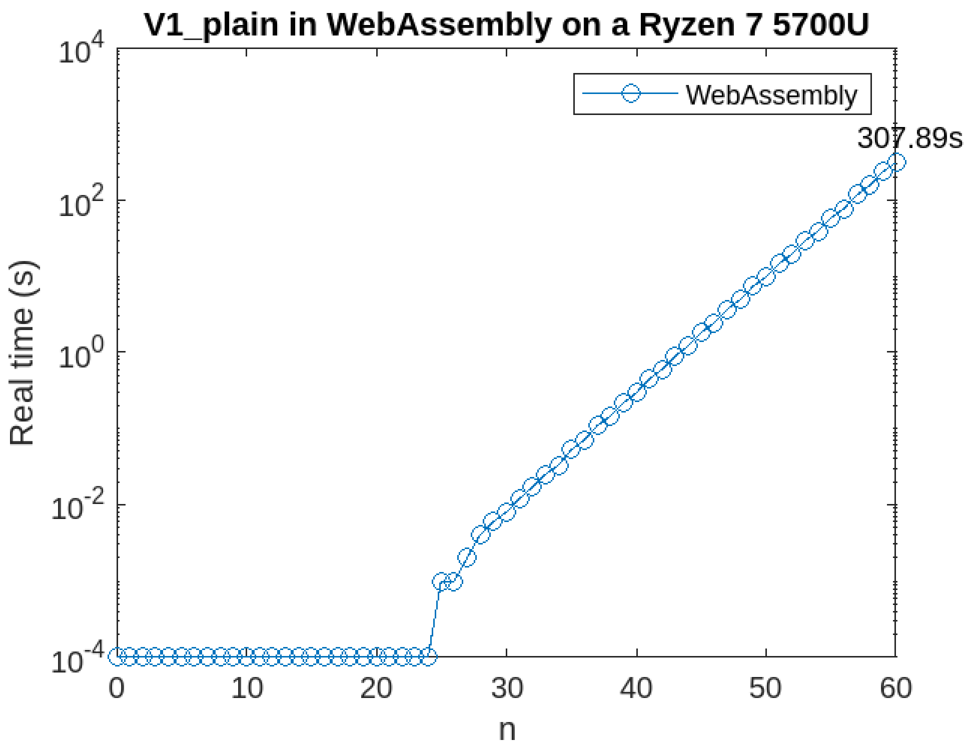


Figura 24: V1 algorithm implemented in Web Assembly

These results demonstrate that Web Assembly compilation makes the algorithm noticeably slower than the original C version (compiled with the gcc tool), even when the Web Assembly compiler used (emcc) was also called with the same level of optimization (level 2).

The Web Assembly implementation took 307.89 seconds to calculate the result of the algorithm with an N value of 60, which is 1.8 times slower than the C implementation.

Despite this, if a web application was necessary in order to execute the program on multiple machines over a web interface, Web Assembly would still provide relatively fast performance, given the increase in overhead caused by the translation to JavaScript and by needing a web browser to be executed.

We can also note that these results are significantly better than the JavaScript implementations, meaning that instead of writing the JavaScript itself, it would be preferable to implement the algorithm in C and then compile it using the WebAssembly tool kit.

This difference in performance may have come from the level of optimization used, which does not exist in the previous JavaScript tests.

Concluding, Web Assembly would be a good tool for distributing the algorithm through a web interface, since the original implementation does not need to be rewritten to another language and the performance is still in the same order of time.

1.7.10 Scratch

This implementation in scratch tries to show how even though this language is block-based it still has the capacity to run complex algorithms at a reasonable performance compared to its simplicity.

Scratch is a visual programming language developed at MIT Media Lab. It is designed primarily for children and beginners in programming, providing an easy-to-understand and highly interactive platform. This language is javascript-based, however it isn't javascript. Due to it being javascript-based, it benefits from its performance, compatibility and integration capabilities.

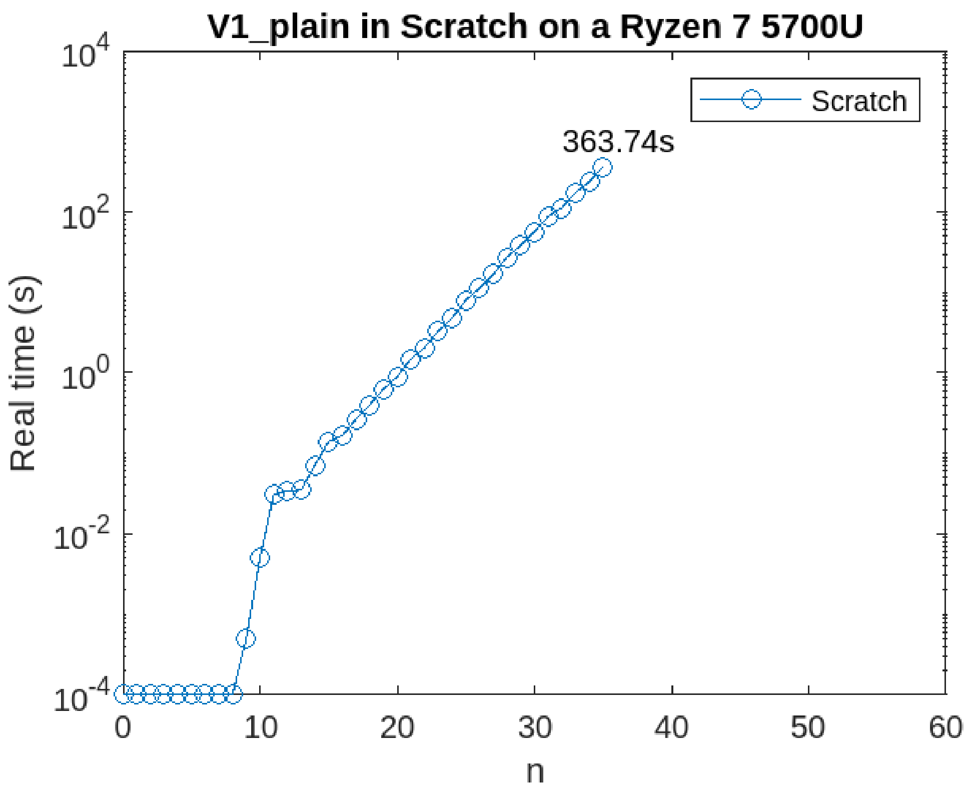


Figura 25: V1 algorithm implemented on the Scratch programming language

These results demonstrate that a simple language like scratch is roughly 1.3 times slower than bash and roughly 36 300 times slower than C. This as a beginning language is more than enough to make children understand the fundamentals and alike, but it should not be used for complex algorithms.

Concluding, Scratch serves as an excellent educational tool for introducing programming concepts to beginners with its visual and interactive nature which makes it highly accessible and engaging for young learners. However, for tasks that require high efficiency and execution speed, languages like C are more appropriate.

1.8 Conclusion

Finally, we arrive at the comparison between the languages tested.

We first should note that all these tests were executed on an AMD Ryzen 7 5700U CPU and on the same hardware, but these results were also confirmed on several other processors from other vendors and with different hardware, operating systems and even CPU architecture (for example, on a ARM system).

Each language ran a ”manually ported” version of the V1 algorithm originally written in C. Due to the nature of how these languages were tested, some issues with performance might come from errors in the porting of the code, but since the algorithm and basic data structures are the same, these problems will not affect the results by such a margin that would make our conclusions invalid.

