Programming Techniques for Scientific Simulations I: An Introduction to the Hardware of Your PC

A Textbook Chapter Based on Lecture Slides

November 19, 2025

Contents

1 Introduction: Know Your Tools

Welcome to the study of programming for scientific simulations. In this field, our primary goal is often to create programs that are not only *correct* but also *fast*. We want to model complex physical phenomena, process vast amounts of data, and get results in a reasonable amount of time.

It is a common temptation for a new programmer to view the computer as a "black box"—a magical device that takes our human-readable code and produces a result. We write our code in a **high-level language** like C++, Fortran, or Python. These languages are "high-level" because they are abstract and human-friendly, full of concepts we understand like "loops," "objects," and "functions."

However, the computer's "brain," the **Central Processing Unit (CPU)**, understands none of this. The CPU understands only **machine language** (or **machine code**), a stream of raw binary numbers (1s and 0s) that represent the most basic operations possible: "add these two numbers," "move this piece of data," "jump to this other instruction."

This creates a critical gap, which is bridged by a special program called a **compiler**. The compiler's job is to act as an expert translator, converting your abstract, high-level C++ code into the brutally simple, low-level machine code that your specific CPU can execute.

This leads us to the central theme of this chapter: **To write a fast program, you must understand your hardware.**

Why? Because the compiler is not magic. It's a tool, and like any tool, it can be used well or poorly. The "smartness" of its translation is heavily influenced by how you write your code. If you write code that is "hardware-aware"—code that "thinks" like the machine—you empower the compiler to produce a lightning-fast translation. If you write code that is "hardware-agnostic," you may unknowingly force the compiler to produce a slow, inefficient translation.

This week, we will get a detailed introduction to the main hardware components of your computer. We will "open the black box." Next week, armed with this knowledge, we will explore specific "optimization" techniques to make our code faster.

2 The Compilation Pipeline

The journey from your C++ source code to a runnable program is not a single step but a multi-stage "assembly line" known as the **compilation pipeline** or "workflow." Understanding this process helps you diagnose problems and understand *where* and *how* optimization can occur.

Let's walk through the process shown in Figure 1, starting with a simple C++ "Hello, World!" program.

```
1 // hello.cpp
```

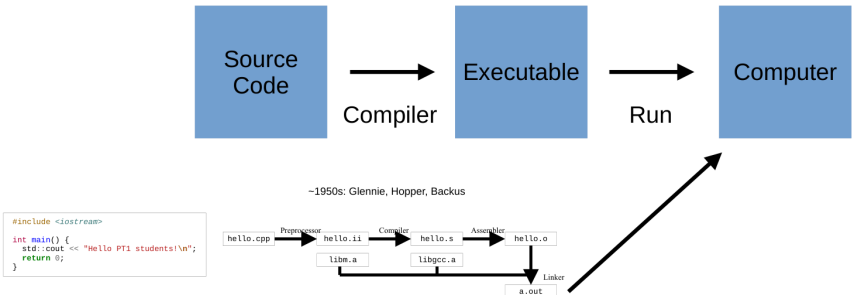

Figure 1: The C++ compilation pipeline. Your human-readable `.cpp` file is transformed through several stages—Preprocessing, Compilation, Assembly, and Linking—to create the final `a.out` executable file that the computer can run.

```cpp
#include <iostream>

int main() {
    std::cout << "Hello PT1 students!\n";
    return 0;
}
```

Listing 1: A simple C++ source file, `hello.cpp`.

2.1 Step 1: The Preprocessor

Your `hello.cpp` file is first given to the **Preprocessor**. The preprocessor is a simple, text-based tool. It does *not* understand C++ logic. It only follows **preprocessor directives**—commands that start with a `#` symbol.

Its main jobs are:

- `#include`: This command finds the file specified (e.g., `<iostream>`) and *literally pastes its entire content* into your file.
- `#define`: This performs a simple "find and replace." For example, `#define PI 3.14` will replace every instance of the text "PI" with "3.14".

The output is a temporary file (e.g., `hello.ii`) that is still C++ code, but now "pre-processed" and often massive from all the included headers.

2.2 Step 2: The Compiler

This is the "smart" part of the process. The **Compiler** takes the "pure" C++ code from the preprocessor (`hello.ii`) and translates it. It parses your C++ logic, checks for syntax errors, and—this is the important part—*optimizes* your code.

It does *not* translate directly to machine code. It translates to an intermediate, low-level (but still human-readable) language called **assembly language**. The output is a file like `hello.s`. We will look at assembly language in detail shortly.

2.3 Step 3: The Assembler

The **Assembler** is the "dumb" translator. Its job is to perform a simple, one-to-one translation of the assembly code (`hello.s`) into pure machine code (1s and 0s). It's not "smart" like the compiler; it just follows a lookup table.

The output is an **object file** (`hello.o`). This file contains the "raw" machine code for your `hello.cpp` file, but it's not yet runnable. It's just a "piece" of a program. It doesn't know where the `iostream` code (which it references) actually *is*.

2.4 Step 4: The Linker

The **Linker** is the final stage. Its job is to be the "bookbinder." It takes all the object files you compiled (like `hello.o`) and any **libraries** you need (like `libm.a` for math functions or the C++ standard library `Libgcc.a` for `iostream`) and "links" them all together.

It resolves all the "I don't know where this is" references. When your `hello.o` says "call the `std::cout` function," the linker finds that function in the library and "stitches" your code to it, creating one, single, complete **executable file** (e.g., `a.out` by default on Linux/macOS, or `hello.exe` on Windows). This is the final program you can run.

3 The Stored-Program Computer Architecture

To understand hardware limitations, we must first understand the basic blueprint of a modern computer. Almost every computer today, from your phone to a supercomputer, is based on the **Stored-Program Computer Architecture**, often called the **Von Neumann Architecture**.

