

REVIEW SUMMARY

COMPUTER SCIENCE

There's plenty of room at the Top: What will drive computer performance after Moore's law?

Charles E. Leiserson, Neil C. Thompson*, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, Tao B. Schardl

BACKGROUND: Improvements in computing power can claim a large share of the credit for many of the things that we take for granted in our modern lives: cellphones that are more powerful than room-sized computers from 25 years ago, internet access for nearly half the world, and drug discoveries enabled by powerful supercomputers. Society has come to rely on computers whose performance increases exponentially over time.

Much of the improvement in computer performance comes from decades of miniaturization of computer components, a trend that was foreseen by the Nobel Prize-winning physicist Richard Feynman in his 1959 address, "There's Plenty of Room at the Bottom," to the American Physical Society. In 1975, Intel founder Gordon Moore predicted the regularity of this miniaturization trend, now called Moore's law, which, until recently, doubled the number of transistors on computer chips every 2 years.

Unfortunately, semiconductor miniaturization is running out of steam as a viable way to grow computer performance—there isn't much more room at the "Bottom." If growth

in computing power stalls, practically all industries will face challenges to their productivity. Nevertheless, opportunities for growth in computing performance will still be available, especially at the "Top" of the computing-technology stack: software, algorithms, and hardware architecture.

ADVANCES: Software can be made more efficient by performance engineering: restructuring software to make it run faster. Performance engineering can remove inefficiencies in programs, known as software bloat, arising from traditional software-development strategies that aim to minimize an application's development time rather than the time it takes to run. Performance engineering can also tailor software to the hardware on which it runs, for example, to take advantage of parallel processors and vector units.

Algorithms offer more-efficient ways to solve problems. Indeed, since the late 1970s, the time to solve the maximum-flow problem improved nearly as much from algorithmic advances as from hardware speedups. But progress on a given algorithmic problem occurs unevenly

and sporadically and must ultimately face diminishing returns. As such, we see the biggest benefits coming from algorithms for new problem domains (e.g., machine learning) and from developing new theoretical machine models that better reflect emerging hardware.

ON OUR WEBSITE

Read the full article at <https://dx.doi.org/10.1126/science.aam9744>

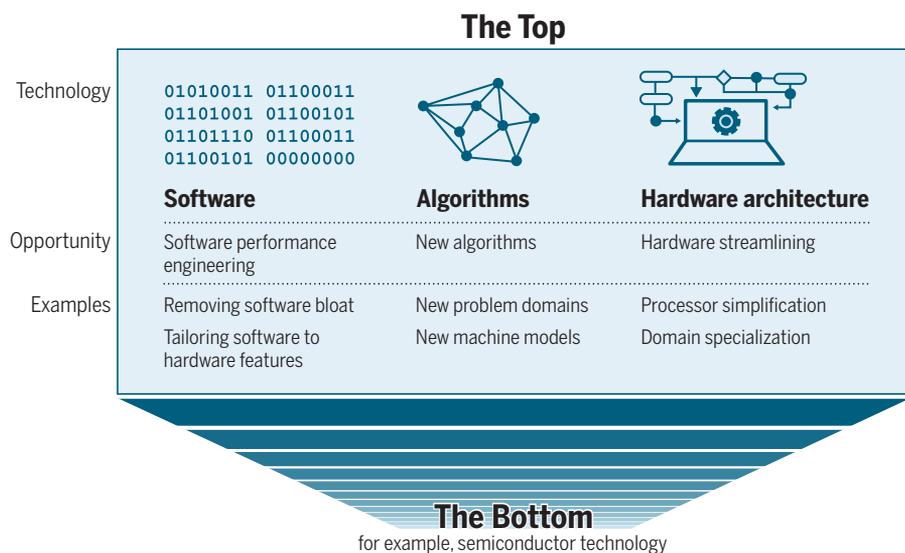
Hardware architectures can be streamlined—for instance, through processor simplification, where a complex processing core is replaced with a simpler core that requires fewer transistors. The freed-up transistor budget can then be redeployed in other ways—for example, by increasing the number of processor cores running in parallel, which can lead to large efficiency gains for problems that can exploit parallelism. Another form of streamlining is domain specialization, where hardware is customized for a particular application domain. This type of specialization jettisons processor functionality that is not needed for the domain. It can also allow more customization to the specific characteristics of the domain, for instance, by decreasing floating-point precision for machine-learning applications.

In the post-Moore era, performance improvements from software, algorithms, and hardware architecture will increasingly require concurrent changes across other levels of the stack. These changes will be easier to implement, from engineering-management and economic points of view, if they occur within big system components: reusable software with typically more than a million lines of code or hardware of comparable complexity. When a single organization or company controls a big component, modularity can be more easily re-engineered to obtain performance gains. Moreover, costs and benefits can be pooled so that important but costly changes in one part of the big component can be justified by benefits elsewhere in the same component.

OUTLOOK: As miniaturization wanes, the silicon-fabrication improvements at the Bottom will no longer provide the predictable, broad-based gains in computer performance that society has enjoyed for more than 50 years. Software performance engineering, development of algorithms, and hardware streamlining at the Top can continue to make computer applications faster in the post-Moore era. Unlike the historical gains at the Bottom, however, gains at the Top will be opportunistic, uneven, and sporadic. Moreover, they will be subject to diminishing returns as specific computations become better explored. ■

The list of author affiliations is available in the full article online.
*Corresponding author. Email: neil_t@mit.edu
Cite this article as C. E. Leiserson et al., *Science* 368, eaam9744 (2020). DOI: 10.1126/science.aam9744

Downloaded from <http://science.sciencemag.org/> on December 30, 2020



Performance gains after Moore's law ends. In the post-Moore era, improvements in computing power will increasingly come from technologies at the "Top" of the computing stack, not from those at the "Bottom," reversing the historical trend.

REVIEW

COMPUTER SCIENCE

There's plenty of room at the Top: What will drive computer performance after Moore's law?

Charles E. Leiserson¹, Neil C. Thompson^{1,2*}, Joel S. Emer^{1,3}, Bradley C. Kuzmaul^{1†}, Butler W. Lampson^{1,4}, Daniel Sanchez¹, Tao B. Schardl¹

The miniaturization of semiconductor transistors has driven the growth in computer performance for more than 50 years. As miniaturization approaches its limits, bringing an end to Moore's law, performance gains will need to come from software, algorithms, and hardware. We refer to these technologies as the "Top" of the computing stack to distinguish them from the traditional technologies at the "Bottom": semiconductor physics and silicon-fabrication technology. In the post-Moore era, the Top will provide substantial performance gains, but these gains will be opportunistic, uneven, and sporadic, and they will suffer from the law of diminishing returns. Big system components offer a promising context for tackling the challenges of working at the Top.

Over the past 50 years, the miniaturization of semiconductor devices has been at the heart of improvements in computer performance, as was foreseen by physicist Richard Feynman in his 1959 address (1) to the American Physical Society, "There's Plenty of Room at the Bottom." Intel founder Gordon Moore (2) observed a steady rate of miniaturization and predicted (3) that the number of transistors per computer chip would double every 2 years—a cadence, called Moore's law, that has held up considerably well until recently. Moreover, until about 2004, new transistors were not only smaller, they were also faster and more energy efficient (4), providing computers with ever more speed and storage capacity. Moore's law has driven economic progress ubiquitously.

