GWDG

Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

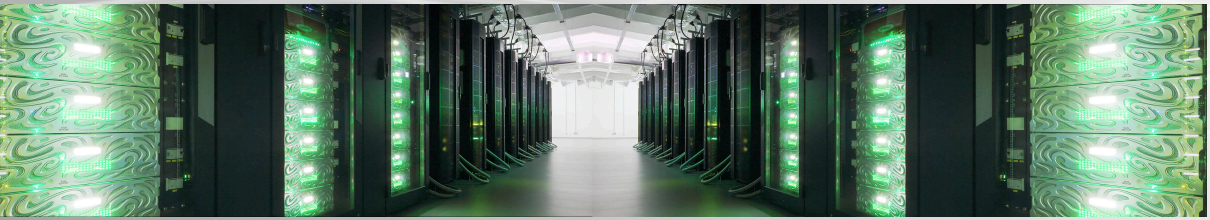
GWDG

hpc.gwdg.de

Performance Engineering

Applied Fun to make your Code FAST

Lars Quentin



Philosophy of the Lecture

- The goal of this lecture is to maximize **fun**
- Not to teach the formal basics of benchmarking or performance engineering
- Thus, it's focused on more hands-on stuff that is outside of HPC as well!
  - Games, HFT, Audio, Big Tech Infra, Databases, ...
- Design Decision: Low Level Benchmarking on HPC is difficult here
  - `perf_event_paranoid` is set so high that perf counters won't fully work
  - This will *not* be a problem on your laptop / prod server
  - But, thus we go light on actual `perf`/`likwid` tooling

Table of Contents

1 Benchmarking

2 CPU Architecture

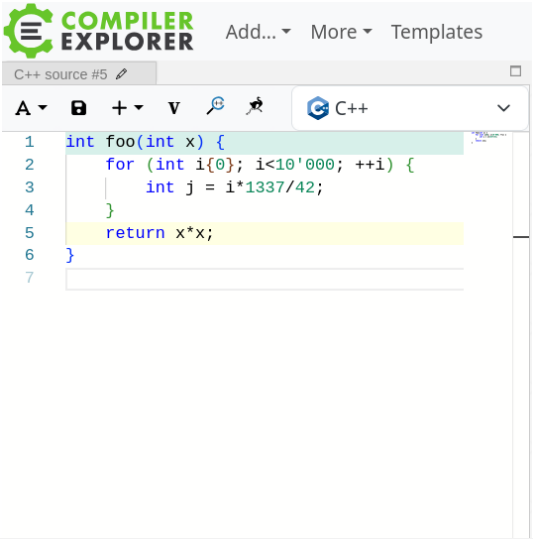
3 CPU Tricks

4 Memory Alignment & SIMD

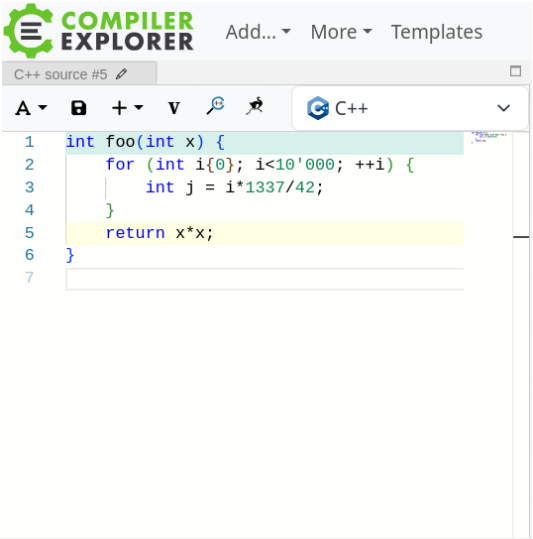
5 Tooling and Tricks

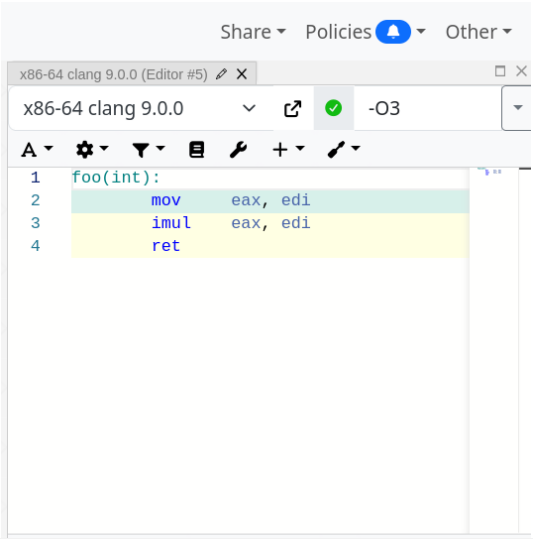
6 What now?

Benchchmarking is hard: What does this code do?



Benchchmarking is hard: What does this code do?





