

Performance Engineering

Applied Fun to make your Code FAST

Lars Quentin



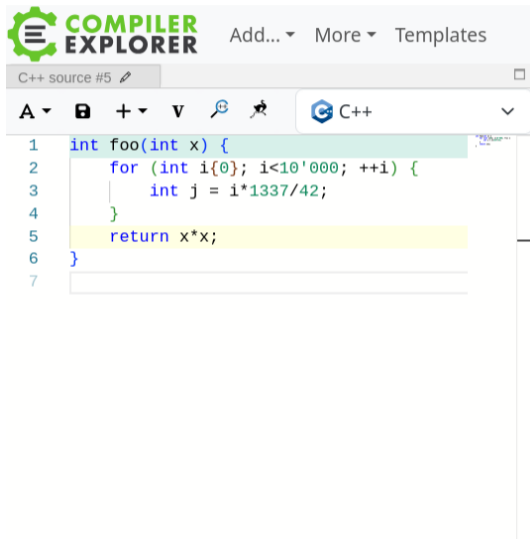
Philosophy of the Lecture

- 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?



The screenshot shows the Visual Studio Code editor with a C++ source file named 'C++ source #5'. The code defines a function 'foo' that takes an integer 'x' and returns 'x*x'. Inside the function, there is a loop that iterates from 0 to 9999, and in each iteration, it calculates 'i*1337/42' and assigns the result to 'j'. The function body is highlighted in light green, and the return statement is highlighted in light yellow.

```
1 int foo(int x) {  
2     for (int i{0}; i<10'000; ++i) {  
3         int j = i*1337/42;  
4     }  
5     return x*x;  
6 }  
7
```

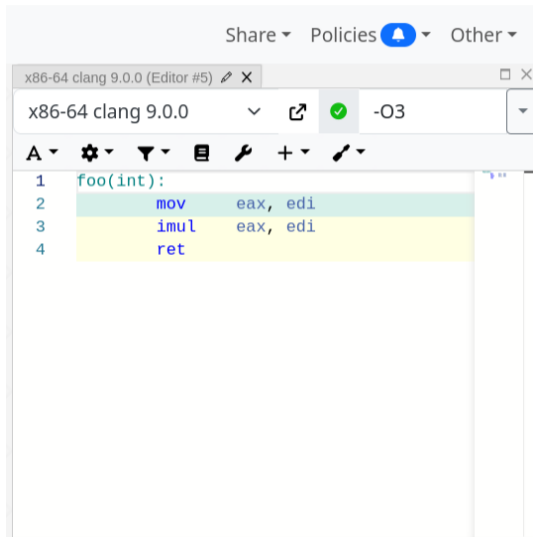
Benchmarking is hard: What does this code do?



COMPILER EXPLORER Add... More Templates

C++ source #5

```
1 int foo(int x) {  
2     for (int i{0}; i<10'000; ++i) {  
3         int j = i*1337/42;  
4     }  
5     return x*x;  
6 }  
7
```



Share Policies Other

x86-64 clang 9.0.0 (Editor #5)

x86-64 clang 9.0.0 -O3

```
1 foo(int):  
2     mov     eax, edi  
3     imul  eax, edi  
4     ret
```

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

```
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 }
```

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 }
```

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
```

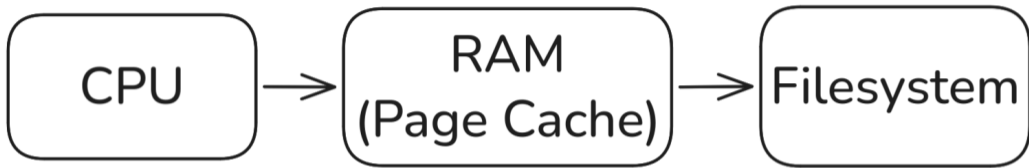
Proof: Which is faster

Let's learn why!

