High Performance Scientific Computing 1

Calcolo Scientifico ad Alte Prestazioni 1

Matteo Cicuttin, Fabio Vicini

Politecnico di Torino

February 28, 2024

What is High Performance Scientific Computing?

High Performance Scientific Computing (HPSC) is a multidisciplinary field

- Serial/Parallel processor architectures
- Programming languages
- Numerical algorithms
- Digital electronics
- High speed, low latency networking
- Cybersecurity

A whole 5-year university program could be filled with HPSC topics.

We will cover everything in 5 three-hour classes.

What is High Performance Scientific Computing?

High Performance Scientific Computing (HPSC) is a multidisciplinary field

- Serial/Parallel processor architectures
- Programming languages
- Numerical algorithms
- Digital electronics
- High speed, low latency networking
- Cybersecurity

A whole 5-year university program could be filled with HPSC topics.

We will cover everything in 5 three-hour classes. Just kidding :)

What do we do with HPC?

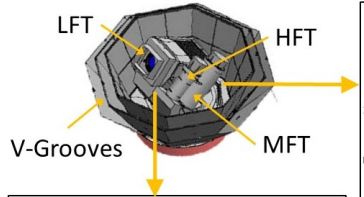
LiteBIRD is a satellite that will be launched in 2029. It will be used to observe the CMB.

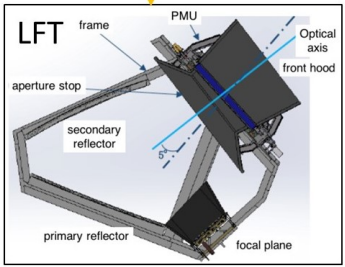
- Simulation of one of the LiteBIRD telescopes
- A couple of hours on 4 AMD MI250 GPUs

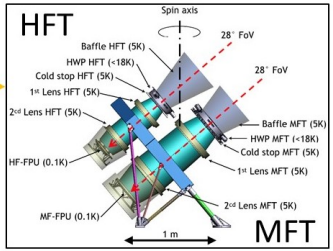
Click to play

LiteBIRD

LiteBIRD Payload Module







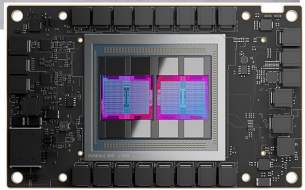
Main features:

- Three telescopes with TES detector arrays
- Frequency range 40 – 450 GHz
- Polarization modulation in each telescope with rotating half-wave plate (HWP) for 1/f noise & systematics reduction
- Cryogenic systems for 0.1 K detectors' temperature

LUMI supercomputer

ww.top500.org/lists/top500/2023/06/

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, **HPE** DOE/SC/Oak Ridge National Laboratory **United States**	8,699,904	1,194.00	1,679.82	22,703
2	**Supercomputer Fugaku** - Supercomputer Fugaku, [illegible] 2GHz, Tofu interconnect D, **Fujitsu** RIKEN Center for Computational Science **Japan**	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, **HPE** EuroHPC/CSC **Finland**	2,220,288	309.10	428.70	6,016



Course outline

In the era of Artificial Intelligence, this course is made to teach you that computers are actually very stupid machines with **definite limits and capabilities**. There's no magic. Santa Claus does not exist.

Objectives are to make you aware of those limits and give you some tools to maximally exploit computer capabilities.

This is a **brand new course** and the tentative program is the following:

- Class 1: Motivation, Anatomy of a computer (processor)
- Class 2: Anatomy of a computer (memory, bus, cache)
- Class 3: The limitations of a computer
- Class 4: Optimization I: basic techniques, simple linear algebra operations
- Class 5: Optimization II: advanced techniques, finite volumes code

This course will be focused on serial machines and algorithms: before going on big machines, you need to know how to go fast on your ordinary desktop PC.

Two important remarks

Don't be afraid to **stop us and ask**, there is **no stupid question**. Really. Any feedback is welcome.