WHY Benchmarking is hard

- The compiler can optimize more than you think
- The compiler can optimize differently based on...
  - the compiler version (and vendor)
  - the targeted architecture (auto vectorization)
  - the surrounding/calling code (i.e. inlining)
  - what the code is linked with
- Instrumentalization changes the characteristics of the code
- Hardware has a lot of variability
  - Process scheduler
  - Interrupts killing your caches
  - Runtime branch predictor
- Bottlenecks can become unpredictable

But: Benchmarking is needed

The CPU vs your mental model

- The programmer only writes high-level, serial code.
- But actually, the CPU does
  - Multiple layers of caching
  - Out-of-order execution (only in-order commits)
  - Branch prediction
  - Memory Prefetching
- The compiler furthermore does
  - Struct alignment
  - Compile Time Transformations
  - Constant folding
  - Vectorization
  - Inverting branches...

Proof: Which is faster

```
~/code/pcpc/perf-eng/slides/code$ batcat row.c
```

File: row.c

```c
int array[10000][10000];

int main() {
  for (int i = 0; i < 10000; i++) {
      for (int j = 0; j < 10000; j++) {
          array[i][j] = 0;
      }
  }
}
```

Proof: Which is faster

~/code/pcpc/perf-eng/slides/code$ batcat row.c

File: row.c

```c
int array[10000][10000];

int main() {
  for (int i = 0; i < 10000; i++) {
      for (int j = 0; j < 10000; j++) {
          array[i][j] = 0;
      }
  }
}
```

~/code/pcpc/perf-eng/slides/code$ batcat col.c

File: col.c

```c
int array[10000][10000];

int main() {
  for (int i = 0; i < 10000; i++) {
      for (int j = 0; j < 10000; j++) {
          array[j][i] = 0;
      }
  }
}
```

Proof: Which is faster

~/code/pcpc/perf-eng/slides/code$ batcat row.c

File: row.c

```
1  int array[10000][10000];
2
3  int main() {
4    for (int i = 0; i < 10000; i++) {
5        for (int j = 0; j < 10000; j++) {
6            array[i][j] = 0;
7        }
8    }
9  }
```

~/code/pcpc/perf-eng/slides/code$ batcat col.c

File: col.c

```
1  int array[10000][10000];
2
3  int main() {
4    for (int i = 0; i < 10000; i++) {
5        for (int j = 0; j < 10000; j++) {
6            array[j][i] = 0;
7        }
8    }
9  }
```

```
~/code/pcpc/perf-eng/slides/code$ gcc -O2 row.c -o row
~/code/pcpc/perf-eng/slides/code$ hyperfine "./row test"
Benchmark 1: ./row test
  Time (mean ± σ):      51.9 ms ±   3.0 ms    [User: 15.0 ms, System: 36.7 ms]
  Range (min …max):    47.4 ms … 59.3 ms    60 runs
```

Proof: Which is faster

```
~/code/pcpc/perf-eng/slides/code$ batcat row.c
```

File: row.c

```
1  int array[10000][10000];
2
3  int main() {
4    for (int i = 0; i < 10000; i++) {
5        for (int j = 0; j < 10000; j++) {
6            array[i][j] = 0;
7        }
8    }
9  }
```

```
~/code/pcpc/perf-eng/slides/code$ gcc -O2 row.c -o row
~/code/pcpc/perf-eng/slides/code$ hyperfine "./row test"
Benchmark 1: ./row test
  Time (mean ± σ):      51.9 ms ±   3.0 ms    [User: 15.0 ms, System: 36.7 ms]
  Range (min …max):    47.4 ms … 59.3 ms    60 runs
```

```
~/code/pcpc/perf-eng/slides/code$ batcat col.c
```

File: col.c

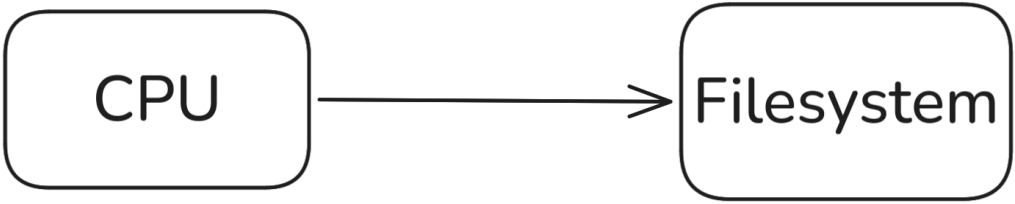
```
1  int array[10000][10000];
2
3  int main() {
4    for (int i = 0; i < 10000; i++) {
5        for (int j = 0; j < 10000; j++) {
6            array[j][i] = 0;
7        }
8    }
9  }
```

```
~/code/pcpc/perf-eng/slides/code$ gcc -O2 col.c -o col
~/code/pcpc/perf-eng/slides/code$ hyperfine "./col test"
Benchmark 1: ./col test
  Time (mean ± σ):     131.2 ms ±   2.9 ms    [User: 78.1 ms, System: 52.7 ms]
  Range (min …max):   128.0 ms … 140.4 ms    22 runs
```

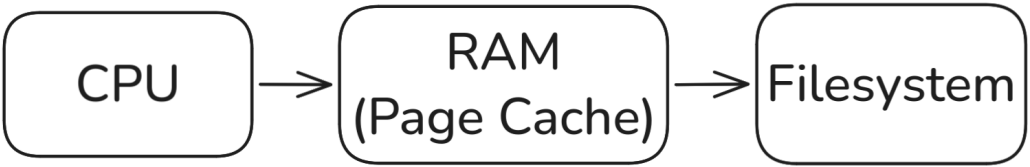
Proof: Which is faster

Let’s learn why!