The plotted results from all the languages are as follows:

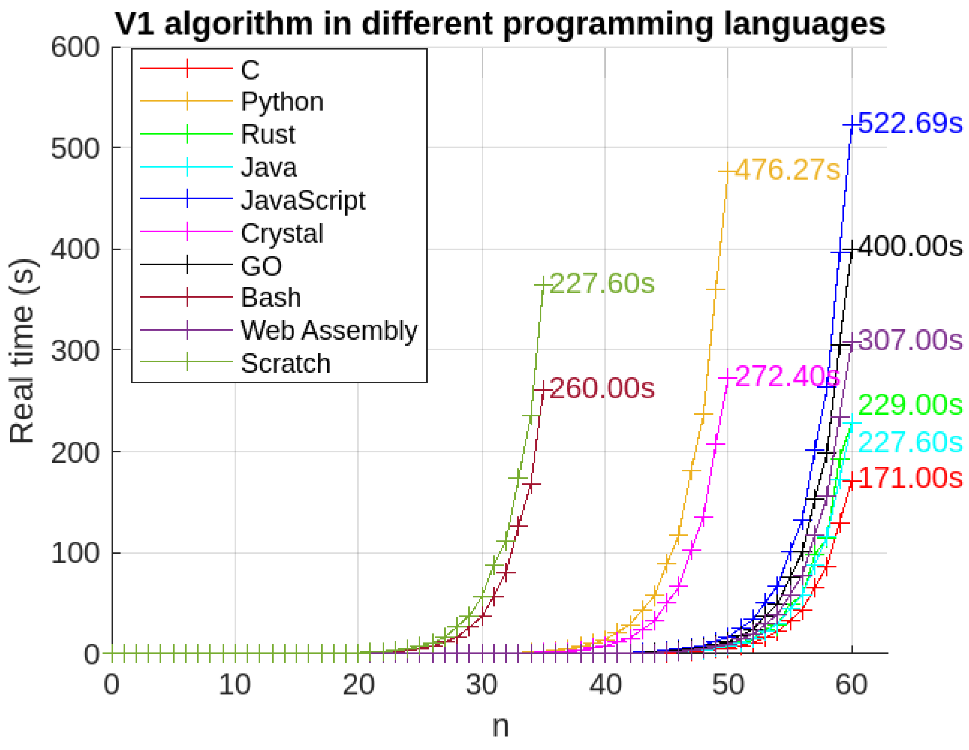


Figura 26: V1 algorithm implemented on the Scratch programming language

From these results, we can conclude that C is, as expected, the best-performing language for such a computationally complex algorithm, but there is still some relevant data to gain from these values.

If the algorithm needs to run on multiple machines with different hardware and even operating systems with minimal configuration or manual deployment, the Java programming language could easily provide such features with a performance close to C's.

Rust also provides great performance as well as a more complete set of default data structures and functions, which could help reduce the development time of algorithms that rely on more complex classes.

Web Assembly is also a great option if server-client web support is a necessity for the project since it can provide the algorithm through a web application while still retaining good performance.

GO has the upside of being relatively easy to write, but our results show that it was still more than two times slower than the C implementation. This might be due to our implementation, but we still expected it to be better performing.

JavaScript performed better than we speculated, but it still was a lot slower than Web Assembly, so unless it is necessary, it shouldn't be used for these sorts of algorithms.

Crystal, like GO, was supposed to closely match the efficiency of C but did not even achieve a performance near the one expected. The Crystal implementation did not even manage to calculate the results for an N value of 60 in a reasonable amount of time. We need to note that this language is still relatively new, so some specific performance issues might have caused our results to lower more than expected.

Python was also much slower than C, which was expected but not by such a large margin. Python is an interpreted language and is not statically typed, so algorithms relying on high recursion and large data structures will lose a lot of time on tasks such as memory allocation and management, leading to high-performance overhead. Like Crystal, the Python implementation did not surpass the N value of 50.

Bash was also very slow, but unlike the Python implementation, whose main issues came from the large and complex feature set, the problem with the Bash implementation of such algorithms begins with the reduced pool of available features and data structures, leading to the usage of improper alternatives that cause massive overhead, such as hash maps instead of normal lists. Despite this, we must keep in mind that Bash is not meant to solve these sorts of tasks but rather be a simple and portable scripting tool for chaining commands.

Finally, the Scratch implementation shows that, despite the horrible performance, this algorithm can easily be adapted to more intricate and "strange" languages and still work as expected. This also shows that Scratch is capable of more than simple programs, and there is more to it than simply being a block-coding tool for beginners.

Concluding, if the goal is portability, Java would be the preferred language to implement this algorithm. If a web application is necessary, Web Assembly would also provide good performance. If the code can be adapted, compiled and executed manually on the intended machine, the C implementation is considerably faster than all the others tested.

Project 2: Memory Access Times

2.1 Context

For a CPU to operate its needs data to process, and that data must be inevitably stored somewhere.

With the exponentially increase in CPU speeds since their first use, the need for more and faster memory has increased as well, being memory latency, quantity and speed a heavy factor in a computer's general performance.

With this problem in mind, several types of memory are available commercially nowadays, such as :

- CPU Caches (fastest, lesser quantity)
- RAM (slower than caches but more quantity)
- Solid State Drives (higher quantity, relatively high speed, used to persist data in disk)
- Hard Disk Drives (higher quantity, relatively slow, mechanical, used to persist data in disk)

These types of memories are made with different purposes in mind some for storing data that will be used "soon", such as RAM and CPU Caches and others for persisting data even if the computer is shut down such as SSD's and HDD's.

In this project, we explored the first two levels mentioned, being that they are the most important for programs that require high CPU usage (lots of CPU operations) but that do not necessarily need to persist data or read data from a disk.

CPU Caches often come with three layers associated, each one having more speed but a lesser size, typically L1, L2 and L3.

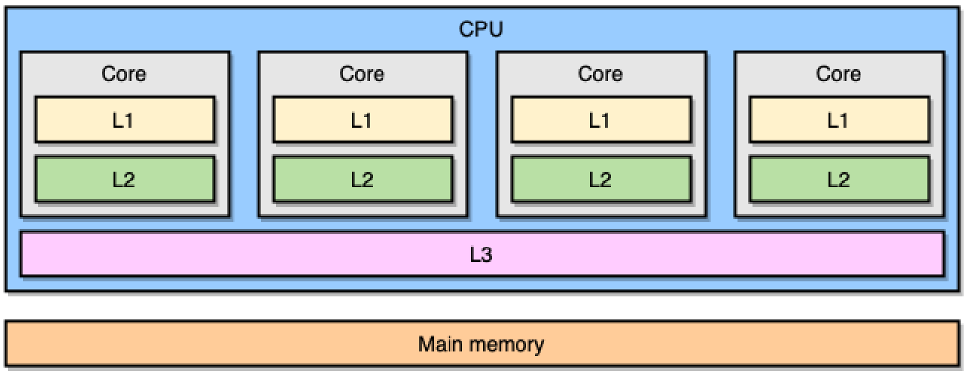


Figura 27: Computer Storage Organization

As can be seen in the image above, L1 and L2 caches exist for each of the CPU cores and there is a last level, L3, which is shared by all cores. For size reference, in a modern CPU, such as an I7 9750H, each level has the following sizes:

- L1 Instruction Cache: 6 x 32 KB
- L1 Data Cache: 6 x 32 KB
- L2 Cache: 6 x 256 KB
- L3 Cache: 12 MB

When the processor is looking for data to carry out an operation, it first tries to find it in the L1 cache.

If the CPU finds it, the condition is called a cache hit. It then proceeds to find it in L2 and then L3.

If the CPU doesn't find the data in any of the memory caches, it attempts to access it from your system memory (RAM). When that happens, it is known as a cache miss.

Associated to a cache miss there is an increase in latency, as the CPU now needs to ask the RAM for the given data. Decreasing the level of storage which the CPU needs to access will increase the time taken, inevitably making a program slower.

In the image below, we can see the time it takes to do various operations, specifically in the language GO with an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz. Note that GO is one of the fastest languages in recent times, only being surpassed by its predecessors, C, C++ and possibly Rust.

Credits to amyangfei for the image (https://amyangfei.me/2021/05/03/golang-operation-latency/).