Probably you won't be able to replicate the experimental results on your computers. **It's not your fault: the experimental setting is not the same**. Different machines do behave very differently.

Requirements

- Basic C++ knowledge
- Basic usage of Unix (WSL, Linux or macOS)
- A C++ compiler (g++ or clang++)
- `apt install libopenblas-dev libsilo-dev`
- LLNL VisIt `https://sd.llnl.gov/simulation/computer-codes/visit`
- For those of you more technically inclined: install Intel VTune `https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler-download.html`

– Motivations –

Why a good understanding of computers is a fundamental skill

The AXPY operation

BLAS stands for Basic Linear Algebra Subprograms. It is a set of standard functions implementing the most common operations occurring in scientific computing:

- Level 1: vector-vector operations, $O(n)$ complexity
- Level 2: matrix-vector operations, $O(n^2)$ complexity
- Level 3: matrix-matrix operations, $O(n^3)$ complexity

The AXPY operation

BLAS stands for Basic Linear Algebra Subprograms. It is a set of standard functions implementing the most common operations occurring in scientific computing:

- Level 1: vector-vector operations, $O(n)$ complexity
- Level 2: matrix-vector operations, $O(n^2)$ complexity
- Level 3: matrix-matrix operations, $O(n^3)$ complexity

One of the operations included in BLAS is the AXPY: $\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$ where $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^N, \alpha \in \mathbb{R}$.

```
for (size_t i = 0; i < N; i++)
    y[i] = alpha*x[i] + y[i];
```

Anatomy of AXPY

Assume to run AXPY on a vector of size N:

```
for (size_t i = 0; i < N; i++)
    y[i] = alpha*x[i] + y[i];
```

Observations:

- N cycles
- For each cycle: 1 multiplication and 1 addition

Total cost: $2N$ FLOPS (FLOP = FLoating point OPeration).

Let's measure how much time an AXPY takes and determine speed in FLOPS/s.

An hypotetical XNPY operation

We define XNPY as follows (XNPY is **NOT** part of BLAS). Let $\boldsymbol{x}, \boldsymbol{y}$ be two vectors of equal length:

$$y_i \leftarrow x_i^n + y_i \qquad i \in 1 \dots \text{length}(\boldsymbol{y})$$

Translated to code:

```
for (size_t i = 0; i < N; i++) {
    double xpow = 1.0;
    for (size_t p = 0; p < pow; p++)
        xpow *= x[i];
    y[i] = xpow + y[i];
}
```

For each **outer** cycle we do `pow+1` FLOPS.

Let's measure the speed of XNPY.

Any idea to improve AXPY?

Why AXPY is so slow compared to XNPY? Any idea to improve it?

```
for (size_t i = 0; i < N; i++)
    y[i] = alpha*x[i] + y[i];
```

Matrix multiplication

Let $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{R}^{N \times N}$. The matrix multiplication $\boldsymbol{C} = \boldsymbol{AB}$ is computed as $c_{ij} = a_{ik} b_{kj}$. This is probably the most important operation in scientific computing.

The code for the matrix multiplication looks like this:

```
for (size_t i = 0; i < N; i++)
    for (size_t j = 0; j < N; j++)
        for (size_t k = 0; k < N; k++)
            c[I(i,j)] += a[I(i,k)] * b[I(k,j)];
```

Total num of FLOPs is $2N^3$. Let's do some measurements!

Another matrix multiplication

We reorder the operations in our matrix multiplication:

```
for (size_t i = 0; i < N; i++)
    for (size_t k = 0; k < N; k++)       // swapped this
        for (size_t j = 0; j < N; j++)   // and this
            c[I(i,j)] += a[I(i,k)] * b[I(k,j)];
```

Total num of FLOPs is **again** $2N^3$. Let's do some measurements!

What happens to performance?

