

Algorithms for Modern Hardware

Sergey Slotin

algorithmica.org

Algorithms for Modern Hardware

By Sergey Slotin

This is the complete collection of the “Algorithms for Modern Hardware” book from algorithmica.org.

The book covers performance engineering, computer architecture, SIMD, caching, algorithms, and data structures optimized for modern hardware.

Contents

1. Complexity Models
 2. Computer Architecture
 3. Instruction-Level Parallelism (Pipelining)
 4. Compilation
 5. Profiling
 6. Arithmetic
 7. Number Theory
 8. External Memory
 9. RAM & CPU Caches
 10. SIMD Parallelism
 11. Algorithm Case Studies
 12. Data Structure Case Studies
 13. Parallel Algorithms
 14. Distributed Computing
-

Generated from <https://github.com/algorithmica-org/algorithmica>

Chapter 1: Complexity Models

Algorithms for Modern Hardware

Contents

Complexity Models	1
Classical Complexity Theory	1
Asymptotic Complexity	3
Modern Hardware	4
How Microchips are Made	4
Dennard Scaling	4
Modern Computing	5
Programming Languages	6
Types of Languages	7
Interpreted languages	7
Managed Languages	8
Compiled Languages	9
BLAS	10
Takeaway	11
Models of Computation	11
Beyond Big O	12
When to Optimize	12
The Levels of Optimization	13
Estimating the impact	14

Complexity Models

If you ever opened a computer science textbook, it probably introduced *computational complexity* somewhere in the very beginning. Simply put, it is the total count of *elementary operations* (additions, multiplications, reads, writes...) that are executed during a computation, optionally weighted by their *costs*.

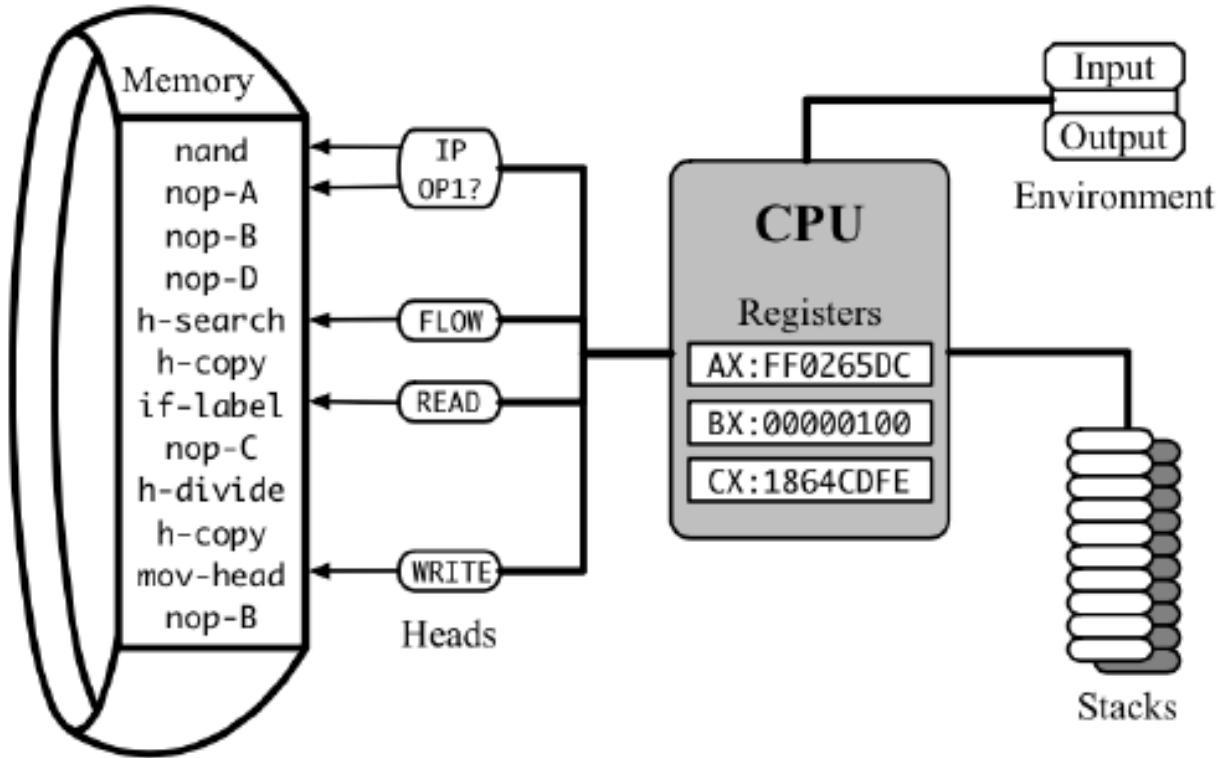
Complexity is an old concept. It was [systematically formulated](#) in the early 1960s, and since then it has been universally used as the cost function for designing algorithms. The reason this model was so quickly adopted is that it was a good approximation of how computers worked at the time.

Classical Complexity Theory

The “elementary operations” of a CPU are called *instructions*, and their “costs” are called *latencies*. Instructions are stored in *memory* and executed one by one by the processor, which has some internal *state* stored in a number of *registers*. One of these registers is the *instruction pointer*, which indicates the address of the next instruction to read and execute. Each instruction changes

the state of the processor in a certain way (including moving the instruction pointer), possibly modifies the main memory, and takes a different number of *CPU cycles* to complete before the next one can be started.

To estimate the real running time of a program, you need to sum all latencies for its executed instructions and divide it by the *clock frequency*, that is, the number of cycles a particular CPU does per second.



The clock frequency is a volatile and often unknown variable that depends on the CPU model, operating system settings, current microchip temperature, power usage of other components, and quite a few other things. In contrast, instruction latencies are static and even somewhat consistent across different CPUs when expressed in clock cycles, so counting them instead is much more useful for analytical purposes.

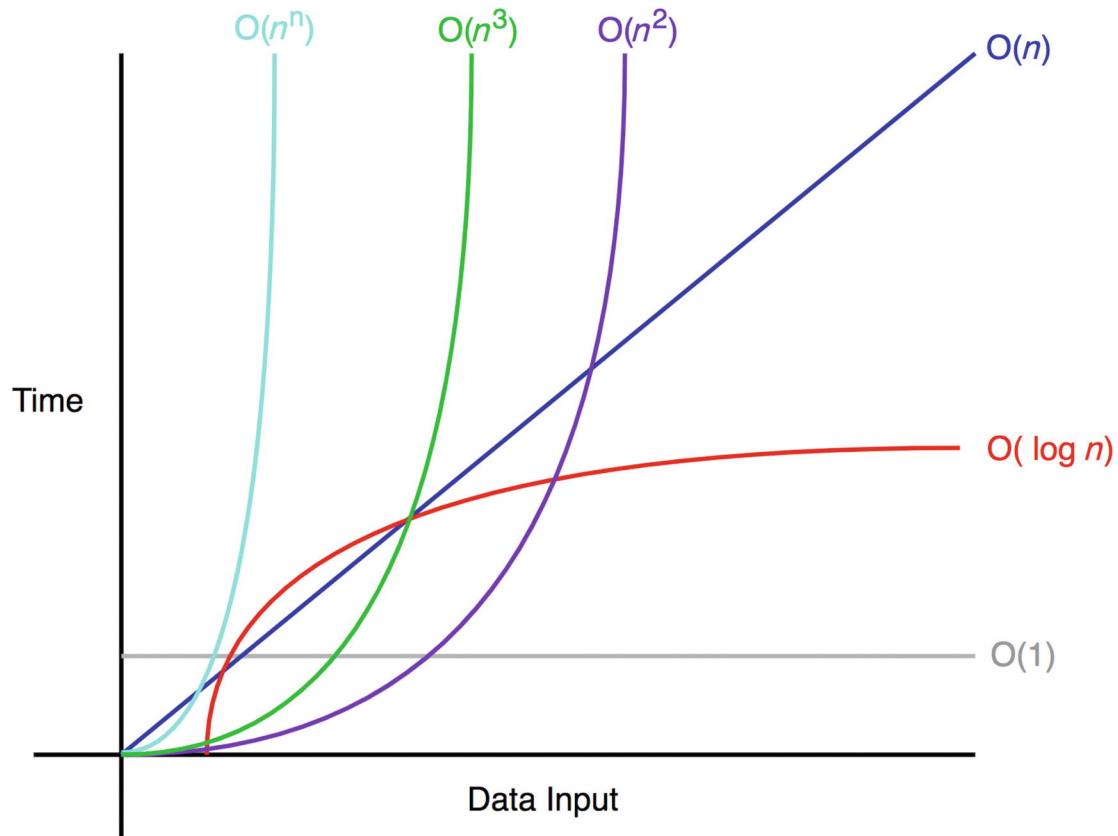
For example, the by-definition matrix multiplication algorithm requires the total of $n^2 \cdot (n + n - 1)$ arithmetic operations: specifically, n^3 multiplications and $n^2 \cdot (n - 1)$ additions. If we look up the latencies for these instructions (in special documents called *instruction tables*, like [this one](#)), we can find that, e.g., multiplication takes 3 cycles, while addition takes 1, so we need a total of $3 \cdot n^3 + n^2 \cdot (n - 1) = 4 \cdot n^3 - n^2$ clock cycles for the entire computation (bluntly ignoring everything else that needs to be done to “feed” these instructions with the right data).

Similar to how the sum of instruction latencies can be used as a clock-independent proxy for total execution time, computational complexity can be used to quantify the intrinsic time requirements of an abstract algorithm, without relying on the choice of a specific computer.

Asymptotic Complexity

The idea to express execution time as a function of input size seems obvious now, but it wasn't so in the 1960s. Back then, [typical computers](#) cost millions of dollars, were so large that they required a separate room, and had clock rates measured in kilohertz. They were used for practical tasks at hand, like predicting the weather, sending rockets into space, or figuring out how far a Soviet nuclear missile can fly from the coast of Cuba — all of which are finite-length problems. Engineers of that era were mainly concerned with how to multiply 3×3 matrices rather than $n \times n$ ones.

What caused the shift was the acquired confidence among computer scientists that computers will continue to become faster — and indeed they have. Over time, people stopped counting execution time, then stopped counting cycles, and then even stopped counting operations exactly, replacing it with an *estimate* that, on sufficiently large inputs, is only off by no more than a constant factor. With *asymptotic complexity*, verbose “ $4 \cdot n^3 - n^2$ operations” turns into plain “ $\Theta(n^3)$,” hiding the initial costs of individual operations in the “Big O,” along with all the other intricacies of the hardware.



The reason we use asymptotic complexity is that it provides simplicity while still being just precise enough to yield useful results about relative algorithm performance on large datasets. Under the promise that computers will eventually become fast enough to handle any *sufficiently large* input in a reasonable amount of time, asymptotically faster algorithms will always be faster in real-time too, regardless of the hidden constant.

But this promise turned out to be not true — at least not in terms of clock speeds and instruction latencies — and in this chapter, we will try to explain why and how to deal with it.

Modern Hardware

The main disadvantage of the supercomputers of the 1960s wasn't that they were slow — relatively speaking, they weren't — but that they were giant, complex to use, and so expensive that only the governments of the world superpowers could afford them. Their size was the reason they were so expensive: they required a lot of custom components that had to be very carefully assembled in the macro-world, by people holding advanced degrees in electrical engineering, in a process that couldn't be scaled up for mass production.

The turning point was the development of *microchips* — single, tiny, complete circuits — which revolutionized the industry and turned out to be probably the most important invention of the 20th century. What was a multimillion-dollar cupboard of computing machinery in 1965 could in 1975 fit on a [4mm × 4mm slice of silicon¹](#) that you can buy for \$25. This dramatic improvement in affordability started the home computer revolution during the following decade, with computers like Apple II, Atari 2600, Commodore 64, and IBM PC becoming available to the masses.

How Microchips are Made

Microchips are “printed” on a slice of crystalline silicon using a process called [photolithography](#), which involves

1. growing and slicing a [very pure silicon crystal](#),
2. covering it with a layer of [a substance that dissolves when photons hit it](#),
3. hitting it with photons in a set pattern,
4. chemically [etching](#) the now-exposed parts,
5. removing the remaining photoresist,

...and then performing another 40-50 steps over several months to complete the rest of the CPU.

Consider now the “hit it with photons” part. For that, we can use a system of lenses that projects a pattern onto a much smaller area, effectively making a tiny circuit with all the desired properties. This way, the optics of the 1970s were able to fit a few thousand transistors on the size of a fingernail, which gives microchips several key advantages that macro-world computers didn't have:

- higher clock rates (that were previously limited by the speed of light);
- the ability to scale the production;
- much lower material and power usage, translating to much lower cost per unit.

Apart from these immediate benefits, photolithography enabled a clear path to improve performance further: you can just make lenses stronger, which in turn would create smaller, but functionally identical devices with relatively little effort.

Dennard Scaling

Consider what happens when we scale a microchip down. A smaller circuit requires proportionally fewer materials, and smaller transistors take less time to switch (along with all other physical processes in the chip), allowing reducing the voltage and increasing the clock rate.

A more detailed observation, known as the *Dennard scaling*, states that reducing transistor dimensions by 30%

¹Actual sizes of CPUs are about centimeter-scale because of power management, heat dissipation, and the need to plug it into the motherboard without excessive swearing.

- doubles the transistor density ($0.7^2 \approx 0.5$),
- increases the clock speed by 40% ($\frac{1}{0.7} \approx 1.4$),
- and leaves the overall *power density* the same.

Since the per-unit manufacturing cost is a function of area, and the exploitation cost is mostly the cost of power², each new “generation” should have roughly the same total cost, but 40% higher clock and twice as many transistors, which can be promptly used, for example, to add new instructions or increase the word size — to keep up with the same miniaturization happening in memory microchips.

Due to the trade-offs between energy and performance you can make during the design, the fidelity of the fabrication process itself, such as “180nm” or “65nm,” directly translating to the density of transistors, became the trademark for CPU efficiency³.

Throughout most of the computing history, optical shrinking was the main driving force behind performance improvements. Gordon Moore, the former CEO of Intel, predicted in 1975 that the transistor count in microprocessors will double every two years. His prediction held to this day and became known as *Moore’s law*.

Both Dennard scaling and Moore’s law are not actual laws of physics, but just observations made by savvy engineers. They are both destined to stop at some point due to fundamental physical limitations, the ultimate one being the size of silicon atoms. In fact, Dennard scaling already did — due to power issues.

Thermodynamically, a computer is just a very efficient device for converting electrical power into heat. This heat eventually needs to be removed, and there are physical limits to how much power you can dissipate from a millimeter-scale crystal. Computer engineers, aiming to maximize performance, essentially just choose the maximum possible clock rate so that the overall power consumption stays the same. If transistors become smaller, they have less capacitance, meaning less required voltage to flip them, which in turn allows increasing the clock rate.

Around 2005–2007, this strategy stopped working because of *leakage* effects: the circuit features became so small that their magnetic fields started to make the electrons in the neighboring circuitry move in directions they are not supposed to, causing unnecessary heating and occasional bit flipping.

The only way to mitigate this is to increase the voltage; and to balance off power consumption you need to reduce clock frequency, which in turn makes the whole process progressively less profitable as transistor density increases. At some point, clock rates could no longer be increased by scaling, and the miniaturization trend started to slow down.

Modern Computing

Dennard scaling has ended, but Moore’s law is not dead yet.

Clock rates plateaued, but the transistor count is still increasing, allowing for the creation of new, *parallel* hardware. Instead of chasing faster cycles, CPU designs started to focus on getting more

²The cost of electricity for running a busy server for 2-3 years roughly equals the cost of making the chip itself.

³At some point, when Moore’s law started to slow down, chip makers stopped delineating their chips by the size of their components — and it is now more like a marketing term. A special committee has a meeting every two years where they take the previous node name, divide it by the square root of two, round to the nearest integer, declare the result to be the new node name, and then drink lots of wine. The “nm” doesn’t mean nanometer anymore.

useful things done in a single cycle. Instead of getting smaller, transistors have been changing shape.

This resulted in increasingly complex architectures capable of doing dozens, hundreds, or even thousands of different things every cycle.

Die shot of a Zen CPU core by AMD (~1,400,000,000 transistors)

Figure 1: Die shot of a Zen CPU core by AMD (~1,400,000,000 transistors)

Here are some core approaches making use of more available transistors that are driving recent computer designs:

- Overlapping the execution of instructions so that different parts of the CPU are kept busy (pipelining);
- Executing operations without necessarily waiting for the previous ones to complete (speculative and out-of-order execution);
- Adding multiple execution units to process independent operations simultaneously (super-scalar processors);
- Increasing the machine word size, to the point of adding instructions capable of executing the same operation on a block of 128, 256, or 512 bits of data split into groups (**SIMD**);
- Adding **layers of cache** on the chip to speed up **RAM and external memory** access time (memory doesn't quite follow the laws of silicon scaling);
- Adding multiple identical cores on a chip (parallel computing, GPUs);
- Using multiple chips in a motherboard and multiple cheaper computers in a data center (distributed computing);
- Using custom hardware to solve a specific problem with better chip utilization (ASICs, FPGAs).

For modern computers, the “**let’s count all operations**” approach for predicting algorithm performance isn’t just slightly wrong but is off by several orders of magnitude. This calls for new computation models and other ways of assessing algorithm performance.

Programming Languages

If you are reading this book, then somewhere on your computer science journey you had a moment when you first started to care about the efficiency of your code.

Mine was in high school, when I realized that making websites and doing *useful* programming won’t get you into a university, and entered the exciting world of algorithmic programming olympiads. I was an okay programmer, especially for a highschooler, but I had never really wondered how much time it took for my code to execute before. But suddenly it started to matter: each problem now has a strict time limit. I started counting my operations. How many can you do in one second?

I didn’t know much about computer architecture to answer this question. But I also didn’t need the right answer — I needed a rule of thumb. My thought process was: “2-3GHz means 2 to 3 billion instructions executed every second, and in a simple loop that does something with array elements, I also need to increment loop counter, check end-of-loop condition, do array indexing and stuff like that, so let’s add room for 3-5 more instructions for every useful one” and ended up with using $5 \cdot 10^8$ as an estimate. None of these statements are true, but counting how many operations my algorithm needed and dividing it by this number was a good rule of thumb for my use case.

The real answer, of course, is much more complicated and highly dependent on what kind of “operation” you have in mind. It can be as low as 10^7 for things like [pointer chasing](#) and as high as 10^{11} for [SIMD-accelerated](#) linear algebra. To demonstrate these striking differences, we will use the case study of matrix multiplication implemented in different languages — and dig deeper into how computers execute them.

Types of Languages

On the lowest level, computers execute *machine code* consisting of binary-encoded *instructions* which are used to control the CPU. They are specific, quirky, and require a great deal of intellectual effort to work with, so one of the first things people did after creating computers was create *programming languages*, which abstract away some details of how computers operate to simplify the process of programming.

A programming language is fundamentally just an interface. Any program written in it is just a nicer higher-level representation which still at some point needs to be transformed into the machine code to be executed on the CPU — and there are several different means of doing that:

- From a programmer’s perspective, there are two types of languages: *compiled*, which pre-process before executing, and *interpreted*, which are executed during runtime using a separate program called *an interpreter*.
- From a computer’s perspective, there are also two types of languages: *native*, which directly execute machine code, and *managed*, which rely on some sort of *runtime* to do it.

Since running machine code in an interpreter doesn’t make sense, this makes a total of three types of languages:

- Interpreted languages, such as Python, JavaScript, or Ruby.
- Compiled languages with a runtime, such as Java, C#, or Erlang (and languages that work on their VMs, such as Scala, F#, or Elixir).
- Compiled native languages, such as C, Go, or Rust.

There is no “right” way of executing computer programs: each approach has its own gains and drawbacks. Interpreters and virtual machines provide flexibility and enable some nice high-level programming features such as dynamic typing, run time code alteration, and automatic memory management, but these come with some unavoidable performance trade-offs, which we will now talk about.

Interpreted languages

Here is an example of a by-definition 1024×1024 matrix multiplication in pure Python:

```
import time
import random

n = 1024

a = [[random.random()
      for row in range(n)]
      for col in range(n)]
```

```

b = [[random.random()
      for row in range(n)]
      for col in range(n)]

c = [[0
      for row in range(n)]
      for col in range(n)]

start = time.time()

for i in range(n):
    for j in range(n):
        for k in range(n):
            c[i][j] += a[i][k] * b[k][j]

duration = time.time() - start
print(duration)

```

This code runs in 630 seconds. That's more than 10 minutes!

Let's try to put this number in perspective. The CPU that ran it has a clock frequency of 1.4GHz, meaning that it does $1.4 \cdot 10^9$ cycles per second, totaling to almost 10^{15} for the entire computation, and about 880 cycles per multiplication in the innermost loop.

This is not surprising if you consider the things that Python needs to do to figure out what the programmer meant:

- it parses the expression `c[i][j] += a[i][k] * b[k][j];`
- tries to figure out what `a`, `b`, and `c` are and looks up their names in a special hash table with type information;
- understands that `a` is a list, fetches its `[]` operator, retrieves the pointer for `a[i]`, figures out it's also a list, fetches its `[]` operator again, gets the pointer for `a[i][k]`, and then the element itself;
- looks up its type, figures out that it's a `float`, and fetches the method implementing `*` operator;
- does the same things for `b` and `c` and finally add-assigns the result to `c[i][j]`.

Granted, the interpreters of widely used languages such as Python are well-optimized, and they can skip through some of these steps on repeated execution of the same code. But still, some quite significant overhead is unavoidable due to the language design. If we get rid of all this type checking and pointer chasing, perhaps we can get cycles per multiplication ratio closer to 1, or whatever the "cost" of native multiplication is?

Managed Languages

The same matrix multiplication procedure, but implemented in Java:

```

import java.util.Random;

public class Matmul {
    static int n = 1024;

```

```

static double[][] a = new double[n][n];
static double[][] b = new double[n][n];
static double[][] c = new double[n][n];

public static void main(String[] args) {
    Random rand = new Random();

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            a[i][j] = rand.nextDouble();
            b[i][j] = rand.nextDouble();
            c[i][j] = 0;
        }
    }

    long start = System.nanoTime();

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];

    double diff = (System.nanoTime() - start) * 1e-9;
    System.out.println(diff);
}
}

```

It now runs in 10 seconds, which amounts to roughly 13 CPU cycles per multiplication — 63 times faster than Python. Considering that we need to read elements of `b` non-sequentially from the memory, the running time is roughly what it is supposed to be.

Java is a *compiled*, but not *native* language. The program first compiles to *bytecode*, which is then interpreted by a virtual machine (JVM). To achieve higher performance, frequently executed parts of the code, such as the innermost `for` loop, are compiled into the machine code during runtime and then executed with almost no overhead. This technique is called *just-in-time compilation*.

JIT compilation is not a feature of the language itself, but of its implementation. There is also a JIT-compiled version of Python called [PyPy](#), which needs about 12 seconds to execute the code above without any changes to it.

Compiled Languages

Now it's turn for C:

```

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define n 1024
double a[n][n], b[n][n], c[n][n];

```

```

int main() {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            a[i][j] = (double) rand() / RAND_MAX;
            b[i][j] = (double) rand() / RAND_MAX;
        }
    }

    clock_t start = clock();

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];

    float seconds = (float) (clock() - start) / CLOCKS_PER_SEC;
    printf("%.4f\n", seconds);

    return 0;
}

```

It takes 9 seconds when you compile it with `gcc -O3`.

It doesn't seem like a huge improvement — the 1-3 second advantage over Java and PyPy can be attributed to the additional time of JIT-compilation — but we haven't yet taken advantage of a far better C compiler ecosystem. If we add `-march=native` and `-ffast-math` flags, time suddenly goes down to 0.6 seconds!

What happened here is we [communicated to the compiler](#) the exact model of the CPU we are running (`-march=native`) and gave it the freedom to rearrange [floating-point computations](#) (`-ffast-math`), and so it took advantage of it and used [vectorization](#) to achieve this speedup.

It's not like it is impossible to tune the JIT-compilers of PyPy and Java to achieve the same performance without significant changes to the source code, but it is certainly easier for languages that compile directly to native code.

BLAS

Finally, let's take a look at what an expert-optimized implementation is capable of. We will test a widely-used optimized linear algebra library called [OpenBLAS](#). The easiest way to use it is to go back to Python and just call it from `numpy`:

```

import time
import numpy as np

n = 1024

a = np.random.rand(n, n)
b = np.random.rand(n, n)

```

```

start = time.time()

c = np.dot(a, b)

duration = time.time() - start
print(duration)

```

Now it takes ~ 0.12 seconds: a $\sim 5x$ speedup over the auto-vectorized C version and $\sim 5250x$ speedup over our initial Python implementation!

You don't typically see such dramatic improvements. For now, we are not ready to tell you exactly how this is achieved. Implementations of dense matrix multiplication in OpenBLAS are typically [5000 lines of handwritten assembly](#) tailored separately for *each* architecture. In later chapters, we will explain all the relevant techniques one by one, and then [return](#) to this example and develop our own BLAS-level implementation using just under 40 lines of C.

Takeaway

The key lesson here is that using a native, low-level language doesn't necessarily give you performance; but it does give you *control* over performance.

Complementary to the “N operations per second” simplification, many programmers also have a misconception that using different programming languages has some sort of multiplier on that number. Thinking this way and [comparing languages](#) in terms of performance doesn't make much sense: programming languages are fundamentally just tools that take away *some* control over performance in exchange for convenient abstractions. Regardless of the execution environment, it is still largely a programmer's job to use the opportunities that the hardware provides.

Models of Computation

In classical theoretical computer science, really exciting things stopped happening in the 70s; everything past that are just attempts to replace logarithms in the asymptotic with something slightly less than logarithms. If 50 years ago such algorithms had hope that eventually there will be enough computing power to process the large datasets for which they beat their asymptotically inferior, but practical counterparts, nowadays we know for certain that they never will.

This is what this book is about: accepting the reality and optimizing for the hardware you have, beyond just asymptotic complexity. You will probably not learn a single asymptotically faster algorithm here, but you will learn how to squeeze performance from all of non-exponentially-increasing transistors you have, which is a far more impactful skill.

Computers are still getting faster, but in ways orthogonal to the computation model, and we need to create new ones

Why this is important to optimize algorithms beyond Big O.

The goal of this chapter is to explain why this is the case,

It became much more important.

This model worked well *at the time*. Computers and program execution environments are much more complex now.

In this chapter we are going to explore the reasons why this happened.

Beyond Big O

This was the case. More computers are not tall, they are broad.

The models typically “charge” only one type of operation.

RAM model random-access machines

Word RAM model

External memory model

Different types of parallel RAM model

Communication complexity

There are many other frameworks. Some in terms of cryptography or communication in general. Some work for hardware designers. “Transistor model” that counts the number of transistors, or “energy model” based on physical entropy laws. Quantum algorithms, or maybe even human-executed algorithms, where you also need to care about errors.

When to Optimize

Why do companies like Google ask to whiteboard-solve algorithm problems in their interviews?

The reason is that organizations with more than 10000 engineers are very different from the outside world.

Sure, as you progress through career, these coding questions get replaced with complex system design problems and assessments of managerial ability. It is just irrelevant yet, because you are not getting into a senior role straight out of college.

This is mostly just my ramblings about career paths in performance engineering.

most of the CS knowledge is abstracted away by high level languages and libraries

I assume that you either want to work in the industry, or in the academia while doing something actually useful.

Algorithm design is the kind of skill that you use 20% of time, but it is responsible for 80% of success.

There are some areas where performance is a killer features. But look at Java ecosystem: it is hellishly slow, and yet it still powers most of big data ecosystem, and runs most of Android apps. Java and other VM-based languages like WebAssembly are not going away anytime soon.

Of course, there is also lot of *stupidity* bias in hiring. There are big companies that don’t ask that at all, and there are companies fully comprised of either former or active competitive programmers. It makes sense, because if people believe some skill is undervalued, why will hire people with that skill, and vice versa.

In any case, the Big-O notation is not what companies really want. It is not about writing optimal problems — it’s more about avoiding terribly slow ones. The ones that don’t scale. Compute indeed keeps getting cheaper.

You get especially frustrated if you had a competitive programming experience. You won't get to solve these type of problems, even if they asked them on an interview. To solve them, you need other type of qualifications. Asymptotically optimal algorithm already exists, you need to optimize the constant factor. Unfortunately, only a handful of universities teach that.

The Levels of Optimization

Programmers can be put in several "levels" in terms of their software optimization abilities:

0. *Newbie*. Those who don't think about performance at all. They usually write in high-level languages, sometimes in declarative / functional languages. Most "programmers" stay there (and there is nothing wrong with it).
1. *Undergraduate student*. Those who know about Big O notation and are familiar with basic data structures and approaches. LeetCode and CodeForces folks are there. This is also the requirement in getting into big companies — they have a lot of in-house software, large scale, and they are looking for people in the long term, so asking things like programming language.
2. *Graduate student*. Those who know that not all operations are created equal; know other cost models such as external memory model (B-tree, external sorting), word model (bitset,) or parallel computing, but still in theory.
3. *Professional developer*. Those who know actual timings of these operations. Aware that branch mispredictions are costly, memory is split into cache lines. Knows some basic SIMD techniques.
4. *Performance engineer*. Know exactly what happens inside their hardware. Know the difference between latency and bandwidth, know about ports. Knows how to use SIMD and the rest of instruction set effectively. Can read assembly and use profilers.
5. *Intel employee*. Knows microarchitecture-specific details. This is outside of the purview of normal engineers.

In this book, we expect that the average reader is somewhere around stage 1, and hopefully by the end of it will get to 4.

You should also go through these levels when designing algorithms. First get it working in the first place, then select a bunch of reasonably asymptotically optimal algorithm. Then think about how they are going to work in terms of their memory operations or ability to execute in parallel (even if you consider single-threaded programs, there is still going to be plenty of parallelism inside a core, so this model is extremely), and then proceed toward actual implementation. Avoid premature optimization, as Knuth once said.

For most web services, efficiency doesn't matter, but *latency* does.

Increasing efficiency is not how it is done nowadays.

A pageview usually generates somewhere on the order of 0.1 to 1 cent per pageview. This is a typical rate at which you monetize user attention. Say, if I simply installed AdSense, i'd be getting something like that — depending on where most of my readers are from and how many of them are using an ad blocker.

At the same time, a server with a dedicated core and 1GB of ram (which is an absurdly large amount of resources for a simple web service) costs around one millionth per second when amortized. You could fetch 100 photos with that.

Amazon had an experiment where they A/B tested their service with artificial delays and found

out that a 100ms delay decreased revenue. This follows for most other services, say, you lose your “flow” at twitter, the user is likely to start thinking on something else and leave. If the delay at Google is more than a few seconds, people will just think that Google isn’t working and quit.

Minimization of latency can be usually done with parallel computing, which is why distributed systems are scaled more on scalability. This part of the book is concerned with improving *efficiency* of algorithms, which makes latency lower as the by-product.

However, there are still use cases when there is a trade-off between quality and cost of servers.

- Search is hierarchical. There are usually many layers of more accurate but slower models. The more documents you rank on each layer, the better the final quality.
- Games. They are more enjoyable on large scale, but computational power also increases. This includes AI.
- AI workloads — those that have large quantities of data such as language models. Heavier models require more compute. The bottleneck in them is not the number of data, but efficiency.

Inherently sequential algorithms, or cases when the resources are constrained. Ctrl+f’ing a large PDF is painful. Factorization.

Estimating the impact

Sometime the optimization needs to happen in the calling layer.

SIMDJSON speeds up JSON parsing, but it may be better to not use JSON in the first place.

Protobuf or flat binary formats.

There is also a chicken and egg problem: people don’t use an approach that much because it is slow and not feasible.

Cost to implement, bugs, maintainability. It is perfectly fine that most software in the world is inefficient.

What does it mean to be a better programmer? Faster programs? Faster speed of work? Fewer bugs? It is a combination of those.

Implementing compiler optimizations or databases are examples of high-leverage activities because they act as a tax on everything else — which is why you see most people writing books on these particular topics rather than software optimization in general.

Factorization is kind of useless by itself, but it helps with understanding how to optimize number theoretic computations in general. Same goes for sorting and binary trees: most people hold some metainformation.

Chapter 2: Computer Architecture

Algorithms for Modern Hardware

Contents

Instruction Set Architectures	1
RISC vs CISC	2
Assembly Language	3
Instructions and Registers	4
Moving Data	4
Addressing Modes	5
Alternative Syntax	5
Computer Architecture	6
Loops and Conditionals	7
Jumps	7
Loop Unrolling	7
An Alternative Approach	8
Functions and Recursion	9
The Stack	9
Calling Conventions	10
Inlining	11
Tail Call Elimination	11
Indirect Branching	13
Multiway Branch	13
Dynamic Dispatch	14
Interrupts and System Calls	15
Machine Code Layout	16
CPU Front-End	16
Code Alignment	16
Instruction Cache	17
Unequal Branches	17

Instruction Set Architectures

As software engineers, we absolutely love building and using abstractions.

Just imagine how much stuff happens when you load a URL. You type something on a keyboard; key presses are somehow detected by the OS and get sent to the browser; browser parses the URL and asks the OS to make a network request; then comes DNS, routing, TCP, HTTP, and all the other OSI layers; browser parses HTML; JavaScript works its magic; some representation of a page

gets sent over to GPU for rendering; image frames get sent to the monitor... and each of these steps probably involves doing dozens of more specific things in the process.

Abstractions help us in reducing all this complexity down to a single *interface* that describes what a certain *module* can do without fixing a concrete implementation. This provides double benefits:

- Engineers working on higher-level modules only need to know the (much smaller) interface.
- Engineers working on the module itself get the freedom to optimize and refactor its implementation as long as it complies with its *contracts*.

Hardware engineers love abstractions too. An abstraction of a CPU is called an *instruction set architecture* (ISA), and it defines how a computer should work from a programmer's perspective. Similar to software interfaces, it gives computer engineers the ability to improve on existing CPU designs while also giving its users — us, programmers — the confidence that things that worked before won't break on newer chips.

An ISA essentially defines how the hardware should interpret the machine language. Apart from instructions and their binary encodings, an ISA also defines the counts, sizes, and purposes of registers, the memory model, and the input/output model. Similar to software interfaces, ISAs can be extended too: in fact, they are often updated, mostly in a backward-compatible way, to add new and more specialized instructions that can improve performance.

RISC vs CISC

Historically, there have been many competing ISAs in use. But unlike [character encodings](#) and [instant messaging protocols](#), developing and maintaining a completely separate ISA is costly, so mainstream CPU designs ended up converging to one of the two families:

- **Arm** chips, which are used in almost all mobile devices, as well as other computer-like devices such as TVs, smart fridges, microwaves, [car autopilots](#), and so on. They are designed by a British company of the same name, as well as a number of electronics manufacturers including Apple and Samsung.
- **x86¹** chips, which are used in almost all servers and desktops, with a few notable exceptions such as Apple's M1 MacBooks, AWS's Graviton processors, and the current [world's fastest supercomputer](#), all of which use Arm-based CPUs. They are designed by a duopoly of Intel and AMD.

The main difference between them is that of architectural complexity, which is more of a design philosophy rather than some strictly defined property:

- Arm CPUs are *reduced instruction set computers* (RISC). They improve performance by keeping the instruction set small and highly optimized, although some less common operations have to be implemented with subroutines involving several instructions.
- x86 CPUs are *complex instruction set computers* (CISC). They improve performance by adding many specialized instructions, some of which may only be rarely used in practical programs.

The main advantage of RISC designs is that they result in simpler and smaller chips, which projects to lower manufacturing costs and power usage. It's not surprising that the market segmented itself

¹Modern 64-bit versions of x86 are known as "AMD64," "Intel 64," or by the more vendor-neutral names of "x86-64" or just "x64." A similar 64-bit extension of Arm is called "AArch64" or "ARM64." In this book, we will just use plain "x86" and "Arm" implying the 64-bit versions.

with Arm dominating battery-powered, general-purpose devices, and leaving the complex neural network and Galois field calculations to server-grade, highly-specialized x86s.

Assembly Language

CPUs are controlled with *machine language*, which is just a stream of binary-encoded instructions that specify

- the instruction number (called *opcode*),
- what its *operands* are (if there are any),
- and where to store the *result* (if one is produced).

A much more human-friendly rendition of machine language, called *assembly language*, uses mnemonic codes to refer to machine code instructions and symbolic names to refer to registers and other storage locations.

Jumping right into it, here is how you add two numbers ($*c = *a + *b$) in Arm assembly:

```
; *a = x0, *b = x1, *c = x2
ldr w0, [x0]      ; load 4 bytes from wherever x0 points into w0
ldr w1, [x1]      ; load 4 bytes from wherever x1 points into w1
add w0, w0, w1   ; add w0 with w1 and save the result to w0
str w0, [x2]      ; write contents of w0 to wherever x2 points
```

Here is the same operation in x86 assembly:

```
; *a = rsi, *b = rdi, *c = rdx
mov eax, DWORD PTR [rsi]  ; load 4 bytes from wherever rsi points into eax
add eax, DWORD PTR [rdi]  ; add whatever is stored at rdi to eax
mov DWORD PTR [rdx], eax  ; write contents of eax to wherever rdx points
```

Assembly is very simple in the sense that it doesn't have many syntactical constructions compared to high-level programming languages. From what you can observe from the examples above:

- A program is a sequence of instructions, each written as its name followed by a variable number of operands.
- The `[reg]` syntax is used for “dereferencing” a pointer stored in a register, and on x86 you need to prefix it with size information (DWORD here means 32 bit).
- The `;` sign is used for line comments, similar to `#` and `//` in other languages.

Assembly is a very minimal language because it needs to be. It reflects the machine language as closely as possible, up to the point where there is almost 1:1 correspondence between machine code and assembly. In fact, you can turn any compiled program back into its assembly form using a process called *disassembly*² — although everything non-essential like comments will not be preserved.

Note that the two snippets above are not just syntactically different. Both are optimized codes produced by a compiler, but the Arm version uses 4 instructions, while the x86 version uses 3. The `add eax, [rdi]` instruction is what's called *fused instruction* that does a load and an add in one go — this is one of the perks that the **CISC** approach can provide.

²On Linux, to disassemble a compiled program, you can call `objdump -d {path-to-binary}`.

Since there are far more differences between the architectures than just this one, from here on and for the rest of the book we will only provide examples for x86, which is probably what most of our readers will optimize for, although many of the introduced concepts will be architecture-agnostic.

Instructions and Registers

For historical reasons, instruction mnemonics in most assembly languages are very terse. Back when people used to write assembly by hand and repeatedly wrote the same set of common instructions, one less character to type was one step away from insanity.

For example, `mov` is for “store/load a word,” `inc` is for “increment by 1,” `mul` is for “multiply,” and `idiv` is for “integer division.” You can look up the description of an instruction by its name in [one of x86 references](#), but most instructions do what you’d think they do.

Most instructions write their result into the first operand, which can also be involved in the computation like in the `add eax, [rdi]` example we saw before. Operands can be either registers, constant values, or memory locations.

Registers are named `rax`, `rbx`, `rcx`, `rdx`, `rdi`, `rsi`, `rbp`, `rsp`, and `r8-r15` for a total of 16 of them. The “letter” ones are named like that for historical reasons: `rax` is “accumulator,” `rcx` is “counter,” `rdx` is “data” and so on — but, of course, they don’t have to be used only for that.

There are also 32-, 16-bit and 8-bit registers that have similar names (`rax` → `eax` → `ax` → `al`). They are not fully separate but *aliased*: the lowest 32 bits of `rax` are `eax`, the lowest 16 bits of `eax` are `ax`, and so on. This is made to save die space while maintaining compatibility, and it is also the reason why basic type casts in compiled programming languages are usually free.

These are just the *general-purpose* registers that you can, with [some exceptions](#), use however you like in most instructions. There is also a separate set of registers for [floating-point arithmetic](#), a bunch of very wide registers used in [vector extensions](#), and a few special ones that are needed for [control flow](#), but we’ll get there in time.

Constants are just integer or floating-point values: `42`, `0x2a`, `3.14`, `6.02e23`. They are more commonly called *immediate values* because they are embedded right into the machine code. Because it may considerably increase the complexity of the instruction encoding, some instructions don’t support immediate values or allow just a fixed subset of them. In some cases, you have to load a constant value into a register and then use it instead of an immediate value.

Apart from numeric values, there are also string constants such as `hello` or `world\n` with their own little subset of operations, but that is a somewhat obscure corner of the assembly language that we are not going to explore here.

Moving Data

Some instructions may have the same mnemonic, but have different operand types, in which case they are considered distinct instructions as they may perform slightly different operations and take different times to execute. The `mov` instruction is a vivid example of that, as it comes in around 20 different forms, all related to moving data: either between the memory and registers or just between two registers. Despite the name, it doesn’t *move* a value into a register, but *copies* it, preserving the original.

When used to copy data between two registers, the `mov` instruction instead performs *register renaming* internally — informs the CPU that the value referred by register X is actually stored in register Y — without causing any additional delay except for maybe reading and decoding the instruction itself. For the same reason, the `xchg` instruction that swaps two registers also doesn't cost anything.

As we've seen above with the fused `add`, you don't have to use `mov` for every memory operation: some arithmetic instructions conveniently support memory locations as operands.

Addressing Modes

Memory addressing is done with the `[]` operator, but it can do more than just reinterpret a value stored in a register as a memory location. The address operand takes up to 4 parameters presented in the syntax:

```
SIZE PTR [base + index * scale + displacement]
```

where `displacement` needs to be an integer constant and `scale` can be either 2, 4, or 8. What it does is calculate the pointer `base + index * scale + displacement` and dereferences it.

Using complex addressing is [at most one cycle slower](#) than dereferencing a pointer directly, and it can be useful when you have, for example, an array of structures and want to load a specific field of its i -th element.

Addressing operator needs to be prefixed with a size specifier for how many bits of data are needed:

- `BYTE` for 8 bits
- `WORD` for 16 bits
- `DWORD` for 32 bits
- `QWORD` for 64 bits

There is also a more rare `TBYTE` for [80 bits](#), and `XMMWORD`, `YMMWORD`, and `ZMMWORD` for [128, 256, and 512 bits](#) respectively. All these types don't have to be written in uppercase, but this is how most compilers emit them.

The address computation is often useful by itself: the `lea` (“load effective address”) instruction calculates the memory address of the operand and stores it in a register in one cycle, without doing any actual memory operations. While its intended use is for actually computing memory addresses, it is also often used as an arithmetic trick that would otherwise involve 1 multiplication and 2 additions — for example, you can multiply by 3, 5, and 9 with it.

It also frequently serves as a replacement for `add` because it doesn't need a separate `mov` instruction if you need to move the result somewhere else: `add` only works in the two-register `a += b` mode, while `lea` lets you do `a = b + c` (or even `a = b + c + d` if one of them is a constant).

Alternative Syntax

There are actually multiple *assemblers* (the programs that produce machine code from assembly) with different assembly languages, but only two x86 syntaxes are widely used now. They are commonly called after the two companies that used them and had a dominant influence on programming during that era:

- The *AT&T syntax*, used by default by all Linux tools.

- The *Intel syntax*, used by default, well, by Intel.

These syntaxes are also sometimes called *GAS* and *NASM* respectively, by the names of the two primary assemblers that use them (*GNU Assembler* and *Netwide Assembler*).

We used Intel syntax in this chapter and will continue to preferably use it for the rest of the book. For comparison, here is how the same `*c = *a + *b` example looks like in AT&T asm:

```
movl (%rsi), %eax
addl (%rdi), %eax
movl %eax, (%rdx)
```

The key differences can be summarized as follows:

1. The *last* operand is used to specify the destination.
2. Registers and constants need to be prefixed by % and \$ respectively (e.g., addl \$1, %rdx increments rdx).
3. Memory addressing looks like this: `displacement(%base, %index, scale)`.
4. Both ; and # can be used for line comments, and also /* */ can be used for block comments.

And, most importantly, in AT&T syntax, the instruction names need to be “suffixed” (`addq`, `movl`, `cmpq`, etc.) to specify what size operands are being manipulated:

- b = byte (8 bit)
- w = word (16 bit)
- l = long (32 bit integer or 64-bit floating-point)
- q = quad (64 bit)
- s = single (32-bit floating-point)
- t = ten bytes (80-bit floating-point)

In Intel syntax, this information is inferred from operands (which is why you also need to specify sizes of pointers).

Most tools that produce or consume x86 assembly can do so in both syntaxes, so you can just pick the one you like more and don’t worry.

Computer Architecture

When I began learning how to optimize programs myself, one big mistake I made was to rely primarily on the empirical approach. Not understanding how computers really worked, I would semi-randomly swap nested loops, rearrange arithmetic, combine branch conditions, inline functions by hand, and follow all sorts of other performance tips I’ve heard from other people, blindly hoping for improvement.

Unfortunately, this is how most programmers approach optimization. Most texts about performance do not teach you to reason about software performance qualitatively. Instead they give you general advice about certain implementation approaches — and general performance intuition is clearly not enough.

It would have probably saved me dozens, if not hundreds of hours if I learned computer architecture *before* doing algorithmic programming. So, even if most people aren’t *excited* about it, we are going to spend the first few chapters studying how CPUs work and start with learning assembly.

Loops and Conditionals

Let's consider a slightly more complex example:

```
loop:  
    add  edx, DWORD PTR [rax]  
    add  rax, 4  
    cmp  rax, rcx  
    jne  loop
```

It calculates the sum of a 32-bit integer array, just as a simple `for` loop would.

The “body” of the loop is `add edx, DWORD PTR [rax]`: this instruction loads data from the iterator `rax` and adds it to the accumulator `edx`. Next, we move the iterator 4 bytes forward with `add rax, 4`. Then, a slightly more complicated thing happens.

Jumps

Assembly doesn't have if-s, for-s, functions, or other control flow structures that high-level languages have. What it does have is `goto`, or “jump,” how it is known in the world of low-level programming.

Jump moves the instruction pointer to a location specified by its operand. This location may be either an absolute address in memory, relative to the current address or even **computed during runtime**. To avoid the headache of managing these addresses directly, you can mark any instruction with a string followed by `:`, and then use this string as a label which gets replaced by the relative address of this instruction when converted to machine code.

Labels can be any string, but compilers don't get creative and **typically** just use the line numbers in the source code and function names with their signatures when picking names for labels.

Unconditional jump `jmp` can only be used to implement `while (true)` kind of loops or stitch parts of a program together. A family of **conditional** jumps is used to implement actual control flow.

It is reasonable to think that these conditions are computed as `bool`-s somewhere and passed to conditional jumps as operands: after all, this is how it works in programming languages. But that is not how it is implemented in hardware. Conditional operations use a special `FLAGS` register, which first needs to be populated by executing instructions that perform some kind of check.

In our example, `cmp rax, rcx` compares the iterator `rax` with the end-of-array pointer `rcx`. This updates the `FLAGS` register, and now it can be used by `jne loop`, which looks up a certain bit there that tells whether the two values are equal or not, and then either jumps back to the beginning or continues to the next instruction, thus breaking the loop.

Loop Unrolling

One thing you might have noticed about the loop above is that there is a lot of overhead to process a single element. During each cycle, there is only one useful instruction executed, and the other 3 are incrementing the iterator and trying to find out if we are done yet.

What we can do is to *unroll* the loop by grouping iterations together — equivalent to writing something like this in C:

```

for (int i = 0; i < n; i += 4) {
    s += a[i];
    s += a[i + 1];
    s += a[i + 2];
    s += a[i + 3];
}

```

In assembly, it would look something like this:

```

loop:
    add edx, [rax]
    add edx, [rax+4]
    add edx, [rax+8]
    add edx, [rax+12]
    add rax, 16
    cmp rax, rsi
    jne loop

```

Now we only need 3 loop control instructions for 4 useful ones (an improvement from $\frac{1}{4}$ to $\frac{4}{7}$ in terms of efficiency), and this can be continued to reduce the overhead almost to zero.

In practice, unrolling loops isn't always necessary for performance because modern processors don't actually execute instructions one-by-one, but maintain a [queue of pending instructions](#) so that two independent operations can be executed concurrently without waiting for each other to finish.

This is our case too: the real speedup from unrolling won't be fourfold, because the operations of incrementing the counter and checking if we are done are independent from the loop body, and can be scheduled to run concurrently with it. But may still be beneficial to [ask the compiler](#) to unroll it to some extent.

An Alternative Approach

You don't have to explicitly use `cmp` or a similar instruction to make a conditional jump. Many other instructions either read or modify the `FLAGS` register, sometimes as a by-product enabling optional exception checks.

For example, `add` always sets a number of flags, denoting whether the result is zero, is negative, whether an overflow or an underflow occurred, and so on. Taking advantage of this mechanism, compilers often produce loops like this:

```

mov rax, -100 ; replace 100 with the array size
loop:
    add edx, DWORD PTR [rax + 100 + rcx]
    add rax, 4
    jnz loop      ; checks if the result is zero

```

This code is a bit harder to read for a human, but it is one instruction shorter in the repeated part, which may meaningfully affect performance.

Functions and Recursion

To “call a function” in assembly, you need to `jump` to its beginning and then jump back. But then two important problems arise:

1. What if the caller stores data in the same registers as the callee?
2. Where is “back”?

Both of these concerns can be solved by having a dedicated location in memory where we can write all the information we need to return from the function before calling it. This location is called *the stack*.

The Stack

The hardware stack works the same way software stacks do and is similarly implemented as just two pointers:

- The *base pointer* marks the start of the stack and is conventionally stored in `rbp`.
- The *stack pointer* marks the last element of the stack and is conventionally stored in `rsp`.

When you need to call a function, you push all your local variables onto the stack (which you can also do in other circumstances; e.g., when you run out of registers), push the current instruction pointer, and then jump to the beginning of the function. When exiting from a function, you look at the pointer stored on top of the stack, jump there, and then carefully read all the variables stored on the stack back into their registers.

You can implement all that with the usual memory operations and jumps, but because of how frequently it is used, there are 4 special instructions for doing this:

- `push` writes data at the stack pointer and decrements it.
- `pop` reads data from the stack pointer and increments it.
- `call` puts the address of the following instruction on top of the stack and jumps to a label.
- `ret` reads the return address from the top of the stack and jumps to it.

You would call them “syntactic sugar” if they weren’t actual hardware instructions — they are just fused equivalents of these two-instruction snippets:

```
; "push rax"
sub rsp, 8
mov QWORD PTR[rsp], rax

; "pop rax"
mov rax, QWORD PTR[rsp]
add rsp, 8

; "call func"
push rip ; <- instruction pointer (although accessing it like that is probably illegal)
jmp func

; "ret"
pop rcx ; <- choose any unused register
jmp rcx
```

The memory region between `rbp` and `rsp` is called a *stack frame*, and this is where local variables of functions are typically stored. It is pre-allocated at the start of the program, and if you push more data on the stack than its capacity (8MB by default on Linux), you encounter a *stack overflow* error. Because modern operating systems don't actually give you memory pages until you read or write to their address space, you can freely specify a very large stack size, which acts more like a limit on how much stack memory can be used, and not a fixed amount every program has to use.

Calling Conventions

The people who develop compilers and operating systems eventually came up with [conventions](#) on how to write and call functions. These conventions enable some important [software engineering marvels](#) such as splitting compilation into separate units, reusing already-compiled libraries, and even writing them in different programming languages.

Consider the following example in C:

```
int square(int x) {
    return x * x;
}

int distance(int x, int y) {
    return square(x) + square(y);
}
```

By convention, a function should take its arguments in `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` (and the rest in the stack if those weren't enough), put the return value into `rax`, and then return. Thus, `square`, being a simple one-argument function, can be implemented like this:

```
square:           ; x = edi, ret = eax
    imul edi, edi
    mov eax, edi
    ret
```

Each time we call it from `distance`, we just need to go through some trouble preserving its local variables:

```
distance:         ; x = rdi/edi, y = rsi/esi, ret = rax/eax
    push rdi
    push rsi
    call square      ; eax = square(x)
    pop  rsi
    pop  rdi

    mov  ebx, eax    ; save x^2
    mov  rdi, rsi    ; move new x=y

    push rdi
    push rsi
    call square      ; eax = square(x=y)
    pop  rsi
    pop  rdi
```

```

add eax, ebx ;  $x^2 + y^2$ 
ret

```

There are a lot more nuances, but we won't go into detail here because this book is about performance, and the best way to deal with functions calls is actually to avoid making them in the first place.

Inlining

Moving data to and from the stack creates noticeable overhead for small functions like these. The reason you have to do this is that, in general, you don't know whether the callee is modifying the registers where you store your local variables. But when you have access to the code of `square`, you can solve this problem by stashing the data in registers that you know won't be modified.

```

distance:
    call square
    mov ebx, eax
    mov edi, esi
    call square
    add eax, ebx
    ret

```

This is better, but we are still implicitly accessing stack memory: you need to push and pop the instruction pointer on each function call. In simple cases like this, we can *inline* function calls by stitching the callee's code into the caller and resolving conflicts over registers. In our example:

```

distance:
    imul edi, edi      ; edi =  $x^2$ 
    imul esi, esi      ; esi =  $y^2$ 
    add edi, esi
    mov eax, edi        ; there is no "add eax, edi, esi", so we need a separate mov
    ret

```

This is fairly close to what optimizing compilers produce out of this snippet — only they use the **lea trick** to make the resulting machine code sequence a few bytes smaller:

```

distance:
    imul edi, edi      ; edi =  $x^2$ 
    imul esi, esi      ; esi =  $y^2$ 
    lea eax, [rdi+rsi] ; eax =  $x^2 + y^2$ 
    ret

```

In situations like these, function inlining is clearly beneficial, and compilers mostly do it [automatically](#), but there are cases when it's not — and we will talk about them [in a bit](#).

Tail Call Elimination

Inlining is straightforward to do when the callee doesn't make any other function calls, or at least if these calls are not recursive. Let's move on to a more complex example. Consider this recursive computation of a factorial:

```

int factorial(int n) {
    if (n == 0)
        return 1;
    return factorial(n - 1) * n;
}

```

Equivalent assembly:

```

; n = edi, ret = eax
factorial:
    test edi, edi      ; test if a value is zero
    jne nonzero        ; (the machine code of "cmp rax, 0" would be one byte longer)
    mov eax, 1         ; return 1
    ret
nonzero:
    push edi           ; save n to use later in multiplication
    sub edi, 1
    call factorial    ; call f(n - 1)
    pop edi
    imul eax, edi
    ret

```

If the function is recursive, it is still often possible to make it “call-less” by restructuring it. This is the case when the function is *tail recursive*, that is, it returns right after making a recursive call. Since no actions are required after the call, there is also no need for storing anything on the stack, and a recursive call can be safely replaced with a jump to the beginning — effectively turning the function into a loop.

To make our `factorial` function tail-recursive, we can pass a “current product” argument to it:

```

int factorial(int n, int p = 1) {
    if (n == 0)
        return p;
    return factorial(n - 1, p * n);
}

```

Then this function can be easily folded into a loop:

```

; assuming n > 0
factorial:
    mov eax, 1
loop:
    imul eax, edi
    sub edi, 1
    jne loop
    ret

```

The primary reason why recursion can be slow is that it needs to read and write data to the stack, while iterative and tail-recursive algorithms do not. This concept is very important in functional programming, where there are no loops and all you can use are functions. Without tail call elimination, functional programs would require way more time and memory to execute.

Indirect Branching

During assembly, all labels are converted to addresses (absolute or relative) and then encoded into jump instructions.

You can also jump by a non-constant value stored inside a register, which is called a *computed jump*:

```
jmp rax
```

This has a few interesting applications related to dynamic languages and implementing more complex control flow.

Multiway Branch

If you have already forgotten what a `switch` statement does, here is a little subroutine for calculating GPA in the American grading system:

```
switch (grade) {
    case 'A':
        return 4.0;
        break;
    case 'B':
        return 3.0;
        break;
    case 'C':
        return 2.0;
        break;
    case 'D':
        return 1.0;
        break;
    case 'E':
    case 'F':
        return 0.0;
        break;
    default:
        return NAN;
}
```

I personally don't remember the last time I used a switch in a non-educational context. In general, switch statements are equivalent to a sequence of "if, else if, else if, else if..." and so on, and for this reason many languages don't even have them. Nonetheless, such control flow structures are important for implementing parsers, interpreters, and other state machines, which are often comprised of a single `while (true)` loop and a `switch (state)` statement inside.

When we have control over the range of values that the variable can take, we can use the following trick utilizing computed jumps. Instead of making n conditional branches, we can create a *branch table* that contains pointers/offsets to possible jump locations, and then just index it with the `state` variable taking values in the $[0, n)$ range.

Compilers use this technique when the values are densely packed together (not necessarily strictly sequentially, but it has to be worth having blank fields in the table). It can also be implemented

explicitly with a *computed goto*:

```
void weather_in_russia(int season) {
    static const void* table[] = {&&winter, &&spring, &&summer, &&fall};
    goto *table[season];

    winter:
        printf("Freezing\n");
        return;
    spring:
        printf("Dirty\n");
        return;
    summer:
        printf("Dry\n");
        return;
    fall:
        printf("Windy\n");
        return;
}
```

Switch-based code is not always straightforward for compilers to optimize, so in the context of state machines, `goto` statements are often used directly. The I/O-related part of glibc is full of examples.

Dynamic Dispatch

Indirect branching is also instrumental in implementing runtime polymorphism.

Consider the cliché example when we have an abstract class of `Animal` with a virtual `.speak()` method, and two concrete implementations: a `Dog` that barks and a `Cat` that meows:

```
struct Animal {
    virtual void speak() { printf("<abstract animal sound>\n"); }
};

struct Dog {
    void speak() override { printf("Bark\n"); }
};

struct Cat {
    void speak() override { printf("Meow\n"); }
};
```

We want to create an animal and, without knowing its type in advance, call its `.speak()` method, which should somehow invoke the right implementation:

```
Dog sparkles;
Cat mittens;

Animal *catdog = (rand() & 1) ? &sparkles : &mittens;
catdog->speak();
```

There are many ways to implement this behavior, but C++ does it using a *virtual method table*.

For all concrete implementations of `Animal`, compiler pads all their methods (that is, their instruction sequences) so that they have the exact same length for all classes (by inserting some **filler instructions** after `ret`) and then just writes them sequentially somewhere in the instruction memory. Then it adds a *run-time type information* field to the structure (that is, to all its instances), which is essentially just the offset in the memory region that points to the right implementation of the virtual methods of the class.

With a virtual method call, that offset field is fetched from the instance of a structure and a normal function call is made with it, using the fact that all methods and other fields of every derived class have exactly the same offsets.

Of course, this adds some overhead:

- You may need to spend another 15 cycles or so for the same pipeline flushing reasons as for **branch misprediction**.
- The compiler most likely won't be able to inline the function call itself.
- Class size increases by a couple of bytes or so (this is implementation-specific).
- The binary size itself increases a little bit.

For these reasons, runtime polymorphism is usually avoided in performance-critical applications.

Interrupts and System Calls

```
global _start

section .text

_start:
    mov rax, 1          ; write(
    mov rdi, 1          ;     STDOUT_FILENO,
    mov rsi, msg        ;     "Hello, world!\n",
    mov rdx, msglen    ;     sizeof("Hello, world!\n")
    syscall             ; );

    mov rax, 60          ; exit(
    mov rdi, 0          ;     EXIT_SUCCESS
    syscall             ; );

section .rodata
    msg: db "Hello, world!", 10
    msglen: equ $ - msg
```

Interrupts are costly. They are not supposed to be on the normal execution path. Exceptions.

There is some overhead associated with doing system calls, so they are usually avoided. For example, all I/O is usually buffered, so that you send a single, say, 4KB piece of data to the OS.

Machine Code Layout

Computer engineers like to mentally split the [pipeline of a CPU](#) into two parts: the *front-end*, where instructions are fetched from memory and decoded, and the *back-end*, where they are scheduled and finally executed. Typically, the performance is bottlenecked by the execution stage, and for this reason, most of our efforts in this book are going to be spent towards optimizing around the back-end.

But sometimes the reverse can happen when the front-end doesn't feed instructions to the back-end fast enough to saturate it. This can happen for many reasons, all ultimately having something to do with how the machine code is laid out in memory, and affect performance in anecdotal ways, such as removing unused code, swapping "if" branches, or even changing the order of function declarations causing performance to either improve or deteriorate.

CPU Front-End

Before the machine code gets transformed into instructions, and the CPU understands what the programmer wants, it first needs to go through two important stages that we are interested in: *fetch* and *decode*.

During the **fetch** stage, the CPU simply loads a fixed-size chunk of bytes from the main memory, which contains the binary encodings of some number of instructions. This block size is typically 32 bytes on x86, although it may vary on different machines. An important nuance is that this block has to be [aligned](#): the address of the chunk must be multiple of its size (32B, in our case).

Next comes the **decode** stage: the CPU looks at this chunk of bytes, discards everything that comes before the instruction pointer, and splits the rest of them into instructions. Machine instructions are encoded using a variable number of bytes: something simple and very common like `inc rax` takes one byte, while some obscure instruction with encoded constants and behavior-modifying prefixes may take up to 15. So, from a 32-byte block, a variable number of instructions may be decoded, but no more than a certain machine-dependent limit called the *decode width*. On my CPU (a [Zen 2](#)), the decode width is 4, which means that on each cycle, up to 4 instructions can be decoded and passed to the next stage.

The stages work in a pipelined fashion: if the CPU can tell (or [predict](#)) which instruction block it needs next, then the fetch stage doesn't wait for the last instruction in the current block to be decoded and loads the next one right away.

Code Alignment

Other things being equal, compilers typically prefer instructions with shorter machine code, because this way more instructions can fit in a single 32B fetch block, and also because it reduces the size of the binary. But sometimes the reverse is preferable, due to the fact that the fetched instructions' blocks must be aligned.

Imagine that you need to execute an instruction sequence that starts on the last byte of a 32B-aligned block. You may be able to execute the first instruction without additional delay, but for the subsequent ones, you have to wait for one additional cycle to do another instruction fetch. If the code block was aligned on a 32B boundary, then up to 4 instructions could be decoded and then executed concurrently (unless they are extra long or interdependent).

Having this in mind, compilers often do a seemingly harmful optimization: they sometimes prefer instructions with longer machine codes, and even insert dummy instructions that do nothing³ in order to get key jump locations aligned on a suitable power-of-two boundary.

In GCC, you can use `-falign-labels=n` flag to specify a particular alignment policy, replacing `-labels` with `-function`, `-loops`, or `-jumps` if you want to be more selective. On `-O2` and `-O3` levels of optimization, it is enabled by default — without setting a particular alignment, in which case it uses a (usually reasonable) machine-dependent default value.

Instruction Cache

The instructions are stored and fetched using largely the same [memory system](#) as for the data, except maybe the lower layers of cache are replaced with a separate *instruction cache* (because you wouldn't want a random data read to kick out the code that processes it).

The instruction cache is crucial in situations when you either:

- don't know what instructions you are going to execute next, and need to fetch the next block with [low latency](#),
- or are executing a long sequence of verbose-but-quick-to-process instructions, and need [high bandwidth](#).

The memory system can therefore become the bottleneck for programs with large machine code. This consideration limits the applicability of the optimization techniques we've previously discussed:

- [Inlining functions](#) is not always optimal, because it reduces code sharing and increases the binary size, requiring more instruction cache.
- [Unrolling loops](#) is only beneficial up to some extent, even if the number of iterations is known during compile time: at some point, the CPU would have to fetch both instructions and data from the main memory, in which case it will likely be bottlenecked by the memory bandwidth.
- Huge [code alignments](#) increase the binary size, again requiring more instruction cache. Spending one more cycle on fetch is a minor penalty compared to missing the cache and waiting for the instructions to be fetched from the main memory.

Another aspect is that placing frequently used instruction sequences on the same [cache lines](#) and [memory pages](#) improves [cache locality](#). To improve instruction cache utilization, you should group hot code with hot code and cold code with cold code, and remove dead (unused) code if possible. If you want to explore this idea further, check out Facebook's [Binary Optimization and Layout Tool](#), which was recently [merged](#) into LLVM.

Unequal Branches

Suppose that for some reason you need a helper function that calculates the length of an integer interval. It takes two arguments, x and y , but for convenience, it may correspond to either $[x, y]$ or $[y, x]$, depending on which one is non-empty. In plain C, you would probably write something like this:

```
int length(int x, int y) {
    if (x > y)
```

³Such instructions are called no-op, or NOP instructions. On x86, the “official way” of doing nothing is `xchq rax, rax` (swap a register with itself): the CPU recognizes it and doesn't spend extra cycles executing it, except for the decode stage. The `nop` shorthand maps to the same machine code.

```

        return x - y;
    else
        return y - x;
}

```

In x86 assembly, there is a lot more variability to how you can implement it, noticeably impacting performance. Let's start with trying to map this code directly into assembly:

```

length:
    cmp edi, esi
    jle less
    ; x > y
    sub edi, esi
    mov eax, edi
done:
    ret
less:
    ; x <= y
    sub esi, edi
    mov eax, esi
    jmp done

```

While the initial C code seems very symmetrical, the assembly version isn't. This results in an interesting quirk that one branch can be executed slightly faster than the other: if $x > y$, then the CPU can just execute the 5 instructions between `cmp` and `ret`, which, if the function is aligned, are all going to be fetched in one go; while in case of $x \leq y$, two more jumps are required.

It may be reasonable to assume that the $x > y$ case is *unlikely* (why would anyone calculate the length of an inverted interval?), more like an exception that mostly never happens. We can detect this case, and simply swap x and y :

```

int length(int x, int y) {
    if (x > y)
        swap(x, y);
    return y - x;
}

```

The assembly would go like this, as it typically does for the if-without-else patterns:

```

length:
    cmp edi, esi
    jle normal      ; if x <= y, no swap is needed, and we can skip the xchg
    xchg edi, esi
normal:
    sub esi, edi
    mov eax, esi
    ret

```

The total instruction length is 6 now, down from 8. But it is still not quite optimized for our assumed case: if we think that $x > y$ never happens, then we are wasteful when loading the `xchg edi, esi` instruction that is never going to be executed. We can solve this by moving it outside the normal execution path:

```

length:
    cmp edi, esi
    jg swap
normal:
    sub esi, edi
    mov eax, esi
    ret
swap:
    xchg edi, esi
    jmp normal

```

This technique is quite handy when handling exceptions cases in general, and in high-level code, you can give the compiler a [hint](#) that a certain branch is more likely than the other:

```

int length(int x, int y) {
    if (x > y) [[unlikely]]
        swap(x, y);
    return y - x;
}

```

This optimization is only beneficial when you know that a branch is very rarely taken. When this is not the case, there are [other aspects](#) more important than the code layout, that compel compilers to avoid any branching at all — in this case by replacing it with a special “conditional move” instruction, roughly corresponding to the ternary expression ($x > y ? y - x : x - y$) or calling `abs(x - y)`:

```

length:
    mov edx, edi
    mov eax, esi
    sub edx, esi
    sub eax, edi
    cmp edi, esi
    cmovg eax, edx ; "mov if edi > esi"
    ret

```

Eliminating branches is an important topic, and we will spend [much of the next chapter](#) discussing it in more detail.

Chapter 3: Instruction-Level Parallelism

Algorithms for Modern Hardware

Contents

Pipeline Hazards	1
The Cost of Branching	2
An Experiment	2
Branch Prediction	3
Pattern Detection	4
Hinting Likeliness of Branches	4
Acknowledgements	5
Instruction-Level Parallelism	5
Instruction Pipelining	5
An Education Analogy	6
Branchless Programming	7
Predication	7
When Predication Is Beneficial	9
Larger Examples	9
Instruction Tables	11
Instruction Scheduling	11
Superscalar Processors	12
Microcode	12
Instruction Scheduling	12
Throughput Computing	13
Example	13
The General Case	14
Theoretical Performance Limits	14
Decode Width	14
Memory-Bandwidth Algorithm	15
Memory-Latency	15
Comparison-Based Sorting	15
Linear Algebra	15

Pipeline Hazards

Pipelining lets you hide the latencies of instructions by running them concurrently, but also creates some potential obstacles of its own — characteristically called *pipeline hazards*, that is, situations when the next instruction cannot execute on the following clock cycle.

There are multiple ways this may happen:

- A *structural hazard* happens when two or more instructions need the same part of CPU (e.g., an execution unit).
- A *data hazard* happens when you have to wait for an operand to be computed from some previous step.
- A *control hazard* happens when a CPU can't tell which instructions it needs to execute next.

The only way to resolve a hazard is to have a *pipeline stall*: stop the progress of all previous steps until the cause of congestion is gone. This creates *bubbles* in the pipeline — analogous with air bubbles in fluid pipes — a time-propagating condition when execution units are idling and no useful work is done.

Pipeline stall on the execution stage

Figure 1: Pipeline stall on the execution stage

Different hazards have different penalties:

- In structural hazards, you have to wait (usually one more cycle) until the execution unit is ready. They are fundamental bottlenecks on performance and can't be avoided — you have to engineer around them.
- In data hazards, you have to wait for the required data to be computed (the latency of the *critical path*). Data hazards are solved by restructuring computations so that the critical path is shorter.
- In control hazards, you generally have to flush the entire pipeline and start over, wasting a whole 15-20 cycles. They are solved by either removing branches completely, or making them predictable so that the CPU can effectively *speculate* on what is going to be executed next.

As they have very different impacts on performance, we are going to go in the reversed order and start with the more grave ones.

The Cost of Branching

When a CPU encounters a conditional jump or [any other type of branching](#), it doesn't just sit idle until its condition is computed — instead, it starts *speculatively executing* the branch that seems more likely to be taken immediately. During execution, the CPU computes statistics about branches taken on each instruction, and after some time, they start to predict them by recognizing common patterns.

For this reason, the true “cost” of a branch largely depends on how well it can be predicted by the CPU. If it is a pure 50/50 coin toss, you have to suffer a **control hazard** and discard the entire pipeline, taking another 15-20 cycles to build up again. And if the branch is always or never taken, you pay almost nothing except checking the condition.

An Experiment

As a case study, we are going to create an array of random integers between 0 and 99 inclusive:

```
for (int i = 0; i < N; i++)
    a[i] = rand() % 100;
```

Then we create a loop where we sum up all its elements under 50:

```

volatile int s;

for (int i = 0; i < N; i++)
    if (a[i] < 50)
        s += a[i];

```

We set $N = 10^6$ and run this loop many times over so that the [cold cache](#) effect doesn't mess up our results. We mark our accumulator variable as `volatile` so that the compiler doesn't vectorize the loop, interleave its iterations, or "cheat" in any other way.

On Clang, this produces assembly that looks like this:

```

mov rcx, -4000000
jmp body
counter:
    add rcx, 4
    jz finished ; "jump if rcx became zero"
body:
    mov edx, dword ptr [rcx + a + 4000000]
    cmp edx, 49
    jg counter
    add dword ptr [rsp + 12], edx
    jmp counter

```

Our goal is to simulate a completely unpredictable branch, and we successfully achieve it: the code takes ~ 14 CPU cycles per element. For a very rough estimate of what it is supposed to be, we can assume that the branches alternate between `<` and `\geq` , and the pipeline is mispredicted every other iteration. Then, every two iterations:

- We discard the pipeline, which is 19 cycles deep on Zen 2 (i.e., it has 19 stages, each taking one cycle).
- We need a memory fetch and a comparison, which costs ~ 5 cycles. We can check the conditions of even and odd iterations concurrently, so let's assume we only pay it once per 2 iterations.
- In the case of the `<` branch, we need another ~ 4 cycles to add `a[i]` to a volatile (memory-stored) variable `s`.

Therefore, on average, we need to spend $(4 + 5 + 19)/2 = 14$ cycles per element, matching what we measured.

Branch Prediction

We can replace the hardcoded 50 with a tweakable parameter `P` that effectively sets the probability of the `<` branch:

```

for (int i = 0; i < N; i++)
    if (a[i] < P)
        s += a[i];

```

Now, if we benchmark it for different values of `P`, we get an interesting-looking graph:

Its peak is at 50-55%, as expected: branch misprediction is the most expensive thing here. This graph is asymmetrical: it takes just ~ 1 cycle to only check conditions that are never satisfied (`P = 0`).

0), and ~ 7 cycles for the sum if the branch is always taken ($P = 100$).

This graph is not unimodal: there is another local minimum at around 85-90%. We spend ~ 6.15 cycles per element there or about 10-15% faster than when we always take the branch, accounting for the fact that we need to perform fewer additions. Branch misprediction stops affecting the performance at this point because when it happens, not the whole instruction buffer is discarded, but only the operations that were speculatively scheduled. Essentially, that 10-15% mispredict rate is the equilibrium point where we can see far enough in the pipeline not to stall but still save 10-15% on taking the cheaper \geq branch.

Note that it costs almost nothing to check for a condition that never or almost never occurs. This is why programmers use runtime exceptions and base case checks so profusely: if they are indeed rare, they don't really cost anything.

Pattern Detection

In our example, everything that was needed for efficient branch prediction is a hardware statistics counter. If we historically took branch A more often than branch B, then it makes sense to speculatively execute branch A. But branch predictors on modern CPUs are considerably more advanced than that and can detect much more complicated patterns.

Let's fix P back at 50, and then sort the array first before the main summation loop:

```
for (int i = 0; i < N; i++)
    a[i] = rand() % 100;

std::sort(a, a + n);
```

We are still processing the same elements, but in a different order, and instead of 14 cycles, it now runs in a little bit more than 4, which is exactly the average of the cost of the pure $<$ and \geq branches.

The branch predictor can pick up on much more complicated patterns than just “always left, then always right” or “left-right-left-right.” If we just decrease the size of the array N to 1000 (without sorting it), then the branch predictor memorizes the entire sequence of comparisons, and the benchmark again measures at around 4 cycles — in fact, even slightly fewer than in the sorted array case, because in the former case branch predictor needs to spend some time flicking between the “always yes” and “always no” states.

Hinting Likeliness of Branches

If you know beforehand which branch is more likely, it may be beneficial to [pass that information](#) to the compiler:

```
for (int i = 0; i < N; i++)
    if (a[i] < P) [[likely]]
        s += a[i];
```

When $P = 75$, it measures around ~ 7.3 cycles per element, while the original version without the hint needs ~ 8.3 .

This hint does not eliminate the branch or communicate anything to the branch predictor, but it changes the [machine code layout](#) in a way that lets the CPU front-end process the more likely

branch slightly faster (although usually by no more than one cycle).

This optimization is only beneficial when you know which branch is more likely to be taken before the compilation stage. When the branch is fundamentally unpredictable, we can try to remove it completely using *predication* — a profoundly important technique that we are going to explore in the next section.

Acknowledgements

This case study is inspired by [the most upvoted Stack Overflow question ever](#).

Instruction-Level Parallelism

When programmers hear the word *parallelism*, they mostly think about *multi-core parallelism*, the practice of explicitly splitting a computation into semi-independent *threads* that work together to solve a common problem.

This type of parallelism is mainly about reducing *latency* and achieving *scalability*, but not about improving *efficiency*. You can solve a problem ten times as big with a parallel algorithm, but it would take at least ten times as many computational resources. Although parallel hardware is becoming [ever more abundant](#) and parallel algorithm design is becoming an increasingly important area, for now, we will limit ourselves to considering only a single CPU core.

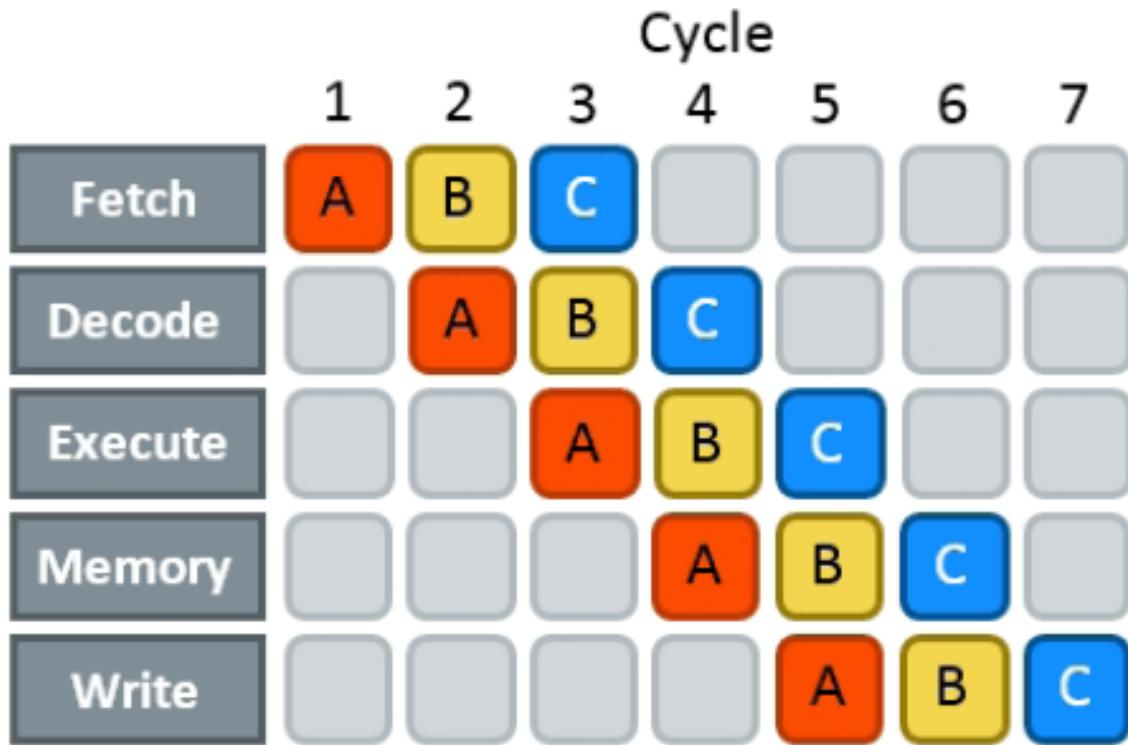
But there are other types of parallelism, already existing inside a CPU core, that you can use *for free*.

Instruction Pipelining

To execute *any* instruction, processors need to do a lot of preparatory work first, which includes:

- **fetching** a chunk of machine code from memory,
- **decoding** it and splitting into instructions,
- **executing** these instructions, which may involve doing some **memory** operations, and
- **writing** the results back into registers.

This whole sequence of operations is *long*. It takes up to 15-20 CPU cycles even for something simple like `add`-ing two register-stored values together. To hide this latency, modern CPUs use *pipelining*: after an instruction passes through the first stage, they start processing the next one right away, without waiting for the previous one to fully complete.



Pipelining does not reduce *actual* latency but functionally makes it seem like if it was composed of only the execution and memory stage. You still need to pay these 15-20 cycles, but you only need to do it once after you've found the sequence of instructions you are going to execute.

Having this in mind, hardware manufacturers prefer to use *cycles per instruction* (CPI) instead of something like “average instruction latency” as the main performance indicator for CPU designs. It is a [pretty good metric](#) for algorithm designs too, if we only consider *useful* instructions.

The CPI of a perfectly pipelined processor should tend to one, but it can actually be even lower if we make each stage of the pipeline “wider” by duplicating it, so that more than one instruction can be processed at a time. Because the cache and most of the ALU can be shared, this ends up being cheaper than adding a fully separate core. Such architectures, capable of executing more than one instruction per cycle, are called *superscalar*, and most modern CPUs are.

You can only take advantage of superscalar processing if the stream of instructions contains groups of logically independent operations that can be processed separately. The instructions don't always arrive in the most convenient order, so, when possible, modern CPUs can execute them *out of order* to improve overall utilization and minimize pipeline stalls. How this magic works is a topic for a more advanced discussion, but for now, you can assume that the CPU maintains a buffer of pending instructions up to some distance in the future, and executes them as soon as the values of its operands are computed and there is an execution unit available.

An Education Analogy

Consider how our education system works:

1. Topics are taught to groups of students instead of individuals as broadcasting the same things to everyone at once is more efficient.
2. An intake of students is split into groups led by different teachers; assignments and other course materials are shared between groups.
3. Each year the same course is taught to a new intake so that the teachers are kept busy.

These innovations greatly increase the *throughput* of the whole system, although the *latency* (time to graduation for a particular student) remains unchanged (and maybe increases a little bit because personalized tutoring is more effective).

You can find many analogies with modern CPUs:

1. CPUs use **SIMD parallelism** to execute the same operation on a block of different data points (comprised of 16, 32, or 64 bytes).
2. There are multiple execution units that can process these instructions simultaneously while sharing other CPU facilities (usually 2-4 execution units).
3. Instructions are processed in pipelined fashion (saving roughly the same number of cycles as the number of years between kindergarten and PhD).

In addition to that, several other aspects also match:

- Execution paths become more divergent with time and need different execution units.
- Some instructions may be stalled for various reasons.
- Some instructions are even speculated (executed ahead of time), but then discarded.
- Some instructions may be split in several distinct micro-operations that can proceed on their own.

Programming pipelined and superscalar processors presents its own challenges, which we are going to address in this chapter.

Branchless Programming

As we established in [the previous section](#), branches that can't be effectively predicted by the CPU are expensive as they may cause a long pipeline stall to fetch new instructions after a branch mispredict. In this section, we discuss the means of removing branches in the first place.

Predication

We are going to continue the same case study we've started before — we create an array of random numbers and sum up all its elements below 50:

```
for (int i = 0; i < N; i++)
    a[i] = rand() % 100;
```

```
volatile int s;
```

```
for (int i = 0; i < N; i++)
    if (a[i] < 50)
        s += a[i];
```

Our goal is to eliminate the branch caused by the `if` statement. We can try to get rid of it like this:

```

for (int i = 0; i < N; i++)
    s += (a[i] < 50) * a[i];

```

The loop now takes ~7 cycles per element instead of the original ~14. Also, the performance remains constant if we change 50 to some other threshold, so it doesn't depend on the branch probability.

But wait... shouldn't there still be a branch? How does `(a[i] < 50)` map to assembly?

There are no Boolean types in assembly, nor any instructions that yield either one or zero based on the result of the comparison, but we can compute it indirectly like this: `(a[i] - 50) >> 31`. This trick relies on the [binary representation of integers](#), specifically on the fact that if the expression `a[i] - 50` is negative (implying `a[i] < 50`), then the highest bit of the result will be set to one, which we can then extract using a right-shift.

```

mov ebx, eax ; t = x
sub ebx, 50 ; t -= 50
sar ebx, 31 ; t >= 31
imul eax, ebx ; x *= t

```

Another, more complicated way to implement this whole sequence is to convert this sign bit into a mask and then use bitwise `and` instead of multiplication: `((a[i] - 50) >> 31 - 1) & a[i]`. This makes the whole sequence one cycle faster, considering that, unlike other instructions, `imul` takes 3 cycles:

```

mov ebx, eax ; t = x
sub ebx, 50 ; t -= 50
sar ebx, 31 ; t >= 31
; imul eax, ebx ; x *= t
sub ebx, 1 ; t -= 1 (causing underflow if t = 0)
and eax, ebx ; x &= t

```

Note that this optimization is not technically correct from the compiler's perspective: for the 50 lowest representable integers — those in the $[-2^{31}, -2^{31} + 49]$ range — the result will be wrong due to underflow. We know that all numbers are all between 0 and 100, and this won't happen, but the compiler doesn't.

But the compiler actually elects to do something different. Instead of going with this arithmetic trick, it used a special `cmove` ("conditional move") instruction that assigns a value based on a condition (which is computed and checked using the flags register, the same way as for jumps):

```

mov     ebx, 0      ; cmove doesn't support immediate values, so we need a zero register
cmp     eax, 50
cmovege eax, ebx    ; eax = (eax >= 50 ? eax : ebx=0)

```

So the code above is actually closer to using a ternary operator like this:

```

for (int i = 0; i < N; i++)
    s += (a[i] < 50 ? a[i] : 0);

```

Both variants are optimized by the compiler and produce the following assembly:

```

mov     eax, 0
mov     ecx, -4000000
loop:

```

```

mov    esi, dword ptr [rdx + a + 4000000] ; load a[i]
cmp    esi, 50
cmovge esi, eax                           ; esi = (esi >= 50 ? esi : eax=0)
add    dword ptr [rsp + 12], esi           ; s += esi
add    rdx, 4
jnz    loop                                ; "iterate while rdx is not zero"

```

This general technique is called *predication*, and it is roughly equivalent to this algebraic trick:

$$x = c \cdot a + (1 - c) \cdot b$$

This way you can eliminate branching, but this comes at the cost of evaluating *both* branches and the `cmove` itself. Because evaluating the “ \geq ” branch costs nothing, the performance is exactly equal to the “always yes” case in the branchy version.

When Predication Is Beneficial

Using predication eliminates a control hazard but introduces a data hazard. There is still a pipeline stall, but it is a cheaper one: you only need to wait for `cmove` to be resolved and not flush the entire pipeline in case of a mispredict.

However, there are many situations when it is more efficient to leave branchy code as it is. This is the case when the cost of computing *both* branches instead of just *one* outweighs the penalty for the potential branch mispredictions.

In our example, the branchy code wins when the branch can be predicted with a probability of more than ~75%.

This 75% threshold is commonly used by the compilers as a heuristic for determining whether to use the `cmove` or not. Unfortunately, this probability is usually unknown at the compile time, so it needs to be provided in one of several ways:

- We can use [profile-guided optimization](#) which will decide for itself whether to use predication or not.
- We can use [likeliness attributes](#) and [compiler-specific intrinsics](#) to hint at the likeliness of branches: `__builtin_expect_with_probability` in GCC and `__builtin_unpredictable` in Clang.
- We can rewrite branchy code using the ternary operator or various arithmetic tricks, which acts as sort of an implicit contract between programmers and compilers: if the programmer wrote the code this way, then it was probably meant to be branchless.

The “right way” is to use branching hints, but unfortunately, the support for them is lacking. Right now [these hints seem to be lost](#) by the time the compiler back-end decides whether a `cmove` is more beneficial. There is [some progress](#) towards making it possible, but currently, there is no good way of forcing the compiler to generate branch-free code, so sometimes the best hope is to just write a small snippet in assembly.

Larger Examples

Strings. Oversimplifying things, an `std::string` is comprised of a pointer to a null-terminated `char` array (also known as a “C-string”) allocated somewhere on the heap and one integer containing

the string size.

A common value for a string is the empty string — which is also its default value. You also need to handle them somehow, and the idiomatic approach is to assign `nullptr` as the pointer and 0 as the string size, and then check if the pointer is null or if the size is zero at the beginning of every procedure involving strings.

However, this requires a separate branch, which is costly (unless the majority of strings are either empty or non-empty). To remove the check and thus also the branch, we can allocate a “zero C-string,” which is just a zero byte allocated somewhere, and then simply point all empty strings there. Now all string operations with empty strings have to read this useless zero byte, but this is still much cheaper than a branch misprediction.

Binary search. The standard binary search [can be implemented](#) without branches, and on small arrays (that fit into cache) it works ~4x faster than the branchy `std::lower_bound`:

```
int lower_bound(int x) {
    int *base = t, len = n;
    while (len > 1) {
        int half = len / 2;
        base += (base[half - 1] < x) * half; // will be replaced with a "cmov"
        len -= half;
    }
    return *base;
}
```

Other than being more complex, it has another slight drawback in that it potentially does more comparisons (constant $\lceil \log_2 n \rceil$ instead of either $\lfloor \log_2 n \rfloor$ or $\lceil \log_2 n \rceil$) and can't speculate on future memory reads (which acts as prefetching, so it loses on very large arrays).

In general, data structures are made branchless by implicitly or explicitly *padding* them so that their operations take a constant number of iterations. Refer to [the article](#) for more complex examples.

Data-parallel programming. Branchless programming is very important for [SIMD](#) applications because they don't have branching in the first place.

In our array sum example, removing the `volatile` type qualifier from the accumulator allows the compiler to [vectorize](#) the loop:

```
/* volatile */ int s = 0;

for (int i = 0; i < N; i++)
    if (a[i] < 50)
        s += a[i];
```

It now works in ~0.3 per element, which is mainly [bottlenecked by the memory](#).

The compiler is usually able to vectorize any loop that doesn't have branches or dependencies between the iterations — and some specific small deviations from that, such as [reductions](#) or simple loops that contain just one if-without-else. Vectorization of anything more complex is a very nontrivial problem, which may involve various techniques such as [masking](#) and [in-register permutations](#).

Instruction Tables

Interleaving the stages of execution is a general idea in digital electronics, and it is applied not only in the main CPU pipeline, but also on the level of separate instructions and [memory](#). Most execution units have their own little pipelines and can take another instruction just one or two cycles after the previous one.

In this context, it makes sense to use two different “[costs](#)” for instructions:

- *Latency*: how many cycles are needed to receive the results of an instruction.
- *Throughput*: how many instructions can be, on average, executed per cycle.