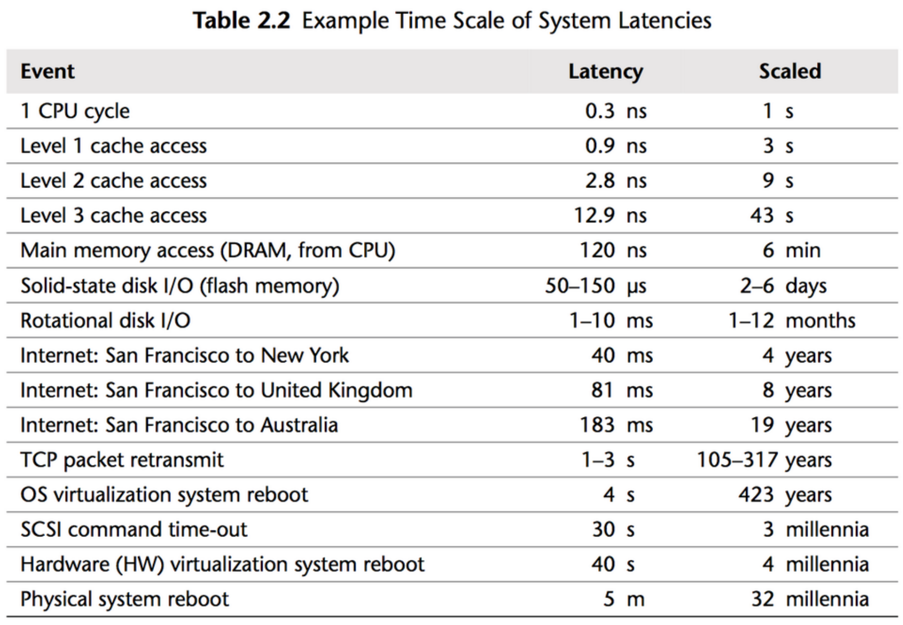


Table 2.2 Example Time Scale of System Latencies

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 μs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

Figura 28: System Latencies

From this table, we can conclude that accessing an L1 cache or the computer's RAM has a 133x difference in speed, with a single RAM access taking about 400 CPU cycles of time.

In a program's optimization is then important to take into account the CPU cache organization to maximize the program's speed as much as possible.

2.2 Implementation

To help us with our objectives, two programs were provided by the professor.

The first program consists of simulating multiplications between squared matrices of a given size.

The storing of matrix elements in memory is done with 3 different layouts, to check how this can affect memory and cache usage.

- Row Major
- Column Major
- Morton Layout

The organization of the first two layouts in memory can be seen in the image below.

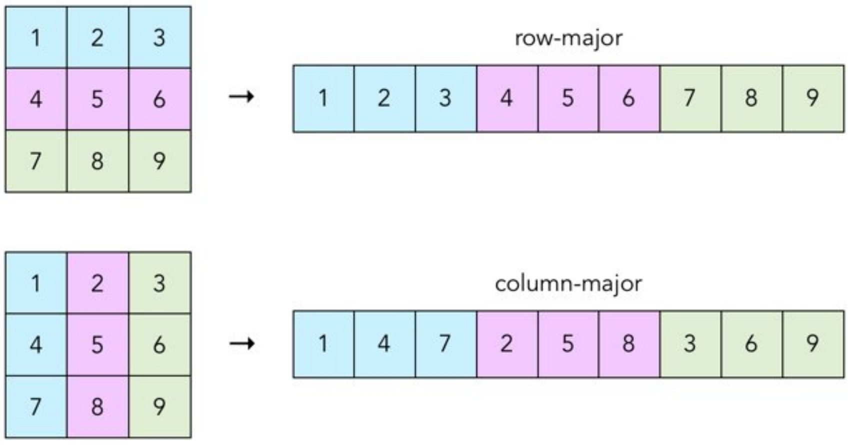


Figura 29: Row Major and Column Major layouts

Morton Layout consists of a Z-shaped layout, which preserves locality in the matrix elements, aiming to be cache-efficient.

The transformation of a normal row-major layout to a Morton Layout is represented in this image:

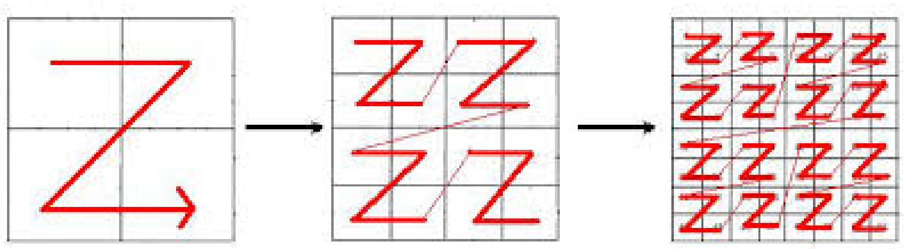


Figura 30: Row Major transformation to Morton Layout

With a Morton Layout, also known as a Z-order curve, elements that are stored near each other in the matrix are also stored near each other in the memory representation, which is especially useful since caches are stored in contiguous blocks of memory, which means its most likely that elements that are to be accessed next in the matrix are present in the CPU caches, improving program performance.

This layout also maintains a hierarchical structure, effectively dividing a matrix into a couple of matrices, which can be useful for multi-level data structures.

A better and most in-depth explanation of how memory blocks are loaded into the CPU caches can be found in this course:

https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec15.pdf

Row Major and Column Major layouts are also present between themselves concerning cache performance.

As memory addresses are loaded to the CPU caches in blocks, it is of interest to keep the next element of the matrix to be used in the same contiguous block of memory as the current one.

Unfortunately, since matrix multiplication is usually done with a row of matrix A and a column of matrix B, finding a proper layout is not easy, since a layout that benefits the storing of elements of matrix A may not benefit matrix B.

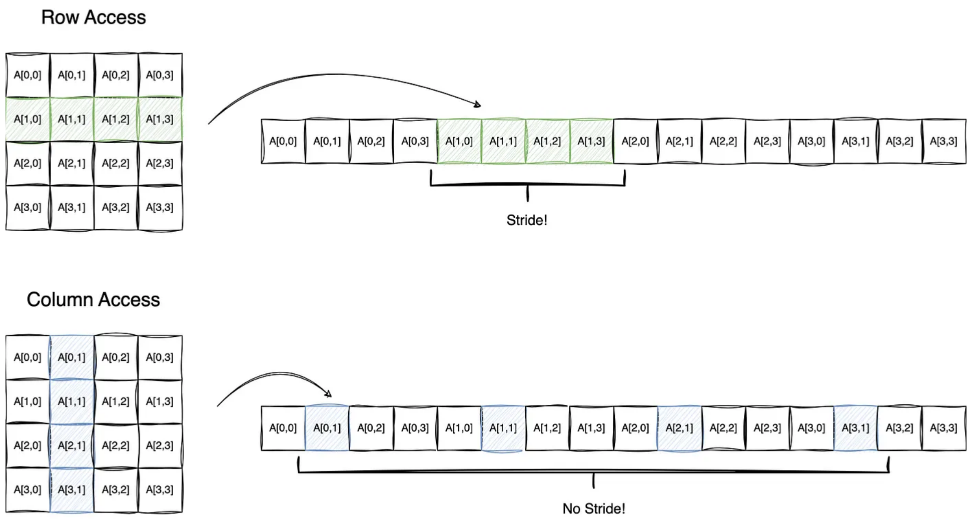

Row Access
A[0,0] A[0,1] A[0,2] A[0,3]
A[1,0] A[1,1] A[1,2] A[1,3]
A[2,0] A[2,1] A[2,2] A[2,3]
A[3,0] A[3,1] A[3,2] A[3,3]
Stride!
Column Access
No Stride!

Figura 31: Memory organization of layouts

For each of these 3 layouts, the program then computes the matrix multiplication for a given size. The matrix multiplication is done with the 6 possible permutations of rows, columns and intermediate indexes, I, J and, K respectively.

A simple implementation of an IJK matrix multiplication in Java is present in the following image.

```
 public float[] baselineIJK(float[] a, float[] b, float[] result, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            float sum = 0.0f;
            for (int k = 0; k < n; k++) {
                sum += a[i * n + k] * b[k * n + j];
            }
            result[i * n + j] = sum;
        }
    }
    return result;
}
```

The other five implementations follow the same approach, changing the order of the ”for”loops.

For each layout and each permutation, the time taken to complete the matrix multiplication is then stored.

