CPU hardware architecture

Hadrien Grasland 2026-02-12

Did you say “architecture”?

- x86, ARM, POWER = *instruction set* architecture (ISA)
- Zen 5, Apple M4 = *micro*-architecture (μarch)

- ISA = **Contract** between binary code and the CPU
  - Public knowledge, shared across many chips
  - Backwards compatible, changes slowly (via extensions)

- μarch = Hardware **implementation** of the ISA
  - Semi-secret, varies across generations & chips
  - No compatibility, changes more quickly

Why study µarch?

- Needed to answer **performance questions**
  - How to compute more operations per second?
  - How to improve data access latency/throughput?
- **Variability is manageable**
  - Big design choices (e.g. caching, superscalar…) are stable
  - Orders of magnitude (e.g. L1 cache size) change slowly
  - Even different ISAs have similar implementations

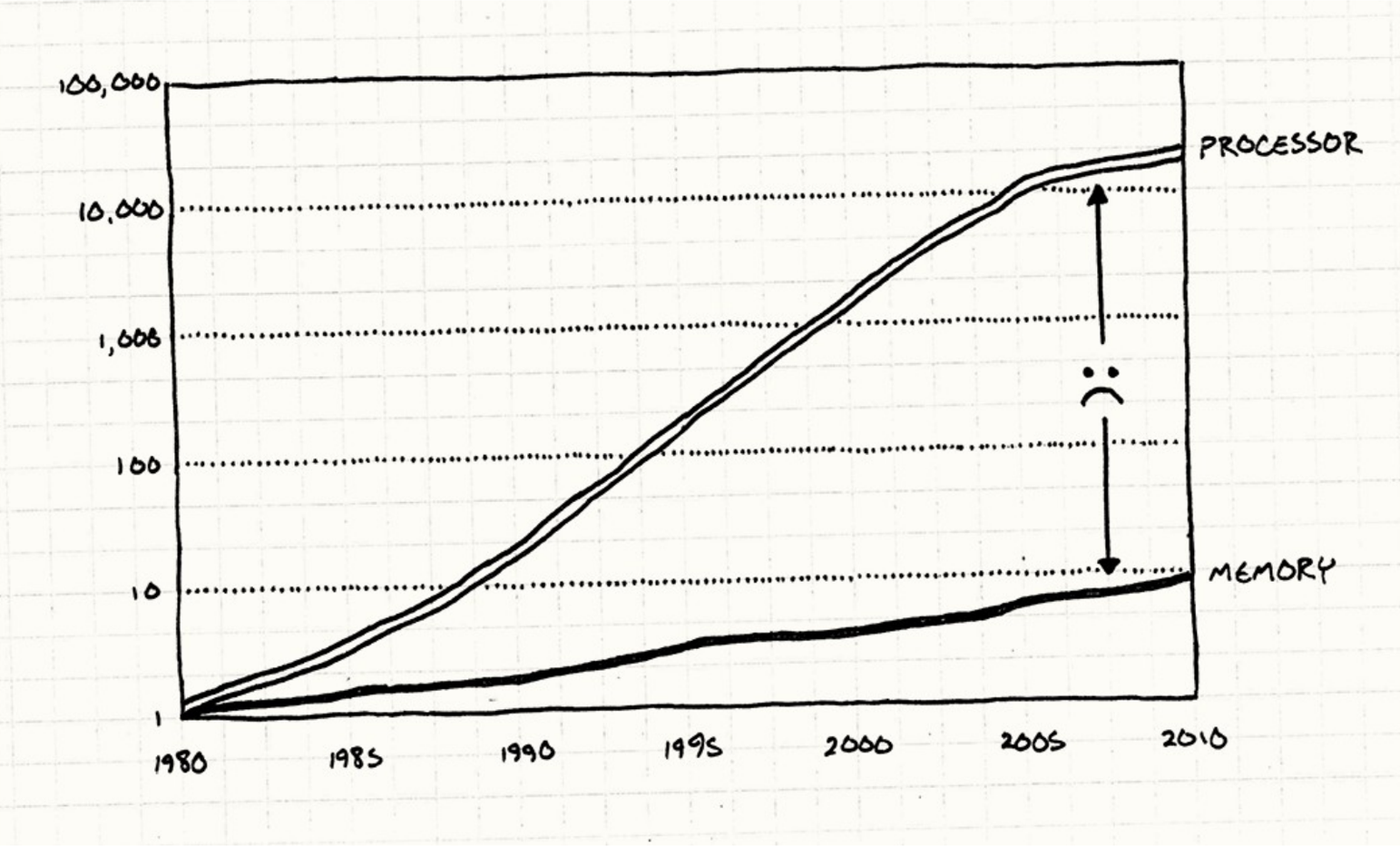
Memory access

Compute vs RAM performance

- Let's take a 2x AMD EPYC 7702 compute server from my lab
- How fast can it handle single-precision floats?
  - **Float arithmetic: 8.2 x 10^{12} f32 operations/sec**
    (2 sockets x 64 cores x 2 GHz x 2 FMA/cy x 16 operations/FMA)
  - **CPU-RAM bandwidth: 1.0 x 10^{11} f32 transfers/sec**
    (2 sockets x 204,8 GB/sec ÷ 4 B/f32)
- Meaning: On this CPU, below **~80 operations/RAM transfer***, CPU-RAM bandwidth is the performance limit!

* This metric, called "arithmetic intensity", can be used to guide optimization priorities.

…and it keeps getting worse*