You can get latency and throughput numbers for a specific architecture from special documents called [instruction tables](#). Here are some sample values for my Zen 2 (all specified for 32-bit operands, if there is any difference):

Instruction	Latency	RThroughput
jmp	-	2
mov r, r	-	1/4
mov r, m	4	1/2
mov m, r	3	1
add	1	1/3
cmp	1	1/4
popcnt	1	1/4
mul	3	1
div	13-28	13-28

Some comments:

- Because our minds are so used to the cost model where “more” means “worse,” people mostly use *reciprocals* of throughput instead of throughput.
- If a certain instruction is especially frequent, its execution unit could be duplicated to increase its throughput — possibly to even more than one, but not higher than the [decode width](#).
- Some instructions have a latency of 0. This means that these instruction are used to control the scheduler and don’t reach the execution stage. They still have non-zero reciprocal throughput because the [CPU front-end](#) still needs to process them.
- Most instructions are pipelined, and if they have the reciprocal throughput of n , this usually means that their execution unit can take another instruction after n cycles (and if it is below 1, this means that there are multiple execution units, all capable of taking another instruction on the next cycle). One notable exception is [integer division](#): it is either very poorly pipelined or not pipelined at all.
- Some instructions have variable latency, depending on not only the size, but also the values of the operands. For memory operations (including fused ones like `add`), the latency is usually specified for the best case (an L1 cache hit).

There are many more important little details, but this mental model will suffice for now.

Instruction Scheduling

Let’s dive a bit deeper.

Superscalar Processors

CPI of a perfectly pipelined processor should tend to one, but it can actually be even lower than one.

As there are many different instructions, It is very common for programs to have groups of logically independent operations that can be processed separately by different execution units. To improve their utilization, we can duplicate everything else the pipeline so that more than one instruction is processed in a time, and then, if possible, schedule the instructions on different parts of the ALU. Such architectures, capable of executing more than one, are called *superscalar*, and most modern CPUs are.

Interleaving the stages of execution is a general idea in digital electronics, and it is applied not only in the main CPU pipeline, but also on the level of separate instructions and [memory](#). Most execution units have their own little pipelines, and can take another instruction just one or two cycles after the previous one. If a certain instruction is frequently used, it makes sense to duplicate its execution unit also, and also place frequently jointly used instructions on the same execution unit: e.g., not using the same for arithmetic and memory operation.

Microcode

While complex instruction sets had the benefit, with superscalar processors you want your instructions to be as tiny and atomic as possible. Such as fused add instruction. This also provides a simple way to “retire” old instructions that nobody is using and you don’t want to support anymore: just replace them with a hundred or so microcoded instruction. They are not used anyway.

Instructions are microcoded.

uOps (“micro-ops,” the first letter is meant to be greek letter mu as in us (microsecond), but nobody cares enough to type it).

Each architecture has its own set of “ports,” each capable of executing its own set of instructions (uOps, to be more exact).

But still, when you use it, it appears and feels like a single instruction. How does CPU achieve that?

Instruction Scheduling

This poses some additional challenges in coordinating how to execute instruction and in which order. This is why modern schedulers take more die space than the entirety of the integer ALU. They are insanely complex, but this mental model works good enough most of the time.

Modern processors don’t actually execute instructions one-by-one, but maintain a *pipeline* of pending instructions so that two independent operations can be executed concurrently without waiting for each other to finish.

Out-of-order execution. A buffer of pending instructions.

A bit more precisely, the CPU will look at the instruction stream up to some distance in the future. If there are branches, it will do branch prediction to produce a sequential stream of instructions. Then it will see which of the instructions are ready for execution. For example, if it sees a future instruction X that only uses registers A and B, and there are no instructions before it that touch

those registers, and none of the instructions that are currently in the pipeline modify those registers, either, then it is safe to start to execute X as soon as there is an execution unit that is available.

All of this happens in the hardware, all the time, fully automatically. The only thing that the programmer needs to do is to make sure there are sufficiently many independent instructions always available for execution. The magic takes place inside the CPU. The compiler just produces two machine language instructions, without any special annotation that indicates whether or not these instructions can be executed in parallel. The CPU will then automatically figure out which of the instructions can be executed in parallel.

You can schedule independent instructions separately, but only up to some extent. This buffer is large, hundreds of operations. Still not enough for something extra long like main memory accesses.

Throughput Computing

Optimizing for *latency* is usually quite different from optimizing for *throughput*:

- When optimizing data structure queries or small one-time or branchy algorithms, you need to **look up the latencies** of its instructions, mentally construct the execution graph of the computation, and then try to reorganize it so that the critical path is shorter.
- When optimizing hot loops and large-dataset algorithms, you need to look up the throughputs of their instructions, count how many times each one is used per iteration, determine which of them is the bottleneck, and then try to restructure the loop so that it is used less often.

The last advice only works for *data-parallel* loops, where each iteration is fully independent of the previous one. When there is some interdependency between consecutive iterations, there may potentially be a pipeline stall caused by a **data hazard** as the next iteration is waiting for the previous one to complete.

Example

As a simple example, consider how the sum of an array is computed:

```
int s = 0;  
  
for (int i = 0; i < n; i++)  
    s += a[i];
```

Let's assume for a moment that the compiler doesn't **vectorize** this loop, **the memory bandwidth** isn't a concern, and that the loop is **unrolled** so that we don't pay any additional cost associated with maintaining the loop variables. In this case, the computation becomes very simple:

```
int s = 0;  
s += a[0];  
s += a[1];  
s += a[2];  
s += a[3];  
// ...
```

How fast can we compute this? At exactly one cycle per element — because we need one cycle each iteration to **add** another value to **s**. The latency of the memory read doesn't matter because the CPU can start it ahead of time.

But we can go higher than that. The *throughput* of `add`¹ is 2 on my CPU (Zen 2), meaning we could theoretically execute two of them every cycle. But right now this isn't possible: while `s` is being used to accumulate i -th element, it can't be used for $(i + 1)$ -th for at least one cycle.

The solution is to use *two* accumulators and just sum up odd and even elements separately:

```
int s0 = 0, s1 = 0;
s0 += a[0];
s1 += a[1];
s0 += a[2];
s1 += a[3];
// ...
int s = s0 + s1;
```

Now our superscalar CPU can execute these two “threads” simultaneously, and our computation no longer has any critical paths that limit the throughput.

The General Case

If an instruction has a latency of x and a throughput of y , then you would need to use $x \cdot y$ accumulators to saturate it. This also implies that you need $x \cdot y$ logical registers to hold their values, which is an important consideration for CPU designs, limiting the maximum number of usable execution units for high-latency instructions.

This technique is mostly used with [SIMD](#) and not in scalar code. You can [generalize](#) the code above and compute sums and other reductions faster than the compiler.

In general, when optimizing loops, you usually have just one or a few *execution ports* that you want to utilize to their fullest, and you engineer the rest of the loop around them. As different instructions may use different sets of ports, it is not always clear which one is going to be overused. In situations like this, [machine code analyzers](#) can be very helpful for finding the bottlenecks of small assembly loops.

Theoretical Performance Limits

Do I want to talk about it here or in ILP section?

- Decode width
- Throughput of a certain instruction or, more precisely, an execution port
- Instruction Latency (on critical path)
- Cache/Memory latency
- Cache/Memory throughput

A very good model is identifying the bottleneck and then working around it.

Decode Width

Add two arrays together and write into a third.

Thanks to fused operations.

¹The throughput of register-register `add` is 4, but since we are reading its second operand from memory, it is bottlenecked by the throughput of memory `mov`, which is 2 on Zen 2.

Memory-Bandwidth Algorithm

Single-pass SIMD algorithms are going to be bottlenecked by memory.

Memory-Latency

You need n random memory accesses. Each .

However hard you try, you can't make the latency lower than the slowest memory read.

Comparison-Based Sorting

Linear Algebra

There is an FMA instruction.

This is the number usually reported.

Exercise: theoretical peak performance. By the way, assuming infinite bandwidth, what would the throughput of that loop be? How to verify that the 14 GFLOPS figure is the CPU limit and not L1 peak bandwidth? For that we need to look a bit closer at how the processor will execute the loop.

Incrementing an array can be done with SIMD; when compiled, it uses just two operations per 8 elements — performing the read-fused addition and writing the result back:

```
vpaddd  ymm0, ymm1, YMMWORD PTR [rax]  
vmovdqa YMMWORD PTR [rax], ymm0
```

This computation is bottlenecked by the write, which has a throughput of 1. This means that we can theoretically increment and write back 8 values per cycle on average, yielding the performance of $2 \text{ GHz} \times 8 = 16 \text{ GFLOPS}$ (or 32.8 in boost mode), which is fairly close to what we observed.

On all modern architectures, you can typically assume that you won't ever be bottlenecked by the throughput of L1 cache, but rather by the read/write execution ports or the arithmetic. In these extreme cases, it may be beneficial to store some data in registers without touching any of the memory, which we will cover later in the book.

Chapter 4: Compilation

Algorithms for Modern Hardware

Contents

Stages of Compilation	1
Interprocedural Optimization	2
Inspecting the Output	2
Flags and Targets	3
Optimization Levels	3
Specifying Targets	3
Multiversioned Functions	4
Situational Optimizations	4
Loop Unrolling	4
Function Inlining	5
Likeliness of Branches	5
Profile-Guided Optimization	5
Compilation	6
Contract Programming	6
Why Undefined Behavior Exists	7
Removing Corner Cases	8
Assumptions	8
Arithmetic	8
Memory Aliasing	9
C++ Contracts	10
Non-Zero-Cost Abstractions	11
Memory	11
Precomputation	12
Constant Expressions	13
Arithmetic Optimizations	14
What Compilers Can and Can't Do	14
Checklist	15

Stages of Compilation

Before jumping straight to compiler optimizations, which is what most of this chapter is about, let's briefly recap the “big picture” first. Skipping the boring parts, there are 4 stages of turning C programs into executables:

1. **Preprocessing** expands macros, pulls included source from header files, and strips off comments from source code: `gcc -E source.c` (outputs preprocessed source to stdout)

2. **Compiling** parses the source, checks for syntax errors, converts it into an intermediate representation, performs optimizations, and finally translates it into assembly language: `gcc -S file.c` (emits an `.s` file)
3. **Assembly** turns assembly language into machine code, except that any external function calls like `printf` are substituted with placeholders: `gcc -c file.c` (emits an `.o` file, called *object file*)
4. **Linking** finally resolves the function calls by plugging in their actual addresses, and produces an executable binary: `gcc -o binary file.c`

There are possibilities to improve program performance in each of these stages.

Interprocedural Optimization

We have the last **stage**, linking, because it is both easier and faster to compile programs on a file-by-file basis and then link those files together — this way you can do this in parallel and also cache intermediate results.

It also gives the ability to distribute code as *libraries*, which can be either *static* or *shared*:

- *Static* libraries are simply collections of precompiled object files that are merged with other sources by the compiler to produce a single executable, just as it normally would.
- *Dynamic* or *shared* libraries are precompiled executables that have additional meta-information about where their callables are, references to which are resolved during runtime. As the name suggests, this allows *sharing* the compiled binaries between multiple programs.

The main advantage of using static libraries is that you can perform various *interprocedural optimizations* that require more context than just the signatures of library functions, such as [function inlining](#) or dead code elimination. To force the linker to look for and only accept static libraries, you can pass the `-static` option.

This process is called *link-time optimization (LTO)*, and it is possible because modern compilers also store some form of *intermediate representation* in object files, which allows them to perform certain lightweight optimizations on the program as a whole. This also allows using different compiled languages in the same program, which can even be optimized across language barriers if their compilers use the same intermediate representation.

LTO is a relatively recent feature (it appeared in GCC only around 2014), and it is still far from perfect. In C and C++, the way to make sure no performance is lost due to separate compilation is to create a *header-only library*. As the name suggests, they are just header files that contain full definitions of all functions, and so by simply including them, the compiler gets access to all optimizations possible. Although you do have to recompile the library code from scratch each time, this approach retains full control and makes sure that no performance is lost.

Inspecting the Output

Examining output from each of these stages can yield useful insights into what's happening in your program.

You can get assembly from the source by passing the `-S` flag to the compiler, which will then generate a human-readable `*.s` file. If you pass `-fverbose-asm`, this file will also contain compiler comments about source code line numbers and some info about variables being used. If it is just a little snippet and you are feeling lazy, you can use [Compiler Explorer](#), which is a very handy

online tool that converts source code to assembly, highlights logical asm blocks by color, includes a small x86 instruction set reference, and also has a large selection of other compilers, targets, and languages.

Apart from the assembly, the other most helpful level of abstraction is the intermediate representation on which compilers perform optimizations. The IR defines the flow of computation itself and is much less dependent on architecture features like the number of registers or a particular instruction set. It is often useful to inspect these to get insight into how the compiler *sees* your program, but this is a bit out of the scope of this book.

We will mainly use [GCC](#) in this chapter, but also try to duplicate examples for [Clang](#) when necessary. The two compilers are largely compatible with each other, for the most part only differing in some optimization flags and minor syntax details.

Flags and Targets

The first step of getting high performance from the compiler is to ask for it, which is done with over a hundred different compiler options, attributes, and pragmas.

Optimization Levels

There are *4 and a half* main levels of optimization for speed in GCC:

- `-O0` is the default one that does no optimizations (although, in a sense, it does optimize: for compilation time).
- `-O1` (also aliased as `-O`) does a few “low-hanging fruit” optimizations, almost not affecting the compilation time.
- `-O2` enables all optimizations that are known to have little to no negative side effects and take a reasonable time to complete (this is what most projects use for production builds).
- `-O3` does very aggressive optimization, enabling almost all *correct* optimizations implemented in GCC.
- `-Ofast` does everything in `-O3`, plus a few more optimizations flags that may break strict standard compliance, but not in a way that would be critical for most applications (e.g., floating-point operations may be rearranged so that the result is off by a few bits in the mantissa).

There are also many other optimization flags that are not included even in `-Ofast`, because they are very situational, and enabling them by default is more likely to hurt performance rather than improve it — we will talk about some of them in [the next section](#).

Specifying Targets

The next thing you may want to do is to tell the compiler more about the computer(s) this code is supposed to be run on: the smaller the set of platforms is, the better. By default, it will generate binaries that can run on any relatively new (>2000) x86 CPU. The simplest way to narrow it down is to pass `-march` flag to specify the exact microarchitecture: `-march=haswell`. If you are compiling on the same computer that will run the binary, you can use `-march=native` for auto-detection.

The instruction sets are generally backward-compatible, so it is often enough to just use the name of the oldest microarchitecture you need to support. A more robust approach is to list specific features that the CPU is guaranteed to have: `-mavx2`, `-mpopcnt`. When you just want to *tune* the

program for a particular machine without using any instructions that may crash it on incompatible CPUs, you can use the `-mtune` flag (by default `-march=x` also implies `-mtune=x`).

These options can also be specified for a compilation unit with pragmas instead of compilation flags:

```
#pragma GCC optimize("O3")
#pragma GCC target("avx2")
```

This is useful when you need to optimize a single high-performance procedure without increasing the build time for the entire project.

Multiversioned Functions

Sometimes you may also want to provide several architecture-specific implementations in a single library. You can use attribute-based syntax to select between multiversioned functions automatically during compile time:

```
--attribute__(( target("default") )) // fallback implementation
int popcnt(int x) {
    int s = 0;
    for (int i = 0; i < 32; i++)
        s += (x>>i&1);
    return s;
}

--attribute__(( target("popcnt") )) // used if popcnt flag is enabled
int popcnt(int x) {
    return __builtin_popcount(x);
}
```

In Clang, you can't use pragmas to set target and optimization flags from the source code, but you can use attributes the same way as in GCC.

Situational Optimizations

Most compiler optimizations enabled by `-O2` and `-O3` are guaranteed to either improve or at least not seriously hurt performance. Those that aren't included in `-O3` are either not strictly standard-compliant, or highly circumstantial and require some additional input from the programmer to help decide whether using them is beneficial.

Let's discuss the most frequently used ones that we've also previously covered in this book.

Loop Unrolling

[Loop unrolling](#) is disabled by default, unless the loop takes a small constant number of iterations known at compile time — in which case it will be replaced with a completely jump-free, repeated sequence of instructions. It can be enabled globally with the `-funroll-loops` flag, which will unroll all loops whose number of iterations can be determined at compile time or upon entry to the loop.

You can also use a pragma to target a specific loop:

```
#pragma GCC unroll 4
for (int i = 0; i < n; i++) {
    // ...
}
```

Loop unrolling makes binary larger, and may or may not make it run faster. Don't use it fanatically.

Function Inlining

Inlining is best left for the compiler to decide, but you can influence it with `inline` keyword:

```
inline int square(int x) {
    return x * x;
}
```

The hint may be ignored though if the compiler thinks that the potential performance gains are not worth it. You can force inlining by adding the `always_inline` attribute:

```
#define FORCE_INLINE inline __attribute__((always_inline))
```

There is also the `-finline-limit=n` option which lets you set a specific threshold on the size of inlined functions (in terms of the number of instructions). Its Clang equivalent is `-inline-threshold`.

Likeliness of Branches

Likeliness of branches can be hinted by `[[likely]]` and `[[unlikely]]` attributes in `if`-s and `switch`-es:

```
int factorial(int n) {
    if (n > 1) [[likely]]
        return n * factorial(n - 1);
    else [[unlikely]]
        return 1;
}
```

This is a new feature that only appeared in C++20. Before that, there were compiler-specific intrinsics similarly used to wrap condition expressions. The same example in older GCC:

```
int factorial(int n) {
    if (__builtin_expect(n > 1, 1))
        return n * factorial(n - 1);
    else
        return 1;
}
```

There are many other cases like this when you need to point the compiler in the right direction, but we will get to them later when they become more relevant.

Profile-Guided Optimization

Adding all this metadata to the source code is tedious. People already hate writing C++ even without having to do it.

It is also not always obvious whether certain optimizations are beneficial or not. To make a decision about branch reordering, function inlining, or loop unrolling, we need answers to questions like these:

- How often is this branch taken?
- How often is this function called?
- What is the average number of iterations in this loop?

Luckily for us, there is a way to provide this real-world information automatically.

Profile-guided optimization (PGO, also called “pogo” because it’s easier and more fun to pronounce) is a technique that uses [profiling data](#) to improve performance beyond what can be achieved with just static analysis. In a nutshell, it involves adding timers and counters to the points of interest in the program, compiling and running it on real data, and then compiling it again, but this time supplying additional information from the test run.

The whole process is automated by modern compilers. For example, the `-fprofile-generate` flag will let GCC instrument the program with profiling code:

```
g++ -fprofile-generate [other flags] source.cc -o binary
```

After we run the program — preferably on input that is as representative of the real use case as possible — it will create a bunch of `*.gcda` files that contain log data for the test run, after which we can rebuild the program, but now adding the `-fprofile-use` flag:

```
g++ -fprofile-use [other flags] source.cc -o binary
```

It usually improves performance by 10-20% for large codebases, and for this reason it is commonly included in the build process of performance-critical projects. This is more reason to invest in solid benchmarking code.

Compilation

The main benefit of [learning assembly language](#) is not the ability to write programs in it, but the understanding of what is happening during the execution of compiled code and its performance implications.

There are rare cases where we *really* need to switch to handwritten assembly for maximal performance, but most of the time compilers are capable of producing near-optimal code all by themselves. When they do not, it is usually because the programmer knows more about the problem than what can be inferred from the source code but failed to communicate this extra information to the compiler.

In this chapter, we will discuss the intricacies of getting the compiler to do exactly what we want and gathering useful information that can guide further optimizations.

Contract Programming

In “safe” languages like Java and Rust, you normally have well-defined behavior for every possible operation and every possible input. There are some things that are *under-defined*, like the order of keys in a hash table or the growth factor of an `std::vector`, but these are usually some minor details that are left up to implementation for potential performance gains in the future.

In contrast, C and C++ take the concept of undefined behavior to another level. Certain operations don't cause an error during compilation or runtime but are just not *allowed* — in the sense of there being a *contract* between the programmer and the compiler, that in case of undefined behavior, the compiler is legally allowed to do literally anything, including blowing up your monitor or formatting your hard drive. But compiler engineers are not interested in doing that. Instead, undefined behavior is used to guarantee a lack of corner cases and help optimization.

Why Undefined Behavior Exists

There are two major groups of actions that cause undefined behavior:

- Operations that are almost certainly unintentional bugs, like dividing by zero, dereferencing a null pointer, or reading from uninitialized memory. You want to catch these as soon as possible during testing, so crashing or having some non-deterministic behavior is better than having them always do a fixed fallback action such as returning zero.

You can compile and run a program with *sanitizers* to catch undefined behavior early. In GCC and Clang, you can use the `-fsanitize=undefined` flag, and some operations that are notorious for causing UB will be instrumented to detect it at runtime.

- Operations that have slightly different observable behavior on different platforms. For example, the result of left-shifting an integer by more than 31 bits is undefined, because the instruction that does it is implemented differently on Arm and x86 CPUs. If you standardize one specific behavior, then all programs compiled for the other platform will have to spend a few more cycles checking for that edge case, so it is best to leave it undefined.

Sometimes, when there is a legitimate use case for some platform-specific behavior, instead of declaring it undefined, it can be left *implementation-defined*. For example, the result of right-shifting a [negative integer](#) depends on the platform: it either shifts in zeros or ones (e.g., right-shifting `11010110 = -42` by one may mean either `01101011 = 107` or `11101011 = -21`, both use cases being realistic).

Designating something as undefined instead of implementation-defined behavior also helps compilers in optimization. Consider the case of signed integer overflow. On almost all architectures, [signed integers](#) overflow the same way as unsigned ones, with `INT_MAX + 1 == INT_MIN`, and yet, this is undefined behavior according to the C++ standard. This is very much intentional: if you disallow signed integer overflow, then `(x + 1) > x` is guaranteed to be always true for `int`, but not for `unsigned int`, because `(x + 1)` may overflow. For signed types, this lets compilers optimize such checks away.

As a more naturally occurring example, consider the case of a loop with an integer control variable. Modern C++ and languages like Rust encourage programmers to use an unsigned integer (`size_t` / `usize`), while C programmers stubbornly keep using `int`. To understand why, consider the following `for` loop:

```
for (unsigned int i = 0; i < n; i++) {  
    // ...  
}
```

How many times does this loop execute? There are technically two valid answers: n and infinity, the second being the case if n exceeds 2^{32} so that i keeps resetting to zero every 2^{32} iterations. While the former is probably the one assumed by the programmer, to comply with the language spec,

the compiler still has to insert additional runtime checks and consider the two cases, which should be optimized differently. Meanwhile, the `int` version would make exactly n iterations because the very possibility of a signed overflow is defined out of existence.

Removing Corner Cases

The “safe” programming style usually involves making a lot of runtime checks, but they do not have to come at the cost of performance.

For example, Rust famously uses bounds checking when indexing arrays and other random access structures. In C++ STL, `vector` and `array` have an “unsafe” `[]` operator and a “safe” `.at()` method that goes something like this:

```
T at(size_t k) {
    if (k >= size())
        throw std::out_of_range("Array index exceeds its size");
    return _memory[k];
}
```

Interestingly, these checks are rarely actually executed during runtime because the compiler can often prove — during compile time — that each access will be within bounds. For example, when iterating in a `for` loop from 1 to the array size and indexing i -th element on each step, nothing illegal can possibly happen, so the bounds checks can be safely optimized away.

Assumptions

When the compiler can’t prove the non-existence of corner cases, but *you* can, this additional information can be provided using the mechanism of undefined behavior.

Clang has a helpful `__builtin_assume` function where you can put a statement that is guaranteed to be true, and the compiler will use this assumption in optimization. In GCC, you can do the same with `__builtin_unreachable`:

```
void assume(bool pred) {
    if (!pred)
        __builtin_unreachable();
}
```

For instance, you can put `assume(k < vector.size())` before `at` in the example above, and then the bounds check will be optimized away.

It is also quite useful to combine `assume` with `assert` and `static_assert` to find bugs: you can use the same function to check preconditions in the debug build and then use them to improve performance in the production build.

Arithmetic

Corner cases are something you should keep in mind, especially when optimizing arithmetic.

For [floating-point arithmetic](#), this is less of a concern because you can just disable strict standard compliance with the `-ffast-math` flag (which is also included in `-Ofast`). You almost have to do it anyway because otherwise, the compiler can’t do anything but execute arithmetic operations in the same order as in the source code without any optimizations.

For integer arithmetic, this is different because the results always have to be exact. Consider the case of division by 2:

```
unsigned div_unsigned(unsigned x) {
    return x / 2;
}
```

A widely known optimization is to replace it with a single right shift ($x \gg 1$):

```
shr eax
```

This is certainly correct for all *positive* numbers, but what about the general case?

```
int div_signed(int x) {
    return x / 2;
}
```

If x is negative, then simply shifting doesn't work — regardless of whether shifting is done in zeros or sign bits:

- If we shift in zeros, we get a non-negative result (the sign bit is zero).
- If we shift in sign bits, then rounding will happen towards negative infinity instead of zero ($-5 / 2$ will be equal to -3 instead of -2).¹

So, for the general case, we have to insert some crutches to make it work:

```
mov ebx, eax
shr ebx, 31      ; extract the sign bit
add eax, ebx     ; add 1 to the value if it is negative to ensure rounding towards zero
sar eax          ; this one shifts in sign bits
```

When only the positive case is what was intended, we can also use the `assume` mechanism to eliminate the possibility of negative x and avoid handling this corner case:

```
int div_assume(int x) {
    assume(x >= 0);
    return x / 2;
}
```

Although in this particular case, perhaps the best syntax to express that we only expect non-negative numbers is to use an unsigned integer type.

Because of nuances like this, it is often beneficial to expand the algebra in intermediate functions and manually simplify arithmetic yourself rather than relying on the compiler to do it.

Memory Aliasing

Compilers are quite bad at optimizing operations that involve memory reads and writes. This is because they often don't have enough context for the optimization to be correct.

Consider the following example:

¹Fun fact: in Python, integer-dividing a negative number for some reason floors the result, so that $-5 // 2 = -3$ and equivalent to $-5 \gg 1 = -3$. I doubt that Guido van Rossum had this optimization in mind when initially designing the language, but, theoretically, a [JIT-compiled](#) Python program with many divisions by two may be faster than an analogous C++ program.

```

void add(int *a, int *b, int n) {
    for (int i = 0; i < n; i++)
        a[i] += b[i];
}

```

Since each iteration of this loop is independent, it can be executed in parallel and [vectorized](#). But is it, technically?

There may be a problem if the arrays **a** and **b** intersect. Consider the case when **b == a + 1**, that is, if **b** is just a memory view of **a** starting from its second element. In this case, the next iteration depends on the previous one, and the only correct solution is to execute the loop sequentially. The compiler has to check for such possibilities even if the programmer knows they can't happen.

This is why we have **const** and **restrict** keywords. The first one enforces that we won't modify memory with the pointer variable, and the second is a way to tell the compiler that the memory is guaranteed to not be aliased.

```

void add(int * __restrict__ a, const int * __restrict__ b, int n) {
    for (int i = 0; i < n; i++)
        a[i] += b[i];
}

```

These keywords are also a good idea to use by themselves for the purpose of self-documenting.

C++ Contracts

Contract programming is an underused but very powerful technique.

There is a late-stage proposal to add design-by-contract into the C++ standard in the form of [contract attributes](#), which are functionally equivalent to our hand-made, compiler-specific **assume**:

```

T at(size_t k) [[ expects: k < n ]] {
    return _memory[k];
}

```

There are 3 types of attributes — **expects**, **ensures**, and **assert** — respectively used for specifying pre- and post-conditions in functions and general assertions that can be put anywhere in the program.

Unfortunately, this exciting new feature is [not yet finally standardized](#), let alone implemented in a major C++ compiler. But maybe, in a few years, we will be able to write code like this:

```

bool is_power_of_two(int m) {
    return m > 0 && (m & (m - 1) == 0);
}

int mod_power_of_two(int x, int m)
[[ expects: x >= 0 ]]
[[ expects: is_power_of_two(m) ]]
[[ ensures r: r >= 0 && r < m ]]
{
    int r = x & (m - 1);
    [[ assert: r = x % m ]];
}

```

```
    return r;
}
```

Some forms of contract programming are also available in other performance-oriented languages such as [Rust](#) and [D](#).

A general and language-agnostic advice is to always [inspect the assembly](#) that the compiler produced, and if it is not what you were hoping for, try to think about corner cases that may be limiting the compiler from optimizing it.

Non-Zero-Cost Abstractions

In general, abstractions are great. Applied well, they reduce the amount of code and the mental burden of a programmer.

But abstractions often come at a cost in terms of performance. When you use a shared library, you have spent some extra cycles moving data around to properly call its functions. When you call a virtual method, you can't reliably predict what code you are going to execute next and effectively suffer a branch mispredict.

Languages like C++ and Rust heavily promote the idea of *zero-cost* abstractions that have no extra runtime overhead, and that can be, at least in principle, completely removed by compilers. But in practice, there is no such thing as a zero-cost abstraction — compiler technology just isn't there yet.

Virtual functions. Any type of runtime polymorphism.

Bounds checking. Compilers are good at eliminating them though.

Generally any complicated code. Here is an example that personally bugs me: `std::min` from the C++ standard library. It repeatedly performs worse than just taking minimum by hand, the cause being that it isn't implemented just as `return (a < b ? a : b)`, but instead using variadic initializer lists and iterators for genericity:

```
template<typename _Tp> GLIBCXX14_CONSTEXPR inline _Tp min(initializer_list<_Tp> __l) {
    return *std::min_element(__l.begin(), __l.end());
}
```

Usually it isn't that hard to rewrite a small program so that it is more straightforward and closer to the hardware. If you start removing layers of abstractions the compiler will eventually give in.

Object-oriented and especially functional languages have some very hard-to-pierce abstractions like these. For this reason, people often prefer to write performance critical software (interpreters, runtimes, databases) in a style closer to C rather than higher-level languages.

Thick-bearded C/assembly programmers.

Memory

Pointer chasing.

```
typedef vector< vector<int> > matrix;
matrix a(n, vector<int>(n, 0));
```

```
int val = a[i][j];
```

This is up to twice as slow: you first need to fetch

```
int a = new int[n * n];
memset(a, 0, 4 * n * n);
```

```
int val = a[i * n + j];
```

You can write a wrapper if you really want an abstraction:

```
template<typename T>
struct Matrix {
    int x, y, n, N;
    T* data;
    T* operator[](int i) { return data + (x + i) * N + y; }
};
```

For example, the [cache-oblivious transposition](#) would go like this:

```
Matrix<T> subset(int _x, int _y, int _n) { return {_n, _x, _y, N, data}; }

Matrix<T> transpose() {
    if (n <= 32) {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < i; j++)
                swap((*this)[j][i], (*this)[i][j]);
    } else {
        auto A = subset(x, y, n / 2).transpose();
        auto B = subset(x + n / 2, y, n / 2).transpose();
        auto C = subset(x, y + n / 2, n / 2).transpose();
        auto D = subset(x + n / 2, y + n / 2, n / 2).transpose();
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                swap(B[i][j], C[i][j]);
    }
    return *this;
}
```

I personally prefer to write low-level code, because it is easier to optimize.

It is cleaner? Don't think so.

Precomputation

When compilers can infer that a certain variable does not depend on any user-provided data, they can compute its value during compile time and turn it into a constant by embedding it into the generated machine code.

This optimization helps performance a lot, but it is not a part of the C++ standard, so compilers

don't *have to* do that. When a compile-time computation is either hard to implement or time-intensive, a compiler may pass on that opportunity.

Constant Expressions

For a more reliable solution, in modern C++ you can mark a function as `constexpr`; if it is called by passing constants its value is guaranteed to be computed during compile time:

```
constexpr int fibonacci(int n) {
    if (n <= 2)
        return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

static_assert(fibonacci(10) == 55);
```

These functions have some restrictions like that they only call other `constexpr` functions and can't do memory allocation, but otherwise, they are executed "as is."

Note that while `constexpr` functions don't cost anything during run time, they still increase compilation time, so at least remotely care about their efficiency and don't put something NP-complete in them:

```
constexpr int fibonacci(int n) {
    int a = 1, b = 1;
    while (n--) {
        int c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

There used to be many more limitations in earlier C++ standards, like you could not use any sort of state inside them and had to rely on recursion, so the whole process felt more like Haskell programming rather than C++. Since C++17, you can even compute static arrays using the imperative style, which is useful for precomputing lookup tables:

```
struct Precalc {
    int isqrt[1000];

    constexpr Precalc() : isqrt{} {
        for (int i = 0; i < 1000; i++)
            isqrt[i] = int(sqrt(i));
    }
};

constexpr Precalc P;

static_assert(P.isqrt[42] == 6);
```

Note that when you call `constexpr` functions while passing non-constants, the compiler may or may not compute them during compile time:

```
for (int i = 0; i < 100; i++)
    cout << fibonacci(i) << endl;
```

In this example, even though technically we perform a constant number of iterations and call `fibonacci` with parameters known at compile time, they are technically not compile-time constants. It's up to the compiler whether to optimize this loop or not — and for heavy computations, it often chooses not to.

Arithmetic Optimizations

...

What Compilers Can and Can't Do

Let's sum up this chapter with a general advice.

Writing optimizing compilers is very hard. The last Turing award went to Alfred Aho and Jeffrey Ullman for a [1000-page book](#) they wrote about compilers, which is considered an *introductory* textbook.

There are about 120 of optimizations in GCC included in `-O3`, and about 60 of them even have their own [Wikipedia pages](#). At a very high level, these optimizations improve performance by:

- doing a good job at fundamental algorithms like register allocation, scheduling, dead code elimination, etc.;
- being good at eliminating unnecessary abstractions (Inlining functions, eliminating temporary objects, etc.);
- knowing the CPU architecture very well (replacing divisions by 2 with binary shifts, replacing multiply-and-add with `lea` when possible, and in general picking optimal instructions);
- knowing lots and lots of “tricks” (specific [peephole optimizations](#) and loop transformations), and applying them whenever profitable.

Unless you are *really* into compiler engineering, I wouldn't recommend going through the list and learning all of them. Instead, gradually build up a broader understanding of what compilers are capable of. Trial and error generally works well here: just assume that everything generic and simple enough is already implemented, but always check assembly output in case you are wrong.

In general, when an optimization doesn't happen, it is usually because one of these three reasons:

- The compiler doesn't have enough information to know it will be beneficial.
- The optimization is actually not always correct: there is an input on which the result doesn't comply with the spec, even if it is correct on every input that the programmer expects.
- It isn't implemented in the compiler yet, either because it is too hard to implement in general, too costly to compute or too rare to be worth the trouble (e.g., writing a tiny library for some specific algorithm is usually better than hardcoding it into compiler).

In addition, optimization sometimes fails just due to the source code being overly complicated.

Checklist

Usually the right approach to performance is to think how the main hot spots of the implementation should look like in assembly, write high-level code that resembles it as much as possible, and then repeatedly ask yourself the following questions until you are happy with its performance:

0. Does the compiler know it is allowed to do these optimizations? (`-O3, -march=native, -ffast-math`)
1. Are there any edge cases where optimized version would not work correctly? (use `__restrict__` and `const` keywords, try the `assume` trick)
2. Is there a real-world dataset for which the optimization may not be beneficial? (hints, pragmas, PGO)
3. Are there at least 1000 other places where this optimization makes sense? (remove abstractions and implement it manually, add a feature request for GCC and Clang)

In the majority of the cases, at least one of these answers will be “no,” and then you will know what to do.

Chapter 5: Profiling

Algorithms for Modern Hardware

Contents

Instrumentation	1
Event Sampling	2
Statistical Profiling	3
Hardware Events	3
Profiling with perf	4
Program Simulation	6
Profiling with Cachegrind	6
Machine Code Analyzers	8
Using <code>llvm-mca</code>	9
Profiling	10
Benchmarking	11
Benchmarking Inside C++	11
Splitting Up Implementations	12
Makefiles	14
Jupyter Notebooks	14
Getting Accurate Results	15
Measuring the Right Thing	15
Reducing Noise	17
Further Reading	18

Instrumentation

Instrumentation is an overcomplicated term that means inserting timers and other tracking code into programs. The simplest example is using the `time` utility in Unix-like systems to measure the duration of execution for the whole program.

More generally, we want to know *which parts* of the program need optimization. There are tools shipped with compilers and IDEs that can time designated functions automatically, but it is more robust to do it by hand using any methods of interacting with time that the language provides:

```
clock_t start = clock();
do_something();
float seconds = float(clock() - start) / CLOCKS_PER_SEC;
printf("do_something() took %.4f", seconds);
```

One nuance here is that you can't measure the execution time of particularly quick functions this way because the `clock` function returns the current timestamp in microseconds (10^{-6}) and also by

itself takes up to a few hundred nanoseconds to complete. All other time-related utilities similarly have at least microsecond granularity, which is an eternity in the world of low-level optimization.

To achieve higher precision, you can invoke the function repeatedly in a loop, time the whole thing once, and then divide the total time by the number of iterations:

```
#include <stdio.h>
#include <time.h>

const int N = 1e6;

int main() {
    clock_t start = clock();

    for (int i = 0; i < N; i++)
        clock(); // benchmarking the clock function itself

    float duration = float(clock() - start) / CLOCKS_PER_SEC;
    printf("%.2fns per iteration\n", 1e9 * duration / N);

    return 0;
}
```

You also need to ensure that nothing gets cached, optimized away by the compiler, or affected by similar side effects. This is a separate and highly complicated topic that we will discuss in more detail at [the end of the chapter](#).

Event Sampling

Instrumentation can also be used to collect other types of information that can give useful insights about the performance of a particular algorithm. For example:

- for a hash function, we are interested in the average length of its input;
- for a binary tree, we care about its size and height;
- for a sorting algorithm, we want to know how many comparisons it does.

In a similar way, we can insert counters in the code that compute these algorithm-specific statistics.

Adding counters has the disadvantage of introducing overhead, although you can mitigate it almost completely by only doing it randomly for a small fraction of invocations:

```
void query() {
    if (rand() % 100 == 0) {
        // update statistics
    }
    // main logic
}
```

If the sample rate is small enough, the only remaining overhead per invocation will be random number generation and a condition check. Interestingly, we can optimize it a bit more with some stats magic.

Mathematically, what we are doing here is repeatedly sampling from [Bernoulli distribution](#) (with p equal to sample rate) until we get a success. There is another distribution that tells us how many iterations of Bernoulli sampling we need until the first positive, called [geometric distribution](#). What we can do is to sample from it instead and use that value as a decrementing counter:

```
void query() {
    static next_sample = geometric_distribution(sample_rate);
    if (next_sample--) {
        next_sample = geometric_distribution(sample_rate);
        // ...
    }
    // ...
}
```

This way we can remove the need to sample a new random number on each invocation, only resetting the counter when we choose to calculate statistics.

Techniques like that are frequently used by library algorithm developers inside large projects to collect profiling data without affecting the performance of the end program too much.

Statistical Profiling

[Instrumentation](#) is a rather tedious way of doing profiling, especially if you are interested in multiple small sections of the program. And even if it can be partially automated by the tooling, it still won't help you gather some fine-grained statistics because of its inherent overhead.

Another, less invasive approach to profiling is to interrupt the execution of a program at random intervals and look where the instruction pointer is. The number of times the pointer stopped in each function's block would be roughly proportional to the total time spent executing these functions. You can also get some other useful information this way, like finding out which functions are called by which functions by inspecting [the call stack](#).

This could, in principle, be done by just running a program with `gdb` and `ctrl+c`'ing it at random intervals but modern CPUs and operating systems provide special utilities for this type of profiling.

Hardware Events

Hardware *performance counters* are special registers built into microprocessors that can store the counts of certain hardware-related activities. They are cheap to add on a microchip, as they are basically just binary counters with an activation wire connected to them.

Each performance counter is connected to a large subset of circuitry and can be configured to be incremented on a particular hardware event, such as a branch mispredict or a cache miss. You can reset a counter at the start of a program, run it, and output its stored value at the end, and it will be equal to the exact number of times a certain event has been triggered throughout the execution.

You can also keep track of multiple events by multiplexing between them, that is, stopping the program in even intervals and reconfiguring the counters. The result in this case will not be exact, but a statistical approximation. One nuance here is that its accuracy can't be improved by simply increasing the sampling frequency because it would affect the performance too much and thus skew the distribution, so to collect multiple statistics, you would need to run the program for longer periods of time.

Overall, event-driven statistical profiling is usually the most effective and easy way to diagnose performance issues.

Profiling with perf

Performance analysis tools that rely on the event sampling techniques described above are called *statistical profilers*. There are many of them, but the one we will mainly use in this book is [perf](#), which is a statistical profiler shipped with the Linux kernel. On non-Linux systems, you can use [VTune](#) from Intel, which provides roughly the same functionality for our purposes. It is available for free, although it is proprietary, and you need to refresh your community license every 90 days, while perf is free as in freedom.

Perf is a command-line application that generates reports based on the live execution of programs. It does not need the source and can profile a very wide range of applications, even those that involve multiple processes and interaction with the operating system.

For explanation purposes, I have written a small program that creates an array of a million random integers, sorts it, and then does a million binary searches on it:

```
void setup() {
    for (int i = 0; i < n; i++)
        a[i] = rand();
    std::sort(a, a + n);
}

int query() {
    int checksum = 0;
    for (int i = 0; i < n; i++) {
        int idx = std::lower_bound(a, a + n, rand()) - a;
        checksum += idx;
    }
    return checksum;
}
```

After compiling it (`g++ -O3 -march=native example.cc -o run`), we can run it with `perf stat ./run`, which outputs the counts of basic performance events during its execution:

```
Performance counter stats for './run':
```

646.07 msec	task-clock:u	# 0.997 CPUs utilized
0	context-switches:u	# 0.000 K/sec
0	cpu-migrations:u	# 0.000 K/sec
1,096	page-faults:u	# 0.002 M/sec
852,125,255	cycles:u	# 1.319 GHz (83.35%)
28,475,954	stalled-cycles-frontend:u	# 3.34% frontend cycles idle (83.30%)
10,460,937	stalled-cycles-backend:u	# 1.23% backend cycles idle (83.28%)
479,175,388	instructions:u	# 0.56 insn per cycle
		# 0.06 stalled cycles per insn (83.28%)
122,705,572	branches:u	# 189.925 M/sec (83.32%)
19,229,451	branch-misses:u	# 15.67% of all branches (83.47%)

```

0.647801770 seconds time elapsed
0.647278000 seconds user
0.000000000 seconds sys

```

You can see that the execution took 0.53 seconds or 852M cycles at an effective 1.32 GHz clock rate, over which 479M instructions were executed. There were also 122.7M branches, and 15.7% of them were mispredicted.

You can get a list of all supported events with `perf list`, and then specify a list of specific events you want with the `-e` option. For example, for diagnosing binary search, we mostly care about cache misses:

```
> perf stat -e cache-references,cache-misses ./run
```

```

91,002,054      cache-references:u
44,991,746      cache-misses:u      # 49.440 % of all cache refs

```

By itself, `perf stat` simply sets up performance counters for the whole program. It can tell you the total number of branch mispredictions, but it won't tell you *where* they are happening, let alone *why* they are happening.

To try the stop-the-world approach we discussed previously, we need to use `perf record <cmd>`, which records profiling data and dumps it as a `perf.data` file, and then call `perf report` to inspect it. I highly advise you to go and try it yourselves because the last command is interactive and colorful, but for those that can't do it right now, I'll try to describe it the best I can.

When you call `perf report`, it first displays a `top`-like interactive report that tells you which functions are taking how much time:

Overhead	Command	Shared Object	Symbol
63.08%	run	run	[.] query
24.98%	run	run	[.] std::__introsort_loop<...>
5.02%	run	libc-2.33.so	[.] __random
3.43%	run	run	[.] setup
1.95%	run	libc-2.33.so	[.] __random_r
0.80%	run	libc-2.33.so	[.] rand

Note that, for each function, just its *overhead* is listed and not the total running time (e.g., `setup` includes `std::__introsort_loop` but only its own overhead is accounted as 3.43%). There are tools for constructing [flame graphs](#) out of perf reports to make them more clear. You also need to account for possible inlining, which is apparently what happened with `std::lower_bound` here. Perf also tracks shared libraries (like `libc`) and, in general, any other spawned processes: if you want, you can launch a web browser with perf and see what's happening inside.

Next, you can "zoom in" on any of these functions, and, among others things, it will offer to show you its disassembly with an associated heatmap. For example, here is the assembly for `query`:

```

20: → call  rand@plt
        mov   %r12,%rsi
        mov   %eax,%edi
        mov   $0xf4240,%eax
        nop

```

```

            30: test    %rax,%rax
4.57      ↓ jle     52
            35: mov     %rax,%rdx
0.52      sar     %rdx
0.33      lea     (%rsi,%rdx,4),%rcx
4.30      cmp     (%rcx),%edi
65.39      ↓ jle     b0
0.07      sub     %rdx,%rax
9.32      lea     0x4(%rcx),%rsi
0.06      dec     %rax
1.37      test    %rax,%rax
1.11      ↑ jg      35
52:       sub     %r12,%rsi
2.22      sar     $0x2,%rsi
0.33      add     %esi,%ebp
0.20      dec     %ebx
↑ jne     20

```

On the left column is the fraction of times that the instruction pointer stopped on a specific line. You can see that we spend ~65% of the time on the jump instruction because it has a comparison operator before it, indicating that the control flow waits there for this comparison to be decided.

Because of intricacies such as [pipelining](#) and out-of-order execution, “now” is not a well-defined concept in modern CPUs, so the data is slightly inaccurate as the instruction pointer drifts a little bit forward. The instruction-level data is still useful, but at the individual cycle level, we need to switch to [something more precise](#).

Program Simulation

The last approach to profiling (or rather a group of them) is not to gather the data by actually running the program but to analyze what should happen by *simulating* it with specialized tools.

There are many subcategories of such profilers, differing in which aspect of computation is simulated. In this article, we are going to focus on [caching](#) and [branch prediction](#), and use [Cachegrind](#) for that, which is a profiling-oriented part of [Valgrind](#), a well-established tool for memory leak detection and memory debugging in general.

Profiling with Cachegrind

Cachegrind essentially inspects the binary for “interesting” instructions — that perform memory reads / writes and conditional / indirect jumps — and replaces them with code that simulates corresponding hardware operations using software data structures. It therefore doesn’t need access to the source code and can work with already compiled programs, and can be run on any program like this:

```
valgrind --tool=cachegrind --branch-sim=yes ./run
#           also simulate branch prediction ^ ^ any command, not necessarily one process
```

It instruments all involved binaries, runs them, and outputs a summary similar to [perf stat](#):

```
I refs: 483,664,426
```

```

I1 misses:          1,858
LLi misses:        1,788
I1 miss rate:      0.00%
LLi miss rate:     0.00%

D   refs:    115,204,359 (88,016,970 rd + 27,187,389 wr)
D1  misses:   9,722,664 ( 9,656,463 rd +     66,201 wr)
LLd misses:    72,587 (     8,496 rd +     64,091 wr)
D1  miss rate: 8.4% (     11.0% +     0.2% )
LLd miss rate: 0.1% (     0.0% +     0.2% )

LL refs:    9,724,522 ( 9,658,321 rd +     66,201 wr)
LL misses:   74,375 (    10,284 rd +     64,091 wr)
LL miss rate: 0.0% (     0.0% +     0.2% )

Branches:    90,575,071 (88,569,738 cond +  2,005,333 ind)
Mispredicts: 19,922,564 (19,921,919 cond +      645 ind)
Mispred rate: 22.0% (     22.5% +     0.0% )

```

We've fed Cachegrind exactly the same example code as in [the previous section](#): we create an array of a million random integers, sort it, and then perform a million binary searches on it. Cachegrind shows roughly the same numbers as perf does, except that that perf's measured numbers of memory reads and branches are slightly inflated due to [speculative execution](#): they really happen in hardware and thus increment hardware counters, but are discarded and don't affect actual performance, and thus ignored in the simulation.

Cachegrind only models the first (D1 for data, I1 for instructions) and the last (LL, unified) levels of cache, the characteristics of which are inferred from the system. It doesn't limit you in any way as you can also set them from the command line, e.g., to model the L2 cache: `--LL=<size>,<associativity>,<line size>`.

It seems like it only slowed down our program so far and hasn't provided us any information that `perf stat` couldn't. To get more out of it than just the summary info, we can inspect a special file with profiling info, which it dumps by default in the same directory named as `cachegrind.out.<pid>`. It is human-readable, but is expected to be read via the `cg_annotate` command:

```
cg_annotate cachegrind.out.4159404 --show=Dr,D1mr,DLmr,Bc,Bcm
#                                         ^ we are only interested in data reads and branches
```

First it shows the parameters that were used during the run, including the characteristics of the cache system:

```
I1 cache:          32768 B, 64 B, 8-way associative
D1 cache:          32768 B, 64 B, 8-way associative
LL cache:         8388608 B, 64 B, direct-mapped
```

It didn't get the L3 cache quite right: it is not unified (8M in total, but a single core only sees 4M) and also 16-way associative, but we will ignore that for now.

Next, it outputs a per-function summary similar to `perf report`:

Dr	D1mr	DLmr	Bc	Bcm	file:function
19,951,476	8,985,458	3	41,902,938	11,005,530	???:query()
24,832,125	585,982	65	24,712,356	7,689,480	???:void std::__introsort_loop<...>
16,000,000	60	3	9,935,484	129,044	???:random_r
18,000,000	2	1	6,000,000	1	???:random
4,690,248	61,999	17	5,690,241	1,081,230	???:setup()
2,000,000	0	0	0	0	???:rand

You can see there are a lot of branch mispredicts in the sorting stage, and also a lot of both L1 cache misses and branch mispredicts during binary searching. We couldn't get this information with perf — it would only tell use these counts for the whole program.

Another great feature that Cachegrind has is the line-by-line annotation of source code. For that, you need to compile the program with debug information (-g) and either explicitly tell `cg_annotate` which source files to annotate or just pass the `--auto=yes` option so that it annotates everything it can reach (including the standard library source code).

The whole source-to-analysis process would therefore go like this:

```
g++ -O3 -g sort-and-search.cc -o run
valgrind --tool=cachegrind --branch-sim=yes --cachegrind-out-file=cachegrind.out ./run
cg_annotate cachegrind.out --auto=yes --show=Dr,D1mr,DLmr,Bc,Bcm
```

Since the glibc implementations are not the most readable, for exposition purposes, we replace `lower_bound` with our own binary search, which will be annotated like this:

Dr	D1mr	DLmr	Bc	Bcm	
.	<code>int binary_search(int x) {</code>
0	0	0	0	0	<code>int l = 0, r = n - 1;</code>
0	0	0	20,951,468	1,031,609	<code>while (l < r) {</code>
0	0	0	0	0	<code>int m = (l + r) / 2;</code>
19,951,468	8,991,917	63	19,951,468	9,973,904	<code>if (a[m] >= x)</code>
.	<code>r = m;</code>
.	<code>else</code>
0	0	0	0	0	<code>l = m + 1;</code>
.	<code>}</code>
.	<code>return l;</code>
.	}

Unfortunately, Cachegrind only tracks memory accesses and branches. When the bottleneck is caused by something else, we need [other simulation tools](#).

Machine Code Analyzers

A *machine code analyzer* is a program that takes a small snippet of assembly code and [simulates](#) its execution on a particular microarchitecture using information available to compilers, and outputs the latency and throughput of the whole block, as well as cycle-perfect utilization of various resources within the CPU.

Using `llvm-mca`

There are many different machine code analyzers, but I personally prefer `llvm-mca`, which you can probably install via a package manager together with `clang`. You can also access it through a web-based tool called [UICA](#) or in the [Compiler Explorer](#) by selecting “Analysis” as the language.

What `llvm-mca` does is it runs a set number of iterations of a given assembly snippet and computes statistics about the resource usage of each instruction, which is useful for finding out where the bottleneck is.

We will consider the array sum as our simple example:

```
loop:  
    addl (%rax), %edx  
    addq $4, %rax  
    cmpq %rcx, %rax  
    jne loop
```

Here is its analysis with `llvm-mca` for the Skylake microarchitecture:

```
Iterations:      100  
Instructions:    400  
Total Cycles:   108  
Total uOps:     500  
  
Dispatch Width: 6  
uOps Per Cycle: 4.63  
IPC:            3.70  
Block RThroughput: 0.8
```

First, it outputs general information about the loop and the hardware:

- It “ran” the loop 100 times, executing 400 instructions in total in 108 cycles, which is the same as executing $\frac{400}{108} \approx 3.7$ [instructions per cycle](#) on average (IPC).
- The CPU is theoretically capable of executing up to 6 instructions per cycle ([dispatch width](#)).
- Each cycle in theory can be executed in 0.8 cycles on average ([block reciprocal throughput](#)).
- The “uOps” here are the micro-operations that the CPU splits each instruction into (e.g., fused load-add is composed of two uOps).

Then it proceeds to give information about each individual instruction:

```
Instruction Info:  
[1]: uOps  
[2]: Latency  
[3]: RThroughput  
[4]: MayLoad  
[5]: MayStore  
[6]: HasSideEffects (U)
```

[1]	[2]	[3]	[4]	[5]	[6]	Instructions:
2	6	0.50	*			addl (%rax), %edx
1	1	0.25				addq \$4, %rax

1	1	0.25								cmpq %rcx, %rax
1	1	0.50								jne -11

There is nothing there that there isn't in the [instruction tables](#):

- how many uOps each instruction is split into;
- how many cycles each instruction takes to complete (latency);
- how many cycles each instruction takes to complete in the amortized sense (reciprocal throughput), considering that several copies of it can be executed simultaneously.

Then it outputs probably the most important part — which instructions are executing when and where:

Resource pressure by instruction:										
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	Instructions:
-	-	0.01	0.98	0.50	0.50	-	-	0.01	-	addl (%rax), %edx
-	-	-	-	-	-	-	0.01	0.99	-	addq \$4, %rax
-	-	-	0.01	-	-	-	0.99	-	-	cmpq %rcx, %rax
-	-	0.99	-	-	-	-	-	0.01	-	jne -11

As the contention for execution ports causes [structural hazards](#), ports often become the bottleneck for throughput-oriented loops, and this chart helps diagnose why. It does not give you a cycle-perfect Gantt chart of something like that, but it gives you the aggregate statistics of the execution ports used for each instruction, which lets you find which one is overloaded.

Profiling

Staring at the source code or its assembly is a popular, but not the most effective way of finding performance issues. When the performance doesn't meet your expectations, you can identify the root cause much faster using one of the special program analysis tools collectively called *profilers*.

There are many different types of profilers. I like to think about them by analogy of how physicists and other natural scientists approach studying small things, picking the right tool depending on the required level of precision:

- When objects are on a micrometer scale, they use optical microscopes.
- When objects are on a nanometer scale, and light no longer interacts with them, they use electron microscopes.
- When objects are smaller than that (e.g., the insides of an atom), they resort to theories and assumptions about how things work (and test these assumptions using intricate and indirect experiments).

Similarly, there are three main profiling techniques, each operating by its own principles, having distinct areas of applicability, and allowing for different levels of precision:

- **Instrumentation** lets you time the program as a whole or by parts and count specific events you are interested in.
- **Statistical profiling** lets you go down to the assembly level and track various *hardware events* such as branch mispredictions or cache misses, which are critical for performance.
- **Program simulation** lets you go down to the individual cycle level and look into what is happening inside the CPU on each cycle when it is executing a small assembly snippet.

Practical algorithm design can be very much considered an empirical field too. We largely rely on the same experimental methods, although this is not because we don't know some of the fundamental secrets of nature but mostly because modern computers are just too complex to analyze — besides, this is also true that we, regular software engineers, can't know some of the details because of IP protection from hardware companies (in fact, considering that the most accurate x86 instruction tables are [reverse-engineered](#), there is a reason to believe that Intel doesn't know these details themselves).

In this chapter, we will study these three key profiling methods, as well as some time-tested practices for managing computational experiments involving performance evaluation.

Benchmarking

Most good software engineering practices in one way or another address the issue of making *development cycles* faster: you want to compile your software faster (build systems), catch bugs as soon as possible (static analysis, continuous integration), release as soon as the new version is ready (continuous deployment), and react to user feedback without much delay (agile development).

Performance engineering is not different. If you do it correctly, it should also resemble a cycle:

1. Run the program and collect metrics.
2. Figure out where the bottleneck is.
3. Remove the bottleneck and go to step 1.

In this section, we will talk about benchmarking and discuss some practical techniques that make this cycle shorter and help you iterate faster. Most of the advice comes from working on this book, so you can find many real examples of described setups in the [code repository](#) for this book.

Benchmarking Inside C++

There are several approaches to writing benchmarking code. Perhaps the most popular one is to include several same-language implementations you want to compare in one file, separately invoke them from the `main` function, and calculate all the metrics you want in the same source file.

The disadvantage of this method is that you need to write a lot of boilerplate code and duplicate it for each implementation, but it can be partially neutralized with metaprogramming. For example, when you are benchmarking multiple `gcd` implementations, you can reduce benchmarking code considerably with this higher-order function:

```
const int N = 1e6, T = 1e9 / N;
int a[N], b[N];

void timeit(int (*f)(int, int)) {
    clock_t start = clock();

    int checksum = 0;

    for (int t = 0; t < T; t++)
        for (int i = 0; i < n; i++)
            checksum ^= f(a[i], b[i]);
```

```

float seconds = float(clock() - start) / CLOCKS_PER_SEC;

printf("checksum: %d\n", checksum);
printf("%.2f ns per call\n", 1e9 * seconds / N / T);
}

int main() {
    for (int i = 0; i < N; i++)
        a[i] = rand(), b[i] = rand();

    timeit(std::gcd);
    timeit(my_gcd);
    timeit(my_another_gcd);
    // ...

    return 0;
}

```

This is a very low-overhead method that lets you run more experiments and [get more accurate results](#) from them. You still have to perform some repeated actions, but they can be largely automated with frameworks, [Google benchmark library](#) being the most popular choice for C++. Some programming languages also have handy built-in tools for benchmarking: special mention here goes to [Python's `timeit` function](#) and [Julia's `@benchmark` macro](#).

Although *efficient* in terms of execution speed, C and C++ are not the most *productive* languages, especially when it comes to analytics. When your algorithm depends on some parameters such as the input size, and you need to collect more than just one data point from each implementation, you really want to integrate your benchmarking code with the outside environment and analyze the results using something else.

Splitting Up Implementations

One way to improve modularity and reusability is to separate all testing and analytics code from the actual implementation of the algorithm, and also make it so that different versions are implemented in separate files, but have the same interface.

In C/C++, you can do this by creating a single header file (e.g., `gcd.hh`) with a function interface and all its benchmarking code in `main`:

```

int gcd(int a, int b); // to be implemented

// for data structures, you also need to create a setup function
// (unless the same preprocessing step for all versions would suffice)

int main() {
    const int N = 1e6, T = 1e9 / N;
    int a[N], b[N];
    // careful: local arrays are allocated on the stack and may cause stack overflow
    // for large arrays, allocate with "new" or create a global array

```

```

for (int i = 0; i < N; i++)
    a[i] = rand(), b[i] = rand();

int checksum = 0;

clock_t start = clock();

for (int t = 0; t < T; t++)
    for (int i = 0; i < n; i++)
        checksum += gcd(a[i], b[i]);

float seconds = float(clock() - start) / CLOCKS_PER_SEC;

printf("%d\n", checksum);
printf("%.2f ns per call\n", 1e9 * seconds / N / T);

return 0;
}

```

Then you create many implementation files for each algorithm version (e.g., `v1.cc`, `v2.cc`, and so on, or some meaningful names if applicable) that all include that single header file:

```

#include "gcd.hh"

int gcd(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}

```

The whole purpose of doing this is to be able to benchmark a specific algorithm version from the command line without touching any source code files. For this purpose, you may also want to expose any parameters that it may have — for example, by parsing them from the command line arguments:

```

int main(int argc, char* argv[]) {
    int N = (argc > 1 ? atoi(argv[1]) : 1e6);
    const int T = 1e9 / N;

    // ...
}

```

Another way to do it is to use C-style global defines and then pass them with the `-D N=...` flag during compilation:

```

#ifndef N
#define N 1000000
#endif

const int T = 1e9 / N;

```

This way you can make use of compile-time constants, which may be very beneficial for the performance of some algorithms, at the expense of having to re-build the program each time you want to change the parameter, which considerably increases the time you need to collect metrics across a range of parameter values.

Makefiles

Splitting up source files allows you to speed up compilation using a caching build system such as [Make](#).

I usually carry a version of this Makefile across my projects:

```
compile = g++ -std=c++17 -O3 -march=native -Wall

%: %.cc gcd.hh
    $(compile) $< -o $@

%.s: %.cc gcd.hh
    $(compile) -S -fverbose-asm $< -o $@

%.run: %
    ./$<

.PHONY: %.run
```

You can now compile `example.cc` with `make example`, and automatically run it with `make example.run`.

You can also add scripts for calculating statistics in the Makefile, or incorporate it with `perf stat` calls to make profiling automatic.

Jupyter Notebooks

To speed up high-level analytics, you can create a Jupyter notebook where you put all your scripts and do all the plots.

It is convenient to add a wrapper for benchmarking an implementation, which just returns a scalar result:

```
def bench(source, n=2**20):
    !make -s {source}
    if _exit_code != 0:
        raise Exception("Compilation failed")
    res = !.{source} {n} {q}
    duration = float(res[0].split()[0])
    return duration
```

Then you can use it to write clean analytics code:

```
ns = list(int(1.17**k) for k in range(30, 60))
baseline = [bench('std_lower_bound', n=n) for n in ns]
results = [bench('my_binary_search', n=n) for n in ns]
```

```

# plotting relative speedup for different array sizes
import matplotlib.pyplot as plt

plt.plot(ns, [x / y for x, y in zip(baseline, results)])
plt.show()

```

Once established, this workflow makes you iterate much faster and focus on optimizing the algorithm itself.

Getting Accurate Results

It is not uncommon for there to be two library algorithm implementations, each maintaining its own benchmarking code, and each claiming to be faster than the other. This confuses everyone involved, especially the users, who have to somehow choose between the two.

Situations like these are usually not caused by fraudulent actions by their authors; they just have different definitions of what “faster” means, and indeed, defining and using just one performance metric is often very problematic.

Measuring the Right Thing

There are many things that can introduce bias into benchmarks.

Differing datasets. There are many algorithms whose performance somehow depends on the dataset distribution. In order to define, for example, what the fastest sorting, shortest path, or binary search algorithms are, you have to fix the dataset on which the algorithm is run.

This sometimes applies even to algorithms that process a single piece of input. For example, it is not a good idea to feed GCD implementations sequential numbers because it makes branches very predictable:

```

// don't do this
int checksum = 0;

for (int a = 0; a < 1000; a++)
    for (int b = 0; b < 1000; b++)
        checksum ^= gcd(a, b);

```

However, if we sample these same numbers randomly, branch prediction becomes much harder, and the benchmark takes longer time, despite processing the same input, but in altered order:

```

int a[1000], b[1000];

for (int i = 0; i < 1000; i++)
    a[i] = rand() % 1000, b[i] = rand() % 1000;

int checksum = 0;

for (int t = 0; t < 1000; t++)
    for (int i = 0; i < 1000; i++)
        checksum += gcd(a[i], b[i]);

```

Although the most logical choices for most cases is to just sample data uniformly at random, many real-world applications have distributions that are far from uniform, so you can't pick just one. In general, a good benchmark should be application-specific, and use the dataset that is as representing of your real use case as possible.

Multiple objectives. Some algorithm design problems have more than one key objective. For example, hash tables, in addition to being highly dependant on the distribution of keys, also need to carefully balance:

- memory usage,
- latency of add query,
- latency of positive membership query,
- latency of negative membership query.

The only way to choose between hash table implementations is to try and put multiple variants into the application.

Latency vs Throughput. Another aspect that people often overlook is that the execution time can be defined in more than one way, even for a single query.

When you write code like this:

```
for (int i = 0; i < N; i++)
    q[i] = rand();

int checksum = 0;

for (int i = 0; i < N; i++)
    checksum ^= lower_bound(q[i]);
```

and then time the whole thing and divide it by the number of iterations, you are actually measuring the *throughput* of the query — how many operations it can process per unit of time. This is usually less than the time it actually takes to process one operation separately because of interleaving.

To measure actual *latency*, you need to introduce a dependency between the invocations:

```
for (int i = 0; i < N; i++)
    checksum ^= lower_bound(checksum ^ q[i]);
```

It usually makes the most difference in algorithms with possible pipeline stall issues, e.g., when comparing branchy and branch-free algorithms.

Cold cache. Another source of bias is the *cold cache effect*, when memory reads initially take longer time because the required data is not in cache yet.

This is solved by making a *warm-up run* before starting measurements:

```
// warm-up run

volatile checksum = 0;

for (int i = 0; i < N; i++)
    checksum ^= lower_bound(q[i]);
```

```
// actual run

clock_t start = clock();
checksum = 0;

for (int i = 0; i < N; i++)
    checksum ^= lower_bound(q[i]);
```

It is also sometimes convenient to combine the warm-up run with answer validation, if it is more complicated than just computing some sort of checksum.

Over-optimization. Sometimes the benchmark is outright erroneous because the compiler just optimized the benchmarked code away. To prevent the compiler from cutting corners, you need to add checksums and either print them somewhere or add the `volatile` qualifier, which also prevents any sort of interleaving of loop iterations.

For algorithms that only write data, you can use the `__sync_synchronize()` intrinsic to add a memory fence and prevent the compiler from accumulating updates.

Reducing Noise

The issues we've described produce *bias* in measurements: they consistently give advantage to one algorithm over the other. There are other types of possible problems with benchmarking that result in either unpredictable skews or just completely random noise, thus increasing *variance*.

These types of issues are caused by side effects and some sort of external noise, mostly due to noisy neighbors and CPU frequency scaling:

- If you benchmark a compute-bound algorithm, measure its performance in cycles using `perf stat`: this way it will be independent of clock frequency, fluctuations of which is usually the main source of noise.
- Otherwise, set core frequency to what you expect it to be and make sure nothing interferes with it. On Linux you can do it with `cpupower frequency-set -g powersave` to put it to minimum or `sudo cpupower frequency-set -g ondemand` to enable turbo boost). I use a [convenient GNOME shell extension](#) that has a separate button to do it.
- If applicable, turn hyper-threading off and attach jobs to specific cores. Make sure no other jobs are running on the system, turn off networking and try not to fiddle with the mouse.

You can't remove noises and biases completely. Even a program's name can affect its speed: the executable's name ends up in an environment variable, environment variables end up on the call stack, and so the length of the name affects stack alignment, which can result in data accesses slowing down due to crossing cache line or memory page boundaries.

It is important to account for the noise when guiding optimizations and especially when reporting results to someone else. Unless you are expecting a 2x kind of improvement, treat all microbenchmarks the same way as A/B testing.

When you run a program on a laptop for under a second, a $\pm 5\%$ fluctuation in performance is completely normal. So, if you want to decide whether to revert or keep a potential $+1\%$ improvement, run it until you reach statistical significance, which you can determine by calculating variances and p-values.

Further Reading

Interested readers can explore this comprehensive [list of experimental computer science resources](#) by Dror Feitelson, perhaps starting with “[Producing Wrong Data Without Doing Anything Obviously Wrong](#)” by Todd Mytkowicz et al.

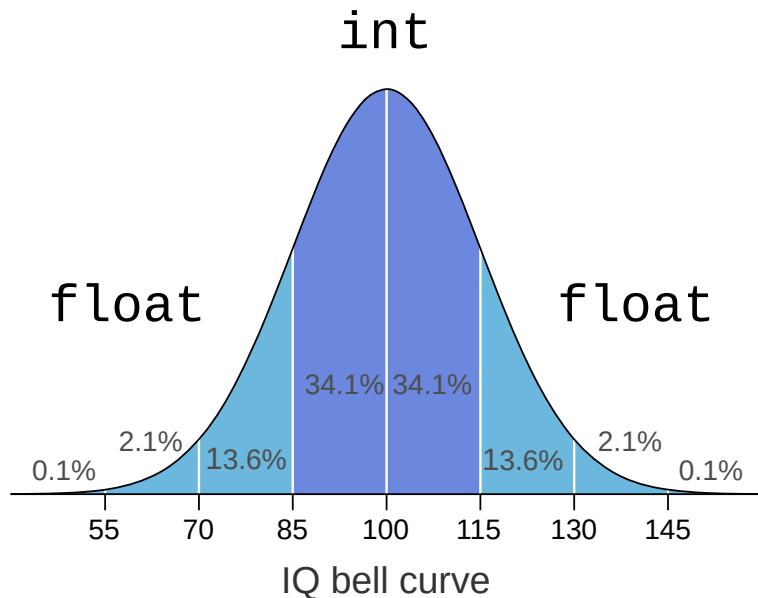
You can also watch [this great talk](#) by Emery Berger on how to do statistically sound performance evaluation.

Arithmetic

The users of floating-point arithmetic deserve one of these IQ bell curve memes — because this is how the relationship between it and most people typically proceeds:

- Beginner programmers use it everywhere as if it was some magic unlimited-precision data type.
- Then they discover that $0.1 + 0.2 \neq 0.3$ or some other quirk like that, freak out, start thinking that some random error term is added to every computation, and for many years avoid any real data types completely.
- Then they finally man up, read the specification of how IEEE-754 floats work and start using them appropriately.

Unfortunately, too many people are still at stage 2, breeding various misconceptions about floating-point arithmetic — thinking that it is fundamentally imprecise and unstable, and slower than integer arithmetic.



But these are all just myths. Floating-point arithmetic is often *faster* than integer arithmetic because of specialized instructions, and real number representations are thoroughly standardized and follow simple and deterministic rules in terms of rounding, allowing you to manage computational errors reliably.

In fact, it is so reliable that some high-level programming languages, most notably JavaScript, don't have integers at all. In JavaScript, there is only one `number` type, which is internally stored as a 64-bit `double`, and due to the way floating-point arithmetic works, all integer numbers between -2^{53} and 2^{53} and results of computations involving them can be stored exactly, so from a programmer's perspective, there is little practical need for a separate integer type.

One notable exception is when you need to perform bitwise operations with numbers, which *floating-point units* (the coprocessors responsible for operations on floating-point numbers) typically don't support. In this case, they need to be converted to integers, which is so frequently used in JavaScript-enabled browsers that arm added a special “FJCVTZS” instruction that stands for “Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero” and does what it says it does — converts real to integer the exact same way as JavaScript — which is an interesting example of the software-hardware feedback loop in action.

But unless you are a JavaScript developer who uses real types exclusively to emulate integer arithmetic, you probably need a more in-depth guide about floating-point arithmetic, so we are going to start with a broader subject.

Real Number Representations

If you need to deal with real (non-integer) numbers, you have several options with varying applicability. Before jumping straight to floating-point numbers, which is what most of this article is about, we want to discuss the available alternatives and the motivation behind them — after all, people who avoid floating-point arithmetic do have a point.

Symbolic Expressions

The first and the most cumbersome approach is to store not the resulting values themselves but the algebraic expressions that produce them.

Here is a simple example. In some applications, such as computational geometry, apart from adding, subtracting and multiplying numbers, you also need to divide without rounding, producing a rational number, which can be exactly represented with a ratio of two integers:

```
struct r {
    int x, y;
};

r operator+(r a, r b) { return {a.x * b.y + a.y * b.x, a.y * b.y}; }
r operator*(r a, r b) { return {a.x * b.x, a.y * b.y}; }
r operator/(r a, r b) { return {a.x * b.x, a.y * b.y}; }
bool operator<(r a, r b) { return a.x * b.y < b.x * a.y; }
// ...and so on, you get the idea
```

This ratio can be made irreducible, which would even make this representation unique:

```
struct r {
    int x, y;
    r(int x, int y) : x(x), y(y) {
        if (y < 0)
            x = -x, y = -y;
        int g = gcd(x, y);
        x /= g;
        y /= g;
    }
};
```

This is how *computer algebra* systems such as WolframAlpha and SageMath work: they operate solely on symbolic expressions and avoid evaluating anything as real values.

With this method, you get absolute precision, and it works well when you have a limited scope such as only supporting rational numbers. But this comes at a large computational cost because in general, you would need to somehow store the whole history of operations that led to the result and take it into account each time you perform a new operation — which quickly becomes unfeasible as the history grows.

Fixed-Point Numbers

Another approach is to stick to integers, but treat them as if they were multiplied by a fixed constant. This is essentially the same as changing units of measurement for more up-to-scale ones.

Because some values can't be represented exactly, this makes computations imprecise: you need to round the results to nearest representable value.

This approach is commonly used in financial software, where you *really* need a straightforward way to manage rounding errors so that the final numbers add up. For example, NASDAQ uses $\frac{1}{10000}$ -th of a dollar as its base unit in its stock listings, meaning that you get the precision of exactly 4 digits after comma in all transactions.

```

struct money {
    uint v; // 1/10000th of a dollar
};

std::string to_string(money) {
    return std::format("${0}.{1:04d}", v / 10000, v % 10000);
}

money operator*(money x, money y) { return {x.v * y.v / 10000}; }

```

Apart from introducing rounding errors, a bigger problem is that they become useless when the scaling constant is misplaced. If the numbers you are working with are too large, then the internal integer value will overflow, and if the numbers are too small, they will be just rounded down to zero. Interestingly, the former case once became an issue on NASDAQ when the Berkshire Hathaway stock price approached $\frac{2^{32}-1}{10000} = \$429,496.7295$ and could no longer fit in an unsigned 32-bit integer.

This problem makes fixed-point arithmetic fundamentally unsuitable for applications where you need to use both small and large numbers, for example, evaluating certain physics equations:

$$E = mc^2$$

In this particular one, m is typically of the same order of magnitude as the mass of a proton ($1.67 \cdot 10^{-27}$ kg) and c is the speed of light ($3 \cdot 10^9$ m/s).

Floating-Point Numbers

In most numerical applications, we are mainly concerned with the relative error. We want the result of our computations to differ from the truth by no more than, say, 0.01%, and we don't really care what that 0.01% equates to in absolute units.

Floating-point numbers solve this by storing a certain number of the most significant digits and the order of magnitude of the number. More precisely, they are represented with an integer (called *significand* or *mantissa*) and scaled using an exponent of some fixed base — most commonly, 2 or 10. For example:

$$1.2345 = \underbrace{12345}_{\text{mantissa}} \times \underbrace{10}_{\text{base}}^{\text{exponent}}$$

Computers operate on fixed-length binary words, so when designing a floating-point format for hardware, you'd want a fixed-length binary format where some bits are dedicated for the mantissa (for more precision) and some for the exponent (for more range).

This handmade float implementation hopefully conveys the idea:

```

// DIY floating-point number
struct fp {
    int m; // mantissa
    int e; // exponent
};

```

This way we can represent numbers in the form $\pm m \times 2^e$ where both m and e are bounded *and possibly negative* integers — which would correspond to negative or small numbers respectively. The distribution of these numbers is very much non-uniform: there are roughly as many numbers in the $[0, 1)$ range as in the $[1, +\infty)$ range.

Note that these representations are not unique for some numbers. For example, number 1 can be represented as

$$1 \times 2^0 = 2 \times 2^{-1} = 256 \times 2^{-8}$$

and in 28 other ways that don't overflow the mantissa.

This can be problematic for some applications, such as comparisons or hashing. To fix this, we can *normalize* these representations using a certain convention. In decimal, the standard form is to always put the comma after the first digit (6.022e23), and for binary, we can do the same:

$$42 = 10101_2 = 1.0101_2 \times 2^5$$

Notice that, following this rule, the first bit is always 1. It is redundant to store it explicitly, so we will just pretend that it's there and only store the other bits, which correspond to some rational number in the $[0, 1)$ range. The set of representable numbers is now roughly

$$\{\pm (1 + m) \cdot 2^e \mid m = \frac{x}{2^{32}}, x \in [0, 2^{32})\}$$

Since m is now a nonnegative value, we will now make it unsigned integer, and instead add a separate Boolean field for the sign of the number:

```
struct fp {
    bool s;      // sign: "0" for "+", "1" for "-"
    unsigned m; // mantissa
    int e;       // exponent
};
```

Now, let's try to implement some arithmetic operation — for example, multiplication — using our handmade floats. Using the new formula, the result should be:

$$\begin{aligned} c &= a \cdot b \\ &= (s_a \cdot (1 + m_a) \cdot 2^{e_a}) \cdot (s_b \cdot (1 + m_b) \cdot 2^{e_b}) \\ &= s_a \cdot s_b \cdot (1 + m_a) \cdot (1 + m_b) \cdot 2^{e_a} \cdot 2^{e_b} \\ &= \underbrace{s_a \cdot s_b}_{s_c} \cdot \underbrace{(1 + m_a + m_b + m_a \cdot m_b)}_{m_c} \cdot 2^{\overbrace{e_a + e_b}^{e_c}} \end{aligned}$$

The groupings now seem straightforward to calculate, but there are two nuances:

- The new mantissa is now in the $[0, 3)$ range. We need to check if it is larger than 1 and normalize the representation, applying the following formula: $1 + m = (1 + 1) + (m - 1) = (1 + \frac{m-1}{2}) \cdot 2$.
- The resulting number can be (and very likely is) not representable exactly due to the lack of precision. We need twice as many bits to account for the $m_a \cdot m_b$ term, and the best we can do here is to round it to the nearest representable number.

Since we need some extra bits to properly handle the mantissa overflow issue, we will reserve one bit from m thus limiting it to $[0, 2^{31})$ range.