As shown in Figure 2, this design has a few key components:

- **CPU (Central Processing Unit):** The "brain" of the computer. It is responsible for executing instructions. It is internally divided into:
  - **Control Unit:** The "manager" or "conductor." It fetches instructions from Memory, decodes them, and directs the other components to act.
  - **ALU (Arithmetic Logic Unit):** The "calculator." It performs all mathematical (add, subtract, multiply) and logical (and, or, not, is-equal?) operations.

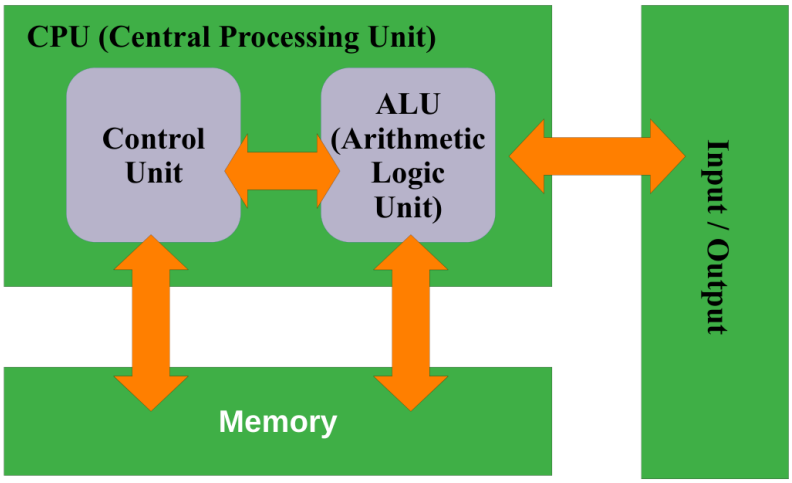

Figure 2: The basic Stored-Program (Von Neumann) Architecture. The CPU, which contains the Control Unit and ALU, is separate from Memory. All instructions and data must travel over the bus between them.

- **Memory (Main Memory, or RAM):** This is the "short-term memory." Crucially, in a stored-program computer, Memory holds *both* the program's instructions *and* the data that program is working on.
- **Input/Output (I/O):** These are all the peripherals that allow the computer to interact with the world: keyboard, mouse, screen, hard drive, network card, etc.

3.1 The Von Neumann Bottleneck

This design has a fundamental flaw, identified by John von Neumann himself. The CPU is incredibly fast, capable of billions of operations per second. The Main Memory, by comparison, is *very* slow.

The "bus" (the pathway in Figure 2) that connects the CPU and Memory is a "narrow bridge." The CPU is constantly "starving," waiting for the slow-moving "trucks" of instructions and data to arrive from Memory.

This traffic jam, shown in Figure 3, is called the **Von Neumann Bottleneck**. It is the single biggest problem in high-performance computing. Almost all the complex, clever hardware we are about to discuss—caches, pipelines, branch predictors—are just sophisticated tricks to *hide* this bottleneck.

4 The CPU's "Language": Instruction Sets

The **Instruction Set Architecture (ISA)** is the "vocabulary" of a CPU. It is the complete, low-level list of all the basic, simple commands that the CPU hardware

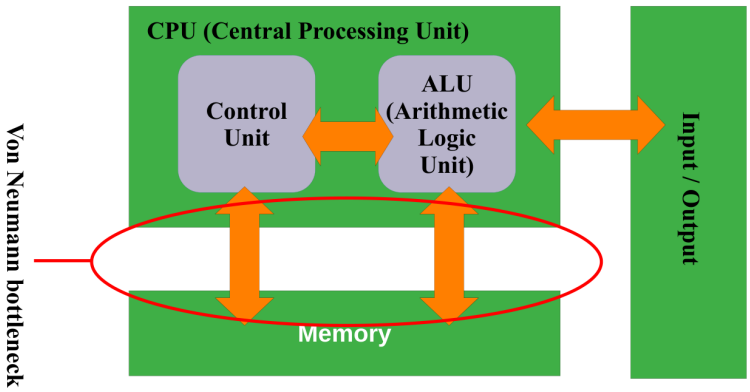

Figure 3: The Von Neumann Bottleneck. The single, shared, and relatively slow bus between the high-speed CPU and the lower-speed Memory creates a traffic jam that limits the entire system's performance.

knows how to execute. Every complex program you run—a video game, a web browser, a scientific simulation—is just a combination of *billions* of these incredibly simple, tiny instructions.

To understand this, let's look at a historical example. The EDSAC computer from 1949 had a tiny "Order Code" of only 17 instructions.

EDSAC Order Code

Order	Explanation
A n	Add the number in storage location *n* into the accumulator.
S n	Subtract the number in storage location *n* from the accumulator.
H n	Transfer the number in storage location *n* into the multiplier register.
V n	Multiply the number in storage location *n* by the number in the multiplier register and add into the accumulator.
N n	Multiply the number in storage location *n* by the number in the multiplier register and subtract from the contents of the accumulator.
T n	Transfer the contents of the accumulator to storage location *n*, and clear the accumulator.
U n	Transfer the contents of the accumulator to storage location *n*, and do not clear the accumulator.
C n	Collate the number in storage location *n* with the number in the multiplier register, i.e., add a 1 into the accumulator in digital positions where both numbers have a 1 and a 0 in other digital positions.
R 2^{n-2}	Shift the number in the accumulator *n* places to the right, i.e., multiply it by 2^{-n}.
L 2^{n-2}	Shift the number in the accumulator *n* places to the left, i.e., multiply it by 2^{n}.
E n	If the number in the accumulator is greater than or equal to zero, execute next the order which stands in storage location *n*; otherwise, proceed serially.
G n	If the number in the accumulator is less than zero, execute next the order which stands in storage location *n*; otherwise, proceed serially.
I n	Read the next row of holes on the tape, and place the resulting 5 digits in the least significant places of storage location *n*.
O n	Print the character now set up on the teleprinter, and set up on the teleprinter the character represented by the five most significant digits in storage location *n*.
F n	Place the five digits which represent the character next to be printed by the teleprinter in the five most significant places of storage location *n*.
Y	Round off the number in the accumulator to 34 binary digits.
Z	Stop the machine, and ring the warning bell.