http://gameprogrammingpatterns.com/data-locality.html

* Longer discussion by BLAS/LAPACK author: https://www.youtube.com/watch?v=cSO0Tc2w5Dg

Can I have faster memory?

- Computer storage balances many concerns
  - **(Non-)Volatility** (preserve data when off)
  - **Performance** (latency, bandwidth)
  - **Energy** efficiency
  - **Density**
  - **Price**

- A **memory hierarchy** gives us a bit of everything

Memory hierarchy example

- Our 2x AMD EPYC 7702 system contains many memories:

– Regs :	**cap.** 16x32B	**lat.** 0cy	**BW** ~unlimited
– L1d :	32 KiB	4-8cy	(2+1)x32B/core/cy
– L2 :	512 KiB	≥ 12cy	32B/core/cy
– L3 :	4 MiB/core	~39cy	32B/core/cy
– RAM :	~2 GiB/core	~10^2cy	1,6B/core/cy*
– SSD :	~1 TiB/**node**	~10^5cy	0,02B/core/cy**
– HDD :	~10 TiB/node	~10^7cy	0,001B/core/cy**
– NFS : See SSD/HDD + network limits + inter-node sharing			

Non volatile

* Each socket has 8 DDR4-3200 channels (204,8 GB/s), shared by 64 cores running at 2 GHz
** Assuming ~100µs lat, 4 GB/s BW for SSDs and ~10ms latency, ~200MB/s BW for HDDs

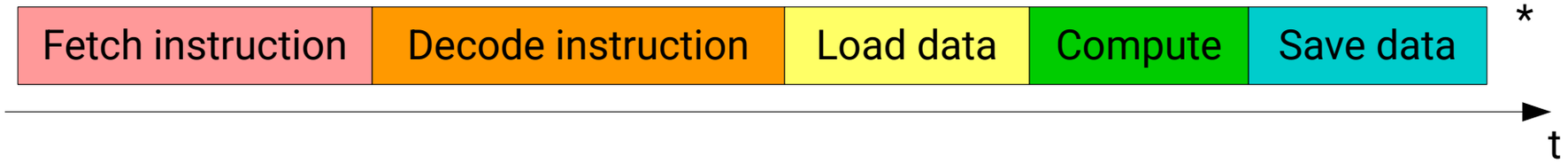
Know your caches

- **Spatial locality:** Caches work with aligned “lines” (~64 bytes)
  → Group data by usage, arrays of structs require care
- **Temporal locality:** New lines replace least-recently-used ones
  → Don’t wait too much before reusing data
- Aim for **exhaustive/linear access patterns**
  → Otherwise issues w/ HW prefetch, associativity, TLB…
  → Can reduce damage with padding, manual prefetch…
- **Beware shared writes** in parallel code (~RWlock per line)