```
fp operator*(fp a, fp b) {
    fp c;
    c.s = a.s ^ b.s;
    c.e = a.e + b.e;

    uint64_t x = a.m, y = b.m; // casting to wider types
    uint64_t m = (x << 31) + (y << 31) + x * y; // 62- or 63-bit intermediate result
    if (m & (1<<62)) { // checking for overflow
```

```

    m -= (1<<62); // m -= 1;
    m >>= 1;
    c.e++;
}
m += (1<<30); // "rounding" by adding 0.5 to a value that will be floored next
c.m = m >> 31;

return c;
}

```

Many applications that require higher levels of precision use software floating-point arithmetic in a similar fashion. But of course, you don't want to execute a sequence of 10 or so instructions that this code compiles to each time you want to multiply two real numbers, so on modern CPUs, floating-point arithmetic is implemented in hardware — usually as separate coprocessors due to its complexity.

The *floating-point unit* of x86 (often referred to as x87) has separate registers and its own tiny instruction set that supports memory operations, basic arithmetic, trigonometry, and some common operations such as logarithm, exponent, and square root. To make these operations properly work together, some additional details of floating-point number representation need to be clarified — which we will do in the next section.

The way rounding works in hardware floats is remarkably simple: it occurs if and only if the result of the operation is not representable exactly, and by default gets rounded to the nearest representable number (in case of a tie preferring the number that ends with a zero).

Consider the following code snippet:

```

float x = 0;
for (int i = 0; i < (1 << 25); i++)
    x++;
printf("%f\n", x);

```

Instead of printing $2^{25} = 33554432$ (what the result mathematically should be), it outputs $16777216 = 2^{24}$. Why?

When we repeatedly increment a floating-point number x , we eventually hit a point where it becomes so big that $(x + 1)$ gets rounded back to x . The first such number is 2^{24} (the number of mantissa bits plus one) because

$$2^{24} + 1 = 2^{24} \cdot 1.\underbrace{0\dots 0}_{\times 23} 1$$

has the exact same distance from 2^{24} and $(2^{24} + 1)$ but gets rounded down to 2^{24} by the above-stated tie-breaker rule. At the same time, the increment of everything lower than that can be represented exactly, so no rounding happens in the first place.

Rounding Errors and Operation Order

The result of a floating-point computation may depend on the order of operations despite being algebraically correct.

For example, while the operations of addition and multiplication are commutative and associative in the pure mathematical sense, their rounding errors are not: when we have three floating-point variables x , y , and z , the result of $(x + y + z)$ depends on the order of summation. The same non-commutativity principle applies to most if not all other floating-point operations.

Compilers are not allowed to produce non-spec-compliant results, so this annoying nuance disables some potential optimizations that involve rearranging operands in arithmetic. You can disable this strict compliance with the `-ffast-math` flag in GCC and Clang. If we add it and re-compile the code snippet above, it runs

considerably faster and also happens to output the correct result, 33554432 (although you need to be aware that the compiler also could have chosen a less precise computation path).

Rounding Modes

Apart from the default mode (also known as Banker's rounding), you can set other rounding logic with 4 more modes:

- round to nearest, with perfect ties always rounding “away” from zero;
- round up (toward $+\infty$; negative results thus round toward zero);
- round down (toward $-\infty$; negative results thus round away from zero);
- round toward zero (a truncation of the binary result).

For example, if you call `fesetround(FE_UPWARD)` before running the loop above, it outputs not 2^{24} , and not even 2^{25} , but $67108864 = 2^{26}$. This happens because when we get to 2^{24} , $(x+1)$ starts rounding to the next nearest representable number ($x+2$), and we reach 2^{25} in half the time, and after that, $(x+1)$ rounds up to $(x+4)$, and we start going four times as fast.

One of the uses for the alternative rounding modes is for diagnosing numerical instability. If the results of an algorithm substantially vary when switching between rounding to the positive and negative infinities, it indicates susceptibility to round-off errors.

This test is often better than switching all computations to lower precision and checking whether the result changed by too much because the default round-to-nearest policy converges to the correct “expected” value given enough averaging: half of the time the errors are rounding up, and the other they are rounding down — so, statistically, they cancel each other.

Measuring Errors

It seems surprising to expect this guarantee from hardware that performs complex calculations such as natural logarithms and square roots, but this is it: you are guaranteed to get the highest precision possible from all operations. This makes it remarkably easy to analyze round-off errors, as we will see in a bit.

There are two natural ways to measure computational errors:

- The engineers who create hardware or spec-compliant exact software are concerned with *units in the last place* (ulp), which is the distance between two numbers in terms of how many representable numbers can fit between the precise real value and the actual result of the computation.
- People that are working on numerical algorithms care about *relative precision*, which is the absolute value of the approximation error divided by the real answer: $|\frac{v-v'}{v}|$.

In either case, the usual tactic to analyze errors is to assume the worst case and simply bound them.

If you perform a single basic arithmetic operation, then the worst thing that can happen is the result rounding to the nearest representable number, meaning that the error does not exceed 0.5 ulps. To reason about relative errors the same way, we can define a number ϵ called *machine epsilon*, equal to the difference between 1 and the next representable value (which should be equal to 2 to the negative power of however many bits are dedicated to mantissa).

This means that if after a single arithmetic operation you get result x , then the real value is somewhere in the range

$$[x \cdot (1 - \epsilon), x \cdot (1 + \epsilon)]$$

The omnipresence of errors is especially important to remember when making discrete “yes or no” decisions based on the results of floating-point calculations. For example, here is how you should check for equality:

```
const float eps = std::numeric_limits<float>::epsilon; // ~2^(-23)
bool eq(float a, float b) {
```

```

    return abs(a - b) <= eps;
}

```

The value of `eps` should depend on the application: the one above — the machine epsilon for `float` — is only good for no more than one floating-point operation.

Interval Arithmetic

An algorithm is called *numerically stable* if its error, whatever its cause, does not grow much larger during the calculation. This can only happen if the problem itself is *well-conditioned*, meaning that the solution changes only by a small amount if the input data are changed by a small amount.

When analyzing numerical algorithms, it is often useful to adopt the same method that is used in experimental physics: instead of working with unknown real values, we will work with the intervals where they may be in.

For example, consider a chain of operations where we consecutively multiply a variable by arbitrary real numbers:

```

float x = 1;
for (int i = 0; i < n; i++)
    x *= a[i];

```

After the first multiplication, the value of x relative to the value of the real product is bounded by $(1 + \epsilon)$, and after each additional multiplication, this upper bound is multiplied by another $(1 + \epsilon)$. By induction, after n multiplications, the computed value is bound by $(1 + \epsilon)^n = 1 + n\epsilon + O(\epsilon^2)$ and a similar lower bound.

This implies that the relative error is $O(n\epsilon)$, which is sort of okay, because usually $n \ll \frac{1}{\epsilon}$.

For example of a numerically *unstable* computation, consider the function

$$f(x, y) = x^2 - y^2$$

Assuming $x > y$, the maximum value this function can return is roughly

$$x^2 \cdot (1 + \epsilon) - y^2 \cdot (1 - \epsilon)$$

corresponding to the absolute error of

$$x^2 \cdot (1 + \epsilon) - y^2 \cdot (1 - \epsilon) - (x^2 - y^2) = (x^2 + y^2) \cdot \epsilon$$

and hence the relative error of

$$\frac{x^2 + y^2}{x^2 - y^2} \cdot \epsilon$$

If x and y are close in magnitude, the error will be $O(\epsilon \cdot |x|)$.

Under direct computation, the subtraction “magnifies” the errors of squaring. But this can be fixed by instead using the following formula:

$$f(x, y) = x^2 - y^2 = (x + y) \cdot (x - y)$$

In this one, it is easy to show that the error is bound by $\epsilon \cdot |x - y|$. It is also faster because it needs 2 additions and 1 multiplication: one fast addition more and one slow multiplication less compared to the original.

Kahan Summation

From the previous example, we can see that long chains of operations are not a problem, but adding and subtracting numbers of different magnitude is. The general approach to dealing with such problems is to try to keep big numbers with big numbers and small numbers with small numbers.

Consider the standard summation algorithm:

```
float s = 0;
for (int i = 0; i < n; i++)
    s += a[i];
```

Since we are performing summations and not multiplications, its relative error is no longer just bounded by $O(\epsilon \cdot n)$, but heavily depends on the input.

In the most ridiculous case, if the first value is 2^{24} and the other values are equal to 1, the sum is going to be 2^{24} regardless of n , which can be verified by executing the following code and observing that it simply prints $16777216 = 2^{24}$ twice:

```
const int n = (1<<24);
printf("%d\n", n);

float s = n;
for (int i = 0; i < n; i++)
    s += 1.0;

printf("%f\n", s);
```

This happens because `float` has only 23 mantissa bits, and so $2^{24} + 1$ is the first integer number that can't be represented exactly and has to be rounded down, which happens every time we try to add 1 to $s = 2^{24}$. The error is indeed $O(n \cdot \epsilon)$ but in terms of the absolute error, not the relative one: in the example above, it is 2, and it would go up to infinity if the last number happened to be -2^{24} .

The obvious solution is to switch to a larger type such as `double`, but this isn't really a scalable method. An elegant solution is to store the parts that weren't added in a separate variable, which is then added to the next variable:

```
float s = 0, c = 0;
for (int i = 0; i < n; i++) {
    float y = a[i] - c; // c is zero on the first iteration
    float t = s + y;    // s may be big and y may be small, losing low-order bits of y
    c = (t - s) - y;   // (t - s) cancels high-order part of y
    s = t;
}
```

This trick is known as *Kahan summation*. Its relative error is bounded by $2\epsilon + O(ne^2)$: the first term comes from the very last summation, and the second term is due to the fact that we work with less-than-epsilon errors on each step.

Of course, a more general approach that works not just for array summation would be to switch to a more precise data type, like `double`, also effectively squaring the machine epsilon. Furthermore, it can (sort of) be scaled by bundling two `double` variables together: one for storing the value and another for its non-representable errors so that they represent the value $a + b$. This approach is known as double-double arithmetic, and it can be similarly generalized to define quad-double and higher precision arithmetic.

When we designed our DIY floating-point type, we omitted quite a lot of important little details:

- How many bits do we dedicate for the mantissa and the exponent?
- Does a 0 sign bit mean +, or is it the other way around?
- How are these bits stored in memory?

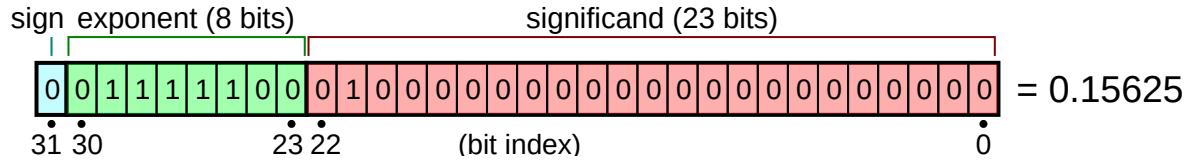
- How do we represent 0?
- How exactly does rounding happen?
- What happens if we divide by zero?
- What happens if we take the square root of a negative number?
- What happens if we increment the largest representable number?
- Can we somehow detect if one of the above three happened?

Most of the early computers didn't support floating-point arithmetic, and when vendors started adding floating-point coprocessors, they had slightly different visions for what the answers to these questions should be. Diverse implementations made it difficult to use floating-point arithmetic reliably and portably — especially for the people who develop compilers.

In 1985, the Institute of Electrical and Electronics Engineers published a standard (called IEEE 754) that provided a formal specification of how floating-point numbers should work, which was quickly adopted by the vendors and is now used in virtually all general-purpose computers.

Float Formats

Similar to our handmade float implementation, hardware floats use one bit for sign and a variable number of bits for the exponent and the mantissa parts. For example, the standard 32-bit `float` encoding uses the first (highest) bit for sign, the next 8 bits for the exponent, and the 23 remaining bits for the mantissa.



One of the reasons why they are stored in this exact order is that it is easier to compare and sort them: you can use mostly the same comparator circuit as for unsigned integers, except for maybe flipping some bits in case one of the numbers is negative.

For the same reason, the exponent is *biased*: the actual value is 127 less than the stored unsigned integer, which lets us also cover the values less than one (with negative exponents). In the example above:

$$(-1)^0 \times 2^{01111100_2 - 127} \times (1 + 2^{-2}) = 2^{124 - 127} \times 1.25 = \frac{1.25}{8} = 0.15625$$

IEEE 754 and a few consequent standards define not one but *several* representations that differ in sizes, most notably:

Type	Sign	Exponent	Mantissa	Total bits	Approx. decimal digits
single	1	8	23	32	~7.2
double	1	11	52	64	~15.9
half	1	5	10	16	~3.3
extended	1	15	64	80	~19.2
quadruple	1	15	112	128	~34.0
bfloat16	1	8	7	16	~2.3

Their availability ranges from chip to chip:

- Most CPUs support single- and double-precision — which is what `float` and `double` types refer to in C.
- Extended formats are exclusive to x86, and are available in C as the `long double` type, which falls back to double precision on Arm CPUs. The choice of 64 bits for mantissa is so that every `long long`

integer can be represented exactly. There is also a 40-bit format that similarly allocates 32 mantissa bits.

- Quadruple as well as the 256-bit “octuple” formats are only used for specific scientific computations and are not supported by general-purpose hardware.
- Half-precision arithmetic only supports a small subset of operations and is generally used for applications such as machine learning, especially neural networks, because they tend to perform large amounts of calculations but don’t require high levels of precision.
- Half-precision is being gradually replaced by bfloat, which trades off 3 mantissa bits to have the same range as single-precision, enabling interoperability with it. It is mostly being adopted by specialized hardware: TPUs, FGPAs, and GPUs. The name stands for “Brain float.”

Lower-precision types need less memory bandwidth to move them around and usually take fewer cycles to operate on (e.g., the division instruction may take x , y , or z cycles depending on the type), which is why they are preferred when error tolerance allows it.

Deep learning, emerging as a very popular and computationally-intensive field, created a huge demand for low-precision matrix multiplication, which led to manufacturers developing separate hardware or at least adding specialized instructions that support these types of computations — most notably, Google developing a custom chip called TPU (*tensor processing unit*) that specializes on multiplying 128-by-128 bfloat matrices, and NVIDIA adding “tensor cores,” capable of performing 4-by-4 matrix multiplication in one go, to all their newer GPUs.

Apart from their sizes, most of the behavior is the same between all floating-point types, which we will now clarify.

Handling Corner Cases

The default way integer arithmetic deals with corner cases such as division by zero is to crash.

Sometimes a software crash, in turn, causes a real, physical one. In 1996, the maiden flight of the Ariane 5 (the space launch vehicle that ESA uses to lift stuff into low Earth orbit) ended in a catastrophic explosion due to the policy of aborting computation on arithmetic error, which in this case was a floating-point to integer conversion overflow, that led to the navigation system thinking that it was off course and making a large correction, eventually causing the disintegration of a \$200M rocket.

There is a way to gracefully handle corner cases like these: hardware interrupts. When an exception occurs, the CPU

- interrupts the execution of a program;
- packs all relevant information into a data structure called “interrupt vector”;
- passes it to the operating system, which in turn either calls the handling code if it exists (the “try-except” block) or terminates the program otherwise.

This is a complex mechanism that deserves an article of its own, but since this is a book about performance, the only thing you need to know is that they are quite slow and not desirable in real-time systems such as navigating rockets.

NaNs, Zeros and Infinities

Floating-point arithmetic often deals with noisy, real-world data. Exceptions there are much more common than in the integer case, and for this reason, the default behavior when handling them is different. Instead of crashing, the result is substituted with a special value without interrupting the program execution (unless the programmer explicitly wants it to).

The first type of such value is the two infinities: a positive and a negative one. They are generated if the result of an operation can’t fit within the representable range, and they are treated as such in arithmetic.

$$\begin{aligned}-\infty < x < \infty \\ \infty + x = \infty \\ x \div \infty = 0\end{aligned}$$

What happens if we, say, divide a value by zero? Should it be a negative or a positive infinity? This case is actually unambiguous because, somewhat less intuitively, there are also two zeros: a positive and a negative one.

$$\frac{1}{+0} = +\infty \quad \frac{1}{-0} = -\infty$$

Fun fact: `x + 0.0` can't be folded to `x`, but `x + (-0.0)` can, so the negative zero is a better initializer value than the positive zero as it is more likely to be optimized away by the compiler. The reason why `+0.0` doesn't work is that IEEE says that `+0.0 + -0.0 == +0.0`, so it will give a wrong answer for `x = -0.0`. The presence of two zeros frequently causes headaches like this — good news that you can pass `-fno-signed-zeros` to the compiler if you want to disable this behavior.

Zeros are encoded by setting all bits to zero, except for the sign bit in the negative case. Infinities are encoded by setting all their exponent bits to one and all mantissa bits to zero, with the sign bit distinguishing between positive and negative infinity.

The other type is the “not-a-number” (NaN), which is generated as the result of mathematically incorrect operations:

$$\log(-1), \arccos(1.01), \infty - \infty, -\infty + \infty, 0 \times \infty, 0 \div 0, \infty \div \infty$$

There are two types of NaNs: a *signaling NaN* and a *quiet NaN*. A signaling NaN raises an exception flag, which may or may not cause immediate hardware interrupt depending on the FPU configuration, while a quiet NaN just propagates through almost every arithmetic operation, resulting in more NaNs.

In binary, both NaNs have their exponent bits all set and the mantissa part being anything other than all zeros (to distinguish them from infinities). Note that there are *very* many valid encodings for a NaN.

Further Reading

If you are so inclined, you can read the classic “What Every Computer Scientist Should Know About Floating-Point Arithmetic” (1991) and the paper introducing Grisu3, the current state-of-the-art for printing floating-point numbers.

Reaching the maximum possible precision is rarely required from a practical algorithm. In real-world data, modeling and measurement errors are usually several orders of magnitude larger than the errors that come from rounding floating-point numbers and such, and we are often perfectly happy with picking an approximate method that trades off precision for speed.

In this section, we introduce one of the most important building blocks in such approximate, numerical algorithms: *Newton's method*.

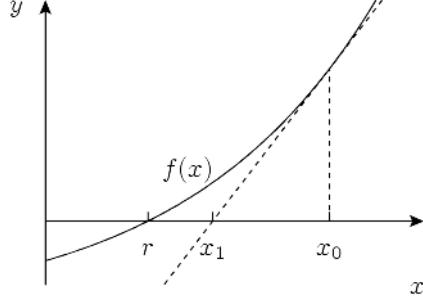
Newton's Method

Newton's method is a simple yet very powerful algorithm for finding approximate roots of real-valued functions, that is, the solutions to the following generic equation:

$$f(x) = 0$$

The only thing assumed about the function f is that at least one root exists and that $f(x)$ is continuous and differentiable on the search interval. There are also some boring corner cases, but they almost never occur in practice, so we will just informally say that the function is “good.”

The main idea of the algorithm is to start with some initial approximation x_0 and then iteratively improve it by drawing the tangent to the graph of the function at $x = x_i$ and setting the next approximation x_{i+1} equal to the x -coordinate of its intersection with the x -axis. The intuition is that if the function f is “good” and x_i is already close enough to the root, then x_{i+1} will be even closer.



To obtain the point of intersection for x_n , we need to equal its tangent line function to zero:

$$0 = f(x_i) + (x_{i+1} - x_i)f'(x_i)$$

from which we derive

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Newton’s method is very important: it is the basis of a wide range of optimization solvers in science and engineering.

Square Root

As a simple example, let’s derive the algorithm for the problem of finding square roots:

$$x = \sqrt{n} \iff x^2 = n \iff f(x) = x^2 - n = 0$$

If we substitute $f(x) = x^2 - n$ into the generic formula above, we can obtain the following update rule:

$$x_{i+1} = x_i - \frac{x_i^2 - n}{2x_i} = \frac{x_i + n/x_i}{2}$$

In practice we also want to stop it as soon as it is close enough to the right answer, which we can simply check after each iteration:

```
const double EPS = 1e-9;

double sqrt(double n) {
    double x = 1;
    while (abs(x * x - n) > eps)
        x = (x + n / x) / 2;
    return x;
}
```

The algorithm converges for many functions, although it does so reliably and provably only for a certain subset of them (e.g., convex functions). Another question is how fast the convergence is, if it occurs.

Rate of Convergence

Let's run a few iterations of Newton's method to find the square root of 2, starting with $x_0 = 1$, and check how many digits it got correct after each iteration:

Looking carefully, we can see that the number of accurate digits approximately doubles on each iteration. This fantastic convergence rate is not a coincidence.

To analyze convergence rate quantitatively, we need to consider a small relative error δ_i on the i -th iteration and determine how much smaller the error δ_{i+1} is on the next iteration:

$$|\delta_i| = \frac{|x_n - x|}{x}$$

We can express x_i as $x \cdot (1 + \delta_i)$. Plugging it into the Newton iteration formula and dividing both sides by x we get

$$1 + \delta_{i+1} = \frac{1}{2}(1 + \delta_i + \frac{1}{1 + \delta_i}) = \frac{1}{2}(1 + \delta_i + 1 - \delta_i + \delta_i^2 + o(\delta_i^2)) = 1 + \frac{\delta_i^2}{2} + o(\delta_i^2)$$

Here we have Taylor-expanded $(1 + \delta_i)^{-1}$ at 0, using the assumption that the error d_i is small (since the sequence converges, $d_i \ll 1$ for sufficiently large n).

Rearranging for δ_{i+1} , we obtain

$$\delta_{i+1} = \frac{\delta_i^2}{2} + o(\delta_i^2)$$

which means that the error roughly squares (and halves) on each iteration once we are close to the solution. Since the logarithm $(-\log_{10} \delta_i)$ is roughly the number of accurate significant digits in the answer x_i , squaring the relative error corresponds precisely to doubling the number of significant digits that we had observed.

This is known as *quadratic convergence*, and in fact, this is not limited to finding square roots. With detailed proof being left as an exercise to the reader, it can be shown that, in general

$$|\delta_{i+1}| = \frac{|f''(x_i)|}{2 \cdot |f'(x_n)|} \cdot \delta_i^2$$

which results in at least quadratic convergence under a few additional assumptions, namely $f'(x)$ not being equal to 0 and $f''(x)$ being continuous.

Further Reading

Introduction to numerical methods at MIT.

The inverse square root of a floating-point number $\frac{1}{\sqrt{x}}$ is used in calculating normalized vectors, which are in turn extensively used in various simulation scenarios such as computer graphics (e.g., to determine angles of incidence and reflection to simulate lighting).

$$\hat{v} = \frac{\vec{v}}{\sqrt{v_x^2 + v_y^2 + v_z^2}}$$

Calculating an inverse square root directly — by first calculating a square root and then dividing 1 by it — is extremely slow because both of these operations are slow even though they are implemented in hardware.

But there is a surprisingly good approximation algorithm that takes advantage of the way floating-point numbers are stored in memory. In fact, it is so good that it has been implemented in hardware, so the algorithm is no longer relevant by itself for software engineers, but we are nonetheless going to walk through it for its intrinsic beauty and great educational value.

Apart from the method itself, quite interesting is the history of its creation. It is attributed to a game studio *id Software* that used it in their iconic 1999 game *Quake III Arena*, although apparently, it got there by a chain of “I learned it from a guy who learned it from a guy” that seems to end on William Kahan (the same one that is responsible for IEEE 754 and Kahan summation algorithm).

It became popular in game developing community around 2005 when they released the source code of the game. Here is the relevant excerpt from it, including the comments:

```
float Q_rsqrt(float number) {
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = * ( long * ) &y;                                // evil floating point bit level hacking
    i = 0x5f3759df - ( i >> 1 );                      // what the fuck?
    y = * ( float * ) &i;
    y = y * ( threehalfs - ( x2 * y * y ) );          // 1st iteration
// y = y * ( threehalfs - ( x2 * y * y ) );          // 2nd iteration, this can be removed

    return y;
}
```

We will go through what it does step by step, but first, we need to take a small detour.

Approximate Logarithm

Before computers (or at least affordable calculators) became an everyday thing, people computed multiplication and related operations using logarithm tables — by looking up the logarithms of a and b , adding them, and then finding the inverse logarithm of the result.

$$a \times b = 10^{\log a + \log b} = \log^{-1}(\log a + \log b)$$

You can do the same trick when computing $\frac{1}{\sqrt{x}}$ using the identity:

$$\log \frac{1}{\sqrt{x}} = -\frac{1}{2} \log x$$

The fast inverse square root is based on this identity, and so it needs to calculate the logarithm of x very quickly. Turns out, it can be approximated by just reinterpreting a 32-bit `float` as an integer.

Recall that floating-point numbers sequentially store the sign bit (equal to zero for positive values, which is our case), exponent e_x and mantissa m_x , which corresponds to

$$x = 2^{e_x} \cdot (1 + m_x)$$

Its logarithm is therefore

$$\log_2 x = e_x + \log_2(1 + m_x)$$

Since $m_x \in [0, 1)$, the logarithm on the right-hand side can be approximated by

$$\log_2(1 + m_x) \approx m_x$$

The approximation is exact at both ends of the intervals, but to account for the average case we need to shift it by a small constant σ , therefore

$$\log_2 x = e_x + \log_2(1 + m_x) \approx e_x + m_x + \sigma$$

Now, having this approximation in mind and defining $L = 2^{23}$ (the number of mantissa bits in a `float`) and $B = 127$ (the exponent bias), when we reinterpret the bit-pattern of x as an integer I_x , we essentially get

$$\begin{aligned} I_x &= L \cdot (e_x + B + m_x) \\ &= L \cdot (e_x + m_x + \sigma + B - \sigma) \\ &\approx L \cdot \log_2(x) + L \cdot (B - \sigma) \end{aligned}$$

(Multiplying an integer by $L = 2^{23}$ is equivalent to left-shifting it by 23.)

When you tune σ to minimize the mean square error, this results in a surprisingly accurate approximation.

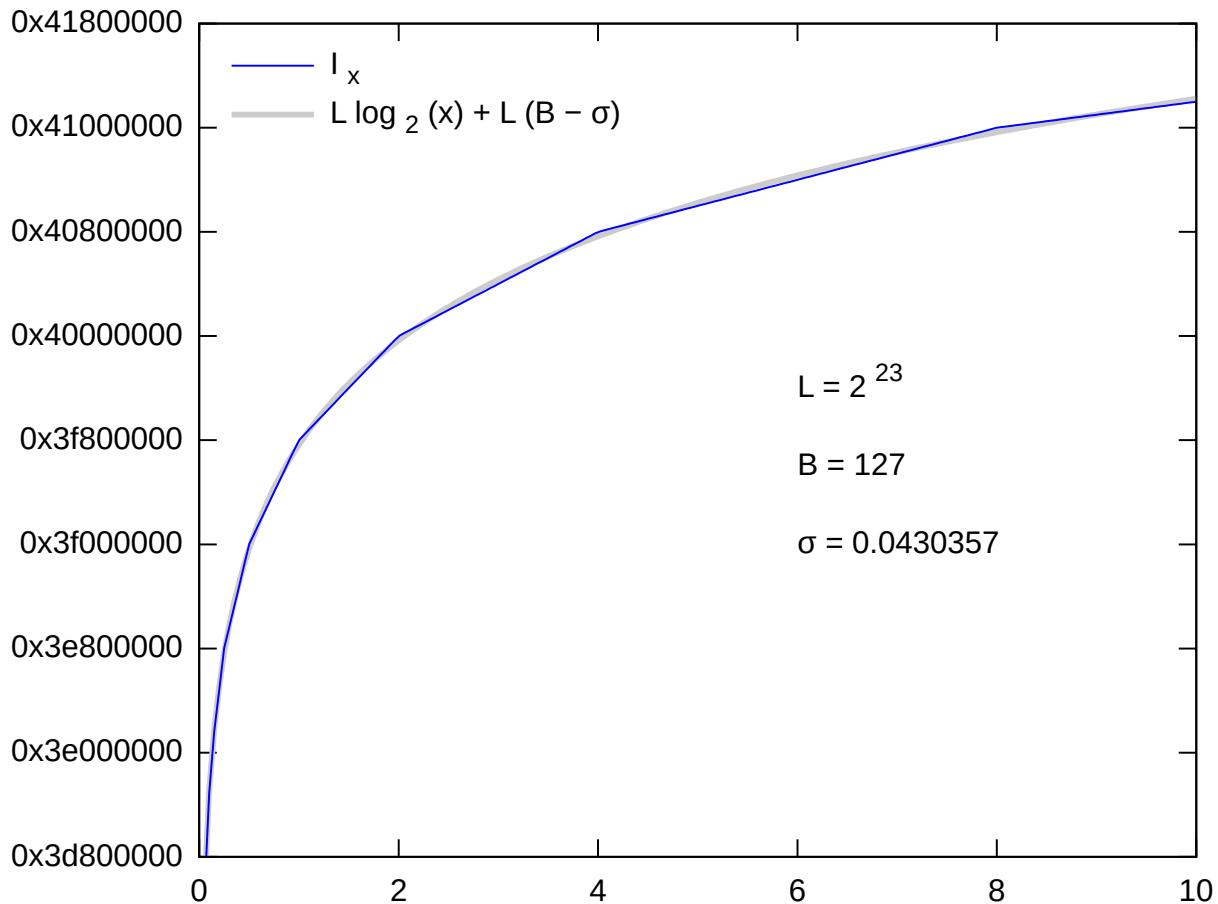


Figure 1: Reinterpreting a floating-point number x as an integer (blue) compared to its scaled and shifted logarithm (gray)

Now, expressing the logarithm from the approximation, we get

$$\log_2 x \approx \frac{I_x}{L} - (B - \sigma)$$

Cool. Now, where were we? Oh, yes, we wanted to calculate the inverse square root.

Approximating the Result

To calculate $y = \frac{1}{\sqrt{x}}$ using the identity $\log_2 y = -\frac{1}{2} \log_2 x$, we can plug it into our approximation formula and get

$$\frac{I_y}{L} - (B - \sigma) \approx -\frac{1}{2} \left(\frac{I_x}{L} - (B - \sigma) \right)$$

Solving for I_y :

$$I_y \approx \frac{3}{2}L(B - \sigma) - \frac{1}{2}I_x$$

It turns out, we don't even need to calculate the logarithm in the first place: the formula above is just a constant minus half the integer reinterpretation of x . It is written in the code as:

```
i = * ( long * ) &y;
i = 0x5f3759df - ( i >> 1 );
```

We reinterpret y as an integer in the first line, and then it plug into the formula on the second, the first term of which is the magic number $\frac{3}{2}L(B - \sigma) = 0x5F3759DF$, while the second is calculated with a binary shift instead of division.

Iterating with Newton's Method

What we have next is a couple hand-coded iterations of Newton's method with $f(y) = \frac{1}{y^2} - x$ and a very good initial value. Its update rule is

$$f'(y) = -\frac{2}{y^3} \implies y_{i+1} = y_i \left(\frac{3}{2} - \frac{x}{2} y_i^2 \right) = \frac{y_i(3 - xy_i^2)}{2}$$

which is written in the code as

```
x2 = number * 0.5F;
y = y * ( threehalfs - ( x2 * y * y ) );
```

The initial approximation is so good that just one iteration was enough for game development purposes. It falls within 99.8% of the correct answer after just the first iteration and can be reiterated further to improve accuracy — which is what is done in the hardware: the x86 instruction does a few of them and guarantees a relative error of no more than 1.5×2^{-12} .

Further Reading

Wikipedia article on fast inverse square root.

If you are reading this chapter sequentially from the beginning, you might be wondering: why would I introduce integer arithmetic after floating-point one? Isn't it supposed to be easier?

True: plain integer representations are simpler. But, counterintuitively, their simplicity allows for more possibilities for operations to be expressed in terms of others. And if floating-point representations are so unwieldy that most of their operations are implemented in hardware, efficiently manipulating integers requires much more creative use of the instruction set.

Binary Formats

Unsigned integers are just natural numbers written in binary:

$$\begin{aligned}5_{10} &= 101_2 = 4 + 1 \\42_{10} &= 101010_2 = 32 + 8 + 2 \\256_{10} &= 100000000_2 = 2^8\end{aligned}$$

When the result of an operation can't fit into the word size (e.g., is more or equal to 2^{32} for 32-bit unsigned integers), it *overflows* by leaving only the lowest 32 bits of the result. Similarly, if the result is a negative value, it *underflows* by adding it to 2^{32} , so that it always stays in the $[0, 2^{32})$ range.

This is equivalent to performing all operations modulo a power of two:

$$\begin{aligned}256 &\equiv 0 \pmod{2^8} \\2021 &\equiv 229 \pmod{2^8} \\-42 &\equiv 256 - 42 \equiv 214 \pmod{2^8}\end{aligned}$$

In either case, it raises a special flag which you can check, but usually when people explicitly use unsigned integers, they are expecting this behavior.

Signed Integers

Signed integers support storing negative values by dedicating the highest bit to represent the sign of the number, in a similar fashion as floating-point numbers do. This halves the range of representable non-negative numbers: the maximum possible 32-bit integer is now $(2^{31} - 1)$ and not $(2^{32} - 1)$. But the encoding of negative values is not quite the same as for floating-point numbers.

Computer engineers are even lazier than programmers — and this is not only motivated by the instinctive desire for simplification, but also by saving transistor space. This can be achieved by reusing circuitry that you already have for other operations, which is what they aimed for when designing the signed integer format:

- For an n -bit signed integer type, the encodings of all numbers in the $[0, 2^{n-1})$ range remain the same as their unsigned binary representations.
- All numbers in the $[-2^{n-1}, 0)$ range are encoded sequentially right after the “positive” range — that is, starting with (-2^{n-1}) that has code (2^{n-1}) and ending with (-1) that has code $(2^n - 1)$.

One way to look at this is that all negative numbers are just encoded as if they were subtracted from 2^n — an operation known as *two's complement*:

$$\begin{aligned}-x &= 2^{32} - x \\&= \bar{x} + 1\end{aligned}$$

Here \bar{x} represents bitwise negation, which can be also thought of as subtracting x from $(2^n - 1)$.

As an exercise, here are some facts about signed integers:

- All positive numbers and zero remain the same as their binary notation.
- All negative numbers have the highest bit set to one.
- There are more negative numbers than positive numbers (exactly by one — because of zero).
- For `int`, if you add 1 to $(2^{31} - 1)$, the result will be -2^{31} , represented as 10000000 (for exposition purposes, we will only write 8 bits instead of 32).
- Knowing a binary notation of a positive number x , you can get the binary notation of $-x$ as $\sim x + 1$.
- -1 is represented as $\sim 1 + 1 = 11111110 + 00000001 = 11111111$.
- -42 is represented as $\sim 42 + 1 = 11010101 + 00000001 = 11010110$.
- The number $-1 = 11111111$ is followed by $0 = -1 + 1 = 11111111 + 00000001 = 00000000$.

The main advantage of this encoding is that you don't have to do anything to convert unsigned integers to signed ones (except maybe check for overflow), and you can reuse the same circuitry for most operations, possibly only flipping the sign bit for comparisons and such.

That said, you need to be careful with signed integer overflows. Even though they almost always overflow the same way as unsigned integers, programming languages usually consider the possibility of overflow as undefined behavior. If you need to overflow integer variables, convert them to unsigned integers: it's free anyway.

Exercise. What is the only integer value for which `std::abs` produces a wrong result? What will this result be?

Integer Types

Integers come in different sizes, but all function roughly the same.

Bits	Bytes	Signed C type	Unsigned C type	Assembly
8	1	<code>signed char</code> ¹	<code>unsigned char</code>	<code>byte</code>
16	2	<code>short</code>	<code>unsigned short</code>	<code>word</code>
32	4	<code>int</code>	<code>unsigned int</code>	<code>dword</code>
64	8	<code>long long</code>	<code>unsigned long long</code>	<code>qword</code>

The bits of an integer are simply stored sequentially. The only ambiguity here is the order in which to store them — left to right or right to left — called *endianness*. Depending on the architecture, the format can be either:

- *Little-endian*, which lists *lower* bits first. For example, 42_{10} will be stored as 010101.
- *Big-endian*, which lists *higher* bits first. All previous examples in this article follow it.

This seems like an important architecture aspect, but in most cases, it doesn't make a difference: just pick one style and stick with it. But in some cases it does:

- Little-endian has the advantage that you can cast a value to a smaller type (e.g., `long long` to `int`) by just loading fewer bytes, which in most cases means doing nothing — thanks to *register aliasing*, `eax` refers to the first 4 bytes of `rax`, so conversion is essentially free. It is also easier to read values in a variety of type sizes — while on big-endian architectures, loading an `int` from a `long long` array would require shifting the pointer by 2 bytes.
- Big-endian has the advantage that higher bytes are loaded first, which in theory can make highest-to-lowest routines such as comparisons and printing faster. You can also perform certain checks such as finding out whether a number is negative by only loading its first byte.

Big-endian is also more “natural” — this is how we write binary numbers on paper — but the advantage of having faster type conversions outweighs it. For this reason, little-endian is used by default on most hardware, although some CPUs are “bi-endian” and can be configured to switch modes on demand.

128-bit Integers

Sometimes we need to multiply two 64-bit integers to get a 128-bit integer — usually to serve as a temporary value and be reduced modulo a 64-bit integer right away.

There are no 128-bit registers to hold the result of such multiplication, so the `mul` instruction, in addition to the normal `mul r r` form where it multiplies the values in registers and keeps the lower half of the result, has another `mul r` mode, where it multiplies whatever is stored in the `rax` register by its operand, and writes the result into two registers — the lower 64 bits of the result will go into `rax`, and the higher 64 bits go into `rdx`:

¹Note that `char`, `unsigned char`, and `signed char` are technically three distinct types. The C standard leaves it up to the implementation whether the plain `char` is signed or unsigned (on most compilers, it is signed).

```
; input: 64-bit integers a and b, stored in rsi and rdi
; output: 128-bit product a * b, stored in rax (lower 64-bit) and rdx (higher 64-bit)
mov    rax, rdi
mov    r8, rdx
imul   rsi
```

Some compilers have a separate type supporting this operation. In GCC and Clang it is available as `_int128`:

```
void prod(int64_t a, int64_t b, _int128 *c) {
    *c = a * (_int128) b;
}
```

Its typical use case is to immediately extract either the lower or the higher part of the multiplication and forget about it:

```
_int128_t x = 1;
int64_t hi = x >> 64;
int64_t lo = (int64_t) x; // will be just truncated
```

For all purposes other than multiplication, 128-bit integers are just bundled as two registers. This makes it too weird to have a full-fledged 128-bit type, so the support for it is limited, other than for basic arithmetic operations. For example:

```
_int128_t add(_int128_t a, _int128_t b) {
    return a + b;
}
```

is compiled into:

```
add:
    mov rax, rdi
    add rax, rdx      ; this sets the carry flag in case of an overflow
    adc rsi, rcx      ; +1 if the carry flag is set
    mov rdx, rsi
    ret
```

Other platforms provide similar mechanisms for dealing with longer-than-word multiplication. For example, Arm has `mulhi` and `mullo` instructions, returning lower and higher parts of the multiplication, and x86 SIMD extensions have similar 32-bit instructions.

Compared to other arithmetic operations, division works very poorly on x86 and computers in general. Both floating-point and integer division is notoriously hard to implement in hardware. The circuitry takes a lot of space in the ALU, the computation has a lot of stages, and as the result, `div` and its siblings routinely take 10-20 cycles to complete, with latency being slightly less on smaller data type sizes.

Division and Modulo in x86

Since nobody wants to duplicate all this mess for a separate modulo operation, the `div` instruction serves both purposes. To perform a 32-bit integer division, you need to put the dividend *specifically* in the `eax` register and call `div` with the divisor as its sole operand. After this, the quotient will be stored in `eax` and the remainder will be stored in `edx`.

The only caveat is that the dividend actually needs to be stored in *two* registers, `eax` and `edx`: this mechanism enables 64-by-32 or even 128-by-64 division, similar to how 128-bit multiplication works. When performing the usual 32-by-32 signed division, we need to sign-extend `eax` to 64 bits and store its higher part in `edx`:

```
div(int, int):
    mov eax, edi
    cdq
```

```
    idiv esi
    ret
```

For unsigned division, you can just set `edx` to zero so that it doesn't interfere:

```
div(unsigned, unsigned):
    mov eax, edi
    xor edx, edx
    div esi
    ret
```

An in both cases, in addition to the quotient in `eax`, you can also access the remainder as `edx`:

```
mod(unsigned, unsigned):
    mov eax, edi
    xor edx, edx
    div esi
    mov eax, edx
    ret
```

You can also divide 128-bit integer (stored in `rdx:rax`) by a 64-bit integer:

```
div(u128, u64):
; a = rdi + rsi, b = rdx
    mov rcx, rdx
    mov rax, rdi
    mov rdx, rsi
    div edx
    ret
```

The high part of the dividend should be less than the divisor, otherwise an overflow occurs. Because of this constraint, it is hard to get compilers to produce this code by themselves: if you divide a 128-bit integer type by a 64-bit integer, the compiler will bubble-wrap it with additional checks which may actually be unnecessary.

Division by Constants

Integer division is painfully slow, even when fully implemented in hardware, but it can be avoided in certain cases if the divisor is constant. A well-known example is the division by a power of two, which can be replaced by a one-cycle binary shift: the binary GCD algorithm is a delightful showcase of this technique.

In the general case, there are several clever tricks that replace division with multiplication at the cost of a bit of precomputation. All these tricks are based on the following idea. Consider the task of dividing one floating-point number x by another floating-point number y , when y is known in advance. What we can do is to calculate a constant

$$d \approx y^{-1}$$

and then, during runtime, we will calculate

$$x/y = x \cdot y^{-1} \approx x \cdot d$$

The result of $\frac{1}{y}$ will be at most ϵ off, and the multiplication $x \cdot d$ will only add another ϵ and therefore will be at most $2\epsilon + \epsilon^2 = O(\epsilon)$ off, which is tolerable for the floating-point case.

Barrett Reduction

How to generalize this trick for integers? Calculating `int d = 1 / y` doesn't seem to work, because it will just be zero. The best thing we can do is to express it as

$$d = \frac{m}{2^s}$$

and then find a “magic” number m and a binary shift s such that $x / y == (x * m) \gg s$ for all x within range.

$$\lfloor x/y \rfloor = \lfloor x \cdot y^{-1} \rfloor = \lfloor x \cdot d \rfloor = \lfloor x \cdot \frac{m}{2^s} \rfloor$$

It can be shown that such a pair always exists, and compilers actually perform an optimization like that by themselves. Every time they encounter a division by a constant, they replace it with a multiplication and a binary shift. Here is the generated assembly for dividing an `unsigned long long` by $(10^9 + 7)$:

```
; input (rdi): x
; output (rax): x mod (m=1e9+7)
mov    rax, rdi
movabs rdx, -8543223828751151131 ; load magic constant into a register
mul    rdx                      ; perform multiplication
mov    rax, rdx
shr    rax, 29                  ; binary shift of the result
```

This technique is called *Barrett reduction*, and it's called “reduction” because it is mostly used for modulo operations, which can be replaced with a single division, multiplication and subtraction by the virtue of this formula:

$$r = x - \lfloor x/y \rfloor \cdot y$$

This method requires some precomputation, including performing one actual division. Therefore, this is only beneficial when you perform not just one but a few divisions, all with the same constant divisor.

Why It Works

It is not very clear why such m and s always exist, let alone how to find them. But given a fixed s , intuition tells us that m should be as close to $2^s/y$ as possible for 2^s to cancel out. So there are two natural choices: $\lfloor 2^s/y \rfloor$ and $\lceil 2^s/y \rceil$. The first one doesn't work, because if you substitute

$$\left\lfloor \frac{x \cdot \lfloor 2^s/y \rfloor}{2^s} \right\rfloor$$

then for any integer $\frac{x}{y}$ where y is not even, the result will be strictly less than the truth. This only leaves the other case, $m = \lceil 2^s/y \rceil$. Now, let's try to derive the lower and upper bounds for the result of the computation:

$$\lfloor x/y \rfloor = \left\lfloor \frac{x \cdot m}{2^s} \right\rfloor = \left\lfloor \frac{x \cdot \lceil 2^s/y \rceil}{2^s} \right\rfloor$$

Let's start with the bounds for m :

$$2^s/y \leq \lceil 2^s/y \rceil < 2^s/y + 1$$

And now for the whole expression:

$$x/y - 1 < \left\lfloor \frac{x \cdot \lceil 2^s/y \rceil}{2^s} \right\rfloor < x/y + x/2^s$$

We can see that the result falls somewhere in a range of size $(1 + \frac{x}{2^s})$, and if this range always has exactly one integer for all possible x/y , then the algorithm is guaranteed to give the right answer. Turns out, we can always set s to be high enough to achieve it.

What will be the worst case here? How to pick x and y so that the $(x/y - 1, x/y + x/2^s)$ range contains two integers? We can see that integer ratios don't work because the left border is not included, and assuming $x/2^s < 1$, only x/y itself will be in the range. The worst case is actually the x/y that comes closest to 1 without exceeding it. For n -bit integers, that is the second-largest possible integer divided by the first-largest:

$$\begin{aligned} x &= 2^n - 2 \\ y &= 2^n - 1 \end{aligned}$$

In this case, the lower bound will be $(\frac{2^n-2}{2^n-1} - 1)$ and the upper bound will be $(\frac{2^n-2}{2^n-1} + \frac{2^n-2}{2^s})$. The left border is as close to a whole number as possible, and the size of the whole range is the second largest possible. And here is the punchline: if $s \geq n$, then the only integer contained in this range is 1, and so the algorithm will always return it.

Lemire Reduction

Barrett reduction is a bit complicated, and also generates a length instruction sequence for modulo because it is computed indirectly. There is a new (2019) method, which is simpler and actually faster for modulo in some cases. It doesn't have a conventional name yet, but I am going to refer to it as Lemire reduction.

Here is the main idea. Consider the floating-point representation of some integer fraction:

$$\frac{179}{6} = 11101.1101010101 \dots = 29\frac{5}{6} \approx 29.83$$

How can we "dissect" it to get the parts we need?

- To get the integer part (29), we can just floor or truncate it before the dot.
- To get the fractional part (), we can just take what is after the dots.
- To get the remainder (5), we can multiply the fractional part by the divisor.

Now, for 32-bit integers, we can set $s = 64$ and look at the computation that we do in the multiply-and-shift scheme:

$$\lfloor x/y \rfloor = \left\lfloor \frac{x \cdot m}{2^s} \right\rfloor = \left\lfloor \frac{x \cdot \lceil 2^s/y \rceil}{2^s} \right\rfloor$$

What we really do here is we multiply x by a floating-point constant ($x \cdot m$) and then truncate the result ($\lfloor \frac{x \cdot m}{2^s} \rfloor$).

What if we took not the highest bits but the lowest? This would correspond to the fractional part — and if we multiply it back by y and truncate the result, this will be exactly the remainder:

$$r = \left\lfloor \frac{(x \cdot \lceil 2^s/y \rceil \bmod 2^s) \cdot y}{2^s} \right\rfloor$$

This works perfectly because what we do here can be interpreted as just three chained floating-point multiplications with the total relative error of $O(\epsilon)$. Since $\epsilon = O(\frac{1}{2^s})$ and $s = 2n$, the error will always be less than one, and hence the result will be exact.

```

uint32_t y;

uint64_t m = uint64_t(-1) / y + 1; // ceil(2^64 / y)

uint32_t mod(uint32_t x) {
    uint64_t lowbits = m * x;
    return ((__uint128_t) lowbits * y) >> 64;
}

uint32_t div(uint32_t x) {
    return ((__uint128_t) m * x) >> 64;
}

```

We can also check divisibility of x by y with just one multiplication using the fact that the remainder of division is zero if and only if the fractional part (the lower 64 bits of $m \cdot x$) does not exceed m (otherwise, it would become a nonzero number when multiplied back by y and right-shifted by 64).

```

bool is_divisible(uint32_t x) {
    return m * x < m;
}

```

The only downside of this method is that it needs integer types four times the original size to perform the multiplication, while other reduction methods can work with just the double.

There is also a way to compute 64x64 modulo by carefully manipulating the halves of intermediate results; the implementation is left as an exercise to the reader.

Further Reading

Check out libdivide and GMP for more general implementations of optimized integer division.

It is also worth reading Hacker's Delight, which has a whole chapter dedicated to integer division.

This article is largely based on Bit Twiddling Hacks by Sean Eron Anderson. Some methods were added, and some removed due to being solved by hardware. Most of these are already optimized by compilers.

A lot of it became obsolete on architectures where `cmov` is available.

This also serves as an exercise.

Basic Operations

`>>`

Note that arithmetic shift shifts in 1 for negative numbers and in 0 for positive ones. (although it can be implementation-defined?)

Left or right-shifting negative numbers invokes undefined behavior in C/C++.

`<<`

`rol` instruction that does “rotate left”

`__builtin_popcount` `popcnt` Returns the number of 1-bits in x .

`__builtin_parity` Returns the *parity* of x (that is, the number of 1-bits in x modulo 2).

This is presumably for error detection.

`__builtin_clrsb` Returns the number of leading redundant sign bits in x , i.e., the number of bits following the most significant bit that are identical to it. There are no special cases for 0 or other values.

`__builtin_ffs` Returns one plus the index of the least significant 1-bit of x , or if x is zero, returns zero.

`__builtin_clz` Returns the number of leading 0-bits in `x`, starting at the most significant bit position. If `x` is 0, the result is undefined

`__builtin_ctz` Returns the number of trailing 0-bits in `x`, starting at the least significant bit position. If `x` is 0, the result is undefined

`ctz, cls -> __lg`

Recipes

Sign of an integer

`(x < 0) or x >> 31`

Check if two integers have the same sign

`x ^ z < 0`

Absolute value of an integer

Extract the sign bit: `int mask = x >> 31`. This will be 1 for negative numbers and 0 for positive.

XOR it with the initial number: `x ^ mask` (which equates to either adding or subtracting 1 depending on the sign).

Subtract mask from the result of step 2: `(x ^ mask) - mask`

Alternatively, you can use `(v + mask) ^ mask` which does the same but in reverse.

Get last 1-bit

`x & -x`

Remove the last 1-bit of an integer

`x & (x - 1)`

Checking for power of two

`(x & (x - 1)) == 0`

Note that 0 will be considered a power of 2 as well.

Reversing bits

Clang has `__builtin_bitreverse{8,16,32,64}`

```
int reverseBits(int x)
{
    unsigned int s = sizeof(x) * 8;
    T mask = ~T(0);
    while ((s >= 1) > 0)
    {
        mask ^= mask << s;
        x = ((x >> s) & mask) | ((x << s) & ~mask);
    }
    return x;
}
```

Swapping numbers with xor

You've probably heard of this one.

```
a ^= b;  
b ^= a;  
a ^= b;
```

This isn't how it is performed under the hood. There is a separate xchng instruction.

Modulus a power of two

If $m = (1 \ll k)$, then $x \% m$ is the same as $x \& (m - 1)$.

Masks

Masking operations.

Brute forcing

You can either go recursive (which is obviously slow). You can also go branchless.

Knapsack problem has a brute force solution $O(2^n)$.

```
int ans = 0;  
for (int mask = 0; mask < (1 << n); mask++) {  
    int s = 0;  
    for (int i = 0; i < n; i++)  
        if (mask >> i & 1)  
            s += a[i];  
    if (s <= C)  
        ans = max(ans, s);  
}
```

Subsets of all subsets

```
for (int submask = mask; submask != 0; submask = (submask - 1) & mask) {  
    // ...  
}
```

It turns out that the total will be 3^n . Each bit on each iteration can be in one of three states: not in m , not in s yet, in both s and m . As there are n bits in total, there can be at most 3^n different combinations.

...

Computers usually store time as the number of seconds that have passed since the 1st of January, 1970 — the start of the “Unix era” — and use these timestamps in all computations that have to do with time.

We humans also keep track of time relative to some point in the past, which usually has a political or religious significance. For example, at the moment of writing, approximately 63882260594 seconds have passed since 1 AD — 6th century Eastern Roman monks’ best estimate of the day Jesus Christ was born.

But unlike computers, we do not always need *all* that information. Depending on the task at hand, the relevant part may be that it’s 2 pm right now, and it’s time to go to dinner; or that it’s Thursday, and so Subway’s sub of the day is an Italian BMT. Instead of the whole timestamp, we use its *remainder* containing just the information we need: it is much easier to deal with 1- or 2-digit numbers than 11-digit ones.

Problem. Today is Thursday. What day of the week will be exactly in a year?

If we enumerate each day of the week, starting with Monday, from 0 to 6 inclusive, Thursday gets number 3. To find out what day it is going to be in a year from now, we need to add 365 to it and then reduce modulo 7. Conveniently, $365 \bmod 7 = 1$, so we know that it will be Friday unless it is a leap year (in which case it will be Saturday).

Residues

Definition. Two integers a and b are said to be *congruent* modulo m if m divides their difference:

$$m \mid (a - b) \iff a \equiv b \pmod{m}$$

For example, the 42nd day of the year is the same weekday as the 161st since $(161 - 42) = 119 = 17 \times 7$.

Congruence modulo m is an equivalence relation that splits all integers into equivalence classes called *residues*. Each residue class modulo m may be represented by any one of its members — although we commonly use the smallest nonnegative integer of that class (equal to the remainder $x \bmod m$ for all nonnegative x).

Modular arithmetic studies these sets of residues, which are fundamental for number theory.

Problem. Our “week” now consists of m days, and our year consists of a days (no leap years). How many distinct days of the week there will be among one, two, three and so on whole years from now?

For simplicity, assume that today is Monday, so that the initial day number d_0 is zero, and after each year, it changes to

$$d_{k+1} = (d_k + a) \bmod m$$

After k years, it will be

$$d_k = k \cdot a \bmod m$$

Since there are only m days in a week, at some point, it will be Monday again, and the sequence of day numbers is going to cycle. The number of distinct days is the length of this cycle, so we need to find the smallest k such that

$$k \cdot a \equiv 0 \pmod{m}$$

First of all, if $a \equiv 0$, it will be eternal Monday. Now, assuming the non-trivial case of $a \not\equiv 0$:

- For a seven-day week, $m = 7$ is prime. There is no k smaller than m such that $k \cdot a$ is divisible by m because m can not be decomposed in such a product by the definition of primality. So, if m is prime, we will cycle through all of m weekdays.

- If m is not prime, but a is *coprime* with it (that is, a and m do not have common divisors), then the answer is still m for the same reason: the divisors of a do not help in zeroing out the product any faster.
- If a and m share some divisors, then it is only possible to get residues that are also divisible by them. For example, if the week is $m = 10$ days long, and the year has $a = 42$ or any other even number of days, then we will cycle through all even day numbers, and if the number of days is a multiple of 5, then we will only oscillate between 0 and 5. Otherwise, we will go through all the 10 remainders.

Therefore, in general, the answer is $\frac{m}{\gcd(a, m)}$, where $\gcd(a, m)$ is the greatest common divisor of a and m .

Fermat's Theorem

Now, consider what happens if, instead of adding a number a , we repeatedly multiply by it, writing out a sequence of

$$d_n = a^n \bmod m$$

Again, since there is a finite number of residues, there is going to be a cycle. But what will its length be? Turns out, if m is prime, it will span all $(m - 1)$ non-zero residues.

Theorem. For any a and a prime p :

$$a^p \equiv a \pmod{p}$$

Proof. Let $P(x_1, x_2, \dots, x_n) = \frac{k}{\prod(x_i!)} = \frac{k}{x_1! x_2! \dots x_n!}$ be the *multinomial coefficient*: the number of times the element $a_1^{x_1} a_2^{x_2} \dots a_n^{x_n}$ appears after the expansion of $(a_1 + a_2 + \dots + a_n)^k$. Then:

$$\begin{aligned} a^p &= (\underbrace{1 + 1 + \dots + 1 + 1}_{a \text{ times}})^p \\ &= \sum_{x_1+x_2+\dots+x_a=p} P(x_1, x_2, \dots, x_a) && \text{(by definition)} \\ &= \sum_{x_1+x_2+\dots+x_a=p} \frac{p!}{x_1! x_2! \dots x_a!} && \text{(which terms will not be divisible by } p\text{?)} \\ &\equiv P(p, 0, \dots, 0) + \dots + P(0, 0, \dots, p) && \text{(everything else will be canceled)} \\ &= a \end{aligned}$$

Note that this is only true for prime p . We can use this fact to test whether a given number is prime faster than by factoring it: we can pick a number a at random, calculate $a^p \bmod p$, and check whether it is equal to a or not.

This is called *Fermat primality test*, and it is probabilistic — only returning either “no” or “maybe” — since it may be that a^p just happened to be equal to a despite p being composite, in which case you need to repeat the test with a different random a until you are satisfied with the false positive probability.

Primality tests are commonly used to generate large primes (for cryptographic purposes). There are roughly $\frac{n}{\ln n}$ primes among the first n numbers (a fact that we are not going to prove), and they are distributed more or less evenly. One can just pick a random number from the required range, perform a primality check, and repeat until a prime is found, performing $O(\ln n)$ trials on average.

An extremely bad input to the Fermat test is the Carmichael numbers, which are composite numbers n that satisfy $a^{n-1} \equiv 1 \pmod{n}$ for all relatively prime a . But these are rare, and the chance of randomly bumping into it is low.

Modular Division

Implementing most “normal” arithmetic operations with residues is straightforward. You only need to take care of integer overflows and remember to take modulo:

```
c = (a + b) % m;
c = (a - b + m) % m;
c = a * b % m;
```

But there is an issue with division: we can’t just bluntly divide two residues. For example, $\frac{8}{2} = 4$, but

$$\frac{8 \bmod 5}{2 \bmod 5} = \frac{3}{2} \neq 4$$

To perform modular division, we need to find an element that “acts” like the reciprocal $\frac{1}{a} = a^{-1}$ and multiply by it. This element is called a *modular multiplicative inverse*, and Fermat’s theorem can help us find it when the modulo p is a prime. When we divide its equivalence twice by a , we get:

$$a^p \equiv a \implies a^{p-1} \equiv 1 \implies a^{p-2} \equiv a^{-1}$$

Therefore, a^{p-2} is like a^{-1} for the purposes of multiplication, which is what we need from a modular inverse of a .

In modular arithmetic (and computational algebra in general), you often need to raise a number to the n -th power — to do modular division, perform primality tests, or compute some combinatorial values — and you usually want to spend fewer than $\Theta(n)$ operations calculating it.

Binary exponentiation, also known as *exponentiation by squaring*, is a method that allows for computation of the n -th power using $O(\log n)$ multiplications, relying on the following observation:

$$\begin{aligned} a^{2k} &= (a^k)^2 \\ a^{2k+1} &= (a^k)^2 \cdot a \end{aligned}$$

To compute a^n , we can recursively compute $a^{\lfloor n/2 \rfloor}$, square it, and then optionally multiply by a if n is odd, corresponding to the following recurrence:

$$a^n = f(a, n) = \begin{cases} 1, & n = 0 \\ f(a, \frac{n}{2})^2, & 2 \mid n \\ f(a, n-1) \cdot a, & 2 \nmid n \end{cases}$$

Since n is at least halved every two recursive transitions, the depth of this recurrence and the total number of multiplications will be at most $O(\log n)$.

Recursive Implementation

As we already have a recurrence, it is natural to implement the algorithm as a case matching recursive function:

```
const int M = 1e9 + 7; // modulo
typedef unsigned long long u64;

u64 binpow(u64 a, u64 n) {
    if (n == 0)
        return 1;
    if (n % 2 == 1)
```

```

        return binpow(a, n - 1) * a % M;
    else {
        u64 b = binpow(a, n / 2);
        return b * b % M;
    }
}

```

In our benchmark, we use $n = m - 2$ so that we compute the multiplicative inverse of a modulo m :

```

u64 inverse(u64 a) {
    return binpow(a, M - 2);
}

```

We use $m = 10^9 + 7$, which is a modulo value commonly used in competitive programming to calculate checksums in combinatorial problems — because it is prime (allowing inverse via binary exponentiation), sufficiently large, not overflowing `int` in addition, not overflowing `long long` in multiplication, and easy to type as `1e9 + 7`.

Since we use it as compile-time constant in the code, the compiler can optimize the modulo by replacing it with multiplication (even if it is not a compile-time constant, it is still cheaper to compute the magic constants by hand once and use them for fast reduction).

The execution path — and consequently the running time — depends on the value of n . For this particular n , the baseline implementation takes around 330ns per call. As recursion introduces some overhead, it makes sense to unroll the implementation into an iterative procedure.

Iterative Implementation

The result of a^n can be represented as the product of a to some powers of two — those that correspond to 1s in the binary representation of n . For example, if $n = 42 = 32 + 8 + 2$, then

$$a^{42} = a^{32+8+2} = a^{32} \cdot a^8 \cdot a^2$$

To calculate this product, we can iterate over the bits of n maintaining two variables: the value of a^{2^k} and the current product after considering k lowest bits of n . On each step, we multiply the current product by a^{2^k} if the k -th bit of n is set, and, in either case, square a^k to get $a^{2^k \cdot 2} = a^{2^{k+1}}$ that will be used on the next iteration.

```

u64 binpow(u64 a, u64 n) {
    u64 r = 1;

    while (n) {
        if (n & 1)
            r = res * a % M;
        a = a * a % M;
        n >>= 1;
    }

    return r;
}

```

The iterative implementation takes about 180ns per call. The heavy calculations are the same; the improvement mainly comes from the reduced dependency chain: $a = a * a \% M$ needs to finish before the loop can proceed, and it can now execute concurrently with $r = res * a \% M$.

The performance also benefits from n being a constant, making all branches predictable and letting the scheduler know what needs to be executed in advance. The compiler, however, does not take advantage of

it and does not unroll the `while(n) n >= 1` loop. We can rewrite it as a `for` loop that performs constant 30 iterations:

```
u64 inverse(u64 a) {
    u64 r = 1;

    #pragma GCC unroll(30)
    for (int l = 0; l < 30; l++) {
        if ((M - 2) >> l & 1)
            r = r * a % M;
        a = a * a % M;
    }

    return r;
}
```

This forces the compiler to generate only the instructions we need, shaving off another 10ns and making the total running time ~170ns.

Note that the performance depends not only on the binary length of n , but also on the number of binary 1s. If n is 2^{30} , it takes around 20ns less as we don't have to perform any off-path multiplications.

Fermat's theorem allows us to calculate modular multiplicative inverses through binary exponentiation in $O(\log n)$ operations, but it only works with prime modula. There is a generalization of it, Euler's theorem, stating that if m and a are coprime, then

$$a^{\phi(m)} \equiv 1 \pmod{m}$$

where $\phi(m)$ is Euler's totient function defined as the number of positive integers $x < m$ that are coprime with m . In the special case when m is a prime, then all the $m - 1$ residues are coprime and $\phi(m) = m - 1$, yielding the Fermat's theorem.

This lets us calculate the inverse of a as $a^{\phi(m)-1}$ if we know $\phi(m)$, but in turn, calculating it is not so fast: you usually need to obtain the factorization of m to do it. There is a more general method that works by modifying the Euclidean algorithm.

Algorithm

Extended Euclidean algorithm, apart from finding $g = \gcd(a, b)$, also finds integers x and y such that

$$a \cdot x + b \cdot y = g$$

which solves the problem of finding modular inverse if we substitute b with m and g with 1:

$$a^{-1} \cdot a + k \cdot m = 1$$

Note that, if a is not coprime with m , there is no solution since no integer combination of a and m can yield anything that is not a multiple of their greatest common divisor.

The algorithm is also recursive: it calculates the coefficients x' and y' for $\gcd(b, a \bmod b)$ and restores the solution for the original number pair. If we have a solution (x', y') for the pair $(b, a \bmod b)$

$$b \cdot x' + (a \bmod b) \cdot y' = g$$

then, to get the solution for the initial input, we can rewrite the expression $(a \bmod b)$ as $(a - \lfloor \frac{a}{b} \rfloor \cdot b)$ and substitute it into the aforementioned equation:

$$b \cdot x' + (a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b) \cdot y' = g$$

Now we rearrange the terms grouping by a and b to get

$$a \cdot y' + b \cdot \underbrace{(x' - \left\lfloor \frac{a}{b} \right\rfloor \cdot y')}_{y} = g$$

Comparing it with the initial expression, we infer that we can just use coefficients of a and b for the initial x and y .

Implementation

We implement the algorithm as a recursive function. Since its output is not one but three integers, we pass the coefficients to it by reference:

```
int gcd(int a, int b, int &x, int &y) {
    if (a == 0) {
        x = 0;
        y = 1;
        return b;
    }
    int x1, y1;
    int d = gcd(b % a, a, x1, y1);
    x = y1 - (b / a) * x1;
    y = x1;
    return d;
}
```

To calculate the inverse, we simply pass a and m and return the x coefficient the algorithm finds. Since we pass two positive numbers, one of the coefficient will be positive and the other one is negative (which one depends on whether the number of iterations is odd or even), so we need to optionally check if x is negative and add m to get a correct residue:

```
int inverse(int a) {
    int x, y;
    gcd(a, M, x, y);
    if (x < 0)
        x += M;
    return x;
}
```

It works in $\sim 160\text{ns} - 10\text{ns}$ faster than inverting numbers with binary exponentiation. To optimize it further, we can similarly turn it iterative — which takes 135ns:

```
int inverse(int a) {
    int b = M, x = 1, y = 0;
    while (a != 1) {
        y -= b / a * x;
        b %= a;
        swap(a, b);
        swap(x, y);
    }
    return x < 0 ? x + M : x;
}
```

Note that, unlike binary exponentiation, the running time depends on the value of a . For example, for this particular value of m ($10^9 + 7$), the worst input happens to be 564400443, for which the algorithm performs 37 iterations and takes 250ns.

Exercise. Try to adapt the same technique for the binary GCD (it won't give performance speedup though unless you are better than me at optimization).

Unsurprisingly, a large fraction of computation in modular arithmetic is often spent on calculating the modulo operation, which is as slow as general integer division and typically takes 15-20 cycles, depending on the operand size.

The best way to deal this nuisance is to avoid modulo operation altogether, delaying or replacing it with predication, which can be done, for example, when calculating modular sums:

```
const int M = 1e9 + 7;

// input: array of n integers in the [0, M) range
// output: sum modulo M
int slow_sum(int *a, int n) {
    int s = 0;
    for (int i = 0; i < n; i++)
        s = (s + a[i]) % M;
    return s;
}

int fast_sum(int *a, int n) {
    int s = 0;
    for (int i = 0; i < n; i++) {
        s += a[i]; // s < 2 * M
        s = (s >= M ? s - M : s); // will be replaced with cmov
    }
    return s;
}

int faster_sum(int *a, int n) {
    long long s = 0; // 64-bit integer to handle overflow
    for (int i = 0; i < n; i++)
        s += a[i]; // will be vectorized
    return s % M;
}
```

However, sometimes you only have a chain of modular multiplications, and there is no good way to eel out of computing the remainder of the division — other than with the integer division tricks requiring a constant modulo and some precomputation.

But there is another technique designed specifically for modular arithmetic, called *Montgomery multiplication*.

Montgomery Space

Montgomery multiplication works by first transforming the multipliers into *Montgomery space*, where modular multiplication can be performed cheaply, and then transforming them back when their actual values are needed. Unlike general integer division methods, Montgomery multiplication is not efficient for performing just one modular reduction and only becomes worthwhile when there is a chain of modular operations.

The space is defined by the modulo n and a positive integer $r \geq n$ coprime to n . The algorithm involves modulo and division by r , so in practice, r is chosen to be 2^{32} or 2^{64} , so that these operations can be done with a right-shift and a bitwise AND respectively.

Definition. The *representative* \bar{x} of a number x in the Montgomery space is defined as

$$\bar{x} = x \cdot r \bmod n$$

Computing this transformation involves a multiplication and a modulo — an expensive operation that we wanted to optimize away in the first place — which is why we only use this method when the overhead of transforming numbers to and from the Montgomery space is worth it and not for general modular multiplication.

Inside the Montgomery space, addition, subtraction, and checking for equality is performed as usual:

$$x \cdot r + y \cdot r \equiv (x + y) \cdot r \bmod n$$

However, this is not the case for multiplication. Denoting multiplication in the Montgomery space as $*$ and the “normal” multiplication as \cdot , we expect the result to be:

$$\bar{x} * \bar{y} = \overline{x \cdot y} = (x \cdot y) \cdot r \bmod n$$

But the normal multiplication in the Montgomery space yields:

$$\bar{x} \cdot \bar{y} = (x \cdot y) \cdot r \cdot r \bmod n$$

Therefore, the multiplication in the Montgomery space is defined as

$$\bar{x} * \bar{y} = \bar{x} \cdot \bar{y} \cdot r^{-1} \bmod n$$

This means that, after we normally multiply two numbers in the Montgomery space, we need to *reduce* the result by multiplying it by r^{-1} and taking the modulo — and there is an efficient way to do this particular operation.

Montgomery reduction

Assume that $r = 2^{32}$, the modulo n is 32-bit, and the number x we need to reduce is 64-bit (the product of two 32-bit numbers). Our goal is to calculate $y = x \cdot r^{-1} \bmod n$.

Since r is coprime with n , we know that there are two numbers r^{-1} and n' in the $[0, n)$ range such that

$$r \cdot r^{-1} + n \cdot n' = 1$$

and both r^{-1} and n' can be computed, e.g., using the extended Euclidean algorithm.

Using this identity, we can express $r \cdot r^{-1}$ as $(1 - n \cdot n')$ and write $x \cdot r^{-1}$ as

$$\begin{aligned} x \cdot r^{-1} &= x \cdot r \cdot r^{-1} / r \\ &= x \cdot (1 - n \cdot n') / r \\ &= (x - x \cdot n \cdot n') / r \\ &\equiv (x - x \cdot n \cdot n' + k \cdot r \cdot n) / r \quad (\bmod n) \quad (\text{for any integer } k) \\ &\equiv (x - (x \cdot n' - k \cdot r) \cdot n) / r \quad (\bmod n) \end{aligned}$$

Now, if we choose k to be $\lfloor x \cdot n' / r \rfloor$ (the upper 64 bits of the $x \cdot n'$ product), it will cancel out, and $(k \cdot r - x \cdot n')$ will simply be equal to $x \cdot n' \bmod r$ (the lower 32 bits of $x \cdot n'$), implying:

$$x \cdot r^{-1} \equiv (x - x \cdot n' \bmod r \cdot n) / r$$

The algorithm itself just evaluates this formula, performing two multiplications to calculate $q = x \cdot n' \bmod r$ and $m = q \cdot n$, and then subtracts it from x and right-shifts the result to divide it by r .

The only remaining thing to handle is that the result may not be in the $[0, n)$ range; but since

$$x < n \cdot n < r \cdot n \implies x/r < n$$

and

$$m = q \cdot n < r \cdot n \implies m/r < n$$

it is guaranteed that

$$-n < (x - m)/r < n$$

Therefore, we can simply check if the result is negative and in that case, add n to it, giving the following algorithm:

```
typedef __uint32_t u32;
typedef __uint64_t u64;

const u32 n = 1e9 + 7, nr = inverse(n, 1ull << 32);

u32 reduce(u64 x) {
    u32 q = u32(x) * nr;           // q = x * n' mod r
    u64 m = (u64) q * n;          // m = q * n
    u32 y = (x - m) >> 32;        // y = (x - m) / r
    return x < m ? y + n : y;    // if y < 0, add n to make it be in the [0, n) range
}
```

This last check is relatively cheap, but it is still on the critical path. If we are fine with the result being in the $[0, 2 \cdot n - 2]$ range instead of $[0, n)$, we can remove it and add n to the result unconditionally:

```
u32 reduce(u64 x) {
    u32 q = u32(x) * nr;
    u64 m = (u64) q * n;
    u32 y = (x - m) >> 32;
    return y + n
}
```

We can also move the `>> 32` operation one step earlier in the computation graph and compute $\lfloor x/r \rfloor - \lfloor m/r \rfloor$ instead of $(x - m)/r$. This is correct because the lower 32 bits of x and m are equal anyway since

$$m = x \cdot n' \cdot n \equiv x \pmod{r}$$

But why would we voluntarily choose to perform two right-shifts instead of just one? This is beneficial because for `((u64) q * n) >> 32` we need to do a 32-by-32 multiplication and take the upper 32 bits of the result (which the x86 `mul` instruction already writes in a separate register, so it doesn't cost anything), and the other right-shift `x >> 32` is not on the critical path.

```

u32 reduce(u64 x) {
    u32 q = u32(x) * nr;
    u32 m = ((u64) q * n) >> 32;
    return (x >> 32) + n - m;
}

```

One of the main advantages of Montgomery multiplication over other modular reduction methods is that it doesn't require very large data types: it only needs a $r \times r$ multiplication that extracts the lower and higher r bits of the result, which has special support on most hardware also makes it easily generalizable to SIMD and larger data types:

```

typedef __uint128_t u128;

u64 reduce(u128 x) const {
    u64 q = u64(x) * nr;
    u64 m = ((u128) q * n) >> 64;
    return (x >> 64) + n - m;
}

```

Note that a 128-by-64 modulo is not possible with general integer division tricks: the compiler falls back to calling a slow long arithmetic library function to support it.

Faster Inverse and Transform

Montgomery multiplication itself is fast, but it requires some precomputation:

- inverting n modulo r to compute n' ,
- transforming a number *to* the Montgomery space,
- transforming a number *from* the Montgomery space.

The last operation is already efficiently performed with the `reduce` procedure we just implemented, but the first two can be slightly optimized.

Computing the inverse $n' = n^{-1} \bmod r$ can be done faster than with the extended Euclidean algorithm by taking advantage of the fact that r is a power of two and using the following identity:

$$a \cdot x \equiv 1 \pmod{2^k} \implies a \cdot x \cdot (2 - a \cdot x) \equiv 1 \pmod{2^{2k}}$$

Proof:

$$\begin{aligned} a \cdot x \cdot (2 - a \cdot x) &= 2 \cdot a \cdot x - (a \cdot x)^2 \\ &= 2 \cdot (1 + m \cdot 2^k) - (1 + m \cdot 2^k)^2 \\ &= 2 + 2 \cdot m \cdot 2^k - 1 - 2 \cdot m \cdot 2^k - m^2 \cdot 2^{2k} \\ &= 1 - m^2 \cdot 2^{2k} \\ &\equiv 1 \pmod{2^{2k}}. \end{aligned}$$

We can start with $x = 1$ as the inverse of a modulo 2^1 and apply this identity exactly $\log_2 r$ times, each time doubling the number of bits in the inverse — somewhat reminiscent of the Newton's method.

Transforming a number into the Montgomery space can be done by multiplying it by r and computing modulo the usual way, but we can also take advantage of this relation:

$$\bar{x} = x \cdot r \bmod n = x * r^2$$

Transforming a number into the space is just a multiplication by r^2 . Therefore, we can precompute $r^2 \bmod n$ and perform a multiplication and reduction instead — which may or may not be actually faster because multiplying a number by $r = 2^k$ can be implemented with a left-shift, while multiplication by $r^2 \bmod n$ can not.

Complete Implementation

It is convenient to wrap everything into a single `constexpr` structure:

```
struct Montgomery {
    u32 n, nr;

    constexpr Montgomery(u32 n) : n(n), nr(1) {
        // log(2^32) = 5
        for (int i = 0; i < 5; i++)
            nr *= 2 - n * nr;
    }

    u32 reduce(u64 x) const {
        u32 q = u32(x) * nr;
        u32 m = ((u64) q * n) >> 32;
        return (x >> 32) + n - m;
        // returns a number in the [0, 2 * n - 2] range
        // (add a "x < n ? x : x - n" type of check if you need a proper modulo)
    }

    u32 multiply(u32 x, u32 y) const {
        return reduce((u64) x * y);
    }

    u32 transform(u32 x) const {
        return (u64(x) << 32) % n;
        // can also be implemented as multiply(x, r^2 mod n)
    }
};
```

To test its performance, we can plug Montgomery multiplication into the binary exponentiation:

```
constexpr Montgomery space(M);

int inverse(int _a) {
    u64 a = space.transform(_a);
    u64 r = space.transform(1);

    #pragma GCC unroll(30)
    for (int l = 0; l < 30; l++) {
        if ((M - 2) >> l & 1)
            r = space.multiply(r, a);
        a = space.multiply(a, a);
    }

    return space.reduce(r);
}
```

While vanilla binary exponentiation with a compiler-generated fast modulo trick requires ~170ns per `inverse` call, this implementation takes ~166ns, going down to ~158ns we omit `transform` and `reduce` (a reasonable

use case is for `inverse` to be used as a subprocedure in a bigger modular computation). This is a small improvement, but Montgomery multiplication becomes much more advantageous for SIMD applications and larger data types.

Exercise. Implement efficient *modular* matrix multiplication.

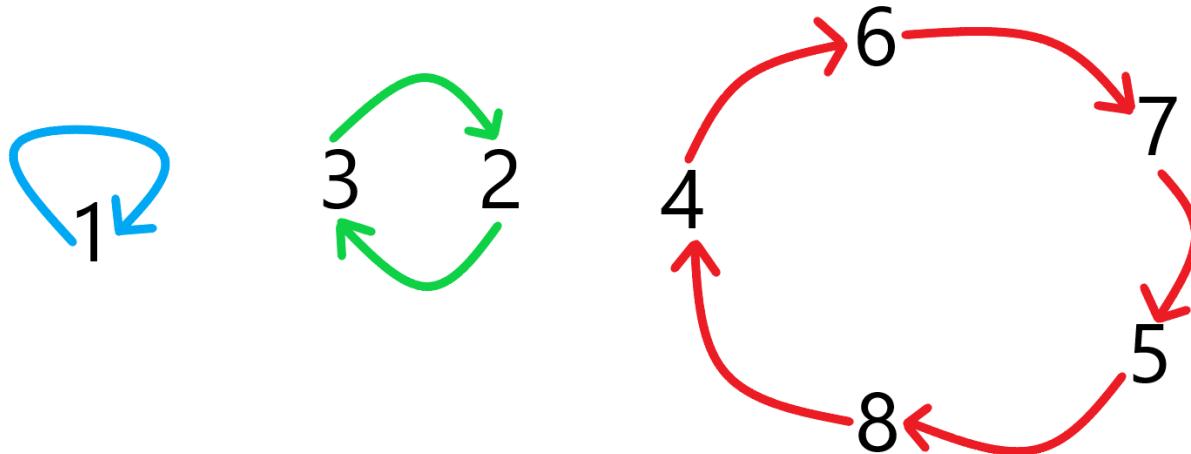
There are other mathematical objects that behave the same way as remainders modulo a prime numbers. Based on what properties these sets of objects retain they are called *groups*, *rings* and *fields*.

Permutations

We can define product of permutations as application of one permutation to another.

This operation is associative, so we can use binary exponentiation to compute n -th power in $O(n \log n)$ time.

$$(1, 3, 2, 6, 8, 7, 5, 4)$$



In general, this is called *permutation group*, and *groups* are sets of elements that have an operation defined on them.

Roots of unity

Complex numbers are numbers of form $a + bi$, where a and b are real numbers and i is called imaginary one: it is the number for which $i^2 = -1$.

Complex numbers are useful in algebra to work with roots of negative numbers, as i in some sense is equal to $\sqrt{-1}$. Similar to negative numbers, they don't really exist in the real world, but only in the minds of mathematicians.

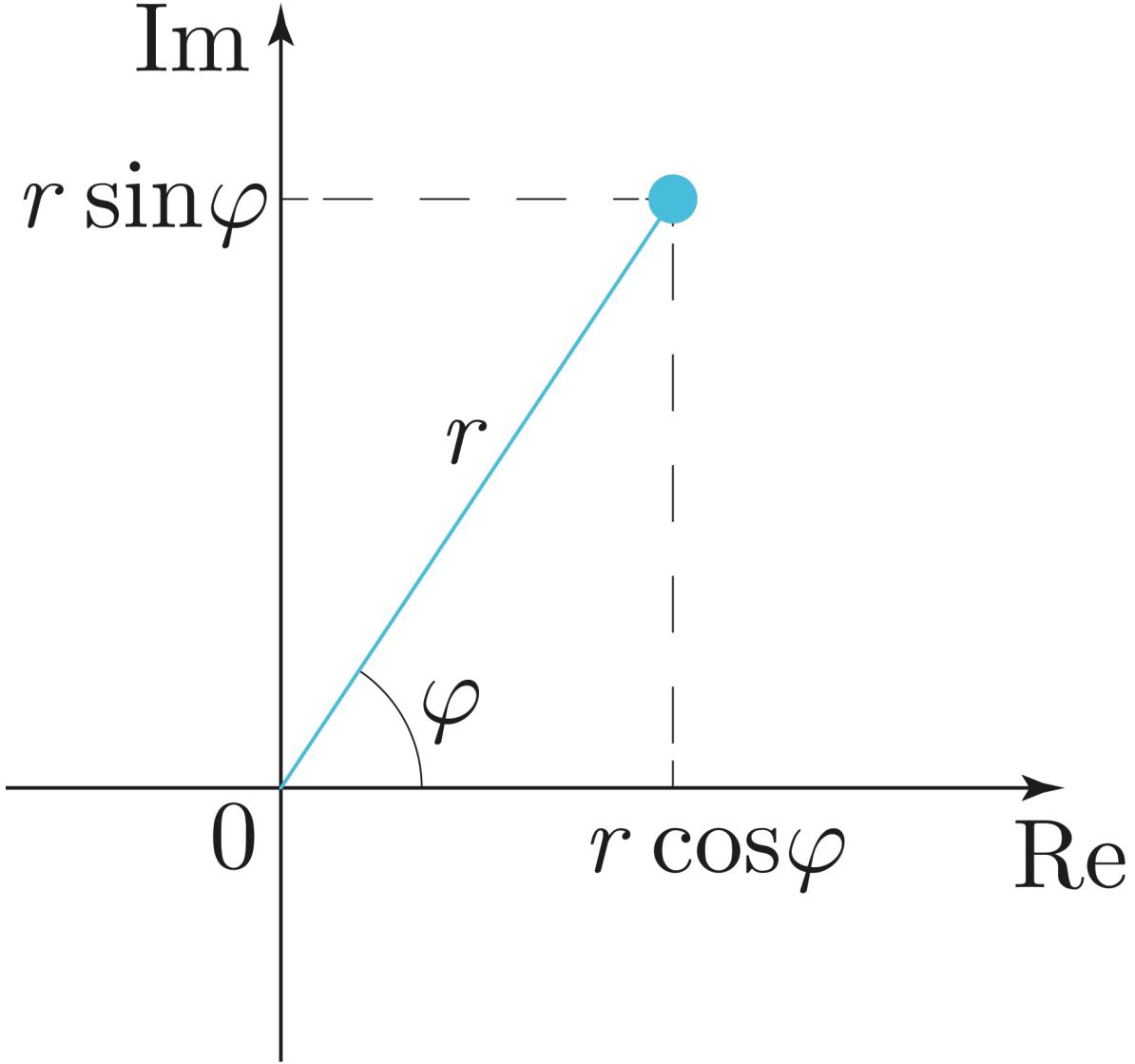
It is most convenient to think about complex numbers geometrically.

Modulus of a complex number as the real number $r = \sqrt{a^2 + b^2}$. Geometrically, this is the length of vector (a, b) .

Argument of a complex number is the real number $\phi \in (-\pi, \pi]$, for which $\tan \phi = \frac{b}{a}$. Geometrically, this is the angle between vectors $(a, 0)$ and (a, b) .

This way we can represent a complex number using polar coordinates:

$$a + bi = r \cdot (\cos \phi + i \sin \phi)$$



The reason this representation is useful is because if we need to multiply two complex numbers it can be shown with high school trigonometry that instead of doing binomial expansion we just need to multiply their moduli and add their arguments:

$$\begin{aligned} (a + bi) \cdot (c + di) &= r_1 \cdot (\cos \phi_1 + i \sin \phi_1) \cdot r_2 \cdot (\cos \phi_2 + i \sin \phi_2) \\ &= r_1 \cdot r_2 \cdot (\cos \phi_1 \cos \phi_2 - \sin \phi_1 \sin \phi_2 + i \cos \phi_1 \sin \phi_2 + i \cos \phi_2 \sin \phi_1) \\ &= r_1 \cdot r_2 \cdot (\cos(\phi_1 + \phi_2) + i \sin(\phi_1 + \phi_2)) \end{aligned}$$

Now, let's **define** Euler's number e as the number for which

$$e^{i\phi} = \cos \phi + i \sin \phi$$

that is, let's just introduce such notation for $(\cos \phi + i \sin \phi)$ without thinking too much about what raising a number to a complex exponent really means. Geometrically, all such points will live on a unitary circle.

Such notation is useful, because we can treat $e^{i\phi}$ as a normal exponent. If we want to multiply two numbers complex on a unit circle with arguments a and b , then we just write

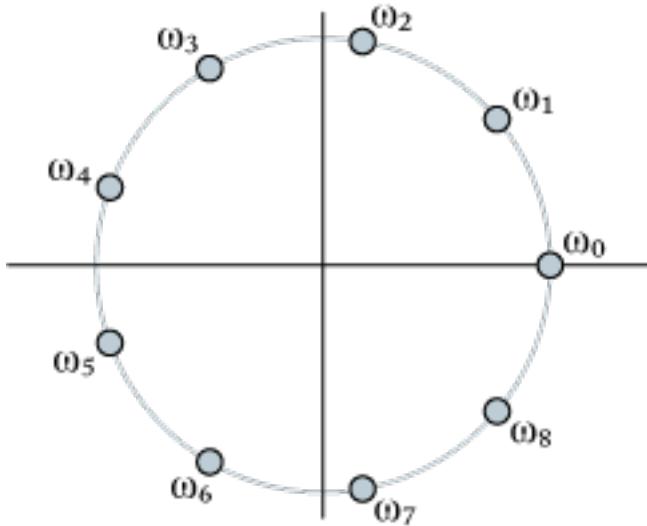
$$(\cos a + i \sin a) \cdot (\cos b + i \sin b) = e^{i(a+b)}$$

Now, what does all that have to do with finite rings you may ask. The thing is that when we multiply complex numbers that live on a unit circle they wrap around the same way integer remainders do. There are still infinitely many of them, so the resemblance is not clear, but consider this: how many n -th roots of one there can be? These are the points that when raised to n -th power arrive at 1.

It turns out that there are exactly n of them, and they will be equally spaced. Precisely, these will be numbers of form

$$w_k = e^{i\tau \frac{k}{n}}$$

where τ means 2π (a modern notation).



The first root w_1 (the second root, to be more precise, because 1 is also a root) is called generating root of unity. Raising it to 1st, 2nd and so on powers will generate a sequence of roots that will cycle after n iterations.

$$w_n = e^{i\tau \frac{n}{n}} = e^{i\tau} = e^{i \cdot 0} = w_0 = 1$$

Everything not on unit circle will have moduli too large or too low. And everything with argument not divided by w_1 will have argument too off.

Structures like these are called *rings*, or, more precisely, *finite rings*.

Galois Fields

Remainders are *finite rings*. What makes remainders modulo a prime different is that there will always exist an inverse operation for multiplication, in which case it is called a *finite fields*. But prime numbers are not the only set sizes that can house a finite field.

The problem with having finite fields of prime numbers is that they are wasteful when dealing with computers, that have power-of-two data.

It turns out that you can also construct fields of size p^k , for any prime p . This is particularly useful for computers, because you can set $p = 2$ and $k = 8$, and one byte will perfectly fit for members of this set.

For non-prime fields, you move away from the idea of using integers, and instead you encode information as a special type of polynomial. First, you pick an *irreducible polynomial*. For example, for GF(8), that $p(x) = x^3 + x + 1$ is the one: it can't be factored into a product.

You define a member of the set by the coefficient of the polynomial. Since every coefficient is either 0 or 1, there will be exactly 256 of them.

You define addition as xor-ing its elements, and multiplication as multiplying the polynomials and reducing them by that irreducible polynomial (which can be done in a manner similar to GCD).

Xor-ing is okay computationally, but instead of doing this weird multiplication we can just exploit the fact that there are not that many of them and just precompute a lookup table of 256 one-byte entries.

This sounds complicated, but it is all done to have efficient implementations:

```
char log[256], ilog[256];

char add(char x, char y) {
    return x ^ y;
}

char sub(char x, char y) {
    return x ^ y;
}

char mul(char x, char y) {
    return ilog[log[x] + log[y]];
}

char div(char x, char y) {
    return ilog[log[x] - log[y]];
}
```

This exact field is a foundation of many cryptographic and data compression. In fact, when you loaded this page, your computer did a transform of everything that was communicated into this form.

...

Hash Functions

Hash functions take an object (a short message, document, image, or basically any binary sequence) and transform it into a fixed-length seemingly random sequence, but in a deterministic way.

Hash function is any function that is:

- Computed fast — at least in linear time, that is.
- Has a limited image — say, 64-bit values.
- “Deterministically-random:” if it takes n different values, then the probability of collision of two random hashes is $\frac{1}{n}$ and can't be predicted well without knowing the hash function.

One good test is that can't create a collision in any better time than by birthday paradox. Square root of the hash space.

- Checksums.
- Hash tables. Memoisation.
- Locality-sensitive hashing

Cryptographic

SHA-1, MD5

Passwords. There are special hash functions that have a parameter associated with it.

Salt and pepper. Dictionary attack. You can precompute a large table of hashes and just join it with a large database.

Non-cryptographic

Identity Polynomial Chess

MurMurHash CRC32

Signatures

We now know enough to implement digital signatures.

JWT. The principle is the same as with certificates and other documents. Instead of carrying. It may or may not be encrypted.

JWT is a new piece of technology. You can essentially store whatever you need about the user on their device. Now, instead of going to a central database and checking for permissions on every request. You can also add “valid for 5 more minutes” to the token. Nobody usually even encrypts it. The simplest way you can do it is to add a fixed random string to the beginning of the message and calculating a hash of it.

Blockchain

Proof-of-work. There are smarter blockchains that actually do something useful for proof of work.

This article is an overview of all topics related to modern internet-era cryptography.

Cryptography (from greek *kryptos* “secret” and *graphein* “to write”) studies the techniques of secure communication in the presence of third (from greek *graphein*) parties called adversaries.

Asymmetric Cryptography

Cryptography is mostly based on the notion that some easy to do in one way and very hard to do in reverse. One of them is integer factoring: it is easy to pick two large primes p and q and multiply them to produce number $n = pq$, but it is very hard to do it in reverse.

1. Pick two primes p and q , and exponent e coprime with $\phi(n) = (p-1) \cdot (q-1)$. Picking small exponents makes more computational sense but is slightly less secure. Usually people don't care and pick $e = 3$.
2. Calculate multiplicative inverse of e modulo $\phi(n)$, that is $d \cdot e \equiv 1 \pmod{\phi(n)}$
3. Set (e, n) as the public key and send it to everyone who wants to communicate with us. (d, n) is the information necessary to decode messages.

Now, actual encoding is to calculate $c = m^e$ and decoding is to calculate $c^d = m^{ed} = m$.

To calculate d and restore the message, the attacker would need to repeat step 2, that is, to restore the inverse of e . You would need to use extended Euclid's algorithm, but you need to feed the value of $\phi(n)$ into it, which is hard to do unless you know the factorization of n .

When doing actual communication, people first exchange their public keys (in any, possibly unsecure way) and then use it to encrypt messages.

This is what web browsers do when establishing connection “https.” You can also do it by hand with GPG.

Man-in-the-middle

There is an issue when establishing initial communication that the attacker could replace it and control the communication.

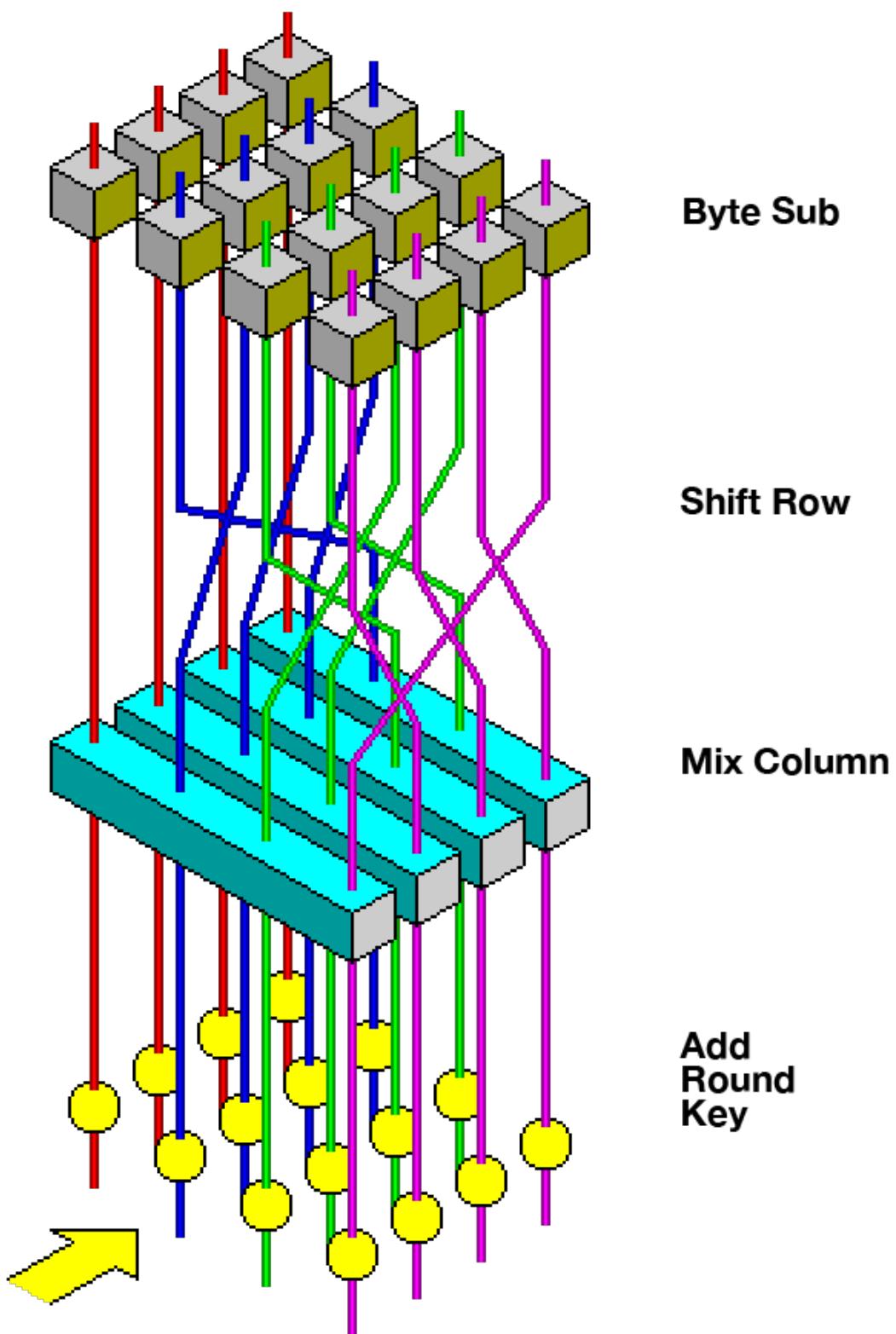
Between your browser and a bank. “Hey this is a message from a bank.”

Trust networks. E.g., everyone can trust Google or whoever makes the device or operating system.

Symmetric Cryptography

Disadvantage of RSA is that it is hard to calculate. You really don't want to execute $512 \text{ modulo } 2^{512}$ multiplications to decode each 64 bytes of a video that you stream from YouTube.

Such schemes exist but they require a secure key. This is where RSA comes in, but after that they switch to AES.



confusion and diffusion.

Confusion means that each binary digit (bit) of the ciphertext should depend on several parts of the key, obscuring the connections between the two

Diffusion means that if we change a single bit of the plaintext, then (statistically) half of the bits in the ciphertext should change, and similarly, if we change one bit of the ciphertext, then approximately one half of the plaintext bits should change.[3] Since a bit can have only two states, when they are all re-evaluated and changed from one seemingly random position to another, half of the bits will have changed state.

The property of confusion hides the relationship between the ciphertext and the key. The idea of diffusion is to hide the relationship between the ciphertext and the plain text.

AES is implemented in hardware.

Perfect Security

Almost all practical cryptography relies on certain assumptions. They inevitably leak at least some information about the message.

The design of cryptographic functions often relies on hardware implementation efficiency rather than math.

One-time pads can't be cracked. If someone from Russia wants to communicate with someone from the United States, they send a representative. They send a suitcase full of entropy.

Relying on not RSA not being feasibly solvable.

AES is approved by NSA for use with 196 and 256-bit keys. It's not that 128-bit keys are insecure, it's just that

Quantum computing is one of them.

. . .
— 2019
— 4088459.

RSA

...

Cryptographic protocols

In this article, we have learned how to:

- Transfer two messages given that you can have some random information between people.
- Transfer two messages without any common information.
- Sign a message verifying its truthfulness.
- Performing any action, although in a very computationally inefficient and non-private way.

If you feel curious, here are a few more things to think about:

- How to compare two numbers without leaking anything? For example, you may want to compare salaries with your colleagues without being fired.
- How to shuffle a card deck and play a round of poker without revealing anything?
- Private stable matching or other multi-party computations?
- Private set intersection?
- How to make it all secure when there is not one adversary, but 49%?

Cryptography is a lot more fun.

Timing information, power consumption, electromagnetic leaks or even sound can provide an extra source of information

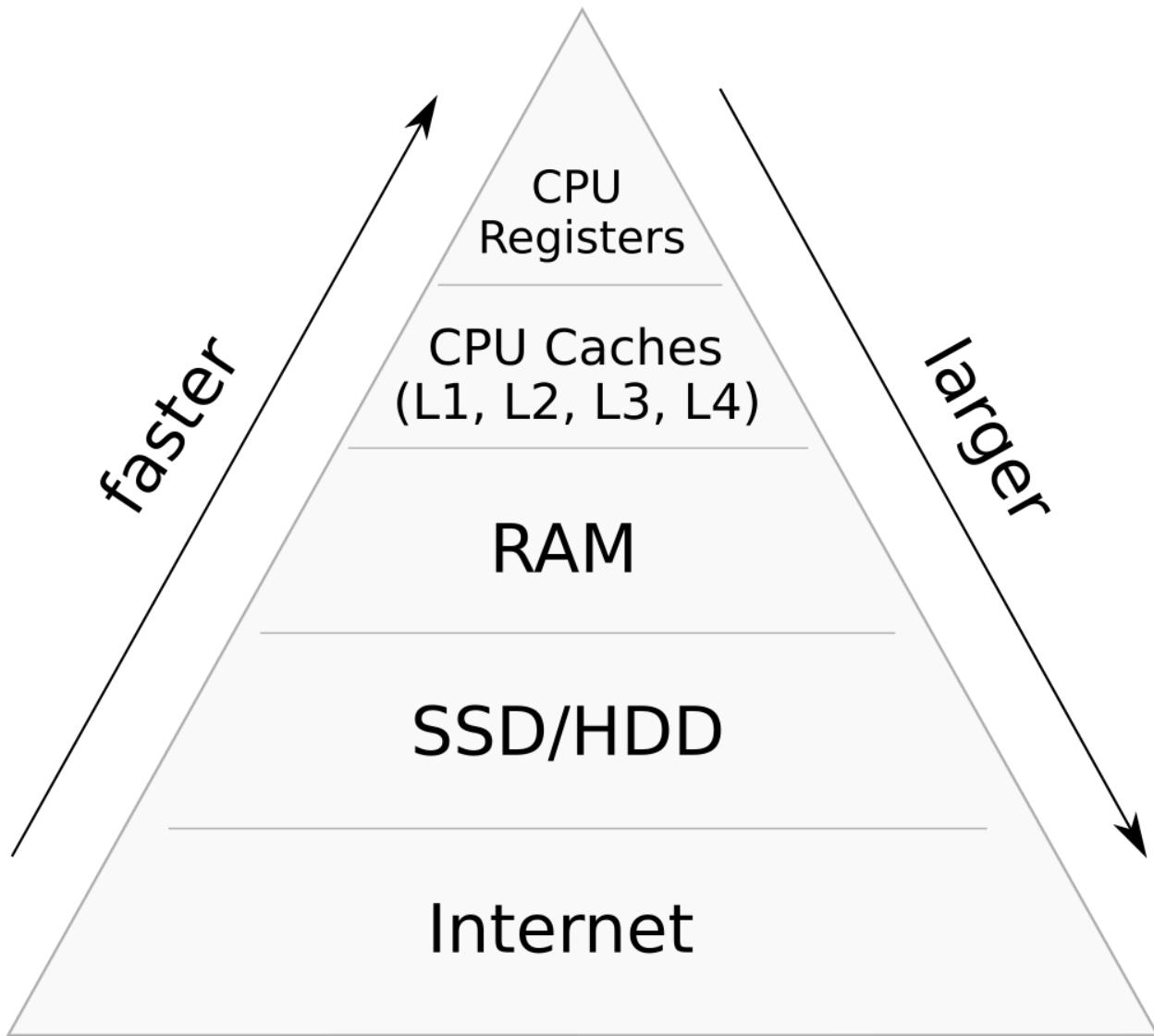
We can use iterated hash function for what it's worth.

Linear congruent generator period of LCG

External Memory

Memory Hierarchy

Modern computer memory is highly hierarchical. It consists of multiple *cache layers* of varying speed and size, where *higher* levels typically store most frequently accessed data from *lower* levels to reduce latency: each next level is usually an order of magnitude faster, but also smaller and/or more expensive.



Abstractly, various memory devices can be described as modules that have a certain storage capacity M and can read or write data in blocks of B (not individual bytes!), taking a fixed time to complete.

From this perspective, each type of memory has a few important characteristics:

- *total size M* ;
- *block size B* ;
- *latency*, that is, how much time it takes to fetch one byte;
- *bandwidth*, which may be higher than just the block size times latency, meaning that I/O operations can “overlap”;
- *cost* in the amortized sense, including the price for the chip, its energy requirements, maintenance, and so on.

Here is an approximate comparison table for commodity hardware in 2021:

Type	M	B	Latency	Bandwidth	\$/GB/mo ¹
L1	10K	64B	2ns	80G/s	-
L2	100K	64B	5ns	40G/s	-
L3	1M/core	64B	20ns	20G/s	-
RAM	GBs	64B	100ns	10G/s	1.5
SSD	TBs	4K	0.1ms	5G/s	0.17
HDD	TBs	-	10ms	1G/s	0.04
S3	∞	-	150ms	∞	0.02 ²

In reality, there are many specifics about each type of memory, which we will now go through.

Volatile Memory

Everything up to the RAM level is called *volatile memory* because it does not persist data in case of a power shortage and other disasters. It is fast, which is why it is used to store temporary data while the computer is powered.

From fastest to slowest:

- **CPU registers**, which are the zero-time access data cells CPU uses to store all its intermediate values, can also be thought of as a memory type. There is only a limited number of them (e.g., just 16 “general purpose” ones), and in some cases, you may want to use all of them for performance reasons.
- **CPU caches**. Modern CPUs have multiple layers of cache (L1, L2, often L3, and rarely even L4). The lowest layer is shared between cores and is usually scaled with their number (e.g., a 10-core CPU should have around 10M of L3 cache).
- **Random access memory**, which is the first scalable type of memory: nowadays you can rent machines with half a terabyte of RAM on the public clouds. This is the one where most of your working data is supposed to be stored.

The CPU cache system has an important concept of a *cache line*, which is the basic unit of data transfer between the CPU and the RAM. The size of a cache line is 64 bytes on most architectures, meaning that all main memory is divided into blocks of 64 bytes, and whenever you request (read or write) a single byte, you are also fetching all its 63 cache line neighbors whether you want them or not.

Caching on the CPU level happens automatically based on the last access times of cache lines. When accessed, the contents of a cache line are emplaced onto the lowest cache layer and then gradually evicted to higher levels unless accessed again in time. The programmer can't control this process explicitly, but it is worthwhile to study how it works in detail, which we will do in the next chapter.

Non-Volatile Memory

While the data cells in CPU caches and the RAM only gently store just a few electrons (that periodically leak and need to be periodically refreshed), the data cells in *non-volatile memory* types store hundreds of them. This lets the data persist for prolonged periods of time without power but comes at the cost of performance and durability — because when you have more electrons, you also have more opportunities for them to collide with silicon atoms.

There are many ways to store data in a persistent way, but these are the main ones from a programmer's perspective:

- **Solid state drives**. These have relatively low latency on the order of 0.1ms (10^5 ns), but they also have a high cost, amplified by the fact that they have limited lifespans as each cell can only be written

¹Pricing information is taken from the Google Cloud Platform.

²Cloud storage typically has multiple tiers, becoming progressively cheaper if you access the data less frequently.

to a limited number of times. This is what mobile devices and most laptops use because they are compact and have no moving parts.

- **Hard disk drives** are unusual because they are actually rotating physical disks with a read/write head attached to them. To read a memory location, you need to wait until the disk rotates to the right position and then very precisely move the head to it. This results in some very weird access patterns where reading one byte randomly may take the same time as reading the next 1MB of data — which is usually on the order of milliseconds. Since this is the only part of a computer, except for the cooling system, that has mechanically moving parts, hard disks break quite often (with the average lifespan of ~3 years for a data center HDD).
- **Network-attached storage**, which is the practice of using other networked devices to store data on them. There are two distinctive types. The first one is the Network File System (NFS), which is a protocol for mounting the file system of another computer over the network. The other is API-based distributed storage systems, most famously Amazon S3, that are backed by a fleet of storage-optimized machines of a public cloud, typically using cheap HDDs or some more exotic storage types internally. While NFS can sometimes work even faster than HDD if it is located in the same data center, object storage in the public cloud usually has latencies of 50-100ms. They are typically highly distributed and replicated for better availability.

Since SDD/HDD are noticeably slower than RAM, everything on or below this level is usually called *external memory*.

Unlike the CPU caches, external memory can be explicitly controlled. This is useful in many cases, but most programmers just want to abstract away from it and use it as an extension of the main memory, and operating systems have the capability to do so by the means of virtual memory.

Virtual Memory

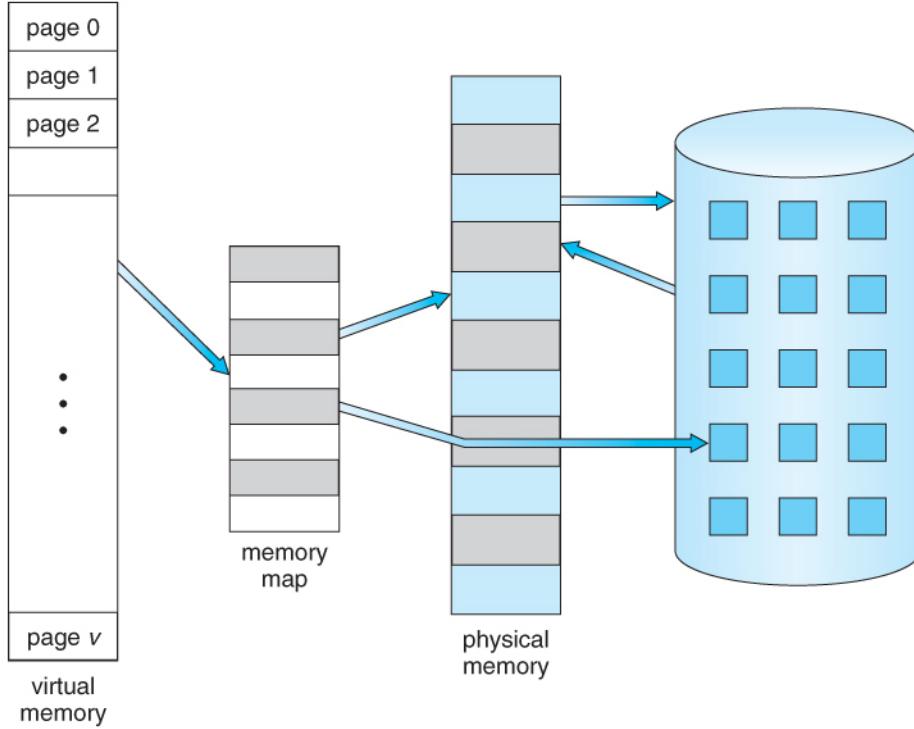
Early operating systems gave every process the freedom of reading and modifying any memory region they want, including those allocated for other processes. While this keeps things simple, it also poses some problems:

- What if one of the processes is buggy or outright malicious? How do we prevent it from modifying the memory allocated for other processes while still keeping inter-process communication through memory possible?
- How do we deal with memory fragmentation? Say, we have 4MB of memory, process A allocates the first 1MB for itself, then process B claims the next 2MB, then A terminates and releases its memory, and then process C comes and asks for a contiguous 2MB region — and can't get it because we only have two separate 1MB slices. Restarting process B or somehow stopping it and shifting all its data and pointers by one megabyte doesn't seem like a good solution.
- How do we access non-RAM memory types? How do we plug a flash drive and read a specific file from it?

These problems are not that critical for some specialized computer systems such as GPUs, where you typically solve just one task at a time and have full control over the computation, but they are absolutely essential for modern multitasking operating systems — and they solve all these problems with a technique called *virtual memory*.

Memory Paging

Virtual memory gives each process the impression that it fully controls a contiguous region of memory, which in reality may be mapped to multiple smaller blocks of the physical memory — which includes both the main memory (RAM) and external memory (HDD, SSD).



To achieve this, the memory address space is divided into *pages* (typically 4KB in size), which are the base units of memory that the programs can request from the operating system. The memory system maintains a special hardware data structure called the *page table*, which contains the mappings of virtual page addresses to the physical ones. When a process accesses data using its virtual memory address, the memory system calculates its page number (by right-shifting it by 12 if $4096 = 2^{12}$ is the page size), looks up in the page table that its physical address is, and forwards the read or write request to where that data is actually stored.

Since the address translation needs to be done for each memory request, and the number of memory pages itself may be large (e.g., 16G RAM / 4K page size = 4M pages), address translation poses a difficult problem in itself. One way to speed it up is to use a special cache for the page table itself called *translation lookaside buffer* (TLB), and the other is to increase the page size so that the total number of memory pages is made smaller at the cost of reduced granularity.

Mapping External Memory

The mechanism of virtual memory also allows using external memory types quite transparently. Modern operating systems support memory mapping, which lets you open a file and use its contents as if they were in the main memory:

```
// open a file containing 1024 random integers for reading and writing
int fd = open("input.bin", O_RDWR);
// map it into memory      size allow reads and writes write changes back to the file
int* data = (int*) mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
// sort it like if it was a normal integer array
std::sort(data, data + 1024);
// changes are eventually propagated to the file
```

Here we map a 4K file, which can fit entirely on just a single memory page, but when we open larger files, its reads will be done lazily when we request a certain page, and its writes will be buffered and committed to the file system when the operating decides to (usually on the program termination or when the system runs out of RAM).

A technique that has the same operating principle, but the reverse intention is the *swap file*, which lets the operating system automatically use parts of an SSD or an HDD as an extension of the main memory when there is not enough real RAM. This lets the systems that run out of memory just experience a terrible slowdown instead of crashing.

This seamless integration of the main and external memory essentially turns RAM into an “L4 cache” for the external memory, which is a convenient way to think about it from the algorithm design perspective.

External Memory Model

To reason about the performance of memory-bound algorithms, we need to develop a cost model that is more sensitive to expensive block I/O operations but is not too rigorous to still be useful.

Cache-Aware Model

In the standard RAM model, we ignore the fact that primitive operations take unequal time to complete. Most importantly, it does not differentiate between operations on different types of memory, equating a read from RAM taking $\sim 50\text{ns}$ in real-time with a read from HDD taking $\sim 5\text{ms}$, or about a 10^5 times as much.

Similar in spirit, in the *external memory model*, we simply ignore every operation that is not an I/O operation. More specifically, we consider one level of cache hierarchy and assume the following about the hardware and the problem:

- The size of the dataset is N , and it is all stored in *external* memory, which we can read and write in blocks of B elements in a unit time (reading a whole block and just one element takes the same time).
- We can store M elements in *internal* memory, meaning that we can store up to $\lfloor \frac{M}{B} \rfloor$ blocks.
- We only care about I/O operations: any computations done in-between the reads and the writes are free.
- We additionally assume $N \gg M \gg B$.

In this model, we measure the performance of an algorithm in terms of its high-level *I/O operations*, or *IOPS* — that is, the total number of blocks read or written to external memory during execution.

We will mostly focus on the case where the internal memory is RAM and the external memory is SSD or HDD, although the underlying analysis techniques that we will develop are applicable to any layer in the cache hierarchy. Under these settings, reasonable block size B is about 1MB, internal memory size M is usually a few gigabytes, and N is up to a few terabytes.

Array Scan

As a simple example, when we calculate the sum of an array by iterating through it one element at a time, we implicitly load it by chunks of $O(B)$ elements and, in terms of the external memory model, process these chunks one by one:

$$\underbrace{a_1, a_2, a_3}_{B_1}, \underbrace{a_4, a_5, a_6}_{B_2}, \dots \underbrace{a_{n-3}, a_{n-2}, a_{n-1}}_{B_{m-1}}$$

Thus, in the external memory model, the complexity of summation and other linear array scans is

$$SCAN(N) \stackrel{\text{def}}{=} O\left(\left\lceil \frac{N}{B} \right\rceil\right) \text{ IOPS}$$

You can implement external array scan explicitly like this:

```
FILE *input = fopen("input.bin", "rb");

const int M = 1024;
```

```

int buffer[M], sum = 0;

// while the file is not fully processed
while (true) {
    // read up to M of 4-byte elements from the input stream
    int n = fread(buffer, 4, M, input);
    // ^ the number of elements that were actually read

    // if we can't read any more elements, finish
    if (n == 0)
        break;

    // sum elements in-memory
    for (int i = 0; i < n; i++)
        sum += buffer[i];
}

fclose(input);
printf("%d\n", sum);

```

Note that, in most cases, operating systems do this buffering automatically. Even when the data is just redirected to the standard input from a normal file, the operating system buffers its stream and reads it in blocks of $\sim 4\text{KB}$ (by default).

External Sorting

Now, let's try to design some actually useful algorithms for the new external memory model. Our goal in this section is to slowly build up more complex things and eventually get to *external sorting* and its interesting applications.

The algorithm will be based on the standard merge sorting algorithm, so we need to derive its main primitive first.

Merge

Problem. Given two sorted arrays a and b of lengths N and M , produce a single sorted array c of length $N + M$ containing all of their elements.

The standard two-pointer technique for merging sorted arrays looks like this:

```

void merge(int *a, int *b, int *c, int n, int m) {
    int i = 0, j = 0;
    for (int k = 0; k < n + m; k++) {
        if (i < n && (j == m || a[i] < b[j]))
            c[k] = a[i++];
        else
            c[k] = b[j++];
    }
}

```

In terms of memory operations, we just linearly read all elements of a and b and linearly write all elements of c . Since these reads and writes can be buffered, it works in $SCAN(N + M)$ I/O operations.

So far the examples have been simple, and their analysis doesn't differ too much from the RAM model, except that we divide the final answer by the block size B . But here is a case where this is not so.

k -way merging. Consider the modification of this algorithm where we need to merge not just two arrays, but k arrays of total size N — by likewise looking at k values, choosing the minimum between them, writing

it into c , and incrementing one of the iterators.

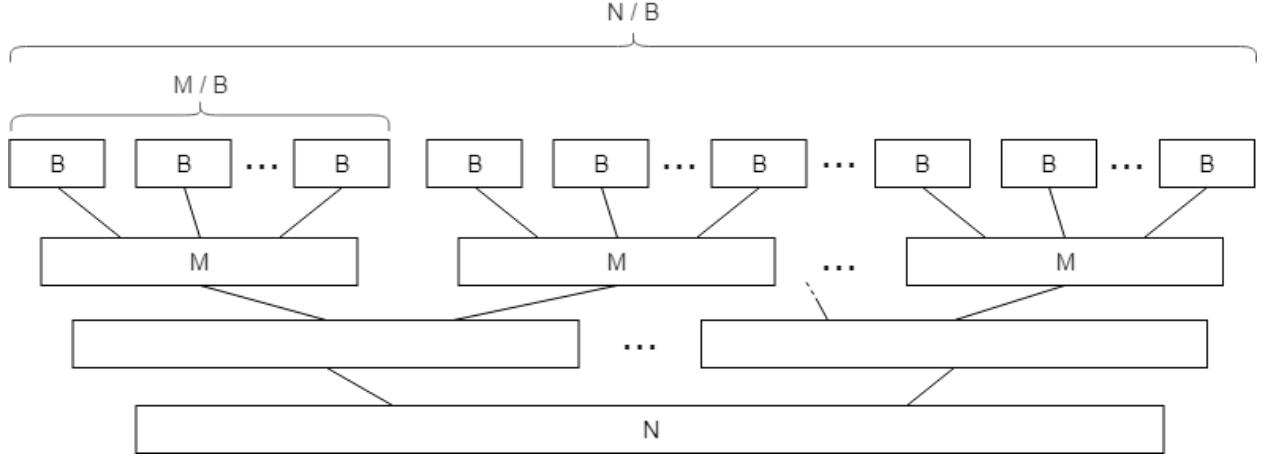
In the standard RAM model, the asymptotic complexity would be multiplied k , since we would need to perform $O(k)$ comparisons to fill each next element. But in the external memory model, since everything we do in-memory doesn't cost us anything, its asymptotic complexity would not change as long as we can fit $(k + 1)$ full blocks in memory, that is, if $k = O(\frac{M}{B})$.

Remember the $M \gg B$ assumption when we introduced the computational model? If we have $M \geq B^{1+\epsilon}$ for $\epsilon > 0$, then we can fit any sub-polynomial number of blocks in memory, certainly including $O(\frac{M}{B})$. This condition is called *tall cache assumption*, and it is usually required in many other external memory algorithms.

Merge Sorting

The “normal” complexity of the standard mergesort algorithm is $O(N \log_2 N)$: on each of its $O(\log_2 N)$ “layers,” the algorithms need to go through all N elements in total and merge them in linear time.

In the external memory model, when we read a block of size M , we can sort its elements “for free,” since they are already in memory. This way we can split the arrays into $O(\frac{N}{M})$ blocks of consecutive elements and sort them separately as the base step, and only then merge them.



This effectively means that, in terms of I/O operations, the first $O(\log M)$ layers of mergesort are free, and there are only $O(\log_2 \frac{N}{M})$ non-zero-cost layers, each mergeable in $O(\frac{N}{B})$ IOPS in total. This brings total I/O complexity to

$$O\left(\frac{N}{B} \log_2 \frac{N}{M}\right)$$

This is quite fast. If we have 1GB of memory and 10GB of data, this essentially means that we need a little bit more than 3 times the effort than just reading the data to sort it. Interestingly enough, we can do better.

k -way Mergesort

Half of a page ago we have learned that in the external memory model, we can merge k arrays just as easily as two arrays — at the cost of reading them. Why don't we apply this fact here?

Let's sort each block of size M in-memory just as we did before, but during each merge stage, we will split sorted blocks not just in pairs to be merged, but take as many blocks we can fit into our memory during a k -way merge. This way the height of the merge tree would be greatly reduced, while each layer would still be done in $O(\frac{N}{B})$ IOPS.

How many sorted arrays can we merge at once? Exactly $k = \frac{M}{B}$, since we need memory for one block for each array. Since the total number of layers will be reduced to $\log_{\frac{M}{B}} \frac{N}{M}$, the total complexity will be reduced to

$$SORT(N) \stackrel{\text{def}}{=} O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M}\right)$$

Note that, in our example, we have 10GB of data, 1GB of memory, and the block size is around 1MB for HDD. This makes $\frac{M}{B} = 1000$ and $\frac{N}{M} = 10$, and so the logarithm is less than one (namely, $\log_{1000} 10 = \frac{1}{3}$). Of course, we can't sort an array faster than reading it, so this analysis applies to the cases when we have a very large dataset, small memory, and/or large block sizes, which rarely happens in real life these days.

Practical Implementation

Under more realistic constraints, instead of using $\log_{\frac{M}{B}} \frac{N}{M}$ layers, we can use just two: one for sorting data in blocks of M elements, and another one for merging all of them at once. This way, from the I/O operations perspective, we just loop around our dataset twice. And with a gigabyte of RAM and a block size of 1MB, this way can sort arrays up to a terabyte in size.

Here is how the first phase looks in C++. This program opens a multi-gigabyte binary file with unsorted integers, reads it in blocks of 256MB, sorts them in memory, and then writes them back in files named `part-000.bin`, `part-001.bin`, `part-002.bin`, and so on:

```
const int B = (1<<20) / 4; // 1 MB blocks of integers
const int M = (1<<28) / 4; // available memory

FILE *input = fopen("input.bin", "rb");
std::vector<FILE*> parts;

while (true) {
    static int part[M]; // better delete it right after
    int n = fread(part, 4, M, input);

    if (n == 0)
        break;

    // sort a block in-memory
    std::sort(part, part + n);

    char fpart[sizeof "part-999.bin"];
    sprintf(fpart, "part-%03d.bin", parts.size());

    printf("Writing %d elements into %s...\n", n, fpart);

    FILE *file = fopen(fpart, "wb");
    fwrite(part, 4, n, file);
    fclose(file);

    file = fopen(fpart, "rb");
    parts.push_back(file);
}

fclose(input);
```

What is left now is to merge them together. The bandwidth of modern HDDs can be quite high, and there may be a lot of parts to merge, so the I/O efficiency of this stage is not our only concern: we also need a

faster way to merge k arrays than by finding minima with $O(k)$ comparisons. We can do that in $O(\log k)$ time per element if we maintain a min-heap for these k elements, in a manner almost identical to heapsort.

Here is how to implement it. First, we are going to need a heap (`priority_queue` in C++):

```
struct Pointer {
    int key, part; // the element itself and the number of its part

    bool operator<(const Pointer& other) const {
        return key > other.key; // std::priority_queue is a max-heap by default
    }
};
```

```
std::priority_queue<Pointer> q;
```

Then, we need to allocate and fill the buffers:

```
const int nparts = parts.size();

auto buffers = new int[nparts][B]; // buffers for each part
int *l = new int[nparts],           // # of already processed buffer elements
     *r = new int[nparts];          // buffer size (in case it isn't full)

// now we add fill the buffer for each part and add their elements to the heap
for (int part = 0; part < nparts; part++) {
    l[part] = 1; // if the element is in the heap, we also consider it "processed"
    r[part] = fread(buffers[part], 4, B, parts[part]);
    q.push({buffers[part][0], part});
}
```

Now we just need to pop elements from the heap into the result file until it is empty, carefully writing and reading elements in batches:

```
FILE *output = fopen("output.bin", "w");

int outbuffer[B]; // the output buffer
int buffered = 0; // number of elements in it

while (!q.empty()) {
    auto [key, part] = q.top();
    q.pop();

    // write the minimum to the output buffer
    outbuffer[buffered++] = key;
    // check if it needs to be committed to the file
    if (buffered == B) {
        fwrite(outbuffer, 4, B, output);
        buffered = 0;
    }

    // fetch a new block of that part if needed
    if (l[part] == r[part]) {
        r[part] = fread(buffers[part], 4, B, parts[part]);
        l[part] = 0;
    }

    // read a new element from that part unless we've already processed all of it
```

```

    if (l[part] < r[part]) {
        q.push({buffers[part][l[part]], part});
        l[part]++;
    }
}

// write what's left of the output buffer
fwrite(outbuffer, 4, buffered, output);

```

```

//clean up
delete[] buffers;
for (FILE *file : parts)
    fclose(file);
fclose(output);

```

This implementation is not particularly effective or safe-looking (well, this is basically plain C), but is a good educational example of how to work with low-level memory APIs.

Joining

Sorting is mainly used not by itself, but as an intermediate step for other operations. One important real-world use case of external sorting is joining (as in “SQL join”), used in databases and other data processing applications.

Problem. Given two lists of tuples (x_i, a_{x_i}) and (y_i, b_{y_i}) , output a list (k, a_{x_k}, b_{y_k}) such that $x_k = y_k$

The optimal solution would be to sort the two lists and then use the standard two-pointer technique to merge them. The I/O complexity here would be the same as sorting, and just $O(\frac{N}{B})$ if the arrays are already sorted. This is why most data processing applications (databases, MapReduce systems) like to keep their tables at least partially sorted.

Other approaches. Note that this analysis is only applicable in the external memory setting — that is, if you don’t have the memory to read the entire dataset. In the real world, alternative methods may be faster.

The simplest of them is probably *hash join*, which goes something like this:

```

def join(a, b):
    d = dict(a)
    for x, y in b:
        if x in d:
            yield d[x]

```

In external memory, joining two lists with a hash table would be unfeasible, as it would involve doing $O(M)$ block reads, even though only one element is used in each of them.

Another method is to use alternative sorting algorithms such as radix sort. In particular, radix sort would work in $O(\frac{N}{B} \cdot w)$ block reads if enough memory is available to maintain buffers for all possible keys, and it could be faster in the case of small keys and large datasets.

List Ranking

In this section, we will apply external sorting and joining to solve a problem that seems useless on the surface but is actually a key primitive used in a large number of external memory and parallel algorithms.

Problem. Given a singly-linked list, compute the *rank* of each element, equal to its distance from the *last* element.

This problem can be trivially solved in the RAM model: you just traverse the entire list with a counter. But this pointer jumping wouldn’t work well in the external memory setting because the list nodes are stored

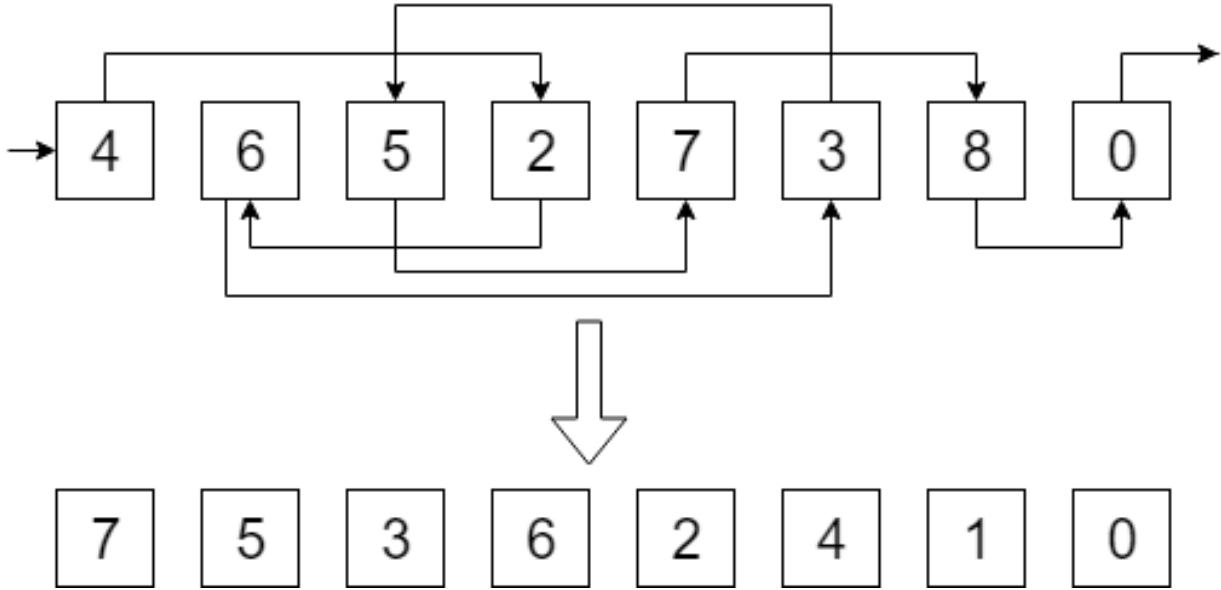


Figure 1: Example input and output for the list ranking problem

arbitrarily, and in the worst case, reading each new node may require reading a new block.

Algorithm

Consider a slightly more general version of the problem. Now, each element has a *weight* w_i , and for each element, we need to compute the sum of the weights of all its preceding elements instead of just its rank. To solve the initial problem, we can just set all weights equal to 1.

The main idea of the algorithm is to remove some fraction of elements, recursively solve the problem, and then use these weight-ranks to reconstruct the answer for the initial problem — which is the tricky part.

Consider some three consecutive elements x , y and z . Assume that we deleted y and solved the problem for the remaining list, which included x and z , and now we need to restore the answer for the original triplet. The weight of x would be correct as it is, but we need to calculate the answer for y and adjust it for z , namely:

- $w'_y = w_y + w_x$
- $w'_z = w_z + w_y + w_x$

Now, we can just delete, say, the first element, solve the problem recursively, and recalculate weights for the original array. But, unfortunately, it would work in quadratic time, because to make the update, we would need to know where its neighbors are, and since we can't hold the entire array in memory, we would need to scan it each time.

Therefore, on each step, we want to remove as many elements as possible. But we also have a constraint: we can't remove two consecutive elements because then merging results wouldn't be that simple.

Ideally, we want to split our list into even and odd elements, but doing this is not simpler than the initial problem. One workaround is to choose the elements at random: toss a coin for each element, and then remove all “heads” after which a “tail” follows. This way no two consecutive elements will ever be selected, and on average we get rid of $\frac{1}{4}$ of the current list. The arithmetic complexity of this solution would still be linear, because

$$T(N) = T\left(\frac{3}{4}N\right) = O(N)$$

The only tricky part here is how to implement the merge step in external memory. To do it efficiently, we need to maintain our list in the following form:

- List of tuples (i, j) indicating that element j follows after element i
- List of tuples (i, w_i) indicating that element i currently has weight w_i
- A list of deleted elements

Now, to restore the answer after randomly deleting some elements and recursively solving the smaller problem, we need to iterate over all lists using three pointers looking for deleted elements. and for each such element, we will write (j, w_i) to a separate table, which would signify that before the recursive step we need to add w_i to j . We can then join this new table with initial weights, add these additional weights to them.

After coming back from the recursion, we need to update weights for the deleted elements, which we can do with the same technique, iterating over reversed connections instead of direct ones.

I/O complexity of this algorithm will therefore be the same as joining, namely $SORT(N) = O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M}\right)$.

Applications

List ranking is especially useful in graph algorithms.

For example, we can obtain the Euler tour of a tree in external memory by constructing a linked list from the tree that corresponds to its Euler tour and then applying the list ranking algorithm — the ranks of each node will be the same as its index tin_v in the Euler tour. To construct this list, we need to:

- split each undirected edge into two directed ones;
- duplicate the parent node for each up-edge (because list nodes can only have one incoming edge, but we visit some vertices multiple times);
- route each such node either to the “next sibling,” if it has one, or otherwise to its own parent;
- and then finally break the resulting cycle at the root.

This general technique is called *tree contraction*, and it serves as the basis for a large number of tree algorithms.

The same approach can be applied to parallel algorithms, and we will cover that much more deeply in part II.

Eviction Policies

You can control the I/O operations of your program manually, but most of the time people just rely on automatic bufferization and caching, either due to laziness or because of the computing environment limitations.

But automatic caching comes with its own challenges. When a program runs out of working memory to store its intermediate data, it needs to get rid of one block to make space for a new one. A concrete rule for deciding which data to retain in the cache in case of conflicts is called an *eviction policy*.

This rule can be arbitrary, but there are several popular choices:

- First in first out (FIFO): simply evict the earliest added block, without any regard to how often it was accessed before (the same way as a FIFO queue).
- Least recently used (LRU): evict the block that has not been accessed for the longest period of time.
- Last in first out (LIFO) and most recently used (MRU): the opposite of the previous two. It seems harmful to delete the hottest blocks, but there are scenarios where these policies are optimal, such as repeatedly looping around a file in a cycle.
- Least-frequently used (LFU): counts how often each block has been requested and discards the one used least often. Some variations also account for changing access patterns over time, such as using a time window to only consider the last n accesses or using exponential averaging to give recent accesses more weight.

- Random replacement (RR): discard a block randomly. The advantage is that it does not need to maintain any data structures with block information.

There is a natural trade-off between the accuracy of eviction policies and the additional overhead due to the complexity of their implementations. For a CPU cache, you need a simple policy that can be easily implemented in hardware with next-to-zero latency, while in more slow-paced and plannable settings such as Netflix deciding in which data centers to store their movies or Google Drive optimizing where to store user data, it makes sense to use more complex policies, possibly involving some machine learning to predict when the data is going to be accessed next.

Optimal Caching

Apart from the aforementioned strategies, there is also the theoretical *optimal policy*, denoted as OPT or MIN , which determines, for a given sequence of queries, which blocks should be retained to minimize the total number of cache misses.

These decisions can be made using a simple greedy approach called *Bélády algorithm*: we can just keep the *latest-to-be-used* block, and it can be shown by contradiction that doing so is always one of the optimal solutions. The downside of this method is that you either need to have these queries in advance or somehow be able to predict the future.

The good thing is that, in terms of asymptotic complexity, it doesn't really matter which particular method is used. Sleator & Tarjan showed that in most cases, the performance of popular policies such as *LRU* differs from OPT just by a constant factor.

Theorem. Let LRU_M and OPT_M denote the number of blocks a computer with M internal memory would need to access while executing the same algorithm following the least recently used cache replacement policy and the theoretical minimum respectively. Then:

$$LRU_M \leq 2 \cdot OPT_{M/2}$$

The main idea of the proof is to consider the worst case scenario. For LRU it would be the repeating series of $\frac{M}{B}$ distinct blocks: each block is new and so LRU has 100% cache misses. Meanwhile, $OPT_{M/2}$ would be able to cache half of them (but not more, because it only has half the memory). Thus LRU_M needs to fetch double the number of blocks that $OPT_{M/2}$ does, which is basically what is expressed in the inequality, and anything better for *LRU* would only weaken it.



Figure 2: Dimmed are the blocks cached by OPT (but not cached by LRU)

This is a very relieving result. It means that, at least in terms of asymptotic I/O complexity, you can just assume that the eviction policy is either LRU or OPT — whichever is easier for you — do complexity analysis with it, and the result you get will normally transfer to any other reasonable cache replacement policy.

Implementing Caching

This is not always a trivial task to find the right block to evict in a reasonable time. While CPU caches are implemented in hardware (usually as some variation of LRU), higher-level eviction policies have to rely on software to store certain statistics about the blocks and maintain data structures on top of them to speed up the process.

Let's think about what we need to implement an LRU cache. Assume we are storing some moderately large objects — say, we need to develop a cache for a database, there both the requests and replies are medium-sized strings in some SQL dialect, so the overhead of our structure is small but non-negligible.

First of all, we need a hash table to find the data itself. Since we are working with large variable-length strings, it makes sense to use the hash of the query as the key and a pointer to a heap-allocated result string as the value.

To implement the LRU logic, the simplest approach would be to create a queue where we put the current time and IDs/keys of objects when we access them, and also store when each object was accessed the last time (not necessarily as a timestamp — any increasing counter will suffice).

Now, when we need to free up space, we can find the least recently used object by popping elements from the front of the queue. We can't just delete them, because it may be that they were accessed again since their record was added to the queue. So we need to check if the time of when we put them in queue matches the time of when they were last accessed, and only then free up the memory.

The only remaining issue here is that we add an entry to the queue each time a block is accessed, and only remove entries when we have a cache miss and start popping them off from the front until we have a match. This may lead to the queue overflowing, and to mitigate this, instead of adding an entry and forgetting about it, we can move it to the end of the queue on a cache hit right away.

To support this, we need to implement the queue over a doubly-linked list and store a pointer to the block's node in the queue in the hash table. Then, when we have a cache hit, we follow the pointer and remove the node from the linked list in constant time, and add a newer node to the end of the queue. This way, at any point in time, there would be exactly as many nodes in the queue as we have objects, and the memory overhead will be guaranteed to be constant per cache entry.

As an exercise, try to think about ways to implement other caching strategies.

Cache-Oblivious Algorithms

In the context of the external memory model, there are two types of efficient algorithms:

- *Cache-aware* algorithms that are efficient for *known* B and M .
- *Cache-oblivious* algorithms that are efficient for *any* B and M .

For example, external merge sorting is a cache-aware, but not cache-oblivious algorithm: we need to know the memory characteristics of the system, namely the ratio of available memory to the block size, to find the right k to perform k -way merge sort.

Cache-oblivious algorithms are interesting because they automatically become optimal for all memory levels in the cache hierarchy, and not just the one for which they were specifically tuned. In this article, we consider some of their applications in matrix calculations.

Matrix Transposition

Assume we have a square matrix A of size $N \times N$, and we need to transpose it. The naive by-definition approach would go something like this:

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < i; j++)
        swap(a[j * N + i], a[i * N + j]);
```

Here we used a single pointer to the beginning of the memory region instead of a 2d array to be more explicit about its memory operations.

The I/O complexity of this code is $O(N^2)$ because the writes are not sequential. If you try to swap the iteration variables, it will be the other way around, but the result is going to be the same.

Algorithm

The *cache-oblivious* algorithm relies on the following block matrix identity:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^T = \begin{pmatrix} A^T & C^T \\ B^T & D^T \end{pmatrix}$$

It lets us solve the problem recursively using a divide-and-conquer approach:

1. Divide the input matrix into 4 smaller matrices.
2. Transpose each one recursively.
3. Combine results by swapping the corner result matrices.

Implementing D&C on matrices is a bit more complex than on arrays, but the main idea is the same. Instead of copying submatrices explicitly, we want to use “views” into them, and also switch to the naive method when the data starts fitting in the L1 cache (or pick something small like 32×32 if you don’t know it in advance). We also need to carefully handle the case when we have odd n and thus can’t split the matrix into 4 equal submatrices.

```
void transpose(int *a, int n, int N) {
    if (n <= 32) {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < i; j++)
                swap(a[i * N + j], a[j * N + i]);
    } else {
        int k = n / 2;

        transpose(a, k, N);
        transpose(a + k, k, N);
        transpose(a + k * N, k, N);
        transpose(a + k * N + k, k, N);

        for (int i = 0; i < k; i++)
            for (int j = 0; j < k; j++)
                swap(a[i * N + (j + k)], a[(i + k) * N + j]);

        if (n & 1)
            for (int i = 0; i < n - 1; i++)
                swap(a[i * N + n - 1], a[(n - 1) * N + i]);
    }
}
```

The I/O complexity of the algorithm is $O(\frac{N^2}{B})$, as we only need to touch roughly half the memory blocks during each merge stage, meaning that on each stage our problem becomes smaller.

Adapting this code for the general case of non-square matrices is left as an exercise to the reader.

Matrix Multiplication

Next, let’s consider something slightly more complex: matrix multiplication.

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

The naive algorithm just translates its definition into code:

```
// don't forget to initialize c[][] with zeroes
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
```

```

for (int k = 0; k < n; k++)
    c[i * n + j] += a[i * n + k] * b[k * n + j];

```

It needs to access $O(N^3)$ blocks in total as each scalar multiplication needs a separate block read.

One well-known optimization is to transpose B first:

```

for (int i = 0; i < n; i++)
    for (int j = 0; j < i; j++)
        swap(b[j][i], b[i][j])
// ^ or use our faster transpose from before

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            c[i * n + j] += a[i * n + k] * b[j * n + k]; // <- note the indices

```

Whether the transpose is done naively or with the cache-oblivious method we previously developed, the matrix multiplication with one of the matrices transposed would work in $O(N^3/B + N^2)$ as all memory accesses are now sequential.

It seems like we can't do better, but it turns out we can.

Algorithm

Cache-oblivious matrix multiplication relies on essentially the same trick as the transposition. We need to divide the data until it fits into lowest cache (i.e., $N^2 \leq M$). For matrix multiplication, this equates to using this formula:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

It is slightly harder to implement though because we now have a total of 8 recursive matrix multiplications:

```

void matmul(const float *a, const float *b, float *c, int n, int N) {
    if (n <= 32) {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                for (int k = 0; k < n; k++)
                    c[i * N + j] += a[i * N + k] * b[k * N + j];
    } else {
        int k = n / 2;

        // c11 = a11 b11 + a12 b21
        matmul(a,           b,           c,   k, N);
        matmul(a + k,       b + k,       c,   k, N);

        // c12 = a11 b12 + a12 b22
        matmul(a,           b + k,       c + k, k, N);
        matmul(a + k,       b + k * N,  c + k, k, N);

        // c21 = a21 b11 + a22 b21
        matmul(a + k * N,   b,           c + k * N, k, N);
        matmul(a + k * N + k, b + k * N, c + k * N, k, N);

        // c22 = a21 b12 + a22 b22
        mul(a + k * N,      b + k,       c + k * N + k, k, N);
    }
}

```

```

mul(a + k * N + k, b + k * N + k, c + k * N + k, k, N);

if (n & 1) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = (i < n - 1 && j < n - 1) ? n - 1 : 0; k < n; k++)
                c[i * N + j] += a[i * N + k] * b[k * N + j];
}
}

```

Because there are many other factors in play here, we are not going to benchmark this implementation, and instead just do its theoretical performance analysis in external memory model.

Analysis

Arithmetic complexity of the algorithm remains the same, because the recurrence

$$T(N) = 8 \cdot T(N/2) + \Theta(N^2)$$

is solved by $T(N) = \Theta(N^3)$.

It doesn't seem like we "conquered" anything yet, but let's think about its I/O complexity:

$$T(N) = \begin{cases} O(\frac{N^2}{B}) & N \leq \sqrt{M} \text{ (we only need to read it)} \\ 8 \cdot T(N/2) + O(\frac{N^2}{B}) & \text{otherwise} \end{cases}$$

The recurrence is dominated by $O((\frac{N}{\sqrt{M}})^3)$ base cases, meaning that the total complexity is

$$T(N) = O\left(\frac{(\sqrt{M})^2}{B} \cdot \left(\frac{N}{\sqrt{M}}\right)^3\right) = O\left(\frac{N^3}{B\sqrt{M}}\right)$$

This is better than just $O(\frac{N^3}{B})$, and by quite a lot.

Strassen Algorithm

In a spirit similar to the Karatsuba algorithm, matrix multiplication can be decomposed in a way that involves 7 matrix multiplications of size $\frac{n}{2}$, and the master theorem tells us that such divide-and-conquer algorithm would work in $O(n^{\log_2 7}) \approx O(n^{2.81})$ time and a similar asymptotic in the external memory model.

This technique, known as the Strassen algorithm, similarly splits each matrix into 4:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

But then it computes intermediate products of the $\frac{N}{2} \times \frac{N}{2}$ matrices and combines them to get matrix C :

$$\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) & C_{11} &= M_1 + M_4 - M_5 + M_7 \\
M_2 &= (A_{21} + A_{22})B_{11} & C_{12} &= M_3 + M_5 \\
M_3 &= A_{11}(B_{21} - B_{22}) & C_{21} &= M_2 + M_4 \\
M_4 &= A_{22}(B_{21} - B_{11}) & C_{22} &= M_1 - M_2 + M_3 + M_6 \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}$$

You can verify these formulas with simple substitution if you feel like it.

As far as I know, none of the mainstream optimized linear algebra libraries use the Strassen algorithm, although there are some prototype implementations that are efficient for matrices larger than 2000 or so.

This technique can and actually has been extended multiple times to reduce the asymptotic even further by considering more submatrix products. As of 2020, current world record is $O(n^{2.3728596})$. Whether you can multiply matrices in $O(n^2)$ or at least $O(n^2 \log^k n)$ time is an open problem.

Further Reading

For a solid theoretical viewpoint, consider reading Cache-Oblivious Algorithms and Data Structures by Erik Demaine.

Spatial and Temporal Locality

To precisely assess the performance of an algorithm in terms of its memory operations, we need to take into account multiple characteristics of the cache system: the number of cache layers, the memory and block sizes of each layer, the exact strategy used for cache eviction by each layer, and sometimes even the details of the memory paging mechanism.

Abstracting away from all these minor details helps a lot in the first stages of designing algorithms. Instead of calculating theoretical cache hit rates, it often makes more sense to reason about cache performance in more qualitative terms.

In this context, we can talk about the degree of cache reuse primarily in two ways:

- *Temporal locality* refers to the repeated access of the same data within a relatively small time period, such that the data likely remains cached between the requests.
- *Spatial locality* refers to the use of elements relatively close to each other in terms of their memory locations, such that they are likely fetched in the same memory block.

In other words, temporal locality is when it is likely that this same memory location will soon be requested again, while spatial locality is when it is likely that a nearby location will be requested right after.

In this section, we will do some case studies to show how these high-level concepts can help in practical optimization.

Depth-First vs. Breadth-First

Consider a divide-and-conquer algorithm such as merge sorting. There are two approaches to implementing it:

- We can implement it recursively, or “depth-first,” the way it is normally implemented: sort the left half, sort the right half and then merge the results.
- We can implement it iteratively, or “breadth-first:” do the lowest “layer” first, looping through the entire dataset and comparing odd elements with even elements, then merge the first two elements with the second two elements, the third two elements with the fourth two elements and so on.

It seems like the second approach is more cumbersome, but faster — because recursion is always slow, right?

Generally, recursion is indeed slow, but this is not the case for this and many similar divide-and-conquer algorithms. Although the iterative approach has the advantage of only doing sequential I/O, the recursive approach has much better temporal locality: when a segment fully fits into the cache, it stays there for all lower layers of recursion, resulting in better access times later on.

In fact, since we only need to split the array $O(\log \frac{N}{M})$ times until this happens, we would only need to read $O(\frac{N}{B} \log \frac{N}{M})$ blocks in total, while in the iterative approach the entire array will be read from scratch $O(\log N)$ times no matter what. This results in the speedup of $O(\frac{\log N}{\log N - \log M})$, which may be up to an order of magnitude.

In practice, there is still some overhead associated with the recursion, and for this reason, it makes sense to use hybrid algorithms where we don't go all the way down to the base case and instead switch to the iterative code on the lower levels of recursion.

Dynamic Programming

Similar reasoning can be applied to the implementations of dynamic programming algorithms but leading to the reverse result. Consider the classic *knapsack problem*: given N items with positive integer costs c_i , pick a subset of items with the maximum total cost that does not exceed a given constant W .

The way to solve it is to introduce the *state* $f[n, w]$, which corresponds to the maximum total cost not exceeding w that can be achieved using only the first n items. These values can be computed in $O(1)$ time per entry if we consider either taking or not taking the n -th item and using the previous states of the dynamic to make the optimal decision.

Python has a handy `@lru_cache` decorator which can be used for implementing it with memoized recursion:

```
@lru_cache
def f(n, w):
    # check if we have no items to choose
    if n == 0:
        return 0

    # check if we can't pick the last item (note zero-based indexing)
    if c[n - 1] > w:
        return f(n - 1, w)

    # otherwise, we can either pick the last item or not
    return max(f(n - 1, w), c[n - 1] + f(n - 1, w - c[n - 1]))
```

When computing $f[N, W]$, the recursion may visit up to $O(N \cdot W)$ different states, which is asymptotically efficient, but rather slow in reality. Even after nullifying the overhead of Python recursion and all the hash table queries required for the LRU cache to work, it would still be slow because it does random I/O throughout most of the execution.

What we can do instead is to create a two-dimensional array for the dynamic and replace the recursion with a nice nested loop like this:

```
int f[N + 1][W + 1] = {0}; // this zero-fills the array

for (int n = 1; n <= N; n++)
    for (int w = 0; w <= W; w++)
        f[n][w] = c[n - 1] > w ?
            f[n - 1][w] :
            max(f[n - 1][k], c[n - 1] + f[n - 1][w - c[n - 1]]);
```

Notice that we are only using the previous layer of the dynamic to calculate the next one. This means that if we can store one layer in the cache, we would only need to write $O(\frac{N \cdot W}{B})$ blocks in external memory.

Moreover, if we only need the answer, we don't actually have to store the whole 2d array but only the last layer. This lets us use just $O(W)$ memory by maintaining a single array of W values. To simplify the code, we can slightly change the dynamic to store a binary value: whether it is possible to get the sum of exactly w using the items that we have already considered. This dynamic is even faster to compute:

```
bool f[W + 1] = {0};
f[0] = 1;
for (int n = 0; n < N; n++)
    for (int x = W - c[n]; x >= 0; x--)
        f[x + c[n]] |= f[x];
```

As a side note, now that it only uses simple bitwise operations, it can be optimized further by using a bitset:

```
std::bitset<W + 1> b;
b[0] = 1;
for (int n = 0; n < N; n++)
    b |= b << c[n];
```

Surprisingly, there is still some room for improvement, and we will come back to this problem later.

Sparse Table

Sparse table is a *static* data structure that is often used for solving the *static RMQ* problem and computing any similar *idempotent range reductions* in general. It can be formally defined as a two-dimensional array of size $\log n \times n$:

$$t[k][i] = \min\{a_i, a_{i+1}, \dots, a_{i+2^k-1}\}$$

In plain English: we store the minimum on each segment whose length is a power of two.

Such array can be used for calculating minima on arbitrary segments in constant time because for each segment we can always find two possibly overlapping segments whose sizes are the same power of two, the union of which gives the whole segment.



This means that we can just take the minimum of these two precomputed minimums as the answer:

```
int rmq(int l, int r) { // half-interval [l; r)
    int t = __lg(r - l);
    return min(mn[t][l], mn[t][r - (1 << t)]);
}
```

The `__lg` function is an intrinsic available in GCC that calculates the binary logarithm of a number rounded down. Internally it uses the `c1z` (“count leading zeros”) instruction and subtracts this count from 32 (in case of a 32-bit integer), and thus takes just a few cycles.

The reason why I bring it up in this article is that there are multiple alternative ways it can be built, with different efficiencies in terms of memory operations. In general, a sparse table can be built in $O(n \log n)$ time in dynamic programming fashion by iterating in the order of increasing i or k and applying the following identity:

$$t[k][i] = \min(t[k-1][i], t[k-1][i + 2^{k-1}])$$

Now, there are two design choices to make: whether the log-size k should be the first or the second dimension, and whether to iterate over k and then i or the other way around. This means that there are $2 \times 2 = 4$ ways to build it, and here is the optimal one:

```
int mn[logn][maxn];

memcpy(mn[0], a, sizeof a);

for (int l = 0; l < logn - 1; l++)
    for (int i = 0; i + (2 << l) <= n; i++)
        mn[l + 1][i] = min(mn[l][i], mn[l][i + (1 << l)]);
```

This is the only combination of the memory layout and the iteration order that results in beautiful linear passes that work ~3x faster. As an exercise, consider the other three variants and think about *why* they are slower.

Array-of-Structs vs. Struct-of-Arrays

Suppose that you want to implement a binary tree and store its fields in separate arrays like this:

```
int left_child[maxn], right_child[maxn], key[maxn], size[maxn];
```

Such memory layout, when we store each field separately from others, is called *struct-of-arrays* (SoA). In most cases, when implementing tree operations, you access a node and then shortly after all or most of its internal data. If these fields are stored separately, this would mean that they are also located in different memory blocks. If some of the requested fields happen to be cached while the others are not, you would still have to wait for the slowest of them to be fetched.

In contrast, if it was instead stored as an array-of-structs (AoS), you would need ~4 times fewer block reads as all the data of a node is stored in the same block and fetched at once:

```
struct Node {
    int left_child, right_child, key, size;
};

Node t[maxn];
```

The AoS layout is usually preferred for data structures, but SoA still has good uses: while it is worse for searching, it is much better for linear scanning.

This difference in design is important in data processing applications. For example, databases can be either *row-* or *column-oriented* (also called *columnar*):

- *Row-oriented* storage formats are used when you need to search for a limited number of objects in a large dataset and/or fetch all or most of their fields. Examples: PostgreSQL, MongoDB.
- *Columnar* storage formats are used for big data processing and analytics, where you need to scan through everything anyway to calculate certain statistics. Examples: ClickHouse, Hbase.

Columnar formats have the additional advantage that you can only read the fields that you need, as different fields are stored in separate external memory regions.

Chapter 9: RAM & CPU Caches

Algorithms for Modern Hardware

Contents

Memory Bandwidth	2
Frequency Scaling	2
Directional Access	3
Bypassing the Cache	3
Memory Latency	4
Theoretical Latency	5
Frequency Scaling	6
Cache Lines	6
Memory Sharing	7
CPU Affinity	8
Saturating Bandwidth	8
Memory-Level Parallelism	9
Direct Experiment	9
Prefetching	10
Hardware Prefetching	10
Software Prefetching	11
Alignment and Packing	12
Aligned Allocation	12
Structure Alignment	13
Optimizing Member Order	14
Structure Packing	15
Bit fields	15
RAM & CPU Caches	16
Experimental Setup	16
Acknowledgements	16
Pointer Alternatives	16
Pointers	17
Bit Fields	17
Cache Associativity	18
Hardware Caches	19
Address Translation	20
Pathological Mappings	20
Memory Paging	21
Changing Page Size	22
Impact of Huge Pages	22

Dotted vertical lines are cache layer sizes

Figure 1: Dotted vertical lines are cache layer sizes

AoS and SoA	23
Experiment	23
Temporary Storage Contention	24
Padded AoS	25
RAM-Specific Timings	25

Memory Bandwidth

On the data path between the CPU registers and the RAM, there is a hierarchy of *caches* that exist to speed up access to frequently used data: the layers closer to the processor are faster but also smaller in size. The word “faster” here applies to two closely related but separate timings:

- The delay between the moment when a read or a write is initiated and when the data arrives (*latency*).
- The number of memory operations that can be processed per unit of time (*bandwidth*).

For many algorithms, memory bandwidth is the most important characteristic of the cache system. And at the same time, it is also the easiest to measure, so we are going to start with it.

For our experiment, we create an array and iterate over it K times, incrementing its values:

```
int a[N];  
  
for (int t = 0; t < K; t++)  
    for (int i = 0; i < N; i++)  
        a[i]++;
```

Changing N and adjusting K so that the total number of array cells accessed remains roughly constant and expressing the total time in “operations per second,” we get a graph like this:

You can clearly see the cache sizes on this graph:

- When the whole array fits into the lowest layer of cache, the program is bottlenecked by the CPU rather than the L1 cache bandwidth. As the array becomes larger, the overhead associated with the first iterations of the loop becomes smaller, and the performance gets closer to its theoretical maximum of 16 GFLOPS.
- But then the performance drops: first to 12-13 GFLOPS when it exceeds the L1 cache, and then gradually to about 2 GFLOPS when it can no longer fit in the L3 cache.

This situation is typical for many lightweight loops.

Frequency Scaling

All CPU cache layers are placed on the same microchip as the processor, so the bandwidth, latency, and all its other characteristics scale with the clock frequency. The RAM, on the other side, lives on its own fixed clock, and its characteristics remain constant. We can observe this by re-running the same benchmarking with turbo boost on:

This detail comes into play when comparing algorithm implementations. When the working dataset fits in the cache, the relative performance of the two implementations may be different depending on the CPU clock rate because the RAM remains unaffected by it (while everything else does not).

For this reason, it is [advised](#) to keep the clock rate fixed, and as the turbo boost isn't stable enough, we run most of the benchmarks in this book at plain 2GHz.

Directional Access

This incrementing loop needs to perform both reads and writes during its execution: on each iteration, we fetch a value, increment it, and then write it back. In many applications, we only need to do one of them, so let's try to measure unidirectional bandwidth.

Calculating the sum of an array only requires memory reads:

```
for (int i = 0; i < N; i++)
    s += a[i];
```

And zeroing an array (or filling it with any other constant value) only requires memory writes:

```
for (int i = 0; i < N; i++)
    a[i] = 0;
```

Both loops are trivially [vectorized](#) by the compiler, and the second one is actually replaced with a `memset`, so the CPU is also not the bottleneck here (except when the array fits into the L1 cache).

The reason why unidirectional and bidirectional memory accesses would perform differently is that they share the cache and memory buses and other CPU facilities. In the case of RAM, this causes a twofold difference in performance between the pure read and simultaneous read and write scenarios because the memory controller has to switch between the modes on the one-way memory bus, thus halving the bandwidth. The performance drop is less severe for the L2 cache: the bottleneck here is not the cache bus, so the incrementing loop loses by only ~15%.

There is one interesting anomaly on the graph, namely that the write-only loop performs the same as the read-and-write one when the array hits the L3 cache and the RAM. This is because the CPU moves the data to the highest level of cache on each access, whether it is a read or a write — which is typically a good optimization, as in many use cases we will be needing it soon. When reading data, this isn't a problem, as the data travels through the cache hierarchy anyway, but when writing, this causes another implicit read to be dispatched right after a write — thus requiring twice the bus bandwidth.

Bypassing the Cache

We can prevent the CPU from prefetching the data that we just have written by using *non-temporal* memory accesses. To do this, we need to re-implement the zeroing loop more directly without relying on compiler vectorization.

Ignoring a few special cases, what `memset` and auto-vectorized assignment loops do under the hood is they just [move](#) 32-byte blocks of data with [SIMD instructions](#):

```
const __m256i zeros = _mm256_set1_epi32(0);
```

```

for (int i = 0; i + 7 < N; i += 8)
    _mm256_store_si256((__m256i*) &a[i], zeros);

```

We can replace the usual vector store intrinsic with a *non-temporal* one:

```

const __m256i zeros = _mm256_set1_epi32(0);

for (int i = 0; i + 7 < N; i += 8)
    _mm256_stream_si256((__m256i*) &a[i], zeros);

```

Non-temporal memory reads or writes are a way to tell the CPU that we won't be needing the data that we have just accessed in the future, so there is no need to read the data back after a write.

On the one hand, if the array is small enough to fit into the cache, and we actually access it some short time after, this has a negative effect because we have to read entirely it from the RAM (or, in this case, we have to *write* it into the RAM instead of using a locally cached version). And on the other, this prevents read-backs and lets us use the memory bus more efficiently.

In fact, the performance increase in the case of the RAM is even more than 2x and faster than the read-only benchmark. This happens because:

- the memory controller doesn't have to switch the bus between read and write modes this way;
- the instruction sequence becomes simpler, allowing for more pending memory instructions;
- and, most importantly, the memory controller can simply "fire and forget" non-temporal write requests — while for reads, it needs to remember what to do with the data once it arrives (similar to connection handles in networking software).

Theoretically, both requests should use the same bandwidth: a read request sends an address and gets data, and a non-temporal write request sends an address *with* data and gets nothing. Not accounting for the direction, we transmit the same data, but the read cycle will be longer because it needs to wait for the data to be fetched. Since **there is a practical limit** on how many concurrent requests the memory system can handle, this difference in read/write cycle latency also results in the difference in their bandwidth.

Also, for these reasons, a single CPU core usually **can't fully saturate the memory bandwidth**.

The same technique generalizes to `memcpy`: it also just moves 32-byte blocks with SIMD load/store instructions, and it can be similarly made non-temporal, increasing the throughput twofold for large arrays. There is also a non-temporal load instruction (`_mm256_stream_load_si256`) for when you want to *read* without polluting cache (e.g., when you don't need the original array after a `memcpy`, but will need some data that you had accessed before calling it).

Memory Latency

Despite that **bandwidth** is a more complicated concept, it is much easier to observe and measure than latency: you can simply execute a long series of independent read or write queries, and the scheduler, having access to them in advance, reorders and overlaps them, hiding their latency and maximizing the total throughput.

To measure *latency*, we need to design an experiment where the CPU can't cheat by knowing the memory locations we will request in advance. One way to ensure this is to generate a random permutation of size N that corresponds to a cycle and then repeatedly follow the permutation:

```

int p[N], q[N];

// generating a random permutation
iota(p, p + N, 0);
random_shuffle(p, p + N);

// this permutation may contain multiple cycles,
// so instead we use it to construct another permutation with a single cycle
int k = p[N - 1];
for (int i = 0; i < N; i++)
    k = q[k] = p[i];

for (int t = 0; t < K; t++)
    for (int i = 0; i < N; i++)
        k = q[k];

```

Compared to linear iteration, it is *much* slower — by multiple orders of magnitude — to visit all elements of an array this way. Not only does it make SIMD impossible, but it also stalls the pipeline, creating a large traffic jam of instructions, all waiting for a single piece of data to be fetched from the memory.

This performance anti-pattern is known as *pointer chasing*, and it is very frequent in data structures, especially those written high-level languages that use lots of heap-allocated objects and pointers to them necessary for dynamic typing.

When talking about latency, it makes more sense to use cycles or nanoseconds rather than throughput units, so we replace this graph with its reciprocal:

Note that the cliffs on both graphs aren't as distinctive as they were for the bandwidth. This is because we still have some chance of hitting the previous layer of cache even if the array can't fit into it entirely.

Theoretical Latency

More formally, if there are k levels in the cache hierarchy with sizes s_i and latencies l_i , then, instead of being equal to the slowest access, their expected latency will be:

$$E[L] = \frac{s_1 \cdot l_1 + (s_2 - s_1) \cdot l_2 + \dots + (N - s_k) \cdot l_{RAM}}{N}$$

If we abstract away from all that happens before the slowest cache layer, we can reduce the formula to just this:

$$E[L] = \frac{N \cdot l_{last} - C}{N} = l_{last} - \frac{C}{N}$$

As N increases, the expected latency slowly approaches l_{last} , and if you squint hard enough, the graph of the throughput (reciprocal latency) should roughly look like if it is composed of a few transposed and scaled hyperbolas:

$$\begin{aligned}
E[L]^{-1} &= \frac{1}{l_{last} - \frac{C}{N}} \\
&= \frac{N}{N \cdot l_{last} - C} \\
&= \frac{1}{l_{last}} \cdot \frac{N + \frac{C}{l_{last}} - \frac{C}{l_{last}}}{N - \frac{C}{l_{last}}} \\
&= \frac{1}{l_{last}} \cdot \left(\frac{1}{N \cdot \frac{l_{last}}{C} - 1} + 1 \right) \\
&= \frac{1}{k \cdot (x - x_0)} + y_0
\end{aligned}$$

To get the actual latency numbers, we can iteratively apply the first formula to deduce l_1 , then l_2 , and so on. Or just look at the values right before the cliff — they should be within 10-15% of the true latency.

There are more direct ways to measure latency, including the use of [non-temporal reads](#), but this benchmark is more representable of practical access patterns.

Frequency Scaling

Similar to bandwidth, the latency of all CPU caches proportionally scales with its clock frequency, while the RAM does not. We can also observe this difference if we change the frequency by turning turbo boost on.

The graph starts making more sense if we plot it as a relative speedup.

You would expect 2x rates for array sizes that fit into CPU cache entirely, but then roughly equal for arrays stored in RAM. But this is not quite what is happening: there is a small, fixed-latency delay on lower clocked run even for RAM accesses. This happens because the CPU first has to check its cache before dispatching a read query to the main memory — to save RAM bandwidth for other processes that potentially need it.

Memory latency is also slightly affected by some details of the [virtual memory implementation](#) and [RAM-specific timings](#), which we will discuss later.

Cache Lines

The basic units of data transfer in the CPU cache system are not individual bits and bytes, but *cache lines*. On most architectures, the size of a cache line is 64 bytes, meaning that all memory is divided in blocks of 64 bytes, and whenever you request (read or write) a single byte, you are also fetching all its 63 cache line neighbors whether you want them or not.

To demonstrate this, we add a “step” parameter to our [incrementing loop](#). Now we only touch every D -th element:

```
for (int i = 0; i < N; i += D)
    a[i]++;
```

If we run it with $D = 1$ and $D = 16$, we can observe something interesting:

Performance is normalized by the total time to run benchmark, not the total number of elements incremented

Figure 2: Performance is normalized by the total time to run benchmark, not the total number of elements incremented

As the problem size grows, the graphs of the two loops meet, despite one doing 16 times less work than the other. This is because, in terms of cache lines, we are fetching the exact same memory in both loops, and the fact that the strided loop only needs one-sixteenth of it is irrelevant.

When the array fits into the L1 cache, the strided version completes faster — although not 16 but just two times as fast. This is because it only needs to do half the work: it only executes a single `inc DWORD PTR [rdx]` instruction for every 16 elements, while the original loop needed two 8-element [vector instructions](#) to process the same 16 elements. Both computations are bottlenecked by writing the result back: Zen 2 can only write one word per cycle — regardless of whether it is composed of one integer or eight.

When we change the step parameter to 8, the graphs equalize, as we now also need two increments and two write-backs per every 16 elements:

We can use this effect to minimize cache sharing in our [latency benchmark](#) to measure it more precisely. We need to *pad* the indices of a permutation so that each of them lies in its own cache line:

```
struct padded_int {
    int val;
    int padding[15];
};

padded_int q[N / 16];

// constructing a cycle from a random permutation
// ...

for (int i = 0; i < N / 16; i++)
    k = q[k].val;
```

Now, each index is much more likely to be kicked out of the cache by the time we loop around and request it again:

The important practical lesson when designing and analyzing memory-bound algorithms is to count the number of cache lines accessed and not just the total count of memory reads and writes.

Memory Sharing

Starting at some level of the hierarchy, the cache becomes *shared* between different cores. This reduces the total die area and lets you add more cores on a single chip but also poses some “noisy neighbor” problems as it limits the effective cache size and bandwidth available to a single execution thread.

On most CPUs, only the last layer of cache is shared, and not always in a uniform manner. On my machine, there are 8 physical cores, and the size of the L3 cache is 8M, but it is split into two

Cache hierarchy of my Ryzen 7 4700U generated by lstopo

Figure 3: Cache hierarchy of my Ryzen 7 4700U generated by lstopo

halves: two groups of 4 cores have access to their own 4M region of the L3 cache, and not all of it.

There are even more complex topologies, where accessing certain regions of memory takes non-constant time, different for each core (which is [sometimes unintended](#)). Such architectural feature is called *non-uniform memory access* (NUMA), and it is the case for multi-socket systems that have several separate CPU chips installed.

On Linux, the topology of the memory system can be retrieved with `lstopo`:

This has some important implications for parallel algorithms: the performance of multi-threaded memory accesses depends on which cores are running which execution threads. To demonstrate this, we will run the [bandwidth benchmarks](#) in parallel.

CPU Affinity

Instead of modifying the source code to run on multiple threads, we can simply run multiple identical processes with [GNU parallel](#). To control which cores are executing them, we set their *processor affinity* with `taskset`. This combined command runs 4 processes that can run on the first 4 cores of the CPU:

```
parallel taskset -c 0,1,2,3 ./run ::: {0..3}
```

Here is what we get when we change the number of processes running simultaneously:

You can now see that the performance decreases with more processes when the array exceeds the L2 cache (which is private to each core), as the cores start competing for the shared L3 cached and the RAM.

We specifically set all processes to run on the first 4 cores because they have a unified L3 cache. If some of the processes were to be scheduled on the other half of the cores, there would be less contention for the L3 cache. The operating system doesn't [monitor](#) such activities — what a process does is its own private business — so by default, it assigns threads to cores arbitrarily during execution, without caring about cache affinity and only taking into account the core load.

Let's run another benchmark, but now with pinning the processes to different 4-core groups that don't share L3 cache:

```
parallel taskset -c 0,1 ./run ::: {0..1} # L3 cache sharing
parallel taskset -c 0,4 ./run ::: {0..1} # no L3 cache sharing
```

It performs better — as if there were twice as much L3 cache and RAM bandwidth available:

These issues are especially tricky when benchmarking and are a huge source of noise when timing parallel applications.

Saturating Bandwidth

When looking at the RAM section of the first graph, it may seem that with more cores, the per-process throughput goes $\frac{1}{2}$, $\frac{1}{4}$, and so on, and the total bandwidth remains constant. But this

isn't quite true: the contention hurts, but a single CPU core usually can't saturate all of the RAM bandwidth.

If we plot it more carefully, we see that the total bandwidth actually increases with the number of cores — although not proportionally, and eventually approaches its theoretical maximum of ~42.4 GB/s:

Note that we still specify processor affinity: the k -threaded run uses the first k cores. This is why we have such a huge performance increase when switching from 4 cores to 5: you can have more RAM bandwidth if the requests go through separate L3 caches.

In general, to achieve maximum bandwidth, you should always split the threads of an application symmetrically.

Memory-Level Parallelism

Memory requests can overlap in time: while you wait for a read request to complete, you can send a few others, which will be executed concurrently with it. This is the main reason why [linear iteration](#) is so much faster than [pointer jumping](#): the CPU knows which memory locations it needs to fetch next and sends memory requests far ahead of time.

The number of concurrent memory operations is large but limited, and it is different for different types of memory. When designing algorithms and especially data structures, you may want to know this number, as it limits the amount of parallelism your computation can achieve.

To find this limit theoretically for a specific memory type, you can multiply its latency (time to fetch a cache line) by its bandwidth (number of cache lines fetched per second), which gives you the average number of memory operations in progress:

The latency of the L1/L2 caches is small, so there is no need for a long pipeline of pending requests, but larger memory types can sustain up to 25-40 concurrent read operations.

Direct Experiment

Let's try to measure available memory parallelism more directly by modifying our pointer chasing benchmark so that we loop around D separate cycles in parallel instead of just one:

```
const int M = N / D;
int p[M], q[D][M];

for (int d = 0; d < D; d++) {
    iota(p, p + M, 0);
    random_shuffle(p, p + M);
    k[d] = p[M - 1];
    for (int i = 0; i < M; i++)
        k[d] = q[d][k[d]] = p[i];
}

for (int i = 0; i < M; i++)
    for (int d = 0; d < D; d++)
        k[d] = q[d][k[d]];
```

Fixing the sum of the cycle lengths constant at a few select sizes and trying different D , we get slightly different results:

The L2 cache run is limited by ~6 concurrent operations, as predicted, but larger memory types all max out between 13 and 17. You can't make use of more memory lanes as there is a conflict over logical registers. When the number of lanes is fewer than the number of registers, you can issue just one read instruction per lane:

```
dec    edx
movsx rdi, DWORD PTR q[0+rdi*4]
movsx rsi, DWORD PTR q[1048576+rsi*4]
movsx rcx, DWORD PTR q[2097152+rcx*4]
movsx rax, DWORD PTR q[3145728+rax*4]
jne    .L9
```

But when it is over ~15, you have to use temporary memory storage:

```
mov    edx, DWORD PTR q[0+rdx*4]
mov    DWORD PTR [rbp-128+rax*4], edx
```

You don't always get to the maximum possible level of memory parallelism, but for most applications, a dozen concurrent requests are more than enough.

Prefetching

Taking advantage of the **free concurrency** available in memory hardware, it can be beneficial to *prefetch* data that is likely to be accessed next if its location can be predicted. This is easy to do when there are no **data of control hazards** in the pipeline and the CPU can just run ahead of the instruction stream and execute memory operations out of order.

But sometimes the memory locations aren't in the instruction stream, and yet they can still be predicted with high probability. In these cases, they can be prefetched by other means:

- Explicitly, by separately reading the next data word or any of the bytes in the same cache line, so that it is lifted in the cache hierarchy.
- Implicitly, by using simple access patterns such as linear iteration, which are detectable by the memory hardware that can start prefetching automatically.

Hiding memory latency is crucial for achieving performance, so in this section, we will look into prefetching techniques.

Hardware Prefetching

Let's modify the **pointer chasing** benchmark to show the effect of hardware prefetching. Now, we generate our permutation in a way that makes the CPU request consecutive cache lines when iterating over the permutation, but still accessing the elements inside a cache line in random order:

```
int p[15], q[N];

iota(p, p + 15, 1);

for (int i = 0; i + 16 < N; i += 16) {
    random_shuffle(p, p + 15);
```

```

int k = i;
for (int j = 0; j < 15; j++)
    k = q[k] = i + p[j];
q[k] = i + 16;
}

```

There is no point in making a graph because it would be just flat: the latency is 3ns regardless of the array size. Even though the instruction scheduler still can't tell what we are going to fetch next, the memory prefetcher can detect a pattern just by looking at the memory accesses and start loading the next cache line ahead of time, mitigating the latency.

Hardware prefetching is smart enough for most use cases, but it only detects simple patterns. You can iterate forward and backward over multiple arrays in parallel, perhaps with small-to-medium strides, but that's about it. For anything more complex, the prefetcher won't figure out what's happening, and we need to help it out ourselves.

Software Prefetching

The simplest way to do software prefetching is to load any byte in the cache line with the `mov` or any other memory instruction, but CPUs have a separate `prefetch` instruction that lifts a cache line without doing anything with it. This instruction isn't a part of the C or C++ standard, but is available in most compilers as the `__builtin_prefetch` intrinsic:

```
__builtin_prefetch(&a[k]);
```

It's quite hard to come up with a *simple* example when it can be useful. To make the pointer chasing benchmark benefit from software prefetching, we need to construct a permutation that at the same time loops around the whole array, can't be predicted by hardware prefetcher, and has easily computable next addresses.

Luckily, the [linear congruential generator](#) has the property that if the modulus n is a prime number, then the period of the generator will be exactly n . So we get all the properties we need if we use a permutation generated by the LCG with the current index as its state:

```

const int n = find_prime(N); // largest prime not exceeding N

for (int i = 0; i < n; i++)
    q[i] = (2 * i + 1) % n;

```

When we run it, the performance matches a normal random permutation. But now we get the ability to peek ahead:

```

int k = 0;

for (int t = 0; t < K; t++) {
    for (int i = 0; i < n; i++) {
        __builtin_prefetch(&q[(2 * k + 1) % n]);
        k = q[k];
    }
}

```

There is some overhead to computing the next address, but for arrays large enough, it is almost two times faster:

Interestingly, we can prefetch more than just one element ahead, making use of this pattern in the LCG function:

$$\begin{aligned}f(x) &= 2 \cdot x + 1 \\f^2(x) &= 4 \cdot x + 2 + 1 \\f^3(x) &= 8 \cdot x + 4 + 2 + 1 \\&\dots \\f^k(x) &= 2^k \cdot x + (2^k - 1)\end{aligned}$$

Hence, to load the D-th element ahead, we can do this:

```
--builtin_prefetch(&q[((1 << D) * k + (1 << D) - 1) % n]);
```

If we execute this request on every iteration, we will be simultaneously prefetching D elements ahead on average, increasing the throughput by D times. Ignoring some issues such as the integer overflow when D is too large, we can reduce the average latency arbitrarily close to the cost of computing the next index (which, in this case, is dominated by the [modulo operation](#)).

Note that this is an artificial example, and you actually fail more often than not when trying to insert software prefetching into practical programs. This is largely because you need to issue a separate memory instruction that may compete for resources with the others. At the same time, hardware prefetching is 100% harmless as it only activates when the memory and cache buses are not busy.

You can also specify a specific level of cache the data needs to be brought to when doing software prefetching — when you aren't sure if you will be using it and don't want to kick out what is already in the L1 cache. You can use it with the `_mm_prefetch` intrinsic, which takes an integer value as the second parameter, specifying the cache level. This is useful in combination with [non-temporal loads and stores](#).

Alignment and Packing

The fact that the memory is partitioned into 64B [cache lines](#) makes it difficult to operate on data words that cross a cache line boundary. When you need to retrieve some primitive type, such as a 32-bit integer, you really want to have it located on a single cache line — both because retrieving two cache lines requires more memory bandwidth and stitching the results in hardware requires precious transistor space.

This aspect heavily influences algorithm designs and how compilers choose the memory layout of data structures.

Aligned Allocation

By default, when you allocate an array of some primitive type, you are guaranteed that the addresses of all elements are a multiple of their size, which ensures that they only span a single cache line. For example, you are guaranteed the address of the first and every other element of an `int` array is a multiple of 4 bytes (`sizeof int`).

Sometimes you need to ensure that this minimum alignment is higher. For example, many SIMD applications read and write data in blocks of 32 bytes, and it is [crucial for performance](#) that these 32 bytes belong to the same cache line. In such cases, you can use the `alignas` specifier when defining a static array variable:

```
alignas(32) float a[n];
```

To allocate a memory-aligned array dynamically, you can use `std::aligned_alloc`, which takes the alignment value and the size of an array in bytes and returns a pointer to the allocated memory — just like the `new` operator does:

```
void *a = std::aligned_alloc(32, 4 * n);
```

You can also align memory to sizes [larger than the cache line](#). The only restriction is that the size parameter must be an integral multiple of alignment.

You can also use the `alignas` specifier when defining a `struct`:

```
struct alignas(64) Data {  
    // ...  
};
```

Whenever an instance of `Data` is allocated, it will be at the beginning of a cache line. The downside is that the effective size of the structure will be rounded up to the nearest multiple of 64 bytes. This has to be done so that, e.g., when allocating an array of `Data`, not just the first element is properly aligned.

Structure Alignment

This issue becomes more complicated when we need to allocate a group of non-uniform elements, which is the case for structures. Instead of playing Tetris trying to rearrange the members of a `struct` so that each of them is within a single cache line — which isn't always possible as the structure itself doesn't have to be placed on the start of a cache line — most C/C++ compilers also rely on the mechanism of memory alignment.

Structure alignment similarly ensures that the address of all its member primitive types (`char`, `int`, `float*`, etc) are multiples of their size, which automatically guarantees that each of them only spans one cache line. It achieves that by:

- *padding*, if necessary, each structure member with a variable number of blank bytes to satisfy the alignment requirement of the next member;
- setting the alignment requirement of the structure itself to the maximum of the alignment requirements of its member types, so that when an array of the structure type is allocated or it is used as a member type in another structure, the alignment requirements of all its primitive types are satisfied.

For better understanding, consider the following toy example:

```
struct Data {  
    char a;  
    short b;  
    int c;
```

```

    char d;
};


```

When stored succinctly, this structure needs a total of $1 + 2 + 4 + 1 = 8$ bytes per instance, but even assuming that the whole structure has the alignment of 4 bytes (its largest member, `int`), only `a` will be fine, while `b`, `c` and `d` are not size-aligned and potentially cross a cache line boundary.

To fix this, the compiler inserts some unnamed members so that each next member gets the right minimum alignment:

```

struct Data {
    char a;      // 1 byte
    char x[1];   // 1 byte for the following "short" to be aligned on a 2-byte boundary
    short b;     // 2 bytes
    int c;       // 4 bytes (largest member, setting the alignment of the whole structure)
    char d;      // 1 byte
    char y[3];   // 3 bytes to make total size of the structure 12 bytes (divisible by 4)
};

// sizeof(Data) = 12
// alignof(Data) = alignof(int) = sizeof(int) = 4

```

This potentially wastes space but saves a lot of CPU cycles. This trade-off is mostly beneficial, so structure alignment is enabled by default in most compilers.

Optimizing Member Order

Padding is only inserted before a not-yet-aligned member or at the end of the structure. By changing the ordering of members in a structure, it is possible to change the required number of padding bytes and the total size of the structure.

In the previous example, we could reorder the structure members like this:

```

struct Data {
    int c;
    short b;
    char a;
    char d;
};


```

Now, each of them is aligned without any padding, and the size of the structure is just 8 bytes. It seems stupid that the size of a structure and consequently its performance depends on the order of definition of its members, but this is required for binary compatibility.

As a rule of thumb, place your type definitions from largest data types to smallest — this greedy algorithm is guaranteed to work unless you have some weird non-power-of-two type sizes such as the [10-byte long double](#)¹.

¹The 80-bit `long double` takes *at least* 10 bytes, but the exact format is up to the compiler — for example, it may pad it to 12 or 16 bytes to minimize alignment issues (64-bit GCC and Clang use 16 bytes by default; you can override this by specifying one of `-mlong-double-64/80/128` or `-m96/128bit-long-double` options).

Structure Packing

If you know what you are doing, you can disable structure padding and pack your data as tight as possible.

You have to ask the compiler to do it, as such functionality is not a part of neither C nor C++ standard yet. In GCC and Clang, this is done with the `packed` attribute:

```
struct __attribute__((packed)) Data {
    long long a;
    bool b;
};
```

This makes the instances of `Data` take just 9 bytes instead of the 16 required by alignment, at the cost of possibly fetching two cache lines to reads its elements.

Bit fields

You can also use packing along with *bit fields*, which allow you to explicitly fix the size of a member in bits:

```
struct __attribute__((packed)) Data {
    char a;      // 1 byte
    int b : 24; // 3 bytes
};
```

This structure takes 4 bytes when packed and 8 bytes when padded. The number of bits a member has doesn't have to be a multiple of 8, and neither does the total structure size. In an array of `Data`, the neighboring elements will be “merged” in the case of a non-whole number of bytes. It also allows you to set a width that exceeds the base type, which acts as padding — although it throws a warning in the process.

This feature is not so widespread because CPUs don't have 3-byte arithmetic or things like that and has to do some inefficient byte-by-byte conversion during loading:

```
int load(char *p) {
    char x = p[0], y = p[1], z = p[2];
    return (x << 16) + (y << 8) + z;
}
```

The overhead is even larger when there is a non-whole byte — it needs to be handled with a shift and an and-mask.

This procedure can be optimized by loading a 4-byte `int` and then using a mask to discard its highest bits.

```
int load(int *p) {
    int x = *p;
    return x & ((1<<24) - 1);
}
```

Compilers usually don't do that because it's technically not legal: that 4th byte may be on a memory page that you don't own, so the operating system won't let you load it even if you are going to discard it right away.

RAM & CPU Caches

In the [previous chapter](#), we studied computer memory from a theoretical standpoint, using the [external memory model](#) to estimate the performance of memory-bound algorithms.

While the external memory model is more or less accurate for computations involving HDDs and network storage, where cost of arithmetic operations on in-memory values is negligible compared to external I/O operations, it is too imprecise for lower levels in the cache hierarchy, where the costs of these operations become comparable.

To perform more fine-grained optimization of in-memory algorithms, we have to start taking into account the many specific details of the CPU cache system. And instead of studying loads of boring Intel documents with dry specs and theoretically achievable limits, we will estimate these parameters experimentally by running numerous small benchmark programs with access patterns that resemble the ones that often occur in practical code.

Experimental Setup

As before, I will be running all experiments on Ryzen 7 4700U, which is a “Zen 2” CPU with the following main cache-related specs:

- 8 physical cores (without hyper-threading) clocked at 2GHz (and 4.1GHz in boost mode — [which we disable](#));
- 256K of 8-way set associative L1 data cache or 32K per core;
- 4M of 8-way set associative L2 cache or 512K per core;
- 8M of 16-way set associative L3 cache, [shared](#) between 8 cores;
- 16GB (2x8G) of DDR4 RAM @ 2667MHz.

You can compare it with your own hardware by running `dmidecode -t cache` or `lshw -class memory` on Linux or by installing [CPU-Z](#) on Windows. You can also find additional details about the CPU on [WikiChip](#) and [7-CPU](#). Not all conclusions will generalize to every CPU platform in existence.

Due to difficulties in [preventing the compiler from optimizing away unused values](#), the code snippets in this article are slightly simplified for exposition purposes. Check the [code repository](#) if you want to reproduce them yourself.

Acknowledgements

This chapter is inspired by “[Gallery of Processor Cache Effects](#)” by Igor Ostrovsky and “[What Every Programmer Should Know About Memory](#)” by Ulrich Drepper, both of which can serve as good accompanying readings.

Pointer Alternatives

In the [pointer chasing benchmark](#), for simplicity, we didn’t use actual pointers, but integer indices relative to a base address:

```
for (int i = 0; i < N; i++)
    k = q[k];
```

The memory addressing operator on x86 is fused with the address computation, so the `k = q[k]` line folds into just a single terse instruction that also does multiplication by 4 and addition under the hood:

```
mov rax, DWORD PTR q[0+rax*4]
```

Although fully fused, these additional computations add some delay to memory operations. The latency of an L1 fetch is either 4 or 5 cycles — the latter being the case if we need to perform a complex computation of the address. For this reason, the permutation benchmark measures 3ns or 6 cycles per jump: 4+1 for the read and address computation and another one to move the result to the right register.

Pointers

We can make our benchmark run slightly faster if we replace “fake pointers” — indices — with actual pointers.

There are some syntactical issues in getting “pointer to pointer to pointer...” constructions to work, so instead we will define a struct that just wraps a pointers to its own type — this is how most pointer chasing works anyway:

```
struct node { node* ptr; };
```

Now we randomly fill our array with pointers and chase them instead:

```
node* k = q + p[N - 1];  
  
for (int i = 0; i < N; i++)  
    k = k->ptr = q + p[i];  
  
for (int i = 0; i < N; i++)  
    k = k->ptr;
```

This code now runs in 2ns / 4 cycles for arrays that fit in the L1 cache. Why not 4+1=5? Because Zen 2 has an interesting feature that allows zero-latency reuse of data accessed just by address, so the “move” here is transparent, resulting in whole two cycles saved.

Unfortunately, there is a problem with it on 64-bit systems as the pointers become twice as large, making the array spill out of cache much sooner compared to using a 32-bit index. The latency-versus-size graph looks like if it was shifted by one power of two to the left — exactly like it should:

This problem is mitigated by switching to the 32-bit mode:

You need to go through some trouble getting 32-bit libs to get this running on a computer made in this century, but this shouldn’t pose other problems unless you need to interoperate with 64-bit software or access more than 4G of RAM

Bit Fields

The fact that on larger problem sizes the performance is bottlenecked by memory rather than CPU lets us try something even more strange: we can use less than 4 bytes for storing indices. This can be done with bit fields:

```
struct __attribute__((packed)) node { int idx : 24; };
```

You don't need to do anything else other than defining a structure for the bit field — the compiler handles the 3-byte integer all by itself:

```
int k = p[N - 1];  
  
for (int i = 0; i < N; i++) {  
    k = q[k].idx = p[i];  
  
for (int i = 0; i < N; i++) {  
    k = q[k].idx;
```

This code measures at 6.5ns for the L1 cache. There is some room for improvement as the default conversion procedure chosen by the compiler is suboptimal. We could manually load a 4-byte integer and truncate it ourselves (we also need to add one more element to the `q` array to ensure we own that extra one byte of memory):

```
k = *((int*) (q + k));  
k &= ((1<<24) - 1);
```

It now runs in 4ns, and produces the following graph:

If you zoom close enough ([the graph is an svg](#)), you'll see that the pointers win on very small arrays, then starting from around the L2-L3 cache boundary our custom bit fields take over, and for very large arrays it doesn't matter because we never hit cache anyway.

This isn't a kind of optimization that can give you a 5x improvement, but it's still something to try when all the other resources are exhausted.

Cache Associativity

Consider a strided incrementing loop over an array of size $N = 2^{21}$ with a fixed step size of 256:

```
for (int i = 0; i < N; i += 256)  
    a[i]++;
```

And then this one, with the step size of 257:

```
for (int i = 0; i < N; i += 257)  
    a[i]++;
```

Which one will be faster to finish? There are several considerations that come to mind:

- At first, you think that there shouldn't be much difference, or maybe that the second loop is $\frac{257}{256}$ times faster or so because it does fewer iterations in total.
- Then you recall that 256 is a nice round number, which may have something to do with SIMD or the memory system, so maybe the first one is faster.

But the right answer is very counterintuitive: the second loop is faster — and by a factor of 10.

This isn't just a single bad step size. The performance degrades for all indices that are multiples of large powers of two:

The array size is normalized so that the total number of iterations is constant

Figure 4: The array size is normalized so that the total number of iterations is constant

Fully associative cache

Figure 5: Fully associative cache

There is no vectorization or anything, and the two loops produce the same assembly except for the step size. This effect is due only to the memory system, in particular to a feature called *cache associativity*, which is a peculiar artifact of how CPU caches are implemented in hardware.

Hardware Caches

When we were studying the memory system [theoretically](#), we discussed different ways one can [implement cache eviction policies](#) in software. One particular strategy we focused on was the *least recently used* (LRU) policy, which is simple and effective but still requires some non-trivial data manipulation.

In the context of hardware, such scheme is called *fully associative cache*: we have M cells, each capable of holding a cache line corresponding to any of the N total memory locations, and in case of contention, the one not accessed the longest gets kicked out and replaced with the new one.

The problem with fully associative cache is that implementing the “find the oldest cache line among millions” operation is pretty hard to do in software and just unfeasible in hardware. You can make a fully associative cache that has 16 entries or so, but managing hundreds of cache lines already becomes either prohibitively expensive or so slow that it’s not worth it.

We can resort to another, much simpler approach: just map each block of 64 bytes in RAM to a single cache line which it can occupy. Say, if we have 4096 blocks in memory and 64 cache lines for them, then each cache line at any time stores the contents of one of $\frac{4096}{64} = 64$ different blocks.

A direct-mapped cache is easy to implement doesn’t require storing any additional meta-information associated with a cache line except its tag (the actual memory location of a cached block). The disadvantage is that the entries can be kicked out too quickly — for example, when bouncing between two addresses that map to the same cache line — leading to lower overall cache utilization.

For that reason, we settle for something in-between direct-mapped and fully associative caches: the *set-associative cache*. It splits the address space into equal groups, which separately act as small fully-associative caches.

Associativity is the size of these sets, or, in other words, how many different cache lines each data block can be mapped to. Higher associativity allows for more efficient utilization of cache but also increases the cost.

For example, on [my CPU](#), the L3 cache is 16-way set-associative, and there are 4MB available to a single core. This means that there are in total $\frac{2^{22}}{2^6} = 2^{16}$ cache lines, which are split into $\frac{2^{16}}{16} = 2^{12}$ groups, each acting as a fully associative cache of their own ($\frac{1}{2^{12}}$)-th fraction of the RAM.

Direct-mapped cache

Figure 6: Direct-mapped cache

Set-associative cache (2-way associative)

Figure 7: Set-associative cache (2-way associative)

Address composition for a 64-entry 2-way set-associative cache

Figure 8: Address composition for a 64-entry 2-way set-associative cache

Most other CPU caches are also set-associative, including the non-data ones such as the instruction cache and the TLB. The exceptions are small specialized caches that only house 64 or fewer entries — these are usually fully associative.

Address Translation

There is only one ambiguity remaining: how exactly the cache line mapping is done.

If we implemented set-associative cache in software, we would compute some hash function of the memory block address and then use its value as the cache line index. In hardware, we can't really do that because it is too slow: for example, for the L1 cache, the latency requirement is 4 or 5 cycles, and even [taking a modulo](#) takes around 10-15 cycles, let alone something more sophisticated.

Instead, the hardware uses the lazy approach. It takes the memory address that needs to be accessed and splits it into three parts — from lower bits to higher:

- *offset* — the index of the word within a 64B cache line ($\log_2 64 = 6$ bits);
- *index* — the index of the cache line set (the next 12 bits as there are 2^{12} cache lines in the L3 cache);
- *tag* — the rest of the memory address, which is used to tell the memory blocks stored in the cache lines apart.

In other words, all memory addresses with the same “middle” part map to the same set.

This makes the cache system simpler and cheaper to implement but also susceptible to certain bad access patterns.

Pathological Mappings

Now, where were we? Oh, yes: the reason why iteration with strides of 256 causes such a terrible slowdown.

When we jump over 256 integers, the pointer always increments by $1024 = 2^{10}$, and the last 10 bits remain the same. Since the cache system uses the lower 6 bits for the offset and the next 12 for the cache line index, we are essentially using just $2^{12-(10-6)} = 2^8$ different sets in the L3 cache instead of 2^{12} , which has the effect of shrinking our L3 cache by a factor of $2^4 = 16$. The array stops fitting into the L3 cache ($N = 2^{21}$) and spills into the order-of-magnitude slower RAM, which causes the performance to decrease.