For additional performance measures, the Linux ”perf”tool is used to measure the cache references and cache misses in each of the multiplications, which is then used to calculate a cache miss percentage.

This program lets us study how the organization of elements in memory affects the cache performance and utilization in a CPU-heavy program, ultimately understanding how performant the program can be.

In the second program provided by the professor, a gigabyte of memory is allocated, varying the size of each page allocated, and consequently, the number of pages allocated to reach 1GB.

The program first starts by printing data about the used CPU, such as its manufacturer, caches, cache levels and respective association.

A gigabyte of memory is then allocated on the computer, on the first run with 4096 bytes page size, then with 2MB pages and then with a single page of 1GB.

In the allocated memory pages, operations are then done to measure the bandwidth of the computer’s memory and its latency, utilizing two functions, *measure_bandwidth* and *measure_latency*.

The measure_bandwidth function executes various read and write operations of 8 bytes on the allocated memory pages and measures their time, to achieve a bandwidth in Gib/s.

The measure_latency function performs walks on a circulary-linked list and measures the time these operations take.

Singular Walk

```
b0 = (block_t *)memory;
for(n = 0;n < n_passes;n++)
    for(i = 0;i < m;i++)
        b0 = b0->next;
```

Walking two linked lists at once

```
b1 = (block_t *)memory + (m / 2);
for(n = 0;n < n_passes;n += 2)
  for(i = 0;i < m;i++)
  {
    b0 = b0->next;
    b1 = b1->next;
  }
```

Walking three linked lists simultaneously

```
b1 = (block_t *)memory + (m / 3);
b2 = (block_t *)memory + ((2 * m) / 3);
for(n = 0;n < n_passes;n += 3)
  for(i = 0;i < m;i++)
  {
    b0 = b0->next;
    b1 = b1->next;
    b2 = b2->next;
  }
```

With these two functions, the program effectively measures the latency and bandwidth in Gib/s of the computer's memory, printing the results in the following format.

page_size=4096

Size (bytes)	Read (GiB/s)	Write (GiB/s)	Latency1 (ns)	Latency2 (ns)	Latency3 (ns)
8192	34.685	20.939	1.421	0.748	0.478
10240	32.241	20.426	1.651	0.753	0.505
12288	26.040	19.101	1.475	0.728	0.464
14336	37.519	20.160	1.469	0.704	0.481
...	...	...	...	...	...

With the help of this 2 programs, we conducted several tests in order to take conclusions about how cache-friendly code can help with its performance and how different RAM's of different manufacturers compare in their specifications and performance.

2.3 Results

To start off, we decided to compare what permutation gets better results within the same layout.

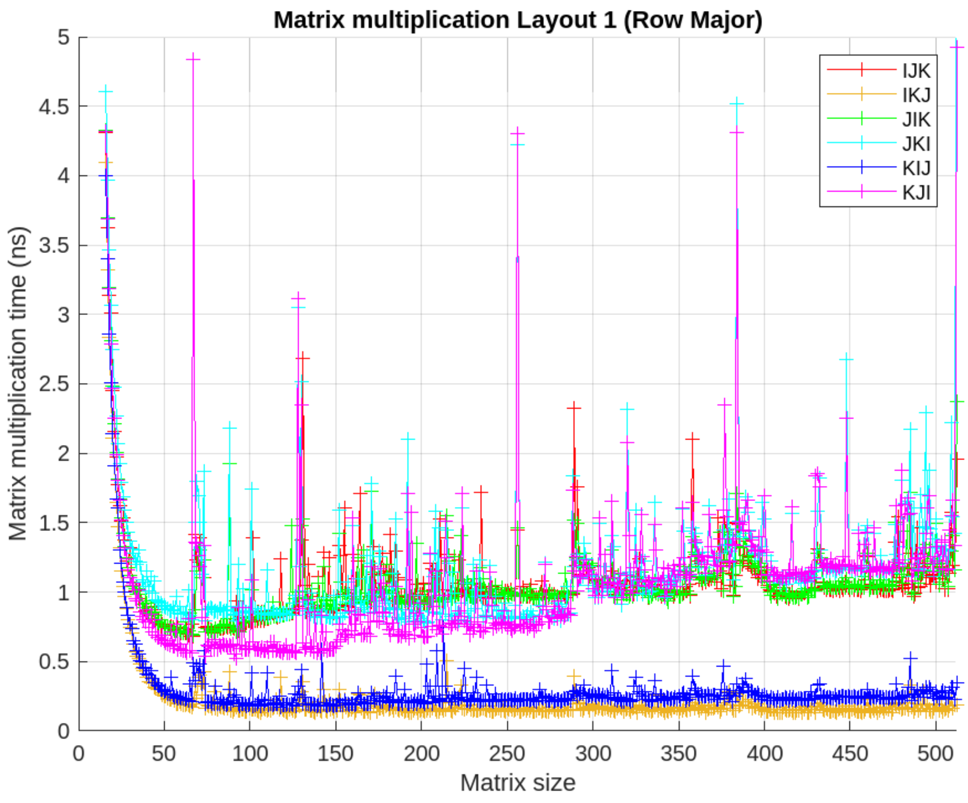


Figura 32: Matrix Multiplication Row Major

In the Row Major matrices multiplication, the initial time taken starts high (most likely due to the program start) and has some peaks at specific matrix sizes which may be attributed to high CPU usage or system interruptions.

From a matrix size (we are assuming multiplications with squared matrices) of 100, the IKJ ordering seems the best one, being the one that achieves the lowest time taken to compute a multiplication at n = 512.

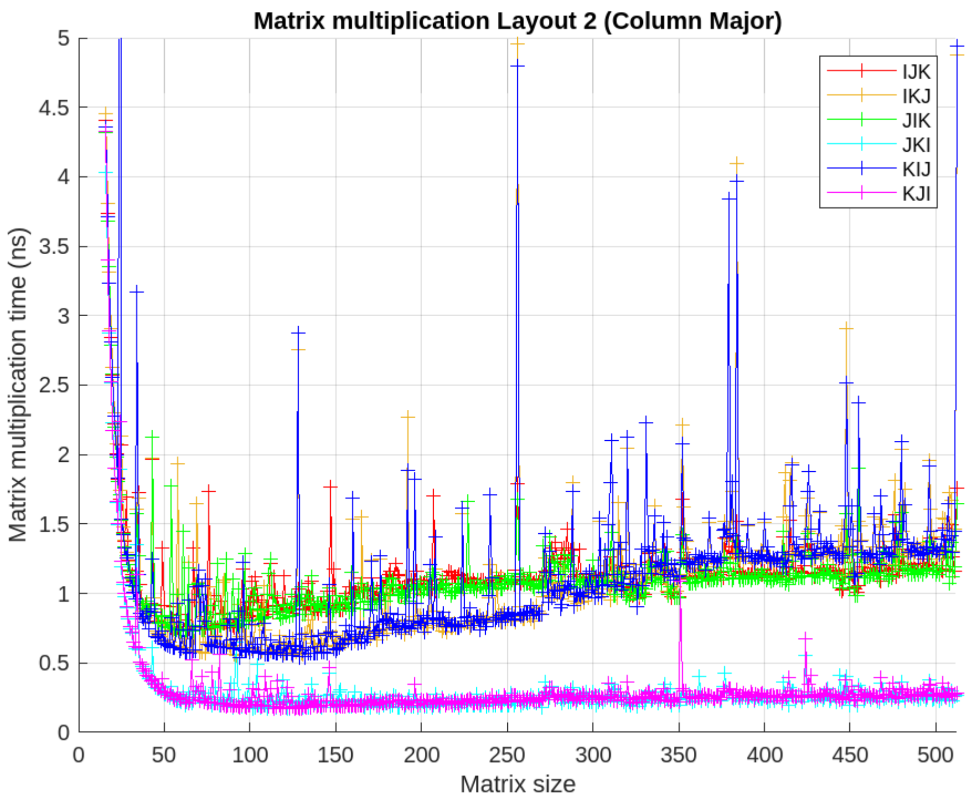


Figura 33: Matrix Multiplication Column Major

In the Column Major matrices multiplication, the winner seems to be the KJI layout, achieving the lowest value for matrix sizes of 512 (similar to the row-major layout)

Comparing the two layouts, we can see that they present similar performance for most of the size values.