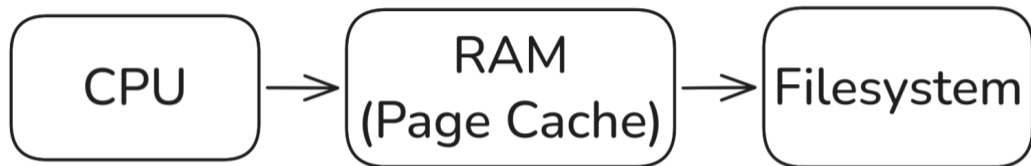
How I/O is taught



How I/O more works like

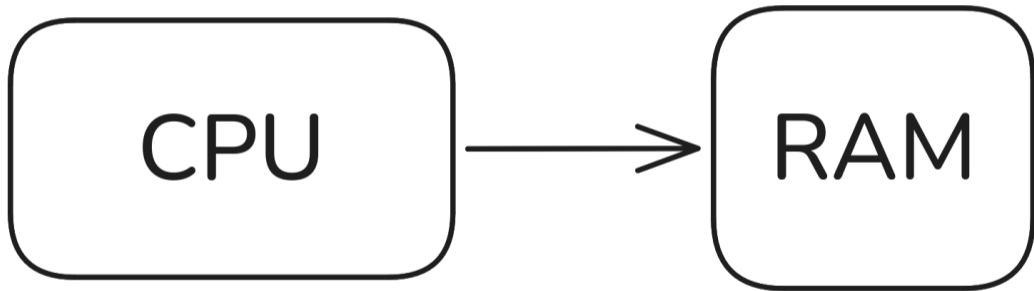


How I/O more works like

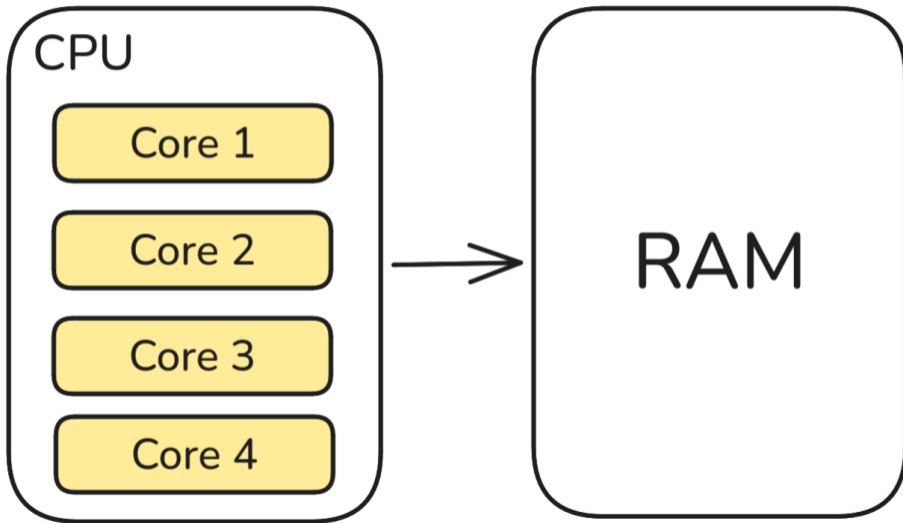


So... RAM is fast right?

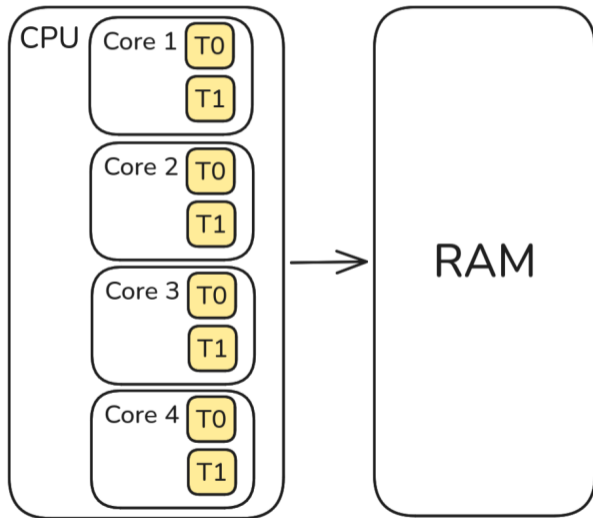
How CPUs are taught (Variable Access)