Figure 4: The "Order Code" (Instruction Set) for the 1949 EDSAC computer. It shows how simple these base operations are: `A` for Add, `S` for Subtract, `T` for Transfer (store) to memory.

As seen in Figure 4, the instructions are basic:

- `An`: **A**dd the number from memory location `n` to a special holding spot in

the CPU (the "accumulator").

- `Sn`: **S**ubtract the number from memory location `n` from the accumulator.
- `Tn`: **T**ransfer (store) the value *from* the accumulator *to* memory location `n`.

This shows the fundamental "Load-Operate-Store" model that all computers still use.

4.1 Machine Code vs. Assembly Language

These instructions are processed by the CPU in two forms:

1. **Machine Code:** This is the "native language" of the CPU. It's "just numbers"—pure binary. For example, the instruction to "add" might be represented as `1000101111000011`. This is impossible for humans to read but is what the hardware actually executes. It is **non-portable**: machine code for an Intel CPU is meaningless to an ARM CPU (in your phone).
2. **Assembly Language:** This is a **one-to-one translation** of machine code into a human-readable text format. The binary code `1000101111000011` might be written in assembly as `add rax, rbx`. It's still low-level and **non-portable**, but it allows programmers (and compilers) to work with the hardware directly.

The "Assembler" (from our pipeline) is just the simple tool that does this one-to-one translation between `hello.s` and `hello.o`.

4.2 Two Philosophies: CISC vs. RISC

When designing an ISA, there are two main philosophies:

CISC (Complex Instruction Set Computer)

- **Philosophy:** Make the hardware "smart." Include powerful, **complex** instructions directly in the hardware. For example, a single instruction might calculate a trigonometric `sin()` or `cos()` function.
- **Analogy:** A fancy scientific calculator with hundreds of buttons, including one for `sin()`.
- **Pros:** Can make assembly programming "easier," as one instruction does a lot of work.
- **Cons:** The CPU hardware becomes extremely complex, power-hungry, and difficult to design.
- **Example:** Intel/AMD x86 processors in your laptop/desktop.

RISC (Reduced Instruction Set Computer)

- **Philosophy:** Make the hardware "dumb" but fast. The ISA should only contain a **reduced** set of very simple, very fast, low-level instructions (like `load`, `add`, `store`).
- **Analogy:** A simple 4-function calculator. To calculate `sin()`, you must perform a Taylor series expansion—a *long sequence* of simple adds and multiplies.
- **Pros:** Hardware is simple, small, and power-efficient. Instructions are fast and easy to "pipeline" (see later).
- **Cons:** Puts the burden on the *compiler* to be "smart" and combine many simple instructions to achieve a complex task.
- **Example:** ARM processors (in all phones, tablets, and Apple M-series chips), IBM Power.

In reality, the lines are now blurry. Modern CISC chips from Intel often translate their "complex" instructions into simple, RISC-like "micro-operations" internally. But the two design philosophies are still fundamental.

5 Solving the Bottleneck: A Modern CPU

If the Von Neumann bottleneck (slow memory) is the main problem, how do we solve it? The answer is to *hide* the latency by building a **Memory Hierarchy**.

5.1 Caches: A "Mini-Memory" for the CPU

The core idea is simple: if Main Memory is slow (like a library across town), we can place a small, *very fast* "mini-memory" right next to the CPU. This is called a **cache**.

This creates a "memory hierarchy," which works on an "access" principle:

- **Analogy:** Your "desk" (L1 Cache) is tiny but holds the papers you're *currently* working on. Your "bookshelf" (L2 Cache) is bigger but slower; it holds your most-used books. The "library" (Main Memory) is huge but very slow to access.
- **L1 Cache:** (Level 1) Tiny (e.g., 64 KB) but *extremely* fast (a few CPU cycles). It is often split into an **L1 instruction cache** (L1i) and an **L1 data cache** (L1d).
- **L2 Cache:** (Level 2) Bigger (e.g., 256 KB) and a bit slower, but still very fast.

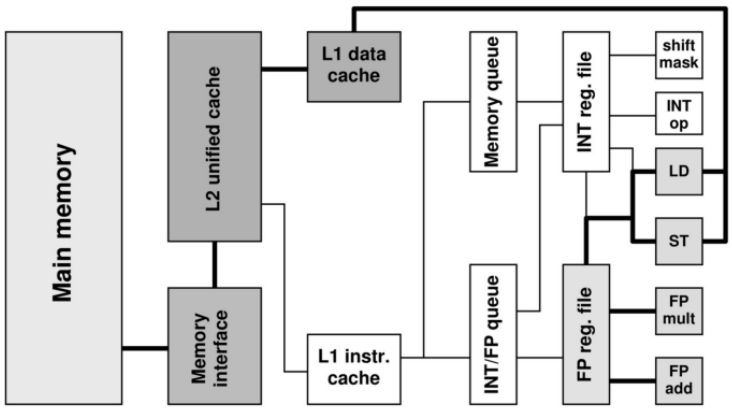

Figure 5: A simplified diagram of a cache-based microprocessor. Instead of going directly to slow Main Memory, the CPU core first checks the extremely fast L1 and L2 caches.

- **L3 Cache:** (Level 3) Even bigger (e.g., 8-32 MB) and slower, often shared by all CPU cores.

When the CPU needs a piece of data, it first asks the L1 cache. If it's there (a **cache hit**), it's a super-fast operation. If it's not (a **cache miss**), it asks the L2 cache. If it misses there, it asks the L3. Only if it's not in *any* cache (a "cold miss") does the CPU make the "long, slow" trip to Main Memory.