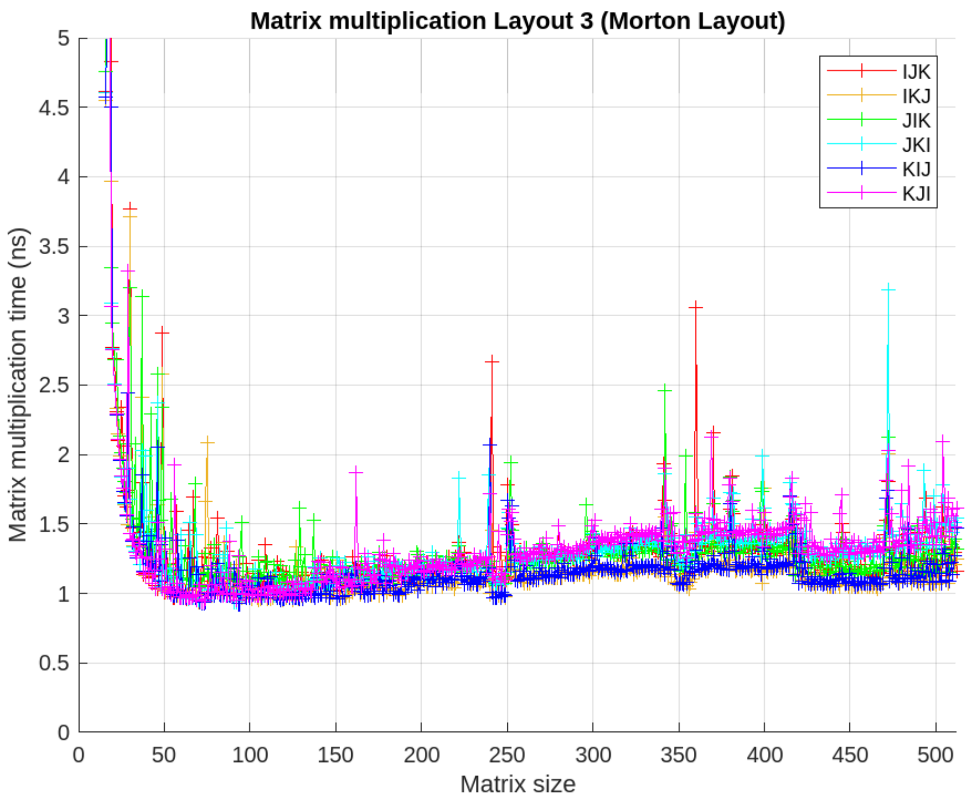


Figura 34: Matrix Multiplication Morton Layout

The Morton Layout seems to be the worst one of the three, even though it's supposed to be the most cache-efficient. This is most likely because of the extra computing time taken to convert the matrix to a Morton layout in memory, which seems significantly higher than the performance and cache-friendliness provided by this layout.

Still, in this layout, the winner is the KIJ ordering.

Comparing now the winner ordering of each layout, the results are as follows:

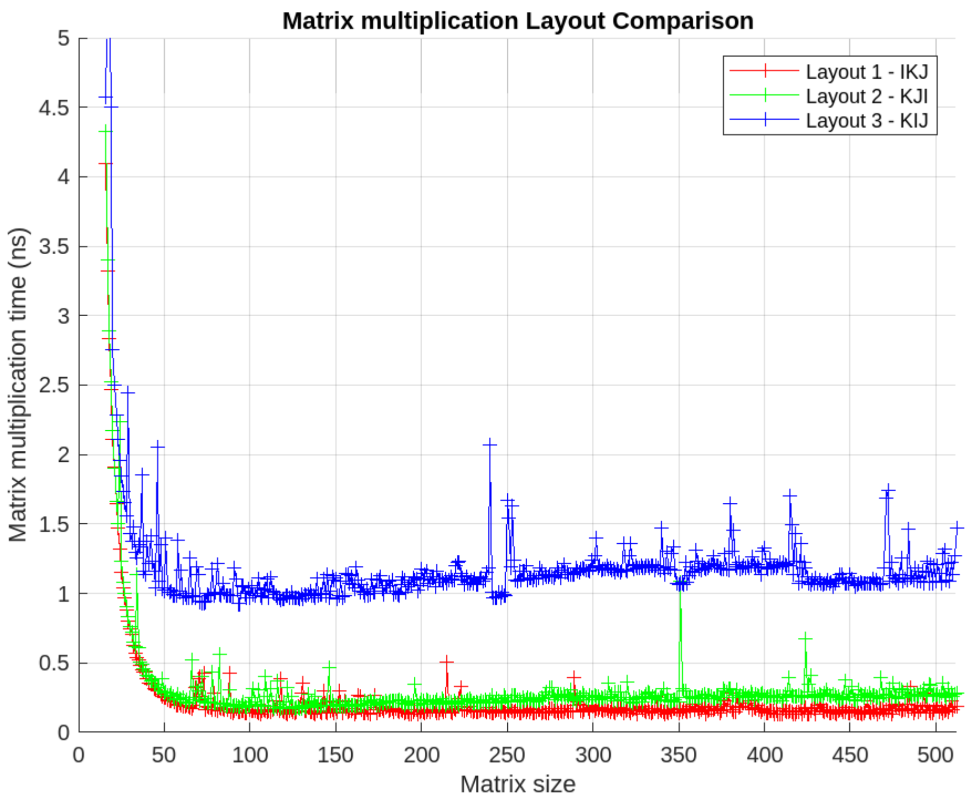


Figura 35: Layout Comparison

As we can see, Layout 1 presents the lowest values of compute time to multiply the matrices for almost all sizes. Therefore, we can conclude that it is the best one and most cache-friendly out of the three layouts tested.

For the 2nd program, we decided to compare how a different page size and operations can affect the memory performance.

We first started by measuring the difference in memory and write speeds with the increasing number of operations.

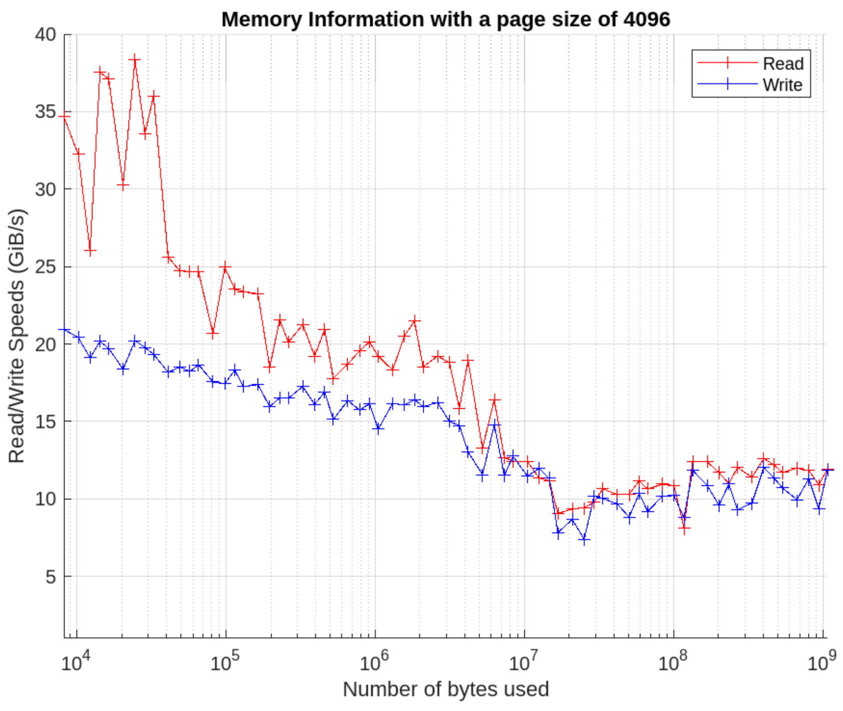


Figura 36: Memory Read and Write tests with a page size of 4096

As was expected, with an increasing number of operations, the read and write speeds decrease has the memory is under a bigger load, and since the accesses done are random (the address is chosen at random), the read and write speed will inevitably decrease.