Unfortunately, Feynman's "room at the bottom" is no longer plentiful. The *International Technology Roadmap for Semiconductors* [(5), p. 36] foresees an end to miniaturization, and Intel [(6), p. 14], a leader in microprocessor technology, has acknowledged an end to the Moore cadence. Indeed, Intel produced its 14-nm technology in 2014, but it stalled on producing its 10-nm technology, due in 2016, until 2019 (7). Although other manufacturers continued to miniaturize—for example, with the Samsung Exynos 9825 (8) and the Apple A13 Bionic (9)—they also failed to meet the Moore cadence. There isn't much more room at the bottom.

Why is miniaturization stalling? It's stalling because of fundamental physical limits—the

physics of materials changes at atomic levels—and because of the economics of chip manufacturing. Although semiconductor technology may be able to produce transistors as small as 2 nm (20 Å), as a practical matter, miniaturization may end around 5 nm because of diminishing returns (10). And even if semiconductor technologists can push things a little further, the cost of doing so rises precipitously as we approach atomic scales (11, 12).

In this review, we discuss alternative avenues for growth in computer performance after Moore's law ends. We believe that opportunities can be found in the higher levels of the computing-technology stack, which we refer to as the "Top." Correspondingly, by "Bottom" we mean the semiconductor technology that improved so dramatically during the Moore era. The layers of the computing stack harness the transistors and other semiconductor devices at the Bottom into useful computation at the Top to solve real-world problems. We divide the Top into three layers: (i) hardware architecture—programmable digital circuits that perform calculations; (ii) software—code that instructs the digital circuits what to compute; and (iii) algorithms—efficient problem-solving routines that organize a computation. We contend that even if device technologies at the Bottom cease to deliver performance gains, the Top will continue to offer opportunities.

Unlike Moore's law, which has driven up performance predictably by "lifting all boats," working at the Top to obtain performance will yield opportunistic, uneven, and sporadic gains, typically improving just one aspect of a particular computation at a time. For any given problem, the gains will suffer from the law of diminishing returns. In the long run, gains will depend on applying computing to new problems, as has been happening since the dawn of digital computers.

Working at the Top to obtain performance also differs from the Bottom in how it affects a computing system overall. The performance provided by miniaturization has not required substantial changes at the upper levels of the computing stack, because the logical behavior of the digital hardware, software, and data in a computation is almost entirely independent of the size of the transistors at the Bottom. As a result, the upper levels can take advantage of smaller and faster transistors with little or no change. By contrast—and unfortunately—many parts of the Top are dependent on each other, and thus when one part is restructured to improve performance, other parts must often adapt to exploit, or even tolerate, the changes. When these changes percolate through a system, it can take considerable human effort to correctly implement and test them, which increases both costs and risks. Historically, the strategies at the Top for improving performance coexisted with Moore's law and were used to accelerate particular applications that needed more than the automatic performance gains that Moore's law could provide.

Here, we argue that there is plenty of room at the Top, and we outline promising opportunities within each of the three domains of software, algorithms, and hardware. We explore the scale of improvements available in these areas through examples and data analyses. We also discuss why "big system components" will provide a fertile ground for capturing these gains at the Top.

Software

Software development in the Moore era has generally focused on minimizing the time it takes to develop an application, rather than the time it takes to run that application once it is deployed. This strategy has led to enormous inefficiencies in programs, often called software bloat. In addition, much existing software fails to take advantage of architectural features of chips, such as parallel processors and vector units. In the post-Moore era, software performance engineering—restructuring software to make it run faster—can help applications run more quickly by removing bloat and by tailoring software to specific features of the hardware architecture.

To illustrate the potential gains from performance engineering, consider the simple problem of multiplying two 4096-by-4096 matrices. Let us start with an implementation coded in Python, a popular high-level programming language. Here is the four-line kernel of the Python 2 code for matrix-multiplication:

```
for i in xrange(4096):
    for j in xrange(4096):
        for k in xrange(4096):
            C[i][j] += A[i][k] * B[k][j]
```

The code uses three nested loops and follows the method taught in basic linear-algebra

¹Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA.

²MIT Initiative on the Digital Economy, Cambridge, MA, USA. ³NVIDIA Research, Westford, MA, USA. ⁴Microsoft Research, Cambridge, MA, USA.

*Corresponding author. Email: neil_t@mit.edu

†Present address: Google, Cambridge, MA, USA.

Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that the version executes. “Absolute speedup” is time relative to Python, and “relative speedup,” which we show with an additional digit of precision, is time relative to the preceding line. “Fraction of peak” is GFLOPS relative to the computer’s peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

classes. It turns out, however, that this naïve code leaves much of the performance available on modern computers “on the table.” The code takes about 7 hours on a modern computer to compute the matrix product, as shown by the first row (version 1) in Table 1, achieving only 0.0006% of the peak performance of the machine. (Incidentally, Python 3 requires about 9 hours for the same computation.)

How can this naïve matrix-multiplication code be performance engineered? Simply choosing a more efficient programming language speeds up this calculation dramatically. For example, coding it in Java (version 2) produces a speedup of 10.8×, and coding it in C (version 3) produces an additional speedup of 4.4×, yielding an execution time that is 47 times faster than the original Python. This performance improvement comes from reducing the number of operations the program performs. In particular, Java and C avoid the extraneous work that Python does under the hood to make programming easier. The price for this performance gain is programmer productivity: Coding in C is more onerous than coding in Python, and Java lies somewhere in between.

Although switching languages gains a speedup of almost 50×, tailoring the matrix code to exploit specific features of the hardware makes it run an additional 1300 times faster. This gain comes from parallelizing the code to run on all 18 of the processing cores (version 4), exploiting the processor’s memory hierarchy (version 5), vectorizing the code (version 6), and using Intel’s special Advanced Vector Extensions (AVX) instructions (version 7). The final optimized code performs the task in only 0.41 s—more than 60,000 times faster than the 7 hours of the original Python code!

The point of this example is to illustrate the potential gains available from performance engineering naïvely coded software. In the particular case of matrix multiplication, a good programmer could avoid this programming effort by using optimized code from existing

software libraries. If she were writing code to solve a new problem, however, she would need to optimize the code herself. And although not every application can improve by nearly five orders of magnitude through performance engineering, most modern software systems contain ample opportunity for performance enhancement, especially if the codebase is large enough.

During the post-Moore era, it will become ever more important to make code run fast and, in particular, to tailor it to the hardware on which it runs. Modern computers provide architectural features designed to make code run fast. For example, versions 4 and 6 exploit parallelism, which is the ability of computers to perform multiple operations at the same time. Version 5 exploits locality, which is the computer’s ability to access data elements efficiently when they are collocated in memory (spatial locality) or have been accessed recently (temporal locality). Version 7 exploits both parallelism and locality through carefully coordinated use of Intel’s AVX instructions. As we shall see in the Hardware architecture section, architectures are likely to become increasingly heterogeneous, incorporating both general-purpose and special-purpose circuitry. To improve performance, programs will need to expose more parallelism and locality for the hardware to exploit. In addition, software performance engineers will need to collaborate with hardware architects so that new processors present simple and compelling abstractions that make it as easy as possible to exploit the hardware.