5.2 Key Components of a Modern Core

If we "zoom in" on a single, modern CPU core, we see it's far more complex than just a "Control Unit + ALU." It's a vast city of specialized hardware, all designed to hide latency and process instructions as fast as possible.

While Figure 6 is intimidating, we can group this logic into a few key jobs:

- **Fetch and Decode Unit:** This is the "manager." It *fetches* instructions from the L1 instruction cache, *decodes* them (figures out what they are), and *dispatches* them to the correct execution units.
- **Registers:** This is the single most important concept for a programmer to understand. Registers are *not* the L1 cache. They are tiny, named storage spots *directly inside* the ALU and Control Unit. They are the *fastest storage in the entire computer* (sub-cycle access).
- **The Load-Store Model:** The ALU (the "calculator") **cannot** operate on data in memory. It can *only* operate on data currently sitting in a register.

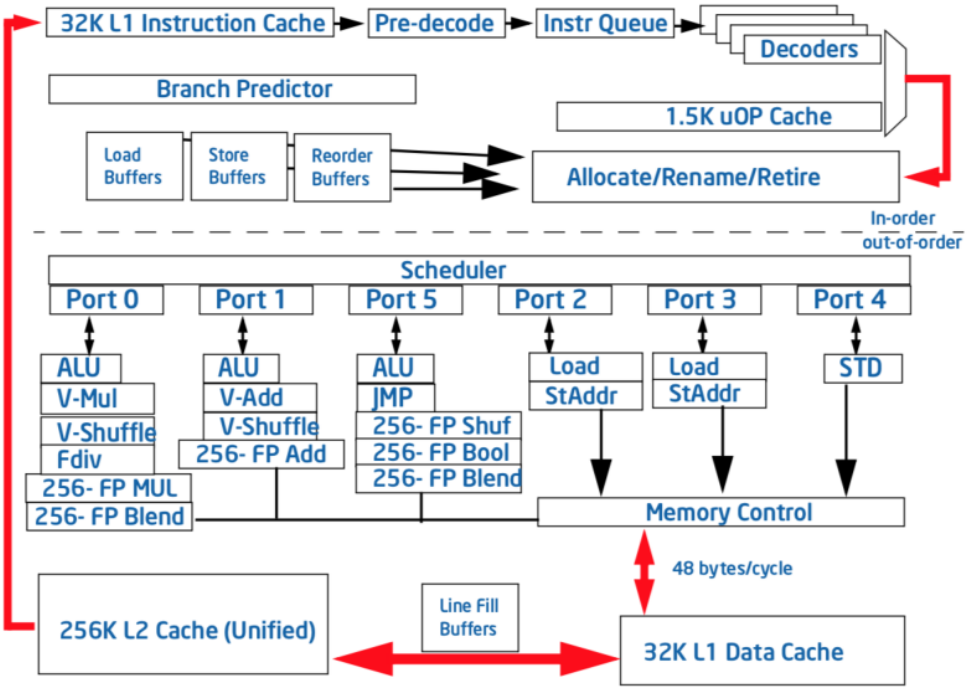

Figure 6: A diagram of a single Intel Sandy Bridge CPU core. The complexity (Schedulers, Decoders, Branch Predictors) is all dedicated to processing instructions efficiently and keeping the "Execution Units" (like the ALUs) busy.

  - **Analogy:** The ALU is your calculator. Registers are your *fingers* on the calculator's number pads. Memory/Cache is the *piece of paper* with the numbers on it.
  - To add `A+B`, the CPU must execute *multiple* instructions: 1. `LOAD` A from memory into Register 1. (Read paper, type 'A' into calculator). 2. `LOAD` B from memory into Register 2. (Read paper, type 'B' into calculator). 3. `ADD` Register 1 and Register 2, put result in Register 3. (Press '+' button). 4. `STORE` result from Register 3 back to memory. (Write calculator result onto paper).

- **Execution Units (ALUs, etc.):** These are the "workers" that do the math. Modern CPUs have many of them, including special units for "Floating Point" (decimal) math, to do multiple calculations at once.

6 From C++ to Assembly: A Practical Look

Let's see how our C++ code *actually* translates into this Load-Store model. You can (and should!) ask your compiler to show you the assembly it generates. With the `g++` compiler, you use the `-S` flag:

`g++ -O2 -S simpleadd.cpp` *(The `-O2` enables optimization)*

This will produce a file named `simpleadd.s`. A great online tool to see this in real-time is the "Compiler Explorer" at `godbolt.org`.

6.1 Example 1: Simple Addition

Here is a very simple C++ function.

```cpp
// simpleadd.cpp
double add_func(double a, double b) {
    return a + b;
}
```

Listing 2: A simple function, `simpleadd.cpp`.

When compiled (with optimizations), the assembly for `add_func` will look conceptually like this (this is Intel x86-64 assembly):

```
add_func:
    ; On entry, 'a' is in register xmm0
    ; On entry, 'b' is in register xmm1

    addsd   xmm0, xmm1    ; Add xmm1 to xmm0, store result in xmm0

    ret                   ; Return (result is in xmm0)
```

Listing 3: Conceptual assembly for `simpleadd.cpp`.

Notice: there is no "load" or "store"! The compiler is smart enough to pass the variables `a` and `b` *in registers* (`xmm0`, `xmm1`) and return the result *in a register* (`xmm0`). This is extremely fast.

6.2 Example 2: A Loop

What about a loop?

```cpp
// loopadd.cpp
void loop_add(double* A, double* B, int n) {
    for (int i = 0; i < n; ++i) {
        A[i] = A[i] + B[i];
    }
}
```

Listing 4: Adding elements of two arrays, `loopadd.cpp`.