Performance issues caused by cache associativity effects arise with remarkable frequency in algorithms because, for multiple reasons, programmers just love using powers of two when indexing arrays:

- It is easier to calculate the address for multi-dimensional array accesses if the last dimension is a power of two, as it only requires a binary shift instead of a multiplication.

- It is easier to calculate modulo a power of two, as it can be done with a single bitwise `and`.
- It is convenient and often even necessary to use power-of-two problem sizes in divide-and-conquer algorithms.
- It is the smallest integer exponent, so using the sequence of increasing powers of two as problem sizes are a popular choice when benchmarking memory-bound algorithms.
- Also, more natural powers of ten are by transitivity divisible by a slightly lower power of two.

This especially often applies to implicit data structures that use a fixed memory layout. For example, [binary searching](#) over arrays of size 2^{20} takes about $\sim 360\text{ns}$ per query while searching over arrays of size $(2^{20} + 123)$ takes $\sim 300\text{ns}$. When the array size is a multiple of a large power of two, then the indices of the “hottest” elements, the ones we likely request on the first dozen or so iterations, will also be divisible by some large powers of two and map to the same cache line — kicking each other out and causing a $\sim 20\%$ performance decrease.

Luckily, such issues are more of an anomaly rather than serious problems. The solution is usually simple: avoid iterating in powers of two, make the last dimensions of multi-dimensional arrays a slightly different size or use any other method to insert “holes” in the memory layout, or create some seemingly random bijection between the array indices and the locations where the data is actually stored.

Memory Paging

Consider [yet again](#) the strided incrementing loop:

```
const int N = (1 << 13);
int a[D * N];

for (int i = 0; i < D * N; i += D)
    a[i] += 1;
```

We change the stride D and increase the array size proportionally so that the total number of iterations N remains constant. As the total number of memory accesses also remains constant, for all $D \geq 16$, we should be fetching exactly N cache lines — or $64 \cdot N = 2^6 \cdot 2^{13} = 2^{19}$ bytes, to be exact. This precisely fits into the L2 cache, regardless of the step size, and the throughput graph should look flat.

This time, we consider a larger range of D values, up to 1024. Starting from around 256, the graph is definitely not flat:

This anomaly is also due to the cache system, although the standard L1-L3 data caches have nothing to do with it. [Virtual memory](#) is at fault, in particular the *translation lookaside buffer* (TLB), which is a cache responsible for retrieving the physical addresses of the virtual memory pages.

On [my CPU](#), there are two levels of TLB:

- The L1 TLB has 64 entries, and if the page size is 4K, then it can handle $64 \times 4K = 512K$ of active memory without going to the L2 TLB.
- The L2 TLB has 2048 entries, and it can handle $2048 \times 4K = 8M$ of memory without going to the page table.

How much memory is allocated when D becomes equal to 256? You’ve guessed it: $8K \times 256 \times 4B =$

$8M$, exactly the limit of what the L2 TLB can handle. When D gets larger than that, some requests start getting redirected to the main page table, which has a large latency and very limited throughput, which bottlenecks the whole computation.

Changing Page Size

That 8MB of slowdown-free memory seems like a very tight restriction. While we can't change the characteristics of the hardware to lift it, we *can* increase the page size, which would in turn reduce the pressure on the TLB capacity.

Modern operating systems allow us to set the page size both globally and for individual allocations. CPUs only support a defined set of page sizes — mine, for example, can use either 4K or 2M pages. Another typical page size is 1G — it is usually only relevant for server-grade hardware with hundreds of gigabytes of RAM. Anything over the default 4K is called *huge pages* on Linux and *large pages* on Windows.

On Linux, there is a special system file that governs the allocation of huge pages. Here is how to make the kernel give you huge pages on every allocation:

```
$ echo always > /sys/kernel/mm/transparent_hugepage/enabled
```

Enabling huge pages globally like this isn't always a good idea because it decreases memory granularity and raises the minimum memory that a process consumes — and some environments have more processes than free megabytes of memory. So, in addition to `always` and `never`, there is a third option in that file:

```
$ cat /sys/kernel/mm/transparent_hugepage/enabled
always [madvise] never
```

`madvise` is a special system call that lets the program advise the kernel on whether to use huge pages or not, which can be used for allocating huge pages on-demand. If it is enabled, you can use it in C++ like this:

```
#include <sys/mman.h>

void *ptr = std::aligned_alloc(page_size, array_size);
madvise(ptr, array_size, MADV_HUGE PAGE);
```

You can only request a memory region to be allocated using huge pages if it has the corresponding alignment.

Windows has similar functionality. Its memory API combines these two functions into one:

```
#include "memoryapi.h"

void *ptr = VirtualAlloc(NULL, array_size,
                         MEM_RESERVE | MEM_COMMIT | MEM_LARGE_PAGES, PAGE_READWRITE);
```

In both cases, `array_size` should be a multiple of `page_size`.

Impact of Huge Pages

Both variants of allocating huge pages immediately flatten the curve:

Enabling huge pages also improves **latency** by up to 10-15% for arrays that don't fit into the L2 cache:

In general, enabling huge pages is a good idea when you have any sort of sparse reads, as they usually slightly improve and (**almost**) never hurt performance.

That said, you shouldn't rely on huge pages if possible, as they aren't always available due to either hardware or computing environment restrictions. There are **many other reasons** why grouping data accesses spatially may be beneficial, which automatically solves the paging problem.

AoS and SoA

It is often beneficial to group together the data you need to fetch at the same time: preferably, on the same or, if that isn't possible, neighboring cache lines. This improves the **spatial locality** of your memory accesses, positively impacting the performance of memory-bound algorithms.

To demonstrate the potential effect of doing this, we modify the **pointer chasing** benchmark so that the next pointer is computed using not one, but a variable number of fields (D).

Experiment

The first approach will locate these fields together as the rows of a two-dimensional array. We will refer to this variant as *array of structures* (AoS):

```
const int M = N / D; // # of memory accesses
int p[M], q[M][D];

iota(p, p + M, 0);
random_shuffle(p, p + M);

int k = p[M - 1];

for (int i = 0; i < M; i++)
    q[k][0] = p[i];

for (int j = 1; j < D; j++)
    q[i][0] ^= (q[j][i] = rand());

k = q[k][0];
}

for (int i = 0; i < M; i++) {
    int x = 0;
    for (int j = 0; j < D; j++)
        x ^= q[k][j];
    k = x;
}
```

And in the second approach, we will place them separately. The laziest way to do this is to transpose the two-dimensional array q and swap the indices in all its subsequent accesses:

Note the logarithmic scale

Figure 9: Note the logarithmic scale

```
int q[D][M];  
// ^--^
```

By analogy, we call this variant *structure of arrays* (SoA). Obviously, for large D 's, it performs much worse:

The performance of both variants grows linearly with D , but AoS needs to fetch up to 16 times fewer total cache lines as the data is stored sequentially. Even when $D = 64$, the additional time it takes to process the other 63 values is less than the latency of the first fetch.

You can also see the spikes at the powers of two. AoS performs slightly better because it can compute [horizontal xor-sum](#) faster with SIMD. In contrast, SoA performs much worse, but this isn't about D , but about $\lfloor N/D \rfloor$, the size of the second dimension, being a large power of two: this causes a pretty complicated [cache associativity](#) effect.

Temporary Storage Contention

At first, it seems like there shouldn't be any cache issues as $N = 2^{23}$ and the array is just too big to fit into the L3 cache in the first place. The nuance is that to process a number of elements from different memory locations in parallel, you still need some space to store them temporarily. You can't simply use registers as there aren't enough of them, so they need to be stored in the cache even though in just a microsecond you won't be needing them.

Therefore, when N / D is a large power of two, and we are iterating over the array $q[D][N / D]$ along the first index, some of the memory addresses we temporarily need will map to the same cache line — and as there isn't enough space there, many of them will have to be re-fetched from the upper layers of the memory hierarchy.

Here is another head-scratcher: if we enable [huge pages](#), it expectedly makes the total latency 10-15% lower for most values of D , but for $D = 64$, it makes things ten times worse:

I doubt that even the engineers who design memory controllers can explain what's happening right off the bat.

In short, the difference is because, unlike the L1/L2 caches that are private to each core, the L3 cache has to use *physical* memory addresses instead of *virtual* ones for synchronization between different cores sharing the cache.

When we are using 4K memory pages, the virtual addresses get somewhat arbitrarily dispersed over the physical memory, which makes the cache associativity problem less severe: the physical addresses will have the same remainder modulo 4K bytes, and not N / D as for the virtual addresses. When we specifically require huge pages, this maximum alignment limit increases to 2M, and the cache lines receive much more contention.

This is the only example I know when enabling huge pages makes performance worse, let alone by a factor of ten.

Padded AoS

As long as we are fetching the same number of cache lines, it doesn't matter where they are located, right? Let's test it and switch to **padded integers** in the AoS code:

```
struct padded_int {
    int val;
    int padding[15];
};

const int M = N / D / 16;
padded_int q[M][D];
```

Other than that, we are still calculating the xor-sum of D padded integers. We fetch exactly D cache lines, but this time sequentially. The running time shouldn't be different from SoA, but this isn't what happens:

The running time is about \sim lower for $D = 63$, but this only applies to arrays that exceed the L3 cache. If we fix D and change N , you can see that the padded version performs slightly worse on smaller arrays because there are less opportunities for random **cache sharing**:

As the performance on smaller arrays sizes is not affected, this clearly has something to do with how RAM works.

RAM-Specific Timings

From the performance analysis point of view, all data in RAM is physically stored in a two-dimensional array of tiny capacitor cells, which is split into rows and columns. To read or write any cell, you need to perform one, two, or three actions:

1. Read the contents of a row in a *row buffer*, which temporarily discharges the capacitors.
2. Read or write a specific cell in this buffer.
3. Write the contents of a row buffer back into the capacitors so that the data is preserved and the row buffer can be used for other memory accesses.

Here is the punchline: you don't have to perform steps 1 and 3 between two memory accesses that correspond to the same row — you can just use the row buffer as a temporary cache. These three actions take roughly the same time, so this optimization makes long sequences of row-local accesses run thrice as fast compared to dispersed access patterns.

The size of the row differs depending on the hardware, but it is usually somewhere between 1024 and 8192 bytes. So even though the padded AoS benchmark places each element in a separate cache line, they are still very likely to be on the same RAM row, and the whole read sequence runs in roughly \sim of the time plus the latency of the first memory access.

SIMD Parallelism

Intrinsics and Vector Types

The most low-level way to use SIMD is to use the assembly vector instructions directly — they aren't different from their scalar equivalents at all — but we are not going to do that. Instead, we will use *intrinsic* functions mapping to these instructions that are available in modern C/C++ compilers.

In this section, we will go through the basics of their syntax, and in the rest of this chapter, we will use them extensively to do things that are actually interesting.

Setup

To use x86 intrinsics, we need to do a little groundwork.

First, we need to determine which extensions are supported by the hardware. On Linux, you can call `cat /proc/cpuinfo`, and on other platforms, you'd better go to WikiChip and look it up there using the name of the CPU. In either case, there should be a `flags` section that lists the codes of all supported vector extensions.

There is also a special CPUID assembly instruction that lets you query various information about the CPU, including the support of particular vector extensions. It is primarily used to get such information in runtime and avoid distributing a separate binary for each microarchitecture. Its output information is returned very densely in the form of feature masks, so compilers provide built-in methods to make sense of it. Here is an example:

```
#include <iostream>
using namespace std;

int main() {
    cout << __builtin_cpu_supports("sse") << endl;
    cout << __builtin_cpu_supports("sse2") << endl;
    cout << __builtin_cpu_supports("avx") << endl;
    cout << __builtin_cpu_supports("avx2") << endl;
    cout << __builtin_cpu_supports("avx512f") << endl;

    return 0;
}
```

Second, we need to include a header file that contains the subset of intrinsics we need. Similar to `<bits/stdc++.h>` in GCC, there is the `<x86intrin.h>` header that contains all of them, so we will just use that.

And last, we need to tell the compiler that the target CPU actually supports these extensions. This can be done either with `#pragma GCC target(...)` as we did before, or with the `-march=...` flag in the compiler options. If you are compiling and running the code on the same machine, you can set `-march=native` to auto-detect the microarchitecture.

In all further code examples, assume that they begin with these lines:

```
#pragma GCC target("avx2")
#pragma GCC optimize("O3")

#include <x86intrin.h>
#include <bits/stdc++.h>

using namespace std;
```

We will focus on AVX2 and the previous SIMD extensions in this chapter, which should be available on 95% of all desktop and server computers, although the general principles transfer on AVX512, Arm Neon, and other SIMD architectures just as well.

SIMD Registers

The most notable distinction between SIMD extensions is the support for wider registers:

- SSE (1999) added 16 128-bit registers called `xmm0` through `xmm15`.
- AVX (2011) added 16 256-bit registers called `ymm0` through `ymm15`.
- AVX512 (2017) added¹ 16 512-bit registers called `zmm0` through `zmm15`.

As you can guess from the naming, and also from the fact that 512 bits already occupy a full cache line, x86 designers are not planning to add wider registers anytime soon.

C/C++ compilers implement special *vector types* that refer to the data stored in these registers:

- 128-bit `__m128`, `__m128d` and `__m128i` types for single-precision floating-point, double-precision floating-point and various integer data respectively;
- 256-bit `__m256`, `__m256d`, `__m256i`;
- 512-bit `__m512`, `__m512d`, `__m512i`.

Registers themselves can hold data of any kind: these types are only used for type checking. You can convert a vector variable to another vector type the same way you would normally convert any other type, and it won't cost you anything.

SIMD Intrinsics

Intrinsics are just C-style functions that do something with these vector data types, usually by simply calling the associated assembly instruction.

For example, here is a cycle that adds together two arrays of 64-bit floating-point numbers using AVX intrinsics:

```
double a[100], b[100], c[100];

// iterate in blocks of 4,
// because that's how many doubles can fit into a 256-bit register
for (int i = 0; i < 100; i += 4) {
    // load two 256-bit segments into registers
    __m256d x = _mm256_loadu_pd(&a[i]);
    __m256d y = _mm256_loadu_pd(&b[i]);

    // add 4+4 64-bit numbers together
    __m256d z = _mm256_add_pd(x, y);

    // write the 256-bit result into memory, starting with c[i]
    _mm256_storeu_pd(&c[i], z);
}
```

The main challenge of using SIMD is getting the data into contiguous fixed-sized blocks suitable for loading into registers. In the code above, we may in general have a problem if the length of the array is not divisible by the block size. There are two common solutions to this:

1. We can “overshoot” by iterating over the last incomplete segment either way. To make sure we don't segfault by trying to read from or write to a memory region we don't own, we need to pad the arrays to the nearest block size (typically with some “neutral” element, e.g., zero).

¹AVX512 also added 8 so-called *mask registers* named `k0` through `k7`, which are used for masking and blending data. We are not going to cover them and will mostly use AVX2 and previous standards.

2. Make one iteration less and write a little loop in the end that calculates the remainder normally (with scalar operations).

Humans prefer #1 because it is simpler and results in less code, and compilers prefer #2 because they don't really have another legal option.

Instruction References

Most SIMD intrinsics follow a naming convention similar to `_mm<size>_<action>_<type>` and correspond to a single analogously named assembly instruction. They become relatively self-explanatory once you get used to the assembly naming conventions, although sometimes it does seem like their names were generated by cats walking on keyboards (explain this: `punpcklqdq`).

Here are a few more examples, just so that you get the gist of it:

- `_mm_add_epi16`: add two 128-bit vectors of 16-bit *extended packed integers*, or simply said, `shorts`.
- `_mm256_acos_pd`: calculate elementwise arccos for 4 *packed doubles*.
- `_mm256_broadcast_sd`: broadcast (copy) a `double` from a memory location to all 4 elements of the result vector.
- `_mm256_ceil_pd`: round up each of 4 `doubles` to the nearest integer.
- `_mm256_cmpeq_epi32`: compare 8+8 packed `ints` and return a mask that contains ones for equal element pairs.
- `_mm256_blendv_ps`: pick elements from one of two vectors according to a mask.

As you may have guessed, there is a combinatorially very large number of intrinsics, and in addition to that, some instructions also have immediate values — so their intrinsics require compile-time constant parameters: for example, the floating-point comparison instruction has 32 different modifiers.

For some reason, there are some operations that are agnostic to the type of data stored in registers, but only take a specific vector type (usually 32-bit float) — you just have to convert to and from it to use that intrinsic. To simplify the examples in this chapter, we will mostly work with 32-bit integers (`epi32`) in 256-bit AVX2 registers.

A very helpful reference for x86 SIMD intrinsics is the Intel Intrinsics Guide, which has groupings by categories and extensions, descriptions, pseudocode, associated assembly instructions, and their latency and throughput on Intel microarchitectures. You may want to bookmark that page.

The Intel reference is useful when you know that a specific instruction exists and just want to look up its name or performance info. When you don't know whether it exists, this cheat sheet may do a better job.

Instruction selection. Note that compilers do not necessarily pick the exact instruction that you specify. Similar to the scalar `c = a + b` we discussed before, there is a fused vector addition instruction too, so instead of using 2+1+1=4 instructions per loop cycle, compiler rewrites the code above with blocks of 3 instructions like this:

```
vmovapd ymm1, YMMWORD PTR a[rax]
vaddpd ymm0, ymm1, YMMWORD PTR b[rax]
vmovapd YMMWORD PTR c[rax], ymm0
```

Sometimes, although quite rarely, this compiler interference makes things worse, so it is always a good idea to check the assembly and take a closer look at the emitted vector instructions (they usually start with a “v”).

Also, some of the intrinsics don't map to a single instruction but a short sequence of them, as a convenient shortcut: broadcasts and extracts are a notable example.

GCC Vector Extensions

If you feel like the design of C intrinsics is terrible, you are not alone. I've spent hundreds of hours writing SIMD code and reading the Intel Intrinsics Guide, and I still can't remember whether I need to type `_mm256`

or `__m256`.

Intrinsics are not only hard to use but also neither portable nor maintainable. In good software, you don't want to maintain different procedures for each CPU: you want to implement it just once, in an architecture-agnostic way.

One day, compiler engineers from the GNU Project thought the same way and developed a way to define your own vector types that feel more like arrays with some operators overloaded to match the relevant instructions.

In GCC, here is how you can define a vector of 8 integers packed into a 256-bit (32-byte) register:

```
typedef int v8si __attribute__((vector_size(32)));
// type ^  ^ typename           size in bytes ^
```

Unfortunately, this is not a part of the C or C++ standard, so different compilers use different syntax for that.

There is somewhat of a naming convention, which is to include size and type of elements into the name of the type: in the example above, we defined a “vector of 8 signed integers.” But you may choose any name you want, like `vec`, `reg` or whatever. The only thing you don't want to do is to name it `vector` because of how much confusion there would be because of `std::vector`.

The main advantage of using these types is that for many operations you can use normal C++ operators instead of looking up the relevant intrinsic.

```
v4si a = {1, 2, 3, 5};
v4si b = {8, 13, 21, 34};

v4si c = a + b;

for (int i = 0; i < 4; i++)
    printf("%d\n", c[i]);

c *= 2; // multiply by scalar

for (int i = 0; i < 4; i++)
    printf("%d\n", c[i]);
```

With vector types we can greatly simplify the “`a + b`” loop we implemented with intrinsics before:

```
typedef double v4d __attribute__((vector_size(32)));
v4d a[100/4], b[100/4], c[100/4];

for (int i = 0; i < 100/4; i++)
    c[i] = a[i] + b[i];
```

As you can see, vector extensions are much cleaner compared to the nightmare we have with intrinsic functions. Their downside is that there are some things that we may want to do are just not expressible with native C++ constructs, so we will still need intrinsics for them. Luckily, this is not an exclusive choice, because vector types support zero-cost conversion to the `_mm` types and back:

```
v8f x;
int mask = _mm256_movemask_ps((__m256) x)
```

There are also many third-party libraries for different languages that provide a similar capability to write portable SIMD code and also implement some, and just in general are nicer to use than both intrinsics and built-in vector types. Notable examples for C++ are Highway, Expressive Vector Engine, Vector Class Library, and xsimd.

Using a well-established SIMD library is recommended as it greatly improves the developer experience. In this book, however, we will try to keep close to the hardware and mostly use intrinsics directly, occasionally switching to the vector extensions for simplicity when we can.

Moving Data

If you took some time to study the reference, you may have noticed that there are essentially two major groups of vector operations:

1. Instructions that perform some elementwise operation (`+`, `*`, `<`, `acos`, etc.).
2. Instructions that load, store, mask, shuffle, and generally move data around.

While using the elementwise instructions is easy, the largest challenge with SIMD is getting the data in vector registers in the first place, with low enough overhead so that the whole endeavor is worthwhile.

Aligned Loads and Stores

Operations of reading and writing the contents of a SIMD register into memory have two versions each: `load` / `loadu` and `store` / `storeu`. The letter “*u*” here stands for “unaligned.” The difference is that the former ones only work correctly when the read / written block fits inside a single cache line (and crash otherwise), while the latter work either way, but with a slight performance penalty if the block crosses a cache line.

Sometimes, especially when the “inner” operation is very lightweight, the performance difference becomes significant (at least because you need to fetch two cache lines instead of one). As an extreme example, this way of adding two arrays together:

```
for (int i = 3; i + 7 < n; i += 8) {  
    __m256i x = _mm256_loadu_si256((__m256i*) &a[i]);  
    __m256i y = _mm256_loadu_si256((__m256i*) &b[i]);  
    __m256i z = _mm256_add_epi32(x, y);  
    _mm256_storeu_si256((__m256i*) &c[i], z);  
}
```

...is ~30% slower than its aligned version:

```
for (int i = 0; i < n; i += 8) {  
    __m256i x = _mm256_load_si256((__m256i*) &a[i]);  
    __m256i y = _mm256_load_si256((__m256i*) &b[i]);  
    __m256i z = _mm256_add_epi32(x, y);  
    _mm256_store_si256((__m256i*) &c[i], z);  
}
```

In the first version, assuming that arrays `a`, `b` and `c` are all 64-byte *aligned* (the addresses of their first elements are divisible by 64, and so they start at the beginning of a cache line), roughly half of reads and writes will be “bad” because they cross a cache line boundary.

Note that the performance difference is caused by the cache system and not by the instructions themselves. On most modern architectures, the `loadu` / `storeu` intrinsics should be equally as fast as `load` / `store` given that in both cases the blocks only span one cache line. The advantage of the latter is that they can act as free run time assertions that all reads and writes are aligned.

This makes it important to properly align arrays and other data on allocation, and it is also one of the reasons why compilers can’t always auto-vectorize efficiently. For most purposes, we only need to guarantee that any 32-byte SIMD block will not cross a cache line boundary, and we can specify this alignment with the `alignas` specifier:

```
alignas(32) float a[n];  
  
for (int i = 0; i < n; i += 8) {
```

```

__m256 x = _mm256_load_ps(&a[i]);
// ...
}

```

The built-in vector types already have corresponding alignment requirements and assume aligned memory reads and writes — so you are always safe when allocating an array of `v8si`, but when converting it from `int*` you have to make sure it is aligned.

Similar to the scalar case, many arithmetic instructions take memory addresses as operands — vector addition is an example — although you can't explicitly use it as an intrinsic and have to rely on the compiler. There are also a few other instructions for reading a SIMD block from memory, notably the non-temporal load and store operations that don't lift accessed data in the cache hierarchy.

Register Aliasing

The first SIMD extension, MMX, started quite small. It only used 64-bit vectors, which were conveniently aliased to the mantissa part of a 80-bit float so that there is no need to introduce a separate set of registers. As the vector size grew with later extensions, the same register aliasing mechanism used in general-purpose registers was adopted for the vector registers to maintain backward compatibility: `xmm0` is the first half (128 bits) of `ymm0`, `xmm1` is the first half of `ymm1`, and so on.

This feature, combined with the fact that the vector registers are located in the FPU, makes moving data between them and the general-purpose registers slightly complicated.

Extract and Insert

To *extract* a specific value from a vector, you can use `_mm256_extract_epi32` and similar intrinsics. It takes the index of the integer to be extracted as the second parameter and generates different instruction sequences depending on its value.

If you need to extract the first element, it generates the `vmovd` instruction (for `xmm0`, the first half of the vector):

```
vmovd eax, xmm0
```

For other elements of an SSE vector, it generates possibly slightly slower `vpextrd`:

```
vpextrd eax, xmm0, 1
```

To extract anything from the second half of an AVX vector, it first has to extract that second half, and then the scalar itself. For example, here is how it extracts the last (eighth) element,

```
vextracti128 xmm0, ymm0, 0x1
vpextrd    eax, xmm0, 3
```

There is a similar `_mm256_insert_epi32` intrinsic for overwriting specific elements:

```
mov      eax, 42

; v = _mm256_insert_epi32(v, 42, 0);
vpinsrd xmm2, xmm0, eax, 0
vinserti128 ymm0, ymm0, xmm2, 0x0

; v = _mm256_insert_epi32(v, 42, 7);
vextracti128 xmm1, ymm0, 0x1
vpinsrd    xmm2, xmm1, eax, 3
vinserti128 ymm0, ymm0, xmm2, 0x1
```

Takeaway: moving scalar data to and from vector registers is slow, especially when this isn't the first element.

Making Constants

If you need to populate not just one element but the entire vector, you can use the `_mm256_setr_epi32` intrinsic:

```
--m256 iota = _mm256_setr_epi32(0, 1, 2, 3, 4, 5, 6, 7);
```

The “r” here stands for “reversed” — from the CPU point of view, not for humans. There is also the `_mm256_set_epi32` (without “r”) that fills the values from the opposite direction. Both are mostly used to create compile-time constants that are then fetched into the register with a block load. If your use case is filling a vector with zeros, use the `_mm256_setzero_si256` instead: it `xor`s the register with itself.

In built-in vector types, you can just use normal braced initialization:

```
vec zero = {};
vec iota = {0, 1, 2, 3, 4, 5, 6, 7};
```

Broadcast

Instead of modifying just one element, you can also *broadcast* a single value into all its positions:

```
; __m256i v = _mm256_set1_epi32(42);
mov        eax, 42
vmovd    xmm0, eax
vpbroadcastd ymm0, xmm0
```

This is a frequently used operation, so you can also use a memory location:

```
; __m256 v = _mm256_broadcast_ss(&a[i]);
vbroadcastss ymm0, DWORD PTR [rdi]
```

When using built-in vector types, you can create a zero vector and add a scalar to it:

```
vec v = 42 + vec{};
```

Mapping to Arrays

If you want to avoid all this complexity, you can just dump the vector in memory and read its values back as scalars:

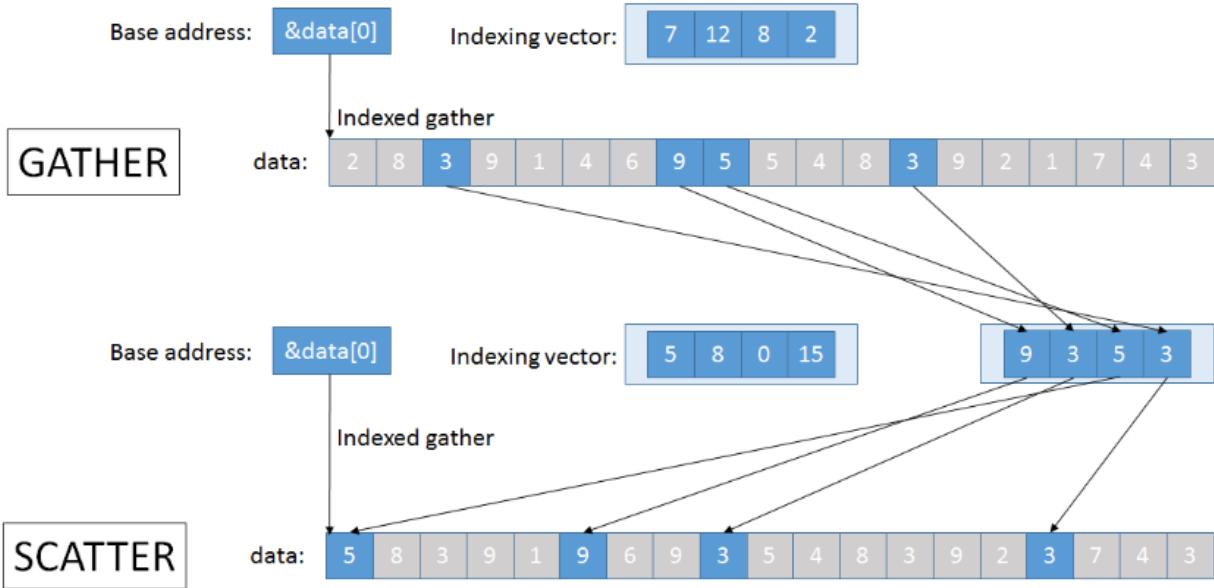
```
void print(__m256i v) {
    auto t = (unsigned*) &v;
    for (int i = 0; i < 8; i++)
        std::cout << std::bitset<32>(t[i]) << " ";
    std::cout << std::endl;
}
```

This may not be fast or technically legal (the C++ standard doesn’t specify what happens when you cast data like this), but it is simple, and I frequently use this code to print out the contents of a vector during debugging.

Non-Contiguous Load

Later SIMD extensions added special “gather” and “scatter instructions that read/write data non-sequentially using arbitrary array indices. These don’t work 8 times faster though and are usually limited by the memory rather than the CPU, but they are still helpful for certain applications such as sparse linear algebra.

Gather is available since AVX2, and various scatter instructions are available since AVX512.



Let's see if they work faster than scalar reads. First, we create an array of size N and Q random read queries:

```
int a[N], q[Q];

for (int i = 0; i < N; i++)
    a[i] = rand();

for (int i = 0; i < Q; i++)
    q[i] = rand() % N;
```

In the scalar code, we add the elements specified by the queries to a checksum one by one:

```
int s = 0;

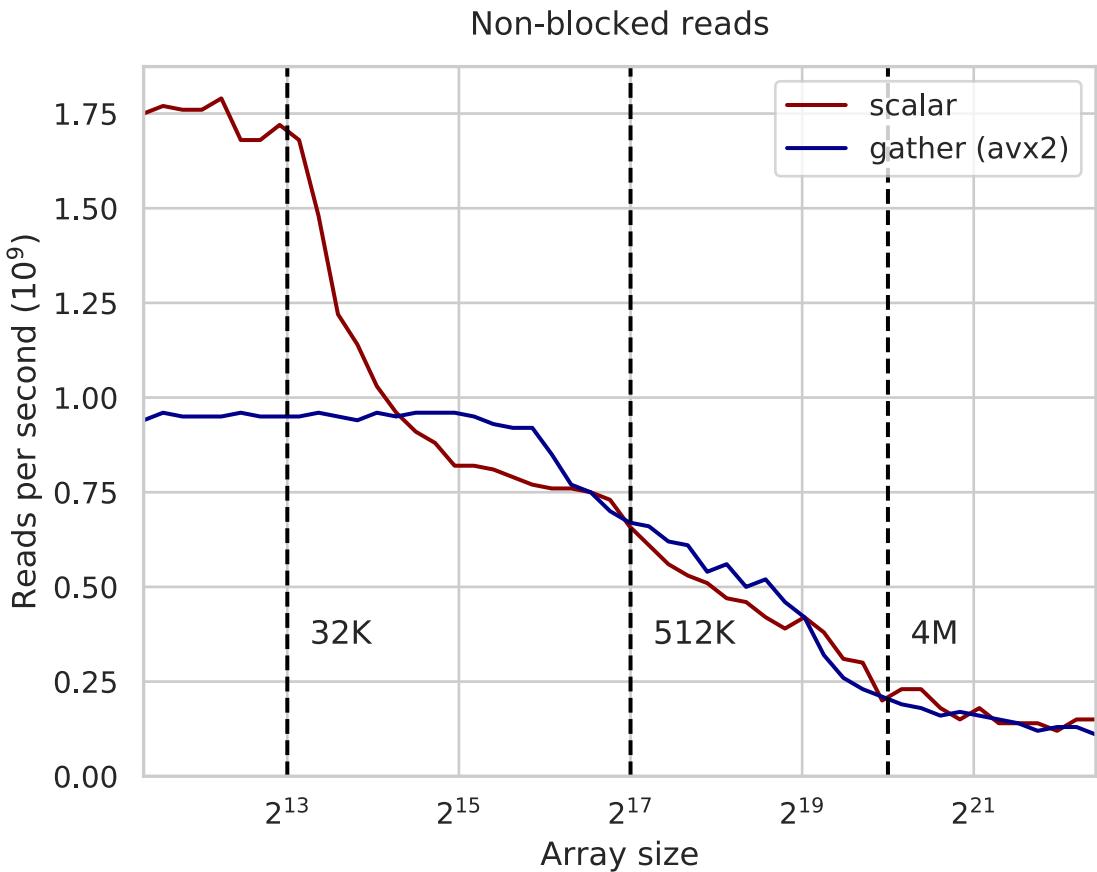
for (int i = 0; i < Q; i++)
    s += a[q[i]];
```

And in the SIMD code, we use the `gather` instruction to do that for 8 different indexes in parallel:

```
reg s = _mm256_setzero_si256();

for (int i = 0; i < Q; i += 8) {
    reg idx = _mm256_load_si256( (reg*) &q[i] );
    reg x = _mm256_i32gather_epi32(a, idx, 4);
    s = _mm256_add_epi32(s, x);
}
```

They perform roughly the same, except when the array fits into the L1 cache:



The purpose of `gather` and `scatter` is not to perform memory operations faster, but to get the data into registers to perform heavy computations on them. For anything costlier than just one addition, they are hugely favorable.

The lack of (fast) gather and scatter instructions makes SIMD programming on CPUs very different from proper parallel computing environments that support independent memory access. You have to always engineer around it and employ various ways of organizing your data sequentially so that it be loaded into registers.

Reductions

Reduction (also known as *folding* in functional programming) is the action of computing the value of some associative and commutative operation (i.e., $(a \circ b) \circ c = a \circ (b \circ c)$ and $a \circ b = b \circ a$) over a range of arbitrary elements.

The simplest example of reduction is calculating the sum an array:

```
int sum(int *a, int n) {
    int s = 0;
    for (int i = 0; i < n; i++)
        s += a[i];
    return s;
}
```

The naive approach is not so straightforward to vectorize, because the state of the loop (sum s on the current prefix) depends on the previous iteration. The way to overcome this is to split a single scalar accumulator s

into 8 separate ones, so that s_i would contain the sum of every 8th element of the original array, shifted by i :

$$s_i = \sum_{j=0}^{n/8} a_{8 \cdot j + i}$$

If we store these 8 accumulators in a single 256-bit vector, we can update them all at once by adding consecutive 8-element segments of the array. With vector extensions, this is straightforward:

```
int sum_simd(v8si *a, int n) {
    // ^ you can just cast a pointer normally, like with any other pointer type
    v8si s = {0};

    for (int i = 0; i < n / 8; i++)
        s += a[i];

    int res = 0;

    // sum 8 accumulators into one
    for (int i = 0; i < 8; i++)
        res += s[i];

    // add the remainder of a
    for (int i = n / 8 * 8; i < n; i++)
        res += a[i];

    return res;
}
```

You can use this approach for other reductions, such as for finding the minimum or the xor-sum of an array.

Instruction-Level Parallelism

Our implementation matches what the compiler produces automatically, but it is actually suboptimal: when we use just one accumulator, we have to wait one cycle between the loop iterations for a vector addition to complete, while the throughput of corresponding instruction is 2 on this microarchitecture.

If we again divide the array in $B \geq 2$ parts and use a *separate* accumulator for each, we can saturate the throughput of vector addition and increase the performance twofold:

```
const int B = 2; // how many vector accumulators to use

int sum_simd(v8si *a, int n) {
    v8si b[B] = {0};

    for (int i = 0; i + (B - 1) < n / 8; i += B)
        for (int j = 0; j < B; j++)
            b[j] += a[i + j];

    // sum all vector accumulators into one
    for (int i = 1; i < B; i++)
        b[0] += b[i];

    int s = 0;
```

```

// sum 8 scalar accumulators into one
for (int i = 0; i < 8; i++)
    s += b[0][i];

// add the remainder of a
for (int i = n / (8 * B) * (8 * B); i < n; i++)
    s += a[i];

return s;
}

```

If you have more than 2 relevant execution ports, you can increase the B constant accordingly, but the n -fold performance increase will only apply to arrays that fit into L1 cache — memory bandwidth will be the bottleneck for anything larger.

Horizontal Summation

The part where we sum up the 8 accumulators stored in a vector register into a single scalar to get the total sum is called “horizontal summation.”

Although extracting and adding every scalar one by one only takes a constant number of cycles, it can be computed slightly faster using a special instruction that adds together pairs of adjacent elements in a register.

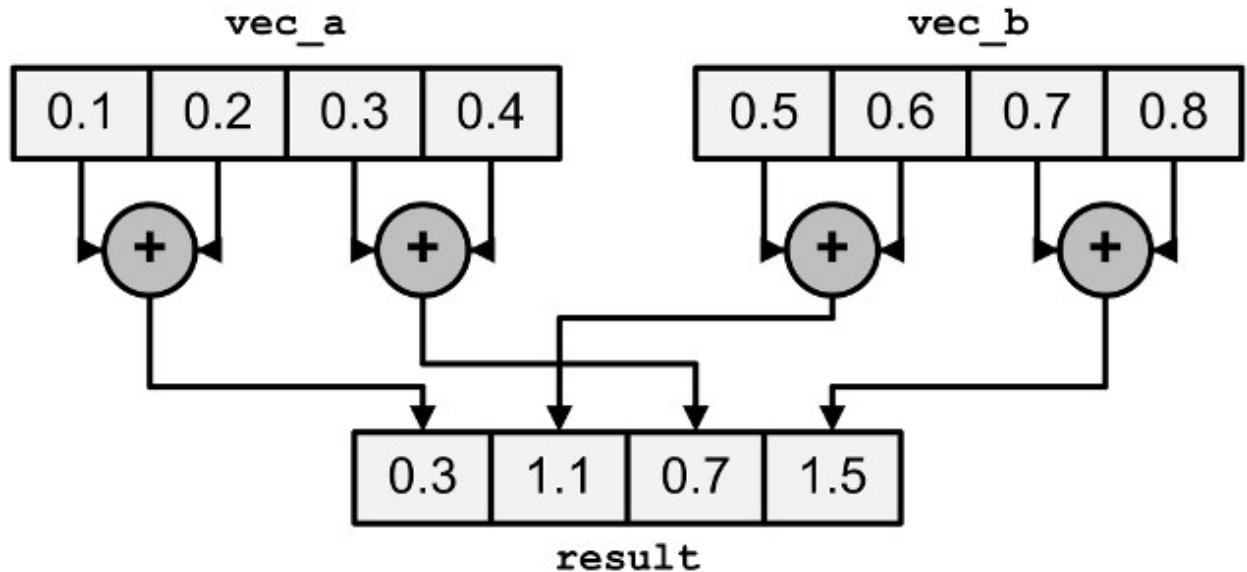


Figure 1: Horizontal summation in SSE/AVX. Note how the output is stored: the (a b a b) interleaving is common for reducing operations

Since it is a very specific operation, it can only be done with SIMD intrinsics — although the compiler probably emits roughly the same procedure for the scalar code anyway:

```

int hsum(__m256i x) {
    __m128i l = _mm256_extracti128_si256(x, 0);
    __m128i h = _mm256_extracti128_si256(x, 1);
    l = _mm_add_epi32(l, h);
    l = _mm_hadd_epi32(l, l);
}

```

```

    return _mm_extract_epi32(l, 0) + _mm_extract_epi32(l, 1);
}

```

There are other similar instructions, e.g., for integer multiplication or calculating absolute differences between adjacent elements (used in image processing).

There is also one specific instruction, `_mm_minpos_epu16`, that calculates the horizontal minimum and its index among eight 16-bit integers. This is the only horizontal reduction that works in one go: all others are computed in multiple steps.

Masking and Blending

One of the bigger challenges of SIMD programming is that its options for control flow are very limited — because the operations you apply to a vector are the same for all its elements.

This makes the problems that are usually trivially resolved with an `if` or any other type of branching much harder. With SIMD, they have to be dealt with by the means of various branchless programming techniques, which aren't always that straightforward to apply.

Masking

The main way to make a computation branchless is through *predication* — computing the results of both branches and then using either some arithmetic trick or a special “conditional move” instruction:

```

for (int i = 0; i < N; i++)
    a[i] = rand() % 100;

int s = 0;

// branch:
for (int i = 0; i < N; i++)
    if (a[i] < 50)
        s += a[i];

// no branch:
for (int i = 0; i < N; i++)
    s += (a[i] < 50) * a[i];

// also no branch:
for (int i = 0; i < N; i++)
    s += (a[i] < 50 ? a[i] : 0);

```

To vectorize this loop, we are going to need two new instructions:

- `_mm256_cmpgt_epi32`, which compares the integers in two vectors and produces a mask of all ones if the first element is more than the second and a mask of full zeros otherwise.
- `_mm256_blendv_epi8`, which blends (combines) the values of two vectors based on the provided mask.

By masking and blending the elements of a vector so that only the selected subset of them is affected by computation, we can perform predication in a manner similar to the conditional move:

```

const reg c = _mm256_set1_epi32(49);
const reg z = _mm256_setzero_si256();
reg s = _mm256_setzero_si256();

for (int i = 0; i < N; i += 8) {
    reg x = _mm256_load_si256( (reg*) &a[i] );
    reg mask = _mm256_cmpgt_epi32(x, c);

```

```

    x = _mm256_blendv_epi8(x, z, mask);
    s = _mm256_add_epi32(s, x);
}

```

(Minor details such as horizontal summation and accounting for the remainder of the array are omitted for brevity.)

This is how predication is usually done in SIMD, but it isn't always the most optimal approach. We can use the fact that one of the blended values is zero, and use bitwise `and` with the mask instead of blending:

```

const reg c = _mm256_set1_epi32(50);
reg s = _mm256_setzero_si256();

for (int i = 0; i < N; i += 8) {
    reg x = _mm256_load_si256( (reg*) &a[i] );
    reg mask = _mm256_cmpgt_epi32(c, x);
    x = _mm256_and_si256(x, mask);
    s = _mm256_add_epi32(s, x);
}

```

This loop performs slightly faster because on this particular CPU, the vector `and` takes one cycle less than `blend`.

Several other instructions support masks as inputs, most notably:

- The `_mm256_blend_epi32` intrinsic is a `blend` that takes an 8-bit integer mask instead of a vector (which is why it doesn't have `v` at the end).
- The `_mm256_maskload_epi32` and `_mm256_maskstore_epi32` intrinsics that load/store a SIMD block from memory and `and` it with a mask in one go.

We can also use predication with built-in vector types:

```

vec *v = (vec*) a;
vec s = {};

for (int i = 0; i < N / 8; i++)
    s += (v[i] < 50 ? v[i] : 0);

```

All these versions work at around 13 GFLOPS as this example is so simple that the compiler can vectorize the loop all by itself. Let's move on to more complex examples that can't be auto-vectorized.

Searching

In the next example, we need to find a specific value in an array and return its position (aka `std::find`):

```

const int N = (1<<12);
int a[N];

int find(int x) {
    for (int i = 0; i < N; i++)
        if (a[i] == x)
            return i;
    return -1;
}

```

To benchmark the `find` function, we fill the array with numbers from 0 to $(N - 1)$ and then repeatedly search for a random element:

```

for (int i = 0; i < N; i++)
    a[i] = i;

```

```

for (int t = 0; t < K; t++)
    checksum ^= find(rand() % N);

```

The scalar version gives ~4 GFLOPS of performance. This number includes the elements we haven't had to process, so divide this number by two in your head (the expected fraction of the elements we have to check).

To vectorize it, we need to compare a vector of its elements with the searched value for equality, producing a mask, and then somehow check if this mask is zero. If it isn't, the needed element is somewhere within this block of 8.

To check if the mask is zero, we can use the `_mm256_movemask_ps` intrinsic, which takes the first bit of each 32-bit element in a vector and produces an 8-bit integer mask out of them. We can then check if this mask is non-zero — and if it is, also immediately get the index with the `ctz` instruction:

```

int find(int needle) {
    reg x = _mm256_set1_epi32(needle);

    for (int i = 0; i < N; i += 8) {
        reg y = _mm256_load_si256((reg*) &a[i]);
        reg m = _mm256_cmpeq_epi32(x, y);
        int mask = _mm256_movemask_ps((__m256) m);
        if (mask != 0)
            return i + __builtin_ctz(mask);
    }

    return -1;
}

```

This version gives ~20 GFLOPS or about 5 times faster than the scalar one. It only uses 3 instructions in the hot loop:

```

vpcmpeqd ymm0, ymm1, YMMWORD PTR a[0+rdx*4]
vmovmskps eax, ymm0
test     eax, eax
je      loop

```

Checking if a vector is zero is a common operation, and there is an operation similar to `test` in SIMD that we can use:

```

int find(int needle) {
    reg x = _mm256_set1_epi32(needle);

    for (int i = 0; i < N; i += 8) {
        reg y = _mm256_load_si256((reg*) &a[i]);
        reg m = _mm256_cmpeq_epi32(x, y);
        if (!_mm256_testz_si256(m, m)) {
            int mask = _mm256_movemask_ps((__m256) m);
            return i + __builtin_ctz(mask);
        }
    }

    return -1;
}

```

We are still using `movemask` to do `ctz` later, but the hot loop is now one instruction shorter:

```

vpcmpeqd ymm0, ymm1, YMMWORD PTR a[0+rdx*4]
vptest   ymm0, ymm0

```

```
je      loop
```

This doesn't improve performance much because both both `vptest` and `vmovmskps` have a throughput of one and will bottleneck the computation regardless of anything else we do in the loop.

To work around this limitation, we can iterate in blocks of 16 elements and combine the results of independent comparisons of two 256-bit AVX2 registers using a bitwise `or`:

```
int find(int needle) {
    reg x = _mm256_set1_epi32(needle);

    for (int i = 0; i < N; i += 16) {
        reg y1 = _mm256_load_si256( (reg*) &a[i] );
        reg y2 = _mm256_load_si256( (reg*) &a[i + 8] );
        reg m1 = _mm256_cmpeq_epi32(x, y1);
        reg m2 = _mm256_cmpeq_epi32(x, y2);
        reg m = _mm256_or_si256(m1, m2);
        if (!_mm256_testz_si256(m, m)) {
            int mask = (_mm256_movemask_ps((__m256) m2) << 8)
                       + _mm256_movemask_ps((__m256) m1);
            return i + __builtin_ctz(mask);
        }
    }

    return -1;
}
```

With this obstacle removed, the performance now peaks at ~34 GFLOPS. But why not 40? Shouldn't it be twice as fast?

Here is how one iteration of the loop looks in assembly:

```
vpcmpeqd ymm2, ymm1, YMMWORD PTR a[0+rdx*4]
vpcmpeqd ymm3, ymm1, YMMWORD PTR a[32+rdx*4]
vpovr ymm0, ymm3, ymm2
vptest ymm0, ymm0
je      loop
```

Every iteration, we need to execute 5 instructions. While the throughputs of all relevant execution ports allow to do that in one cycle on average, we can't do that because the decode width of this particular CPU (Zen 2) is 4. Therefore, the performance is limited by 4/5 of what it could have been.

To mitigate this, we can once again double the number of SIMD blocks we process on each iteration:

```
unsigned get_mask(reg m) {
    return _mm256_movemask_ps((__m256) m);
}

reg cmp(reg x, int *p) {
    reg y = _mm256_load_si256( (reg*) p );
    return _mm256_cmpeq_epi32(x, y);
}

int find(int needle) {
    reg x = _mm256_set1_epi32(needle);

    for (int i = 0; i < N; i += 32) {
        reg m1 = cmp(x, &a[i]);
    }
}
```

```

    reg m2 = cmp(x, &a[i + 8]);
    reg m3 = cmp(x, &a[i + 16]);
    reg m4 = cmp(x, &a[i + 24]);
    reg m12 = _mm256_or_si256(m1, m2);
    reg m34 = _mm256_or_si256(m3, m4);
    reg m = _mm256_or_si256(m12, m34);
    if (!_mm256_testz_si256(m, m)) {
        unsigned mask = (get_mask(m4) << 24)
                        + (get_mask(m3) << 16)
                        + (get_mask(m2) << 8)
                        + get_mask(m1);
        return i + __builtin_ctz(mask);
    }
}

return -1;
}

```

It now shows the throughput of 43 GFLOPS — or about 10x faster than the original scalar implementation.

Extending it to 64 values per cycle doesn't help: small arrays suffer from the overhead of all these additional `movmask`s when we hit the condition, and larger arrays are bottlenecked by memory bandwidth anyway.

Counting Values

As the final exercise, let's find the count of a value in an array instead of just its first occurrence:

```

int count(int x) {
    int cnt = 0;
    for (int i = 0; i < N; i++)
        cnt += (a[i] == x);
    return cnt;
}

```

To vectorize it, we just need to convert the comparison mask to either one or zero per element and calculate the sum:

```

const reg ones = _mm256_set1_epi32(1);

int count(int needle) {
    reg x = _mm256_set1_epi32(needle);
    reg s = _mm256_setzero_si256();

    for (int i = 0; i < N; i += 8) {
        reg y = _mm256_load_si256((reg*) &a[i] );
        reg m = _mm256_cmpeq_epi32(x, y);
        m = _mm256_and_si256(m, ones);
        s = _mm256_add_epi32(s, m);
    }

    return hsum(s);
}

```

Both implementations yield ~15 GFLOPS: the compiler can vectorize the first one all by itself.

But a trick that the compiler can't find is to notice that the mask of all ones is minus one when reinterpreted as an integer. So we can skip the and-the-lowest-bit part and use the mask itself, and then just negate the final result:

```

int count(int needle) {
    reg x = _mm256_set1_epi32(needle);
    reg s = _mm256_setzero_si256();

    for (int i = 0; i < N; i += 8) {
        reg y = _mm256_load_si256( (reg*) &a[i] );
        reg m = _mm256_cmpeq_epi32(x, y);
        s = _mm256_add_epi32(s, m);
    }

    return -hsum(s);
}

```

This doesn't improve the performance in this particular architecture because the throughput is actually bottlenecked by updating `s`: there is a dependency on the previous iteration, so the loop can't proceed faster than one iteration per CPU cycle. We can make use of instruction-level parallelism if we split the accumulator in two:

```

int count(int needle) {
    reg x = _mm256_set1_epi32(needle);
    reg s1 = _mm256_setzero_si256();
    reg s2 = _mm256_setzero_si256();

    for (int i = 0; i < N; i += 16) {
        reg y1 = _mm256_load_si256( (reg*) &a[i] );
        reg y2 = _mm256_load_si256( (reg*) &a[i + 8] );
        reg m1 = _mm256_cmpeq_epi32(x, y1);
        reg m2 = _mm256_cmpeq_epi32(x, y2);
        s1 = _mm256_add_epi32(s1, m1);
        s2 = _mm256_add_epi32(s2, m2);
    }

    s1 = _mm256_add_epi32(s1, s2);

    return -hsum(s1);
}

```

It now gives ~22 GFLOPS of performance, which is as high as it can get.

When adapting this code for shorter data types, keep in mind that the accumulator may overflow. To work around this, add another accumulator of larger size and regularly stop the loop to add the values in the local accumulator to it and then reset the local accumulator. For example, for 8-bit integers, this means creating another inner loop that does $\lfloor \frac{256-1}{8} \rfloor = 15$ iterations.

In-Register Shuffles

Masking lets you apply operations to only a subset of vector elements. It is a very effective and frequently used data manipulation technique, but in many cases, you need to perform more advanced operations that involve permuting values inside a vector register instead of just blending them with other vectors.

The problem is that adding a separate element-shuffling instruction for each possible use case in hardware is unfeasible. What we can do though is to add just one general permutation instruction that takes the indices of a permutation and produces these indices using precomputed lookup tables.

This general idea is perhaps too abstract, so let's jump straight to the examples.

Shuffles and Popcount

Population count, also known as the *Hamming weight*, is the count of 1 bits in a binary string.

It is a frequently used operation, so there is a separate instruction on x86 that computes the population count of a word:

```
const int N = (1<<12);
int a[N];

int popcnt() {
    int res = 0;
    for (int i = 0; i < N; i++)
        res += __builtin_popcount(a[i]);
    return res;
}
```

It also supports 64-bit integers, improving the total throughput twofold:

```
int popcnt_ll() {
    long long *b = (long long*) a;
    int res = 0;
    for (int i = 0; i < N / 2; i++)
        res += __builtin_popcountl(b[i]);
    return res;
}
```

The only two instructions required are load-fused population count and addition. They both have a high throughput, so the code processes about $8 + 8 = 16$ bytes per cycle as it is limited by the decode width of 4 on this CPU.

These instructions were added to x86 CPUs around 2008 with SSE4. Let's temporarily go back in time before vectorization even became a thing and try to implement popcount by other means.

The naive way is to go through the binary string bit by bit:

```
__attribute__(( optimize("no-tree-vectorize") ))
int popcnt() {
    int res = 0;
    for (int i = 0; i < N; i++)
        for (int l = 0; l < 32; l++)
            res += (a[i] >> l & 1);
    return res;
}
```

As anticipated, it works just slightly faster than 1/8-th of a byte per cycle — at around 0.2.

We can try to process in bytes instead of individual bits by precomputing a small 256-element *lookup table* that contains the population counts of individual bytes and then query it while iterating over raw bytes of the array:

```
struct Precalc {
    alignas(64) char counts[256];

    constexpr Precalc() : counts{} {
        for (int m = 0; m < 256; m++)
            for (int i = 0; i < 8; i++)
                counts[m] += (m >> i & 1);
    }
};
```

```

constexpr Precalc P;

int popcnt() {
    auto b = (unsigned char*) a; // careful: plain "char" is signed
    int res = 0;
    for (int i = 0; i < 4 * N; i++)
        res += P.counts[b[i]];
    return res;
}

```

It now processes around 2 bytes per cycle, rising to ~ 2.7 if we switch to 16-bit words (`unsigned short`).

This solution is still very slow compared to the `popcnt` instruction, but now it can be vectorized. Instead of trying to speed it up through gather instructions, we will go for another approach: make the lookup table small enough to fit inside a register and then use a special `pshufb` instruction to look up its values in parallel.

The original `pshufb` introduced in 128-bit SSE3 takes two registers: the lookup table containing 16 byte values and a vector of 16 4-bit indices (0 to 15), specifying which bytes to pick for each position. In 256-bit AVX2, instead of a 32-byte lookup table with awkward 5-bit indices, we have an instruction that independently the same shuffling operation over two 128-bit lanes.

So, for our use case, we create a 16-byte lookup table with population counts for each nibble (half-byte), repeated twice:

```

const reg lookup = _mm256_setr_epi8(
    /* 0 */ 0, /* 1 */ 1, /* 2 */ 1, /* 3 */ 2,
    /* 4 */ 1, /* 5 */ 2, /* 6 */ 2, /* 7 */ 3,
    /* 8 */ 1, /* 9 */ 2, /* a */ 2, /* b */ 3,
    /* c */ 2, /* d */ 3, /* e */ 3, /* f */ 4,
    /* 0 */ 0, /* 1 */ 1, /* 2 */ 1, /* 3 */ 2,
    /* 4 */ 1, /* 5 */ 2, /* 6 */ 2, /* 7 */ 3,
    /* 8 */ 1, /* 9 */ 2, /* a */ 2, /* b */ 3,
    /* c */ 2, /* d */ 3, /* e */ 3, /* f */ 4
);

```

Now, to compute the population count of a vector, we split each of its bytes into the lower and higher nibbles and then use this lookup table to retrieve their counts. The only thing left is to carefully sum them up:

```

const reg low_mask = _mm256_set1_epi8(0x0f);

int popcnt() {
    int k = 0;

    reg t = _mm256_setzero_si256();

    for (; k + 15 < N; k += 15) {
        reg s = _mm256_setzero_si256();

        for (int i = 0; i < 15; i += 8) {
            reg x = _mm256_load_si256((reg*) &a[k + i]);

            reg l = _mm256_and_si256(x, low_mask);
            reg h = _mm256_and_si256(_mm256_srli_epi16(x, 4), low_mask);

            reg pl = _mm256_shuffle_epi8(lookup, l);
            reg ph = _mm256_shuffle_epi8(lookup, h);

```

```

        s = _mm256_add_epi8(s, pl);
        s = _mm256_add_epi8(s, ph);
    }

    t = _mm256_add_epi64(t, _mm256_sad_epu8(s, _mm256_setzero_si256()));
}

int res = hsum(t);

while (k < N)
    res += __builtin_popcount(a[k++]);

return res;
}

```

This code processes around 30 bytes per cycle. Theoretically, the inner loop could do 32, but we have to stop it every 15 iterations because the 8-bit counters can overflow.

The `pshufb` instruction is so instrumental in some SIMD algorithms that Wojciech Mula — the guy who came up with this algorithm — took it as his Twitter handle. You can calculate population counts even faster: check out his GitHub repository with different vectorized `popcount` implementations and his recent paper for a detailed explanation of the state-of-the-art.

Permutations and Lookup Tables

Our last major example in this chapter is the `filter`. It is a very important data processing primitive that takes an array as input and writes out only the elements that satisfy a given predicate (in their original order).

In a single-threaded scalar case, it is trivially implemented by maintaining a counter that is incremented on each write:

```

int a[N], b[N];

int filter() {
    int k = 0;

    for (int i = 0; i < N; i++)
        if (a[i] < P)
            b[k++] = a[i];

    return k;
}

```

To vectorize it, we will use the `_mm256_permutevar8x32_epi32` intrinsic. It takes a vector of values and individually selects them with a vector of indices. Despite the name, it doesn't *permute* values but just *copies* them to form a new vector: duplicates in the result are allowed.

The general idea of our algorithm is as follows:

- calculate the predicate on a vector of data — in this case, this means performing the comparisons to get the mask;
- use the `movemask` instruction to get a scalar 8-bit mask;
- use this mask to index a lookup table that returns a permutation moving the elements that satisfy the predicate to the beginning of the vector (in their original order);
- use the `_mm256_permutevar8x32_epi32` intrinsic to permute the values;

- write the whole permuted vector to the buffer — it may have some trailing garbage, but its prefix is correct;
- calculate the population count of the scalar mask and move the buffer pointer by that number.

First, we need to precompute the permutations:

```
struct Precalc {
    alignas(64) int permutation[256][8];

    constexpr Precalc() : permutation{} {
        for (int m = 0; m < 256; m++) {
            int k = 0;
            for (int i = 0; i < 8; i++)
                if ((m >> i) & 1)
                    permutation[m][k++] = i;
        }
    }
};
```

```
constexpr Precalc T;
```

Then we can implement the algorithm itself:

```
const reg p = _mm256_set1_epi32(P);

int filter() {
    int k = 0;

    for (int i = 0; i < N; i += 8) {
        reg x = _mm256_load_si256((reg*) &a[i] );

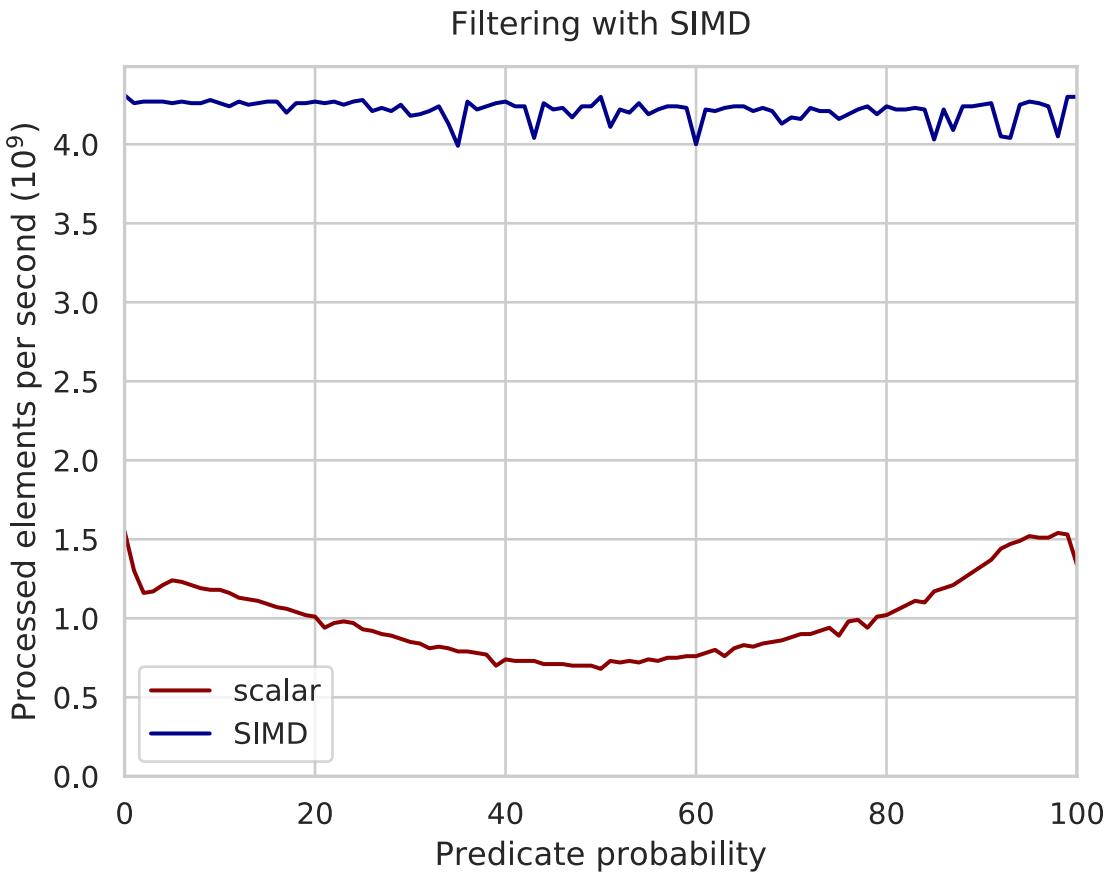
        reg m = _mm256_cmpgt_epi32(p, x);
        int mask = _mm256_movemask_ps((__m256) m);
        reg permutation = _mm256_load_si256((reg*) &T.permutation[mask] );

        x = _mm256_permutevar8x32_epi32(x, permutation);
        _mm256_storeu_si256((reg*) &b[k], x);

        k += __builtin_popcount(mask);
    }

    return k;
}
```

The vectorized version takes some work to implement, but it is 6-7x faster than the scalar one (the speedup is slightly less for either low or high values of P as the branch becomes predictable).



The loop performance is still relatively low — taking 4 CPU cycles per iteration — because, on this particular CPU (Zen 2), `movmask`, `permute`, and `store` have low throughput and all have to go through the same execution port (P2). On most other x86 CPUs, you can expect it to be $\sim 2x$ faster.

Filtering can also be implemented considerably faster on AVX-512: it has a special “compress” instruction that takes a vector of data and a mask and writes its unmasked elements contiguously. It makes a huge difference in algorithms that rely on various filtering subroutines, such as quicksort.

Auto-Vectorization and SPMD

SIMD parallelism is most often used for *embarrassingly parallel* computations: the kinds where all you do is apply some elementwise function to all elements of an array and write it back somewhere else. In this setting, you don’t even need to know how SIMD works: the compiler is perfectly capable of optimizing such loops by itself — you just need to be aware that such optimization exists and that it usually yields a 5-10x speedup.

Doing nothing and relying on auto-vectorization is actually the most popular way of using SIMD. In fact, in many cases, it even advised to stick with the plain scalar code for its simplicity and maintainability.

But often even the loops that seem straightforward to vectorize are not optimized because of some technical nuances. As in many other cases, the compiler may need some additional input from the programmer as he may know a bit more about the problem than what can be inferred from static analysis.

Potential Problems

Consider the “ $a + b$ ” example we started with:

```

void sum(int *a, int *b, int *c, int n) {
    for (int i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}

```

Let's step into a compiler's shoes and think about what can go wrong when this loop is vectorized.

Array size. If the array size is unknown beforehand, it may be that it is too small for vectorization to be beneficial in the first place. Even if it is sufficiently large, we need to insert an additional check for the remainder of the loop to process it scalar, which would cost us a branch.

To eliminate these runtime checks, use array sizes that are compile-time constants, and preferably pad arrays to the nearest multiple of the SIMD block size.

Memory aliasing. Even when array size issues are out of the question, vectorizing this loop is not always technically correct. For example, the arrays `a` and `c` can intersect in a way that their beginnings differ by a single position — because who knows, maybe the programmer wanted to calculate the Fibonacci sequence through a convolution this way. In this case, the data in the SIMD blocks will intersect and the observed behavior will differ from the one in the scalar case.

When the compiler can't prove that the function may be used for intersecting arrays, it has to generate two implementation variants — a vectorized and a “safe” one — and insert runtime checks to choose between the two. To avoid them, we can tell the compiler that we are that no memory is aliased by adding the `__restrict__` keyword:

```

void add(int * __restrict__ a, const int * __restrict__ b, int n) {
    for (int i = 0; i < n; i++)
        a[i] += b[i];
}

```

The other way, specific to SIMD, is the “ignore vector dependencies” pragma. It is a general way to inform the compiler that there are no dependencies between the loop iterations:

```

#pragma GCC ivdep
for (int i = 0; i < n; i++)
    // ...

```

Alignment. The compiler also doesn't know anything about the alignment of these arrays and has to either process some elements at the beginning of these arrays before starting the vectorized section or potentially lose some performance by using unaligned memory accesses.

To help the compiler eliminate this corner case, we can use the `alignas` specifier on static arrays and the `std::assume_aligned` function to mark pointers aligned.

Checking if vectorization happened. In either case, it is useful to check if the compiler vectorized the loop the way you intended. You can either compiling it to assembly and look for blocks for instructions that start with a “v” or add the `-fopt-info-vec-optimized` compiler flag so that the compiler indicates where auto-vectorization is happening and what SIMD width is being used. If you swap `optimized` for `missed` or `all`, you may also get some reasoning behind why it is not happening in other places.

There are many other ways of telling the compiler exactly what we mean, but in especially complex cases — e.g., when there are a lot of branches or function calls inside the loop — it is easier to go one level of abstraction down and vectorize manually.

SPMD

There is a neat compromise between auto-vectorization and the manual use of SIMD intrinsics: “single program, multiple data” (SPMD). This is a model of computation in which the programmer writes what appears to be a regular serial program, but that is actually executed in parallel on the hardware.

The programming experience is largely the same, and there is still the fundamental limitation in that the computation must be data-parallel, but SPMD ensures that the vectorization will happen regardless of the compiler and the target CPU architecture. It also allows for the computation to be automatically parallelized across multiple cores and, in some cases, even offloaded to other types of parallel hardware.

There is support for SPMD in some modern languages (Julia), multiprocessing APIs (OpenMP), and specialized compilers (Intel ISPC), but it has seen the most success in the context of GPU programming where both problems and hardware are massively parallel.

We will cover this model of computation in much more depth in Part 2

Chapter 11: Algorithms Case Studies

Algorithms for Modern Hardware

Contents

Binary GCD	2
Euclid's Algorithm	2
Binary GCD	4
Implementation	4
Acknowledgements	7
Integer Factorization	7
Benchmark	7
Trial division	8
Lookup Table	8
Wheel factorization	9
Precomputed Primes	10
Pollard's Rho Algorithm	11
Pollard-Brent Algorithm	13
Optimizing the Modulo	14
Further Improvements	15
Optimizing Logistic Regression	16
Quantization	18
Sorting	19
Argmin with SIMD	19
Scalar Baseline	19
Vector of Indices	20
Branches Aren't Scary	21
Find the Minimum, Then Find the Index	24
Summary	25
Acknowledgements	25
Prefix Sum with SIMD	25
Baseline	26
Vectorization	26
Blocking	28
Continuous Loads	29
Other Relevant Work	30
Reading Decimal Integers	30
Iostream	31
Scanf	31
Synchronization	31
Getchar	31

Buffering	31
SIMD	31
Serial	31
Transpose-based approach	31
Instruction-level parallelism	31
Modifications	31
Future work	31
Acknowledgements	31
Algorithms Case Studies	31
Matrix Multiplication	32
Baseline	32
Transposition	32
Vectorization	33
Memory efficiency	34
Register reuse	35
Designing the kernel	36
Blocking	38
Optimization	39
Generalizations	40
Acknowledgements	41

Binary GCD

In this section, we will derive a variant of `gcd` that is $\sim 2x$ faster than the one in the C++ standard library.

Euclid's Algorithm

Euclid's algorithm solves the problem of finding the *greatest common divisor* (GCD) of two integer numbers a and b , which is defined as the largest such number g that divides both a and b :

$$\text{gcd}(a, b) = \max_{g: g|a \wedge g|b} g$$

You probably already know this algorithm from a CS textbook, but I will summarize it here. It is based on the following formula, assuming that $a > b$:

$$\text{gcd}(a, b) = \begin{cases} a, & b = 0 \\ \text{gcd}(b, a \bmod b), & b > 0 \end{cases}$$

This is true, because if $g = \text{gcd}(a, b)$ divides both a and b , it should also divide $(a \bmod b = a - k \cdot b)$, but any larger divisor d of b will not: $d > g$ implies that d couldn't divide a and thus won't divide $(a - k \cdot b)$.

The formula above is essentially the algorithm itself: you can simply apply it recursively, and since each time one of the arguments strictly decreases, it will eventually converge to the $b = 0$ case.

You can see bright blue lines at the proportions of the golden ratio

Figure 1: You can see bright blue lines at the proportions of the golden ratio

The textbook also probably mentioned that the worst possible input to Euclid's algorithm — the one that maximizes the total number of steps — are consecutive Fibonacci numbers, and since they grow exponentially, the running time of the algorithm is logarithmic in the worst case. This is also true for its *average* running time if we define it as the expected number of steps for pairs of uniformly distributed integers. [The Wikipedia article](#) also has a cryptic derivation of a more precise $0.84 \cdot \ln n$ asymptotic estimate.

There are many ways you can implement Euclid's algorithm. The simplest would be just to convert the definition into code:

```
int gcd(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}
```

You can rewrite it more compactly like this:

```
int gcd(int a, int b) {
    return (b ? gcd(b, a % b) : a);
}
```

You can rewrite it as a loop, which will be closer to how it is actually executed by the hardware. It won't be faster though, because compilers can easily optimize tail recursion.

```
int gcd(int a, int b) {
    while (b > 0) {
        a %= b;
        std::swap(a, b);
    }
    return a;
}
```

You can even write the body of the loop as this confusing one-liner — and it will even compile without causing undefined behavior warnings since C++17:

```
int gcd(int a, int b) {
    while (b) b ^= a ^= b ^= a %= b;
    return a;
}
```

All of these, as well as `std::gcd` which was introduced in C++17, are almost equivalent and get compiled into functionally the following assembly loop:

```
; a = eax, b = edx
loop:
    ; modulo in assembly:
    mov r8d, edx
```

```

cdq
idiv r8d
mov eax, r8d
; (a and b are already swapped now)
; continue until b is zero:
test edx, edx
jne loop

```

If you run `perf` on it, you will see that it spends ~90% of the time on the `idiv` line. This isn't surprising: general [integer division](#) works notoriously slow on all computers, including x86.

But there is one kind of division that works well in hardware: division by a power of 2.

Binary GCD

The *binary GCD algorithm* was discovered around the same time as Euclid's, but on the other end of the civilized world, in ancient China. In 1967, it was rediscovered by Josef Stein for use in computers that either don't have division instruction or have a very slow one — it wasn't uncommon for CPUs of that era to use hundreds or thousands of cycles for rare or complex operations.

Analogous to the Euclidean algorithm, it is based on a few similar observations:

1. $\gcd(0, b) = b$ and symmetrically $\gcd(a, 0) = a$;
2. $\gcd(2a, 2b) = 2 \cdot \gcd(a, b)$;
3. $\gcd(2a, b) = \gcd(a, b)$ if b is odd and symmetrically $\gcd(a, b) = \gcd(a, 2b)$ if a is odd;
4. $\gcd(a, b) = \gcd(|a - b|, \min(a, b))$, if a and b are both odd.

Likewise, the algorithm itself is just a repeated application of these identities.

Its running time is still logarithmic, which is even easier to show because in each of these identities one of the arguments is divided by 2 — except for the last case, in which the new first argument, the absolute difference of two odd numbers, is guaranteed to be even and thus will be divided by 2 on the next iteration.

What makes this algorithm especially interesting to us is that the only arithmetic operations it uses are binary shifts, comparisons, and subtractions, all of which typically take just one cycle.

Implementation

The reason this algorithm is not in the textbooks is because it can't be implemented as a simple one-liner anymore:

```

int gcd(int a, int b) {
    // base cases (1)
    if (a == 0) return b;
    if (b == 0) return a;
    if (a == b) return a;

    if (a % 2 == 0) {
        if (b % 2 == 0) // a is even, b is even (2)
            return 2 * gcd(a / 2, b / 2);
        else           // a is even, b is odd (3)

```

```

        return gcd(a / 2, b);
    } else {
        if (b % 2 == 0) // a is odd, b is even (3)
            return gcd(a, b / 2);
        else           // a is odd, b is odd (4)
            return gcd(std::abs(a - b), std::min(a, b));
    }
}

```

Let's run it, and... it sucks. The difference in speed compared to `std::gcd` is indeed 2x, but on the other side of the equation. This is mainly because of all the branching needed to differentiate between the cases. Let's start optimizing.

First, let's replace all divisions by 2 with divisions by whichever highest power of 2 we can. We can do it efficiently with `__builtin_ctz`, the “count trailing zeros” instruction available on modern CPUs. Whenever we are supposed to divide by 2 in the original algorithm, we will call this function instead, which will give us the exact number of bits to right-shift the number by. Assuming that we are dealing with large random numbers, this is expected to decrease the number of iterations by almost a factor 2, because $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots \rightarrow 2$.

Second, we can notice that condition 2 can now only be true once — in the very beginning — because every other identity leaves at least one of the numbers odd. Therefore we can handle this case just once in the beginning and not consider it in the main loop.

Third, we can notice that after we've entered condition 4 and applied its identity, a will always be even and b will always be odd, so we already know that on the next iteration we are going to be in condition 3. This means that we can actually “de-evenize” a right away, and if we do so we will again hit condition 4 on the next iteration. This means that we can only ever be either in condition 4 or terminating by condition 1, which removes the need to branch.

Combining these ideas, we get the following implementation:

```

int gcd(int a, int b) {
    if (a == 0) return b;
    if (b == 0) return a;

    int az = __builtin_ctz(a);
    int bz = __builtin_ctz(b);
    int shift = std::min(az, bz);
    a >>= az, b >>= bz;

    while (a != 0) {
        int diff = a - b;
        b = std::min(a, b);
        a = std::abs(diff);
        a >>= __builtin_ctz(a);
    }

    return b << shift;
}

```

It runs in 116ns, while `std::gcd` takes 198ns. Almost twice as fast — maybe we can even optimize it below 100ns?

For that we need to stare at [its assembly](#) again, in particular at this block:

```
; a = edx, b = eax
loop:
    mov    ecx, edx
    sub    ecx, eax        ; diff = a - b
    cmp    eax, edx
    cmovg eax, edx        ; b = min(a, b)
    mov    edx, ecx
    neg    edx
    cmovs edx, ecx        ; a = max(diff, -diff) = abs(diff)
    tzcnt ecx, edx        ; az = __builtin_ctz(a)
    sarx  edx, edx, ecx   ; a >= az
    test   edx, edx        ; a != 0?
    jne   loop
```

Let's draw the dependency graph of this loop:

Modern processors can execute many instructions in parallel, essentially meaning that the true “cost” of this computation is roughly the sum of latencies on its critical path. In this case, it is the total latency of `diff`, `abs`, `ctz`, and `shift`.

We can decrease this latency using the fact that we can actually calculate `ctz` using just `diff = a - b`, because a [negative number](#) divisible by 2^k still has k zeros at the end of its binary representation. This lets us not wait for `max(diff, -diff)` to be computed first, resulting in a shorter graph like this:

Hopefully you will be less confused when you think about how the final code will be executed:

```
int gcd(int a, int b) {
    if (a == 0) return b;
    if (b == 0) return a;

    int az = __builtin_ctz(a);
    int bz = __builtin_ctz(b);
    int shift = std::min(az, bz);
    b >>= bz;

    while (a != 0) {
        a >>= az;
        int diff = b - a;
        az = __builtin_ctz(diff);
        b = std::min(a, b);
        a = std::abs(diff);
    }

    return b << shift;
}
```

It runs in 91ns, which is good enough to leave it there.

If somebody wants to try to shave off a few more nanoseconds by rewriting the assembly by hand or trying a lookup table to save a few last iterations, please [let me know](#).

Acknowledgements

The main optimization ideas belong to Daniel Lemire and Ralph Corderoy, who [had nothing better to do](#) on the Christmas holidays of 2013.

Integer Factorization

The problem of factoring integers into primes is central to computational [number theory](#). It has been [studied](#) since at least the 3rd century BC, and [many methods](#) have been developed that are efficient for different inputs.

In this case study, we specifically consider the factorization of *word-sized* integers: those on the order of 10^9 and 10^{18} . Unusual for this book, in this one, you may actually learn an asymptotically better algorithm: we start with a few basic approaches and gradually build up to the $O(\sqrt[4]{n})$ -time *Pollard's rho algorithm* and optimize it to the point where it can factorize 60-bit semiprimes in 0.3-0.4ms and ~3 times faster than the previous state-of-the-art.

Benchmark

For all methods, we will implement `find_factor` function that takes a positive integer n and returns any of its non-trivial divisors (or 1 if the number is prime):

```
// I don't feel like typing "unsigned long long" each time
typedef __uint16_t u16;
typedef __uint32_t u32;
typedef __uint64_t u64;
typedef __uint128_t u128;

u64 find_factor(u64 n);
```

To find the full factorization, you can apply it to n , reduce it, and continue until a new factor can no longer be found:

```
vector<u64> factorize(u64 n) {
    vector<u64> factorization;
    do {
        u64 d = find_factor(n);
        factorization.push_back(d);
        n /= d;
    } while (d != 1);
    return factorization;
}
```

After each removed factor, the problem becomes considerably smaller, so the worst-case running time of full factorization is equal to the worst-case running time of a `find_factor` call.

For many factorization algorithms, including those presented in this section, the running time scales with the smaller prime factor. Therefore, to provide worst-case input, we use *semiprimes*: products of two prime numbers $p \leq q$ that are on the same order of magnitude. We generate a k -bit semiprime as the product of two random $\lfloor k/2 \rfloor$ -bit primes.

Since some of the algorithms are inherently randomized, we also tolerate a small ($< 1\%$) percentage of false-negative errors (when `find_factor` returns 1 despite number n being composite), although this rate can be reduced to almost zero without significant performance penalties.

Trial division

The most basic approach is to try every integer smaller than n as a divisor:

```
u64 find_factor(u64 n) {
    for (u64 d = 2; d < n; d++)
        if (n % d == 0)
            return d;
    return 1;
}
```

We can notice that if n is divided by $d < \sqrt{n}$, then it is also divided by $\frac{n}{d} > \sqrt{n}$, and there is no need to check for it separately. This lets us stop trial division early and only check for potential divisors that do not exceed \sqrt{n} :

```
u64 find_factor(u64 n) {
    for (u64 d = 2; d * d <= n; d++)
        if (n % d == 0)
            return d;
    return 1;
}
```

In our benchmark, n is a semiprime, and we always find the lesser divisor, so both $O(n)$ and $O(\sqrt{n})$ implementations perform the same and are able to factorize $\sim 2k$ 30-bit numbers per second — while taking whole 20 seconds to factorize a single 60-bit number.

Lookup Table

Nowadays, you can type `factor 57` in your Linux terminal or Google search bar to get the factorization of any number. But before computers were invented, it was more practical to use *factorization tables*: special books containing factorizations of the first N numbers.

We can also use this approach to compute these lookup tables [during compile time](#). To save space, we can store only the smallest divisor of a number. Since the smallest divisor does not exceed the \sqrt{n} , we need just one byte per a 16-bit integer:

```
template <int N = (1<<16)>
struct Precalc {
    unsigned char divisor[N];

    constexpr Precalc() : divisor{} {
        for (int i = 0; i < N; i++)
            divisor[i] = 1;
```

```

        for (int i = 2; i * i < N; i++)
            if (divisor[i] == 1)
                for (int k = i * i; k < N; k += i)
                    divisor[k] = i;
    }
};

constexpr Precalc P{};

u64 find_factor(u64 n) {
    return P.divisor[n];
}

```

With this approach, we can process 3M 16-bit integers per second, although it would probably get slower for larger numbers. While it requires just a few milliseconds and 64KB of memory to calculate and store the divisors of the first 2^{16} numbers, it does not scale well for larger inputs.

Wheel factorization

To save paper space, pre-computer era factorization tables typically excluded numbers divisible by 2 and 5, making the factorization table $\frac{1}{2} \times \frac{1}{5} = 0.4$ of its original size. In the decimal numeral system, you can quickly determine whether a number is divisible by 2 or 5 (by looking at its last digit) and keep dividing the number n by 2 or 5 while it is possible, eventually arriving at some entry in the factorization table.

We can apply a similar trick to trial division by first checking if the number is divisible by 2 and then only considering odd divisors:

```

u64 find_factor(u64 n) {
    if (n % 2 == 0)
        return 2;
    for (u64 d = 3; d * d <= n; d += 2)
        if (n % d == 0)
            return d;
    return 1;
}

```

With 50% fewer divisions to perform, this algorithm works twice as fast.

This method can be extended: if the number is not divisible by 3, we can also ignore all multiples of 3, and the same goes for all other divisors. The problem is, as we increase the number of primes to exclude, it becomes less straightforward to iterate only over the numbers not divisible by them as they follow an irregular pattern — unless the number of primes is small.

For example, if we consider 2, 3, and 5, then, among the first 90 numbers, we only need to check:

```
(1,) 7, 11, 13, 17, 19, 23, 29,
31, 37, 41, 43, 47, 49, 53, 59,
61, 67, 71, 73, 77, 79, 83, 89...
```

You can notice a pattern: the sequence repeats itself every 30 numbers. This is not surprising since the remainder modulo $2 \times 3 \times 5 = 30$ is all we need to determine whether a number is divisible by

2, 3, or 5. This means that we only need to check 8 numbers with specific remainders out of every 30, proportionally improving the performance:

```
u64 find_factor(u64 n) {
    for (u64 d : {2, 3, 5})
        if (n % d == 0)
            return d;
    u64 offsets[] = {0, 4, 6, 10, 12, 16, 22, 24};
    for (u64 d = 7; d * d <= n; d += 30) {
        for (u64 offset : offsets) {
            u64 x = d + offset;
            if (n % x == 0)
                return x;
        }
    }
    return 1;
}
```

As expected, it works $\frac{30}{8} = 3.75$ times faster than the naive trial division, processing about 7.6k 30-bit numbers per second. The performance can be improved further by considering more primes, but the returns are diminishing: adding a new prime p reduces the number of iterations by $\frac{1}{p}$ but increases the size of the skip-list by a factor of p , requiring proportionally more memory.

Precomputed Primes

If we keep increasing the number of primes in wheel factorization, we eventually exclude all composite numbers and only check for prime factors. In this case, we don't need this array of offsets but just the array of primes:

```
const int N = (1 << 16);

struct Precalc {
    u16 primes[6542]; // # of primes under N=2^16

    constexpr Precalc() : primes{} {
        bool marked[N] = {};
        int n_primes = 0;

        for (int i = 2; i < N; i++) {
            if (!marked[i]) {
                primes[n_primes++] = i;
                for (int j = 2 * i; j < N; j += i)
                    marked[j] = true;
            }
        }
    }
};

constexpr Precalc P{};
```

```

u64 find_factor(u64 n) {
    for (u16 p : P.primes)
        if (n % p == 0)
            return p;
    return 1;
}

```

This approach lets us process almost 20k 30-bit integers per second, but it does not work for larger (64-bit) numbers unless they have small ($< 2^{16}$) factors.

Note that this is actually an asymptotic optimization: there are $O(\frac{n}{\ln n})$ primes among the first n numbers, so this algorithm performs $O(\frac{\sqrt{n}}{\ln \sqrt{n}})$ operations, while wheel factorization only eliminates a large but constant fraction of divisors. If we extend it to 64-bit numbers and precompute every prime under 2^{32} (storing which would require several hundred megabytes of memory), the relative speedup would grow by a factor of $\frac{\ln \sqrt{n^2}}{\ln \sqrt{n}} = 2 \cdot \frac{1/2}{1/2} \cdot \frac{\ln n}{\ln n} = 2$.

All variants of trial division, including this one, are bottlenecked by the speed of integer division, which can be [optimized](#) if we know the divisors in advance and allow for some additional precomputation. In our case, it is suitable to use [the Lemire division check](#):

```

// ...precomputation is the same as before,
// but we store the reciprocal instead of the prime number itself
u64 magic[6542];
// for each prime i:
magic[n_primes++] = u64(-1) / i + 1;

u64 find_factor(u64 n) {
    for (u64 m : P.magic)
        if (m * n < m)
            return u64(-1) / m + 1;
    return 1;
}

```

This makes the algorithm $\sim 18x$ faster: we can now factorize $\sim 350k$ 30-bit numbers per second, which is actually the most efficient algorithm we have for this number range. While it can probably be optimized even further by performing these checks in parallel with [SIMD](#), we will stop there and try a different, asymptotically better approach.

Pollard's Rho Algorithm

Pollard's rho is a randomized $O(\sqrt[4]{n})$ integer factorization algorithm that makes use of the [birthday paradox](#):

One only needs to draw $d = \Theta(\sqrt{n})$ random numbers between 1 and n to get a collision with high probability.

The reasoning behind it is that each of the d added element has a $\frac{d}{n}$ chance of colliding with some other element, implying that the expected number of collisions is $\frac{d^2}{n}$. If d is asymptotically smaller than \sqrt{n} , then this ratio grows to zero as $n \rightarrow \infty$, and to infinity otherwise.

The trajectory of an element resembles the greek letter ρ (rho), which is what the algorithm is named after

Figure 2: The trajectory of an element resembles the greek letter ρ (rho), which is what the algorithm is named after

Consider some function $f(x)$ that takes a remainder $x \in [0, n]$ and maps it to some other remainder of n in a way that seems random from the number theory point of view. Specifically, we will use $f(x) = x^2 + 1 \bmod n$, which is random enough for our purposes.

Now, consider a graph where each number-vertex x has an edge pointing to $f(x)$. Such graphs are called *functional*. In functional graphs, the “trajectory” of any element — the path we walk if we start from that element and keep following the edges — is a path that eventually loops around (because the set of vertices is limited, and at some point, we have to go to a vertex we have already visited).

Consider a trajectory of some particular element x_0 :

$$x_0, f(x_0), f(f(x_0)), \dots$$

Let’s make another sequence out of this one by reducing each element modulo p , the smallest prime divisor of n .

Lemma. The expected length of the reduced sequence before it turns into a cycle is $O(\sqrt[4]{n})$.

Proof: Since p is the smallest divisor, $p \leq \sqrt{n}$. Each time we follow a new edge, we essentially generate a random number between 0 and p (we treat f as a “deterministically-random” function). The birthday paradox states that we only need to generate $O(\sqrt{p}) = O(\sqrt[4]{n})$ numbers until we get a collision and thus enter a loop.

Since we don’t know p , this mod- p sequence is only imaginary, but if find a cycle in it — that is, i and j such that

$$f^i(x_0) \equiv f^j(x_0) \pmod{p}$$

then we can also find p itself as

$$p = \gcd(|f^i(x_0) - f^j(x_0)|, n)$$

The algorithm itself just finds this cycle and p using this GCD trick and Floyd’s “[tortoise and hare](#)” algorithm: we maintain two pointers i and $j = 2i$ and check that

$$\gcd(|f^i(x_0) - f^j(x_0)|, n) \neq 1$$

which is equivalent to comparing $f^i(x_0)$ and $f^j(x_0)$ modulo p . Since j (hare) is increasing at twice the rate of i (tortoise), their difference is increasing by 1 each iteration and eventually will become equal to (or a multiple of) the cycle length, with i and j pointing to the same elements. And as we proved half a page ago, reaching a cycle would only require $O(\sqrt[4]{n})$ iterations:

```

u64 f(u64 x, u64 mod) {
    return ((u128) x * x + 1) % mod;
}

u64 diff(u64 a, u64 b) {
    // a and b are unsigned and so is their difference, so we can't just call abs(a - b)
    return a > b ? a - b : b - a;
}

const u64 SEED = 42;

u64 find_factor(u64 n) {
    u64 x = SEED, y = SEED, g = 1;
    while (g == 1) {
        x = f(f(x, n), n); // advance x twice
        y = f(y, n);         // advance y once
        g = gcd(diff(x, y));
    }
    return g;
}

```

While it processes only $\sim 25k$ 30-bit integers — which is almost 15 times slower than by checking each prime using a fast division trick — it dramatically outperforms every $\tilde{O}(\sqrt{n})$ algorithm for 60-bit numbers, factorizing around 90 of them per second.

Pollard-Brent Algorithm

Floyd's cycle-finding algorithm has a problem in that it moves iterators more than necessary: at least half of the vertices are visited one additional time by the slower iterator.