Beyond the tailoring of software to hardware is the question of bloat: Where does software bloat come from? Certainly, some bloat comes from trading off efficiency for other desirable traits, such as coding ease, as versions 1 to 3 of the matrix-multiplication code illustrate. Bloat also comes from a failure to tailor code to the underlying architecture, as versions 4 to 7 show. But much software

bloat arises from software-development strategies (13, 14), such as reduction.

The idea of reduction is this. Imagine that you are a programmer who has been given a problem *A* to solve (for example, distinguishing between a yes or no spoken response). You could write specialized code to solve *A* directly, but instead, you might notice that a related problem *B* has already been solved (existing speech-recognition software that understands many words, including yes and no). It will take you far less effort to solve *A* by converting it into a problem that can be solved with the existing code for *B*, that is, by reducing *A* to *B*.

Inefficiencies can arise both from the reduction itself (translating *A* to *B*) and from the generality of *B* (the solution to *B* is not tailored specifically to *A*). But the largest bloat arises from the compounding of reductions: reducing *A* to *B*, *B* to *C*, *C* to *D*, and so on. Even if each reduction achieves an impressive 80% efficiency, a sequence of two independent reductions achieves just $80\% \times 80\% = 64\%$. Compounding 20 more times yields an efficiency of less than 1%, or 100× in bloat.

Because of the accumulated bloat created by years of reductionist design during the Moore era, there are great opportunities to make programs run faster. Unfortunately, directly solving problem *A* using specialized software requires expertise both in the domain of *A* and in performance engineering, which makes the process more costly and risky than simply using reductions. The resulting specialized software to solve *A* is often more complex than the software that reduces *A* to *B*. For example, the fully optimized code in Table 1 (version 7) is more than 20 times longer than the source code for the original Python version (version 1).

Indeed, simple code tends to be slow, and fast code tends to be complicated. To create a world where it is easy to write fast code, application programmers must be equipped with the knowledge and skills to performance-engineer

their code, and productivity tools to assist must be improved considerably.

Abstractly, software performance engineering can be viewed as a simple process involving a single loop: (i) Measure the performance of program A . (ii) Make a change to program A to produce a hopefully faster program A' . (iii) Measure the performance of program A' . (iv) If A' beats A , set $A = A'$. (v) If A is still not fast enough, go to (ii). But today, our systems are sufficiently complicated that measurements must often be repeated many times to develop confidence that one version of a program outperforms another.

As hardware becomes increasingly specialized and heterogeneous (see the Hardware architecture section), high-performing code will become even more difficult to write. Because faster software will be increasingly important for better performance in the post-Moore era, however, the computing industry, researchers, and government should all be well motivated to develop performance-engineering technologies.

Algorithms

Algorithmic advances have already made many contributions to performance growth and will continue to do so in the future. A major goal is to solve a problem with less computational work. For example, by using Strassen's algorithm (15) for matrix multiplication, the highly optimized code in version 7 of Table 1 can be improved by about 10%. For some problems,

the gains can be much more impressive: The President's Council of Advisors on Science and Technology concluded in 2010 that "*performance gains due to improvements in algorithms have vastly exceeded even the dramatic performance gains due to increased processor speed*" (emphasis theirs) [(16), p. 71].

Because algorithm design requires human ingenuity, however, it is hard to anticipate advances. To illustrate the nature of algorithmic progress, consider the classical operations-research problem of finding the maximum flow in a network [(17), chap. 26], which can be used to model the movement of traffic in a road network, blood through the circulatory system, or electricity in a circuit. Linear programming is a straightforward way to solve the maximum-flow problem, but the 20 years between 1975 and 1995 saw a sequence of algorithmic innovations that greatly improved on it.

Figure 1 shows the progress in maximum-flow algorithms over time. The performance gain of the best algorithm has rivaled the gain due to Moore's law over the 38 years of the data (just over four orders of magnitude versus just under five), even though no new algorithm has improved the performance of this particular problem over the past 20 years. This example highlights three salient observations about algorithms: (i) Progress on a given algorithmic problem occurs unevenly and sporadically. (ii) The benefits from algorithmic

innovation can rival gains from Moore's law. (iii) The algorithmic improvements in solving any given problem must eventually diminish.

Because this example focuses on a well-known problem, however, it misses a key aspect of how algorithmic performance engineering can speed up computing: by providing efficient solutions to new problems. For example, more than a quarter of the 146 papers at the 2016 Association for Computing Machinery (ACM) Symposium on Discrete Algorithms focus on problems that had not previously been studied algorithmically. Consequently, although research on old problems may still yield marginal gains, much of the progress in algorithms will come from three sources: (i) attacking new problem domains, (ii) addressing scalability concerns, and (iii) tailoring algorithms to take advantage of modern hardware. We discuss each source in turn.

New problem domains continually create a need for new algorithms. Domains such as machine learning, social networks, security, robotics, game theory, sensor networks, and video coding were tiny or nonexistent 30 years ago but are now economically important enough to demand efficient algorithms. Many companies have gained a competitive advantage thanks to algorithms. Google's PageRank algorithm (18) made its World Wide Web search superior, and the auction algorithms of Google AdWords (19), which allow advertisers to bid for display space based on users' search terms, made it highly profitable. Content-delivery networks, which delivered more than half of internet traffic in 2016 (20), depend on efficient algorithms to avoid congestion. Many sciences also depend on good algorithms. For example, DNA sequencing in computational biology depends on efficient dynamic-programming algorithms (21).

Moore's law has enabled today's high-end computers to store over a terabyte of data in main memory, and because problem sizes have grown correspondingly, efficient algorithms are needed to make solutions affordable. Sublinear algorithms (22, 23) provide one example of how to address problems of scale. For instance, to find the median of a trillion numbers, it takes at least a trillion memory operations just to read the input data. But many problems do not need the exact median and work perfectly well with only a good estimate of the median. For these problems, we can instead extract a random sample of, say, a million numbers and compute the median of that sample. The result is a highly accurate estimate of the true median that can be computed a million times faster. The field of algorithms is full of strategies for dealing with scalability.

Tailoring an algorithm to exploit modern hardware can make it much faster (Table 1). Nevertheless, most algorithms today are still