The conceptual assembly for this is much more complex:

```
loop_add:
    ; On entry: A is in rdi, B is in rsi, n is in edx
    xor     eax, eax        ; eax = 0 (this is 'i')

.L3:                        ; This is the 'label' for the loop
    ; --- Start of loop body ---
    movsd   xmm0, [rdi + rax*8] ; LOAD A[i] into xmm0
    movsd   xmm1, [rsi + rax*8] ; LOAD B[i] into xmm1

    addsd   xmm0, xmm1          ; ADD: xmm0 = xmm0 + xmm1

    movsd   [rdi + rax*8], xmm0 ; STORE result back into A[i]
    ; --- End of loop body ---

    inc     rax             ; i++
    cmp     rax, rdx        ; Compare 'i' with 'n'
    jl      .L3             ; JUMP if LESS to .L3 (the loop label)

    ret                     ; Return
```

Listing 5: Conceptual assembly for `loopadd.cpp`.

Here, we see the full Load-Operate-Store model, plus the **jump** instruction (`jl`) that creates the loop.

6.3 Compiler Optimization: Loop Unrolling

A "smart" compiler might look at that `loopadd.s` assembly and see a problem. The `cmp` (compare) and `jl` (jump) instructions take time. To avoid this overhead, the compiler can "unroll" the loop. If it knows `n=4`, instead of writing a loop, it might just "paste" the loop body four times:

```
; --- Unrolled loop for n=4 ---
movsd   xmm0, [rdi]         ; A[0]
movsd   xmm1, [rsi]         ; B[0]
addsd   xmm0, xmm1
movsd   [rdi], xmm0         ; Store A[0]

movsd   xmm0, [rdi+8]       ; A[1]
movsd   xmm1, [rsi+8]       ; B[1]
addsd   xmm0, xmm1
movsd   [rdi+8], xmm0       ; Store A[1]

movsd   xmm0, [rdi+16]      ; A[2]
... etc ...
```

Listing 6: A "loop unrolled" version of `loopadd.s`.

This is much *longer* code, but it is much *faster* because it contains no "control flow" (jumps or branches), just a straight line of data operations.

7 Hardware Speed-Tricks: Pipelining & Prediction

Modern CPUs use several "assembly line" tricks to execute instructions faster.

7.1 Pipelining

A CPU doesn't process one instruction at a time (start-to-finish). It uses an "assembly line" called a **pipeline**. A single instruction is broken into stages.

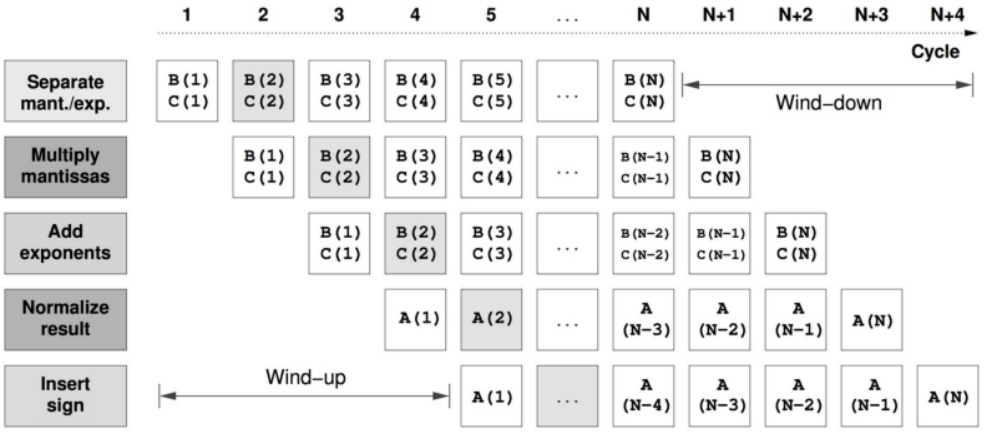

Figure 7: A CPU pipeline compared to a car assembly line. By breaking the task into stages (e.g., Fetch, Decode, Execute, Store) and working on multiple instructions at once, the "throughput" is much higher.

For example, a floating-point multiplication might be a 5-stage pipeline (e.g., 1. Decode, 2. Load Registers, 3. Multiply, 4. Normalize, 5. Store Register).

- **Latency:** The time for *one* instruction to go through all 5 stages is the **latency** (e.g., 5 cycles).
- **Throughput:** But, as soon as Instruction 1 moves to Stage 2, Instruction 2 can *enter* Stage 1. The pipeline is full, and a new result "pops out" of the end on *every cycle*.

This means that even though one multiplication has a 5-cycle latency, a loop of 1000 multiplications can be completed in (roughly) 1000 + 5 cycles, not 5000. This is called **Instruction Level Parallelism (ILP)**.

7.2 The Pipeline's Enemy: Branch Prediction

This beautiful pipeline has one major weakness: **branches** (i.e., `if` statements and loops).

The Problem: The CPU is at the "Fetch" stage. It needs to know the *next* instruction to "fetch" and put into the pipeline. But it's an `if` statement. The

CPU doesn't know whether to fetch the `if` block or the `else` block until the `if` condition is *executed* (which is 3 stages *later* in the pipeline). The pipeline must "stall" (stop and wait), which is devastatingly slow.

The Solution: Branch Prediction The CPU hardware *guesses* which path the branch will take.

- **Analogy:** You are a short-order cook. An order for a hamburger comes in. You *predict* (guess) they will also want fries, so you "speculatively" drop a batch of fries in the fryer.
- **If correct prediction (they want fries):** The fries are ready at the same time as the burger. You look like a genius. The pipeline is full and ran at full speed.
- **If wrong prediction (they wanted onion rings):** You must *abort* (throw away) the fries, and start the onion rings. This is a "misprediction penalty" and is very slow. The CPU must "flush" its pipeline of all the wrong, speculatively-fetched instructions and restart from the correct branch.