One way to solve it is to memorize the values x_i that the faster iterator visits and, every two iterations, compute the GCD using the difference of x_i and $x_{\lfloor i/2 \rfloor}$. But it can also be done without extra memory using a different principle: the tortoise doesn't move on every iteration, but it gets reset to the value of the faster iterator when the iteration number becomes a power of two. This lets us save additional iterations while still using the same GCD trick to compare x_i and $x_{2^{\lfloor \log_2 i \rfloor}}$ on each iteration:

```

u64 find_factor(u64 n) {
    u64 x = SEED;

    for (int l = 256; l < (1 << 20); l *= 2) {
        u64 y = x;
        for (int i = 0; i < l; i++) {
            x = f(x, n);
            if (u64 g = gcd(diff(x, y), n); g != 1)
                return g;
        }
    }
}

```

```

    return 1;
}

```

Note that we also set an upper limit on the number of iterations so that the algorithm finishes in a reasonable amount of time and returns 1 if n turns out to be a prime.

It actually does *not* improve performance and even makes the algorithm $\sim 1.5x$ *slower*, which probably has something to do with the fact that x is stale. It spends most of the time computing the GCD and not advancing the iterator — in fact, the time requirement of this algorithm is currently $O(\sqrt[4]{n} \log n)$ because of it.

Instead of [optimizing the GCD itself](#), we will optimize the number of its invocations. We can use the fact that if one of a and b contains factor p , then $a \cdot b \bmod n$ will also contain it, so instead of computing $\gcd(a, n)$ and $\gcd(b, n)$, we can compute $\gcd(a \cdot b \bmod n, n)$. This way, we can group the calculations of GCP in groups of $M = O(\log n)$ we remove $\log n$ out of the asymptotic:

```

const int M = 1024;

u64 find_factor(u64 n) {
    u64 x = SEED;

    for (int l = M; l < (1 << 20); l *= 2) {
        u64 y = x, p = 1;
        for (int i = 0; i < l; i += M) {
            for (int j = 0; j < M; j++) {
                y = f(y, n);
                p = (u128) p * diff(x, y) % n;
            }
            if (u64 g = gcd(p, n); g != 1)
                return g;
        }
    }
}

return 1;
}

```

Now it performs 425 factorizations per second, bottlenecked by the speed of modulo.

Optimizing the Modulo

The final step is to apply [Montgomery multiplication](#). Since the modulo is constant, we can perform all computations — advancing the iterator, multiplication, and even computing the GCD — in the Montgomery space where reduction is cheap:

```

struct Montgomery {
    u64 n, nr;

    Montgomery(u64 n) : n(n) {
        nr = 1;
        for (int i = 0; i < 6; i++)
            nr *= 2 - n * nr;
    }
}

```

```

    }

u64 reduce(u128 x) const {
    u64 q = u64(x) * nr;
    u64 m = ((u128) q * n) >> 64;
    return (x >> 64) + n - m;
}

u64 multiply(u64 x, u64 y) {
    return reduce((u128) x * y);
}
};

u64 f(u64 x, u64 a, Montgomery m) {
    return m.multiply(x, x) + a;
}

const int M = 1024;

u64 find_factor(u64 n, u64 x0 = 2, u64 a = 1) {
    Montgomery m(n);
    u64 x = SEED;

    for (int l = M; l < (1 << 20); l *= 2) {
        u64 y = x, p = 1;
        for (int i = 0; i < l; i += M) {
            for (int j = 0; j < M; j++) {
                x = f(x, m);
                p = m.multiply(p, diff(x, y));
            }
            if (u64 g = gcd(p, n); g != 1)
                return g;
        }
    }
    return 1;
}

```

This implementation can process around 3k 60-bit integers per second, which is ~3x faster than what [PARI](#) / [SageMath's factor](#) / `cat semiprimes.txt | time factor` measures.

Further Improvements

Optimizations. There is still a lot of potential for optimization in our implementation of the Pollard's algorithm:

- We could probably use a better cycle-finding algorithm, exploiting the fact that the graph is random. For example, there is little chance that we enter the loop in within the first few iterations (the length of the cycle and the path we walk before entering it should be equal

in expectation since before we loop around, we choose the vertex of the path we've walked independently), so we may just advance the iterator for some time before starting the trials with the GCD trick.

- Our current approach is bottlenecked by advancing the iterator (the latency of Montgomery multiplication is much higher than its reciprocal throughput), and while we are waiting for it to complete, we could perform more than just one trial using the previous values.
- If we run p independent instances of the algorithm with different seeds in parallel and stop when one of them finds the answer, it would finish \sqrt{p} times faster (the reasoning is similar to the Birthday paradox; try to prove it yourself). We don't have to use multiple cores for that: there is a lot of untapped [instruction-level parallelism](#), so we could concurrently run two or three of the same operations on the same thread, or use [SIMD](#) instructions to perform 4 or 8 multiplications in parallel.

I would not be surprised to see another 3x improvement and throughput of $\sim 10\text{k/sec}$. If you [implement](#) some of these ideas, please [let me know](#).

Errors. Another aspect that we need to handle in a practical implementation is possible errors. Our current implementation has a 0.7% error rate for 60-bit integers, and it grows higher if the numbers are lower. These errors come from three main sources:

- A cycle simply not being found (the algorithm is inherently random, and there is no guarantee that it will be found). In this case, we need to perform a primality test and optionally start again.
- The p variable becoming zero (because both p and q can get into the product). It becomes increasingly more likely as we decrease size of the inputs or increase the constant M . In this case, we need to either restart the process or (better) roll back the last M iterations and perform the trials one by one.
- Overflows in the Montgomery multiplication. Our current implementation is pretty loose with them, and if n is large, we need to add more $x > \text{mod} ? x - \text{mod} : x$ kind of statements to deal with overflows.

Larger numbers. These issues become less important if we exclude small numbers and numbers with small prime factors using the algorithms we've implemented before. In general, the optimal approach should depend on the size of the numbers:

- Smaller than 2^{16} : use a lookup table;
- Smaller than 2^{32} : use a list of precomputed primes with a fast divisibility check;
- Smaller than 2^{64} or so: use Pollard's rho algorithm with Montgomery multiplication;
- Smaller than 10^{50} : switch to [Lenstra elliptic curve factorization](#);
- Smaller than 10^{100} : switch to [Quadratic Sieve](#);
- Larger than 10^{100} : switch to [General Number Field Sieve](#).

The last three approaches are very different from what we've been doing and require much more advanced number theory, and they deserve an article (or a full-length university course) of their own.

Optimizing Logistic Regression

In machine learning, perhaps the most popular way of doing black-box classification is logistic regression.

Numbers that some Canadian students wrote on test blanks make MNIST dataset

Figure 3: Numbers that some Canadian students wrote on test blanks make MNIST dataset

Say, we want to classify 28×28 black-and-white pictures of digits into one of 10 categories (0..9):

Computationally, it works like this. If we are working with n -dimensional data, which we need to classify into one of m classes, then we multiply the input vector by a parameter matrix of size $n \times m$, and then apply a special “softmax” function to the output m -element vector:

$$\text{softmax}(x)_k = \frac{e^{x_k}}{\sum_i^m e_{x_i}}$$

In other words, what this function does is it calculates elementwise exponent of its input vector and then normalized it so that the elements add up to 1. Since the output has m positive elements that add up to 1, it can be treated as a probability distribution that the sample belongs to a certain class. We can then look at the highest probability prediction and take it as the answer of the model.

We look at a large dataset of samples and fit our parameter matrix so that we get the most answers correct. We will not go into detail about how to fit it, but once we did, we need to *inference* it, that is, to feed it new data and get predictions.

This is pretty much all that a performance engineer needs to know about machine learning. For now, we are only concerned with the computational side of things.

```
float w[10][28*28];
// 796 x 10
int predict(float a) {
    float s[10] = {0};

    for (int k = 0; k < 10; k++)
        for (int i = 0; i < 28*28; i++)
            s[k] += a[i] * w[k][i];

    // there is not problem with calculating exponent of small numbers,
    // but exponent of large numbers may overflow
    int mx = *std::max_element(s, s + 10);
    float sumexp = 0;
    for (int i = 0; i < 10; i++) {
        s[i] = exp(s[i] - mx);
        sumexp += s[i];
    }

    int argmax = 0;

    for (int i = 0; i < 10; i++) {
        s[i] /= sumexp;
        if (s[i] > s[argmax])
            argmax = i;
    }
}
```

```

    }

    return argmax;
}

```

This isn't exactly worth optimizing, because that's just $\sim 10k$ operations anyway, but our use case could be bigger. For example, neural networks are not fundamentally different: they just use longer chains of transformations, and not just matrix multiplication followed by a softmax.

This can also be used in a hot spot. For example, computer chess programs use similar models to determine the value of a position (the probability of winning). By the way, this is how "1-3-3-5-9" heuristic approach: you can train a logistic regression on a large dataset of chess positions that are turned into piece count differences, and that's what weights are going to look like. Score in other games works in a similar way.

The first thing we can notice is that we don't actually need to implement softmax, because we can notice that the largest logit (this is how pre-softmax numbers are called) will be largest after the softmax, so we only need to take argmax after the matrix multiplication.

Nobody in their sane mind uses C++ for training ML models.

Quantization

Machine learning is one of the cases where we need neither range nor precision. The whole point of machine learning is to learn functions that are robust to small perturbations in data. The input data is noisy, so why our computations shouldn't be? Plus, errors should cancel each other. We can also force the matrix parameters to be in a certain range.

Using lower precision has two advantages:

1. It takes less time to fetch data.
2. We can use SIMD instructions that packs more values together.

```

char w[10][28*28];

int predict(char a) {
    short max = 0, argmax = 0;

    for (int k = 0; k < 10; k++) {
        short s = 0;
        for (int i = 0; i < 28*28; i++)
            s += a[i] * w[k][i];
        if (s > max)
            s = 0, argmax = k;
    }

    return argmax;
}

```

Sorting

Argmin with SIMD

Computing the *minimum* of an array is [easily vectorizable](#), as it is not different from any other reduction: in AVX2, you just need to use a convenient `_mm256_min_epu32` intrinsic as the inner operation. It computes the minimum of two 8-element vectors in one cycle — even faster than in the scalar case, which requires at least a comparison and a conditional move.

Finding the *index* of that minimum element (*argmin*) is much harder, but it is still possible to vectorize very efficiently. In this section, we design an algorithm that computes the argmin (almost) at the speed of computing the minimum and $\sim 15x$ faster than the naive scalar approach.

Scalar Baseline

For our benchmark, we create an array of random 32-bit integers, and then repeatedly try to find the index of the minimum among them (the first one if it isn't unique):

```
const int N = (1 << 16);
alignas(32) int a[N];

for (int i = 0; i < N; i++)
    a[i] = rand();
```

For the sake of exposition, we assume that N is a power of two, and run all our experiments for $N = 2^{13}$ so that the [memory bandwidth](#) is not a concern.

To implement argmin in the scalar case, we just need to maintain the index instead of the minimum value:

```
int argmin(int *a, int n) {
    int k = 0;

    for (int i = 0; i < n; i++)
        if (a[i] < a[k])
            k = i;

    return k;
}
```

It works at around 1.5 GFLOPS — meaning $1.5 \cdot 10^9$ values per second processed on average, or about 0.75 values per cycle (the CPU is clocked at 2GHz).

Let's compare it to `std::min_element`:

```
int argmin(int *a, int n) {
    int k = std::min_element(a, a + n) - a;
    return k;
}
```

The version from GCC gives ~ 0.28 GFLOPS — apparently, the compiler couldn't pierce through all the abstractions. Another reminder to never use STL.

Vector of Indices

The problem with vectorizing the scalar implementation is that there is a dependency between consequent iterations. When we optimized [array sum](#), we faced the same problem, and we solved it by splitting the array into 8 slices, each representing a subset of its indices with the same remainder modulo 8. We can apply the same trick here, except that we also have to take array indices into account.

When we have the consecutive elements and their indices in vectors, we can process them in parallel using [predication](#):

```
typedef __m256i reg;

int argmin(int *a, int n) {
    // indices on the current iteration
    reg cur = _mm256_setr_epi32(0, 1, 2, 3, 4, 5, 6, 7);
    // the current minimum for each slice
    reg min = _mm256_set1_epi32(INT_MAX);
    // its index (argmin) for each slice
    reg idx = _mm256_setzero_si256();

    for (int i = 0; i < n; i += 8) {
        // load a new SIMD block
        reg x = _mm256_load_si256((reg*) &a[i]);
        // find the slices where the minimum is updated
        reg mask = _mm256_cmpgt_epi32(min, x);
        // update the indices
        idx = _mm256_blendv_epi8(idx, cur, mask);
        // update the minimum (can also similarly use a "blend" here, but min is faster)
        min = _mm256_min_epi32(x, min);
        // update the current indices
        const reg eight = _mm256_set1_epi32(8);
        cur = _mm256_add_epi32(cur, eight);           //
        // can also use a "blend" here, but min is faster
    }

    // find the argmin in the "min" register and return its real index

    int min_arr[8], idx_arr[8];

    _mm256_storeu_si256((reg*) min_arr, min);
    _mm256_storeu_si256((reg*) idx_arr, idx);

    int k = 0, m = min_arr[0];

    for (int i = 1; i < 8; i++)
        if (min_arr[i] < m)
            m = min_arr[k = i];
```

```

    return idx_arr[k];
}

```

It works at around 8-8.5 GFLOPS. There is still some inter-dependency between the iterations, so we can optimize it by considering more than 8 elements per iteration and taking advantage of the [instruction-level parallelism](#).

This would help performance a lot, but not enough to match the speed of computing the minimum (~24 GFLOPS) because there is another bottleneck. On each iteration, we need a load-fused comparison, a load-fused minimum, a blend, and an addition — that is 4 instructions in total to process 8 elements. Since the decode width of this CPU (Zen 2) is just 4, the performance will still be limited by $8 \times 2 = 16$ GFLOPS even if we somehow got rid of all the other bottlenecks.

Instead, we will switch to another approach that requires fewer instructions per element.

Branches Aren't Scary

When we run the scalar version, how often do we update the minimum?

Intuition tells us that, if all the values are drawn independently at random, then the event when the next element is less than all the previous ones shouldn't be frequent. More precisely, it equals the reciprocal of the number of processed elements. Therefore, the expected number of times the $a[i] < a[k]$ condition is satisfied equals the sum of the harmonic series:

$$\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} = O(\ln(n))$$

So the minimum is updated around 5 times for a hundred-element array, 7 for a thousand-element, and just 14 for a million-element array — which isn't large at all when looked at as a fraction of all is-new-minimum checks.

The compiler probably couldn't figure it out on its own, so let's [explicitly provide](#) this information:

```

int argmin(int *a, int n) {
    int k = 0;

    for (int i = 0; i < n; i++)
        if (a[i] < a[k]) [[unlikely]]
            k = i;

    return k;
}

```

The compiler [optimized the machine code layout](#), and the CPU is now able to execute the loop at around 2 GFLOPS — a slight but sizeable improvement from 1.5 GFLOPS of the non-hinted loop.

Here is the idea: if we are only updating the minimum a dozen or so times during the entire computation, we can ditch all the vector-blending and index updating and just maintain the minimum and regularly check if it has changed. Inside this check, we can use however slow method of updating the argmin we want because it will only be called a few times.

To implement it with SIMD, all we need to do on each iteration is a vector load, a comparison, and a test-if-zero:

```

int argmin(int *a, int n) {
    int min = INT_MAX, idx = 0;

    reg p = _mm256_set1_epi32(min);

    for (int i = 0; i < n; i += 8) {
        reg y = _mm256_load_si256((reg*) &a[i]);
        reg mask = _mm256_cmpgt_epi32(p, y);
        if (!_mm256_testz_si256(mask, mask)) { [[unlikely]]
            for (int j = i; j < i + 8; j++)
                if (a[j] < min)
                    min = a[idx = j];
            p = _mm256_set1_epi32(min);
        }
    }

    return idx;
}

```

It already performs at ~8.5 GFLOPS, but now the loop is bottlenecked by the `testz` instruction which only has a throughput of one. The solution is to load two consecutive SIMD blocks and use the minimum of them so that the `testz` effectively processes 16 elements in one go:

```

int argmin(int *a, int n) {
    int min = INT_MAX, idx = 0;

    reg p = _mm256_set1_epi32(min);

    for (int i = 0; i < n; i += 16) {
        reg y1 = _mm256_load_si256((reg*) &a[i]);
        reg y2 = _mm256_load_si256((reg*) &a[i + 8]);
        reg y = _mm256_min_epi32(y1, y2);
        reg mask = _mm256_cmpgt_epi32(p, y);
        if (!_mm256_testz_si256(mask, mask)) { [[unlikely]]
            for (int j = i; j < i + 16; j++)
                if (a[j] < min)
                    min = a[idx = j];
            p = _mm256_set1_epi32(min);
        }
    }

    return idx;
}

```

This version works in ~10 GFLOPS. To remove the other obstacles, we can do two things:

- Increase the block size to 32 elements to allow for more instruction-level parallelism.
- Optimize the local argmin: instead of calculating its exact location, we can just save the index of the block and then come back at the end and find it just once. This lets us only compute the minimum on each positive check and broadcast it to a vector, which is simpler and much

faster.

With these two optimizations implemented, the performance increases to a whopping ~22 GFLOPS:

```
int argmin(int *a, int n) {
    int min = INT_MAX, idx = 0;

    reg p = _mm256_set1_epi32(min);

    for (int i = 0; i < n; i += 32) {
        reg y1 = _mm256_load_si256((reg*) &a[i]);
        reg y2 = _mm256_load_si256((reg*) &a[i + 8]);
        reg y3 = _mm256_load_si256((reg*) &a[i + 16]);
        reg y4 = _mm256_load_si256((reg*) &a[i + 24]);
        y1 = _mm256_min_epi32(y1, y2);
        y3 = _mm256_min_epi32(y3, y4);
        y1 = _mm256_min_epi32(y1, y3);
        reg mask = _mm256_cmpgt_epi32(p, y1);
        if (!_mm256_testz_si256(mask, mask)) { [[unlikely]]
            idx = i;
            for (int j = i; j < i + 32; j++)
                min = (a[j] < min ? a[j] : min);
            p = _mm256_set1_epi32(min);
        }
    }

    for (int i = idx; i < idx + 31; i++)
        if (a[i] == min)
            return i;

    return idx + 31;
}
```

This is almost as high as it can get as just computing the minimum itself works at around 24-25 GFLOPS.

The only problem of all these branch-happy SIMD implementations is that they rely on the minimum being updated very infrequently. This is true for random input distributions, but not in the worst case. If we fill the array with a sequence of decreasing numbers, the performance of the last implementation drops to about 2.7 GFLOPS — almost 10 times as slow (although still faster than the scalar code because we only calculate the minimum on each block).

One way to fix this is to do the same thing that the quicksort-like randomized algorithms do: just shuffle the input yourself and iterate over the array in random order. This lets you avoid this worst-case penalty, but it is tricky to implement due to RNG- and [memory](#)-related issues. There is a simpler solution.

Find the Minimum, Then Find the Index

We know how to [calculate the minimum of an array](#) fast and how to [find an element in an array](#) fast — so why don't we just separately compute the minimum and then find it?

```
int argmin(int *a, int n) {
    int needle = min(a, n);
    int idx = find(a, n, needle);
    return idx;
}
```

If we implement the two subroutines optimally (check the linked articles), the performance will be ~18 GFLOPS for random arrays and ~12 GFLOPS for decreasing arrays — which makes sense as we are expected to read the array 1.5 and 2 times respectively. This isn't that bad by itself — at least we avoid the 10x worst-case performance penalty — but the problem is that this penalized performance also translates to larger arrays, when we are bottlenecked by the [memory bandwidth](#) rather than compute.

Luckily, we already know how to fix it. We can split the array into blocks of fixed size B and compute the minima on these blocks while also maintaining the global minimum. When the minimum on a new block is lower than the global minimum, we update it and also remember the block number of where the global minimum currently is. After we've processed the entire array, we just return to that block and scan through its B elements to find the argmin.

This way we only process $(N + B)$ elements and don't have to sacrifice neither $\frac{1}{2}$ nor $\frac{1}{3}$ of the performance:

```
const int B = 256;

// returns the minimum and its first block
pair<int, int> approx_argmin(int *a, int n) {
    int res = INT_MAX, idx = 0;
    for (int i = 0; i < n; i += B) {
        int val = min(a + i, B);
        if (val < res) {
            res = val;
            idx = i;
        }
    }
    return {res, idx};
}

int argmin(int *a, int n) {
    auto [needle, base] = approx_argmin(a, n);
    int idx = find(a + base, B, needle);
    return base + idx;
}
```

This results for the final implementation are ~22 and ~19 GFLOPS for random and decreasing arrays respectively.

The full implementation, including both `min()` and `find()`, is about 100 lines long. [Take a look](#) if you want, although it is still far from being production-grade.

Summary

Here are the results combined for all implementations:

algorithm	rand	decr	reason for the performance difference
std	0.28	0.28	
scalar	1.54	1.89	efficient branch prediction
+ hinted	1.95	0.75	wrong hint
index	8.17	8.12	
simd	8.51	1.65	scalar-based argmin on each iteration
+ ilp	10.22	1.74	^ same
+ optimized	22.44	2.70	^ same, but faster because there are less inter-dependencies
min+find	18.21	12.92	find() has to scan the entire array
+ blocked	22.23	19.29	we still have an optional horizontal minimum every B elements

Take these results with a grain of salt: the measurements are [quite noisy](#), they were done for just two input distributions, for a specific array size ($N = 2^{13}$, the size of the L1 cache), for a specific architecture (Zen 2), and for a specific and slightly outdated compiler (GCC 9.3) — the compiler optimizations were also very fragile to little changes in the benchmarking code.

There are also still some minor things to optimize, but the potential improvement is less than 10% so I didn't bother. One day I may pluck up the courage, optimize the algorithm to the theoretical limit, handle the non-divisible-by-block-size array sizes and non-aligned memory cases, and then re-run the benchmarks properly on many architectures, with p-values and such. In case someone does it before me, please [ping me back](#).

Acknowledgements

The first, index-based SIMD algorithm was [originally designed](#) by Wojciech Muła in 2018.

Thanks to Zach Wegner for [pointing out](#) that the performance of the Muła's algorithm is improved when implemented manually using intrinsics (I originally used the [GCC vector types](#)).

After publication, I've discovered that [Marshall Lochbaum](#), the creator of [BQN](#), designed a [very similar algorithm](#) while he was working on Dyalog APL in 2019. Pay more attention to the world of array programming languages!

Prefix Sum with SIMD

The *prefix sum*, also known as *cumulative sum*, *inclusive scan*, or simply *scan*, is a sequence of numbers b_i generated from another sequence a_i using the following rule:

$$\begin{aligned} b_0 &= a_0 \\ b_1 &= a_0 + a_1 \\ b_2 &= a_0 + a_1 + a_2 \\ &\dots \end{aligned}$$

In other words, the k -th element of the output sequence is the sum of the first k elements of the input sequence.

Prefix sum is a very important primitive in many algorithms, especially in the context of parallel algorithms, where its computation scales almost perfectly with the number of processors. Unfortunately, it is much harder to speed up with SIMD parallelism on a single CPU core, but we will try it nonetheless — and derive an algorithm that is $\sim 2.5x$ faster than the baseline scalar implementation.

Baseline

For our baseline, we could just invoke `std::partial_sum` from the STL, but for clarity, we will implement it manually. We create an array of integers and then sequentially add the previous element to the current one:

```
void prefix(int *a, int n) {
    for (int i = 1; i < n; i++)
        a[i] += a[i - 1];
}
```

It seems like we need two reads, an add, and a write on each iteration, but of course, the compiler optimizes the extra read away and uses a register as the accumulator:

```
loop:
    add    edx, DWORD PTR [rax]
    mov    DWORD PTR [rax-4], edx
    add    rax, 4
    cmp    rax, rcx
    jne    loop
```

After [unrolling](#) the loop, just two instructions effectively remain: the fused read-add and the write-back of the result. Theoretically, these should work at 2 GFLOPS (1 element per CPU cycle, by the virtue of [superscalar processing](#)), but since the memory system has to constantly [switch](#) between reading and writing, the actual performance is between 1.2 and 1.6 GFLOPS, depending on the array size.

Vectorization

One way to implement a parallel prefix sum algorithm is to split the array into small blocks, independently calculate *local* prefix sums on them, and then do a second pass where we adjust the computed values in each block by adding the sum of all previous elements to them.

This allows processing each block in parallel — both during the computation of the local prefix sums and the accumulation phase — so you usually split the array into as many blocks as you have processors. But since we are only allowed to use one CPU core, and [non-sequential memory access](#) in SIMD doesn't work well, we are not going to do that. Instead, we will use a fixed block size equal to the size of a SIMD lane and calculate prefix sums within a register.

Now, to compute these prefix sums locally, we are going to use another parallel prefix sum method that is generally inefficient (the total work is $O(n \log n)$ instead of linear) but is good enough for the case when the data is already in a SIMD register. The idea is to perform $\log n$ iterations where on k -th iteration, we add a_{i-2^k} to a_i for all applicable i :

```

for (int l = 0; l < logn; l++)
    // (atomically and in parallel):
    for (int i = (1 << l); i < n; i++)
        a[i] += a[i - (1 << l)];

```

We can prove that this algorithm works by induction: if on k -th iteration every element a_i is equal to the sum of the $(i - 2^k, i]$ segment of the original array, then after adding a_{i-2^k} to it, it will be equal to the sum of $(i - 2^{k+1}, i]$. After $O(\log n)$ iterations, the array will turn into its prefix sum.

To implement it in SIMD, we could use [permutations](#) to place i -th element against $(i - 2^k)$ -th, but they are too slow. Instead, we will use the `slli` (“shift lanes left”) instruction that does exactly that and also replaces the unmatched elements with zeros:

```

typedef __m128i v4i;

v4i prefix(v4i x) {
    // x = 1, 2, 3, 4
    x = _mm_add_epi32(x, _mm_slli_si128(x, 4));
    // x = 1, 2, 3, 4
    // + 0, 1, 2, 3
    // = 1, 3, 5, 7
    x = _mm_add_epi32(x, _mm_slli_si128(x, 8));
    // x = 1, 3, 5, 7
    // + 0, 0, 1, 3
    // = 1, 3, 6, 10
    return x;
}

```

Unfortunately, the 256-bit version of this instruction performs this byte shift independently within two 128-bit lanes, which is typical to AVX:

```

typedef __m256i v8i;

v8i prefix(v8i x) {
    // x = 1, 2, 3, 4, 5, 6, 7, 8
    x = _mm256_add_epi32(x, _mm256_slli_si256(x, 4));
    x = _mm256_add_epi32(x, _mm256_slli_si256(x, 8));
    x = _mm256_add_epi32(x, _mm256_slli_si256(x, 16)); // <- this does nothing
    // x = 1, 3, 6, 10, 5, 11, 18, 26
    return x;
}

```

We still can use it to compute 4-element prefix sums twice as fast, but we’ll have to switch to 128-bit SSE when accumulating. Let’s write a handy function that computes a local prefix sum end-to-end:

```

void prefix(int *p) {
    v8i x = _mm256_load_si256((v8i*) p);
    x = _mm256_add_epi32(x, _mm256_slli_si256(x, 4));
    x = _mm256_add_epi32(x, _mm256_slli_si256(x, 8));
    _mm256_store_si256((v8i*) p, x);
}

```

```
}
```

Now, for the accumulate phase, we will create another handy function that similarly takes the pointer to a 4-element block and also the 4-element vector of the previous prefix sum. The job of this function is to add this prefix sum vector to the block and update it so that it can be passed on to the next block (by broadcasting the last element of the block before the addition):

```
v4i accumulate(int *p, v4i s) {
    v4i d = (v4i) _mm_broadcast_ss((float*) &p[3]);
    v4i x = _mm_load_si128((v4i*) p);
    x = _mm_add_epi32(s, x);
    _mm_store_si128((v4i*) p, x);
    return _mm_add_epi32(s, d);
}
```

With `prefix` and `accumulate` implemented, the only thing left is to glue together our two-pass algorithm:

```
void prefix(int *a, int n) {
    for (int i = 0; i < n; i += 8)
        prefix(&a[i]);

    v4i s = _mm_setzero_si128();

    for (int i = 4; i < n; i += 4)
        s = accumulate(&a[i], s);
}
```

The algorithm already performs slightly more than twice as fast as the scalar implementation but becomes slower for large arrays that fall out of the L3 cache — roughly at half the [two-way RAM bandwidth](#) as we are reading the entire array twice.

Another interesting data point: if we only execute the `prefix` phase, the performance would be ~ 8.1 GFLOPS. The `accumulate` phase is slightly slower at ~ 5.8 GFLOPS. Sanity check: the total performance should be $\frac{1}{\frac{1}{8.1} + \frac{1}{5.8}} \approx 3.4$.

Blocking

So, we have a memory bandwidth problem for large arrays. We can avoid re-fetching the entire array from RAM if we split it into blocks that fit in the cache and process them separately. All we need to pass to the next block is the sum of the previous ones, so we can design a `local_prefix` function with an interface similar to `accumulate`:

```
const int B = 4096; // <- ideally should be slightly less or equal to the L1 cache

v4i local_prefix(int *a, v4i s) {
    for (int i = 0; i < B; i += 8)
        prefix(&a[i]);

    for (int i = 0; i < B; i += 4)
        s = accumulate(&a[i], s);
```

```

    return s;
}

void prefix(int *a, int n) {
    v4i s = _mm_setzero_si128();
    for (int i = 0; i < n; i += B)
        s = local_prefix(a + i, s);
}

```

(We have to make sure that N is a multiple of B , but we are going to ignore such implementation details for now.)

The blocked version performs considerably better, and not just for when the array is in the RAM:

The speedup in the RAM case compared to the non-blocked implementation is only ~ 1.5 and not 2. This is because the memory controller is sitting idle while we iterate over the cached block for the second time instead of fetching the next one — the [hardware prefetcher](#) isn't advanced enough to detect this pattern.

Continuous Loads

There are several ways to solve this under-utilization problem. The obvious one is to use [software prefetching](#) to explicitly request the next block while we are still processing the current one.

It is better to add prefetching to the `accumulate` phase because it is slower and less memory-intensive than `prefix`:

```

v4i accumulate(int *p, v4i s) {
    __builtin_prefetch(p + B); // <-- prefetch the next block
    // ...
    return s;
}

```

The performance slightly decreases for in-cache arrays, but approaches closer to 2 GFLOPS for the in-RAM ones:

Another approach is to do *interleaving* of the two phases. Instead of separating and alternating between them in large blocks, we can execute the two phases concurrently, with the `accumulate` phase lagging behind by a fixed number of iterations — similar to the [CPU pipeline](#):

```

const int B = 64;
//           ^ small sizes cause pipeline stalls
//           large sizes cause cache system inefficiencies

void prefix(int *a, int n) {
    v4i s = _mm_setzero_si128();

    for (int i = 0; i < B; i += 8)
        prefix(&a[i]);

    for (int i = B; i < n; i += 8) {

```

```

    prefix(&a[i]);
    s = accumulate(&a[i - B], s);
    s = accumulate(&a[i - B + 4], s);
}

for (int i = n - B; i < n; i += 4)
    s = accumulate(&a[i], s);
}

```

This has more benefits: the loop progresses at a constant speed, reducing the pressure on the memory system, and the scheduler sees the instructions of both subroutines, allowing it to be more efficient at assigning instruction to execution ports — sort of like hyper-threading, but in code.

For these reasons, the performance improves even on small arrays:

And finally, it doesn't seem that we are bottlenecked by the [memory read port](#) or the [decode width](#), so we can add prefetching for free, which improves the performance even more:

The total speedup we were able to achieve is between $\frac{4.2}{1.5} \approx 2.8$ for small arrays and $\frac{2.1}{1.2} \approx 1.75$ for large arrays.

The speedup may be higher for lower-precision data compared to the scalar code, as it is pretty much limited to executing one iteration per cycle regardless of the operand size, but it is still sort of “meh” when compared to some [other SIMD-based algorithms](#). This is largely because there isn’t a full-register byte shift in AVX that would allow the `accumulate` stage to proceed twice as fast, let alone a dedicated prefix sum instruction.

Other Relevant Work

You can read [this paper from Columbia](#) that focuses on the multi-core setting and AVX-512 (which `sort of` has a fast 512-bit register byte shift) and [this StackOverflow question](#) for a more general discussion.

Most of what I’ve described in this article was already known. To the best of my knowledge, my contribution here is the interleaving technique, which is responsible for a modest ~20% performance increase. There probably are ways to improve it further, but not by a lot.

There is also this professor at CMU, [Guy Blelloch](#), who [advocated](#) for a dedicated prefix sum hardware back in the 90s when [vector processors](#) were still a thing. Prefix sums are very important for parallel applications, and the hardware is becoming increasingly more parallel, so maybe, in the future, the CPU manufacturers will revitalize this idea and make prefix sum calculations slightly easier.

Reading Decimal Integers

I wrote a new integer parsing algorithm that is ~35x faster than `scanf`.

(No, this is not an April Fools’ joke — although it does sound ridiculous.)

Zen 2 @ 2GHz. The compiler is Clang 13.

Ridiculous.

Iostream

Scanf

Synchronization

Getchar

Buffering

SIMD

<http://0x80.pl/notesen/2014-10-12-parsing-decimal-numbers-part-1-swar.html>

Serial

Transpose-based approach

Instruction-level parallelism

Modifications

ILP benefits would not be that huge.

One huge asterisk. We get the integers, and we can even do other parsing algorithms on them.

1.75 cycles per byte.

AVX-512 both due to larger SIMD lane size and dedicated operations for filtering.

It accounts for ~2% of all time, but it can be optimized by using special procedures. Pad buffer with any digits.

Future work

Next time, we will be *writing* integers.

You can create a string searching algorithm by computing hashes in rabin-karp algorithm — although it does not seem to be possible to make an *exact* algorithm for that.

Acknowledgements

<http://0x80.pl/articles/simd-parsing-int-sequences.html>

<https://stackoverflow.com/questions/25622745/transpose-an-8x8-float-using-avx-avx2/25627536#25627536>

Algorithms Case Studies

When you try to explain a complex concept, it is generally a good idea to give a very simple and minimal example illustrating it. This is why in this book you see about a dozen different ways of calculating the sum of an array, each highlighting a certain CPU feature.

But the main purpose of this book is not to learn computer architecture just for the sake of learning it, but to acquire real-world skills in software optimization. The next two chapters exist to help you achieve this goal, as they contain detailed case studies of various algorithms that are much harder to optimize than the sum of an array.

Matrix Multiplication

In this case study, we will design and implement several algorithms for matrix multiplication.

We start with the naive “for-for-for” algorithm and incrementally improve it, eventually arriving at a version that is 50 times faster and matches the performance of BLAS libraries while being under 40 lines of C.

All implementations are compiled with GCC 13 and run on a [Zen 2](#) CPU clocked at 2GHz.

Baseline

The result of multiplying an $l \times n$ matrix A by an $n \times m$ matrix B is defined as an $l \times m$ matrix C such that:

$$C_{ij} = \sum_{k=1}^n A_{ik} \cdot B_{kj}$$

For simplicity, we will only consider *square* matrices, where $l = m = n$.

To implement matrix multiplication, we can simply transfer this definition into code, but instead of two-dimensional arrays (aka matrices), we will be using one-dimensional arrays to be explicit about pointer arithmetic:

```
void matmul(const float *a, const float *b, float *c, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i * n + j] += a[i * n + k] * b[k * n + j];
}
```

For reasons that will become apparent later, we will only use matrix sizes that are multiples of 48 for benchmarking, but the implementations remain correct for all others. We also use [32-bit floats](#) specifically, although all implementations can be easily [generalized](#) to other data types and operations.

Compiled with `g++ -O3 -march=native -ffast-math -funroll-loops`, the naive approach multiplies two matrices of size $n = 1920 = 48 \times 40$ in ~ 16.7 seconds. To put it in perspective, this is approximately $\frac{1920^3}{16.7 \times 10^9} \approx 0.42$ useful operations per nanosecond (GFLOPS), or roughly 5 CPU cycles per multiplication, which doesn’t look that good yet.

Transposition

In general, when optimizing an algorithm that processes large quantities of data — and $1920^2 \times 3 \times 4 \approx 42$ MB clearly is a large quantity as it can’t fit into any of the [CPU caches](#) — one should always start with memory before optimizing arithmetic, as it is much more likely to be the bottleneck.

The field C_{ij} can be thought of as the dot product of row i of matrix A and column j of matrix B . As we increment k in the inner loop above, we are reading the matrix a sequentially, but we are jumping over n elements as we iterate over a column of b , which is [not as fast](#) as sequential iteration.

One [well-known](#) optimization that tackles this problem is to store matrix B in *column-major* order — or, alternatively, to *transpose* it before the matrix multiplication. This requires $O(n^2)$ additional operations but ensures sequential reads in the innermost loop:

```
void matmul(const float *a, const float *_b, float *c, int n) {
    float *b = new float[n * n];

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            b[i * n + j] = _b[j * n + i];

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i * n + j] += a[i * n + k] * b[j * n + k]; // <- note the indices
}
```

This code runs in ~ 12.4 s, or about 30% faster.

As we will see in a bit, there are more important benefits to transposing it than just the sequential memory reads.

Vectorization

Now that all we do is just sequentially read the elements of **a** and **b**, multiply them, and add the result to an accumulator variable, we can use [SIMD](#) instructions to speed it all up. It is pretty straightforward to implement using [GCC vector types](#) — we can [memory-align](#) matrix rows, pad them with zeros, and then compute the multiply-sum as we would normally compute any other reduction:

```
// a vector of 256 / 32 = 8 floats
typedef float vec __attribute__(( vector_size(32) ));

// a helper function that allocates n vectors and initializes them with zeros
vec* alloc(int n) {
    vec* ptr = (vec*) std::aligned_alloc(32, 32 * n);
    memset(ptr, 0, 32 * n);
    return ptr;
}

void matmul(const float *_a, const float *_b, float *c, int n) {
    int nB = (n + 7) / 8; // number of 8-element vectors in a row (rounded up)

    vec *a = alloc(n * nB);
    vec *b = alloc(n * nB);

    // move both matrices to the aligned region
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            a[i * nB + j / 8][j % 8] = _a[i * n + j];
        }
    }
}
```

```

        b[i * nB + j / 8][j % 8] = _b[j * n + i]; // <- b is still transposed
    }
}

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        vec s{}; // initialize the accumulator with zeros

        // vertical summation
        for (int k = 0; k < nB; k++)
            s += a[i * nB + k] * b[j * nB + k];

        // horizontal summation
        for (int k = 0; k < 8; k++)
            c[i * n + j] += s[k];
    }
}

std::free(a);
std::free(b);
}

```

The performance for $n = 1920$ is now around 2.3 GFLOPS — or another ~ 4 times higher compared to the transposed but not vectorized version.

This optimization looks neither too complex nor specific to matrix multiplication. Why can't the compiler [auto-vectorize](#) the inner loop by itself?

It actually can; the only thing preventing that is the possibility that `c` overlaps with either `a` or `b`. To rule it out, you can communicate to the compiler that you guarantee `c` is not [aliased](#) with anything by adding the `__restrict__` keyword to it:

```

void matmul(const float *a, const float *_b, float * __restrict__ c, int n) {
    // ...
}

```

Both manually and auto-vectorized implementations perform roughly the same.

Memory efficiency

What is interesting is that the implementation efficiency depends on the problem size.

At first, the performance (defined as the number of useful operations per second) increases as the overhead of the loop management and the horizontal reduction decreases. Then, at around $n = 256$, it starts smoothly decreasing as the matrices stop fitting into the [cache](#) ($2 \times 256^2 \times 4 = 512$ KB is the size of the L2 cache), and the performance becomes bottlenecked by the [memory bandwidth](#).

It is also interesting that the naive implementation is mostly on par with the non-vectorized transposed version — and even slightly better because it doesn't need to perform a transposition.

One might think that there would be some general performance gain from doing sequential reads since we are fetching fewer cache lines, but this is not the case: fetching the first column of `b` indeed

takes more time, but the next 15 column reads will be in the same cache lines as the first one, so they will be cached anyway — unless the matrix is so large that it can't even fit $n * \text{cache_line_size}$ bytes into the cache, which is not the case for any practical matrix sizes.

Instead, the performance deteriorates on only a few specific matrix sizes due to the effects of [cache associativity](#): when n is a multiple of a large power of two, we are fetching the addresses of \mathbf{b} that all likely map to the same cache line, which reduces the effective cache size. This explains the 30% performance dip for $n = 1920 = 2^7 \times 3 \times 5$, and you can see an even more noticeable one for $1536 = 2^9 \times 3$: it is roughly 3 times slower than for $n = 1535$.

So, counterintuitively, transposing the matrix doesn't help with caching — and in the naive scalar implementation, we are not really bottlenecked by the memory bandwidth anyway. But our vectorized implementation certainly is, so let's work on its I/O efficiency.

Register reuse

Using a Python-like notation to refer to submatrices, to compute the cell $C[x][y]$, we need to calculate the dot product of $A[x][:]$ and $B[:][y]$, which requires fetching $2n$ elements, even if we store \mathbf{B} in column-major order.

To compute $C[x : x + 2][y : y + 2]$, a 2×2 submatrix of C , we would need two rows from A and two columns from B , namely $A[x : x + 2][:]$ and $B[:][y : y + 2]$, containing $4n$ elements in total, to update *four* elements instead of *one* — which is $\frac{2n/1}{4n/4} = 2$ times better in terms of I/O efficiency.

To avoid fetching data more than once, we need to iterate over these rows and columns in parallel and calculate all 2×2 possible combinations of products. Here is a proof of concept:

```
void kernel_2x2(int x, int y) {
    int c00 = 0, c01 = 0, c10 = 0, c11 = 0;

    for (int k = 0; k < n; k++) {
        // read rows
        int a0 = a[x][k];
        int a1 = a[x + 1][k];

        // read columns
        int b0 = b[k][y];
        int b1 = b[k][y + 1];

        // update all combinations
        c00 += a0 * b0;
        c01 += a0 * b1;
        c10 += a1 * b0;
        c11 += a1 * b1;
    }

    // write the results to C
    c[x][y]      = c00;
    c[x][y + 1]  = c01;
    c[x + 1][y]  = c10;
}
```

```

    c[x + 1][y + 1] = c11;
}

```

We can now simply call this kernel on all 2×2 submatrices of C , but we won't bother evaluating it: although this algorithm is better in terms of I/O operations, it would still not beat our SIMD-based implementation. Instead, we will extend this approach and develop a similar *vectorized* kernel right away.

Designing the kernel

Instead of designing a kernel that computes an $h \times w$ submatrix of C from scratch, we will declare a function that *updates* it using columns from l to r of A and rows from l to r of B . For now, this seems like an over-generalization, but this function interface will prove useful later.

To determine h and w , we have several performance considerations:

- In general, to compute an $h \times w$ submatrix, we need to fetch $2 \cdot n \cdot (h + w)$ elements. To optimize the I/O efficiency, we want the $\frac{h \cdot w}{h + w}$ ratio to be high, which is achieved with large and square-ish submatrices.
- We want to use the **FMA** (“fused multiply-add”) instruction available on all modern x86 architectures. As you can guess from the name, it performs the $c += a * b$ operation — which is the core of a dot product — on 8-element vectors in one go, which saves us from executing vector multiplication and addition separately.
- To achieve better utilization of this instruction, we want to make use of **instruction-level parallelism**. On Zen 2, the **fma** instruction has a latency of 5 and a throughput of 2, meaning that we need to concurrently execute at least $5 \times 2 = 10$ of them to saturate its execution ports.
- We want to avoid register spill (move data to and from registers more than necessary), and we only have 16 logical vector registers that we can use as accumulators (minus those that we need to hold temporary values).

For these reasons, we settle on a 6×16 kernel. This way, we process 96 elements at once that are stored in $6 \times 2 = 12$ vector registers. To update them efficiently, we use the following procedure:

```

// update 6x16 submatrix C[x:x+6][y:y+16]
// using A[x:x+6][l:r] and B[l:r][y:y+16]
void kernel(float *a, vec *b, vec *c, int x, int y, int l, int r, int n) {
    vec t[6][2]{};
    // will be zero-filled and stored in ymm registers

    for (int k = l; k < r; k++) {
        for (int i = 0; i < 6; i++) {
            // broadcast a[x + i][k] into a register
            vec alpha = vec{} + a[(x + i) * n + k]; // converts to a broadcast
            // multiply b[k][y:y+16] by it and update t[i][0] and t[i][1]
            for (int j = 0; j < 2; j++)
                t[i][j] += alpha * b[(k * n + y) / 8 + j]; // converts to an fma
        }
    }

    // write the results back to C
}

```

```

for (int i = 0; i < 6; i++)
    for (int j = 0; j < 2; j++)
        c[((x + i) * n + y) / 8 + j] += t[i][j];
}

```

We need `t` so that the compiler stores these elements in vector registers. We could just update their final destinations in `c`, but, unfortunately, the compiler re-writes them back to memory, causing a slowdown (wrapping everything in `__restrict__` keywords doesn't help).

After unrolling these loops and hoisting `b` out of the `i` loop (`b[(k * n + y) / 8 + j]` does not depend on `i` and can be loaded once and reused in all 6 iterations), the compiler generates something more similar to this:

```

for (int k = 1; k < r; k++) {
    __m256 b0 = _mm256_load_ps((__m256*) &b[k * n + y]);
    __m256 b1 = _mm256_load_ps((__m256*) &b[k * n + y + 8]);

    __m256 a0 = _mm256_broadcast_ps((__m128*) &a[x * n + k]);
    t00 = _mm256_fmadd_ps(a0, b0, t00);
    t01 = _mm256_fmadd_ps(a0, b1, t01);

    __m256 a1 = _mm256_broadcast_ps((__m128*) &a[(x + 1) * n + k]);
    t10 = _mm256_fmadd_ps(a1, b0, t10);
    t11 = _mm256_fmadd_ps(a1, b1, t11);

    // ...
}

```

We are using $12 + 3 = 15$ vector registers and a total of $6 \times 3 + 2 = 20$ instructions to perform $16 \times 6 = 96$ updates. Assuming that there are no other bottlenecks, we should be hitting the throughput of `_mm256_fmadd_ps`.

Note that this kernel is architecture-specific. If we didn't have `fma`, or if its throughput/latency were different, or if the SIMD width was 128 or 512 bits, we would have made different design choices. Multi-platform BLAS implementations ship [many kernels](#), each written in assembly by hand and optimized for a particular architecture.

The rest of the implementation is straightforward. Similar to the previous vectorized implementation, we just move the matrices to memory-aligned arrays and call the kernel instead of the innermost loop:

```

void matmul(const float *_a, const float *_b, float *_c, int n) {
    // to simplify the implementation, we pad the height and width
    // so that they are divisible by 6 and 16 respectively
    int nx = (n + 5) / 6 * 6;
    int ny = (n + 15) / 16 * 16;

    float *a = alloc(nx * ny);
    float *b = alloc(nx * ny);
    float *c = alloc(nx * ny);
}

```

```

for (int i = 0; i < n; i++) {
    memcpy(&a[i * ny], &_a[i * n], 4 * n);
    memcpy(&b[i * ny], &_b[i * n], 4 * n); // we don't need to transpose b this time
}

for (int x = 0; x < nx; x += 6)
    for (int y = 0; y < ny; y += 16)
        kernel(a, (vec*) b, (vec*) c, x, y, 0, n, ny);

for (int i = 0; i < n; i++)
    memcpy(&_c[i * n], &c[i * ny], 4 * n);

std::free(a);
std::free(b);
std::free(c);
}

```

This improves the benchmark performance, but only by ~40%:

The speedup is much higher (2-3x) on smaller arrays, indicating that there is still a memory bandwidth problem:

Now, if you've read the section on [cache-oblivious algorithms](#), you know that one universal solution to these types of things is to split all matrices into four parts, perform eight recursive block matrix multiplications, and carefully combine the results together. This solution is okay in practice, but there is some [overhead to recursion](#), and it also doesn't allow us to fine-tune the algorithm, so instead, we will follow a different, simpler approach.

Blocking

The *cache-aware* alternative to the divide-and-conquer trick is *cache blocking*: splitting the data into blocks that can fit into the cache and processing them one by one. If we have more than one layer of cache, we can do hierarchical blocking: we first select a block of data that fits into the L3 cache, then we split it into blocks that fit into the L2 cache, and so on. This approach requires knowing the cache sizes in advance, but it is usually easier to implement and also faster in practice.

Cache blocking is less trivial to do with matrices than with arrays, but the general idea is this:

- Select a submatrix of B that fits into the L3 cache (say, a subset of its columns).
- Select a submatrix of A that fits into the L2 cache (say, a subset of its rows).
- Select a submatrix of the previously selected submatrix of B (a subset of its rows) that fits into the L1 cache.
- Update the relevant submatrix of C using the kernel.

Here is a good [visualization](#) by Jukka Suomela (it features many different approaches; you are interested in the last one).

Note that the decision to start this process with matrix B is not arbitrary. During the kernel execution, we are reading the elements of A much slower than the elements of B : we fetch and broadcast just one element of A and then multiply it with 16 elements of B . Therefore, we want B to be in the L1 cache while A can stay in the L2 cache and not the other way around.

This sounds complicated, but we can implement it with just three more outer `for` loops, which are collectively called *macro-kernel* (and the highly optimized low-level function that updates a 6x16 submatrix is called *micro-kernel*):

```
const int s3 = 64; // how many columns of B to select
const int s2 = 120; // how many rows of A to select
const int s1 = 240; // how many rows of B to select

for (int i3 = 0; i3 < ny; i3 += s3)
    // now we are working with b[:, i3:i3+s3]
    for (int i2 = 0; i2 < nx; i2 += s2)
        // now we are working with a[i2:i2+s2]::
        for (int i1 = 0; i1 < ny; i1 += s1)
            // now we are working with b[i1:i1+s1][i3:i3+s3]
            // and we need to update c[i2:i2+s2][i3:i3+s3] with [l:r] = [i1:i1+s1]
            for (int x = i2; x < std::min(i2 + s2, nx); x += 6)
                for (int y = i3; y < std::min(i3 + s3, ny); y += 16)
                    kernel(a, (vec*) b, (vec*) c, x, y, i1, std::min(i1 + s1, n), ny);
```

Cache blocking completely removes the memory bottleneck:

The performance is no longer (significantly) affected by the problem size:

Notice that the dip at 1536 is still there: cache associativity still affects the performance. To mitigate this, we can adjust the step constants or insert holes into the layout, but we will not bother doing that for now.

Optimization

To approach closer to the performance limit, we need a few more optimizations:

- Remove memory allocation and operate directly on the arrays that are passed to the function.
Note that we don't need to do anything with `a` as we are reading just one element at a time, and we can use an `unaligned store` for `c` as we only use it rarely, so our only concern is reading `b`.
- Get rid of the `std::min` so that the size parameters are (mostly) constant and can be embedded into the machine code by the compiler (which also lets it `unroll` the micro-kernel loop more efficiently and avoid runtime checks).
- Rewrite the micro-kernel by hand using 12 vector variables (the compiler seems to struggle with keeping them in registers and writes them first to a temporary memory location and only then to `C`).

These optimizations are straightforward but quite tedious to implement, so we are not going to list the code here in the article. It also requires some more work to effectively support “weird” matrix sizes, which is why we only run benchmarks for sizes that are multiple of $48 = \frac{6 \cdot 16}{\gcd(6, 16)}$.

These individually small improvements compound and result in another 50% improvement:

We are actually not that far from the theoretical performance limit — which can be calculated as the SIMD width times the `fma` instruction throughput times the clock frequency:

$$\frac{8}{SIMD \text{ thr.}} \cdot \frac{2}{cycles/sec} \cdot \frac{2 \cdot 10^9}{GFLOPS} = 32 GFLOPS \quad (3.2 \cdot 10^{10})$$

It is more representative to compare against some practical library, such as [OpenBLAS](#). The laziest way to do it is to simply [invoke matrix multiplication from NumPy](#). There may be some minor overhead due to Python, but it ends up reaching 80% of the theoretical limit, which seems plausible (a 20% overhead is okay: matrix multiplication is not the only thing that CPUs are made for).

We've reached ~93% of BLAS performance and ~75% of the theoretical performance limit, which is really great for what is essentially just 40 lines of C.

Interestingly, the whole thing can be rolled into just one deeply nested `for` loop with a BLAS level of performance (assuming that we're in 2050 and using GCC version 35, which finally stopped screwing up with register spilling):

```
for (int i3 = 0; i3 < n; i3 += s3)
    for (int i2 = 0; i2 < n; i2 += s2)
        for (int i1 = 0; i1 < n; i1 += s1)
            for (int x = i2; x < i2 + s2; x += 6)
                for (int y = i3; y < i3 + s3; y += 16)
                    for (int k = i1; k < i1 + s1; k++)
                        for (int i = 0; i < 6; i++)
                            for (int j = 0; j < 2; j++)
                                c[x * n / 8 + i * n / 8 + y / 8 + j]
                                += (vec{} + a[x * n + i * n + k])
                                    * b[n / 8 * k + y / 8 + j];
```

There is also an approach that performs asymptotically fewer arithmetic operations — [the Strassen algorithm](#) — but it has a large constant factor, and it is only efficient for [very large matrices](#) ($n > 4000$), where we typically have to use either multiprocessing or some approximate dimensionality-reducing methods anyway.

Generalizations

FMA also supports 64-bit floating-point numbers, but it does not support integers: you need to perform addition and multiplication separately, which results in decreased performance. If you can guarantee that all intermediate results can be represented exactly as 32- or 64-bit floating-point numbers (which is [often the case](#)), it may be faster to just convert them to and from floats.

This approach can be also applied to some similar-looking computations. One example is the “min-plus matrix multiplication” defined as:

$$(A \circ B)_{ij} = \min_{1 \leq k \leq n} (A_{ik} + B_{kj})$$

It is also known as the “distance product” due to its graph interpretation: when applied to itself ($D \circ D$), the result is the matrix of shortest paths of length two between all pairs of vertices in a fully-connected weighted graph specified by the edge weight matrix D .

A cool thing about the distance product is that if we iterate the process and calculate

$$D_2 = D \circ DD_4 = D_2 \circ D_2 D_8 = D_4 \circ D_4 \dots$$

...we can find all-pairs shortest paths in $O(\log n)$ steps:

```
for (int l = 0; l < logn; l++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
```

This requires $O(n^3 \log n)$ operations. If we do these two-edge relaxations in a particular order, we can do it with just one pass, which is known as the [Floyd-Warshall algorithm](#):

```
for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
```

Interestingly, similarly vectorizing the distance product and executing it $O(\log n)$ times ([or possibly fewer](#)) in $O(n^3 \log n)$ total operations is faster than naively executing the Floyd-Warshall algorithm in $O(n^3)$ operations, although not by a lot.

As an exercise, try to speed up this “for-for-for” computation. It is harder to do than in the matrix multiplication case because now there is a logical dependency between the iterations, and you need to perform updates in a particular order, but it is still possible to design [a similar kernel](#) and a [block iteration order](#) that achieves a 30-50x total speedup.

Acknowledgements

The final algorithm was originally designed by Kazushige Goto, and it is the basis of GotoBLAS and OpenBLAS. The author himself describes it in more detail in “[Anatomy of High-Performance Matrix Multiplication](#)”.

The exposition style is inspired by the “[Programming Parallel Computers](#)” course by Jukka Suomela, which features a [similar case study](#) on speeding up the distance product.

Data Structures Case Studies

Binary Search

While improving the speed of user-facing applications is the end goal of performance engineering, people don't really get excited over 5-10% improvements in some databases. Yes, this is what software engineers are paid for, but these types of optimizations tend to be too intricate and system-specific to be readily generalized to other software.

Instead, the most fascinating showcases of performance engineering are multifold optimizations of textbook algorithms: the kinds that everybody knows and deemed so simple that it would never even occur to try to optimize them in the first place. These optimizations are simple and instructive and can very much be adopted elsewhere. And they are surprisingly not as rare as you'd think.

In this section, we focus on one such fundamental algorithm – *binary search* – and implement two of its variants that are, depending on the problem size, up to 4x faster than `std::lower_bound`, while being under just 15 lines of code.

The first algorithm achieves that by removing branches, and the second also optimizes the memory layout to achieve better cache system performance. This technically disqualifies it from being a drop-in replacement for `std::lower_bound` as it needs to permute the elements of the array before it can start answering queries – but I can't recall a lot of scenarios where you obtain a sorted array but can't afford to spend linear time on preprocessing.

The usual disclaimer: the CPU is a Zen 2, the RAM is a DDR4-2666, and the compiler we will be using by default is Clang 10. The performance on your machine may be different, so I highly encourage to go and test it for yourself.

Binary Search

Here is the standard way of searching for the first element not less than `x` in a sorted array `t` of `n` integers that you can find in any introductory computer science textbook:

```
int lower_bound(int x) {
    int l = 0, r = n - 1;
    while (l < r) {
        int m = (l + r) / 2;
        if (t[m] >= x)
            r = m;
        else
            l = m + 1;
    }
    return t[l];
}
```

Find the middle element of the search range, compare it to `x`, shrink the range in half. Beautiful in its simplicity.

A similar approach is employed by `std::lower_bound`, except that it needs to be more generic to support containers with non-random-access iterators and thus uses the first element and the size of the search interval instead of the two of its ends. To this end, implementations from both Clang and GCC use this metaprogramming monstrosity:

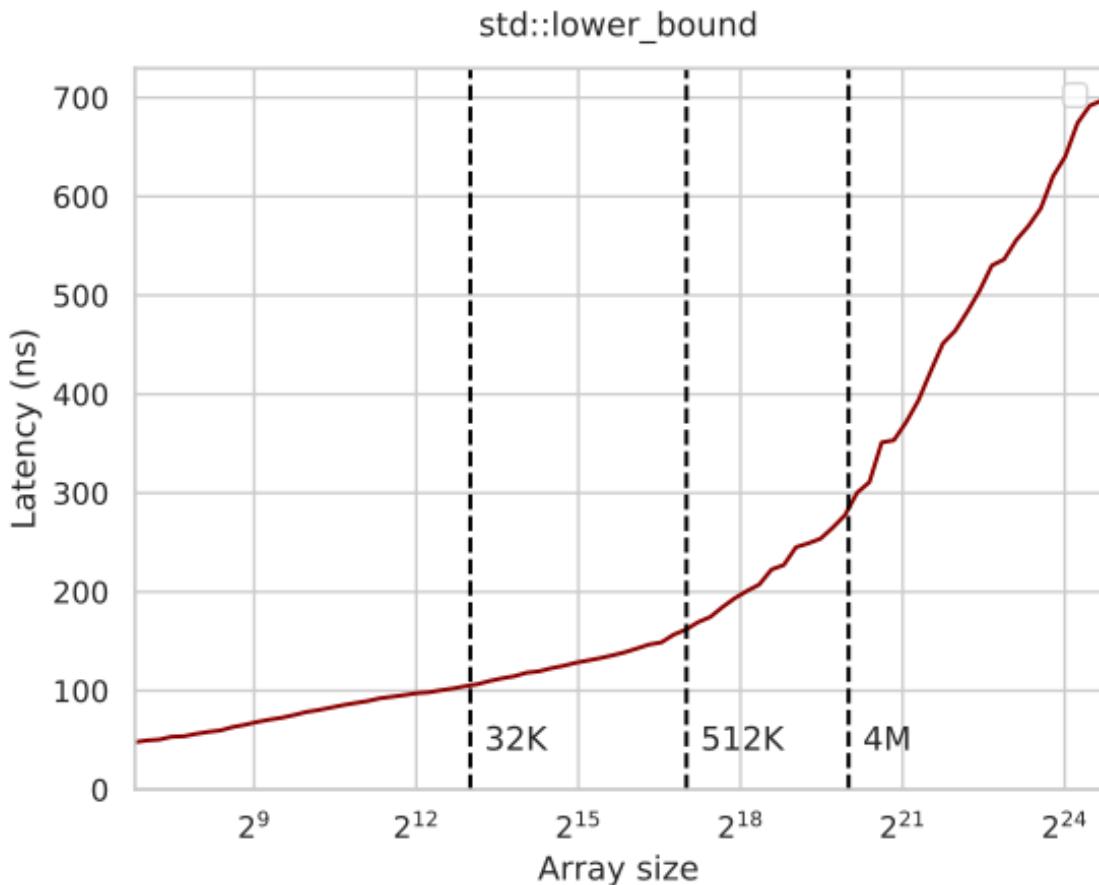
```
template <class _Compare, class _ForwardIterator, class _Tp>
_LIBCPP_CONSTEXPR_AFTER_CXX17 _ForwardIterator
__lower_bound(_ForwardIterator __first, _ForwardIterator __last, const _Tp& __value_, _Compare __comp)
{
    typedef typename iterator_traits<_ForwardIterator>::difference_type difference_type;
```

```

difference_type __len = _VSTD::distance(__first, __last);
while (__len != 0)
{
    difference_type __l2 = _VSTD::__half_positive(__len);
    _ForwardIterator __m = __first;
    _VSTD::advance(__m, __l2);
    if (__comp(*__m, __value_))
    {
        __first = ++__m;
        __len -= __l2 + 1;
    }
    else
        __len = __l2;
}
return __first;
}

```

If the compiler is successful in removing the abstractions, it compiles to roughly the same machine code and yields roughly the same average latency, which expectedly grows with the array size:



Since most people don't implement binary search by hand, we will use `std::lower_bound` from Clang as the baseline.

The Bottleneck

Before jumping to the optimized implementations, let's briefly discuss why binary search is slow in the first place.

If you run `std::lower_bound` with perf, you'll see that it spends most of its time on a conditional jump instruction:

```
|35:  mov    %rax,%rdx
0.52 |  sar    %rdx
0.33 |  lea    (%rsi,%rdx,4),%rcx
4.30 |  cmp    (%rcx),%edi
65.39 |  v jle  b0
0.07 |  sub    %rdx,%rax
9.32 |  lea    0x4(%rcx),%rsi
0.06 |  dec    %rax
1.37 |  test   %rax,%rax
1.11 | ^ jg    35
```

This pipeline stall stops the search from progressing, and it is mainly caused by two factors:

- We suffer a *control hazard* because we have a branch that is impossible to predict (queries and keys are drawn independently at random), and the processor has to halt for 10-15 cycles to flush the pipeline and fill it back on each branch mispredict.
- We suffer a *data hazard* because we have to wait for the preceding comparison to complete, which in turn waits for one of its operands to be fetched from the memory – and it may take anywhere between 0 and 300 cycles, depending on where it is located.

Now, let's try to get rid of these obstacles one by one.

Removing Branches

We can replace branching with predication. To make the task easier, we can adopt the STL approach and rewrite the loop using the first element and the size of the search interval (instead of its first and last element):

```
int lower_bound(int x) {
    int *base = t, len = n;
    while (len > 1) {
        int half = len / 2;
        if (base[half - 1] < x) {
            base += half;
            len = len - half;
        } else {
            len = half;
        }
    }
    return *base;
}
```

Note that, on each iteration, `len` is essentially just halved and then either floored or ceiled, depending on how the comparison went. This conditional update seems unnecessary; to avoid it, we can simply say that it's always ceiled:

```
int lower_bound(int x) {
    int *base = t, len = n;
    while (len > 1) {
        int half = len / 2;
        if (base[half - 1] < x)
```

```

        base += half;
        len -= half; // = ceil(len / 2)
    }
    return *base;
}

```

This way, we only need to update the first element of the search interval with a conditional move and halve its size on each iteration:

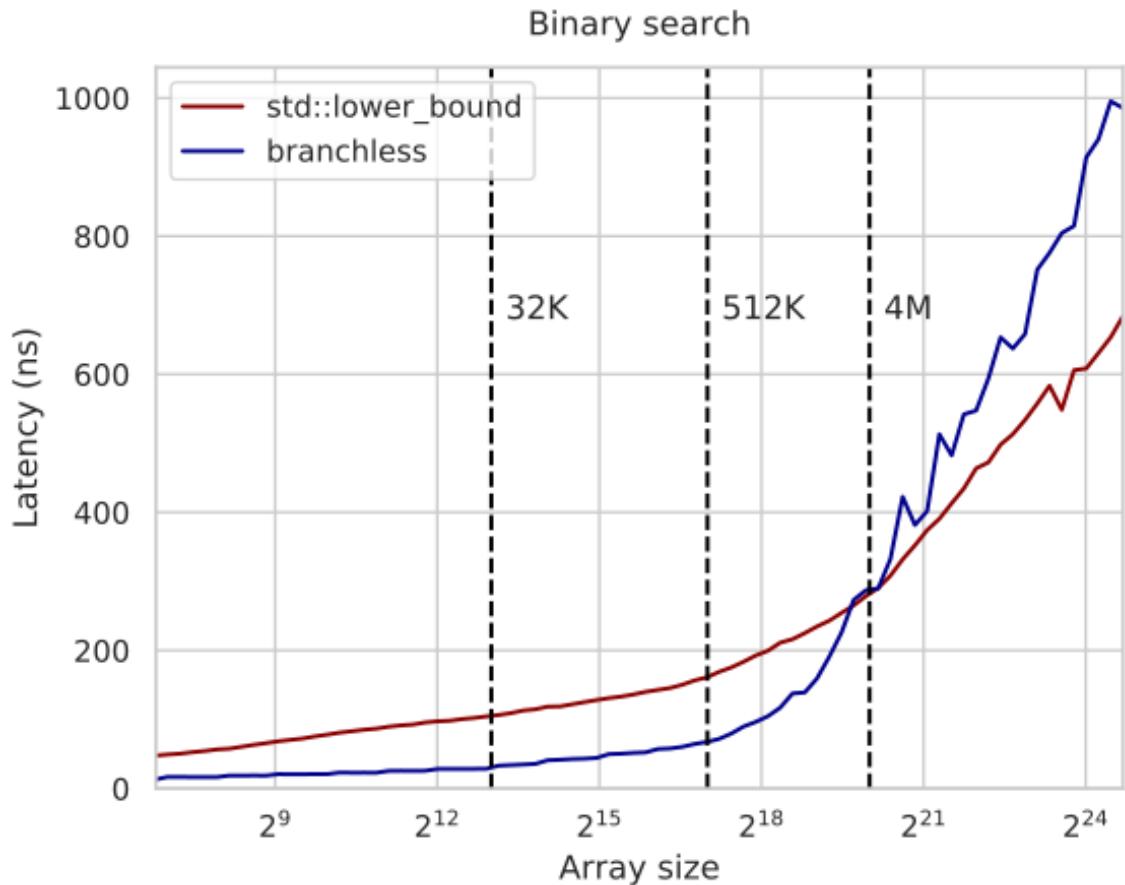
```

int lower_bound(int x) {
    int *base = t, len = n;
    while (len > 1) {
        int half = len / 2;
        base += (base[half - 1] < x) * half; // will be replaced with a "cmov"
        len -= half;
    }
    return *base;
}

```

Note that this loop is not always equivalent to the standard binary search. Since it always rounds *up* the size of the search interval, it accesses slightly different elements and may perform one comparison more than needed. Apart from simplifying computations on each iteration, it also makes the number of iterations constant if the array size is constant, removing branch mispredictions completely.

As typical for predication, this trick is very fragile to compiler optimizations – depending on the compiler and how the function is invoked, it may still leave a branch or generate suboptimal code. It works fine on Clang 10, yielding a 2.5-3x improvement on small arrays:



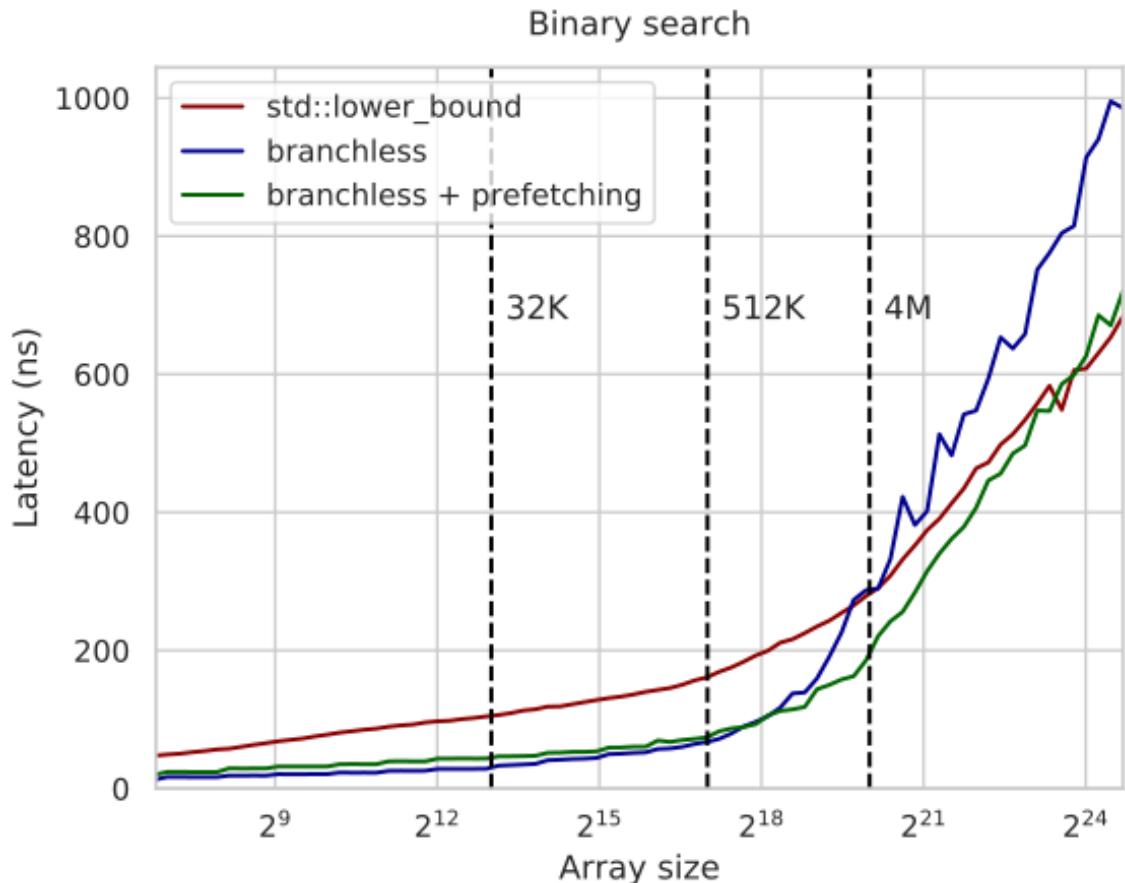
One interesting detail is that it performs worse on large arrays. It seems weird: the total delay is dominated by the RAM latency, and since it does roughly the same memory accesses as the standard binary search, it should be roughly the same or even slightly better.

The real question you need to ask is not why the branchless implementation is worse but why the branchy version is better. It happens because when you have branching, the CPU can speculate on one of the branches and start fetching either the left or the right key before it can even confirm that it is the right one – which effectively acts as implicit prefetching.

For the branchless implementation, this doesn't happen, as `cmove` is treated as every other instruction, and the branch predictor doesn't try to peek into its operands to predict the future. To compensate for this, we can prefetch the data in software by explicitly requesting the left and right child key:

```
int lower_bound(int x) {
    int *base = t, len = n;
    while (len > 1) {
        int half = len / 2;
        len -= half;
        __builtin_prefetch(&base[len / 2 - 1]);
        __builtin_prefetch(&base[half + len / 2 - 1]);
        base += (base[half - 1] < x) * half;
    }
    return *base;
}
```

With prefetching, the performance on large arrays becomes roughly the same:

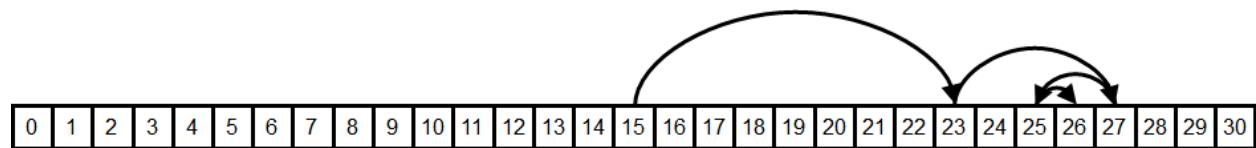


The graph still grows faster as the branchy version also prefetches “grandchildren,” “great-grandchildren,” and so on – although the usefulness of each new speculative read diminishes exponentially as the prediction is less and less likely to be correct.

In the branchless version, we could also fetch ahead by more than one layer, but the number of fetches we’d need also grows exponentially. Instead, we will try a different approach to optimize memory operations.

Optimizing the Layout

The memory requests we perform during binary search form a very specific access pattern:



How likely is it that the elements on each request are cached? How good is their data locality?

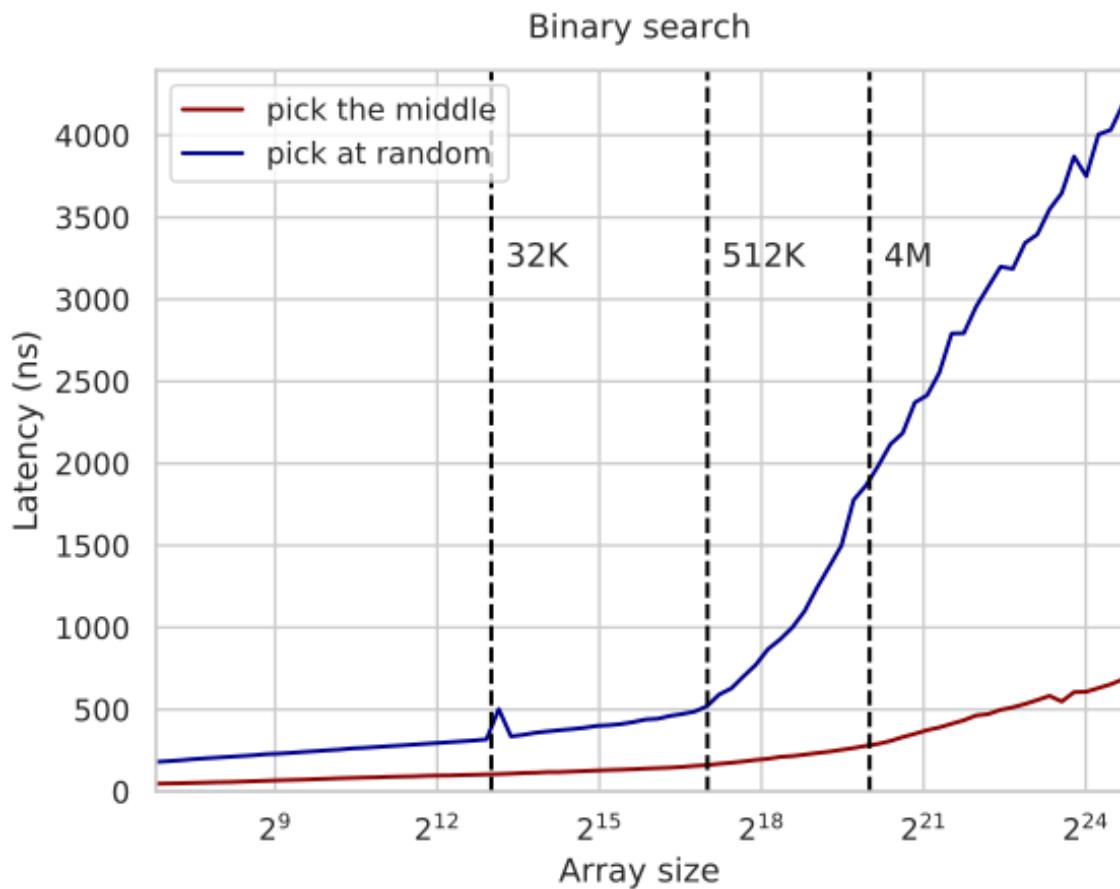
- *Spatial locality* seems to be okay for the last 3 to 4 requests that are likely to be on the same cache line – but all the previous requests require huge memory jumps.
- *Temporal locality* seems to be okay for the first dozen or so requests – there aren’t that many different comparison sequences of this length, so we will be comparing against the same middle elements over and over, which are likely to be cached.

To illustrate how important the second type of cache sharing is, let’s try to pick the element we will compare

to on each iteration randomly among the elements of the search interval, instead of the middle one:

```
int lower_bound(int x) {
    int l = 0, r = n - 1;
    while (l < r) {
        int m = l + rand() % (r - 1);
        if (t[m] >= x)
            r = m;
        else
            l = m + 1;
    }
    return t[l];
}
```

Theoretically, this randomized binary search is expected to do 30-40% more comparisons than the normal one, but on a real computer, the running time goes $\sim 6x$ on large arrays:



This isn't just caused by the `rand()` call being slow. You can clearly see the point on the L2-L3 boundary where memory latency outweighs the random number generation and modulo. The performance degrades because all of the fetched elements are unlikely to be cached and not just some small suffix of them.

Another potential negative effect is that of cache associativity. If the array size is a multiple of a large power of two, then the indices of these “hot” elements will also be divisible by some large powers of two and map to the same cache line, kicking each other out. For example, binary searching over arrays of size 2^{20} takes about ~ 360 ns per query while searching over arrays of size $(2^{20} + 123)$ takes ~ 300 ns – a 20% difference. There are ways to fix this problem, but to not get distracted from more pressing matters, we are just going to ignore

it: all array sizes we use are in the form of $\lfloor 1.17^k \rfloor$ for integer k so that any cache side effects are unlikely.

The real problem with our memory layout is that it doesn't make the most efficient use of temporal locality because it groups hot and cold elements together. For example, we likely store the element $\lfloor n/2 \rfloor$, which we request the first thing on each query, in the same cache line with $\lfloor n/2 \rfloor + 1$, which we almost never request.

Here is the heatmap visualizing the expected frequency of comparisons for a 31-element array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

So, ideally, we'd want a memory layout where hot elements are grouped with hot elements, and cold elements are grouped with cold elements. And we can achieve this if we permute the array in a more cache-friendly way by renumbering them. The numeration we will use is actually half a millennium old, and chances are, you already know it.

Eytzinger Layout

Michael Eytzinger is a 16th-century Austrian nobleman known for his work on genealogy, particularly for a system for numbering ancestors called *ahnentafel* (German for “ancestor table”).

Ancestry mattered a lot back then, but writing down that data was expensive. *Ahnentafel* allows displaying a person's genealogy compactly, without wasting extra space by drawing diagrams.

It lists a person's direct ancestors in a fixed sequence of ascent. First, the person themselves is listed as number 1, and then, recursively, for each person numbered k , their father is listed as $2k$ and their mother as $(2k + 1)$.

Here is the example for Paul I, the great-grandson of Peter the Great:

1. Paul I
2. Peter III (Paul's father)
3. Catherine II (Paul's mother)
4. Charles Frederick (Peter's father, Paul's paternal grandfather)
5. Anna Petrovna (Peter's mother, Paul's paternal grandmother)
6. Christian August (Catherine's father, Paul's maternal grandfather)
7. Johanna Elisabeth (Catherine's mother, Paul's maternal grandmother)

Apart from being compact, it has some nice properties, like that all even-numbered persons are male and all odd-numbered (possibly except for 1) are female. One can also find the number of a particular ancestor only knowing the genders of their descendants. For example, Peter the Great's bloodline is Paul I \rightarrow Peter III \rightarrow Anna Petrovna \rightarrow Peter the Great, so his number should be $((1 \times 2) \times 2 + 1) \times 2 = 10$.

In computer science, this enumeration has been widely used for implicit (pointer-free) implementations of heaps, segment trees, and other binary tree structures – where instead of names, it stores underlying array items.

Here is how this layout looks when applied to binary search:

When searching in this layout, we just need to start from the first element of the array, and then on each iteration jump to either $2k$ or $(2k + 1)$, depending on how the comparison went:

<img alt="Diagram showing a binary search path on an array of 31 elements. The array is: 15, 7, 23, 3, 11, 19, 27, 1, 5, 9, 13, 17, 21, 25, 29, 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30. A series of curved arrows starts at index 0 and points to indices 1, 3, 7, 15, 23, 47, 94, 188, 376, 752, 1504, 3008, 6016, 12032, 24064, 48128, 96256, 192512, 385024, 770048, 1540096, 3080192, 6160384, 12320768, 24641536, 49283072, 98566144, 197132288, 394264576, 788529152, 1577058304, 3154116608, 6308233216, 12616466432, 25232932864, 50465865728, 100931731456, 201863462912, 403726925824, 807453851648, 1614907703296, 3229815406592, 6459630813184, 12919261626368, 25838523252736, 51677046505472, 103354093010944, 206708186021888, 413416372043776, 826832744087552, 1653665488175104, 3307330976350208, 6614661952700416, 13229323905400832, 26458647810801664, 52917295621603328, 105834591243206656, 211669182486413312, 423338364972826624, 846676729945653248, 1693353459891306496, 3386706919782612992, 6773413839565225984, 13546827679130451968, 27093655358260903936, 54187310716521807872, 108374621433043615744, 216749242866087231488, 433498485732174462976, 866996971464348925952, 1733993942928697851904, 3467987885857395703808, 6935975771714791407616, 13871951543429582815232, 27743903086859165630464, 55487806173718331260928, 110975612347436662521856, 221951224694873325043712, 443902449389746650087424, 887804898779493300174848, 1775609797558986600349696, 3551219595117973200699392, 7102439190235946401398784, 1420487838047189280279768, 2840975676094378560559536, 5681951352188757121119072, 11363902704375514242238144, 22727805408751028484476288, 45455610817502056968952576, 90911221635004113937905152, 18182244327000822787580288, 36364488654001645575160576, 72728977308003291150321152, 145457954616006582300642304, 290915909232001364601284608, 581831818464002729202569216, 1163663636928005558405138432, 2327327273856011116810276864, 4654654547712022233620553728, 9309309095424044467241107456, 18618618190848088934822014912, 37237236381696177869644029824, 74474472763392355739288059648, 148948945486784711478576119296, 297897890973569422957152238592, 595795781947138845914304477184, 1191591563894277691828568954368, 2383183127788555383657137908736, 4766366255577110767314275817472, 9532732511154221534628551634944, 19065465022308443069257103269888, 38130930044616886138514206539776, 76261860089233772277028413079552, 15252372017846754455405682615904, 30504744035693508910811365231808, 61009488071387017821622730463616, 122018976142774035643244608927232, 244037952285548071286489217854464, 488075854571096142572978435708928, 976151709142192285145956871417856, 1952303418284384570291917742835712, 3904606836568769140583835485671424, 7809213673137538281167670971342848, 15618427346275676562335419942685696, 31236854692551353124670839885371392, 62473709385102706249341679770742784, 124947418770205412498683595541495568, 249894837540410824997367185082991136, 499789675080821649994734370165982272, 999579350161643299989468740331964544, 199915870032328659997937548066392988, 399831740064657319995875096132785976, 799663480129314639991750192265571952, 159932696025862927998350038453114384, 319865392051725855996700076906228768, 639730784103451711993400153812457536, 127946156820690342398680030762495072, 25589231364138068479736006152490144, 51178462728276136959472001231280288, 102356925456532273918944002462560576, 204713850913064547837888004925121152, 409427701826129095675776009850242304, 818855403652258191351552019700484608, 163771080730451638270304039400969216, 327542161460903276540608078800938432, 655084322921806553081216017600976864, 1310168645843613106162432035200953728, 2620337291687226212324864070400907456, 5240674583374452424649728140800814912, 1048134916674885484929456828160082984, 2096269833349770969858913656320085968, 4192539666699541939717827312640081936, 8385079333398883879435654625280083872, 1677015866679776775887310925560087744, 3354031733359553551774621851120081548, 6708063466719107103549243702240083096, 1341612693343821420703846740448008692, 2683225386687642841407693480896008384, 5366450773375285682815386961792008768, 10732901546750571365630738923584008136, 21465803093501142731261477847168008272, 42931606187002285462528955694336008544, 85863212374004570925057911388672008088, 171726424748009141850158822777344008176, 343452849496018283700317645554688008352, 686905698992036567400635291109376008704, 1373811397984073134801270582218752008408, 2747622795968146269602541164437504008816, 549524559193629253920508232887500801632, 109904911838725850784101646575500803264, 219809823677451701568203293151000806528, 439619647354903403136406586302000813056, 879239294709806806272813172604000826112, 1758478589419613612545626345208000852224, 3516957178839227225091252680416000874448, 7033914357678454450182505360832000898896, 1406782871535690890365005721664000917792, 2813565743071381780730005443232000935584, 562713148614276356146000522664000971168, 112542629322855271229000511332000942336, 225085258645710542458000505664000924672, 450170517291421084916000491328000912344, 900341034582842169832000480656000904688, 1800682069165684339664000460312000898376, 3601364138331368679328000440624000891752, 7202728276662737358656000420348000883504, 14405456553325474717312000400696000871752, 28810913106650949434624000380392000860876, 57621826213301898869248000360784000849752, 115243652426603797784976000340368000839504, 230487304853207595569952000320736000828752, 460974609706415191139904000300712000817504, 921949219412830382279808000280356000806252, 184389843882566076459616000260178000795125, 368779687765132152919232000240356000784250, 737559375530264305838464000220712000773125, 1475118751060528611676928000201424000762050, 2950237502121057223353856000182848000751025, 5900475004242114446707712000165696000740050, 1180095008488422889341544000151392000729025, 2360190016976845778683088000140784000718050, 47203800339536915573661760001301568000707025, 94407600679073831147323520001203136000696050, 188815200139147662294647040001106272000685025, 377630400278295324589294080001012544000674050, 755260800556590649178588160000915088000663025, 1510521601113181282357776320000812176000652050, 3021043202226362564715552640000714352000641025, 6042086404452725129431105280000618704000630050, 12084172808905450258862210560000517408000619025, 24168345617810900517724421120000414816000608050, 48336691235621801035448842240000313632000597025, 96673382471243602070897684480000216764000586050, 1933467649424872041417537689600001133824000575025, 386693529884974408283507537920000056764800056250, 773387059769948816567015075840000028382400053125, 154677411953989733133023015168000001419120005125, 3093548239079794662660460303360000007095600048125, 61870964781595893253209206067200000035478400046050, 12374192956319178656604603033600000017739200043025, 247483859126383573132092060672000000088696000415125, 494967718252767146264084030336000000044348000407550, 989935436505534292528168030336000000022174400040375, 1979870873011068585056336030336000000011087200039750, 3959741746022137170112672030336000000005543600039375, 7919483492044274340225344030336000000002771800038650, 1583896698408854668045068030336000000001385900037325, 31677933968177093360901360303360000000006929500036650, 6335586793635418672180272030336000000003464800035325, 1267117358727083734436054403033600000001732400034650, 2534234717454167468872108803033600000000866200033325, 5068469434908334937744217603033600000000433100032650, 10136938689166684875484352030336000000002165500031325, 20273877378333379750968704030336000000001082750030650, 40547754756666759501937408030336000000005413750030325, 810955095133335190038752160303360000000027068750030150, 16219101902666759501937521603033600000000135343750030075, 3243820380533351900387521603033600000000067671500029950, 64876407610667038007750432030336000000000338357500029925, 129752815221335190038752160303360000000001691787500029850, 259505630442670380077504320303360000000000845895000029750, 5190112608853407600155086403033600000000004229475000029375, 103802252177068152003101728030336000000000021147375000029150, 207604504354136304006203456030336000000000010573675000028925, 415209008708272608003101728030336000000000005286835000028450, 8304180174165452160062034560303360000000000026434175000028225, 16608360348330904320124069203033600000000000132170875000028125, 332167206966618086402481384030336000000000000660854375000028050, 66433441393323617280496276803033600000000000033042715000027975, 1328668827866472345609925536030336000000000001652135375000027925, 2657337655733944691219851072030336000000000000826067675000027850, 53146753114678893824397021440303360000000000004130338375000027725, 10629350622935778764879404288030336000000000000206516915000027650, 212587012458715575297588085760303360000000000001032584575000027525, 4251740249174311505951761715203033600000000000005162922875000027450, 85034804983486230119035234304030336000000000000025814614375000027325, 1700696099669724602380706760803033600000000000001290730715000027250, 34013921993394492047614135216030336000000000000006453653575000027125, 68027843986788984095228270432030336000000000000003226826715000027050, 1360556879735779681904565408640303360000000000000016134134375000026925, 272111375947155936380913081728030336000000000000000806706715000026850, 5442227518943118727618261634560303360000000000000004033534375000026725, 1088445503788623545323652326912030336000000000000000201676715000026650, 21768910075772470906473046538240303360000000000000001008383437500026525, 43537820151544941812946093076480303360000000000000000504191715000026450, 870756403030898836258921861529603033600000000000000002520958375000026325, 1741512806061797672

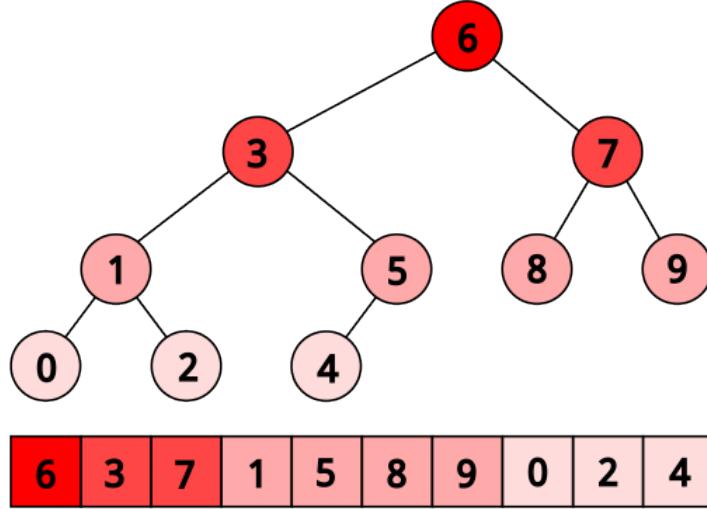


Figure 1: Note that the tree is slightly imbalanced (because of the last layer is continuous)

Another way to look at it is that we write every even-indexed element to the end of the new array, then write every even-indexed element of the remaining ones right before them, and so on, until we place the root as the first element.