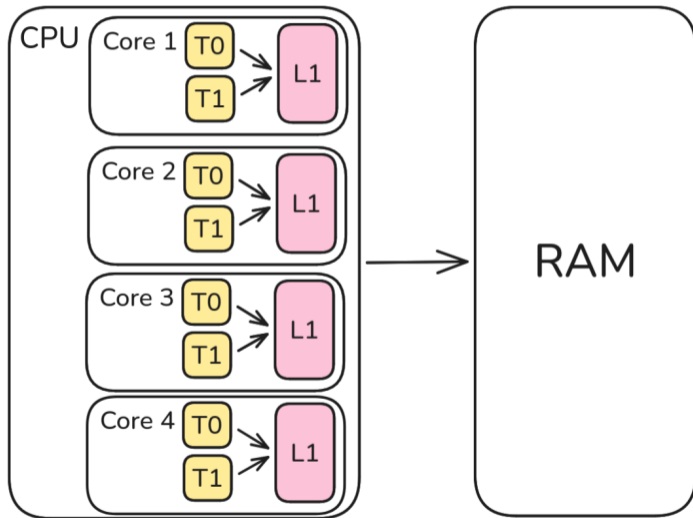
How CPUs actually are



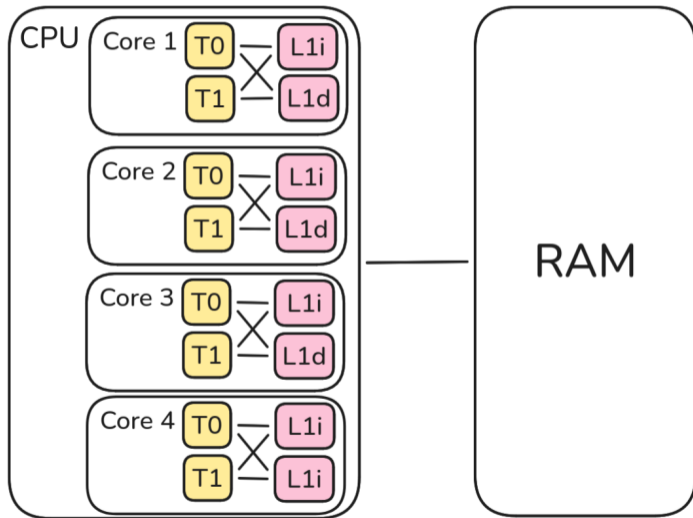
How CPUs actually are (if you are pedantic)