Modern CPUs are *extremely* good at this (often >95% accurate). But this implies that code with *unpredictable* branches (e.g., `if (rand() > 0.5)`) is *much slower* than code with predictable branches (e.g., a loop 'for(i=0; i<1000)' where the 'i<1000' branch is "true" 999 times and "false" only once).

8 The End of an Era: Moore's Law

For 50 years, the computer industry was driven by a trend called **Moore's Law**. In 1965, Gordon Moore observed that the number of transistors one could fit on a chip was doubling roughly every 18 months.

This had a magical effect: as transistors got smaller, they got *faster* and *cheaper*. For decades, programmers could write lazy, slow code, and a year later, new hardware would "magically" make it run fast.

This era is over.

As Figure 8 shows, we hit a "Power Wall."

- We are still packing *more transistors* on chips.
- But we **cannot** make the clock **frequency** (the "GHz") any faster. Why? **Power and Heat.** A faster clock "switches" more often, which uses exponentially more power and produces heat we can no longer cool. Your chip would melt.

This is the most important change in computing in 30 years. It means **software optimization will become more important,** because we can no longer rely on hardware to save us.

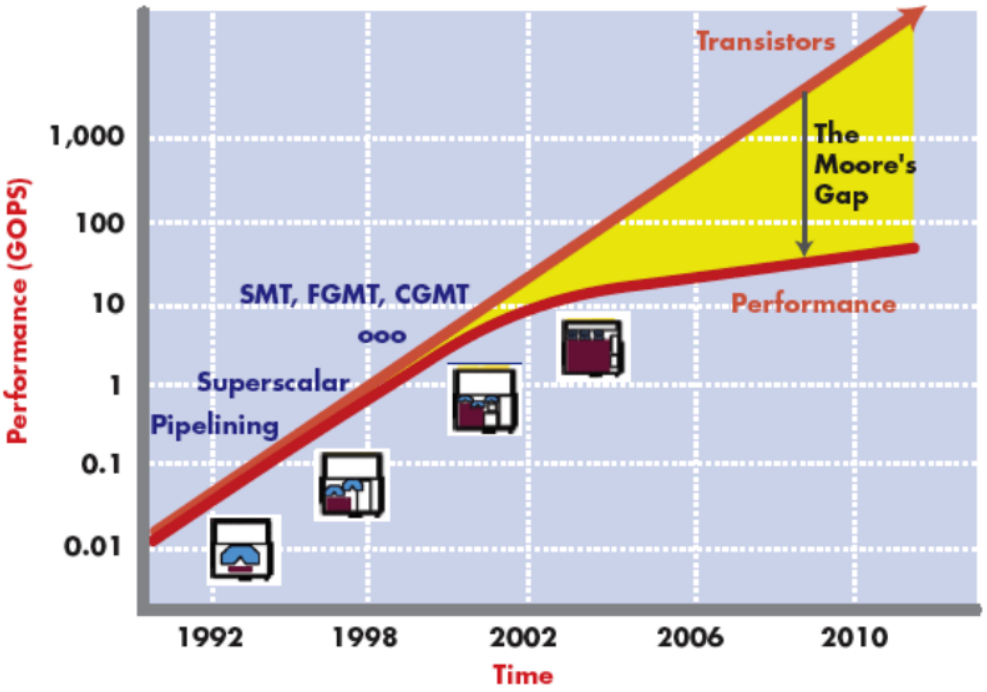

Figure 8: The "Moore's Gap" or "Power Wall." Transistor counts (red) continue to follow Moore's Law. But Clock Frequency (blue), the "speed" of the chip, *stopped* increasing around 2005. (Data from K. Rupp).

9 The New Solution: Parallelism

So what are we *doing* with all those extra transistors from Moore's Law, if not increasing clock speed? The answer: **Parallelism**.

Instead of building one *faster* brain (which we can't do), we are using the extra transistors to build *more* brains (cores) on a single chip.

- **Old Way:** A single-core 3.8 GHz Pentium 4. (One super-fast, super-hot chef).
- **New Way:** An 8-core 3.2 GHz Core i7. (Eight slightly-slower, cooler chefs working together).

This shift to **multicore** processors means that the *only* way to make your program faster is to *parallelize* it—to break your problem into pieces that can be solved by multiple "chefs" (cores) at the same time.

9.1 Parallelization 1: SIMD (In a Single Core)

The first type of parallelism is **SIMD (Single Instruction, Multiple Data)**. This uses special "vector" or "packed" instructions (with names like MMX, SSE, AVX) that perform the same operation on many values at once.

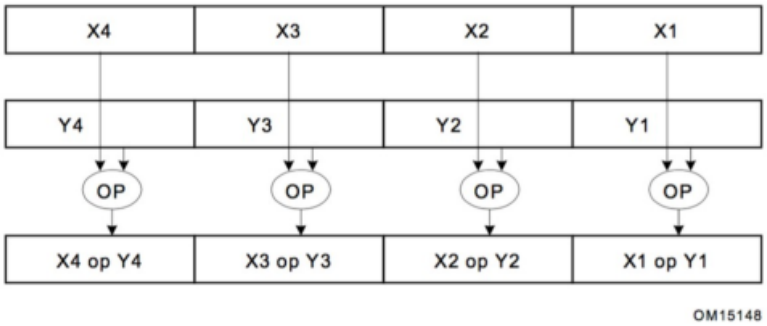

Figure 9: A SIMD "packed" add instruction. Instead of 4 separate 'add' instructions, a single instruction adds four pairs of numbers simultaneously.

Analogy: A SIMD instruction is like a large rubber stamp (the "instruction") that can stamp 4 documents (the "data") at once, instead of stamping them one-by-one.

This is perfect for scientific code, which often does the same operation on large arrays (like our `loopadd.cpp` example). A smart compiler will "vectorize" your loop, replacing it with these much faster SIMD instructions.