We also measured the latency of the memory as the number of operations increased.

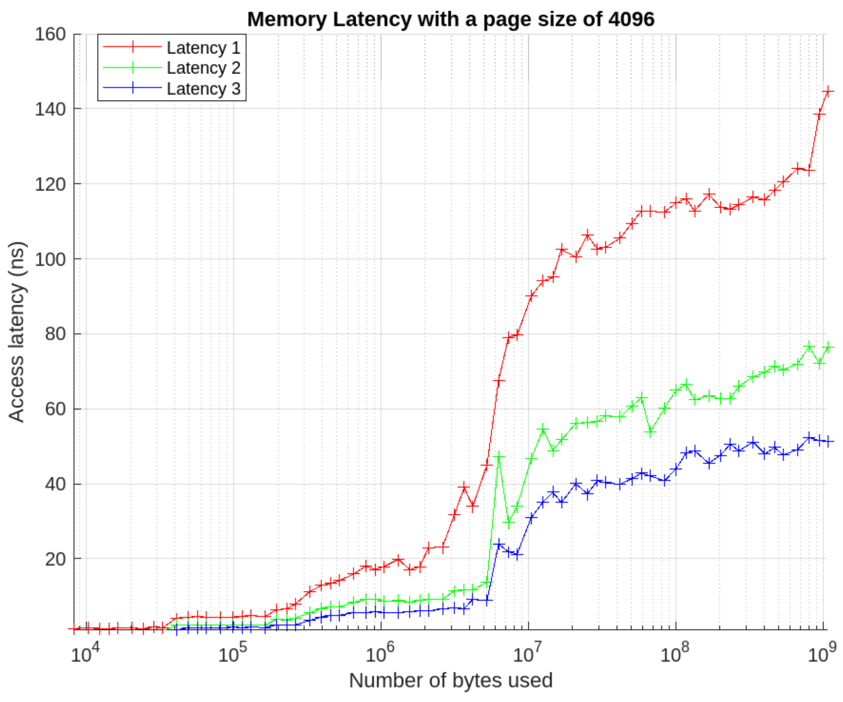


Figura 37: Memory Latency tests with a page size of 4096

Since the read and write speeds decrease with the number of operations, it is only normal that the latency in accessing each ram address will increase, resulting in more time to do the operation.

We also decided to test if the page size used affected the memory performance.

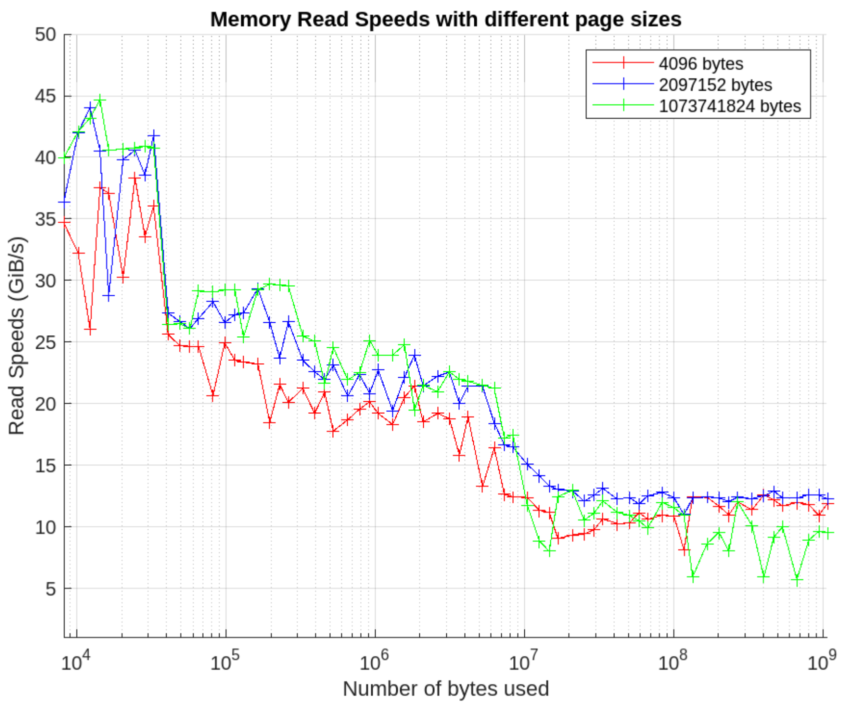


Figura 38: Comparison of Read Speeds with a variable page size

We concluded that the page size didn't affect the memory performance, most likely because the Operating System controls the mapping between virtual memory (which is the one we allocate) to real memory, and optimizes it regardless of what page size we try to use.

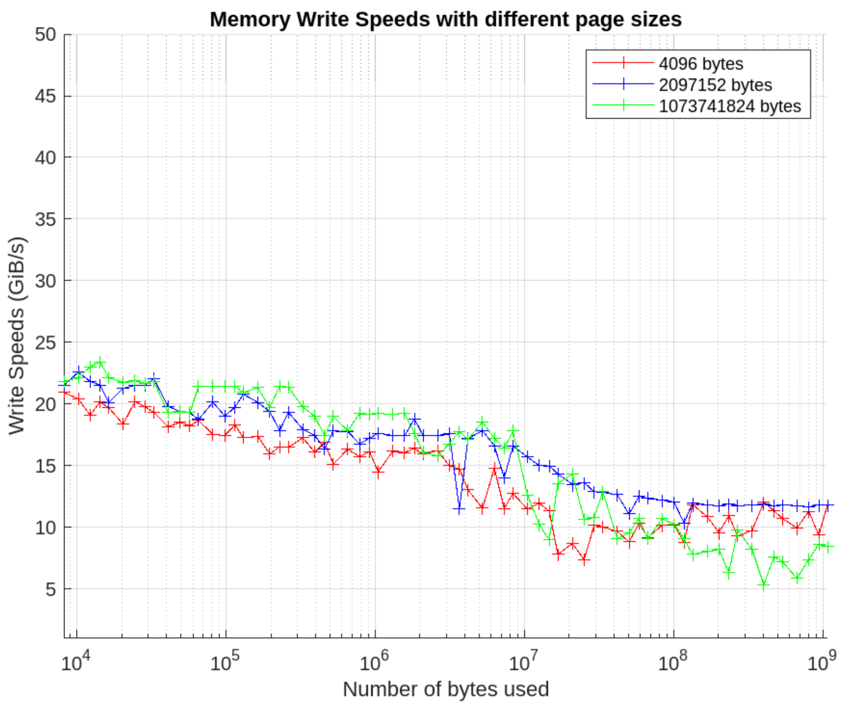


Figura 39: Comparison of Write Speeds with a variable page size

The same happened for memory latency, as can be seen in the image below, so we can conclude that using different page sizes does not affect memory performance significantly.