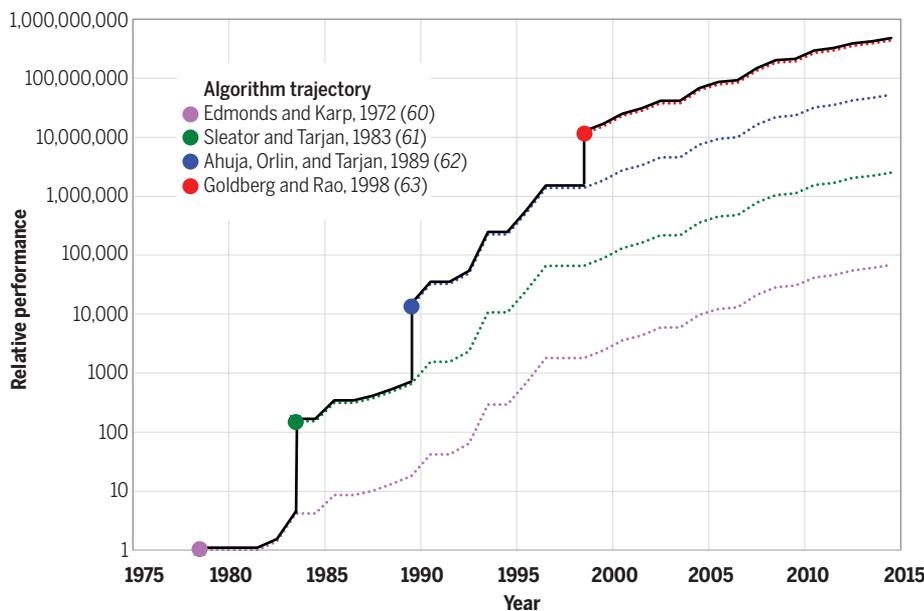


Fig. 1. Major algorithmic advances in solving the maximum-flow problem on a graph with $n = 10^{12}$ vertices and $m = n^{11}$ edges. The vertical axis shows how many problems (normalized to the year 1978) could theoretically be solved in a fixed time on the best microprocessor system available in that year. Each major algorithm is shown as a circle in the year of its invention, except the first, which was the best algorithm in 1978. The dotted trajectory shows how faster computers [as measured by SPECint scores in Stanford's CPU database (56)] make each algorithm faster over time. The solid black line shows the best algorithm using the best computer at any point in time. Each algorithm's performance is measured by its asymptotic complexity, as described in the Methods.

designed using the serial random-access machine model (24) originally developed in the 1960s and 1970s, which assumes that a processor can do only one operation at a time and that the cost to access any part of the memory is the same. Such algorithms often use modern hardware inefficiently because they underutilize the machine's many parallel-processing cores and vector units, each of which can perform many operations per clock cycle, and they fail to exploit caching, which can speed up data accesses by two orders of magnitude.

Although algorithms research has developed mathematical models for salient features of modern computers, such as parallel and vector processing (25–32) and cache hierarchies (33–35), a substantial gap between algorithm and implementation remains. Part of the problem is that each model tends to address just one aspect—such as parallelism, vector units, or caching—and yet tailoring an algorithm to a modern computer requires an understanding of all of them. Moreover, in an effort to gain every bit of performance, some hardware features—such as simultaneous multithreading, dynamic voltage and frequency scaling, direct-mapped caches, and various special-purpose instructions—actually make it more difficult to tailor algorithms to hardware, because they cause variability and unpredictability that simple theoretical models cannot easily capture.

One possible solution is autotuning (36, 37), which searches a parametrized space of possible implementations to find the fastest one. With modern machine learning, it may even be possible to include implementations that differ by more than the values of a few parameters. Unfortunately, autotuning and machine learning tend to be too time consuming to ask that every algorithm incur this large up-front cost. Furthermore, these approaches actually make algorithm design harder, because the designer cannot easily understand the ramifications of a design choice. In the post-Moore era, it will be essential for algorithm designers and hardware architects to work together to find simple abstractions that designers can understand and that architects can implement efficiently.

Hardware architecture

Historically, computer architects used more and more transistors to make serial computations run faster, vastly increasing the complexity of processing cores, even though gains in performance suffered from diminishing returns over time (38). We argue that in the post-Moore era, architects will need to adopt the opposite strategy and focus on hardware streamlining: implementing hardware functions using fewer transistors and less silicon area.

As we shall see, the primary advantage of hardware streamlining comes from providing additional chip area for more circuitry to operate in parallel. Thus, the greatest benefit accrues to applications that have ample parallelism. Indeed, the performance of hardware for applications without much parallelism has already stagnated. But there is plenty of parallelism in many emerging application domains, such as machine learning, graphics, video and image processing, sensory computing, and signal processing. Computer architects should be able to design streamlined architectures to provide increasing performance for these and other domains for many years after Moore's law ends.

We can use historical data to observe the trend of architectural reliance on parallelism. Figure 2 plots three sets of benchmark data for microprocessors: SPECint performance (black squares and gray diamonds), SPECint-rate performance (black, orange, blue, and red squares), and microprocessor clock frequency (green dots). As the green dots in the figure show, clock speed increased by a factor of more than 200 from 1985 to 2005, when it plateaued owing to the end of Dennard scaling, which we shall discuss shortly. Driven by increasing clock speed and other architec-

tural changes during the Dennard-scaling era, microprocessor performance rapidly improved, as measured by the SPECint and SPECint-rate benchmarks (black squares), which aim to model computer performance on typical user workloads (39). The SPECint benchmark consists of mostly serial code, whereas the SPECint-rate benchmark is parallel. The two benchmarks perform the same on single-processor computers. But after 2004, as machines added multiple cores and other forms of explicit parallelism, the two diverge. Indeed, the performance of parallel applications on the best-performing chips in each year (colored squares) grew by a factor of 30 from 2004 to 2015, improving on average by about a factor of two every 2 years. By contrast, over the same time period, the largely serial SPECint benchmark (gray diamonds) scaled up by only a factor of three.

Besides parallelism, an application needs locality to benefit from streamlining. As an example, when data are transferred from external dynamic random access memory (DRAM) memory chips to a processing chip, it should be used multiple times before being transferred back. For an application with little locality, increasing parallelism causes traffic to off-chip memory to increase proportionally and

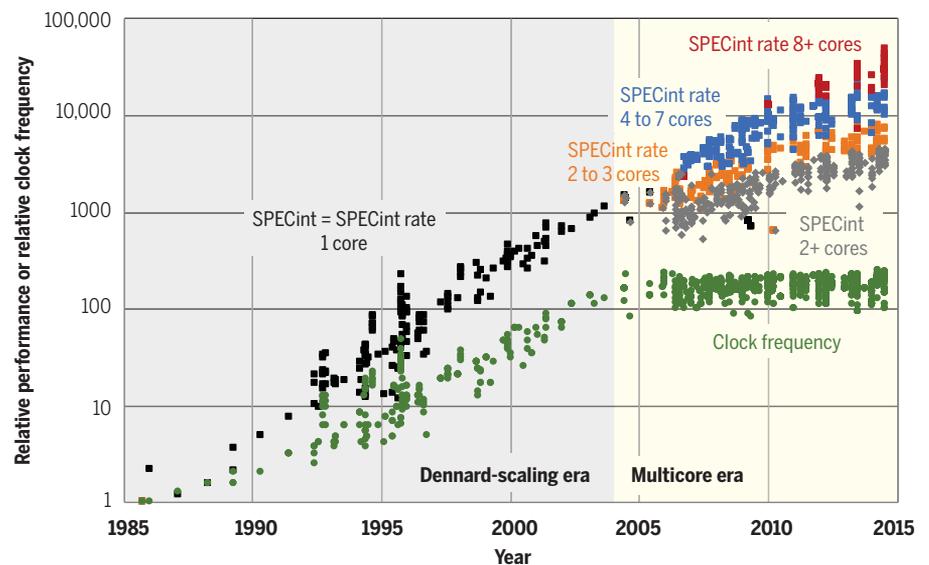


Fig. 2. SPECint (largely serial) performance, SPECint-rate (parallel) performance, and clock-frequency scaling for microprocessors from 1985 to 2015, normalized to the Intel 80386 DX microprocessor in 1985. Microprocessors and their clock frequencies were obtained from the Stanford CPU database (56). Microprocessor performance is measured in terms of scaled performance scores on the SPECint and SPECint-rate performance benchmarks obtained from (39). (See Methods for details.) Black squares identify single-core processors, for which SPECint and SPECint-rate benchmark performances are the same. Orange, blue, and red squares plot the SPECint-rate benchmark performance of various multicore processors, where orange squares identify processors with two to three cores, blue squares identify processors with four to seven cores, and red squares identify processors with eight or more cores. The gray diamonds plot the SPECint benchmark performance on multicore processors. The round green dots plot processor clock frequencies (also normalized to the Intel 80386). The gray background highlights the Dennard-scaling era (nominally up to 2004), and the white background highlights the multicore era (beyond 2004).

eventually exceeds the bandwidth of its channel to memory, so that the application is memory bound. For an application with good locality, however, as parallelism increases, the amount of off-chip memory traffic increases much more slowly, enabling all the chip's computing engines to do useful work without idling. Fortunately, many important application domains contain plenty of both locality and parallelism.