9.2 Parallelization 2: MIMD (With Multiple Cores)

The second type is **MIMD (Multiple Instruction, Multiple Data)**. This is what we normally mean by "parallelism." This is our "kitchen" of 8 chefs, who can all be doing *different* things (one chops, one sautés) at the same time. There are two main models for MIMD:

1. Shared Memory (Multicore)

- **How it works:** All CPU cores on your chip share access to the *same* main memory.
- **Analogy:** All chefs share a single, large refrigerator (Memory).
- **Pros:** Easy to share data. (To give Chef 2 an onion, Chef 1 just puts it on the counter).
- **Cons:** "Cache Coherency." What if Chef 1 takes the *last* milk, but Chef 2's *local inventory list* (his L1 cache) still says there is milk? This "cache-fighting" is a huge, complex problem.

2. Distributed Memory (Cluster)

- **How it works:** A "supercomputer" is just many separate computers (nodes), each with its *own* CPU and *own* private memory, connected by a fast network.

- **Analogy:** Every chef has their *own* private kitchen and refrigerator.
- **Pros:** Massively scalable. You can build a "kitchen" with 10,000 chefs.
- **Cons:** Sharing data is *very* slow. To give Chef 2 an onion, Chef 1 must stop cooking, package the onion, and mail it via a "network" (walkie-talkie and delivery boy). This is called "message passing."

9.3 Parallelization 3: The GPU (Massive Parallelism)

Finally, there is the **GPU (Graphics Processing Unit)**. A GPU is a form of "parallelism" taken to an extreme.

- **Analogy:** A GPU is *not* 8 smart chefs. It is a football stadium with **thousands** of "dumb" cores (the audience) who can all do *one simple task at the same time* (e.g., "everyone with a red card, hold it up now!").
- **Use Case:** GPUs are *brilliant* at massively parallel, SIMD-like tasks (graphics, machine learning, simple physics). They are *terrible* at complex, "branchy" (lots of `if`s) logic, which is what CPUs are good at.

10 The Memory Hierarchy in Detail

We must understand the memory hierarchy, as it is the source of most performance bottlenecks. The key principle is: **As you get closer to the CPU, storage gets faster, smaller, and more expensive (per-byte).**

Cache type	Access time (cycles)	Cache size (C)	Assoc. (E)	Block size (B)	Sets (S)
L1 i-cache	4	32 KB	8	64 B	64
L1 d-cache	4	32 KB	8	64 B	64
L2 unified cache	10	256 KB	8	64 B	512
L3 unified cache	40–75	8 MB	16	64 B	8,192

Figure 10: The Memory Hierarchy. Access times are orders of magnitude different. A 'cache miss' that goes to Main Memory can cost the CPU hundreds of cycles it could have spent doing work. A 'miss' that goes to disk costs *millions* of cycles.

The "latency" (access time) numbers in Figure 10 are critical.

- **SRAM** (Static RAM): Used in **Caches (L1-L3)**. It's made of 6 transistors (a "flip-flop").
  - **Analogy:** A light switch. It "statically" holds its value (0 or 1) as long as it has power.
  - **Pros/Cons:** Very fast, but big and expensive (6 transistors per bit).

- **DRAM** (Dynamic RAM): Used in **Main Memory (RAM)**. It's made of 1 transistor + 1 capacitor.
  - **Analogy:** A tiny, leaky bucket. A full bucket is a '1'. Because it "leaks," the computer must "dynamically" run a "refresh circuit" to refill all the buckets, thousands of times per second.
  - **Pros/Cons:** Very dense and cheap (1 transistor per bit), but slower (due to leaks and refresh).

This technology difference is *why* we have the hierarchy. We can't afford to make our whole RAM out of fast SRAM.

10.1 Why Caches Work: The Principle of Locality

Caches are effective because programs are predictable.

1. **Temporal Locality (Locality in Time):** If you access a piece of data, you are very likely to access it *again* soon. (e.g., the variable `sum` inside a loop).
2. **Spatial Locality (Locality in Space):** If you access a piece of data, you are very likely to access the data *right next to it* soon. (e.g., `A[0]`, then `A[1]`, then `A[2]`...).

10.2 Cache Lines: The Key to Spatial Locality

To exploit spatial locality, the memory system does *not* move data 1 byte at a time. It moves data in 64-byte "chunks" called **cache lines**.

- When your code asks for `A[0]` (a **cache miss**)...
- The CPU goes to Main Memory and fetches the *entire 64-byte cache line* that contains `A[0]`. This chunk also contains `A[1]` through `A[7]` (assuming `double`s are 8 bytes).
- This chunk is placed in the L1 cache.
- When your code *then* asks for `A[1]`, it's an L1 **cache hit**.
- ...When you ask for `A[2]`, `A[3]`... `A[7]`, they are *all* L1 cache hits.

This means iterating through a contiguous array (`std::vector`) is **extremely fast**.

10.3 A Parallel Bug: False Sharing

This cache line system creates a subtle but terrible bug in parallel programs: **False Sharing**.

- **Analogy:** Two chefs, Core 1 and Core 2. Chef 1 needs `salt`. Chef 2 needs `pepper`. These are *unrelated* variables.
- **The Problem:** You "saved space" by putting `salt` and `pepper` in the *same spice caddy* (the 64-byte cache line).
- **The Bug:** 1. Core 1 needs `salt`. It "loads" the whole caddy into its L1 cache. 2. Core 2 needs `pepper`. It "loads" the whole caddy into *its* L1 cache. This "invalidates" Core 1's copy. 3. Core 1 needs `salt` again. It *steals* the caddy back, invalidating Core 2's copy. 4. Core 2 needs `pepper` again. It *steals* the caddy back...
- **Result:** The two cores spend all their time "ping-ponging" the cache line over the bus, even though they aren't sharing *any* data. This is "false" sharing. The program runs 100x slower than it should.
- **Solution:** "Pad" your data structure (add empty space) to ensure `salt` and `pepper` are on *different* cache lines.