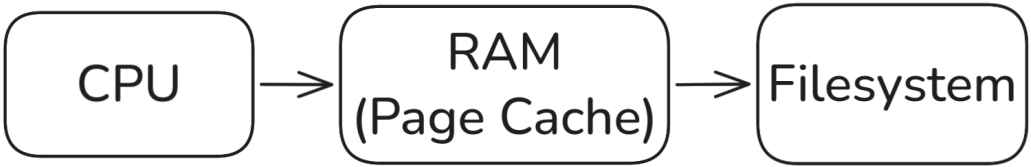
How I/O is taught



How I/O more works like

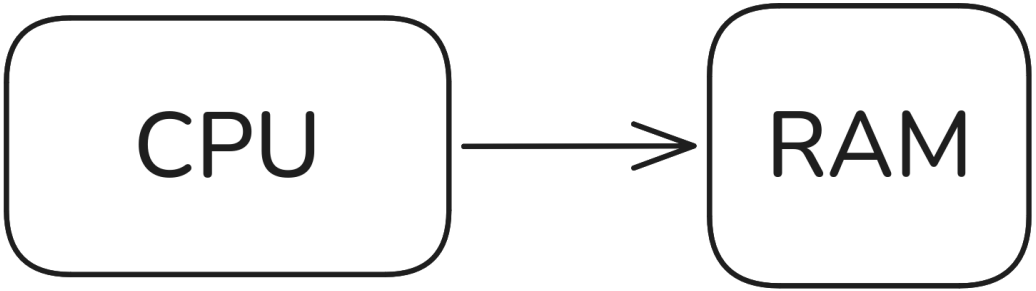


How I/O more works like

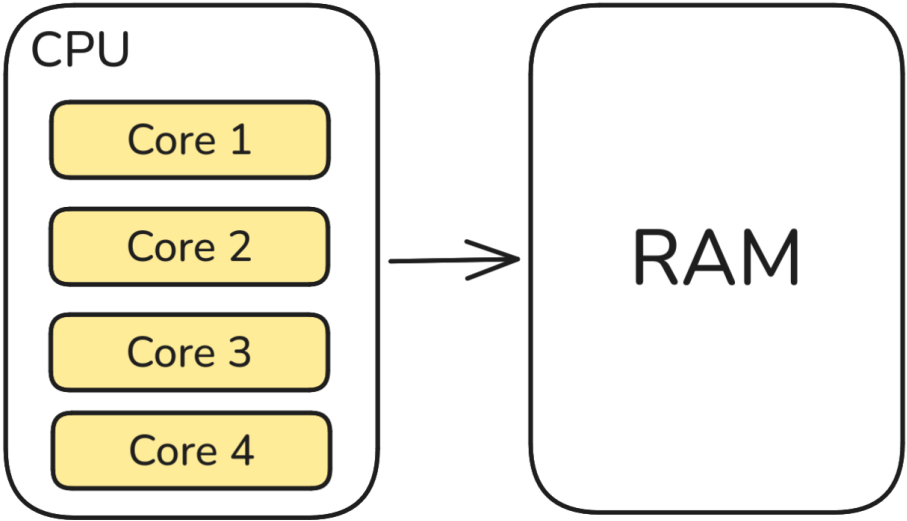


So... RAM is fast right?

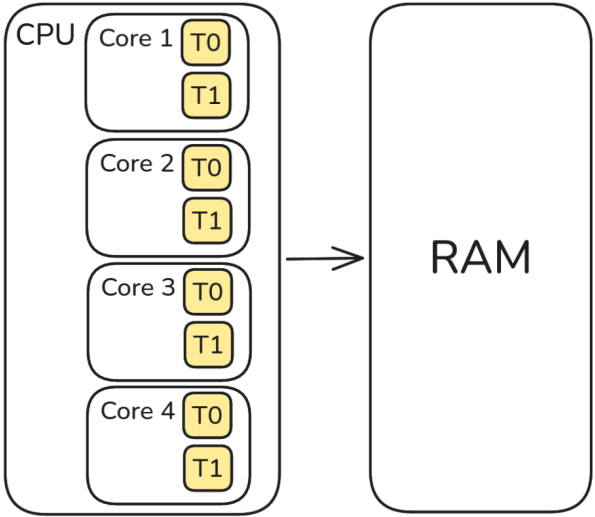
How CPUs are taught (Variable Access)