Hardware streamlining can exploit locality in other ways, especially for domain-specific processors, which we shall discuss shortly. For example, explicit data orchestration (40) exploits locality to increase the efficiency with which data are moved throughout the memory hierarchy [(41), chap. 4]. On-chip interconnects can become simpler and consume less power and area if the application using them contains locality. For example, systolic arrays (42) can perform matrix computations more efficiently using an area-efficient mesh interconnect than a general-purpose interconnect.

Although hardware will increase in capability because of streamlining, we do not think that average clock speed will increase after Moore's law ends, and it may in fact diminish slightly. Figure 2 shows that clock speed plateaued in 2005, when microprocessor design became power constrained. Before 2004, computer architects found ways to increase clock frequency without hitting hard power limits. Dennard scaling—reducing voltage as clock frequency increased—allowed processors to run faster without increasing power usage. (In practice, processor manufacturers often increased clock frequency without reducing voltage proportionally, which did increase chip power.) Since 2004, however, Moore's law has provided many more transistors per chip, but because the ability to power them has not grown appreciably (43), architects have been forced to innovate just to prevent clock rates from falling. Slightly lower clock frequency and supply voltage reduce the power per transistor enough that substantially more circuitry can run in parallel. If the workload has enough parallelism, the added computing more than compensates for the slower clock. Serial applications may see somewhat worse performance, but cleverness can reduce this cost. For example, Intel's "Turbo" mode [(41), p. 28], runs the clock faster when fewer cores are active. (Other techniques to reduce transistor switching include using more transistors in caches, power-gating unused circuitry, and minimizing signal switching.)

Now that designers have embraced parallelism, the main question will be how to streamline processors to exploit application parallelism. We expect two strategies to dominate: processor simplification and domain specialization.

Processor simplification (44) replaces a complex processing core with a simpler core that requires fewer transistors. A modern core contains many expensive mechanisms to make serial instruction streams run faster, such as speculative execution [(41), section 3.6], where the hardware guesses and pursues future paths of code execution, aborting and reexecuting if the guess is wrong. If a core can be simplified to occupy, say, half as many transistors, then twice as many cores can fit on the chip. For this trade-off to be worthwhile, the workload must have enough parallelism that the additional cores are kept busy, and the two simplified cores must do more useful computing than the single complex one.

Domain specialization (11, 43, 45) may be even more important than simplification. Hardware that is customized for an application domain can be much more streamlined and use many fewer transistors, enabling applications to run tens to hundreds of times faster (46). Perhaps the best example today is the graphics-processing unit (GPU) [(41), section 4.4], which contains many parallel "lanes" with streamlined processors specialized to computer graphics. GPUs deliver much more performance on graphics computations, even though their clocks are slower, because they can exploit much more parallelism. GPU logic integrated into laptop microprocessors grew from 15 to 25% of chip area in 2010 to more than 40% by 2017 (see Methods), which shows the importance of GPU accelerators. Moreover, according to the Top 500 website, which tracks high-performance computing technology, only about 10% of supercomputers that were added to the Top 500 list in the year 2012 contained accelerators (often GPUs), but by 2017, that share had grown to 38% (12). Hennessy and Patterson (45) foresee a move from general-purpose toward domain-specific architectures that run small compute-intensive kernels of larger systems for tasks such as object recognition or speech understanding. The key requirement is that the most expensive computations in the application domain have plenty of parallelism and locality.

A specialized processor is often first implemented as an attached device to a general-purpose processor. But the forces that encourage specialization must be balanced with the forces that demand broadening: expanding the functionality of the specialized processor to make it more autonomous from the general-purpose processor and more widely useful to other application domains (47).

The evolution of GPUs demonstrates this trade-off. GPUs were originally developed specifically for rendering graphics, and as a result, GPUs are next to useless for many other computational tasks, such as compiling computer programs or running an operating system. But

GPUs have nevertheless broadened to be handy for a variety of nongraphical tasks, such as linear algebra. Consider the matrix-multiplication problem from the Software section. An Advanced Micro Devices (AMD) FirePro S9150 GPU (48) can produce the result in only 70 ms, which is 5.4 times faster than the optimized code (version 7) and a whopping 360,000 times faster than the original Python code (version 1).

As another example of the trade-off between broadening and specialization, GPUs were crucial to the "deep-learning" revolution (49), because they were capable of training large neural networks that general-purpose processors could not train (50, 51) fast enough. But specialization has also succeeded. Google has developed a tensor-processing unit (TPU) (52) specifically designed for deep learning, embracing special-purpose processing and eschewing the broader functionality of GPUs.

During the Moore era, specialization usually yielded to broadening, because the return on investment for developing a special-purpose device had to be amortized by sales over the limited time before Moore's law produced a general-purpose processor that performs just as well. In the post-Moore era, however, we expect to see more special-purpose devices, because they will not have comparably performing general-purpose processors right around the corner to compete with. We also expect a diversity of hardware accelerators specialized for different application domains, as well as hybrid specialization, where a single device is tailored for more than one domain, such as both image processing and machine learning for self-driving vehicles (53). Cloud computing will encourage this diversity by aggregating demand across users (12).

Big components

In the post-Moore era, performance engineering, development of algorithms, and hardware streamlining will be most effective within big system components (54). A big component is reusable software with typically more than a million lines of code, hardware of comparable complexity, or a similarly large software-hardware hybrid. This section discusses the technical and economic reasons why big components are a fertile ground for obtaining performance at the Top.

Changes to a system can proceed without much coordination among engineers, as long as the changes do not interfere with one another. Breaking code into modules and hiding its implementation behind an interface make development faster and software more robust (55). Modularity aids performance engineering, because it means that code within a module can be improved without requiring the rest of the system to adapt. Likewise, modularity aids in hardware streamlining, because the hardware can be restructured

without affecting application programming interfaces (APIs). Performance engineering and hardware streamlining that do not require coordination already occur this way, and we expect them to continue to do so.

Many of the most valuable future opportunities for improving performance will not arise locally within a single module, however, but from broad-based and systematic changes across many modules affecting major parts of a system. For example, because so much software is reductionist, the larger a system, the greater the opportunity for huge performance gains when layer upon layer of reductions are replaced by a lean and direct implementation. But large-scale reengineering that affects many modules requires the coordination of many engineers, which is costly and risky.