Construction

To construct the Eytzinger array, we could do this even-odd filtering $O(\log n)$ times – and, perhaps, this is the fastest approach – but for brevity, we will instead build it by traversing the original search tree:

```
int a[n], t[n + 1]; // the original sorted array and the eytzinger array we build
// ^ we need one element more because of one-based indexing

void eytzinger(int k = 1) {
    static int i = 0; // <- careful running it on multiple arrays
    if (k <= n) {
        eytzinger(2 * k);
        t[k] = a[i++];
        eytzinger(2 * k + 1);
    }
}
```

This function takes the current node number k , recursively writes out all elements to the left of the middle of the search interval, writes out the current element we'd compare against, and then recursively writes out all the elements on the right. It seems a bit complicated, but to convince yourself that it works, you only need three observations:

- It writes exactly n elements as we enter the body of `if` for each k from 1 to n just once.
- It writes out sequential elements from the original array as it increments the i pointer each time.
- By the time we write the element at node k , we will have already written all the elements to its left (exactly i).

Despite being recursive, it is actually quite fast as all the memory reads are sequential, and the memory writes are only in $O(\log n)$ different memory blocks at a time. Maintaining the permutation is both logically and computationally harder to maintain though: adding an element to a sorted array only requires shifting a suffix of its elements one position to the right, while Eytzinger array practically needs to be rebuilt from

scratch.

Note that this traversal and the resulting permutation are not exactly equivalent to the “tree” of vanilla binary search: for example, the left child subtree may be larger than the right child subtree – up to twice as large – but it doesn’t matter much since both approaches result in the same $\lceil \log_2 n \rceil$ tree depth.

Also note that the Eytzinger array is one-indexed – this will be important for performance later. You can put in the zeroth element the value that you want to be returned in the case when the lower bound doesn’t exist (similar to `a.end()` for `std::lower_bound`).

Search Implementation

We can now descend this array using only indices: we just start with $k = 1$ and execute $k := 2k$ if we need to go left and $k := 2k + 1$ if we need to go right. We don’t even need to store and recalculate the search boundaries anymore. This simplicity also lets us avoid branching:

```
int k = 1;
while (k <= n)
    k = 2 * k + (t[k] < x);
```

The only problem arises when we need to restore the index of the resulting element, as k does not directly point to it. Consider this example (its corresponding tree is listed above):

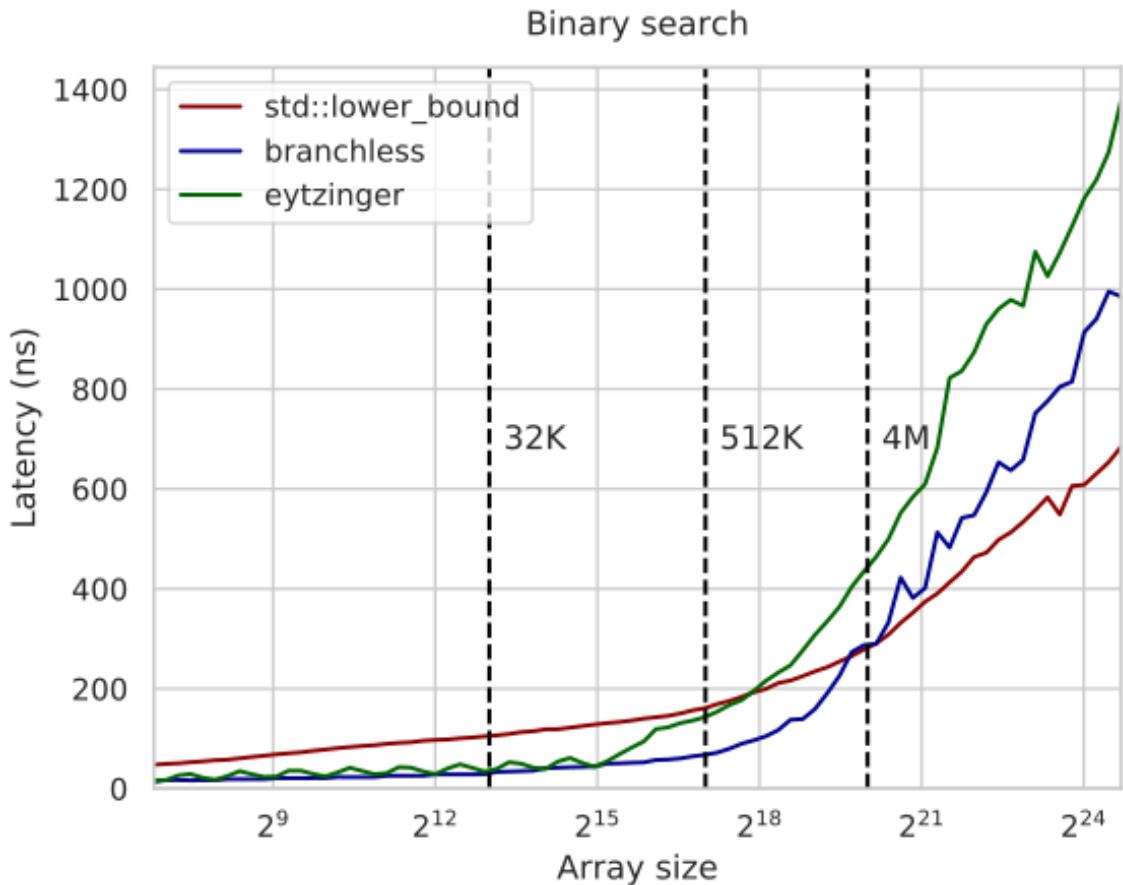
Here we query the array of $[0, \dots, 9]$ for the lower bound of $x = 3$. We compare it against 6, 3, 1, and 2, go left-left-right-right, and end up with $k = 19$, which isn’t even a valid array index.

The trick is to notice that, unless the answer is the last element of the array, we compare x against it at some point, and after we’ve learned that it is not less than x , we go left exactly once and then keep going right until we reach a leaf (because we will only be comparing x against lesser elements). Therefore, to restore the answer, we just need to “cancel” some number of right turns and then one more.

This can be done in an elegant way by observing that the right turns are recorded in the binary representation of k as 1-bits, and so we just need to find the number of trailing 1s in the binary representation and right-shift k by exactly that number of bits plus one. To do this, we can invert the number ($\sim k$) and call the “find first set” instruction:

```
int lower_bound(int x) {
    int k = 1;
    while (k <= n)
        k = 2 * k + (t[k] < x);
    k >>= __builtin_ffs(~k);
    return t[k];
}
```

We run it, and... well, it doesn’t look *that* good:



The latency on smaller arrays is on par with the branchless binary search implementation – which isn’t surprising as it is just two lines of code – but it starts taking off much sooner. The reason is that the Eytzinger binary search doesn’t get the advantage of spatial locality: the last 3-4 elements we compare against are not in the same cache line anymore, and we have to fetch them separately.

If you think about it deeper, you might object that the improved temporal locality should compensate for that. Before, we were using only about $\frac{1}{16}$ -th of the cache line to store one hot element, and now we are using all of it, so the effective cache size is larger by a factor of 16, which lets us cover $\log_2 16 = 4$ more first requests.

But if you think about it more, you understand that this isn’t enough compensation. Caching the other 15 elements wasn’t completely useless, and also, the hardware prefetcher could fetch the neighboring cache lines of our requests. If this was one of our last requests, the rest of what we will be reading will probably be cached elements. So actually, the last 6-7 accesses are likely to be cached, not 3-4.

It seems like we did an overall stupid thing switching to this layout, but there is a way to make it worthwhile.

Prefetching

To hide the memory latency, we can use software prefetching similar to how we did for branchless binary search. But instead of issuing two separate prefetch instructions for the left and right child nodes, we can notice that they are neighbors in the Eytzinger array: one has index $2k$ and the other $(2k + 1)$, so they are likely in the same cache line, and we can use just one instruction.

This observation extends to the grand-children of node k – they are also stored sequentially:

$$2 * 2 * k = 4 * k$$

```

2 * 2 * k + 1      = 4 * k + 1
2 * (2 * k + 1)    = 4 * k + 2
2 * (2 * k + 1) + 1 = 4 * k + 3

```

Their cache line can also be fetched with one instruction. Interesting... what if we continue this, and instead of fetching direct children, we fetch ahead as many descendants as we can cramp into one cache line? That would be $\frac{64}{4} = 16$ elements, our great-great-grandchildren with indices from $16k$ to $(16k + 15)$.

Now, if we prefetch just one of these 16 elements, we will probably only get some but not all of them, as they may cross a cache line boundary. We can prefetch the first *and* the last element, but to get away with just one memory request, we need to notice that the index of the first element, $16k$, is divisible by 16, so its memory address will be the base address of the array plus something divisible by $16 \cdot 4 = 64$, the cache line size. If the array were to begin on a cache line, then these 16 great-great-grandchildren elements will be guaranteed to be on a single cache line, which is just what we needed.

Therefore, we only need to align the array:

```
t = (int*) std::aligned_alloc(64, 4 * (n + 1));
```

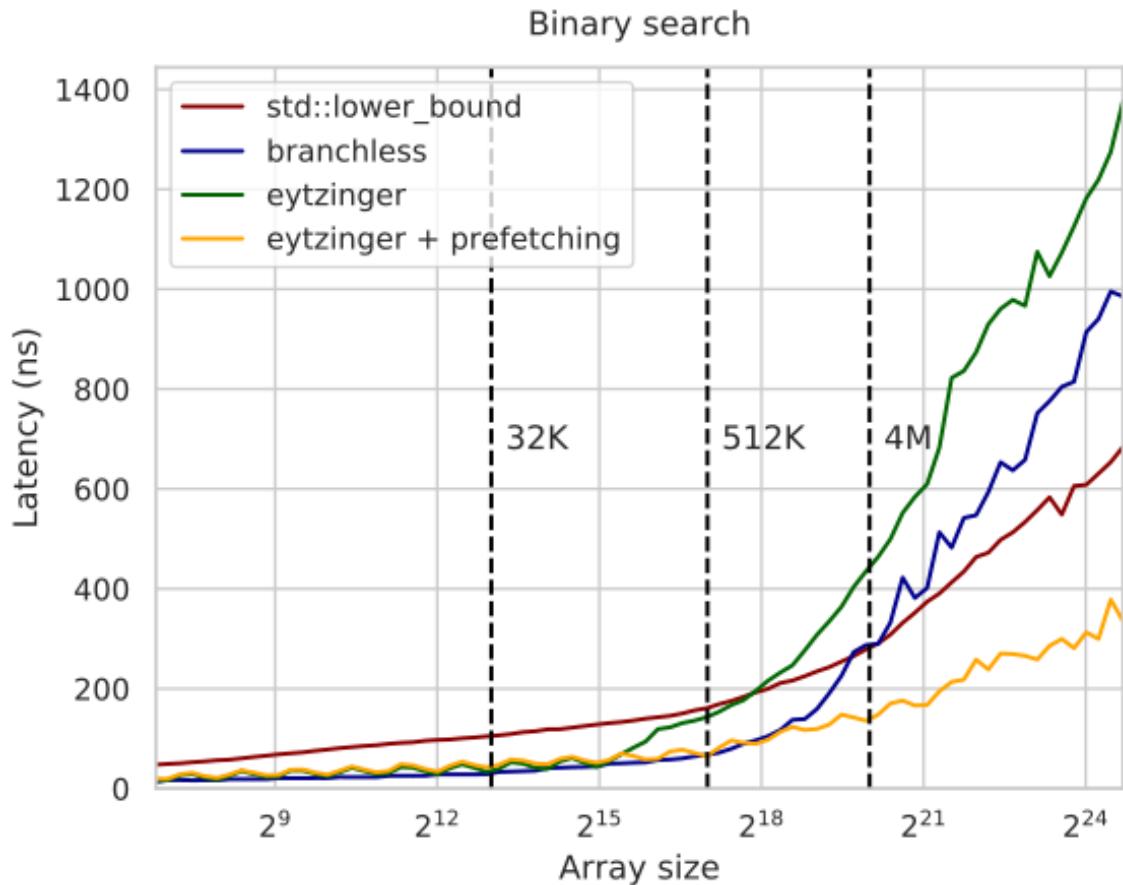
And then prefetch the element indexed $16k$ on each iteration:

```

int lower_bound(int x) {
    int k = 1;
    while (k <= n) {
        __builtin_prefetch(t + k * 16);
        k = 2 * k + (t[k] < x);
    }
    k >>= __builtin_ffs(~k);
    return t[k];
}

```

The performance on large arrays improves 3-4x from the previous version and ~2x compared to `std::lower_bound`. Not bad for just two more lines of code:



Essentially, what we do here is hide the latency by prefetching four steps ahead and overlapping memory requests. Theoretically, if the compute didn't matter, we would expect a ~4x speedup, but in reality, we get a somewhat more moderate speedup.

We can also try to prefetch further than that four steps ahead, and we don't even have to use more than one prefetch instruction for that: we can try to request only the first cache line and rely on the hardware to prefetch its neighbors. This trick may or may not improve actual performance – depends on the hardware:

```
__builtin_prefetch(t + k * 32);
```

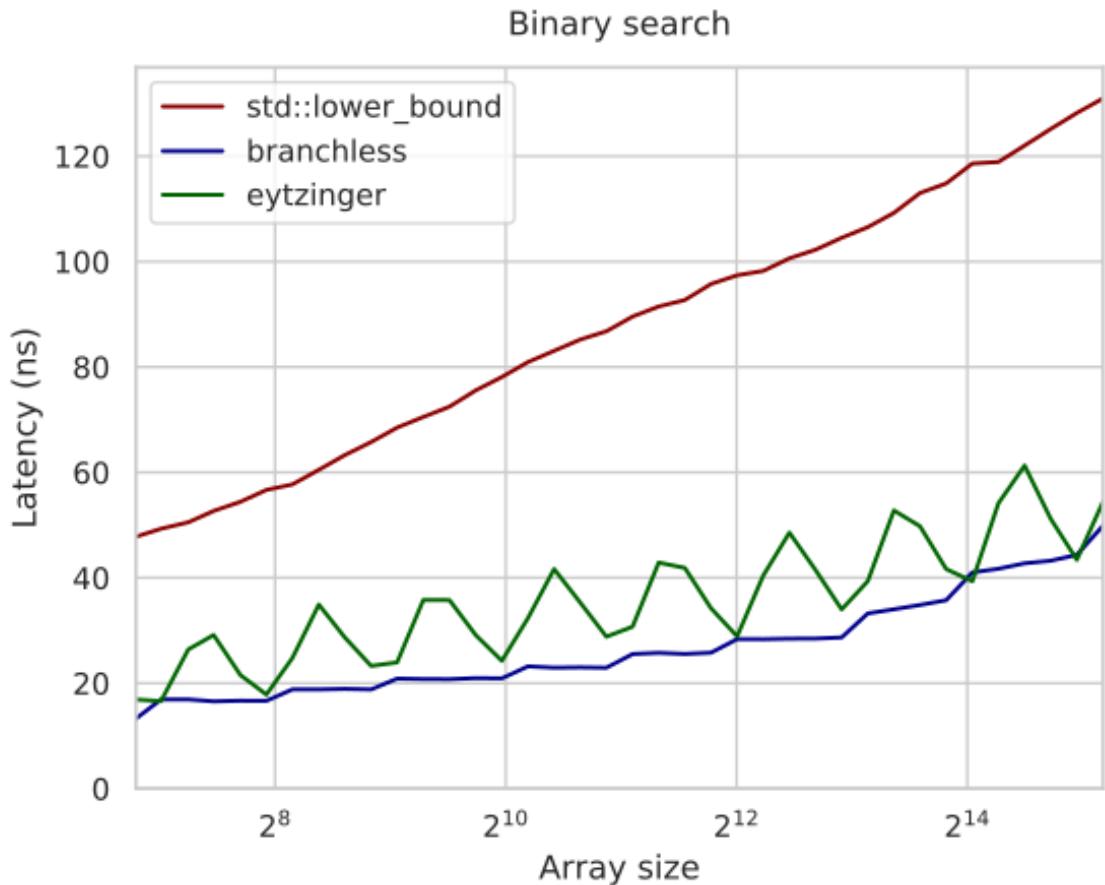
Also, note that the last few prefetch requests are actually not needed, and in fact, they may even be outside the memory region allocated for the program. On most modern CPUs, invalid prefetch instructions get converted into no-ops, so it isn't a problem, but on some platforms, this may cause a slowdown, so it may make sense, for example, to split off the last ~4 iterations from the loop to try to remove them.

This prefetching technique allows us to read up to four elements ahead, but it doesn't really come for free – we are effectively trading off excess memory bandwidth for reduced latency. If you run more than one instance at a time on separate hardware threads or just any other memory-intensive computation in the background, it will significantly affect the benchmark performance.

But we can do better. Instead of fetching four cache lines at a time, we could fetch four times *fewer* cache lines. And in the next section, we will explore the approach.

Removing the Last Branch

Just one finishing touch: did you notice the bumpiness of the Eytzinger search? This isn't random noise – let's zoom in:



The latency is ~ 10 ns higher for the array sizes in the form of $1.5 \cdot 2^k$. These are mispredicted branches from the loop itself – the last branch, to be exact. When the array size is far from a power of two, it is hard to predict whether the loop will make $\lfloor \log_2 n \rfloor$ or $\lfloor \log_2 n \rfloor + 1$ iterations, so we have a 50% chance to suffer exactly one branch mispredict.

One way to address it is to pad the array with infinities to the closest power of two, but this wastes memory. Instead, we get rid of that last branch by always executing a constant minimum number of iterations and then using predication to optionally make the last comparison against some dummy element – that is guaranteed to be less than x so that its comparison will be canceled:

```
t[0] = -1; // an element that is less than x
iters = std::__lg(n + 1);

int lower_bound(int x) {
    int k = 1;

    for (int i = 0; i < iters; i++)
        k = 2 * k + (t[k] < x);

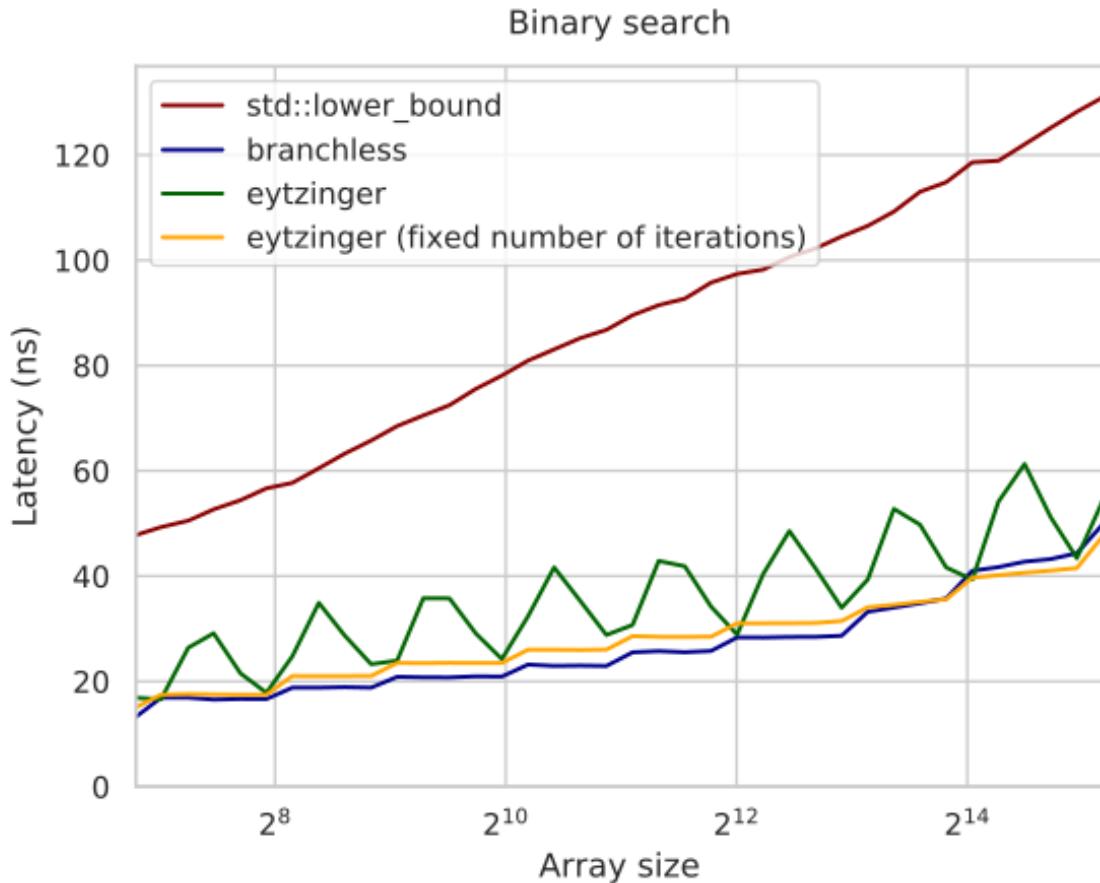
    int *loc = (k <= n ? t + k : t);
    k = 2 * k + (*loc < x);

    k >>= __builtin_ffs(~k);

    return t[k];
```

}

The graph is now smooth, and on small arrays, it is just a couple of cycles slower than the branchless binary search:



Interestingly, now GCC fails to replace the branch with `cmove`, but Clang doesn't. 1-1.

Appendix: Random Binary Search

By the way, finding the exact expected number of comparisons for random binary search is quite an interesting math problem in and of itself. Try solving it yourself first!

The way to compute it *algorithmically* is through dynamic programming. If we denote f_n as the expected number of comparisons to find a random lower bound on a search interval of size n , it can be calculated from the previous f_n by considering all the $(n - 1)$ possible splits:

$$f_n = \sum_{l=1}^{n-1} \frac{1}{n-1} \cdot \left(f_l \cdot \frac{l}{n} + f_{n-l} \cdot \frac{n-l}{n} \right) + 1$$

Directly applying this formula gives us an $O(n^2)$ algorithm, but we can optimize it by rearranging the sum like this:

$$\begin{aligned} f_n &= \sum_{i=1}^{n-1} \frac{f_i \cdot i + f_{n-i} \cdot (n-i)}{n \cdot (n-1)} + 1 \\ &= \frac{2}{n \cdot (n-1)} \cdot \sum_{i=1}^{n-1} f_i \cdot i + 1 \end{aligned}$$

To update f_n , we only need to calculate the sum of $f_i \cdot i$ for all $i < n$. To do that, let's introduce two new variables:

$$g_n = f_n \cdot n, \quad s_n = \sum_{i=1}^n g_i$$

Now they can be sequentially calculated as:

$$\begin{aligned} g_n &= f_n \cdot n = \frac{2}{n-1} \cdot \sum_{i=1}^{n-1} g_i + n = \frac{2}{n-1} \cdot s_{n-1} + n \\ s_n &= s_{n-1} + g_n \end{aligned}$$

This way we get an $O(n)$ algorithm, but we can do even better. Let's substitute g_n in the update formula for s_n :

$$\begin{aligned} s_n &= s_{n-1} + \frac{2}{n-1} \cdot s_{n-1} + n \\ &= \left(1 + \frac{2}{n-1}\right) \cdot s_{n-1} + n \\ &= \frac{n+1}{n-1} \cdot s_{n-1} + n \end{aligned}$$

The next trick is more complicated. We define r_n like this:

$$\begin{aligned} r_n &= \frac{s_n}{n} \\ &= \frac{1}{n} \cdot \left(\frac{n+1}{n-1} \cdot s_{n-1} + n\right) \\ &= \frac{n+1}{n} \cdot \frac{s_{n-1}}{n-1} + 1 \\ &= \left(1 + \frac{1}{n}\right) \cdot r_{n-1} + 1 \end{aligned}$$

We can substitute it into the formula we got for g_n before:

$$g_n = \frac{2}{n-1} \cdot s_{n-1} + n = 2 \cdot r_{n-1} + n$$

Recalling that $g_n = f_n \cdot n$, we can express r_{n-1} using f_n :

$$f_n \cdot n = 2 \cdot r_{n-1} + n \implies r_{n-1} = \frac{(f_n - 1) \cdot n}{2}$$

Final step. We've just expressed r_n through r_{n-1} and r_{n-1} through f_n . This lets us express f_{n+1} through f_n :

$$\begin{aligned}
r_n &= \left(1 + \frac{1}{n}\right) \cdot r_{n-1} + 1 \\
\Rightarrow \frac{(f_{n+1} - 1) \cdot (n + 1)}{2} &= \left(1 + \frac{1}{n}\right) \cdot \frac{(f_n - 1) \cdot n}{2} + 1 \\
&= \frac{n + 1}{2} \cdot (f_n - 1) + 1 \\
\Rightarrow (f_{n+1} - 1) &= (f_n - 1) + \frac{2}{n + 1} \\
\Rightarrow f_{n+1} &= f_n + \frac{2}{n + 1} \\
\Rightarrow f_n &= f_{n-1} + \frac{2}{n} \\
\Rightarrow f_n &= \sum_{k=2}^n \frac{2}{k}
\end{aligned}$$

The last expression is double the harmonic series, which is well known to approximate $\ln n$ as $n \rightarrow \infty$. Therefore, the random binary search will perform $\frac{2 \ln n}{\log_2 n} = 2 \ln 2 \approx 1.386$ more comparisons compared to the normal one.

Acknowledgements

The article is loosely based on “Array Layouts for Comparison-Based Searching” by Paul-Virak Khuong and Pat Morin. It is 46 pages long and discusses these and many other (less successful) approaches in more detail. I highly recommend also checking it out – this is one of my favorite performance engineering papers.

Thanks to Marshall Lochbaum for providing the proof for the random binary search. No way I could do it myself.

I also stole these lovely layout visualizations from some blog a long time ago, but I don’t remember the name of the blog and what license they had, and inverse image search doesn’t find them anymore. If you don’t sue me, thank you, whoever you are!

Static B-Trees

This section is a follow-up to the previous one, where we optimized binary search by the means of removing branching and improving the memory layout. Here, we will also be searching in sorted arrays, but this time we are not limited to fetching and comparing only one element at a time.

In this section, we generalize the techniques we developed for binary search to *static B-trees* and accelerate them further using SIMD instructions. In particular, we develop two new implicit data structures:

- The first is based on the memory layout of a B-tree, and, depending on the array size, it is up to 8x faster than `std::lower_bound` while using the same space as the array and only requiring a permutation of its elements.
- The second is based on the memory layout of a B+ tree, and it is up to 15x faster than `std::lower_bound` while using just 6-7% more memory – or 6-7% of the memory if we can keep the original sorted array.

To distinguish them from B-trees – the structures with pointers, hundreds to thousands of keys per node, and empty spaces in them – we will use the names *S-tree* and *S+ tree* respectively to refer to these particular memory layouts¹.

To the best of my knowledge, this is a significant improvement over the existing approaches. As before, we are using Clang 10 targeting a Zen 2 CPU, but the performance improvements should approximately

¹Similar to B-trees, “the more you think about what the S in S-trees means, the better you understand S-trees.”

transfer to most other platforms, including Arm-based chips. Use this single-source benchmark of the final implementation if you want to test it on your machine.

This is a long article, and since it also serves as a textbook case study, we will improve the algorithm incrementally for pedagogical goals. If you are already an expert and feel comfortable reading intrinsic-heavy code with little to no context, you can jump straight to the final implementation.

B-Tree Layout

B-trees generalize the concept of binary search trees by allowing nodes to have more than two children. Instead of a single key, a node of a B-tree of order k can contain up to $B = (k - 1)$ keys stored in sorted order and up to k pointers to child nodes. Each child i satisfies the property that all keys in its subtree are between keys $(i - 1)$ and i of the parent node (if they exist).

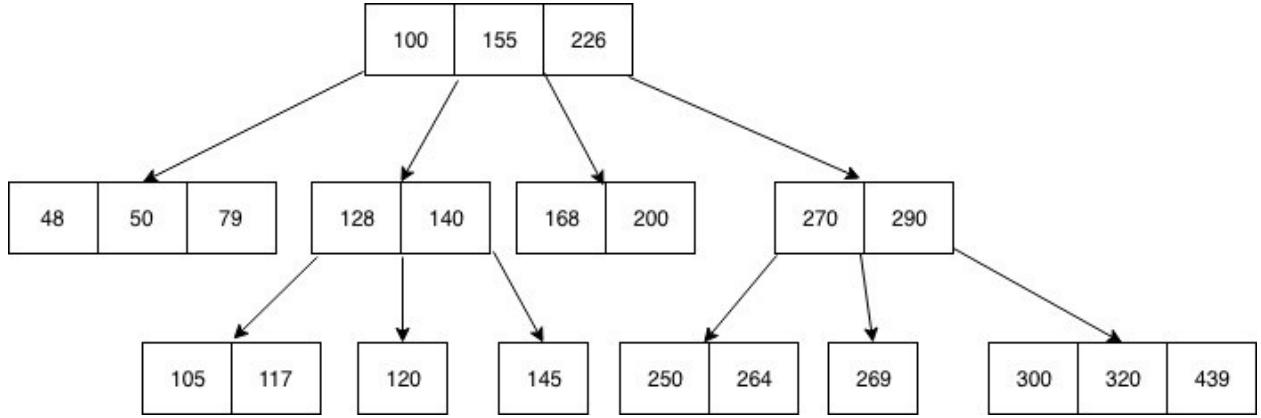


Figure 2: A B-tree of order 4

The main advantage of this approach is that it reduces the tree height by $\frac{\log_2 n}{\log_k n} = \frac{\log k}{\log 2} = \log_2 k$ times, while fetching each node still takes roughly the same time – as long it fits into a single memory block.

B-trees were primarily developed for the purpose of managing on-disk databases, where the latency of randomly fetching a single byte is comparable with the time it takes to read the next 1MB of data sequentially. For our use case, we will be using the block size of $B = 16$ elements – or 64 bytes, the size of the cache line – which makes the tree height and the total number of cache line fetches per query $\log_2 17 \approx 4$ times smaller compared to the binary search.

Implicit B-Tree

Storing and fetching pointers in a B-tree node wastes precious cache space and decreases performance, but they are essential for changing the tree structure on inserts and deletions. But when there are no updates and the structure of a tree is *static*, we can get rid of the pointers, which makes the structure *implicit*.

One of the ways to achieve this is by generalizing the Eytzinger numeration to $(B + 1)$ -ary trees:

- The root node is numbered 0.
- Node k has $(B + 1)$ child nodes numbered $k \cdot (B + 1) + i + 1$ for $i \in [0, B]$.

This way, we can only use $O(1)$ additional memory by allocating one large two-dimensional array of keys and relying on index arithmetic to locate children nodes in the tree:

```

const int B = 16;

int nblocks = (n + B - 1) / B;
  
```

```

int btree[nblocks][B];

int go(int k, int i) { return k * (B + 1) + i + 1; }

```

This numeration automatically makes the B-tree complete or almost complete with the height of $\Theta(\log_{B+1} n)$. If the length of the initial array is not a multiple of B , the last block is padded with the largest value of its data type.

Construction

We can construct the B-tree similar to how we constructed the Eytzinger array – by traversing the search tree:

```

void build(int k = 0) {
    static int t = 0;
    if (k < nblocks) {
        for (int i = 0; i < B; i++) {
            build(go(k, i));
            btree[k][i] = (t < n ? a[t++] : INT_MAX);
        }
        build(go(k, B));
    }
}

```

It is correct because each value of the initial array will be copied to a unique position in the resulting array, and the tree height is $\Theta(\log_{B+1} n)$ because k is multiplied by $(B+1)$ each time we descend into a child node.

Note that this numeration causes a slight imbalance: left-er children may have larger subtrees, although this is only true for $O(\log_{B+1} n)$ parent nodes.

Searches

To find the lower bound, we need to fetch the B keys in a node, find the first key a_i not less than x , descend to the i -th child – and continue until we reach a leaf node. There is some variability in how to find that first key. For example, we could do a tiny internal binary search that makes $O(\log B)$ iterations, or maybe just compare each key sequentially in $O(B)$ time until we find the local lower bound, hopefully exiting from the loop a bit early.

But we are not going to do that – because we can use SIMD. It doesn't work well with branching, so essentially what we want to do is to compare against all B elements regardless, compute a bitmask out of these comparisons, and then use the `ffs` instruction to find the bit corresponding to the first non-lesser element:

```

int mask = (1 << B);

for (int i = 0; i < B; i++)
    mask |= (btree[k][i] >= x) << i;

int i = __builtin_ffs(mask) - 1;
// now i is the number of the correct child node

```

Unfortunately, the compilers are not smart enough to auto-vectorize this code yet, so we have to optimize it manually. In AVX2, we can load 8 elements, compare them against the search key, producing a vector mask, and then extract the scalar mask from it with `movemask`. Here is a minimized illustrated example of what we want to do:

y = 4	17	65	103	
x = 42	42	42	42	
y >= x =	00000000	00000000	11111111	11111111

```

+++++-----+
movemask = 0011
    ++
ffs = 3

```

Since we are limited to processing 8 elements at a time (half our block / cache line size), we have to split the elements into two groups and then combine the two 8-bit masks. To do this, it will be slightly easier to swap the condition for $x > y$ and compute the inverted mask instead:

```

typedef __m256i reg;

int cmp(reg x_vec, int* y_ptr) {
    reg y_vec = _mm256_load_si256((reg*) y_ptr); // load 8 sorted elements
    reg mask = _mm256_cmpgt_epi32(x_vec, y_vec); // compare against the key
    return _mm256_movemask_ps((__m256) mask); // extract the 8-bit mask
}

```

Now, to process the entire block, we need to call it twice and combine the masks:

```

int mask = ~(
    cmp(x, &btree[k][0]) +
    (cmp(x, &btree[k][8]) << 8)
);

```

To descend down the tree, we use `ffs` on that mask to get the correct child number and just call the `go` function we defined earlier:

```

int i = __builtin_ffs(mask) - 1;
k = go(k, i);

```

To actually return the result in the end, we'd want to just fetch `btree[k][i]` in the last node we visited, but the problem is that sometimes the local lower bound doesn't exist ($i \geq B$) because x happens to be greater than all the keys in the node. We could, in theory, do the same thing we did for the Eytzinger binary search and restore the correct element *after* we calculate the last index, but we don't have a nice bit trick this time and have to do a lot of divisions by 17 to compute it, which will be slow and almost certainly not worth it.

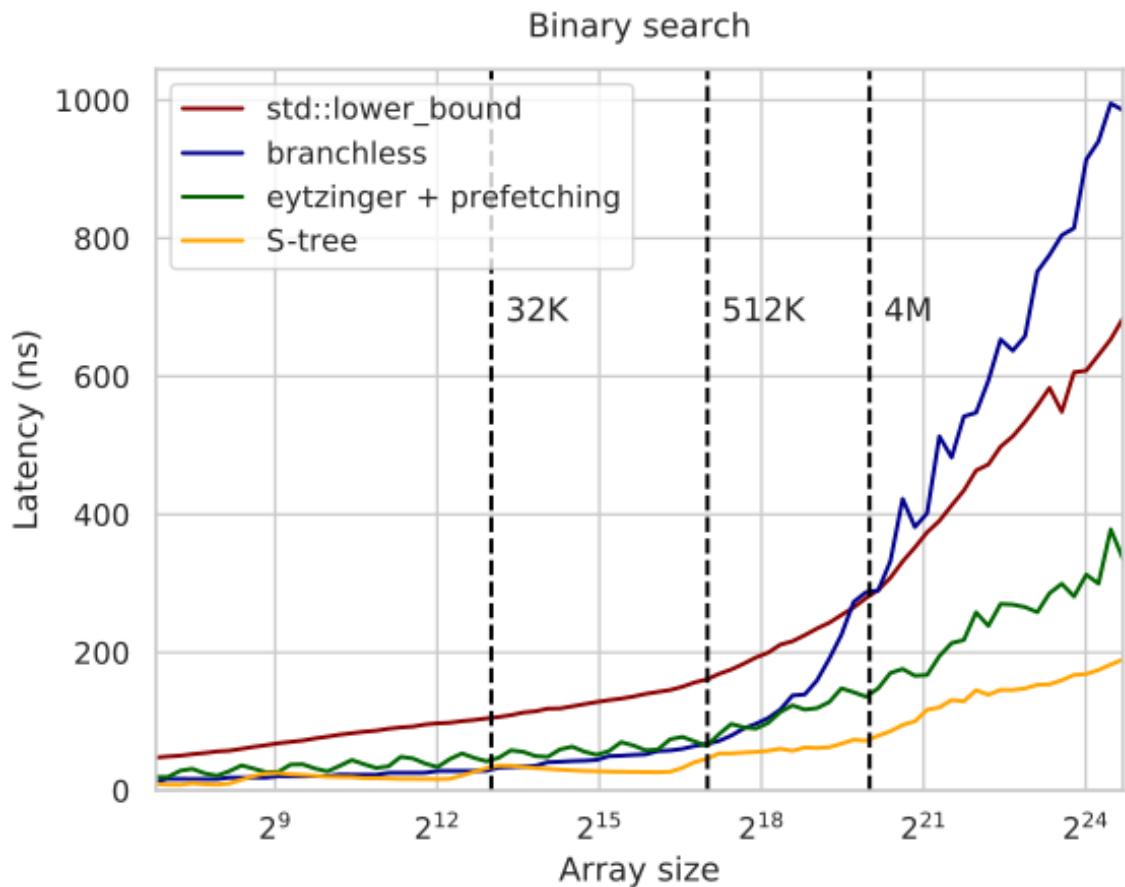
Instead, we can just remember and return the last local lower bound we encountered when we descended the tree:

```

int lower_bound(int _x) {
    int k = 0, res = INT_MAX;
    reg x = _mm256_set1_epi32(_x);
    while (k < nblocks) {
        int mask = ~(
            cmp(x, &btree[k][0]) +
            (cmp(x, &btree[k][8]) << 8)
        );
        int i = __builtin_ffs(mask) - 1;
        if (i < B)
            res = btree[k][i];
        k = go(k, i);
    }
    return res;
}

```

This implementation outperforms all previous binary search implementations, and by a huge margin:



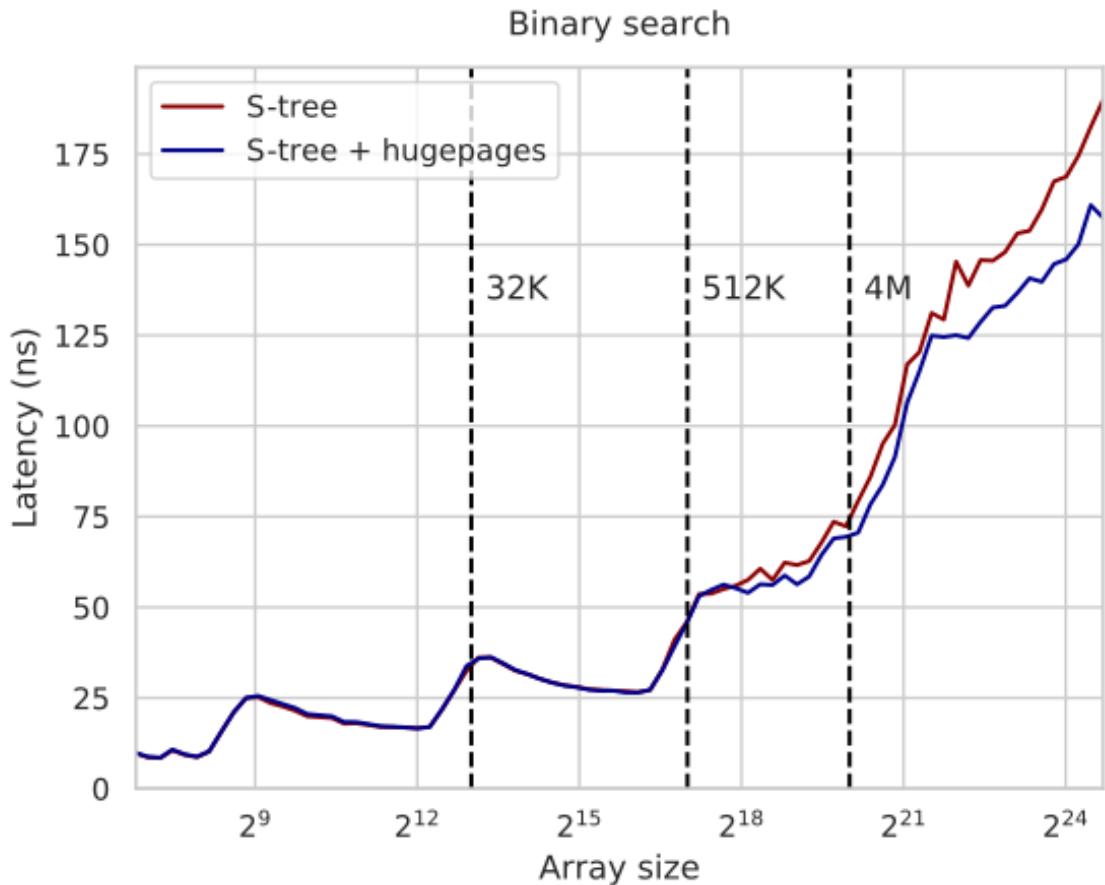
This is very good – but we can optimize it even further.

Optimization

Before everything else, let's allocate the memory for the array on a hugepage:

```
const int P = 1 << 21; // page size in bytes (2MB)
const int T = (64 * nblocks + P - 1) / P * P; // can only allocate whole number of pages
btree = (int(*)[16]) std::aligned_alloc(P, T);
madvise(btreet, T, MADV_HUGEPAGE);
```

This slightly improves the performance on larger array sizes:



Ideally, we'd also need to enable hugepages for all previous implementations to make the comparison fair, but it doesn't matter that much because they all have some form of prefetching that alleviates this problem.

With that settled, let's begin real optimization. First of all, we'd want to use compile-time constants instead of variables as much as possible because it lets the compiler embed them in the machine code, unroll loops, optimize arithmetic, and do all sorts of other nice stuff for us for free. Specifically, we want to know the tree height in advance:

```
constexpr int height(int n) {
    // grow the tree until its size exceeds n elements
    int s = 0, // total size so far
        l = B, // size of the next layer
        h = 0; // height so far
    while (s + l - B < n) {
        s += l;
        l *= (B + 1);
        h++;
    }
    return h;
}
```

```
const int H = height(N);
```

Next, we can find the local lower bound in nodes faster. Instead of calculating it separately for two 8-element blocks and merging two 8-bit masks, we combine the vector masks using the `packs` instruction and readily

extract it using `movemask` just once:

```
unsigned rank(reg x, int* y) {
    reg a = _mm256_load_si256((reg*) y);
    reg b = _mm256_load_si256((reg*) (y + 8));

    reg ca = _mm256_cmpgt_epi32(a, x);
    reg cb = _mm256_cmpgt_epi32(b, x);

    reg c = _mm256_packs_epi32(ca, cb);
    int mask = _mm256_movemask_epi8(c);

    // we need to divide the result by two because we call movemask_epi8 on 16-bit masks:
    return __tzcnt_u32(mask) >> 1;
}
```

This instruction converts 32-bit integers stored in two registers to 16-bit integers stored in one register – in our case, effectively joining the vector masks into one. Note that we've swapped the order of comparison – this lets us not invert the mask in the end, but we have to subtract² one from the search key once in the beginning to make it correct (otherwise, it works as `upper_bound`).

The problem is, it does this weird interleaving where the result is written in the `a1 b1 a2 b2` order instead of `a1 a2 b1 b2` that we want – many AVX2 instructions tend to do that. To correct this, we need to permute the resulting vector, but instead of doing it during the query time, we can just permute every node during preprocessing:

```
void permute(int *node) {
    const reg perm = _mm256_setr_epi32(4, 5, 6, 7, 0, 1, 2, 3);
    reg* middle = (reg*) (node + 4);
    reg x = _mm256_loadu_si256(middle);
    x = _mm256_permutevar8x32_epi32(x, perm);
    _mm256_storeu_si256(middle, x);
}
```

Now we just call `permute(&btree[k])` right after we are done building the node. There are probably faster ways to swap the middle elements, but we will leave it here as the preprocessing time is not that important for now.

This new SIMD routine is significantly faster because the extra `movemask` is slow, and also blending the two masks takes quite a few instructions. Unfortunately, we now can't just do the `res = btree[k][i]` update anymore because the elements are permuted. We can solve this problem with some bit-level trickery in terms of `i`, but indexing a small lookup table turns out to be faster and also doesn't require a new branch:

```
const int translate[17] = {
    0, 1, 2, 3,
    8, 9, 10, 11,
    4, 5, 6, 7,
    12, 13, 14, 15,
    0
};

void update(int &res, int* node, unsigned i) {
    int val = node[translate[i]];

```

²If you need to work with floating-point keys, consider whether `upper_bound` will suffice – because if you need `lower_bound` specifically, then subtracting one or the machine epsilon from the search key doesn't work: you need to get the previous representable number instead. Aside from some corner cases, this essentially means reinterpreting its bits as an integer, subtracting one, and reinterpreting it back as a float (which magically works because of how IEEE-754 floating-point numbers are stored in memory).

```

    res = (i < B ? val : res);
}

```

This update procedure takes some time, but it's not on the critical path between the iterations, so it doesn't affect the actual performance that much.

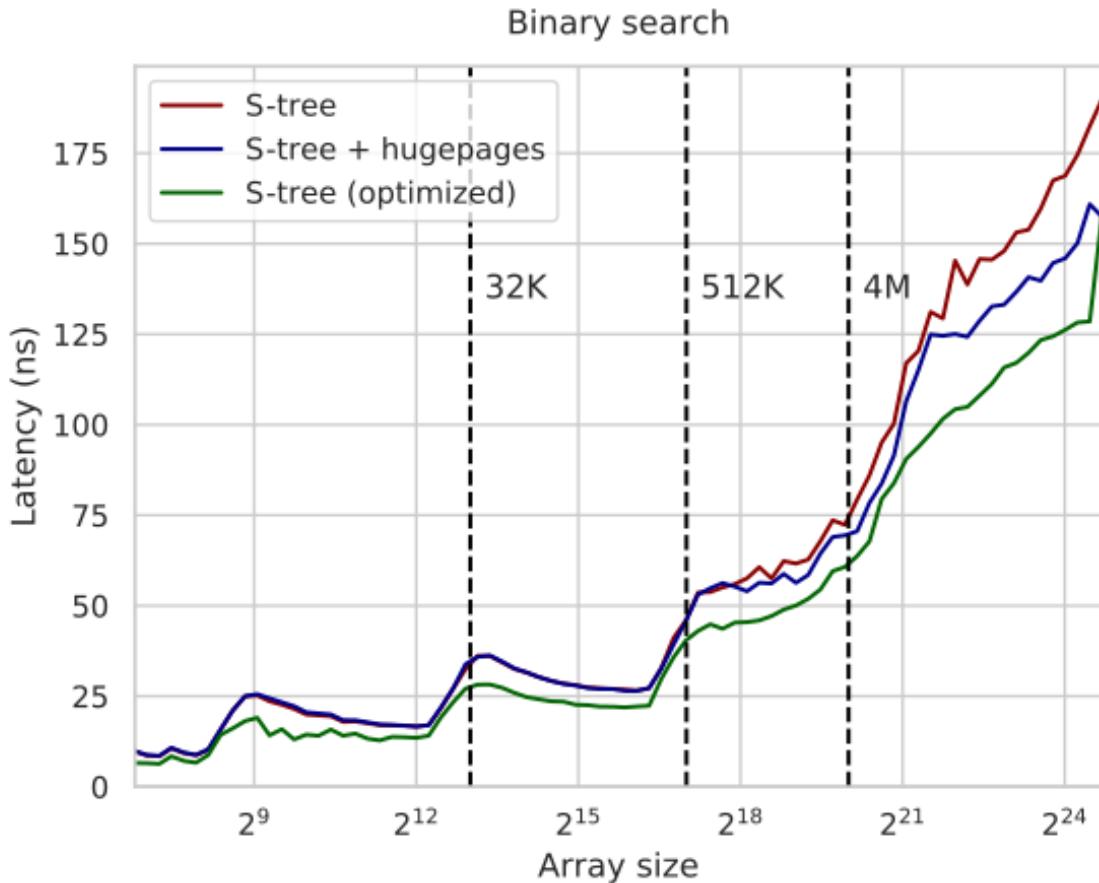
Stitching it all together (and leaving out some other minor optimizations):

```

int lower_bound(int _x) {
    int k = 0, res = INT_MAX;
    reg x = _mm256_set1_epi32(_x - 1);
    for (int h = 0; h < H - 1; h++) {
        unsigned i = rank(x, &btree[k]);
        update(res, &btree[k], i);
        k = go(k, i);
    }
    // the last branch:
    if (k < nblocks) {
        unsigned i = rank(x, btree[k]);
        update(res, &btree[k], i);
    }
    return res;
}

```

All this work saved us 15-20% or so:



It doesn't feel very satisfying so far, but we will reuse these optimization ideas later.

There are two main problems with the current implementation:

- The `update` procedure is quite costly, especially considering that it is very likely going to be useless: 16 out of 17 times, we can just fetch the result from the last block.
- We do a non-constant number of iterations, causing branch prediction problems similar to how it did for the Eytzinger binary search; you can also see it on the graph this time, but the latency bumps have a period of 2^4 .

To address these problems, we need to change the layout a little bit.

B+ Tree Layout

Most of the time, when people talk about B-trees, they really mean *B+ trees*, which is a modification that distinguishes between the two types of nodes:

- *Internal nodes* store up to B keys and $(B + 1)$ pointers to child nodes. The key number i is always equal to the smallest key in the subtree of the $(i + 1)$ -th child node.
- *Data nodes or leaves* store up to B keys, the pointer to the next leaf node, and, optionally, an associated value for each key – if the structure is used as a key-value map.

The advantages of this approach include faster search time (as the internal nodes only store keys) and the ability to quickly iterate over a range of entries (by following next leaf node pointers), but this comes at the cost of some memory overhead: we have to store copies of keys in the internal nodes.

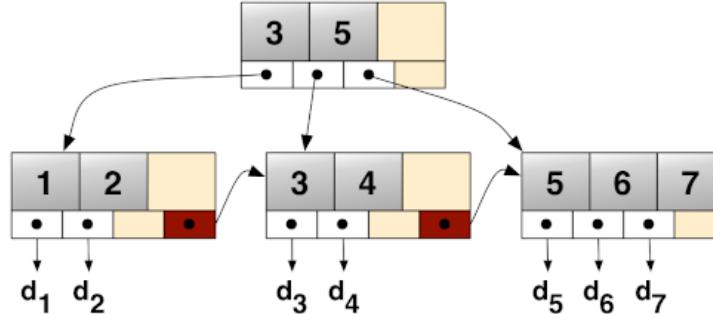


Figure 3: A B+ tree of order 4

Back to our use case, this layout can help us solve our two problems:

- Either the last node we descend into has the local lower bound, or it is the first key of the next leaf node, so we don't need to call `update` on each iteration.
- The depth of all leaves is constant because B+ trees grow at the root and not at the leaves, which removes the need for branching.

The disadvantage is that this layout is not *succinct*: we need some additional memory to store the internal nodes – about $\frac{1}{16}$ -th of the original array size, to be exact – but the performance improvement will be more than worth it.

Implicit B+ Tree

To be more explicit with pointer arithmetic, we will store the entire tree in a single one-dimensional array. To minimize index computations during run time, we will store each layer sequentially in this array and use compile time computed offsets to address them: the keys of the node number k on layer h start with `btree[offset(h) + k * B]`, and its i -th child will at `btree[offset(h - 1) + (k * (B + 1) + i) * B]`.

To implement all that, we need slightly more `constexpr` functions:

```

// number of B-element blocks in a layer with n keys
constexpr int blocks(int n) {
    return (n + B - 1) / B;
}

// number of keys on the layer previous to one with n keys
constexpr int prev_keys(int n) {
    return (blocks(n) + B) / (B + 1) * B;
}

// height of a balanced n-key B+ tree
constexpr int height(int n) {
    return (n <= B ? 1 : height(prev_keys(n)) + 1);
}

// where the layer h starts (layer 0 is the largest)
constexpr int offset(int h) {
    int k = 0, n = N;
    while (h--) {
        k += blocks(n) * B;
        n = prev_keys(n);
    }
    return k;
}

const int H = height(N);
const int S = offset(H); // the tree size is the offset of the (non-existent) layer H

int *btree; // the tree itself is stored in a single hugepage-aligned array of size S

```

Note that we store the layers in reverse order, but the nodes within a layer and data in them are still left-to-right, and also the layers are numbered bottom-up: the leaves form the zeroth layer, and the root is the layer $H - 1$. These are just arbitrary decisions – it is just slightly easier to implement in code.

Construction

To construct the tree from a sorted array \mathbf{a} , we first need to copy it into the zeroth layer and pad it with infinities:

```

memcpy(btree, a, 4 * N);

for (int i = N; i < S; i++)
    btree[i] = INT_MAX;

```

Now we build the internal nodes, layer by layer. For each key, we need to descend to the right of it in, always go left until we reach a leaf node, and then take its first key – it will be the smallest on the subtree:

```

for (int h = 1; h < H; h++) {
    for (int i = 0; i < offset(h + 1) - offset(h); i++) {
        // i = k * B + j
        int k = i / B,
        j = i - k * B;
        k = k * (B + 1) + j + 1; // compare to the right of the key
        // and then always to the left
        for (int l = 0; l < h - 1; l++)
            k *= (B + 1);
    }
}

```

```

    // pad the rest with infinities if the key doesn't exist
    btree[offset(h) + i] = (k * B < N ? btree[k * B] : INT_MAX);
}
}

```

And just the finishing touch – we need to permute keys in internal nodes to search them faster:

```

for (int i = offset(1); i < S; i += B)
    permute(btree + i);

```

We start from `offset(1)`, and we specifically don't permute leaf nodes and leave the array in the original sorted order. The motivation is that we'd need to do this complex index translation we do in `update` if the keys were permuted, and it is on the critical path when this is the last operation. So, just for this layer, we switch to the original mask-blending local lower bound procedure.

Searching

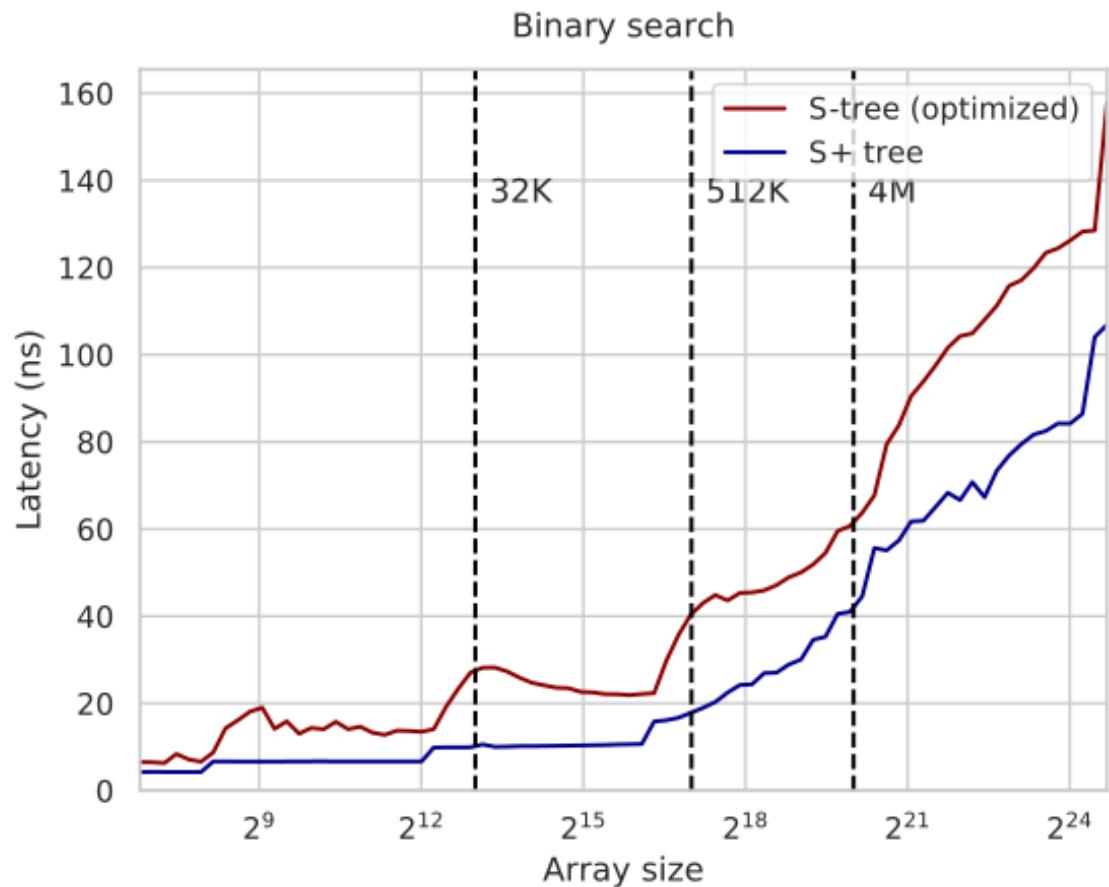
The search procedure becomes simpler than for the B-tree layout: we don't need to do `update` and only execute a fixed number of iterations – although the last one with some special treatment:

```

int lower_bound(int _x) {
    unsigned k = 0; // we assume k already multiplied by B to optimize pointer arithmetic
    reg x = _mm256_set1_epi32(_x - 1);
    for (int h = H - 1; h > 0; h--) {
        unsigned i = permuted_rank(x, btree + offset(h) + k);
        k = k * (B + 1) + i * B;
    }
    unsigned i = direct_rank(x, btree + k);
    return btree[k + i];
}

```

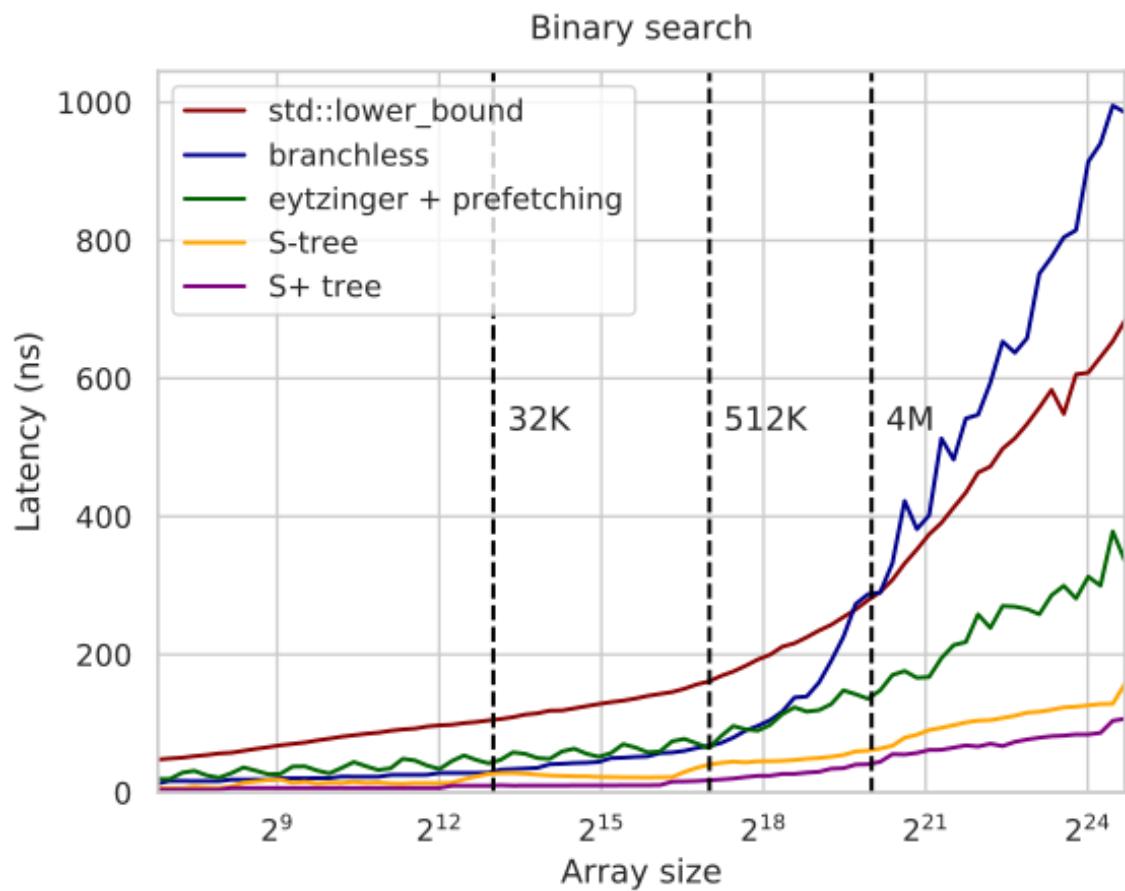
Switching to the B+ layout more than paid off: the S+ tree is 1.5-3x faster compared to the optimized S-tree:



The spikes at the high end of the graph are caused by the L1 TLB not being large enough: it has 64 entries, so it can handle at most $64 \times 2 = 128\text{MB}$ of data, which is exactly what is required for storing 2^{25} integers. The S+ tree hits this limit slightly sooner because of the ~7% memory overhead.

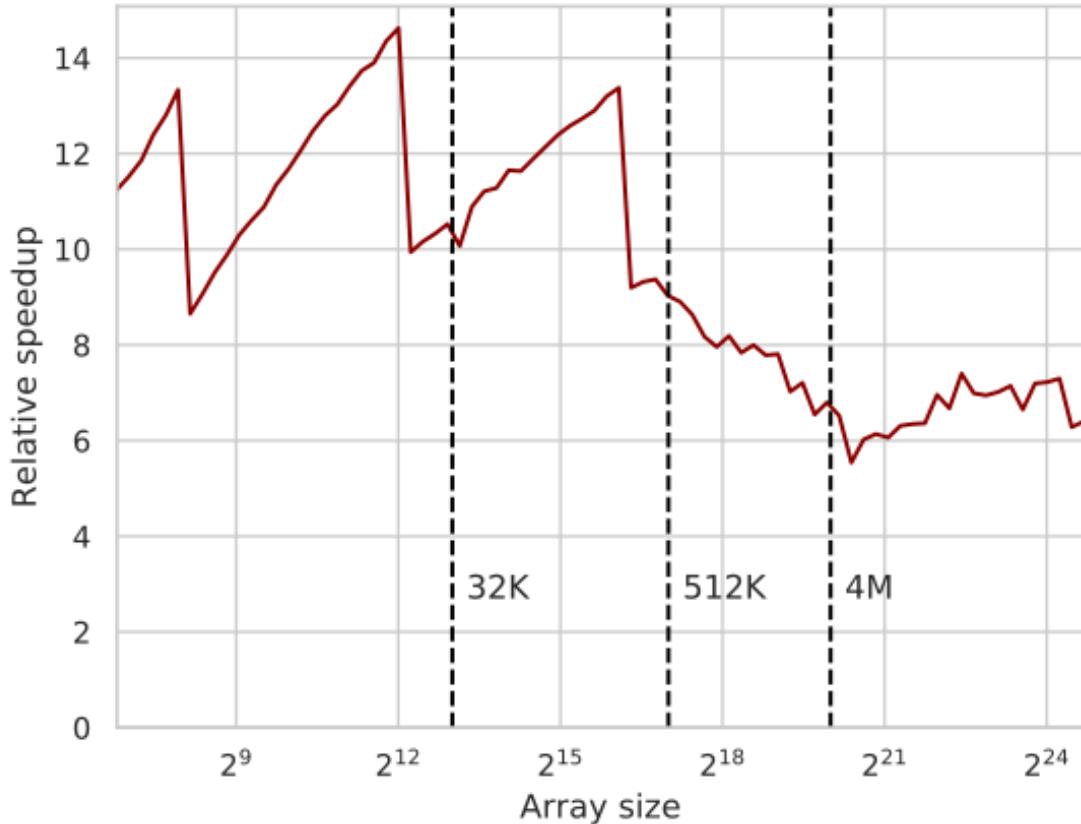
Comparison with `std::lower_bound`

We've come a long way from binary search:



On these scales, it makes more sense to look at the relative speedup:

S+ Tree vs. std::lower_bound



The cliffs at the beginning of the graph are because the running time of `std::lower_bound` grows smoothly with the array size, while for an S+ tree, it is locally flat and increases in discrete steps when a new layer needs to be added.

One important asterisk we haven't discussed is that what we are measuring is not real latency, but the *reciprocal throughput* – the total time it takes to execute a lot of queries divided by the number of queries:

```
clock_t start = clock();

for (int i = 0; i < m; i++)
    checksum ^= lower_bound(q[i]);

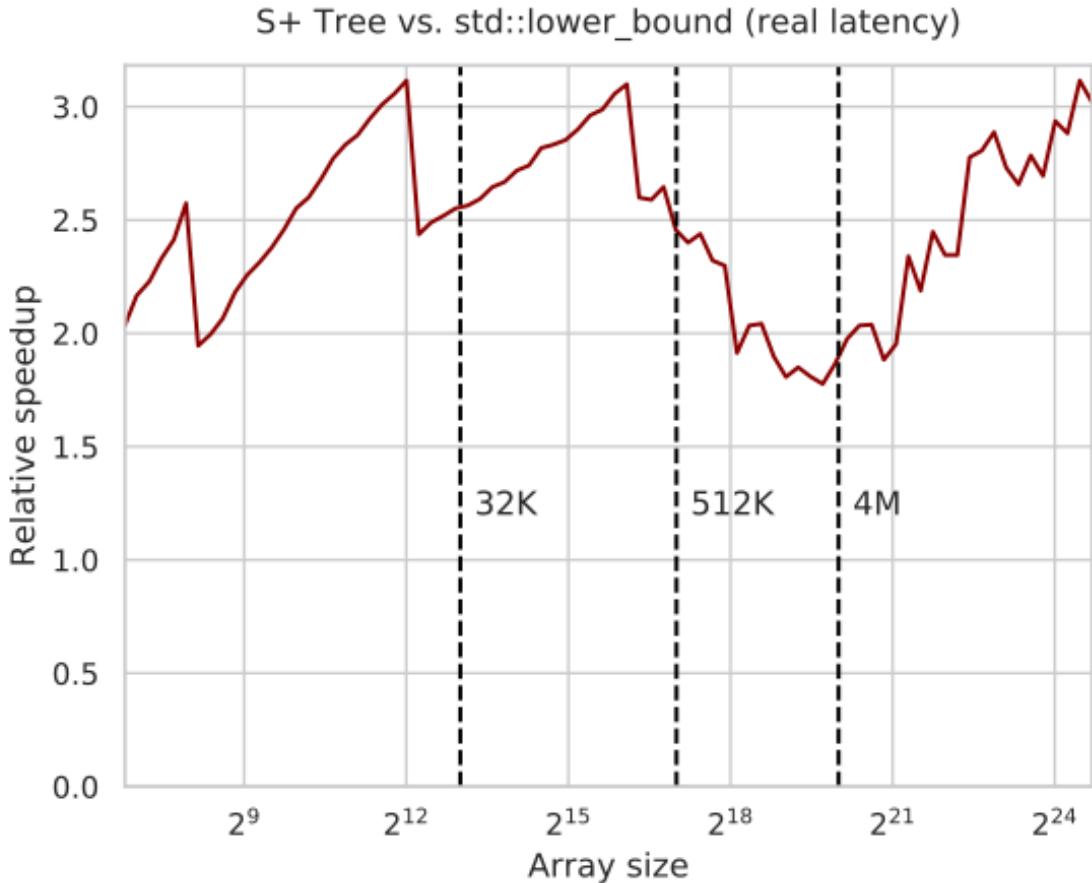
float seconds = float(clock() - start) / CLOCKS_PER_SEC;
printf("%.2f ns per query\n", 1e9 * seconds / m);
```

To measure *actual* latency, we need to introduce a dependency between the loop iterations so that the next query can't start before the previous one finishes:

```
int last = 0;

for (int i = 0; i < m; i++) {
    last = lower_bound(q[i] ^ last);
    checksum ^= last;
}
```

In terms of real latency, the speedup is not that impressive:



A lot of the performance boost of the S+ tree comes from removing branching and minimizing memory requests, which allows overlapping the execution of more adjacent queries – apparently, around three on average.

Although nobody except maybe the HFT people cares about real latency, and everybody actually measures throughput even when using the word “latency,” this nuance is still something to take into account when predicting the possible speedup in user applications.

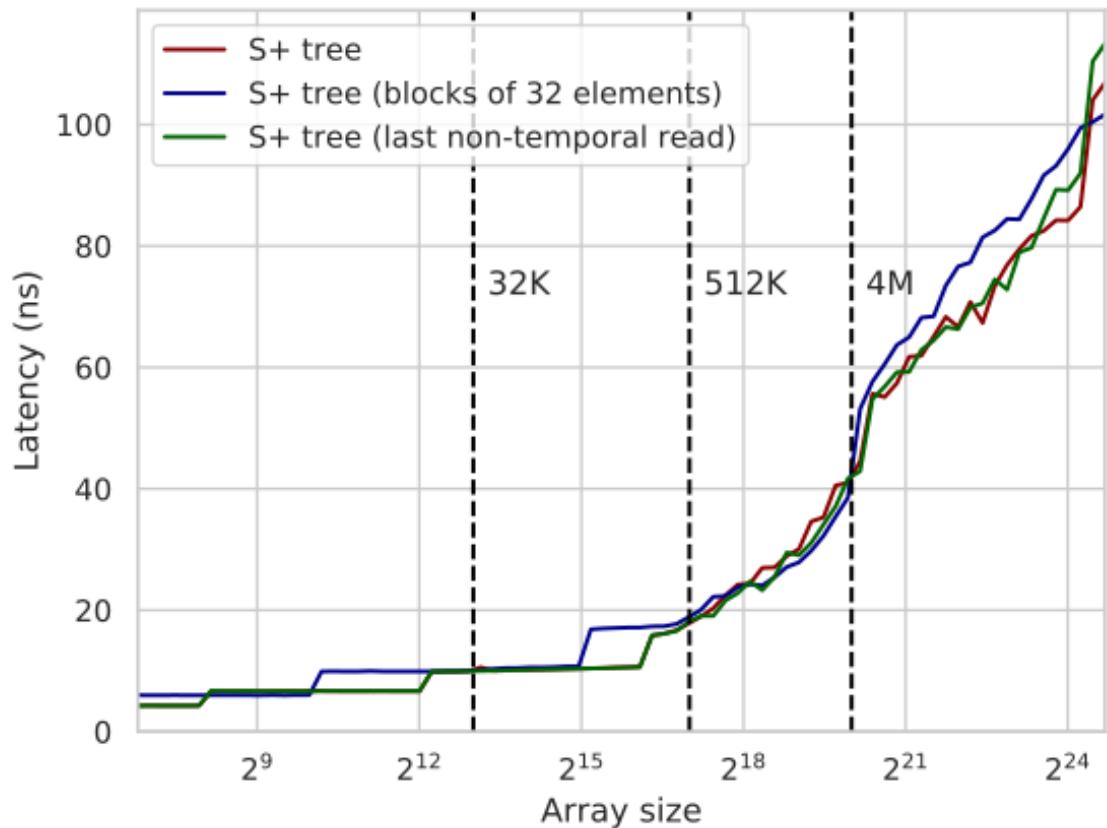
Modifications and Further Optimizations

To minimize the number of memory accesses during a query, we can increase the block size. To find the local lower bound in a 32-element node (spanning two cache lines and four AVX2 registers), we can use a similar trick that uses two `packs_epi32` and one `packs_epi16` to combine masks.

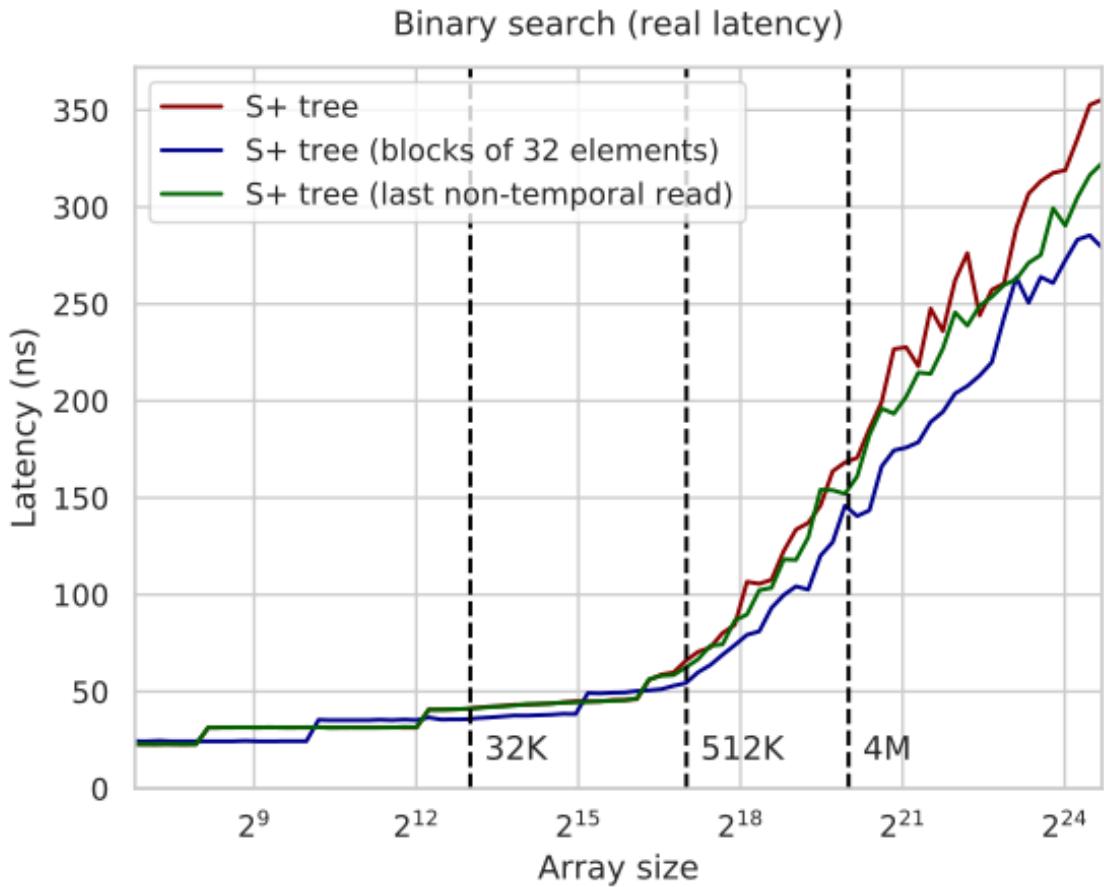
We can also try to use the cache more efficiently by controlling where each tree layer is stored in the cache hierarchy. We can do that by prefetching nodes to a specific level and using non-temporal reads during queries.

I implemented two versions of these optimizations: the one with a block size of 32 and the one where the last read is non-temporal. They don’t improve the throughput:

Binary search



...but they do make the latency lower:



Ideas that I have not yet managed to implement but consider highly perspective are:

- Make the block size non-uniform. The motivation is that the slowdown from having one 32-element layer is less than from having two separate layers. Also, the root is often not full, so perhaps sometimes it should have only 8 keys or even just one key. Picking the optimal layer configuration for a given array size should remove the spikes from the relative speedup graph and make it look more like its upper envelope.

I know how to do it with code generation, but I went for a generic solution and tried to implement it with the facilities of modern C++, but the compiler can't produce optimal code this way.

- Group nodes with one or two generations of its descendants (~300 nodes / ~5k keys) so that they are close in memory – in the spirit of what FAST calls hierarchical blocking. This reduces the severity of TLB misses and also may improve the latency as the memory controller may choose to keep the RAM row buffer open, anticipating local reads.
- Optionally use prefetching on some specific layers. Aside from the $\frac{1}{17}$ -th chance of it fetching the node we need, the hardware prefetcher may also get some of its neighbors for us if the data bus is not busy. It also has the same TLB and row buffer effects as with blocking.

Other possible minor optimizations include:

- Permuting the nodes of the last layer as well – if we only need the index and not the value.
- Reversing the order in which the layers are stored to left-to-right so that the first few layers are on the same page.
- Rewriting the whole thing in assembly, as the compiler seems to struggle with pointer arithmetic.

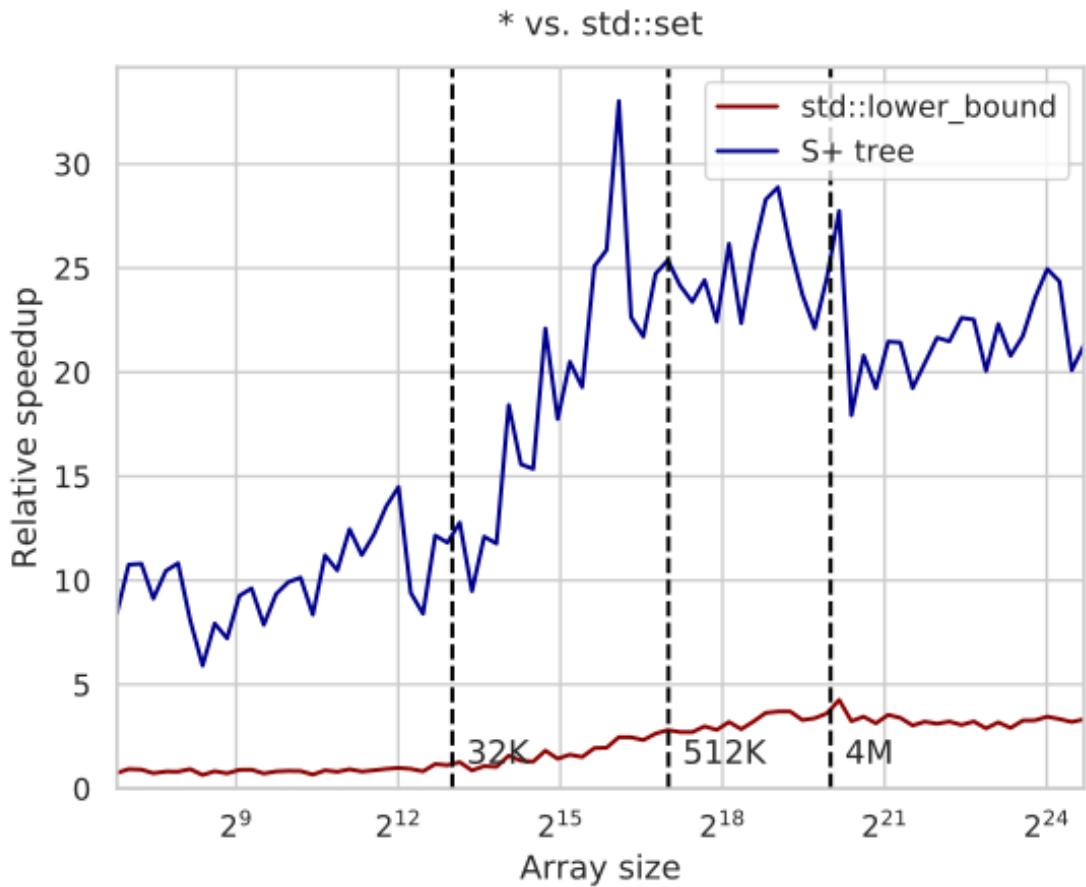
- Using blending instead of `packs`: you can odd-even shuffle node keys ($[1\ 3\ 5\ 7]\ [2\ 4\ 6\ 8]$), compare against the search key, and then blend the low 16 bits of the first register mask with the high 16 bits of the second. Blending is slightly faster on many architectures, and it may also help to alternate between packing and blending as they use different subsets of ports. (Thanks to Const-me from HackerNews for suggesting it.)
- Using `popcount` instead of `tzcnt`: the index i is equal to the number of keys less than x , so we can compare x against all keys, combine the vector mask any way we want, call `maskmov`, and then calculate the number of set bits with `popcnt`. This removes the need to store the keys in any particular order, which lets us skip the permutation step and also use this procedure on the last layer as well.
- Defining the key i as the *maximum* key in the subtree of child i instead of the *minimum* key in the subtree of child $(i + 1)$. The correctness doesn't change, but this guarantees that the result will be stored in the last node we access (and not in the first element of the next neighbor node), which lets us fetch slightly fewer cache lines.

Note that the current implementation is specific to AVX2 and may require some non-trivial changes to adapt to other platforms. It would be interesting to port it for Intel CPUs with AVX-512 and Arm CPUs with 128-bit NEON, which may require some trickery to work.

With these optimizations implemented, I wouldn't be surprised to see another 10-30% improvement and over 10x speedup over `std::lower_bound` on large arrays for some platforms.

As a Dynamic Tree

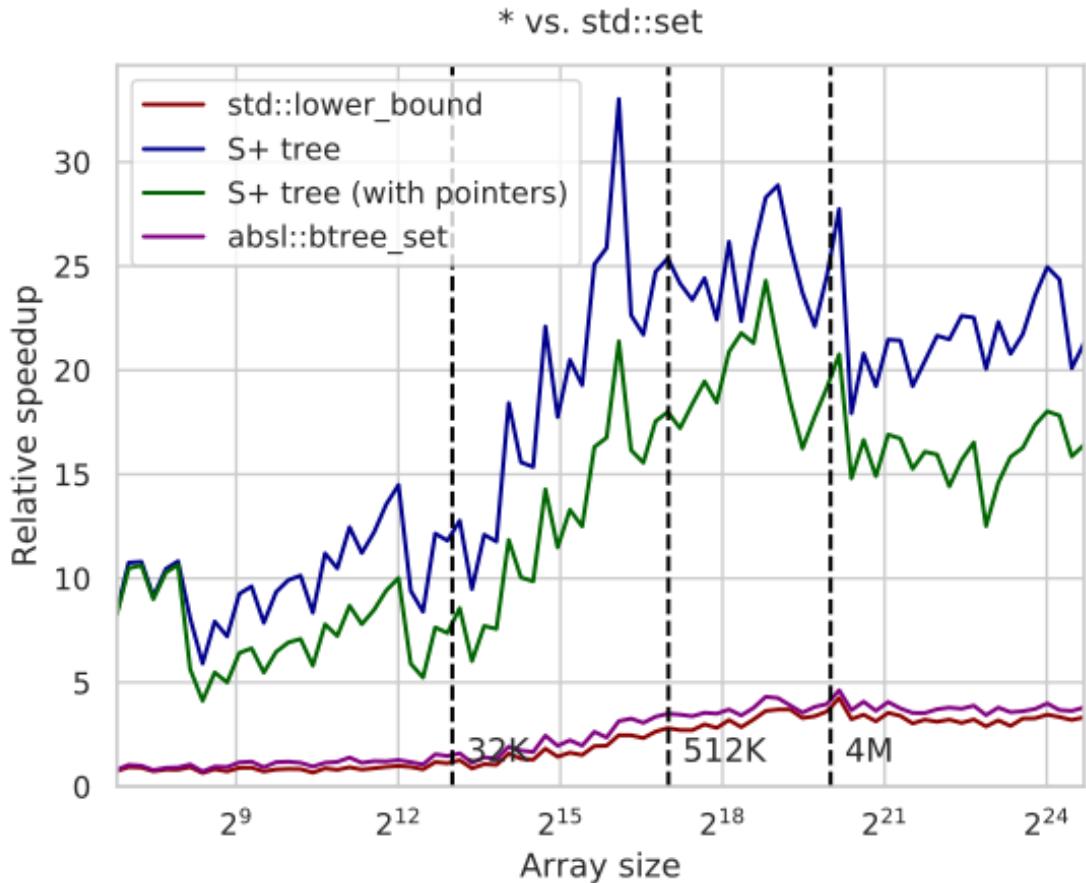
The comparison is even more favorable against `std::set` and other pointer-based trees. In our benchmark, we add the same elements (without measuring the time it takes to add them) and use the same lower bound queries, and the S+ tree is up to 30x faster:



This suggests that we can probably use this approach to also improve on *dynamic* search trees by a large margin.

To validate this hypothesis, I added an array of 17 indices for each node that point to where their children should be and used this array to descend the tree instead of the usual implicit numbering. This array is separate from the tree, not aligned, and isn't even on a hugepage – the only optimization we do is prefetch the first and the last pointer of a node.

I also added B-tree from Abseil to the comparison, which is the only widely-used B-tree implementation I know of. It performs just slightly better than `std::lower_bound`, while the S+ tree with pointers is ~15x faster for large arrays:



Of course, this comparison is not fair, as implementing a dynamic search tree is a more high-dimensional problem.

We'd also need to implement the update operation, which will not be that efficient, and for which we'd need to sacrifice the fanout factor. But it still seems possible to implement a 10-20x faster `std::set` and a 3-5x faster `absl::btree_set`, depending on how you define "faster" – and this is one of the things we'll attempt to do next.

Acknowledgements

This StackOverflow answer by Cory Nelson is where I took the permuted 16-element search trick from.

Search Trees

In the previous article, we designed and implemented *static* B-trees to speed up binary searching in sorted arrays. In its last section, we briefly discussed how to make them *dynamic* back while retaining the performance gains from SIMD and validated our predictions by adding and following explicit pointers in the internal nodes of the S+ tree.

In this article, we follow up on that proposition and design a minimally functional search tree for integer keys, achieving up to 18x/8x speedup over `std::set` and up to 7x/2x speedup over `absl::btree` for `lower_bound` and `insert` queries, respectively – with yet ample room for improvement.

The memory overhead of the structure is around 30% for 32-bit integers, and the final implementation is under 150 lines of C++. It can be easily generalized to other arithmetic types and small/fixed-length strings such as hashes, country codes, and stock symbols.

B- Tree

Instead of making small incremental improvements like we usually do in other case studies, in this article, we will implement just one data structure that we name *B- tree*, which is based on the B+ tree, with a few minor differences:

- Nodes in the B- tree do not store pointers or any metadata except for the pointers to internal node children (while the B+ tree leaf nodes store a pointer to the next leaf node). This lets us perfectly place the keys in the leaf nodes on cache lines.
- We define key i to be the *maximum* key in the subtree of the child i instead of the *minimum* key in the subtree of the child $(i + 1)$. This lets us not fetch any other nodes after we reach a leaf (in the B+ tree, all keys in the leaf node may be less than the search key, so we need to go to the next leaf node to fetch its first element).

We also use a node size of $B = 32$, which is smaller than typical. The reason why it is not 16, which was optimal for the S+ tree, is because we have the additional overhead associated with fetching the pointer, and the benefit of reducing the tree height by ~20% outweighs the cost of processing twice the elements per node, and also because it improves the running time of the `insert` query that needs to perform a costly node split every $\frac{B}{2}$ insertions on average.

Memory Layout

Although this is probably not the best approach in terms of software engineering, we will simply store the entire tree in a large pre-allocated array, without discriminating between leaves and internal nodes:

```
const int R = 1e8;
alignas(64) int tree[R];
```

We also pre-fill this array with infinities to simplify the implementation:

```
for (int i = 0; i < R; i++)
    tree[i] = INT_MAX;
```

(In general, it is technically cheating to compare against `std::set` or other structures that use `new` under the hood, but memory allocation and initialization are not the bottlenecks here, so this does not significantly affect the evaluation.)

Both nodes types store their keys sequentially in sorted order and are identified by the index of its first key in the array:

- A leaf node has up to $(B - 1)$ keys but is padded to B elements with infinities.
- An internal node has up to $(B - 2)$ keys padded to B elements and up to $(B - 1)$ indices of its child nodes, also padded to B elements.

These design decisions are not arbitrary:

- The padding ensures that leaf nodes occupy exactly 2 cache lines and internal nodes occupy exactly 4 cache lines.
- We specifically use indices instead of pointers to save cache space and make moving them with SIMD faster.
(We will use “pointer” and “index” interchangeably from now on.)
- We store indices right after the keys even though they are stored in separate cache lines because we have reasons.
- We intentionally “waste” one array cell in leaf nodes and $2 + 1 = 3$ cells in internal nodes because we need it to store temporary results during a node split.

Initially, we only have one empty leaf node as the root:

```
const int B = 32;

int root = 0; // where the keys of the root start
```

```

int n_tree = B; // number of allocated array cells
int H = 1;      // current tree height

```

To “allocate” a new node, we simply increase `n_tree` by B if it is a leaf node or by $2B$ if it is an internal node.

Since new nodes can only be created by splitting a full node, each node except for the root will be at least half full. This implies that we need between 4 and 8 bytes per integer element (the internal nodes will contribute $\frac{1}{16}$ -th or so to that number), the former being the case when the inserts are sequential, and the latter being the case when the input is adversarial. When the queries are uniformly distributed, the nodes are $\sim 75\%$ full on average, projecting to ~ 5.2 bytes per element.

B-trees are very memory-efficient compared to the pointer-based binary trees. For example, `std::set` needs at least three pointers (the left child, the right child, and the parent), alone costing $3 \times 8 = 24$ bytes, plus at least another 8 bytes to store the key and the meta-information due to structure padding.

