Beyond Hardware: Why Efficient Code is Necessary

Enis Brajevic

1 Introduction

A single Google search today uses the same amount of computing power as the entire Apollo program [1]. Knowing this, one could think that technology has advanced so much that highly efficient code is no longer required due to today's powerful CPUs. Even though sometimes this is the case and inefficient code still returns the result in fractions of a second, computers are not infinitely fast: they still have a limit on how many operations are performed at a given time, and memory is also limited.

Correct use of system resources (CPU and memory) through efficient code makes the system more scalable, improves user experience with a faster execution time, and leads to lower energy consumption and less impact on the environment.

2 The impact of code efficiency on computation time

Nowadays, the high-level languages we use, such as Python, equip us with libraries containing functions that help programmers do their job. In particular, these libraries implement the most efficient and optimized algorithm for the specific problem [2].
This gives us an insight into how much code efficiency is valued, and the important role that it plays.

If a novice programmer were to implement a sorting algorithm, with no library at his help, he/she would probably end up implementing the Insertion Sort, the most intuitive sorting algorithm there is. The novice programmer does not know that Insertion Sort has a time complexity of $O(n^2)$, meaning it takes time roughly proportional to n^2 to sort n elements, and is thus quite inefficient because its running time grows quadratically as the input size increases. The algorithmic inefficiency is not immediately seen: our CPUs are powerful enough to outperform the inefficiency of the sorting algorithm for a relatively small input, but in real-world applications, especially in data-driven contexts such as Data Analytics, the input size is often so big that inefficient algorithms as the Insertion Sort are not applicable due to disadvantages in time or memory usage.

Inspired by [3], Table 1 shows a comparison between Merge Sort, taken as an example of efficient sorting algorithm ($O(n \cdot log(n))$ time complexity, with $log(n)$ being $log_2(n)$), and Insertion Sort, taken as an example of inefficient sorting algorithm.

Algorithm\Size	$\mathbf{10^4}$	$\mathbf{10^7}$
Insertion Sort	$\frac{n^2}{10^{11} \text{ instructions/s}} = \frac{(10^4)^2}{10^{11}} = 0.001$ s	16 minutes (approx.)
Merge Sort	$\frac{n \cdot \log(n)}{10^{11} \text{ instructions/s}} = \frac{10^4 \cdot \log(10^4)}{10^{11}} \approx 10^{-6}$ s	0.002 s (approx.)

Table 1: Estimated execution time for Insertion Sort and Merge Sort at different input sizes, assuming a speed of 10^{11} IPS (similar to modern CPUs [4])

For a smaller input, the execution times are so fast that the difference between Insertion Sort and Merge Sort goes undetected to a human observer.

On the other hand, for a bigger input size of 10^7 elements, we can see how algorithmic efficiency plays an important role in the resulting computation time: by using a more efficient algorithm, whose running time grows more slowly, the execution is 500,000 times faster!

This demonstrates how different algorithms devised to solve the same problem can differ dramatically in their efficiency, and how these differences can be much more significant than hardware.

3 The importance of code efficiency

As well described in [5], tasks can be split in three groups, based on their reasonable amount of computation time: "real-time", "quick decision" and "worth-the-wait". Real-time tasks are requested to have an execution time that is faster than we can notice, quick-decision tasks need to be executed in few seconds, while worth-the-wait tasks (complex problems) can have a computation time that is a few minutes, hours, or months. If a system is unable to uphold the reasonable computation time for these tasks, it is considered unfit for the purpose, especially for interactive applications, where delays could hinder the user's experience.

Efficient code and optimized algorithms make it possible to execute tasks in a reasonable amount of time (as we have seen in the previous section), and while doing so, they also lead to a better impact on the environment [6], an improvement which is much needed since computing is now using more energy than ever [7], and the trend is not likely to stop due to modern technologies [8].

4 Coming to a stagnation of hardware performance increase

In 1965, Gordon Moore stated that the number of transistors in a semiconductor would double every two years [9]. Although this was an empirical observation and not a mathematical theorem, the statement was named Moore's law, and since then it has set the pace for innovation within the semiconductor industry, guiding long-term planning and, hence, driving improvements in the performance of computing systems.

For as long as Moore's law has existed, we have been able to validate Wirth's law, which states that software becomes slower more rapidly than hardware becomes faster [10], since as hardware performance increases so does the demand for computing resources, with developers adding new features, very rarely optimized, that make our applications much heavier than before [11] but more appealing to users [12].
Another factor adding to this is the increasing level of abstraction of today's programming languages, that makes it easier to write a program but slows down computation time [13].

As in the mid 2010s Moore's law began to slow down due to physical challenges and is now considered dead by many, performance gains started to decline: we continue to increase the number of transistors in a processor, but at a much lower speed than before [14], translating into a low increase in hardware performance, that is coming to a stagnation (see Figure 1).