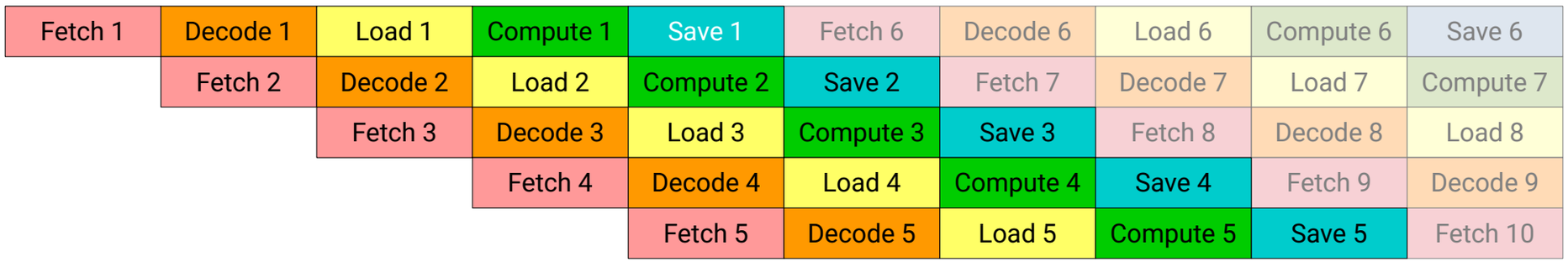
Concurrency & parallelism(s)

Pipelining

- Instruction execution is decomposed in **stages**



- Given enough HW parallelism, stage execution can overlap



- Ideal scenario: all stages run in parallel → **N_{stages} speedup**

* This is *highly* simplified, real CPUs are closer to 10-20 pipeline stages

Pipeline hazards

- **Branching** control flow (if/else, loop, `virtual`…) is a problem
  - How to fetch next instruction before outcome known?
  - CPU predicts & speculates → Wrong guesses are expensive
- **Data dependencies** (e.g. store → load) also problematic
  - CPU tries to reduce cost: bypass, out-of-order execution…
  - Dependency chains remain an issue (next slide)
- A lot more pipeline complexity* beyond scope of this talk

* Structural hazards, register renaming, memory consistency, Tomasulo algorithm, μops…

Out-of-Order execution

Out-of-Order execution

- Instructions have variable **latency** (time-to-result)
  - CPU starts more instructions as it waits
- Can't have **too many instructions in flight**
  - Many μarch limits: register file, load/store queue…
  - On embedded/HPC CPUs, the limits are smaller
- To avoid hitting those limits, be careful with…
  - Accesses to **high-latency memory** (L3, RAM…)
  - **Dependency chains:** (((a + b) + c) + d), x[y[i]], a->b->c…

SIMD aka vectorization

- Executing each instruction is a lot of work

  → Amortize with instructions that do more work?

- Enter **Single Instruction / Multiple Data**
  - e.g. x86 ADDPS does x[0:4] ← y[0:4] + z[0:4]

- Not so easy to use
  - **Few operations** with weird limitations
  - **Portability is hard**, even across a single ISA
  - **Sensitive to data layout**, abstract data with care

More instructions in parallel

Cheap hardware

- **Superscalar*:** N instructions from the same thread
  - Must be independent: (a + b) + (c + d) ≠ (((a + b) + c) + d)
- **SMT aka hyperthreading:** N threads on same compute units
  - Good for high-latency memory, blocking I/O…
  - Bad for optimized threads with same compute workload
- **Multicore:** N threads on separate compute units
  - Still share resources: last-level cache, RAM bandwidth…

Easy code

* Think “wider” pipelines that fetch/decode/execute/etc multiple instructions each cycle.