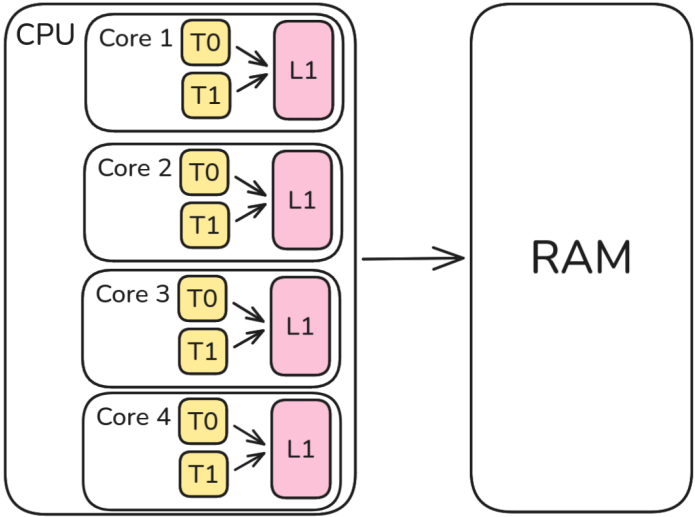
How CPUs actually are



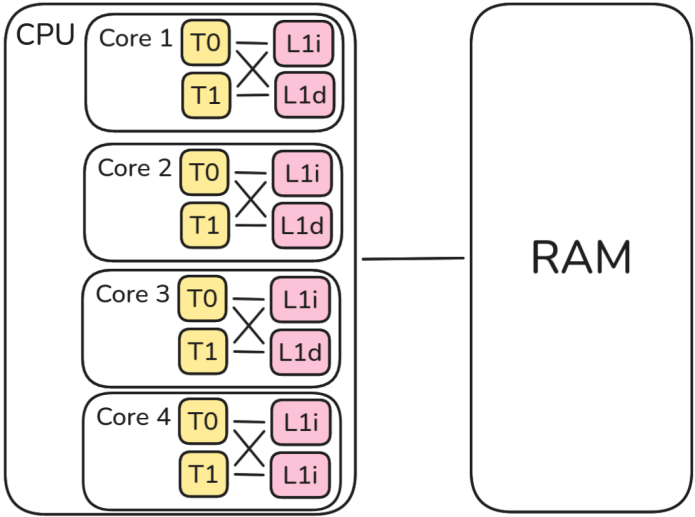
How CPUs actually are (if you are pedantic)



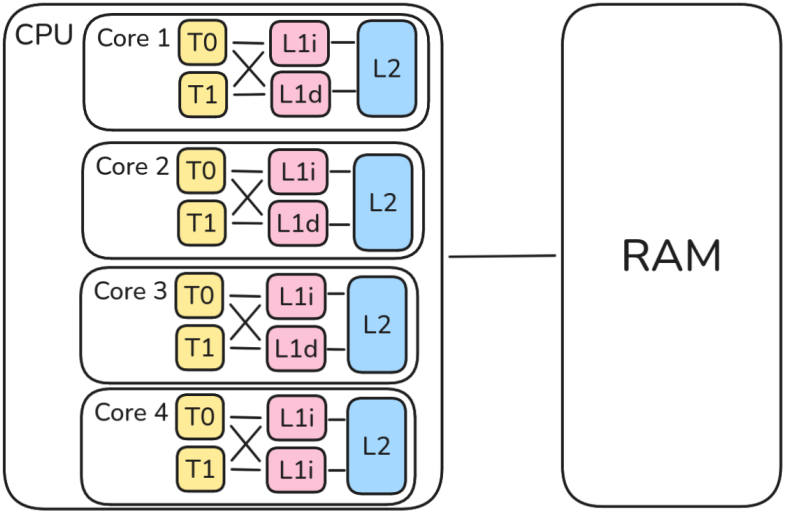
How CPUs *actually* are



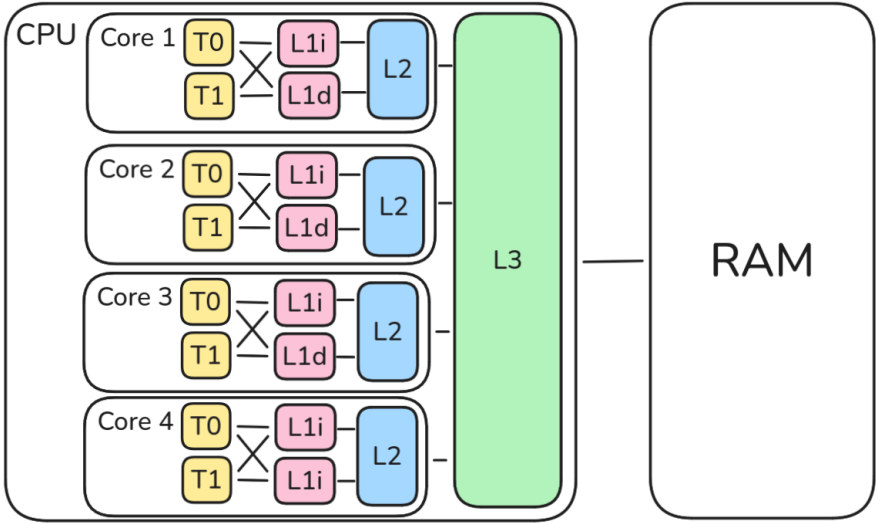
How CPUs **actually** are



How CPUs **ACTUALLY** are



How CPUs actually are... simplified



Why are CPUs complicated like that? [1]

Memory	Size	Latency	Bandwidth
L1 cache	32 KB	1 nanosecond	1 TB/second
L2 cache	256 KB	4 nanoseconds	1 TB/second Sometimes shared by two cores
L3 cache	8 MB or more	10x slower than L2	>400 GB/second
Main Memory on DDR DIMMs	4 GB-1 TB	2x slower than L3	400 GB/second

But: CPU caches are small

- i5-1245U has 12 threads
- But only 10 physical cores
  - 2 P-cores (×2 threads via HT)
  - 8 E-cores (no HT)
- L1 is *per physical core*, not per thread