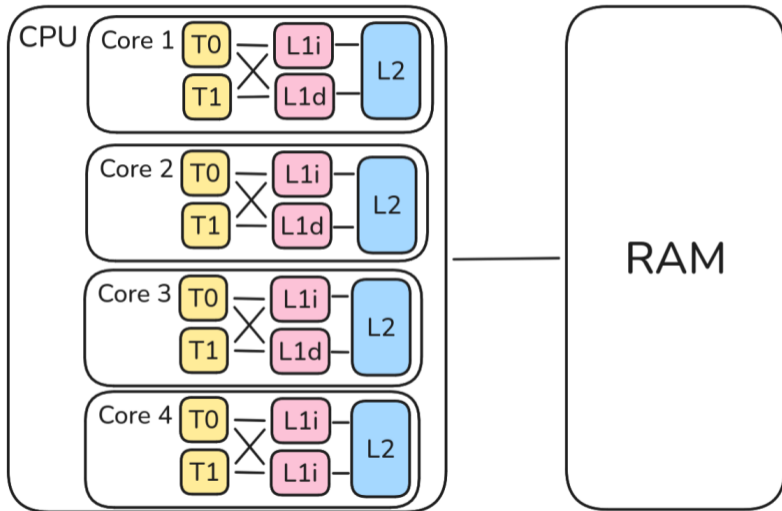
How CPUs *actually* are



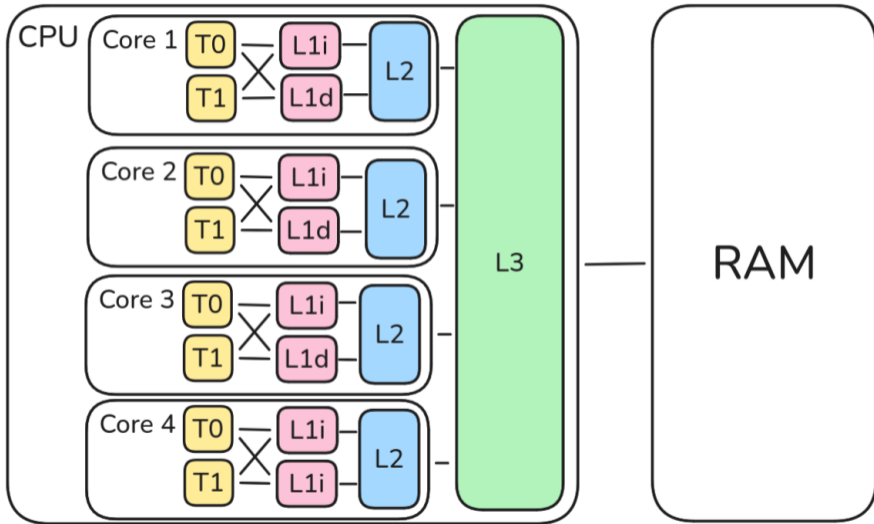
How CPUs **actually** are



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
- ⇒ Spectre & Meltdown

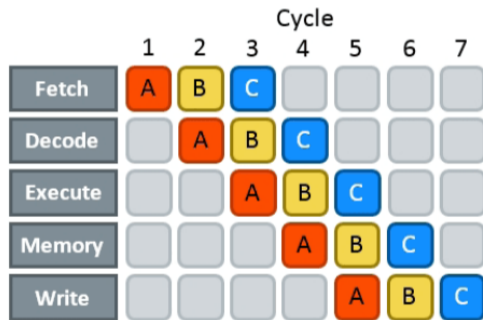
Instruction Pipelining

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
- ⇒ 1 instruction **per cycle**
- No work needed!



Exemplary Pipeline [2]

Pipeline Hazards

■ Structural Hazard

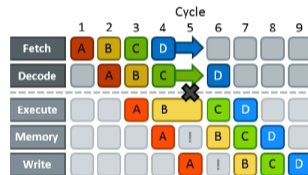
- ▶ Two instructions need the *same* unit
- ▶ Stall until unit is free (~1 cycle)
- ▶ Not really an easy fix

■ 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 \Rightarrow 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
- \Rightarrow slower or broken on some architectures
- Avoid unless you really know why

Struct Padding: Bad Example

```
C
1  struct Bad {
2      char    a;        // 1 byte
3                          // 3 bytes padding
4      int     b;        // 4 bytes
5      char    c;        // 1 byte
6                          // 7 bytes padding
7      double  d;        // 8 bytes
8      short   e;        // 2 bytes
9                          // 6 bytes padding
10 };
11 // Total: 24 bytes (only 16 bytes of actual data!)
```

Struct Padding: Good Example

C

```
1 struct Good {
2     double d;    // 8 bytes
3     int    b;    // 4 bytes
4     short  e;    // 2 bytes
5     char   a;    // 1 byte
6     char   c;    // 1 byte
7 };
8 // Total: 16 bytes (zero padding!)
```

SIMD and Auto-Vectorization

What is SIMD?

- **Single Instruction, Multiple Data**
- 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 continuous**
- Clear loop conditions
- Pro-Tipp: Struct of Arrays (SoAs)
 - ▶ More on that later!