Cheat sheet

	Pipelining	SIMD	Superscalar	SMT	Multicore
Needs	Predictable control flow	Special instructions	Independent instructions	Threads w/ blocking/varied workloads	Threads w/ low shared resource pressure
Local gain	10-20x	2-8x (64-bit) 16-64x (8-bit)	2-8x	2x	4-200x
Tuning work	Medium	Easy (library) Medium (basic) Hard (other)	Easy	Hard (once rest optimized)	Medium (data/loop) Hard (tasks)
Strategy	• Profile to find mispredicts • Try coarser branching • Sort/group by code branch • Go branchless if you must	• Look for good 3rd-party libs • Else use HW abstraction • Best for low bit widths, simple computations • Sensitive to data layout	• Profile to find dependency chains • Break them into several sub-chains	• Rarely worth optimizing • Can slow down some parallel code • Control it via CPU pinning	• Data/loop parallelism is a lot easier • Not always your code (can be a layer above/below) • Core count varies *a lot*

Other topics

Float arithmetic

- Not all floating-point operations are the same*
  - ADD, SUB, MUL, FMA — 2 ops/cy — Fast SIMD
  - DIV, SQRT — ~6x slower — Slow SIMD
  - EXP, LOG — ~20x FMA — Painful SIMD**
  - SIN, COS — ~20x FMA — Painful SIMD**
  - ATAN — ~50x FMA — Painful SIMD**
- **Consequence:** Prefer simple operations, reuse inverses, use trigonometry to avoid e.g. sin(atan2(x, y))

* Measured on Intel i9-10900 CPU (Comet Lake, 2020), with L1 cache inputs → best-case!
** Not supported by standard libm, requires dedicated lib + slow hardware instructions (gather)

Hardware gotchas

- Other patterns can be **slow for non-obvious reasons**, e.g.
  - Subnormal floats (diffusion/exponential decay)
  - Read+write separated by multiple of 2^{12} bytes (Nd arrays)
  - Integer modulo, complex addressing modes…
- Suggested strategy:
  - Check perf annotate for slow instructions
  - (Intel-only) Check weird entries in toplev/VTune topdown
  - Read CPU vendor optimization guide carefully

Typical priorities

- Compilers don’t optimize **float computations** much
  - Changes the result, might break numerical stability
- Be extra-careful inside **innermost loops**
  - Aim for spatial/temporal locality, linear access patterns
  - Reduce branching (if, switch…), keep it predictable
  - Avoid polymorphism (breaks inlining, adds indirection…)
  - Break your dependency chains (see appendix)
- **System calls** are expensive* → Coarsen I/O and thread sync

* …and keep getting more expensive over time with e.g. Meltdown & Spectre mitigations.

To conclude

- Modern chips (both CPU and GPU) are very complex
  - Many mechanisms that affect performance
- Some code patterns are easier for the CPU
  - Work with your CPU ally, not against it
- You don’t need to sacrifice **portability**
  - x86 and ARM CPUs, GPUs… have a lot in common
  - Main differences: orders of magnitude + SIMD instructions
  - Can cover everyone with right abstractions/tunables

Resources

- "Performance analysis and tuning of modern CPUs" by Bakhvalov
- Hardware documentation
  - Datasheets like https://ark.intel.com/content/www/us/en/ark.html
  - https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
  - https://www.amd.com/en/support/tech-docs
- Independent experts
  - "Computer Architecture : A Quantitative Approach" by Hennessy & Patterson
  - https://www.agner.org/optimize/
  - https://en.wikichip.org + wikipedia pages like "List of AMD Ryzen processors"
  - https://uops.info/
- Specialized news
  - https://chipsandcheese.com/
  - https://www.nextplatform.com/

Thanks for your attention!

Break down dependency chains

- Let’s say you want to sum an array of floats
- Avoid single-accumulator loop:

```
float acc = 0.0;
for (size_t i = 0; i < N; ++i) acc += input[i];
```

- Use multiple accumulators instead:

```
std::array<float, M> accs { 0.0, 0.0, …, 0.0 };
for (size_t i = 0; i < N; i += M) {
  for (size_t j = 0; j < M; ++j) accs[j] += input[i+j];
}
// ...sum remainder, then sum accs...
```

* If you use the C++ STL, prefer std::reduce to std::accumulate… and check that it works.