As long as Moore's law held true, the increase in software sluggishness didn't matter, a new generation of computers would arrive and speed up software that had become too slow. However, since hardware performance increase has slowed down, it now becomes even more important for software developers to focus on performance and efficiency of code during development, otherwise software inefficiency would definitely doom us to slow applications. [15] comes to exactly this conclusion and suggests how, in the post-Moore era, restructuring software by making it more efficient is the best solution for ensuring applications run quickly.

5 Conclusion

In conclusion, this essay has demonstrated through various arguments that code efficiency is the only way forward.

Through a comparison of sorting algorithms, we have seen how algorithmic efficiency is much more significant than hardware when considering how to achieve a lower computation time, and we have then analyzed how this is important since tasks have specific execution time constraints. Furthermore, we have asserted that code efficiency is also important to achieve a better impact on the environment.

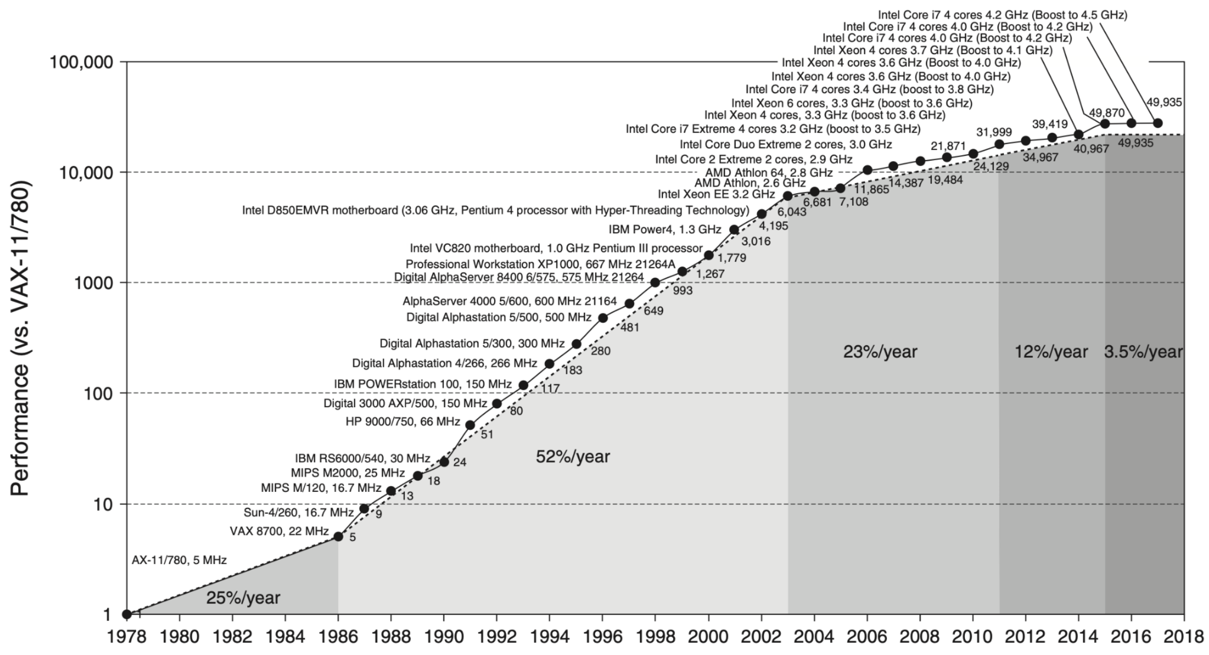

Figure 1: 'Growth in processor performance over 40 years', from [16]

Considering this, along with the stagnation in hardware performance improvements, we conclude that code efficiency is more crucial than ever, making it imperative for programmers to prioritize writing efficient code.

References

[1] Udi Manber and Peter Norvig, *"The power of the Apollo missions in a single Google search"*.

[2] Wikipedia, *"Introsort"*.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *"Introduction to Algorithms"*.

[4] Wikipedia, *"Instructions per second"*.

[5] David Harris-Birtill and Rose Harris-Birtill, *"Understanding computation time: A critical discussion of time as a computational performance metric"*.

[6] Loïc Lannelongue, Jason Grealey, Michael Inouye *"Green Algorithms: Quantifying the carbon emissions of computation"*.

[7] Frontier Group, Susan Rakov, Abigail Ham *"Fact file: Computing is using more energy than ever"*.

[8] Emma Strubell , Ananya Ganesh, Andrew McCallum *"Energy and Policy Considerations for Deep Learning in NLP"*.

[9] Gordon Mooore, *"Cramming more components onto integrated circuits"*.

[10] Niklaus Wirth, *"A Plea for Lean Software"*.

[11] KeyCDN, *"The Growth of Web Page Size"*.

[12] PCMag, *"Definition of software bloat"*.

[13] PCMag, *"Definition of abstraction layer"*.

[14] Audrey Woods, *"The Death of Moore's Law: What it means and what might fill the gap going forward"*.

[15] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, Tao B. Schardl, *"There's plenty of room at the Top: What will drive computer performance after Moore's law?"*.

[16] John L. Hennessy, David A. Patterson, "Computer Architecture: A Quantitative Approach".