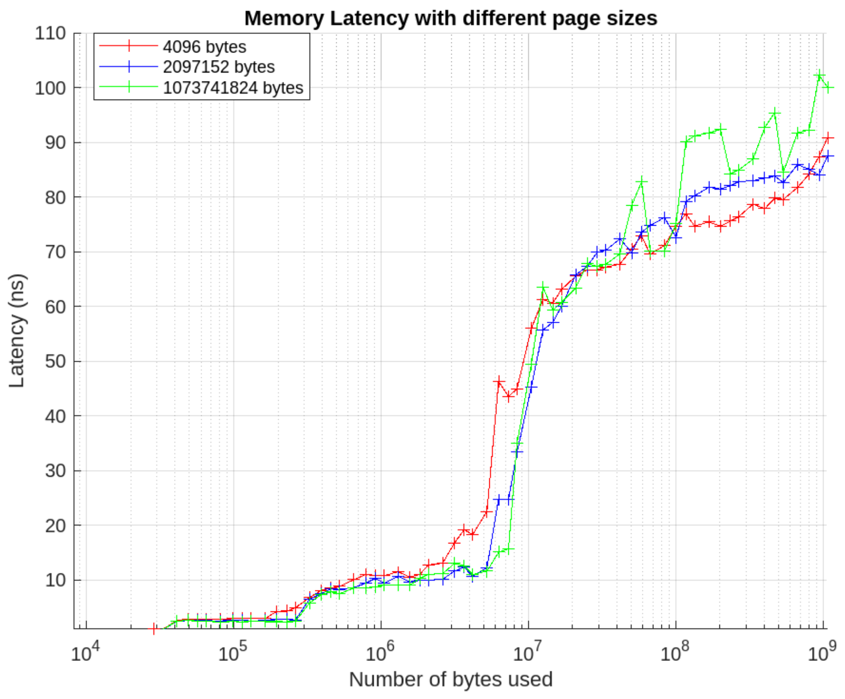


Figura 40: Comparison of Memory Latency with a variable page size

2.4 Conclusion

With this project, we concluded that when developing code it is important to be aware of the CPU caches since their correct use significantly increases the performance of a program.

The tendency in recent times is for programming languages to move to higher abstraction. While this provides a better developer experience, it takes away the programmers ability to directly control the hardware, which can be done in lower level languages such as C, which grants programs in this language a significant performance boost.

It is important to also take in account a computer's RAM memory, its size and performance in general (CAS latencies, DDR version, Frequency) so we can better measure the performance of a computer overall.

In general, we think it was a important project to undertake, especially because of the theoretical concepts we could learn while developing it, which may help us in the future for other tasks, such as for example, buying a new computer.

Project 3: Pierce Expansion in OpenCL

3.1 Context

The final Project for this course is similar with the two previous project, being composed of an algorithm used to calculate the Pierce Expansion of a specific number and then testing our implementation in multiple environments.

The Pierce Expansion algorithm is derived from the Engel Expansion algorithm.

The Pierce Expansion of a positive real number, x, is the unique non-decreasing sequence of positive integers (a1, a2, a3, ...) such that:

$$x = \frac{1}{a1} + \frac{1}{a1a2} + \frac{1}{a1a2a3} + ...$$

This algorithm calculates the maximum number of whole divisions using the Euclidean algorithm for the greatest common divisor of the previous division's result.

So, for example, the number 12 would divide by 2 to become 6, which will then be divided by 2 to become 3 and then by 3 to become 1.

Our implementation executes this algorithm for every number from 1 to a given parameter, called "b", and calculates the maximum number of whole divisions for each number between 1 and "b", returning the largest found.

This implementation seems rather simple, but the real task with this project is implementing the algorithm on a Graphical Processor Unit using the OpenCL toolkit.

This algorithm performs a massive amount of similar algebraic calculations with few flow control instructions, so it should perform much better on a GPU than a CPU, since it can perform many more operations per second due to the much higher core count and our implementation does not require any special instructions that might not be available on a GPU.

OpenCL was chosen instead of CUDA since it provides better support to most of the more recent AMD GPUs, meaning that OpenCL should keep the comparisons fair between the hardware manufacturers without compromising much of the overall performance.

3.2 Goals

As mentioned above, our goal with tto adaptect is adapting the provided Pierce Expansion algorithm to be executed on a GPU using the OpenCL library.

We then will perform a series of tests to obtain the most relevant features of each GPU and determine if the differences in technical performance correlate to a decrease in the execution time of the algorithm or a difference in arithmetic complexity.

The tests will be performed on a representative GPU of each manufacturer, as well as a mix of integrated and dedicated GPUs. The rest of the hardware of the machines tested is slightly different, but since this implementation mostly utilized the GPU, the differences in performance pertaining to different CPUs, RAM or other components are mostly irrelevant. Still, it should be noted that all the machines tested were semi-recent (no older than 5 years) and the tests were performed with no intense graphical applications executing in parallel.

3.3 Algorithm and Implementation

The code itself can be separated into two components:

- The main code to be executed on the CPU, that prepares the GPU kernel code and memory, launches the required GPU threads, collects the results of the execution from the memory and processes the results;
- The GPU kernel code, that is executed on each GPU thread and calculates the result to a given "b" parameter.

3.3.1 The CPU Code

At the beginning of the program's execution, the main program first retrieves the GPU kernel from the .cl file and stores the kernel in memory.

Then the program obtains the information about the OpenCL driver and the selected GPU device to adapt its configuration and perform as expected independently of how many threads the GPU can provide.

Then a memory area for the results of the calculations will be created. This means that after a GPU thread is done calculating the result of an operation, it will store this value in the main machine's memory, not on the GPU memory itself, allowing it to be easily shared with the processor.

After all the required parameters and memory buffers are set up, the program transfers the OpenCL code stored in the memory to an OpenCL context, which is then compiled similarly to most C compilers.

Finally, the code is executed inside a loop, where "x" iterations are called with the maximum number of threads available until the desired maximum value is achieved.

The maximum number of threads is defined in the OpenCL data retrieved at the start of the program and is bound to the actual GPU hardware. This value is then multiplied by a constant (default is 64) to account for the whole "3D block" of threads available in the GPU.

Finally, the application prints the values obtained and calculates some simple statistics about the execution of the program, such as the total amount of threads called, the maximum number of whole divisions achieved and the total real time spent executing the whole program.

3.3.2 The GPU Kernel

The GPU kernel is simpler, defining only the steps to retrieve the current thread id (which is used to determine which "b" value to be used in the calculations), calculate the results and then place them in the RAM buffer allocated by the CPU.

Initially the kernel retrieves the local index (which is the index corresponding to the current block of threads), which is then summed to the multiplication of the current iteration counter with the number of threads per iteration, to obtain the "total" thread id corresponding to the maximum overall number of threads.

With this value as the "b" parameter, the kernel then loops from 1 to b-1 calculates the maximum number of divisions for each of the numbers, and saves the largest found.

Finally, the kernel stores the highest result obtained in the memory buffer index corresponding to the local thread count of the iteration and exits.

3.4 Results

3.4.1 Largest Whole Division Count

After executing the program, these were the "b" values where the largest whole division count was found:

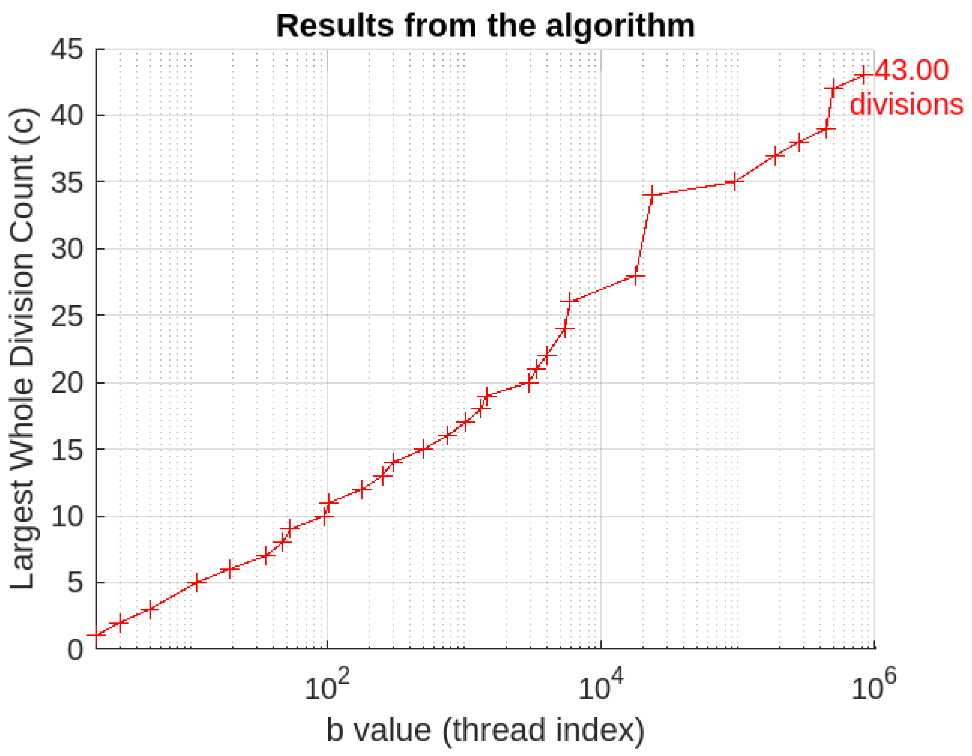


Figura 41: Results obtained from the execution of the algorithm

From these results we can conclude that the largest whole division count steadily rises as the tested value increases, but we must note that the X axis is in a logarithmic scale, so the space between newly found highest values increases logarithmically, meaning that the largest whole division count will slowly reach a maximum overall value.