Searching

It is a very common scenario when $>90\%$ of operations are lookups, and even if this is not the case, every other tree operation typically begins with locating a key anyway, so we will start with implementing and optimizing the searches.

When we implemented S-trees, we ended up storing the keys in permuted order due to the intricacies of how the blending/packs instructions work. For the *dynamic tree* problem, storing the keys in permuted order would make inserts much harder to implement, so we will change the approach instead.

An alternative way to think about finding the would-be position of the element `x` in a sorted array is not “the index of the first element that is not less than `x`” but “the number of elements that are less than `x`.`”`This observation generates the following idea: compare the keys against `x`, aggregate the vector masks into a 32-bit mask (where each bit can correspond to any element as long as the mapping is bijective), and then call `popcnt` on it, returning the number of elements less than `x`.

This trick lets us perform the local search efficiently and without requiring any shuffling:

```

typedef __m256i reg;

reg cmp(reg x, int *node) {
    reg y = _mm256_load_si256((reg*) node);
    return _mm256_cmpgt_epi32(x, y);
}

// returns how many keys are less than x
unsigned rank32(reg x, int *node) {
    reg m1 = cmp(x, node);
    reg m2 = cmp(x, node + 8);
    reg m3 = cmp(x, node + 16);
    reg m4 = cmp(x, node + 24);

    // take lower 16 bits from m1/m3 and higher 16 bits from m2/m4
    m1 = _mm256_blend_epi16(m1, m2, 0b01010101);
    m3 = _mm256_blend_epi16(m3, m4, 0b01010101);
    m1 = _mm256_packs_epi16(m1, m3); // can also use blendv here, but packs is simpler

    unsigned mask = _mm256_movemask_epi8(m1);
    return __builtin_popcount(mask);
}

```

Note that, because of this procedure, we have to pad the “key area” with infinities, which prevents us from

storing metadata in the vacated cells (unless we are also willing to spend a few cycles to mask it out when loading a SIMD lane).

Now, to implement `lower_bound`, we can descend the tree just like we did in the S+ tree, but fetching the pointer after we compute the child number:

```
int lower_bound(int _x) {
    unsigned k = root;
    reg x = _mm256_set1_epi32(_x);

    for (int h = 0; h < H - 1; h++) {
        unsigned i = rank32(x, &tree[k]);
        k = tree[k + B + i];
    }

    unsigned i = rank32(x, &tree[k]);

    return tree[k + i];
}
```

Implementing search is easy, and it doesn't introduce much overhead. The hard part is implementing insertion.

Insertion

On the one side, correctly implementing insertion takes a lot of code, but on the other, most of that code is executed very infrequently, so we don't have to care about its performance that much. Most often, all we need to do is to reach the leaf node (which we've already figured out how to do) and then insert a new key into it, moving some suffix of the keys one position to the right. Occasionally, we also need to split the node and/or update some ancestors, but this is relatively rare, so let's focus on the most common execution path first.

To insert a key into an array of $(B - 1)$ sorted elements, we can load them in vector registers and then mask-store them one position to the right using a precomputed mask that tells which elements need to be written for a given i :

```
struct Precalc {
    alignas(64) int mask[B][B];

    constexpr Precalc() : mask{} {
        for (int i = 0; i < B; i++)
            for (int j = i; j < B - 1; j++)
                // everything from i to B - 2 inclusive needs to be moved
                mask[i][j] = -1;
    }
};

constexpr Precalc P;

void insert(int *node, int i, int x) {
    // need to iterate right-to-left to not overwrite the first element of the next lane
    for (int j = B - 8; j >= 0; j -= 8) {
        // load the keys
        reg t = _mm256_load_si256((reg*) &node[j]);
        // load the corresponding mask
        reg mask = _mm256_load_si256((reg*) &P.mask[i][j]);
        // mask-write them one position to the right
    }
}
```

```

        _mm256_maskstore_epi32(&node[j + 1], mask, t);
    }
    node[i] = x; // finally, write the element itself
}

```

This constexpr magic is the only C++ feature we use.

There are other ways to do it, some possibly more efficient, but we are going to stop there for now.

When we split a node, we need to move half of the keys to another node, so let's write another primitive that does it:

```

// move the second half of a node and fill it with infinities
void move(int *from, int *to) {
    const reg infs = _mm256_set1_epi32(INT_MAX);
    for (int i = 0; i < B / 2; i += 8) {
        reg t = _mm256_load_si256((reg*) &from[B / 2 + i]);
        _mm256_store_si256((reg*) &to[i], t);
        _mm256_store_si256((reg*) &from[B / 2 + i], infs);
    }
}

```

With these two vector functions implemented, we can now very carefully implement insertion:

```

void insert(int _x) {
    // the beginning of the procedure is the same as in lower_bound,
    // except that we save the path in case we need to update some of our ancestors
    unsigned sk[10], si[10]; // k and i on each iteration
    //           ^-----^ We assume that the tree height does not exceed 10
    //                   (which would require at least 16^10 elements)

    unsigned k = root;
    reg x = _mm256_set1_epi32(_x);

    for (int h = 0; h < H - 1; h++) {
        unsigned i = rank32(x, &tree[k]);

        // optionally update the key i right away
        tree[k + i] = (_x > tree[k + i] ? _x : tree[k + i]);
        sk[h] = k, si[h] = i; // and save the path

        k = tree[k + B + i];
    }

    unsigned i = rank32(x, &tree[k]);

    // we can start computing the is-full check before insertion completes
    bool filled = (tree[k + B - 2] != INT_MAX);

    insert(tree + k, i, _x);

    if (filled) {
        // the node needs to be split, so we create a new leaf node
        move(tree + k, tree + n_tree);

        int v = tree[k + B / 2 - 1]; // new key to be inserted
        int p = n_tree;             // pointer to the newly created node
    }
}

```

```

n_tree += B;

for (int h = H - 2; h >= 0; h--) {
    // ascend and repeat until we reach the root or find a node is not split
    k = sk[h], i = si[h];

    filled = (tree[k + B - 3] != INT_MAX);

    // the node already has a correct key (the right one)
    // and a correct pointer (the left one)
    insert(tree + k, i, v);
    insert(tree + k + B, i + 1, p);

    if (!filled)
        return; // we're done

    // create a new internal node
    move(tree + k, tree + n_tree); // move keys
    move(tree + k + B, tree + n_tree + B); // move pointers

    v = tree[k + B / 2 - 1];
    tree[k + B / 2 - 1] = INT_MAX;

    p = n_tree;
    n_tree += 2 * B;
}

// if reach here, this means we've reached the root,
// and it was split into two, so we need a new root
tree[n_tree] = v;

tree[n_tree + B] = root;
tree[n_tree + B + 1] = p;

root = n_tree;
n_tree += 2 * B;
H++;
}
}

```

There are many inefficiencies, but, luckily, the body of `if (filled)` is executed very infrequently – approximately every $\frac{B}{2}$ insertions – and the insertion performance is not really our top priority, so we will just leave it there.

Evaluation

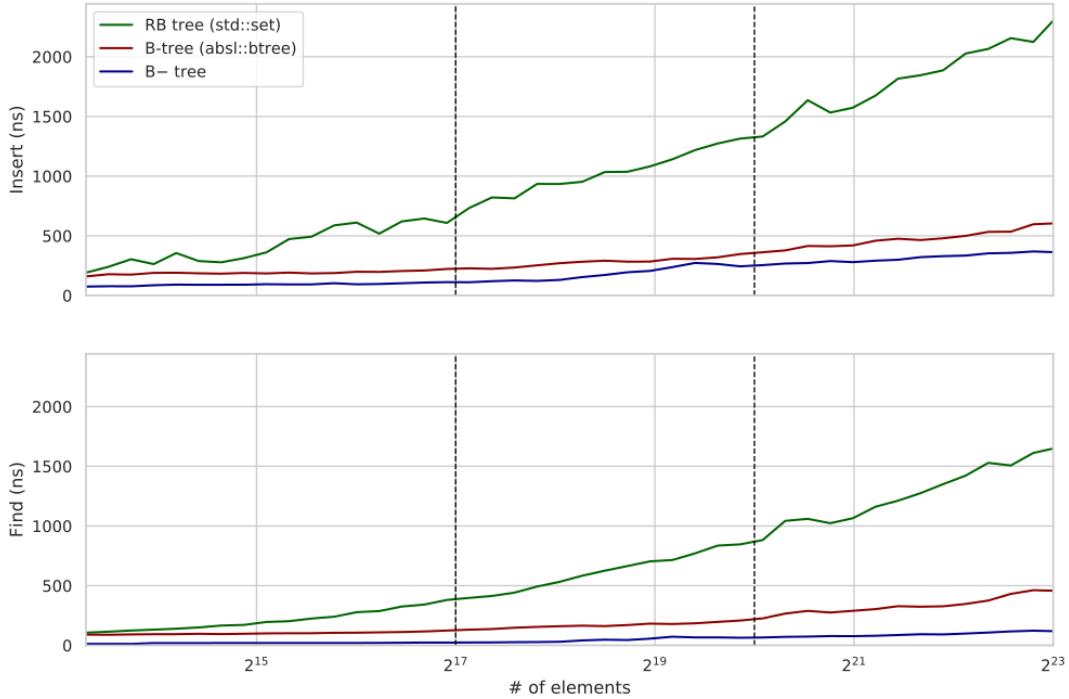
We have only implemented `insert` and `lower_bound`, so this is what we will measure.

We want the evaluation to take a reasonable time, so our benchmark is a loop that alternates between two steps:

- Increase the structure size from 1.17^k to 1.17^{k+1} using individual `inserts` and measure the time it took.
- Perform 10^6 random `lower_bound` queries and measure the time it took.

We start at the size 10^4 and end at 10^7 , for around 50 data points in total. We generate the data for both query types uniformly in the $[0, 2^{30}]$ range and independently between the stages. Since the data generation process allows for repeated keys, we compared against `std::multiset` and `absl::btree_multiset`³, although we still refer to them as `std::set` and `absl::btree` for brevity. We also enable hugepages on the system level for all three runs.

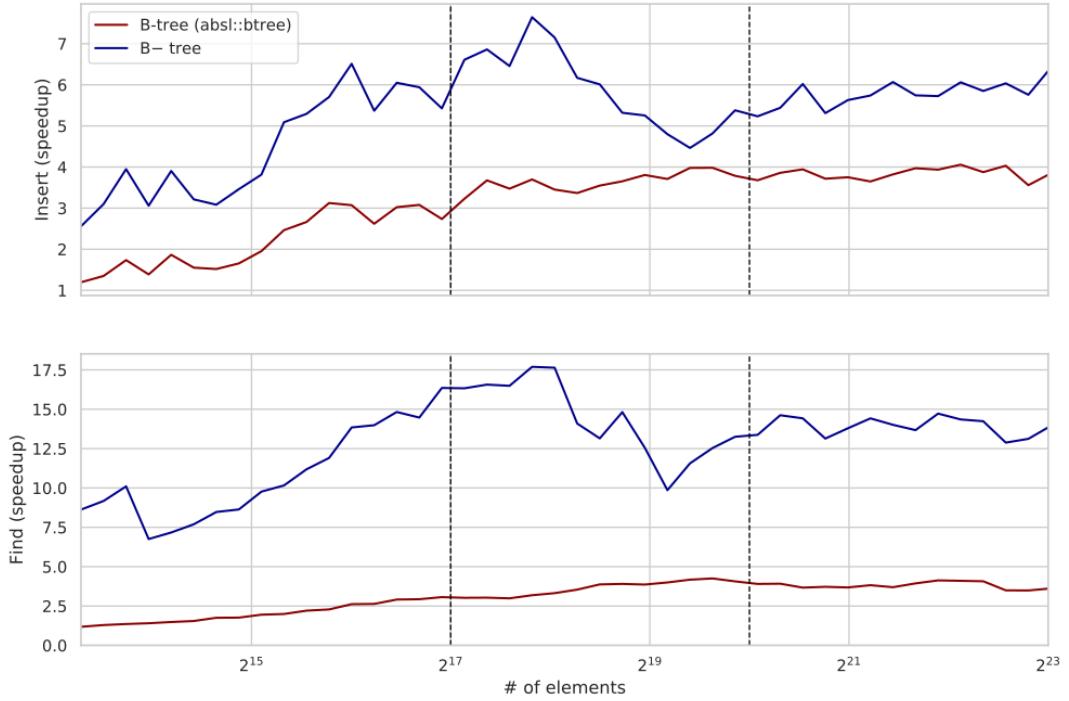
The performance of the B-tree matches what we originally predicted – at least for the lookups:



The relative speedup varies with the structure size – 7-18x/3-8x over STL and 3-7x/1.5-2x over Abseil:

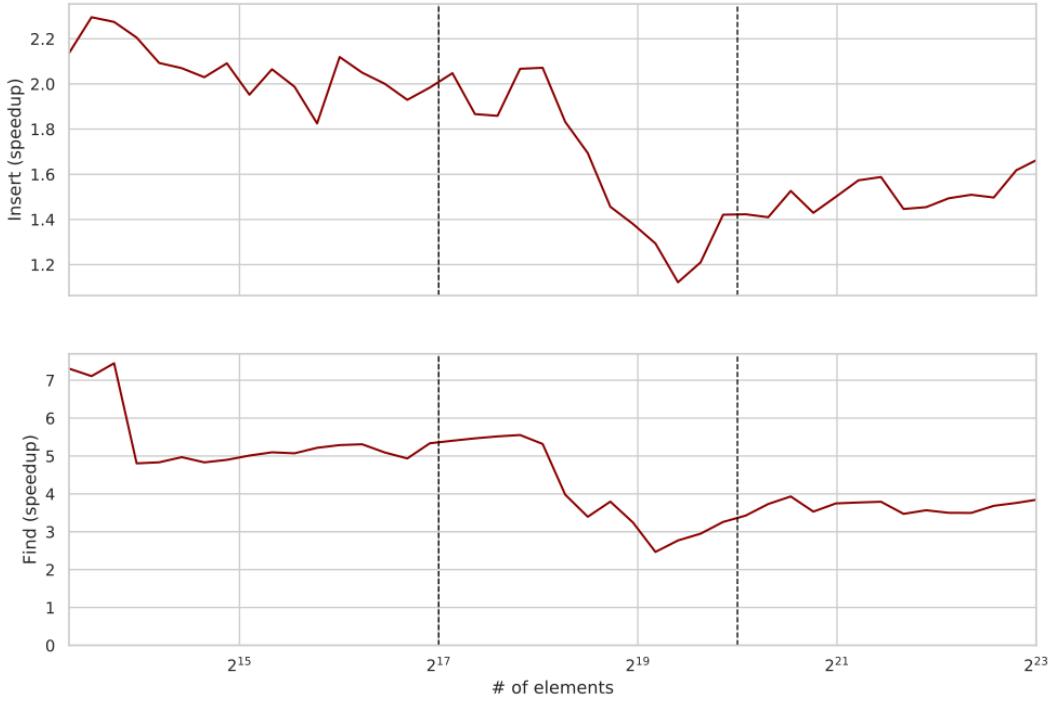
³If you also think that only comparing with Abseil's B-tree is not convincing enough, feel free to add your favorite search tree to the benchmark.

* vs. std::set



Insertions are only 1.5-2 faster than for `absl::btree`, which uses scalar code to do everything. My best guess why insertions are *that* slow is due to data dependency: since the tree nodes may change, the CPU can't start processing the next query before the previous one finishes (the true latency of both queries is roughly equal and ~3x of the reciprocal throughput of `lower_bound`).

B- tree vs. `absl::btree`



When the structure size is small, the reciprocal throughput of `lower_bound` increases in discrete steps: it starts with 3.5ns when there is only the root to visit, then grows to 6.5ns (two nodes), and then to 12ns (three nodes), and then hits the L2 cache (not shown on the graphs) and starts increasing more smoothly, but still with noticeable spikes when the tree height increases.

Interestingly, B- tree outperforms `absl::btree` even when it only stores a single key: it takes around 5ns stalling on branch misprediction, while (the search in) the B- tree is entirely branchless.

Possible Optimizations

In our previous endeavors in data structure optimization, it helped a lot to make as many variables as possible compile-time constants: the compiler can hardcode these constants into the machine code, simplify the arithmetic, unroll all the loops, and do many other nice things for us.

This would not be a problem at all if our tree were of constant height, but it is not. It is *largely* constant, though: the height rarely changes, and in fact, under the constraints of the benchmark, the maximum height was only 6.

What we can do is pre-compile the `insert` and `lower_bound` functions for several different compile-time constant heights and switch between them as the tree grows. The idiomatic C++ way is to use virtual functions, but I prefer to be explicit and use raw function pointers like this:

```
void (*insert_ptr)(int);
int (*lower_bound_ptr)(int);

void insert(int x) {
    insert_ptr(x);
}
```

```

int lower_bound(int x) {
    return lower_bound_ptr(x);
}

```

We now define template functions that have the tree height as a parameter, and in the grow-tree block inside the `insert` function, we change the pointers as the tree grows:

```

template <int H>
void insert_impl(int _x) {
    // ...
}

template <int H>
void insert_impl(int _x) {
    // ...
    if /* tree grows */ {
        // ...
        insert_ptr = &insert_impl<H + 1>;
        lower_bound_ptr = &lower_bound_impl<H + 1>;
    }
}

template <>
void insert_impl<10>(int x) {
    std::cerr << "This depth was not supposed to be reached" << std::endl;
    exit(1);
}

```

I tried but could not get any performance improvement with this, but I still have high hope for this approach because the compiler can (theoretically) remove `sk` and `si`, completely removing any temporary storage and only reading and computing everything once, greatly optimizing the `insert` procedure.

Insertion can also probably be optimized by using a larger block size as node splits would become rare, but this comes at the cost of slower lookups. We could also try different node sizes for different layers: leaves should probably be larger than the internal nodes.

Another idea is to move extra keys on `insert` to a sibling node, delaying the node split as long as possible.

One such particular modification is known as the B* tree. It moves the last key to the next node if the current one is full, and when both nodes become full, it jointly splits both of them, producing three nodes that are 2/3 full. This reduces the memory overhead (the nodes will be 5/6 full on average) and increases the fanout factor, reducing the height, which helps all operations.

This technique can even be extended to, say, three-to-four splits, although further generalization would come at the cost of a slower `insert`.

And yet another idea is to get rid of (some) pointers. For example, for large trees, we can probably afford a small S+ tree for $16 \cdot 17$ or so elements as the root, which we rebuild from scratch on each infrequent occasion when it changes. You can't extend it to the whole tree, unfortunately: I believe there is a paper somewhere saying that we can't turn a dynamic structure fully implicit without also having to do $\Omega(\sqrt{n})$ operations per query.

We could also try some non-tree data structures, such as the skip list. There has even been a successful attempt to vectorize it – although the speedup was not that impressive. I have low hope that skip-list, in particular, can be improved, although it may achieve a higher total throughput in the concurrent setting.

Other Operations

To *delete* a key, we can similarly locate and remove it from a node with the same mask-store trick. After that, if the node is at least half-full, we're done. Otherwise, we try to borrow a key from the next sibling. If the sibling has more than $\frac{B}{2}$ keys, we append its first key and shift its keys one to the left. Otherwise, both the current node and the next node have less than $\frac{B}{2}$ keys, so we can merge them, after which we go to the parent and iteratively delete a key there.

Another thing we may want to implement is *iteration*. Bulk-loading each key from 1 to r is a very common pattern – for example, in `SELECT abc ORDER BY xyz` type of queries in databases – and B+ trees usually store pointers to the next node in the data layer to allow for this type of rapid iteration. In B-trees, as we're using a much smaller node size, we can experience pointer chasing problems if we do this. Going to the parent and reading all its B pointers is probably faster as it negates this problem. Therefore, a stack of ancestors (the `sk` and `si` arrays we used in `insert`) can serve as an iterator and may even be better than separately storing pointers in nodes.

We can easily implement almost everything that `std::set` does, but the B-tree, like any other B-tree, is very unlikely to become a drop-in replacement to `std::set` due to the requirement of pointer stability: a pointer to an element should remain valid unless the element is deleted, which is hard to achieve when we split and merge nodes all the time. This is a major problem not only for search trees but most data structures in general: having both pointer stability and high performance at the same time is next to impossible.

Acknowledgements

Thanks to Danila Kutenin from Google for meaningful discussions of applicability and the usage of B-trees in Abseil.

Segment Trees

The lessons learned from optimizing binary search can be applied to a broad range of data structures.

In this article, instead of trying to optimize something from the STL again, we focus on *segment trees*, the structures that may be unfamiliar to most *normal* programmers and perhaps even most computer science researchers⁴, but that are used very extensively in programming competitions for their speed and simplicity of implementation.

(If you already know the context, jump straight to the last section for the novelty: the *wide segment tree* that works 4 to 12 times faster than the Fenwick tree.)

Dynamic Prefix Sum

Segment trees are cool and can do lots of different things, but in this article, we will focus on their simplest non-trivial application – the *dynamic prefix sum problem*:

```
void add(int k, int x); // react to a[k] += x (zero-based indexing)
int sum(int k);        // return the sum of the first k elements (from 0 to k - 1)
```

As we have to support two types of queries, our optimization problem becomes multi-dimensional, and the optimal solution depends on the distribution of queries. For example, if one type of the queries were extremely rare, we would only optimize for the other, which is relatively easy to do:

- If we only cared about the cost of *updating the array*, we would store it as it is and calculate the sum directly on each `sum` query.
- If we only cared about the cost of *prefix sum queries*, we would keep it ready and re-calculate them entirely from scratch on each update.

⁴Segment trees are rarely mentioned in the theoretical computer science literature because they are relatively novel (invented ~2000), mostly don't do anything that any other binary tree can't do, and *asymptotically* aren't faster – although, in practice, they often win by a lot in terms of speed.

Both of these options perform $O(1)$ work on one query type but $O(n)$ work on the other. When the query frequencies are relatively close, we can trade off some performance on one type of query for increased performance on the other. Segment trees let you do exactly that, achieving the equilibrium of $O(\log n)$ work for both queries.

Segment Tree Structure

The main idea behind segment trees is this:

- calculate the sum of the entire array and write it down somewhere;
- split the array into two halves, calculate the sum on both halves, and also write them down somewhere;
- split these halves into halves, calculate the total of four sums on them, and also write them down;
- ...and so on, until we recursively reach segments of length one.

These computed subsegment sums can be logically represented as a binary tree – which is what we call a *segment tree*:

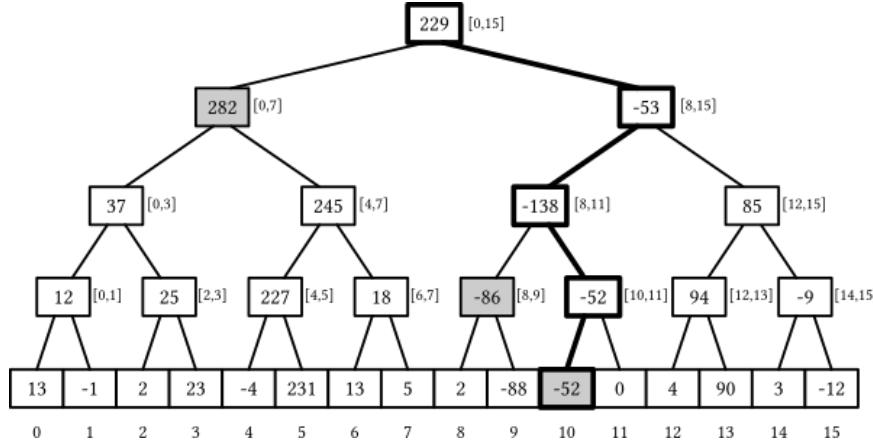


Figure 4: A segment tree with the nodes relevant for the sum(11) and add(10) queries highlighted

Segment trees have some nice properties:

- If the underlying array has n elements, the segment tree has exactly $(2n - 1)$ nodes – n leaves and $(n - 1)$ internal nodes – because each internal node splits a segment in two, and you only need $(n - 1)$ of them to completely split the original $[0, n - 1]$ range.
- The height of the tree is $\Theta(\log n)$: on each next level starting from the root, the number of nodes roughly doubles and the size of their segments roughly halves.
- Each segment can be split into $O(\log n)$ non-intersecting segments that correspond to the nodes of the segment tree: you need at most two from each layer.

When n is not a perfect power of two, not all levels are filled entirely – the last layer may be incomplete – but the truthfulness of these properties remains unaffected. The first property allows us to use only $O(n)$ memory to store the tree, and the last two let us solve the problem in $O(\log n)$ time:

- The `add(k, x)` query can be handled by adding the value x to all nodes whose segments contain the element k , and we've already established that there are only $O(\log n)$ of them.
- The `sum(k)` query can be answered by finding all nodes that collectively compose the $[0, k]$ prefix and summing the values stored in them – and we've also established that there would be at most $O(\log n)$ of them.

But this is still theory. As we'll see later, there are remarkably many ways one can implement this data structure.

Pointer-Based Implementation

The most straightforward way to implement a segment tree is to store everything we need in a node explicitly: including the array segment boundaries, the sum, and the pointers to its children.

If we were at the “Introduction to OOP” class, we would implement a segment tree recursively like this:

```
struct segtree {
    int lb, rb;                                // the range this node is responsible for
    int s = 0;                                    // the sum of elements [lb, rb)
    segtree *l = nullptr, *r = nullptr; // pointers to its children

    segtree(int lb, int rb) : lb(lb), rb(rb) {
        if (lb + 1 < rb) { // if the node is not a leaf, create children
            int m = (lb + rb) / 2;
            l = new segtree(lb, m);
            r = new segtree(m, rb);
        }
    }

    void add(int k, int x) { /* react to a[k] += x */ }
    int sum(int k) { /* compute the sum of the first k elements */ }
};
```

If we needed to build it over an existing array, we would rewrite the body of the constructor like this:

```
if (lb + 1 == rb) {
    s = a[lb]; // the node is a leaf -- its sum is just the element a[lb]
} else {
    int t = (lb + rb) / 2;
    l = new segtree(lb, t);
    r = new segtree(t, rb);
    s = l->s + r->s; // we can use the sums of children that we've just calculated
}
```

The construction time is of no significant interest to us, so to reduce the mental burden, we will just assume that the array is zero-initialized in all future implementations.

Now, to implement `add`, we need to descend down the tree until we reach a leaf node, adding the delta to the `s` fields:

```
void add(int k, int x) {
    s += x;
    if (l != nullptr) { // check whether it is a leaf node
        if (k < l->rb)
            l->add(k, x);
        else
            r->add(k, x);
    }
}
```

To calculate the sum on a segment, we can check if the query covers the current segment fully or doesn’t intersect with it at all – and return the result for this node right away. If neither is the case, we recursively pass the query to the children so that they figure it out themselves:

```
int sum(int lq, int rq) {
    if (rb <= lq && rb <= rq) // if we're fully inside the query, return the sum
        return s;
    if (rq <= lb || lq >= rb) // if we don't intersect with the query, return zero
        return 0;
```

```

    return 0;
return l->sum(lq, rq) + r->sum(lq, rq);
}

```

This function visits a total of $O(\log n)$ nodes because it only spawns children when a segment only partially intersects with the query, and there are at most $O(\log n)$ of such segments.

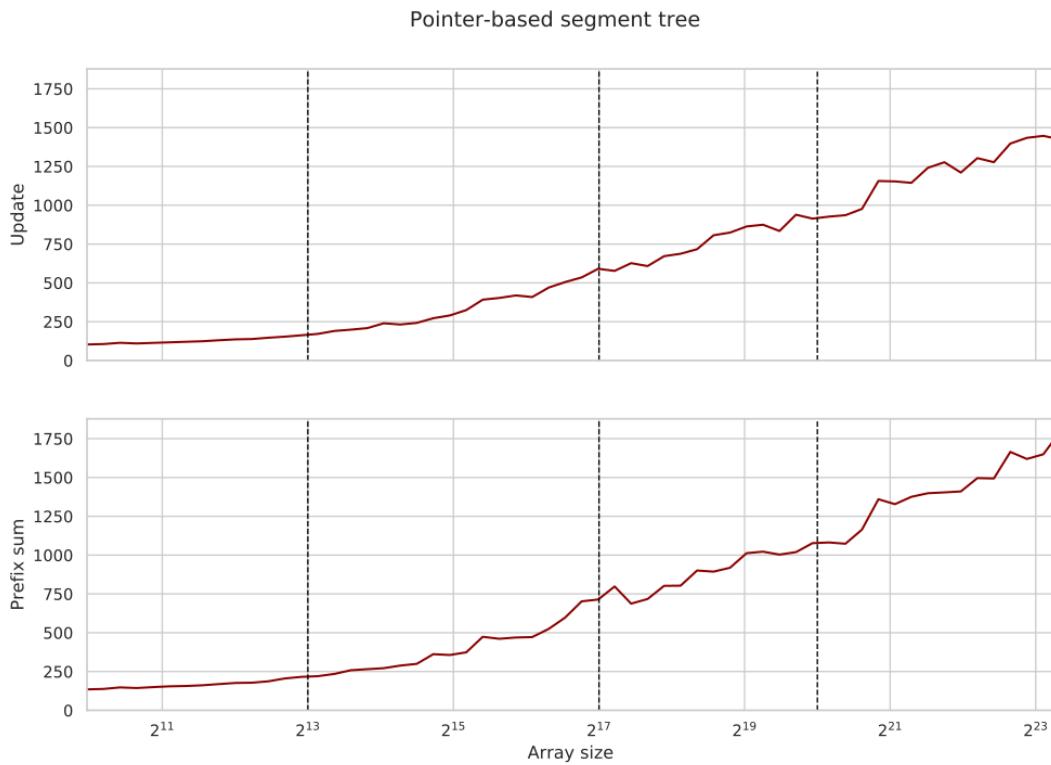
For *prefix sums*, these checks can be simplified as the left border of the query is always zero:

```

int sum(int k) {
    if (rb <= k)
        return s;
    if (lb >= k)
        return 0;
    return l->sum(k) + r->sum(k);
}

```

Since we have two types of queries, we also got two graphs to look at:



While this object-oriented implementation is quite good in terms of software engineering practices, there are several aspects that make it terrible in terms of performance:

- Both query implementations use recursion – although the `add` query can be tail-call optimized.
- Both query implementations use unpredictable branching, which stalls the CPU pipeline.
- The nodes store extra metadata. The structure takes $4 + 4 + 4 + 8 + 8 = 28$ bytes and gets padded to 32 bytes for memory alignment reasons, while only 4 bytes are really necessary to hold the integer sum.
- Most importantly, we are doing a lot of pointer chasing: we have to fetch the pointers to the children to descend into them, even though we can infer, ahead of time, which segments we'll need just from the query.

Pointer chasing outweighs all other issues by orders of magnitude – and to negate it, we need to get rid of pointers, making the structure *implicit*.

Implicit Segment Trees

As a segment tree is a type of binary tree, we can use the Eytzinger layout to store its nodes in one large array and use index arithmetic instead of explicit pointers to navigate it.

More formally, we define node 1 to be the root, holding the sum of the entire array $[0, n)$. Then, for every node v corresponding to the range $[l, r]$, we define:

- the node $2v$ to be its left child corresponding to the range $[l, \lfloor \frac{l+r}{2} \rfloor)$;
- the node $(2v + 1)$ to be its right child corresponding to the range $[\lfloor \frac{l+r}{2} \rfloor, r)$.

When n is a perfect power of two, this layout packs the entire tree very nicely:

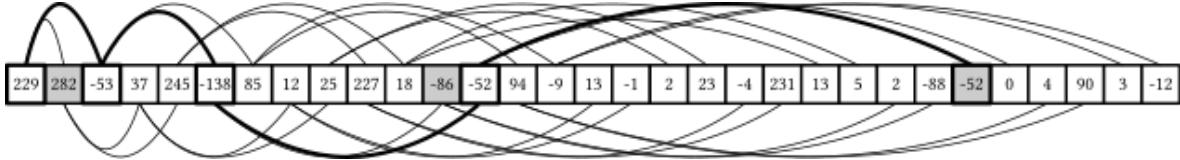


Figure 5: The memory layout of the implicit segment tree with the same query path highlighted

However, when n is not a power of two, the layout stops being compact: although we still have exactly $(2n - 1)$ nodes regardless of how we split segments, they are no longer mapped perfectly to the $[1, 2n)$ range.

For example, consider what happens when we descend to the rightmost leaf in a segment tree of size $17 = 2^4 + 1$:

- we start with the root numbered 1 representing the range $[0, 16]$,
- we go to node $3 = 2 \times 1 + 1$ representing the range $[8, 16]$,
- we go to node $7 = 2 \times 2 + 1$ representing the range $[12, 16]$,
- we go to node $15 = 2 \times 7 + 1$ representing the range $[14, 16]$,
- we go to node $31 = 2 \times 15 + 1$ representing the range $[15, 16]$,
- and we finally reach node $63 = 2 \times 31 + 1$ representing the range $[16, 16]$.

So, as $63 > 2 \times 17 - 1 = 33$, there are some empty spaces in the layout, but the structure of the tree is still the same, and its height is still $O(\log n)$. For now, we can ignore this problem and just allocate a larger array for storing the nodes – it can be shown that the index of the rightmost leaf never exceeds $4n$, so allocating that many cells will always suffice:

```
int t[4 * N]; // contains the node sums
```

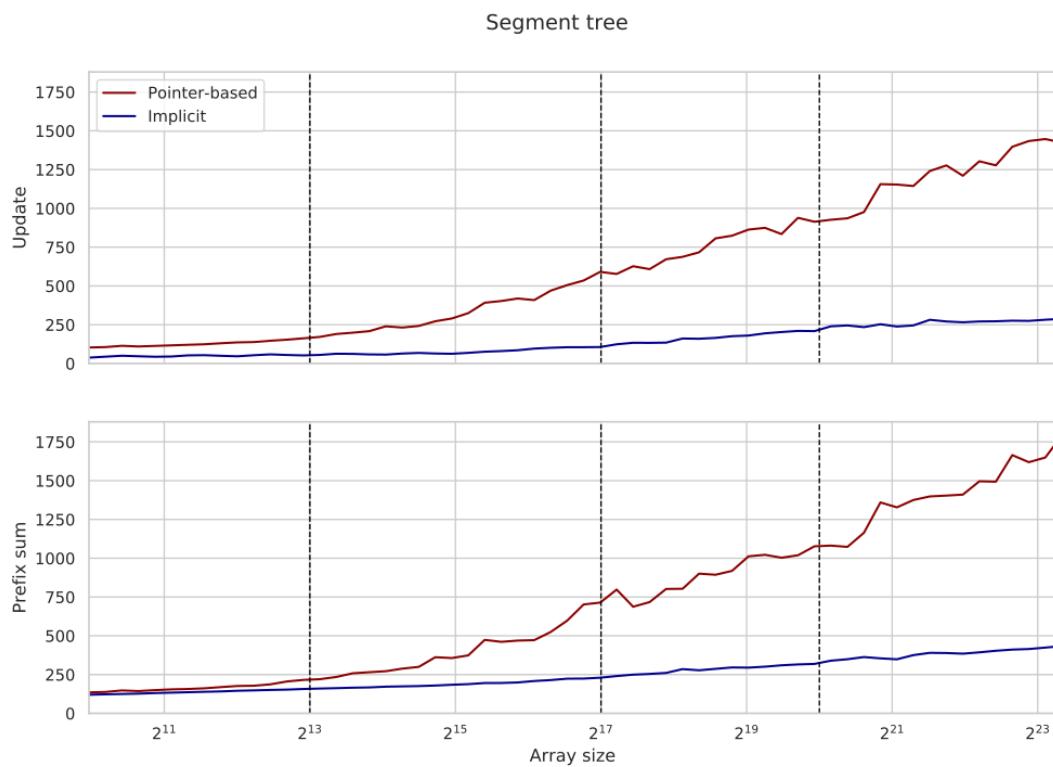
Now, to implement `add`, we create a similar recursive function but using index arithmetic instead of pointers. Since we've also stopped storing the borders of the segment in the nodes, we need to re-calculate them and pass them as parameters for each recursive call:

```
void add(int k, int x, int v = 1, int l = 0, int r = N) {
    t[v] += x;
    if (l + 1 < r) {
        int m = (l + r) / 2;
        if (k < m)
            add(k, x, 2 * v, l, m);
        else
            add(k, x, 2 * v + 1, m, r);
    }
}
```

The implementation of the prefix sum query is largely the same:

```
int sum(int k, int v = 1, int l = 0, int r = N) {
    if (l >= k)
        return 0;
    if (r <= k)
        return t[v];
    int m = (l + r) / 2;
    return sum(k, 2 * v, l, m)
        + sum(k, 2 * v + 1, m, r);
}
```

Passing around five variables in a recursive function seems clumsy, but the performance gains are clearly worth it:



Apart from requiring much less memory, which is good for fitting into the CPU caches, the main advantage of this implementation is that we can now make use of the memory parallelism and fetch the nodes we need in parallel, considerably improving the running time for both queries.

To improve the performance further, we can:

- manually optimize the index arithmetic (e.g., noticing that we need to multiply v by 2 either way),
- replace division by two with an explicit binary shift (because compilers aren't always able to do it themselves),
- and, most importantly, get rid of recursion and make the implementation fully iterative.

As `add` is tail-recursive and has no return value, it is easy turn it into a single `while` loop:

```
void add(int k, int x) {
    int v = 1, l = 0, r = N;
    while (l + 1 < r) {
```

```

        t[v] += x;
        v <= 1;
        int m = (l + r) >> 1;
        if (k < m)
            r = m;
        else
            l = m, v++;
    }
    t[v] += x;
}

```

Doing the same for the `sum` query is slightly harder as it has two recursive calls. The key trick is to notice that when we make these calls, one of them is guaranteed to terminate immediately as `k` can only be in one of the halves, so we can simply check this condition before descending the tree:

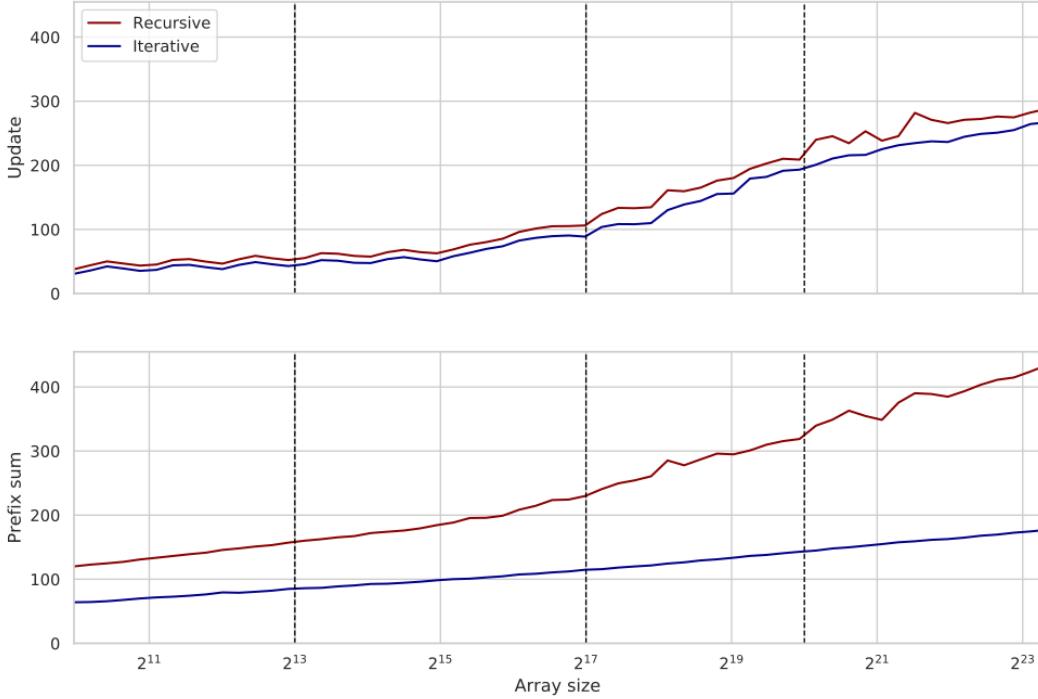
```

int sum(int k) {
    int v = 1, l = 0, r = N, s = 0;
    while (true) {
        int m = (l + r) >> 1;
        v <= 1;
        if (k >= m) {
            s += t[v++];
            if (k == m)
                break;
            l = m;
        } else {
            r = m;
        }
    }
    return s;
}

```

This doesn't improve the performance for the update query by a lot (because it was tail-recursive, and the compiler already performed a similar optimization), but the running time on the prefix sum query has roughly halved for all problem sizes:

Implicit segment tree



This implementation still has some problems: we are using up to twice as much memory as necessary, we have costly branching, and we have to maintain and re-compute array bounds on each iteration. To get rid of these problems, we need to change our approach a little bit.

Bottom-Up Implementation

Let's change the definition of the implicit segment tree layout. Instead of relying on the parent-to-child relationship, we first forcefully assign all the leaf nodes numbers in the $[n, 2n)$ range, and then recursively define the parent of node k to be equal to node $\lfloor \frac{k}{2} \rfloor$.

This structure is largely the same as before: you can still reach the root (node 1) by dividing any node number by two, and each node still has at most two children: $2k$ and $(2k + 1)$, as anything else yields a different parent number when floor-divided by two. The advantage we get is that we've forced the last layer to be contiguous and start from n , so we can use the array of half the size:

```
int t[2 * N];
```

When n is a power of two, the structure of the tree is exactly the same as before and when implementing the queries, we can take advantage of this bottom-up approach and start from the k -th leaf node (simply indexed $N + k$) and ascend the tree until we reach the root:

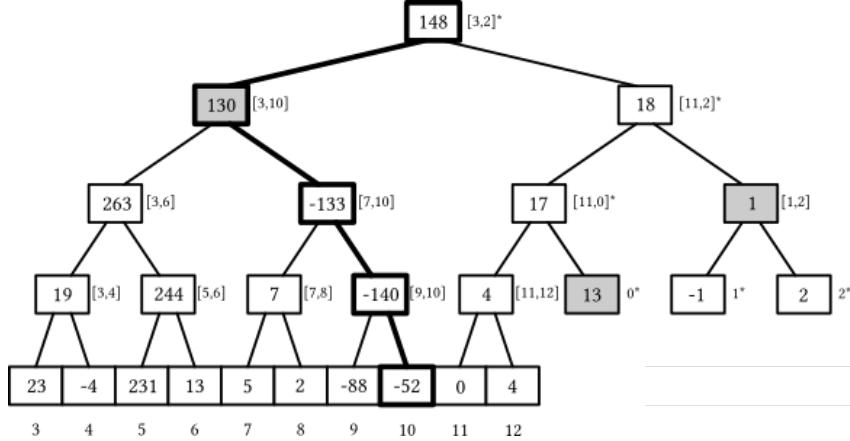
```
void add(int k, int x) {
    k += N;
    while (k != 0) {
        t[k] += x;
        k >>= 1;
    }
}
```

To calculate the sum on the $[l, r)$ subsegment, we can maintain pointers to the first and the last element that

needs to be added, increase/decrease them respectively when we add a node and stop after they converge to the same node (which would be their least common ancestor):

```
int sum(int l, int r) {
    l += N;
    r += N - 1;
    int s = 0;
    while (l <= r) {
        if (l & 1) s += t[l++]; // l is a right child: add it and move to a cousin
        if (~r & 1) s += t[r--]; // r is a left child: add it and move to a cousin
        l >>= 1, r >>= 1;
    }
    return s;
}
```

Surprisingly, both queries work correctly even when n is not a power of two. To understand why, consider a 13-element segment tree:



The first index of the last layer is always a power of two, but when the array size is not a perfect power of two, some prefix of the leaf elements gets wrapped around to the right side of the tree. Magically, this fact does not pose a problem for our implementation:

- The `add` query still updates its parent nodes, even though some of them correspond to some prefix and some suffix of the array instead of a contiguous subsegment.
- The `sum` query still computes the sum on the correct subsegment, even when 1 is on that wrapped prefix and logically “to the right” of r because eventually 1 becomes the last node on a layer and gets incremented, suddenly jumping to the first element of the next layer and proceeding normally after adding just the right nodes on the wrapped-around part of the tree (look at the dimmed nodes in the illustration).

Compared to the top-down approach, we use half the memory and don't have to maintain query ranges, which results in simpler and consequently faster code:



When running the benchmarks, we use the `sum(1, r)` procedure for computing a general subsegment sum and just fix `l` equal to 0. To achieve higher performance on the prefix sum query, we want to avoid maintaining `l` and only move the right border like this:

```
int sum(int k) {
    int s = 0;
    k += N - 1;
    while (k != 0) {
        if (~k & 1) // if k is a right child
            s += t[k--];
        k = k >> 1;
    }
    return s;
}
```

In contrast, this prefix sum implementation doesn't work unless n is not a power of two – because k could be on that wrapped-around part, and we'd sum almost the entire array instead of a small prefix.

To make it work for arbitrary array sizes, we can permute the leaves so that they are in the left-to-right logical order in the last two layers of the tree. In the example above, this would mean adding 3 to all leaf indexes and then moving the last three leaves one level higher by subtracting 13.

In the general case, this can be done using predication in a few cycles like this:

```
const int last_layer = 1 << __lg(2 * N - 1);

// calculate the index of the leaf k
int leaf(int k) {
    k += last_layer;
    k -= (k >= 2 * N) * N;
```

```
        return k;  
    }
```

When implementing the queries, all we need to do is to call the `leaf` function to get the correct leaf index:

```

void add(int k, int x) {
    k = leaf(k);
    while (k != 0) {
        t[k] += x;
        k >>= 1;
    }
}

int sum(int k) {
    k = leaf(k - 1);
    int s = 0;
    while (k != 0) {
        if (~k & 1)
            s += t[k--];
        k >>= 1;
    }
    return s;
}

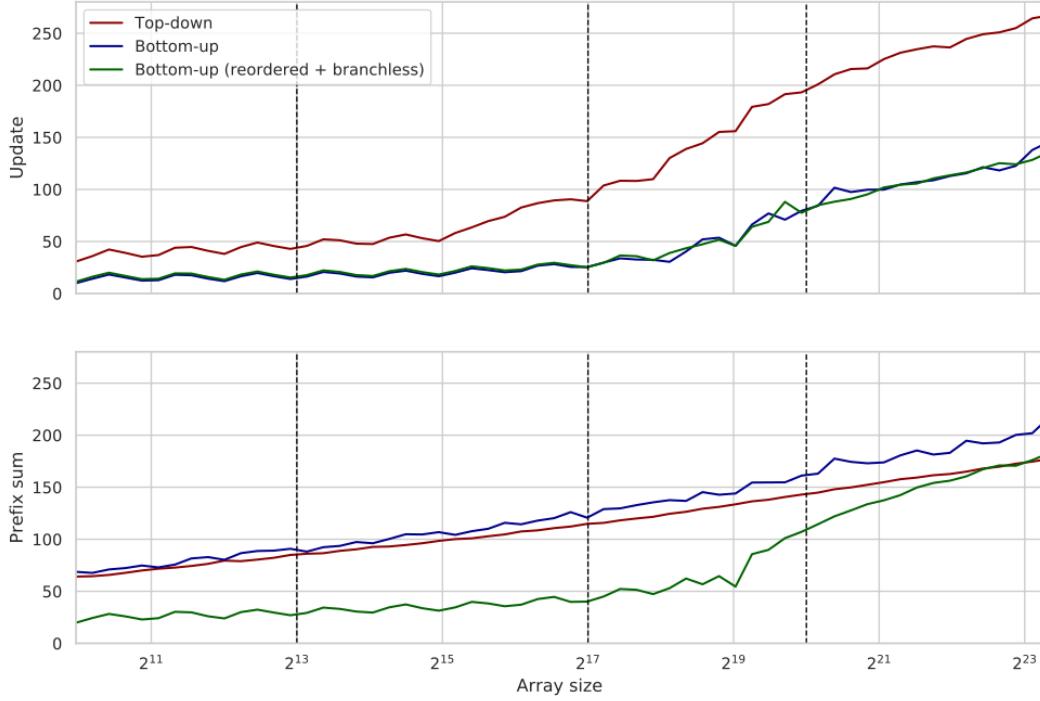
```

The last touch: by replacing the `s += t[k--]` line with predication, we can make the implementation branchless (except for the last branch – we still need to check the loop condition):

```
int sum(int k) {
    k = leaf(k - 1);
    int s = 0;
    while (k != 0) {
        s += (~k & 1) ? t[k] : 0; // will be replaced with a cmov
        k = (k - 1) >> 1;
    }
    return s;
}
```

When combined, these optimizations make the prefix sum queries run much faster:

Segment tree

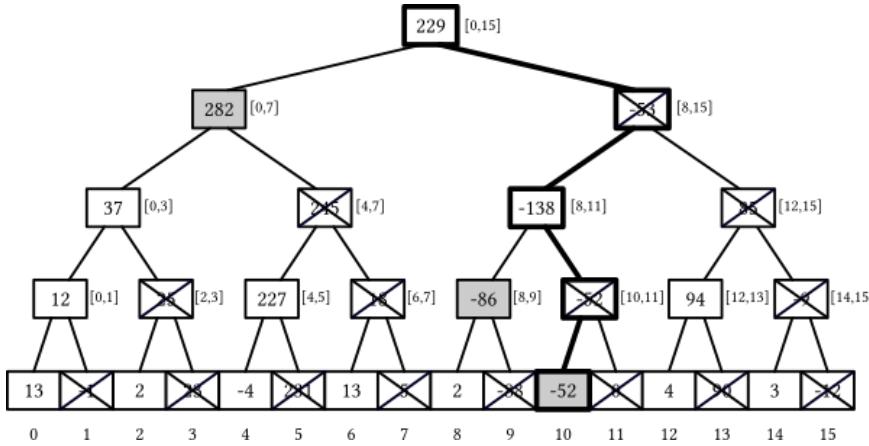


Notice that the bump in the latency for the prefix sum query starts at 2^{19} and not at 2^{20} , the L3 cache boundary. This is because we are still storing $2n$ integers and also fetching the $t[k]$ element regardless of whether we will add it to s or not. We can actually solve both of these problems.

Fenwick trees

Implicit structures are great: they avoid pointer chasing, allow visiting all the relevant nodes in parallel, and take less space as they don't store metadata in nodes. Even better than implicit structures are *succinct* structures: they only require the information-theoretical minimum space to store the structure, using only $O(1)$ additional memory.

To make a segment tree succinct, we need to look at the values stored in the nodes and search for redundancies – the values that can be inferred from others – and remove them. One way to do this is to notice that in every implementation of prefix sum, we've never used the sums stored in right children – therefore, for computing prefix sums, such nodes are redundant:



The Fenwick tree (also called *binary indexed tree* – soon you’ll understand why) is a type of segment tree that uses this consideration and gets rid of all *right* children, essentially removing every second node in each layer and making the total node count the same as the underlying array.

```
int t[N + 1]; // +1 because we use one-based indexing
```

To store these segment sums compactly, the Fenwick tree ditches the Eytzinger layout: instead, in place of every element k that would be a leaf in the last layer of a segment tree, it stores the sum of its first non-removed ancestor. For example:

- the element 7 would hold the sum on the $[0, 7]$ range (282),
- the element 9 would hold the sum on the $[8, 9]$ range (-86),
- the element 10 would hold the sum on the $[10, 10]$ range (-52 , the element itself).

How to compute this range for a given element k (the left boundary, to be more specific: the right boundary is always the element k itself) quicker than simulating the descend down the tree? Turns out, there is a smart bit trick that works when the tree size is a power of two and we use one-based indexing – just remove the least significant bit of the index:

- the left bound for element $7 + 1 = 8 = 1000_2$ is $0000_2 = 0$,
- the left bound for element $9 + 1 = 10 = 1010_2$ is $1000_2 = 8$,
- the left bound for element $10 + 1 = 11 = 1011_2$ is $1010_2 = 10$.

And to get the last set bit of an integer, we can use this procedure:

```
int lowbit(int x) {
    return x & -x;
}
```

This trick works by the virtue of how signed numbers are stored in binary using two’s complement. When we compute $-x$, we implicitly subtract it from a large power of two: some prefix of the number flips, some suffix of zeros at the end remains, and the only one-bit that stays unchanged is the last set bit – which will be the only one surviving $x \& -x$. For example:

$$\begin{aligned} +90 &= 64 + 16 + 8 + 2 = (0)10110 \\ -90 &= 00000 - 10110 = (1)01010 \\ \rightarrow (+90) \& (-90) &= (0)00010 \end{aligned}$$

We’ve established what a Fenwick tree is just an array of size n where each element k is defined to be the sum of elements from $k - \text{lowbit}(k) + 1$ and k inclusive in the original array, and now it’s time to implement some queries.

Implementing the prefix sum query is easy. The $t[k]$ holds the sum we need except for the first $k - \text{lowbit}(k)$ elements, so we can just add it to the result and then jump to $k - \text{lowbit}(k)$ and continue doing this until we reach the beginning of the array:

```

int sum(int k) {
    int s = 0;
    for (; k != 0; k -= lowbit(k))
        s += t[k];
    return s;
}

```

Since we are repeatedly removing the lowest set bit from k , and also since this procedure is equivalent to visiting the same left-child nodes in a segment tree, each `sum` query can touch at most $O(\log n)$ nodes:

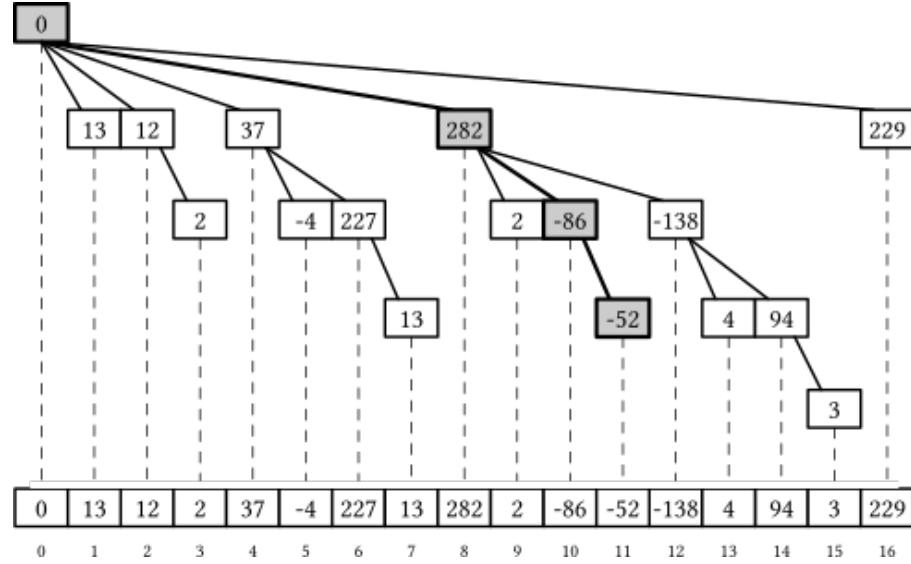


Figure 6: A path for a prefix sum query in a Fenwick tree

To slightly improve the performance of the `sum` query, we use $k \&= k - 1$ to remove the lowest bit in one go, which is one instruction faster than $k = k \& -k$:

```

int sum(int k) {
    int s = 0;
    for (; k != 0; k &= k - 1)
        s += t[k];
    return s;
}

```

Unlike all previous segment tree implementations, a Fenwick tree is a structure where it is easier and more efficient to calculate the sum on a subsegment as the difference of two prefix sums:

```

// [l, r)
int sum (int l, int r) {
    return sum(r) - sum(l);
}

```

The update query is easier to code but less intuitive. We need to add a value x to all nodes that are left-child ancestors of leaf k . Such nodes have indices m larger than k but $m - \text{lowbit}(m) < k$ so that k is included in their ranges.

All such indices need to have a common prefix with k , then a 1 where it was 0 in k , and then a suffix of zeros so that that 1 canceled and the result of $m - \text{lowbit}(m)$ is less than k . All such indices can be generated iteratively like this:

```

void add(int k, int x) {

```

```

for (k += 1; k <= N; k += k & -k)
    t[k] += x;
}

```

Repeatedly adding the lowest set bit to k makes it “more even” and lifts it to its next left-child segment tree ancestor:

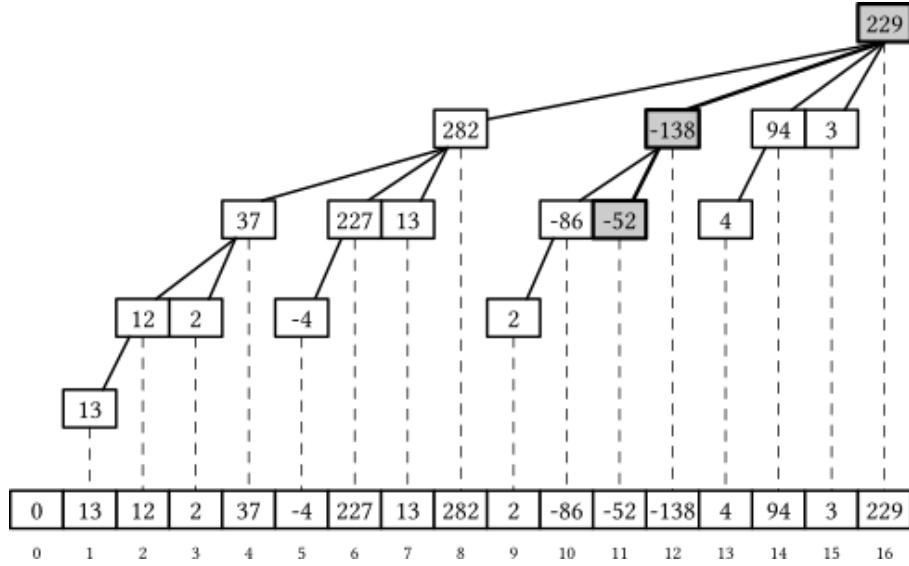
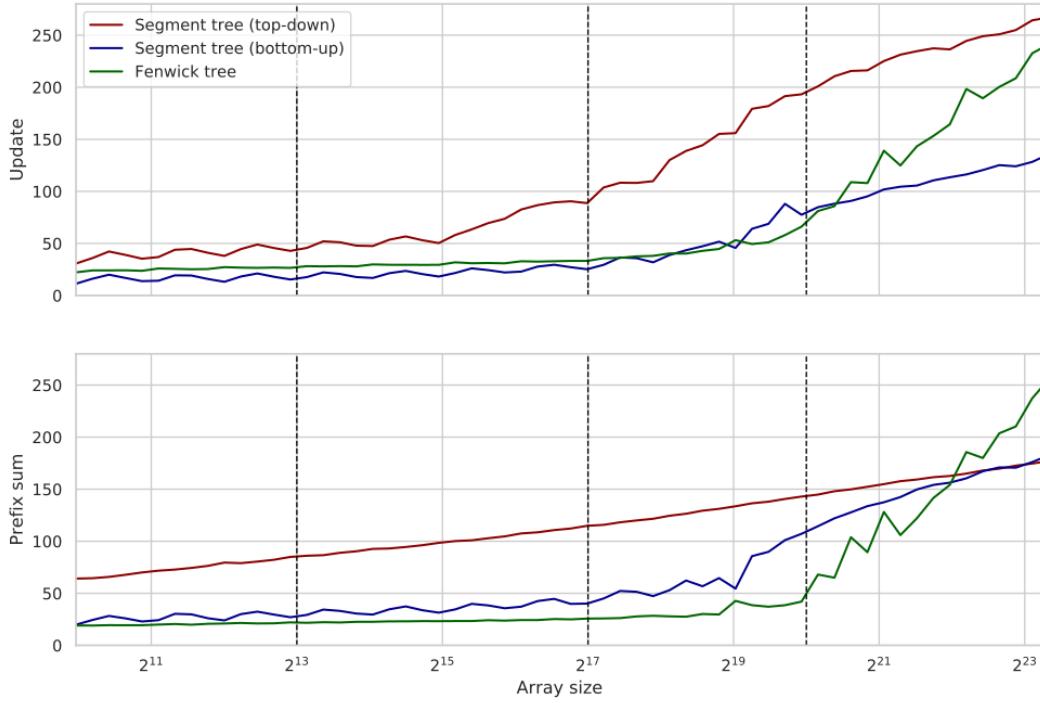


Figure 7: A path for an update query in a Fenwick tree

Now, if we leave all the code as it is, it works correctly even when n is not a power of two. In this case, the Fenwick tree is not equivalent to a segment tree of size n but to a *forest* of up to $O(\log n)$ segment trees of power-of-two sizes – or to a single segment tree padded with zeros to a large power of two, if you like to think this way. In either case, all procedures still work correctly as they never touch anything outside the $[1, n]$ range.

The performance of the Fenwick tree is similar to the optimized bottom-up segment tree for the update queries and slightly faster for the prefix sum queries:



There is one weird thing on the graph. After we cross the L3 cache boundary, the performance takes off very rapidly. This is a cache associativity effect: the most frequently used cells all have their indices divisible by large powers of two, so they get aliased to the same cache set, kicking each other out and effectively reducing the cache size.

One way to negate this effect is to insert “holes” in the layout like this:

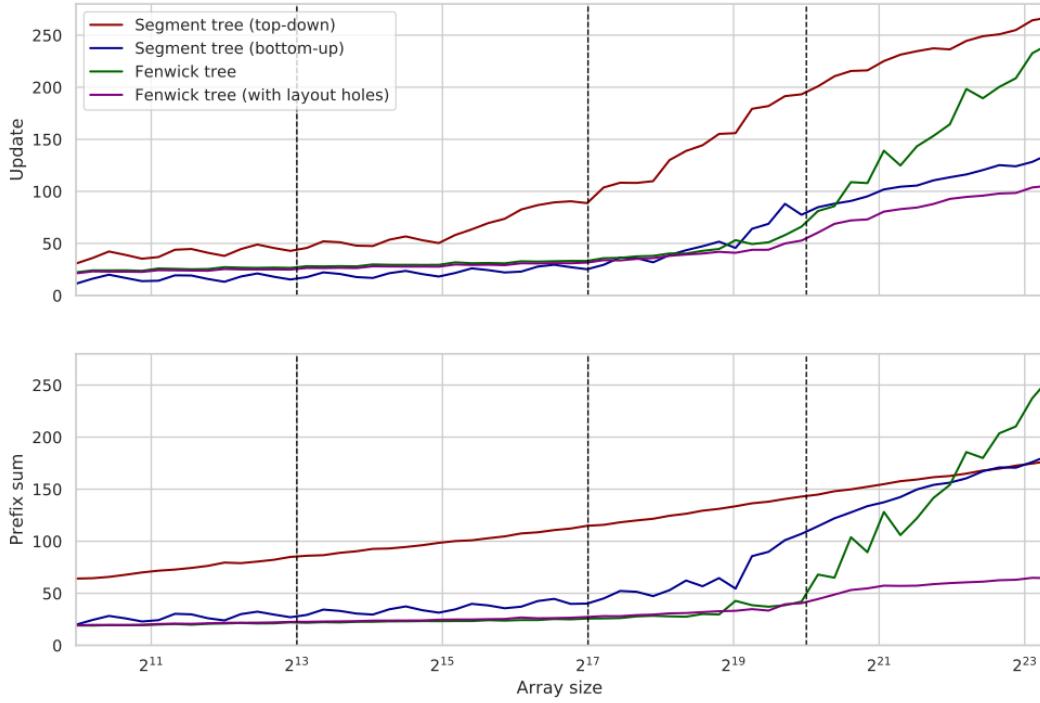
```
inline constexpr int hole(int k) {
    return k + (k >> 10);
}

int t[hole(N) + 1];

void add(int k, int x) {
    for (k += 1; k <= N; k += k & -k)
        t[hole(k)] += x;
}

int sum(int k) {
    int res = 0;
    for (; k != 0; k &= k - 1)
        res += t[hole(k)];
    return res;
}
```

Computing the `hole` function is not on the critical path between iterations, so it does not introduce any significant overhead but completely removes the cache associativity problem and shrinks the latency by up to 3x on large arrays:

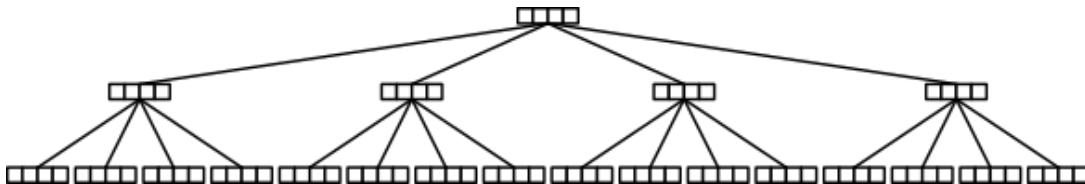


Fenwick trees are fast, but there are still other minor issues with them. Similar to binary search, the temporal locality of their memory accesses is not the greatest, as rarely accessed elements are grouped with the most frequently accessed ones. Fenwick trees also execute a non-constant number of iterations and have to perform end-of-loop checks, very likely causing a branch misprediction – although just a single one.

There are probably still some things to optimize, but we are going to leave it there and focus on an entirely different approach, and if you know S-trees, you probably already know where this is headed.

Wide Segment Trees

Here is the main idea: if the memory system is fetching a full cache line for us anyway, let's fill it to the maximum with information that lets us process the query quicker. For segment trees, this means storing more than one data point in a node. This lets us reduce the tree height and perform fewer iterations when descending or ascending it:



We will use the term *wide (B-ary) segment tree* to refer to this modification.

To implement this layout, we can use a similar constexpr-based approach we used in S+ trees:

```
const int b = 4, B = (1 << b); // cache line size (in integers, not bytes)
```

```
// the height of the tree over an n-element array
constexpr int height(int n) {
```

```

    return (n <= B ? 1 : height(n / B) + 1);
}

// where the h-th layer starts
constexpr int offset(int h) {
    int s = 0, n = N;
    while (h--) {
        n = (n + B - 1) / B;
        s += n * B;
    }
    return s;
}

constexpr int H = height(N);
alignas(64) int t[offset(H)]; // an array for storing nodes

```

This way, we effectively reduce the height of the tree by approximately $\frac{\log_B n}{\log_2 n} = \log_2 B$ times (~ 4 times if $B = 16$), but it becomes non-trivial to implement in-node operations efficiently. For our problem, we have two main options:

1. We could store B *sums* in each node (for each of its B children).
2. We could store B *prefix sums* in each node (the i -th being the sum of the first $(i + 1)$ children).

If we go with the first option, the `add` query would be largely the same as in the bottom-up segment tree, but the `sum` query would need to add up to B scalars in each node it visits. And if we go with the second option, the `sum` query would be trivial, but the `add` query would need to add `x` to some suffix on each node it visits.

In either case, one operation would perform $O(\log_B n)$ operations, touching just one scalar in each node, while the other would perform $O(B \cdot \log_B n)$ operations, touching up to B scalars in each node. We can, however, use SIMD to accelerate the slower operation, and since there are no fast horizontal reductions in SIMD instruction sets, but it is easy to add a vector to a vector, we will choose the second approach and store prefix sums in each node.

This makes the `sum` query extremely fast and easy to implement:

```

int sum(int k) {
    int s = 0;
    for (int h = 0; h < H; h++)
        s += t[offset(h) + (k >> (h * b))];
    return s;
}

```

The `add` query is more complicated and slower. We need to add a number only to a suffix of a node, and we can do this by masking out the positions that should not be modified.

We can pre-calculate a $B \times B$ array corresponding to B such masks that tell, for each of B positions within a node, whether a certain prefix sum value needs to be updated or not:

```

struct Precalc {
    alignas(64) int mask[B][B];

    constexpr Precalc() : mask{} {
        for (int k = 0; k < B; k++)
            for (int i = 0; i < B; i++)
                mask[k][i] = (i > k ? -1 : 0);
    }
};

```

```
constexpr Precalc T;
```

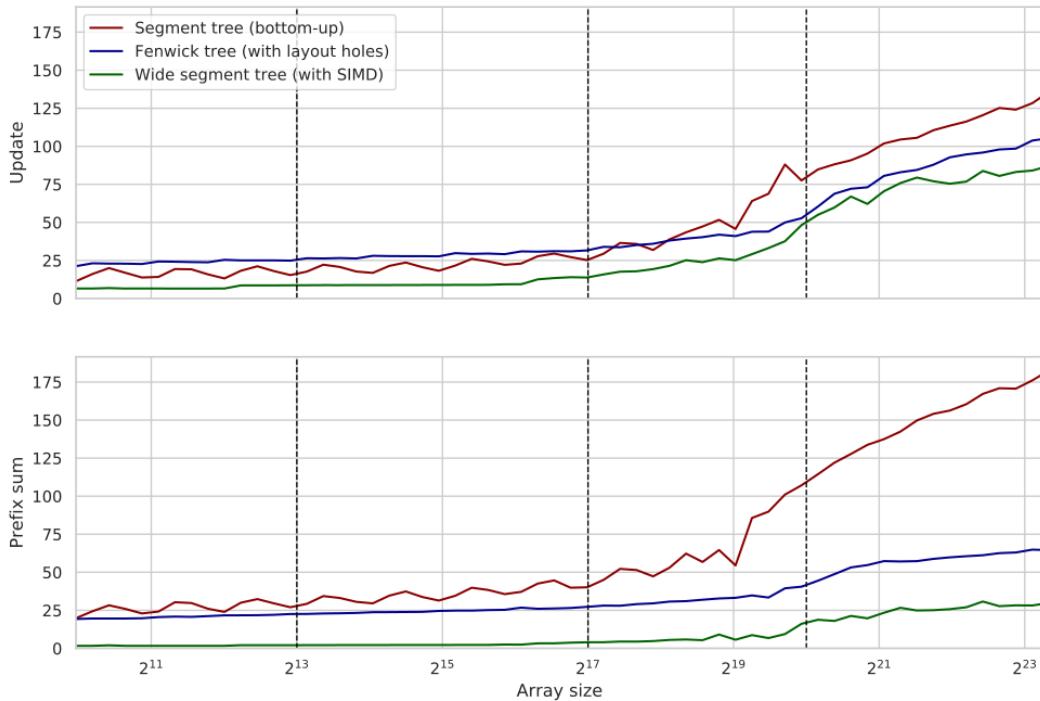
Apart from this masking trick, the rest of the computation is simple enough to be handled with GCC vector types only. When processing the `add` query, we just use these masks to bitwise-and them with the broadcasted `x` value to mask it and then add it to the values stored in the node:

```
typedef int vec __attribute__((vector_size(32)));

constexpr int round(int k) {
    return k & ~(B - 1); // = k / B * B
}

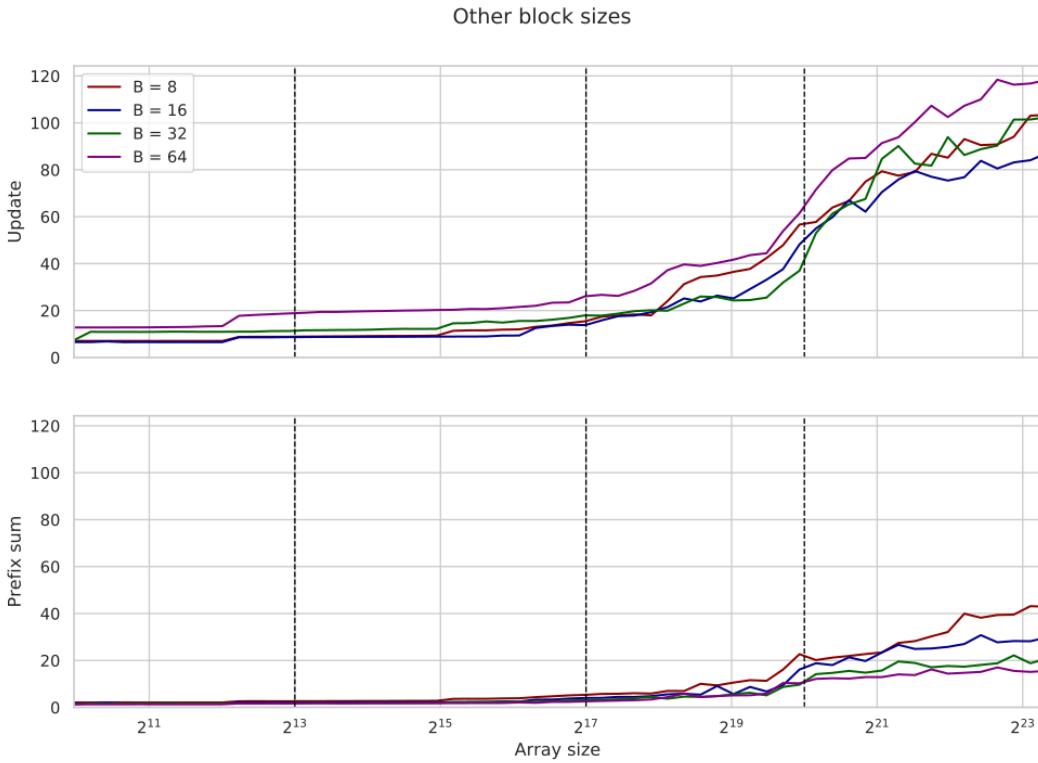
void add(int k, int x) {
    vec v = x + vec{0};
    for (int h = 0; h < H; h++) {
        auto a = (vec*) &t[offset(h)] + round(k);
        auto m = (vec*) T.mask[k % B];
        for (int i = 0; i < B / 8; i++)
            a[i] += v & m[i];
        k >>= b;
    }
}
```

This speeds up the `sum` query by more than 10x and the `add` query by up to 4x compared to the Fenwick tree:



Unlike S-trees, the block size can be easily changed in this implementation (by literally changing one character). Expectedly, when we increase it, the update time also increases as we need to fetch more cache lines

and process them, but the `sum` query time decreases as the height of the tree becomes smaller:

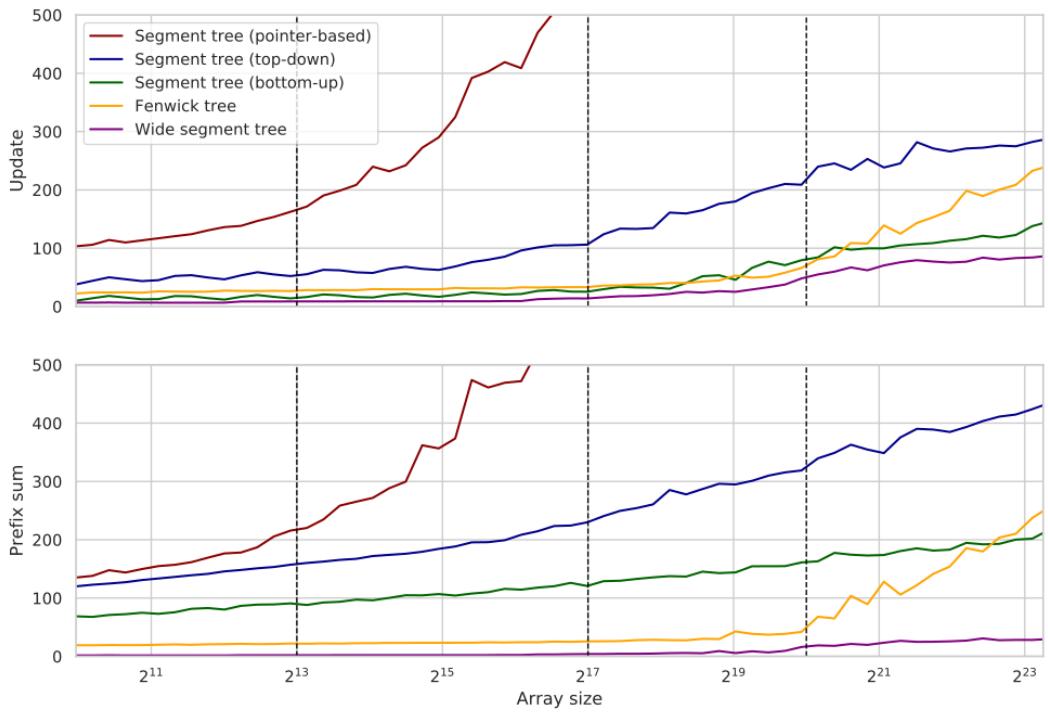


Similar to the S+ trees, the optimal memory layout probably has non-uniform block sizes, depending on the problem size and the distribution of queries, but we are not going to explore this idea and just leave the optimization here.

Comparisons

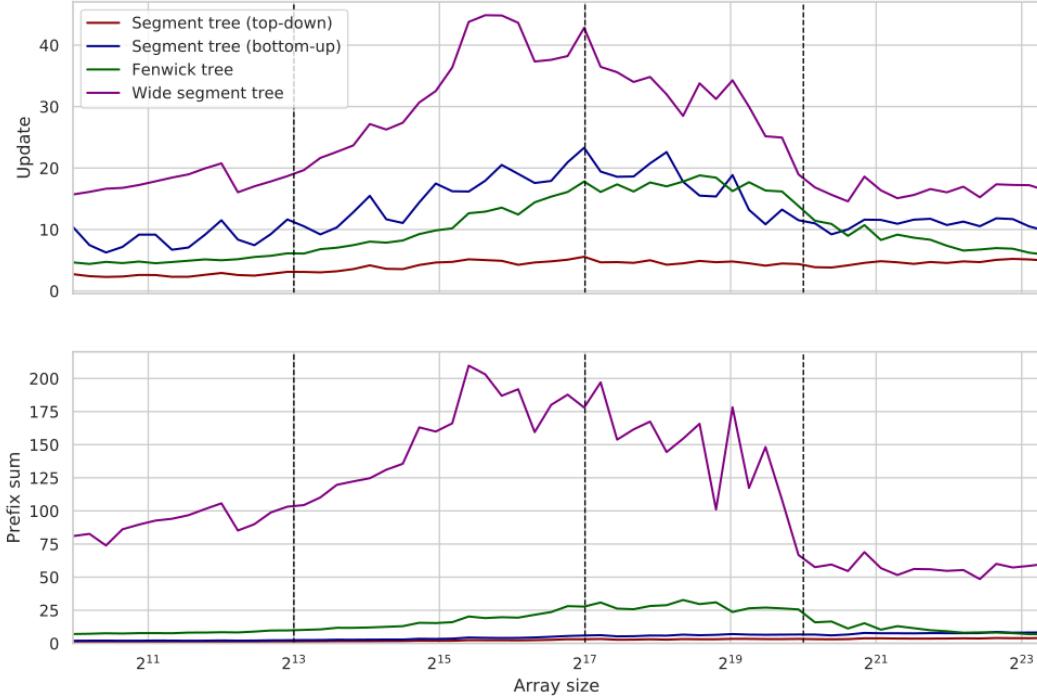
Wide segment trees are significantly faster compared to other popular segment tree implementations:

All implementations (initial, not optimized)



The relative speedup is in the orders of magnitude:

* vs. pointer-based segment tree



Compared to the original pointer-based implementation, the wide segment tree is up to 200 and 40 times faster for the prefix sum and update queries, respectively – although, for sufficiently large arrays, both implementations become purely memory-bound, and this speedup goes down to around 60 and 15 respectively.

Modifications

We have only focused on the prefix sum problem for 32-bit integers – to make this already long article slightly less long and also to make the comparison with the Fenwick tree fair – but wide segment trees can be used for other common range operations, although implementing them efficiently with SIMD requires some creativity.

Disclaimer: I haven't implemented any of these ideas, so some of them may be fatally flawed.

Other data types can be trivially supported by changing the vector type and, if they differ in size, the node size B – which also changes the tree height and hence the total number of iterations for both queries.

It may also be that the queries have different limits on the updates and the prefix sum queries. For example, it is not uncommon to have only “ ± 1 ” update queries with a guarantee that the result of the prefix sum query always fits into a 32-bit integer. If the result could fit into 8 bits, we'd simply use a 8-bit **char** with block size of $B = 64$ bytes, making the total tree height $\frac{\log_{16} n}{\log_{64} n} = \log_{16} 64 = 1.5$ times smaller and both queries proportionally faster.

Unfortunately, that doesn't work in the general case, but we still have a way to speed up queries when the update deltas are small: we can *buffer* the updates queries. Using the same “ ± 1 ” example, we can make the branching factor $B = 64$ as we wanted, and in each node, we store B 32-bit integers, B 8-bit signed chars, and a single 8-bit counter variable that starts at 127 and decrements each time we update a node. Then, when we process the queries in nodes:

- For the update query, we add a vector of masked 8-bit plus-or-minus ones to the **char** array, decrement the counter, and, if it is zero, convert the values in the **char** array to 32-bit integers, add them to the

integer array, set the `char` array to zero, and reset the counter back to 127.

- For the prefix sum query, we visit the same nodes but add *both* `int` and `char` values to the result.

This update accumulation trick lets us increase the performance by up to 1.5x at the cost of using ~25% more memory.

Having a conditional branch in the `add` query and adding the `char` array to the `int` array is rather slow, but since we only have to do it every 127 iterations, it doesn't cost us anything in the amortized sense. The processing time for the `sum` query increases, but not significantly – because it mostly depends on the slowest read rather than the number of iterations.

General range queries can be supported the same way as in the Fenwick tree: just decompose the range $[l, r)$ as the difference of two prefix sums $[0, r)$ and $[0, l)$.

This also works for some operations other than addition (multiplication modulo prime, xor, etc.), although they have to be *reversible*: there should be a way to quickly “cancel” the operation on the left prefix from the final result.

Non-reversible operations can also be supported, although they should still satisfy some other properties:

- They must be *associative*: $(a \circ b) \circ c = a \circ (b \circ c)$.
- They must have an *identity element*: $a \circ e = e \circ a = a$.

(Such algebraic structures are called monoids if you're a snob.)

Unfortunately, the prefix sum trick doesn't work when the operation is not reversible, so we have to switch to option one and store the results of these operations separately for each segment. This requires some significant changes to the queries:

- The update query should replace one scalar at the leaf, perform a horizontal reduction at the leaf node, and then continue upwards, replacing one scalar of its parent and so on.
- The range reduction query should, separately for left and right borders, calculate a vector with vertically reduced values on their paths, combine these two vectors into one, and then reduce it horizontally to return the final answer. Note that we still need to use masking to replace values outside of query with neutral elements, and this time, it probably requires some conditional moves/blending and either $B \times B$ precomputed masks or using two masks to account for both left and right borders of the query.

This makes both queries much slower – especially the reduction – but this should still be faster compared to the bottom-up segment tree.

Minimum is a nice exception where the update query can be made slightly faster if the new value of the element is less than the current one: we can skip the horizontal reduction part and just update $\log_B n$ nodes using a scalar procedure.

This works very fast when we mostly have such updates, which is the case, e.g., for the sparse-graph Dijkstra algorithm when we have more edges than vertices. For this problem, the wide segment tree can serve as an efficient fixed-universe min-heap.

Lazy propagation can be done by storing a separate array for the delayed operations in a node. To propagate the updates, we need to go top to bottom (which can be done by simply reversing the direction of the `for` loop and using `k >> (h * b)` to calculate the `h`-th ancestor), broadcast and reset the delayed operation value stored in the parent of the current node, and apply it to all values stored in the current node with SIMD.

One minor problem is that for some operations, we need to know the lengths of the segments: for example, when we need to support a sum and a mass assignment. It can be solved by either padding the elements so that each segment on a layer is uniform in size, pre-calculating the segment lengths and storing them in the node, or using predication to check for the problematic nodes (there will be at most one on each layer).

Acknowledgements

Many thanks to Giulio Ermanno Pibiri for collaborating on this case study, which is largely based on his 2020 paper “Practical Trade-Offs for the Prefix-Sum Problem” co-authored with Rossano Venturini. I highly recommend reading the original article if you are interested in the details we’ve skipped through here for brevity.

The code and some ideas regarding bottom-up segment trees were adapted from a 2015 blog post “Efficient and easy segment trees” by Oleksandr Bacherikov.

Bitmaps

Hash Tables

Hash Tables

[External image removed]

Chaining

[External image removed]

A lot of linked lists or growable arrays

Open Addressing

[External image removed]

Fixed number of cells and a hash function $f_i(x)$ that decides where to look on i -th step

Implementation with a cyclic array:

```
struct hashmap {
    const int size = (1<<24);
    int a[size] = {-1}, b[size];

    static inline int h(int x) { return (x^179)*7; }

    void add(int x, int y) {
        int k = h(x) % size;
        while (a[k] != -1 && a[k] != x)
            k = (k + 1) % size;
        a[k] = x, b[k] = y;
    }

    int get(int x) {
        for (int k = h(x) % size; a[k] != -1; k = (k + 1) % size)
            if (a[k] == x)
                return b[k];
        return -1;
    }
};
```

Same asymptotic complexity, but 2-3x difference in real speed

[External image removed]

The only downside is that you need to rehash it more often

Probabilistic Filters

bloom filters have the inverse behavior of caches* - bloom filter: miss == definitely not present, hit == probably present - cache: miss == probably not present, hit == definitely present

Chapter 13: Parallel Computing

Algorithms for Modern Hardware

Contents

Threading Runtimes	1
Parallel Computing	1
Moore’s Law	1
Multithreading Hardware	2

Threading Runtimes

Parallel Computing

Classic computer science algorithms aim to minimize *work*: the total sum of operations needed.

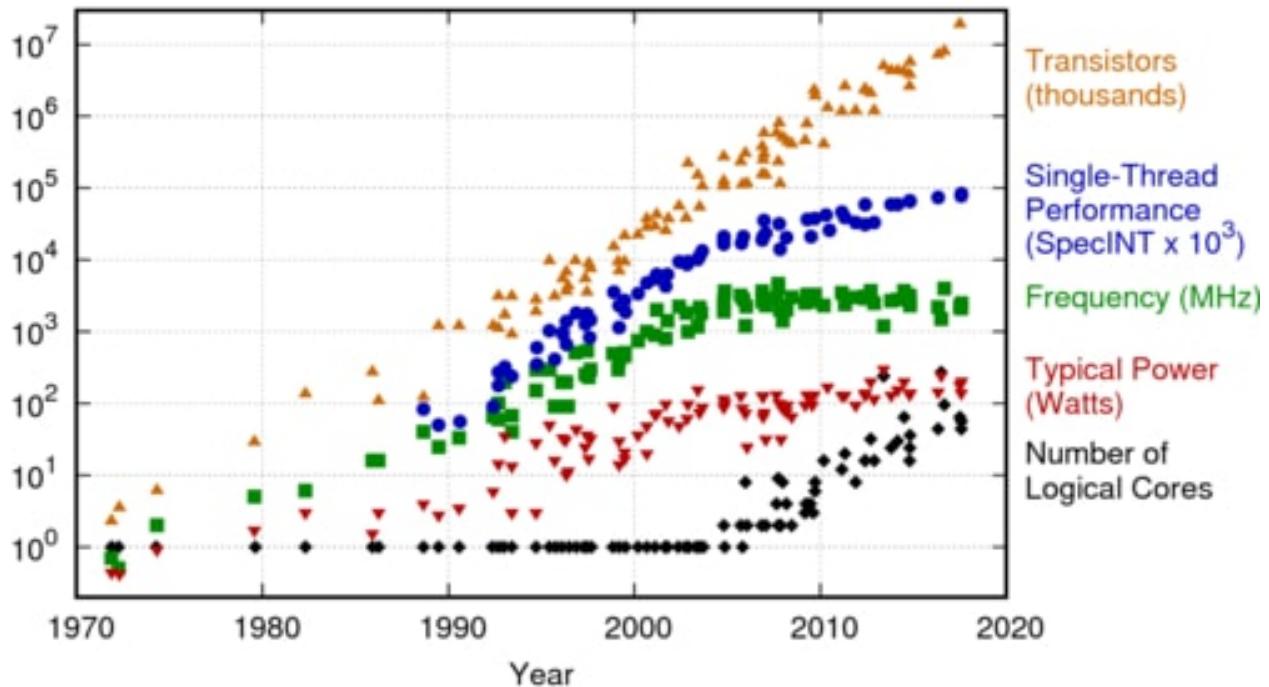
But how many arithmetic operations a computer performed to do a task is not something that people really care about. Moreover, as we previously saw, due to the complex nature of computer hardware, less work doesn’t necessarily mean less running time or less resources used.

What we really care about is the running time and, occasionally, the cost of computation (expressed in CPU-seconds, watts, or straight dollars).

Moore’s Law

Moore’s law is the observation (made by [Gordon Moore](#), the co-founder and former CEO of Intel) that the number of transistors in a microprocessor doubles about every two years. From practical perspective, this roughly means that the performance doubles as well.

42 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Laborde, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Throughout most of computing history, the main thing manufacturers did to improve performance was simply increasing clock frequency. This was possible by the virtue of more precise technology allowing to smaller and smaller chips. However, around mid-2000s they ran into the fundamental problem of removing extra heat from the die. Running CPUs at higher frequencies is not only energy-inefficient, but at some point just physically impossible.

So, in the last 20 years there has been a shift in direction. Instead of trying to execute more cycles per second, let the processor do more useful work per cycle. We have already seen applications of this in practice: pipelining allows some interlapping, and SIMD instructions allow processing data in small batches.

But a much more scalable solution is to put more independent cores on a die that can do independent work and communicate intermediate results to synchronize.

Multithreading Hardware

Hyperthreading

Multiple CPU sockets per motherboard

You can get 400 cores at google

SIMD

GPUs

Chapter 14: Distributed Computing

Algorithms for Modern Hardware

Contents

Actor Model	1
Cloud Computing	1
Distributed Computing	1
Actor Model	
Cloud Computing	
Distributed Computing	