But: CPU caches are small

- i5-1245U has 12 threads
- But only 10 physical cores
  - 2 P-cores (×2 threads via HT)
  - 8 E-cores (no HT)
- L1 is *per physical core*, not per thread

Size:

L1d	352 KiB (×10)
L1i	576 KiB (×10)
L2	6.5 MiB (×4)
L3	12 MiB (×1)

But: CPU caches are small

- i5-1245U has 12 threads
- But only 10 physical cores
  - 2 P-cores (×2 threads via HT)
  - 8 E-cores (no HT)
- L1 is *per physical core*, not per thread

Size:

L1d	352 KiB (×10)
L1i	576 KiB (×10)
L2	6.5 MiB (×4)
L3	12 MiB (×1)

On Access:

1. Check L1 → miss
2. Check L2 → miss
3. Check L3 → miss
4. Fetch from RAM
5. Fill L3 ← data
6. Fill L2 ← data
7. Fill L1 ← data
8. CPU reads from L1

Outline

1 Benchmarking

2 CPU Architecture

3 CPU Tricks

4 Memory Alignment & SIMD

5 Tooling and Tricks

6 What now?

Cache Lines

What is a Cache Line?

- Smallest unit of data transfer between RAM and cache
- Typically **64 bytes** on x86
- Must be **aligned** in memory (i.e. starts at a multiple of 64)
- Query your system:

```
$ getconf LEVEL1_DCACHE_LINESIZE
```

Good: Spatial Locality

- Access element $i \Rightarrow$ neighbors come for free
- Sequential array iteration is fast

Bad: Poor Utilization

- Need 1 byte? Pay for 64 bytes of RAM latency
- Sparse/pointer-heavy structures waste cache space
- Only $\frac{1}{64}$ utilization in the worst case

Tricks CPUs Do

Pipelining

- Instructions split in stages
- These stages need different units
- Thus: Parallelizable

Superscalar Execution

- Multiple ALUs can do multiple things
- Only works with *independent* instructions

Speculative Execution

- Pipelining + Superscalar: How to handle branching?
- CPU predicts outcomes
- Correct prediction: free speedup
- Wrong prediction: pipeline must be rolled back (costly)
- Speculative state is observable via side channels

$\Rightarrow$ Spectre & Meltdown

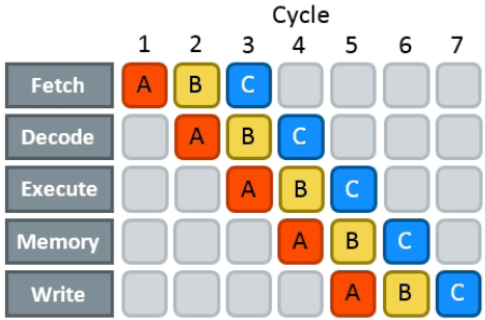
Instruction Pipelining

Stages (classic RISC)

- **IF**: Fetch instruction
- **ID**: Decode + read registers
- **EX**: Execute / ALU
- **MEM**: Memory access
- **WB**: Write back result

The Payoff

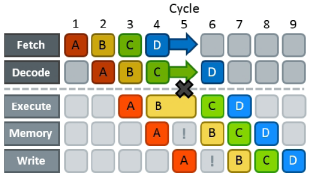
- Each stage uses *different* hardware
- $\Rightarrow$ 1 instruction **per cycle**
- No work needed!



Exemplary Pipeline [2]

Pipeline Hazards

- Structural Hazard
  - Two instructions need the *same* unit
  - Stall until unit is free (∼1 cycle)
  - Not really an easy fix
- Data Hazard
  - Data Dependency (See Pointer Chasing Exercise)
  - Solution: shorten the critical path
- Control Hazard
  - Branch: CPU doesn't know what's next
  - Full pipeline flush (∼15–20 cycles!)
  - Solution: remove or make branches predictable



Hazard [3]

Outline

1 Benchmarking

2 CPU Architecture

3 CPU Tricks

4 Memory Alignment & SIMD

5 Tooling and Tricks

6 What now?

Alignment and Padding

What is Alignment?

- Data must sit at an address that is a **multiple of its size**
- e.g. 4-byte `int` ⇒ address divisible by 4

Structs and Padding

- Compiler inserts **invisible padding bytes**
- See: `sizeof(struct)`
- Reorder fields to optimize!

`__attribute__((packed))`

- Forces **no padding** between fields
- Saves memory...
- ...but causes misaligned accesses
- ⇒ slower or broken on some architectures
- Avoid unless you really know why

Struct Padding: Bad Example

C

```c
struct Bad {
    char   a;      // 1 byte
                   // 3 bytes padding
    int    b;      // 4 bytes
    char   c;      // 1 byte
                   // 7 bytes padding
    double d;      // 8 bytes
    short  e;      // 2 bytes
                   // 6 bytes padding
};
// Total: 24 bytes (only 16 bytes of actual data!)
```

Struct Padding: Good Example

C

```c
struct Good {
    double d;      // 8 bytes
    int    b;      // 4 bytes
    short  e;      // 2 bytes
    char   a;      // 1 byte
    char   c;      // 1 byte
};
// Total: 16 bytes (zero padding!)
```