Big system components (web browsers, database systems, operating systems, microprocessors, GPUs, compilers, disk-storage systems, and so on) offer both technical opportunities for making applications run fast and a context where it is economically viable to take advantage of them. As an example of the types of changes that can be made within a big component, consider a microprocessor. An instruction-set architecture (ISA) is the interface by which software tells the processor what to do. Manufacturers routinely make major internal changes to improve performance without changing the ISA so that old software continues to run correctly. For example, the Intel 8086 released in 1978 had 29,000 transistors (56), whereas the 22-core Xeon E5-2699 v4, released in 2016, has about 248,000 times more transistors (57) that produce more than a million times better performance, according to SPECint rate (39). In that time, the ISA has grown by less than a factor of four (58), and old programs continue to work even as the interface adds new functionality.

Big components provide good opportunities for obtaining performance, but opportunity alone is not enough. To outweigh the costs and risks of making changes, an economic incentive must exist. If a commercial enterprise or a nonprofit organization owns a big component, it can justify the investment to enhance performance because it reaps the benefit when the job is done. Single ownership also helps with coordination costs. If many parties must agree to make a change, but it takes only a few to veto it, then it can be hard for change to happen. Even the fear of high coordination costs can block change in a large codebase if too many parties are involved. But when a single entity owns a big component, it has the power to make vast changes and pool the costs and benefits. It can choose to reengineer as many modules as it can justify economically and coordinate with the outside world only at the big component's interface.

Conclusions

As miniaturization wanes, the silicon-fabrication improvements at the Bottom will no longer provide the predictable, broad-based gains in computer performance that society has enjoyed for more than 50 years. Performance-engineering of software, development of algorithms, and hardware streamlining at the Top can continue to make computer applications faster in the post-Moore era, rivaling the gains accrued over many years by Moore's law. Unlike the historical gains at the Bottom, however, the gains at the Top will be opportunistic, uneven, sporadic, and subject to diminishing returns as problems become better explored. But even where opportunities exist, it may be hard to exploit them if the necessary modifications to a component require compatibility with other components. Big components can allow their owners to capture the economic advantages from performance gains at the Top while minimizing external disruptions.

In addition to the potential at the Top, nascent technologies—such as 3D stacking, quantum computing, photonics, superconducting circuits, neuromorphic computing, and graphene chips—might provide a boost from the Bottom. At the moment, these technologies are in their infancy and lack the maturity to compete with today's silicon-based semiconductor technology in the near future (59). Although we applaud investment in these technologies at the Bottom because of their long-term potential, we view it as far more likely that, at least in the near term, performance gains for most applications will originate at the Top.

Methods

Table 1

Each running time is the minimum of five runs on an Amazon AWS c4.8xlarge spot instance, a dual-socket Intel Xeon E5-2666 v3 system with a total of 60 gibibytes of memory. Each Xeon is a 2.9-GHz 18-core CPU with a shared 25-mebibyte L3-cache. Each processor core has a 32-kibibyte (KiB) private L1-data-cache and a 256-KiB private L2-cache. The machine was running Fedora 22, using version 4.0.4 of the Linux kernel. The Python version was executed using Python 2.7.9. The Java version was compiled and run using OpenJDK version 1.8.0_51. All other versions were compiled using GNU Compiler Collection (GCC) 5.2.1 20150826.

Figure 1

Each curve models how hardware improvements grow the performance of a fixed maximum-flow algorithm, starting from the year the algorithm was published. The performance of each algorithm was measured as the number of graphs of $n = 10^{12}$ vertices, $m = n^{1.1} \sim 15.8 \times 10^{12}$ edges, and 64-bit integer capacities that can be solved in a fixed amount of time, normalized such that the first point starts at 1. We

calculated the performance of each algorithm based on its asymptotic complexity in terms of big- O notation, assuming that the constant hidden by the big- O is 1. We excluded approximate algorithms from consideration, because these algorithms do not necessarily return the same qualitative answer. We also excluded algorithms whose stated asymptotic bounds ignore logarithmic factors to simplify the performance comparisons. We identified four maximum-flow algorithms developed since 1978 that each delivered a performance improvement of more than a factor of four over its predecessor: Edmonds and Karp's $O(m^2 \log U)$ algorithm (60); Sleator and Tarjan's $O(mn \log n)$ algorithm (61); Ahuja, Orlin, and Tarjan's $O(mn \log(n\sqrt{\log U}/m + 2))$ algorithm (62); and Goldberg and Rao's $O(n^{2/3} m \log(n^2/m) \log U)$ algorithm (63). Each curve plots the best performance achieved by any processor up to that date, on the basis of scaled MIPS (million instructions per second) estimates and SPECint scores recorded in Stanford's CPU database (56). These processor performance measurements are normalized using the same method as in Fig. 2.

Figure 1 ignores the constant factors in the performance of maximum-flow algorithms because comparably engineered implementations of these algorithms are not available. Nevertheless, the effects from changes in constant factors between algorithms can be inferred from the plot. For example, if a constant factor were 10 times larger between one algorithm and a later one, then the curve for the later algorithm would be one order of magnitude lower (i.e., one division less on the y axis).

Figure 2

Performance measurements for different processors were gathered from Stanford's CPU database (56). For a processor released before 1992, its performance is measured as the recorded estimate of how many MIPS that processor can perform. For subsequent processors, each performance measurement corresponds to the best recorded score for that processor on either the SPECint1992, SPECint1995, SPECint2000, or SPECint2006 benchmarks. Performance for multicore processors is measured using both standard SPECint scores (gray diamonds) and in terms of throughput (multicolored squares), using SPECint2006 rate scores. All SPECint performance scores were obtained from (39). The date for an Intel processor released after 2004 is plotted as the first day in the quarter when that processor was released, unless a more precise release date was found. Scores for different SPECint benchmarks were normalized by scaling factors, which were computed from the geometric-mean ratios of scores for consecutive versions of SPECint for processors that were evaluated on both versions. The MIPS estimates were normalized with

the SPECint scores such that a score of 1 on SPECint1992 corresponds to 1 MIPS.

GPU logic integrated into laptop microprocessors

We obtained, from WikiChip (57), annotated die photos for Intel microprocessors with GPUs integrated on die, which began in 2010 with Sandy Bridge. We measured the area in each annotated photo dedicated to a GPU and calculated the ratio of this area to the total area of the chip. Intel's quad-core chips had approximately the following percentage devoted to the GPU: Sandy Bridge (18%), Ivy Bridge (33%), Haswell (32%), Skylake (40 to 60%, depending on version), Kaby Lake (37%), and Coffee Lake (36%). Annotated die photos for Intel microarchitectures newer than Coffee Lake were not available and therefore not included in the study. We did not find enough information about modern AMD processors to include them in this study.