SIMD example (unoptimized)



COMPILER EXPLORER

Add... ▾ More ▾ Templates

Share ▾ Policies 🔔 ▾ Other ▾

C++ source #5

A ▾ C++ ▾

```
1 int array[32];
2
3 int main(int argc, char **argv) {
4     for (int i = 0; i < 32; i++) {
5         array[i] = i;
6     }
7 }
8
```

x86-64 clang 9.0.0 (Editor #5)

x86-64 clang 9.0.0

-O1

A ▾

```
1 main:                                # @main
2     xor     eax, eax
3     .LBB0_1:                            # =>This Inn
4     mov     dword ptr [4*rax + array], eax
5     add     rax, 1
6     cmp     rax, 32
7     jne    .LBB0_1
8     xor     eax, eax
9     ret
10 array:
11     .zero 128
```

SIMD example (optimized (512-bit))

COMPILER
EXPLORER

Add... ▾ More ▾ Templates

Share ▾ Policies ▾ Other ▾

C++ source #5

A ▾ C++ ▾

```
1 int array[32];
2
3 int main(int argc, char **argv) {
4     for (int i = 0; i < 32; i++) {
5         array[i] = i;
6     }
7 }
8
```

x86-64 clang 9.0.0 (Editor #5) X

x86-64 clang 9.0.0 ▾

-march=skylake-avx512 -O3 ▾

A ▾

```
25     .long    22                # 0x16
26     .long    23                # 0x17
27     .long    24                # 0x18
28     .long    25                # 0x19
29     .long    26                # 0x1a
30     .long    27                # 0x1b
31     .long    28                # 0x1c
32     .long    29                # 0x1d
33     .long    30                # 0x1e
34     .long    31                # 0x1f
35 main:
36     vmovaps  zmm0, zmmword ptr [rip + .LCPI0_0] #
37     vmovaps  zmmword ptr [rip + array], zmm0
38     vmovaps  zmm0, zmmword ptr [rip + .LCPI0_1] #
39     vmovaps  zmmword ptr [rip + array+64], zmm0
40     xor     eax, eax
41     vzeroupper
42     ret
43 array:
44     .zero   128
```

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?
- ⇒ Matters for: HFT, Audio, Real-Time

Throughput

- How many operations per second?
 - e.g. how many FLOPS?
- ⇒ 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 (~ 15-20 cycles)

The Hint

- Tell compiler common branch
- ⇒ Hot path stays linear
- ⇒ Better L1i cache usage
- ⇒ Less pipeline flushing

In Practice

- C++20: `[[likely]] / [[unlikely]]`
- GCC/Clang: `__builtin_expect`
- ALWAYS BENCHMARK BEFORE
- ⇒ Wrong hint is worse than no hint

Branch Optimization: Likely/Unlikely

C++

```
1  template <class T>
2  void process_data(const std::vector<T>& data) {
3      // If the sensor works, it can't be below zero
4      for (int value : data) {
5          if (value > 0) [[likely]] {
6              process_data_fast(value);
7          }
8          else [[unlikely]] {
9              process_error_slow(value);
10         }
11     }
12 }
```

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 ⇒ Firefox Build Server!
- ⇒ 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

The screenshot shows the Compiler Explorer interface. On the left, the C++ source code is displayed in a file named 'C++ source #5'. It defines a struct 'ParticlesSoA' with three float arrays of size 1024 (x, y, z) and a function 'apply_wind()' that iterates over these arrays, adding 30.0f to the x component of each particle.

```
1 struct ParticlesSoA {  
2     float x[1024];  
3     float y[1024];  
4     float z[1024];  
5 };  
6  
7 ParticlesSoA particles;  
8  
9 void apply_wind() {  
10     for (int i = 0; i < 1024; i++) {  
11         particles.x[i] += 30.0f;  
12     }  
13 }
```