SIMD and Auto-Vectorization

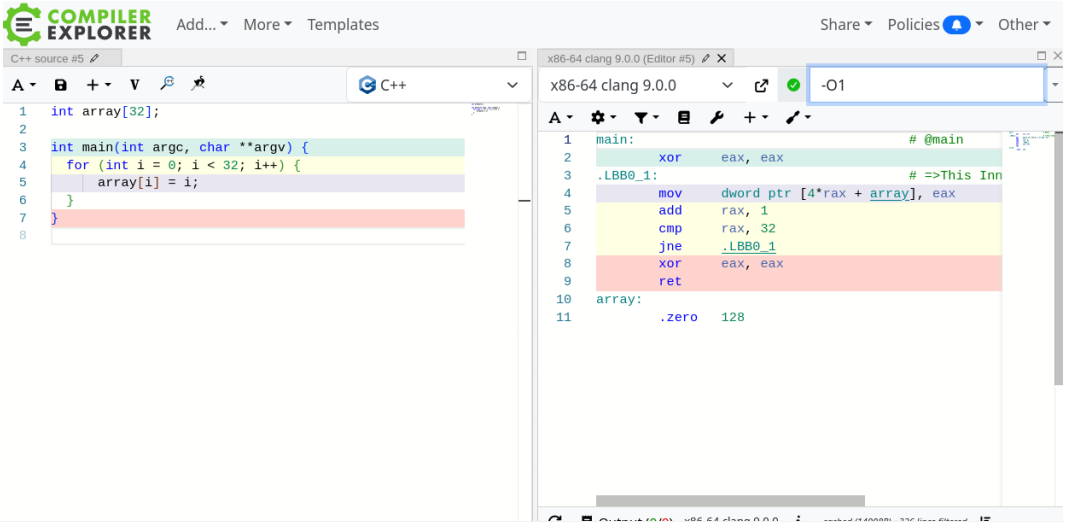
What is SIMD?

- **S**ingle **I**nstruction, **M**ultiple **D**ata
- One instruction operates on **multiple values at once**
- Add 8 floats in a single cycle (AVX)
- *Auto-Vectorization* supported by compiler!

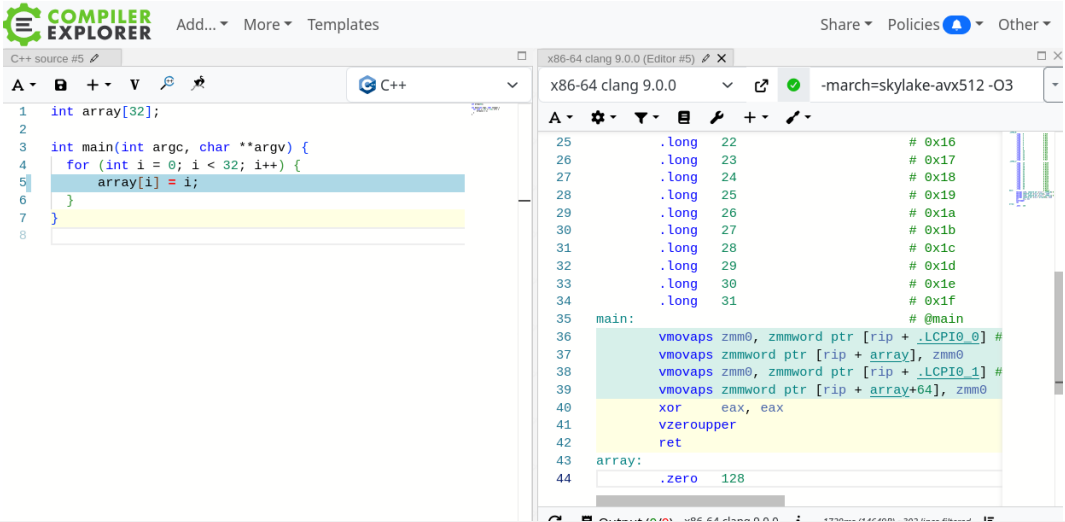
Optimizing for Auto-Vectorization

- Requires `-O3`
- **Keep memory accesses continous**
- Clear loop conditions
- Pro-Tipp: Struct of Arrays (SoAs)
  - More on that later!

SIMD example (unoptimized)



SIMD example (optimized (512-bit))



Outline

1 Benchmarking

2 CPU Architecture

3 CPU Tricks

4 Memory Alignment & SIMD

5 Tooling and Tricks

6 What now?

Types of Profiling

Instrumentation

- Add instructions into the program
- Exact, deterministic results
- But: changes the code itself

⇒ Observer effect!

Statistical Profiling

- Sample the program at intervals
- Done via interrupts
- Results are approximations

⇒ the longer the better

Program Simulation

- Run program in a simulated CPU
- Perfect insight into hardware behavior
- e.g. Valgrind / Cachegrind
- But: extremely slow
- Might not reflect your *actual* Hardware!

Getting Accurate Results

How to Compile

- Use `-O2` or `-O3`
  - Same as prod!
- Prevent the compiler from optimizing away your benchmark
  - `volatile` for variables
  - `__attribute__((noinline))` for functions

⇒ Better: use a benchmarking library

How to Run

- Minimize background processes
- Warm up caches
  - Compute: L1-L3
  - I/O: Page Cache
- Outliers are relevant
- Measure a lot
- When to stop: Small stddev

Latency vs Throughput

Latency

- Time start to finish of **one** op
- e.g. how long does one trade take?