REFERENCES AND NOTES

- R. P. Feynman, There's plenty of room at the bottom. *Eng. Sci.* **23**, 22–36 (1960).
- G. E. Moore, Cramping more components onto integrated circuits. *Electronics* **38**, 1–4 (1965).
- G. E. Moore, "Progress in digital integrated electronics" in *International Electron Devices Meeting Technical Digest* (IEEE, 1975), pp. 11–13.
- R. H. Dennard et al., Design of ion-implanted MOSFET's with very small physical dimensions. *JSSC* **9**, 256–268 (1974).
- ITRS, International Technology Roadmap for Semiconductors 2.0, executive report (2015); www.semiconductors.org/wp-content/uploads/2018/06/0_2015-ITRS-2.0-Executive-Report-1.pdf.
- Intel Corporation, Form 10K (annual report). SEC filing (2016); www.sec.gov/Archives/edgar/data/50863/00005086316000105/a10kdocument12262015q4.htm.
- I. Cutress, Intel's 10nm Cannon Lake and Core i3-8121U deep dive review (2019); www.anandtech.com/show/13405/intel-10nm-cannon-lake-and-core-i3-8121u-deep-dive-review.
- K. Hinum, Samsung Exynos 9825 (2019); www.notebookcheck.net/Samsung-Exynos-9825-SoC-Benchmarks-and-Specs.432496.0.html.
- K. Hinum, Apple A13 Bionic (2019); www.notebookcheck.net/Apple-A13-Bionic-SoC.434834.0.html.
- R. Merritt, "Path to 2 nm may not be worth it." *EE Times*, 23 March 2018; www.eetimes.com/document.asp?doc_id=1333109.
- R. Colwell, "The chip design game at the end of Moore's Law," presented at Hot Chips, Palo Alto, CA, 25 to 27 August 2013.
- N. C. Thompson, S. Spanuth, The decline of computers as a general-purpose technology: Why deep learning and the end of Moore's Law are fragmenting computing. SSRN 3287769 [Preprint], 20 November 2019; doi: [10.2139/ssrn.3287769](https://doi.org/10.2139/ssrn.3287769)
- J. Larus, Spending Moore's dividend. *Commun. ACM* **52**, 62–69 (2009). doi: [10.1145/1506409.1506425](https://doi.org/10.1145/1506409.1506425)
- G. Xu, N. Mitchell, M. Arnold, A. Rountev, G. Sevitsky, "Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications" in *FoSER '10: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (ACM, 2010), pp. 421–426.
- V. Strassen, Gaussian elimination is not optimal. *Numer. Math.* **13**, 354–356 (1969). doi: [10.1007/BF02165411](https://doi.org/10.1007/BF02165411)
- President's Council of Advisors on Science and Technology, "Designing a digital future: Federally funded research and development in networking and information technology" (Technical report, Executive Office of the President, 2010); www.cis.upenn.edu/~mkearns/papers/nitrd.pdf.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (MIT Press, ed. 3, 2009).
- S. Brin, L. Page, The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.* **30**, 107–117 (1998). doi: [10.1016/S0169-7552\(98\)00110-X](https://doi.org/10.1016/S0169-7552(98)00110-X)
- A. Mehta, A. Saberi, U. Vazirani, V. Vazirani, Adwords and generalized online matching. *J. Assoc. Comput. Mach.* **54**, 22 (2007). doi: [10.1145/1284320.1284321](https://doi.org/10.1145/1284320.1284321)
- Cisco Systems, Inc., "Cisco visual networking index (VNI): Complete forecast update, 2017–2022." Presentation 1465272001663118, Cisco Systems, Inc., San Jose, CA, December 2018; https://web.archive.org/web/20190916132155/https://www.cisco.com/c/dam/m/en_us/network-intelligence/service-provider/digital-transformation/knowledge-network-webinars/pdfs/1211_BUSINESS_SERVICES_CKN_PDF.pdf.
- D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology* (Cambridge Univ. Press, 1997).
- R. Kumar, R. Rubinfeld, Algorithms column: Sublinear time algorithms. *SIGACT News* **34**, 57–67 (2003). doi: [10.1145/954092.954103](https://doi.org/10.1145/954092.954103)
- R. Rubinfeld, A. Shapira, Sublinear time algorithms. *SIDMA* **25**, 1562–1588 (2011). doi: [10.1137/100791075](https://doi.org/10.1137/100791075)
- A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley Publishing Company, 1974).
- R. L. Graham, Bounds for certain multiprocessing anomalies. *Bell Syst. Tech. J.* **45**, 1563–1581 (1966). doi: [10.1002/j.1538-7305.1966.tb01709.x](https://doi.org/10.1002/j.1538-7305.1966.tb01709.x)
- R. P. Brent, The parallel evaluation of general arithmetic expressions. *J. Assoc. Comput. Mach.* **21**, 201–206 (1974). doi: [10.1145/321812.321815](https://doi.org/10.1145/321812.321815)
- S. Fortune, J. Wyllie, "Parallelism in random access machines" in *STOC '78: Proceedings of the 10th Annual ACM Symposium on Theory of Computing* (ACM, 1978), pp. 114–118.
- R. M. Karp, V. Ramachandran, "Parallel algorithms for shared-memory machines" in *Handbook of Theoretical Computer Science: Volume A, Algorithms and Complexity* (MIT Press, 1990), chap. 17, pp. 869–941.
- G. E. Blelloch, *Vector Models for Data-Parallel Computing* (MIT Press, 1990).
- L. G. Valiant, A bridging model for parallel computation. *Commun. ACM* **33**, 103–111 (1990). doi: [10.1145/79173.79181](https://doi.org/10.1145/79173.79181)
- D. Culler et al., "LogP: Towards a realistic model of parallel computation" in *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (ACM, 1993), pp. 1–12.
- R. D. Blumofe, C. E. Leiserson, Space-efficient scheduling of multithreaded computations. *SIAM J. Comput.* **27**, 202–229 (1998). doi: [10.1137/S0097539793259471](https://doi.org/10.1137/S0097539793259471)
- J. S. Vitter, Algorithms and data structures for external memory. *Found. Trends Theor. Comput. Sci.* **2**, 305–474 (2008). doi: [10.1561/0400000004](https://doi.org/10.1561/0400000004)
- J.-W. Hong, H. T. Kung, "I/O complexity: The red-blue pebble game" in *STOC '81: Proceedings of the 13th Annual ACM Symposium on Theory of Computing* (ACM, 1981), pp. 326–333.
- M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran, "Cache-oblivious algorithms" in *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science* (IEEE, 1999), pp. 285–297.
- M. Frigo, A fast Fourier transform compiler. *ACM SIGPLAN Not.* **34**, 169–180 (1999). doi: [10.1145/301631.301661](https://doi.org/10.1145/301631.301661)
- J. Ansel et al., "OpenTuner: An extensible framework for program autotuning" in *PACT '14: 2014 23rd International Conference on Parallel Architecture and Compilation Techniques* (ACM, 2014), pp. 303–316.
- S. Borkar, "Thousand core chips: A technology perspective" in *DAC '07: Proceedings of the 44th Annual Design Automation Conference* (ACM, 2007), pp. 746–749.
- Standard Performance Evaluation Corporation, SPEC CPU 2006 (2017); www.spec.org/cpu2006.
- M. Pellauer et al., "Buflets: An efficient and composable storage idiom for explicit decoupled data orchestration" in *ASPLOS '19: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems* (ACM, 2019), pp. 137–151.
- J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann, ed. 6, 2019).
- H. T. Kung, C. E. Leiserson, "Systolic arrays (for VLSI)" in *Sparse Matrix Proceedings 1978*, I. S. Duff, G. W. Stewart, Eds. (SIAM, 1979), pp. 256–282.
- M. B. Taylor, "Is dark silicon useful?: Harnessing the four horsemen of the coming dark silicon apocalypse" in *DAC '12: Proceedings of the 49th Annual Design Automation Conference* (ACM, 2012), pp. 1131–1136.
- A. Agarwal, M. Levy, "The kill rule for multicore" in *DAC '07: Proceedings of the 44th Annual Design Automation Conference* (ACM, 2007), pp. 750–753.
- J. L. Hennessy, D. A. Patterson, A new golden age for computer architecture. *Commun. ACM* **62**, 48–60 (2019). doi: [10.1145/3282307](https://doi.org/10.1145/3282307)
- R. Hameed et al., "Understanding sources of inefficiency in general-purpose chips" in *ISCA '10: Proceedings of the 37th Annual International Symposium on Computer Architecture* (ACM, 2010), pp. 37–47.
- T. H. Myer, I. E. Sutherland, On the design of display processors. *Commun. ACM* **11**, 410–414 (1968). doi: [10.1145/363347.363368](https://doi.org/10.1145/363347.363368)
- Advanced Micro Devices Inc., FirePro S9150 Server GPU Datasheet (2014); <https://usermanual.wiki/Document/AMDFirePROS9150DataSheet.2051023599>.
- A. Krizhevsky, I. Sutskever, G. E. Hinton, "Imagenet classification with deep convolutional neural networks" in *Advances in Neural Information Processing Systems 25 (NIPS 2012)*, F. Pereira, C. J. C. Burges, L. Bottou, Eds. (Curran Associates, 2012).
- D. C. Ciresan, U. Meier, L. M. Gambardella, J. Schmidhuber, Deep, big, simple neural nets for handwritten digit recognition. *Neural Comput.* **22**, 3207–3220 (2010). doi: [10.1162/NECO_a_00052](https://doi.org/10.1162/NECO_a_00052); pmid: [20858131](https://pubmed.ncbi.nlm.nih.gov/20858131/)
- R. Raina, A. Madhavan, A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors" in *ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning* (ACM, 2009), pp. 873–880.
- N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit" in *ISCA '17: Proceedings of the 44th Annual International Symposium on Computer Architecture* (ACM, 2017).
- D. Shapiro, NVIDIA DRIVE Xavier, world's most powerful SoC, brings dramatic new AI capabilities (2018); <https://blogs.nvidia.com/blog/2018/01/07/drive-xavier-processor/>.
- B. W. Lampson, "Software components: Only the giants survive" in *Computer Systems: Theory, Technology, and Applications*, A. Herbert, K. S. Jones, Eds. (Springer, 2004), chap. 20, pp. 137–145.
- D. L. Parnas, On the criteria to be used in decomposing systems into modules. *Commun. ACM* **15**, 1053–1058 (1972). doi: [10.1145/361598.361623](https://doi.org/10.1145/361598.361623)
- A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, M. Horowitz, CPU DB: Recording microprocessor history. *Queue* **10**, 10–27 (2012). doi: [10.1145/2181796.2181798](https://doi.org/10.1145/2181796.2181798)
- WikiChip LLC, WikiChip (2019); <https://en.wikichip.org/>.
- B. Lopes, R. Auler, R. Azevedo, E. Borin, "ISA aging: A X86 case study," presented at WIVOSCA 2013: Seventh Annual Workshop on the Interaction amongst Virtualization, Operating Systems and Computer Architecture, Tel Aviv, Israel, 23 June 2013.
- H. Khan, D. Hounshell, E. R. H. Fuchs, Science and research policy at the end of Moore's law. *Nat. Electron.* **1**, 14–21 (2018). doi: [10.1038/s41928-017-0005-9](https://doi.org/10.1038/s41928-017-0005-9)
- J. Edmonds, R. M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems. *J. Assoc. Comput. Mach.* **19**, 248–264 (1972). doi: [10.1145/321694.321699](https://doi.org/10.1145/321694.321699)
- D. D. Sleator, R. E. Tarjan, A data structure for dynamic trees. *JCSS* **26**, 362–391 (1983).
- R. K. Ahuja, J. B. Orlin, R. E. Tarjan, Improved time bounds for the maximum flow problem. *SICOMP* **18**, 939–954 (1989). doi: [10.1137/0218065](https://doi.org/10.1137/0218065)
- A. V. Goldberg, S. Rao, Beyond the flow decomposition barrier. *J. Assoc. Comput. Mach.* **45**, 783–797 (1998). doi: [10.1145/290179.290181](https://doi.org/10.1145/290179.290181)
- T. B. Schardl, neboat/Moore: Initial release. Zenodo (2020); <https://zenodo.org/record/3715525>.