3.4.2 Comparison between GPUs

The code was tested on three different systems, with two different GPUs.

Firstly, we tested the code on two Nvidia GTX 1660 paired with an AMD and an Intel CPU, to show that the performance only depends on the actual GPU used and not on external factors:

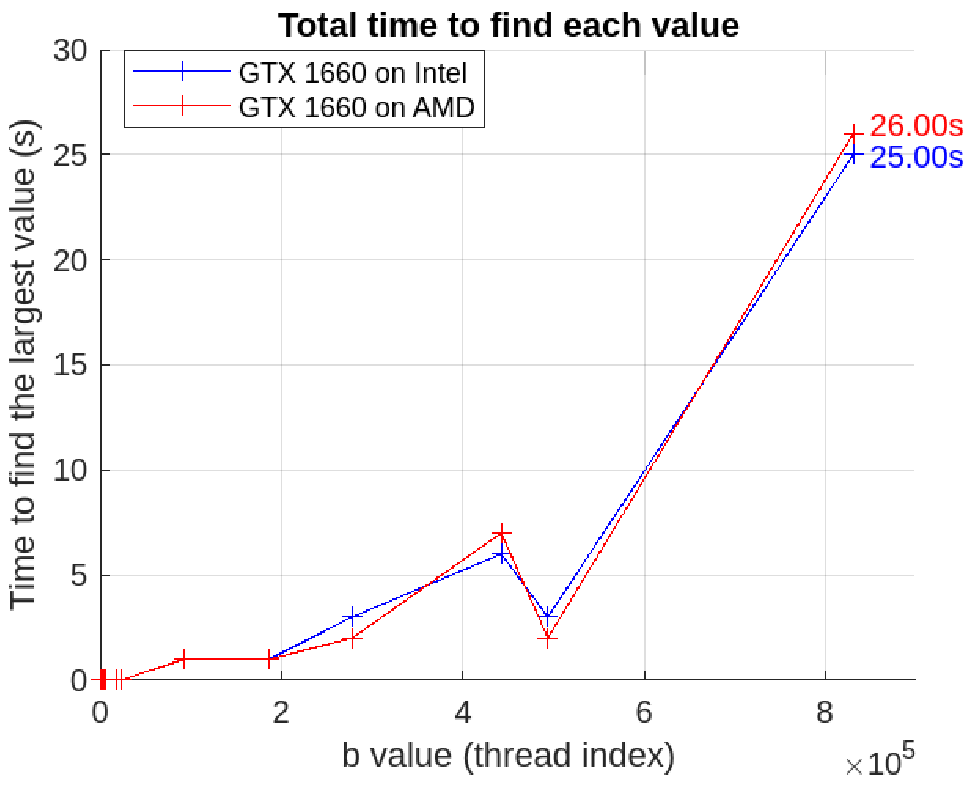


Figura 42: Results obtained from the execution of the algorithm on different CPUs

From these results, we can conclude that a different CPU (and a completely different system) does not influence the performance results by much of a margin. This means that the performance is solely based on the GPU hardware.

Now to determine how much of a difference the GPU makes on the execution times of the algorithm, we tested the same code with the same parameters as before but on an AMD Vega 8 1.7GHz, which is an Integrated GPU provided inside an AMD Ryzen 7 5700U.

This iGPU was tested against the GTX 1660 seen before running on an AMD processor (Ryzen 5 5 5600x).

Both machines have similar hardware, and as we saw before any difference apart from the GPU only results in minor deviations in the total time efficiency of the program.

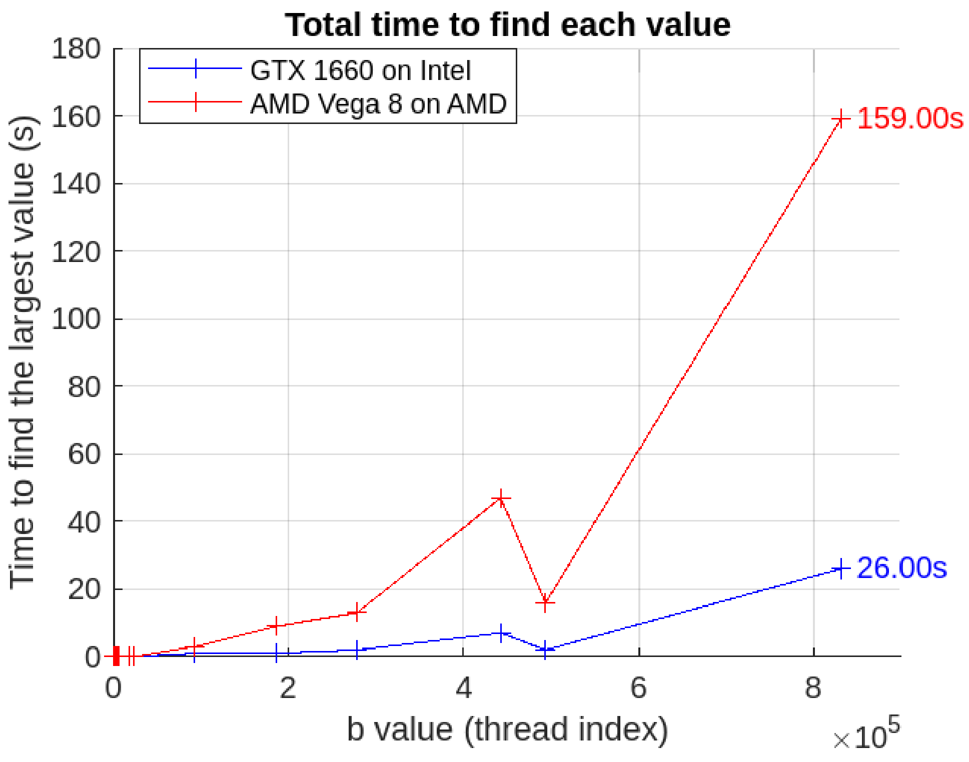


Figura 43: Results obtained from the execution of the algorithm on different GPUs

From these results we can conclude that the Integrated GPU is more than 6.1 times slower than the Dedicated GPU, proving that the performance is tightly bound to the actual GPU hardware itself.

Since the code automatically adapts the number of threads per iteration by using the maximum number of local threads of each GPU, the dedicated GPU used 65536 threads each iteration, resulting in a total of 16 iterations, while the integrated GPU used 16384 threads per iteration (4 times less), which resulted in 43 total iterations.

This seems to make the largest difference in the actual real-time execution of the program since the actual frequencies of each thread is similar on both GPUs (1.8GHz on the GTX 1660 and 1.6GHz on the Vega 8).

3.5 Conclusion

From the results obtained, we can say that the performance obtained on a simple OpenCL adaptation of an algorithm scales well with hardware increases.

We can also conclude that the advances in computing performance made with the development of the CUDA and OpenCL toolkits considerably increased the real-world performance of many applications and algorithms that rely on heavy arithmetic computations, by allowing a much simpler and more efficient use of a GPU for such purposes.

Finally, we conclude that since our algorithm is quite simple and does not take much time to run per thread, the maximum number of threads on a GPU heavily influences the final performance levels more than the maximum thread frequency of the GPU.

We should note that the overall OpenCL programmer experience was much better than expected, since the GPU kernel code was easy to create and had a syntax like C's, meaning that the actual task of developing a GPU-bound application is not as difficult as expected.