$\Rightarrow$ Matters for: HFT, Audio, Real-Time

Throughput

- How many operations per second?
- e.g. how many FLOPS?

$\Rightarrow$ Matters for: HPC, Simulations, ML

Latency vs Throughput

Latency

- Time start to finish of **one** op
- e.g. how long does one trade take?

⇒ Matters for: HFT, Audio, Real-Time

Throughput

- How many operations per second?
- e.g. how many FLOPS?

⇒ Matters for: HPC, Simulations, ML

These are not the same goal!

- Optimizing one can *hurt* the other
- e.g. batching increases throughput...
- ...but increases latency per item
- Pipelining hides latency behind throughput
- Sometimes you only care about *hot path* latency

⇒ Always know which one you care about!

Branch Optimization: Likely/Unlikely

The Problem

- CPU assumes branch outcome
- Wrong guess: flush pipeline ($\sim$ 15-20 cycles)

The Hint

- Tell compiler common branch

$\Rightarrow$ Hot path stays linear

$\Rightarrow$ Better L1i cache usage

$\Rightarrow$ Less pipeline flushing

In Practice

- C++20: `[[likely]]` / `[[unlikely]]`
- GCC/Clang: `__builtin_expect`
- ALWAYS BENCHMARK BEFORE

$\Rightarrow$ Wrong hint is worse than no hint

Branch Optimization: Likely/Unlikely

C++

```cpp
template <class T>
void process_data(const std::vector<T>& data) {
  // If the sensor works, it can't be below zero
  for (int value : data) {
    if (value > 0) [[likely]] {
      process_data_fast(value);
    }
    else [[unlikely]] {
      process_error_slow(value);
    }
  }
}
```

Profile Guided Optimization (PGO)

What is PGO?

- Compiler optimizes based on **real** runtime data
- No manual annotations needed
- Works by instrumenting the binary, profiling it, then recompiling
  1. Compile with PGO instrumentation enabled
  2. Run with a normal workload, records branching
  3. Recompile with that branching info!
- Very good for low-variance tasks $\Rightarrow$ Firefox Build Server!

$\Rightarrow$ Profile must be representative of real usage!

Data-oriented Design: AoS vs SoA

Array of Structs (AoS)

- Each element is a struct with all fields
- Fields of one element are contiguous

⇒ Great for accessing *one* element at a time

⇒ Poor SIMD and cache utilization

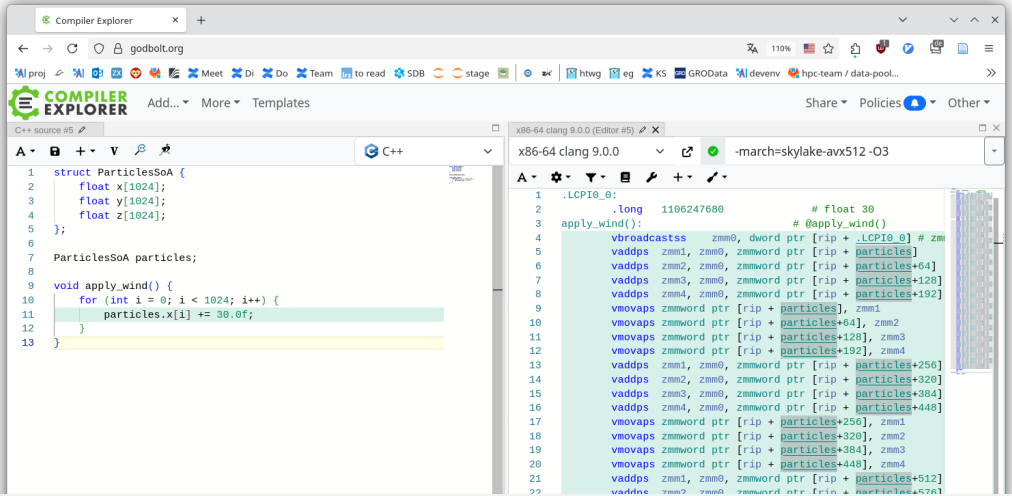
Struct of Arrays (SoA)

- Each field is its own array
- Same field of all elements are contiguous

⇒ Great for processing *one field* across all elements

⇒ SIMD and cache friendly!

Data-oriented Design: AoS vs SoA



hyperfine

What is it?

- CLI macrobenchmarking tool
- Auto-determines number of runs
- Statistical analysis built in
- Warmup runs: `-warmup`
- Cold cache: `-prepare`
- Parameterized: `-parameter-scan`

Shell

```
# Compare two commands
hyperfine 'find . -name foo' \
          'fd foo'

# Warmup + cold cache
hyperfine --warmup 3 \
  --prepare 'sync' \
  './my_program'

# Vary a parameter
hyperfine \
  --parameter-scan n 1 8 \
  './my_program --threads {n}'
```

perf

What is it?