ACKNOWLEDGMENTS

We thank our many colleagues at MIT who engaged us in discussions regarding the end of Moore's law and, in particular, S. Devadas, J. Dennis, and Arvind. S. Amarasinghe inspired the matrix-multiplication example from the Software section. J. Kelper compiled a long history of maximum-flow algorithms that served as the basis for the study in the Algorithms section. We acknowledge our many colleagues who provided feedback on early drafts of this article: S. Aaronson, G. Blelloch, B. Colwell, B. Dally, J. Dean, P. Denning, J. Dongarra, J. Hennessy, J. Kepner, T. Mudge, Y. Patt, G. Lowney, L. Valiant, and M. Vardi. Thanks also to the anonymous referees, who provided excellent feedback and constructive criticism. **Funding:** This research was supported in part by NSF grants 1314547, 1452994 Sanchez, and 1533644. **Competing interests:** J.S.E. is also employed at Nvidia, B.W.L. is also employed by Microsoft, and B.C.K. is now employed at Google. **Data and materials availability:** The data and code used in the paper have been archived at Zenodo (64).

10.1126/science.aam9744

There's plenty of room at the Top: What will drive computer performance after Moore's law?

Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez and Tao B. Schardl

Science **368** (6495), eaam9744.
DOI: 10.1126/science.aam9744

From bottom to top

The doubling of the number of transistors on a chip every 2 years, a seemingly inevitable trend that has been called Moore's law, has contributed immensely to improvements in computer performance. However, silicon-based transistors cannot get much smaller than they are today, and other approaches should be explored to keep performance growing. Leiserson *et al.* review recent examples and argue that the most promising place to look is at the top of the computing stack, where improvements in software, algorithms, and hardware architecture can bring the much-needed boost.

Science, this issue p. eaam9744

ARTICLE TOOLS

<http://science.sciencemag.org/content/368/6495/eaam9744>

REFERENCES

This article cites 25 articles, 0 of which you can access for free
<http://science.sciencemag.org/content/368/6495/eaam9744#BIBL>

PERMISSIONS

<http://www.sciencemag.org/help/reprints-and-permissions>

Use of this article is subject to the [Terms of Service](#)

Science (print ISSN 0036-8075; online ISSN 1095-9203) is published by the American Association for the Advancement of Science, 1200 New York Avenue NW, Washington, DC 20005. The title *Science* is a registered trademark of AAAS.

Copyright © 2020 The Authors, some rights reserved; exclusive licensee American Association for the Advancement of Science. No claim to original U.S. Government Works