On the right, the assembly output for 'x86-64 clang 9.0.0' with the flag '-march=skylake-avx512 -O3' is shown. The assembly uses AVX-512 instructions to process the data in a SoA (Structure of Arrays) manner. It starts with a broadcast of the scalar value 30.0f into a 512-bit register (zmm0). Then, it uses 'vaddps' instructions to add this broadcasted value to the x-components of the float arrays in the 'particles' struct, processing 512 elements at a time. The assembly is highly optimized, using vector registers and pointer arithmetic to access memory efficiently.

```
1 .LCPI0_0:  
2     .long    1106247680          # float 30  
3 apply_wind():  
4     vbroadcastss    zmm0, dword ptr [rip + .LCPI0_0] # zmm0  
5     vaddps    zmm1, zmm0, zmmword ptr [rip + particles]  
6     vaddps    zmm2, zmm0, zmmword ptr [rip + particles+64]  
7     vaddps    zmm3, zmm0, zmmword ptr [rip + particles+128]  
8     vaddps    zmm4, zmm0, zmmword ptr [rip + particles+192]  
9     vmovaps   zmmword ptr [rip + particles], zmm1  
10    vmovaps   zmmword ptr [rip + particles+64], zmm2  
11    vmovaps   zmmword ptr [rip + particles+128], zmm3  
12    vmovaps   zmmword ptr [rip + particles+192], zmm4  
13    vaddps    zmm1, zmm0, zmmword ptr [rip + particles+256]  
14    vaddps    zmm2, zmm0, zmmword ptr [rip + particles+320]  
15    vaddps    zmm3, zmm0, zmmword ptr [rip + particles+384]  
16    vaddps    zmm4, zmm0, zmmword ptr [rip + particles+448]  
17    vmovaps   zmmword ptr [rip + particles+256], zmm1  
18    vmovaps   zmmword ptr [rip + particles+320], zmm2  
19    vmovaps   zmmword ptr [rip + particles+384], zmm3  
20    vmovaps   zmmword ptr [rip + particles+448], zmm4  
21    vaddps    zmm1, zmm0, zmmword ptr [rip + particles+512]  
22    vaddps    zmm2, zmm0, zmmword ptr [rip + particles+576]
```

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

```
1  # Compare two commands
2  hyperfine 'find . -name foo' \
3          'fd foo'
4
5  # Warmup + cold cache
6  hyperfine --warmup 3 \
7      --prepare 'sync' \
8      './my_program'
9
10 # Vary a parameter
11 hyperfine \
12     --parameter-scan n 1 8 \
13     './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

Cachegrind

What is it?

- Valgrind tool for cache simulation
 - Literally simulates a fake CPU
 - Thus: Not just statistical approximation
 - Exact, deterministic results
 - But: 20-100x slower than native!
 - ▶ Multithreading: Serialized; Timings are fully off!
- ⇒ Use for targeted cache investigation, not general profiling

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

```
tmp: bash — Konsole
New Tab Split View
Copy Paste Find...
~/tmp$ batcat test.c
File: test.c
1 struct Bad {
2   char a;
3   int b;
4   char c;
5   double d;
6   short e;
7 };
8 volatile struct Bad dont_optimize_me;
9 int main() {}

~/tmp$ gcc -g test.c -o test
~/tmp$ pahole test
struct Bad {
  char          a;           /*  0  1 */
  /* XXX 3 bytes hole, try to pack */
  int           b;           /*  4  4 */
  char          c;           /*  8  1 */
  /* XXX 7 bytes hole, try to pack */
  double        d;           /* 16  8 */
  short int     e;           /* 24  2 */
  /* size: 32, cachelines: 1, members: 5 */
  /* sum members: 16, holes: 2, sum holes: 10 */
  /* padding: 6 */
  /* last cacheline: 32 bytes */
};
~/tmp$
```

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> \(visited on 06/07/2026\).](https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html)
- [2] [Instruction-Level Parallelism - Algorithmica. URL: <https://en.algorithmica.org/hpc/pipelining/> \(visited on 06/08/2026\).](https://en.algorithmica.org/hpc/pipelining/)
- [3] [Pipeline Hazards - Algorithmica. URL: <https://en.algorithmica.org/hpc/pipelining/hazards/> \(visited on 06/08/2026\).](https://en.algorithmica.org/hpc/pipelining/hazards/)