- Linux kernel's built-in profiler
- Statistical profiling via hardware counters
- Samples the call stack at intervals

⇒ Low overhead, production safe

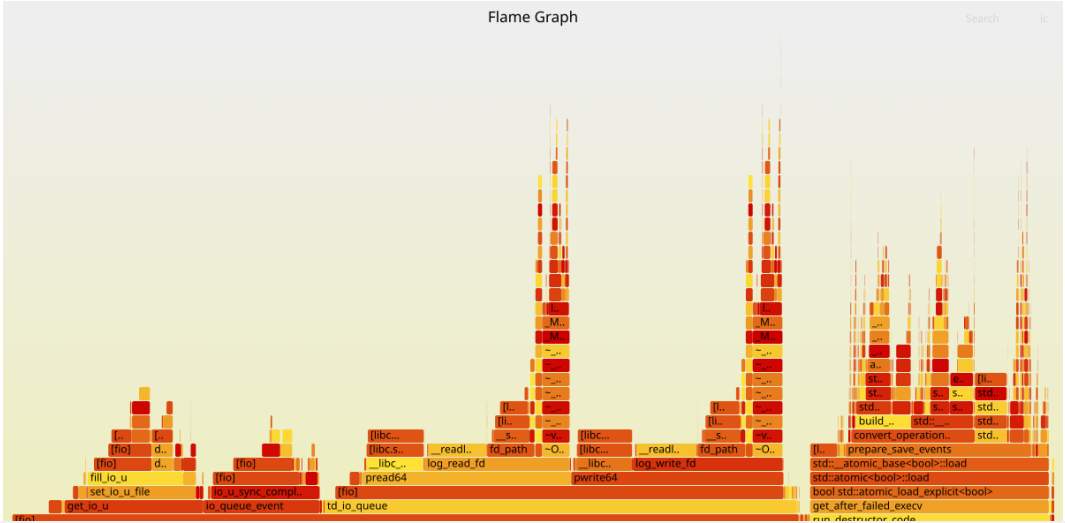
What can it measure?

- FIND OUT: WHERE IS THE WORK DONE
- CPU cycles, instructions
- Cache/Branch/Page misses

Key Commands

- `perf stat`: summary of counters
- `perf record`: record a profile
- `perf report`: inspect results
- `perf top`: live, system-wide view

perf: Flame Graphs



Cachegrind

What is it?

- Valgrind tool for cache simulation
- Literally simulates a fake CPU
- Thus: Not just statistical approixmation
- Exact, deterministic results
- But: 20-100x slower than native!
  - Multithreading: Serialized; Timings are fully off!

$\Rightarrow$ Use for targeted cache investigation, not general profiling

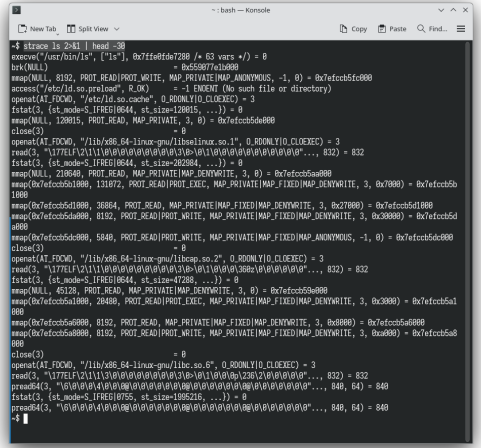
Niche Tools: strace

What is it?

- Traces all **system calls** a program makes
- Shows arguments and return values
- Great for: unexpected I/O, missing files, permission issues

⇒ Not a performance tool per se

⇒ But reveals *why* something is slow



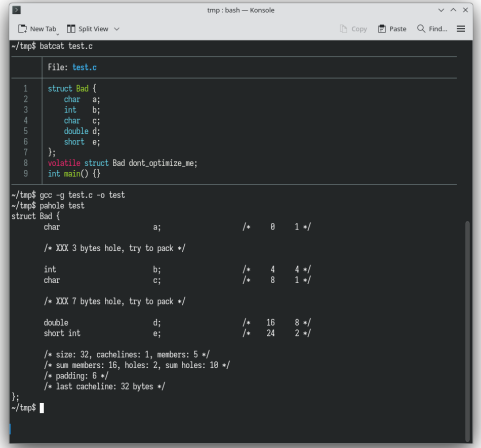
Niche Tools: pahole

What is it?

- Shows struct layout and padding holes
- Ties back to alignment and padding!

⇒ Find wasted bytes in your structs

⇒ Suggest optimal field ordering



Want to learn more?

- Goated: Algorithmica: Algorithms for Modern Hardware
- Deep Dive: What Every Programmer Should Know About Memory
- Legendary talks:
  - Scott Meyers: Cpu Caches and Why You Care
  - Mike Acton: Data-Oriented Design and C++ (The hawaii shirt guy)
  - Matt Godbolt: What Every Programmer Should Know about How CPUs Work
  - Herb Sutter: atomic<> Weapons: The C++ Memory Model and Modern Hardware (Multithreading bible!)

References

[1] Bevin Brett. *Memory Performance in a Nutshell*. Intel. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html (visited on 06/07/2026).

[2] *Instruction-Level Parallelism - Algorithmica*. URL: https://en.algorithmica.org/hpc/pipelining/ (visited on 06/08/2026).

[3] *Pipeline Hazards - Algorithmica*. URL: https://en.algorithmica.org/hpc/pipelining/hazards/ (visited on 06/08/2026).