11 Beyond Physical RAM: Virtual Memory

What happens when your 32 GB simulation needs to run on your 8 GB laptop? The program crashes, right? No. This is solved by **Virtual Memory**.

The key idea is to use the (slow) **disk** as "overflow" for RAM. The operating system (OS) and CPU hardware (the **MMU - Memory Management Unit**) trick your program.

- **Virtual Address Space (VAS):** Your program *thinks* it has a private, massive, contiguous block of memory (e.g., 256 Terabytes on a 64-bit system). This is the "virtual" view.
- **Physical Address Space (PAS):** The *actual*, small, 8GB of RAM chips, which is shared by all running programs.

The OS and MMU manage this "illusion" by breaking memory into "pages."

11.1 Pages, Page Tables, and the TLB

- **Pages:** Memory is managed in 4KB "pages."
- **Page Table:** The OS keeps a "map" called the **Page Table** for your program. This map translates virtual pages to physical pages.

  - `Virtual Page 1 -> Physical Page 5 (in RAM)`
  - `Virtual Page 2 -> Physical Page 2 (in RAM)`
  - `Virtual Page 3 -> (On Disk at location X)`

- **Page Fault:** When your program tries to access `Virtual Page 3`, the MMU checks the page table and sees it's "on disk." This triggers a **page fault**, which is *extremely* slow. The OS must: 1. Pause your program. 2. Find an "old" page in RAM (e.g., `Virtual Page 2`). 3. Write that page out to disk (if it was changed). 4. Read `Virtual Page 3` from disk into that now-empty spot in RAM. 5. Update the page table: (`VP2 -> On Disk`, `VP3 -> Physical Page 2`). 6. Resume your program.

- **The TLB (Translation Lookaside Buffer):** This creates a new problem. The Page Table is *itself* in RAM! This means *every* memory access (e.g., `LOAD A[i]`) would require *two* memory accesses: one to read the page table, and one to read `A[i]`. This would cut performance in half.

- **Solution:** The MMU contains a small, fast *cache for the page table*, called the **TLB**. It stores recent translations (e.g., `VP1 -> PP5`). This is another example of temporal locality!

12 Putting It All Together: A Case Study

This all seems very abstract. How does it affect our C++ code?

Problem: In our "Penna model" simulation, we need to store a large collection of "animals." We need to iterate over all animals every "turn," and we need to "remove" animals from the collection when they die.

What is the best C++ data structure? `std::vector` or `std::list`?

12.1 The "Textbook" Answer: `std::list`

A "by-the-book" CS student might argue for `std::list`.

- **How it works:** A "linked list" stores each animal in a separate, small block of memory, which also contains "pointers" to the `next` and `previous` animals.

- **Memory Layout:** The animals are scattered *all over RAM*.

- **Pro:** Removing an animal from the middle is O(1) (very fast), *if* you already have an iterator to it. You just re-wire the `next`/`prev` pointers of its neighbors.

12.2 The "Hardware-Aware" Answer: `std::vector`

A `std::vector` seems like a bad choice.

- **How it works:** A single, contiguous block of memory.
- **Memory Layout:** All animals are side-by-side in RAM: `A[0]`, `A[1]`, `A[2]`....
- **Con:** Removing an animal from the middle is O(N) (very slow). You have to "shift" *all* subsequent animals (`A[i+1]`, `A[i+2]`...) down by one to fill the "gap."

12.3 The Surprising Winner: `std::vector`

In almost all real-world tests, the `std::vector` is *dramatically* faster. Why?

Reason 1: Caches!

- **Iterating a `std::vector`:** This is a *perfect* example of **Spatial Locality**. When you iterate to "process" all the animals, you access `A[0]`. The CPU fetches the *entire cache line*, loading `A[1]`...`A[7]` into the L1 cache "for free." The entire loop is a blazing-fast series of L1 hits.
- **Iterating a `std::list`:** This is a *cache nightmare*. It's called **pointer chasing**. You access `A[0]`. To get to the next animal, you follow a pointer, which could be *anywhere* in RAM. This is a **cache miss**. The CPU stalls. Once it (slowly) arrives, you process it and follow its `next` pointer... to another random location. Another **cache miss**.

The `std::list` iteration spends 99% of its time stalled on the Von Neumann bottleneck, while the `std::vector` screams along at L1 cache speed.

Reason 2: A Better Removal Algorithm The "slow" O(N) removal of `std::vector` was based on a false assumption: that we need to *preserve the order* of the animals. The problem (slide 45) states: "We do not care about the order of the animals."

This allows for a *brilliant* O(1) removal trick for a `std::vector`:

```
// To remove the animal at 'index_to_remove' from 'animal_vector'

// 1. Take the \emph{last} animal in the vector...
animal_vector[index_to_remove] = animal_vector.back();

// 2. ...and pop off the (now redundant) last element.
animal_vector.pop_back();

// Done. O(1) time. No shifting.
```

Listing 7: A fast O(1) "swap-and-pop" removal for `std::vector`.

Conclusion: The `std::vector` (with the "swap-and-pop" trick) is O(1) for removal *and* is perfectly cache-friendly for iteration. The `std::list` is O(1) for removal but is *disastrous* for the cache during iteration.

The "hardware-aware" `std::vector` solution wins, and it's not even close. This is the perfect example of why we must "Know Your Tools."

13 Further Reading

For a much deeper dive on these topics, please see the following excellent (and very dense) textbooks:

- Bryant & O'Hallaron, "Computer Systems: A Programmer's Perspective", 2016.
- Hennessy & Patterson, "Computer Architecture: A Quantitative Approach", 2011.