- Why the performance of XNPY is better than AXPY despite being more complex?
- Why parallel XNPY improves but parallel AXPY does not?
- Why swapping two cycles in matrix multiplication changes the execution time so much?
- More generally, how do you choose a computer to solve a specific problem?

What happens to performance?

- Why the performance of XNPY is better than AXPY despite being more complex?
- Why parallel XNPY improves but parallel AXPY does not?
- Why swapping two cycles in matrix multiplication changes the execution time so much?
- More generally, how do you choose a computer to solve a specific problem?

The answer to these questions is rooted deeply

- in the way computers work
- in the way compilers work

Some tales about floating point

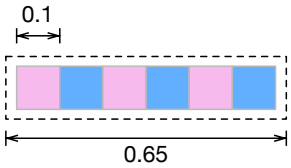
Before being fast, code must be correct.

Your code can go infinitely fast if you don't care about correctness, but people won't be interested in offering you a good job.

In scientific computing, code correctness comes also from hardware knowledge.

I - The missing block

How many blocks of width w_b fit in a space of width w_s? Of course $\lfloor w_s/w_b \rfloor$. For example, $\lfloor 0.65/0.1 \rfloor = 6$.




```
1 double total_width = 0.65;
2 double block_width = 0.1;
3 int blocks = (int) floor(total_width/block_width);
```

What happens with $w_s = 0.6$?

I - The missing block: the lesson

Computers have only **finite precision**: in base 2, the numbers $1/10$ and $6/10$ do not have a finite decimal expansion. Notice that in base 10 you have exactly the same problem with for example $1/3$.

- $\mathbb{F}(0.6) = 0.599999999999999978$.
- $\mathbb{F}(0.1) = 0.100000000000000006$.
- In general, $\mathbb{F}(x)$ can be **slightly smaller or slightly greater** than x and this can have really nasty effects.

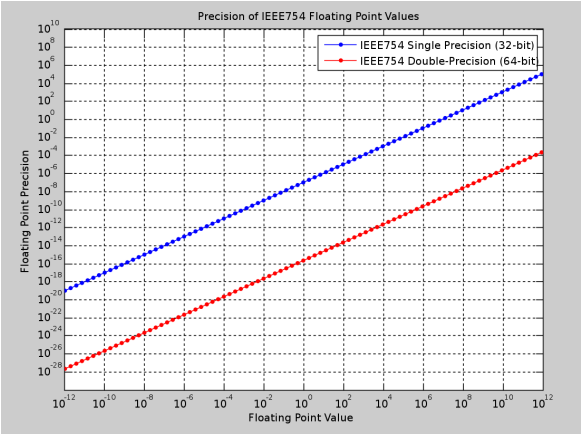
II - The odometer that stopped

Design an *odometer* that keeps track of the distance with a precision of 0.01 meters.

```
1  float meters;
2
3  void interrupt isr() {
4    if (odometer_interrupt)
5      meters += 0.01f;
6  }
7
8  int main(void) {
9    meters = 0;
10   while(1)
11     ;
12 }
```

II - The odometer that stopped: the lesson

Remember that in floating point **precision is not constant**: it depends on the magnitude of the number you are representing.



III - The Chaotic Bank Society

Your bank proposes to you to open a new account with the following scheme:

- When you open the account, you deposit $e - 1$ euros
- Each year the bank multiplies your savings by the number of years, but it takes a fee of 1 euro.

$$\begin{cases} b_0 = e - 1 \\ b_i = i \cdot b_{i-1} - 1 \end{cases}$$

You make a little program to figure out what happens after 25 years:

```
double account = 1.71828182845904523536028747135;
for (int i = 1; i <= 25; i++)
    account = i*account - 1;
```

Would you open the account?

III - The Chaotic Bank Society: the lesson

Let's unroll the series:

$$b_0 = e - 1 \qquad b_2 = 2b_1 - 1 = 2(1b_0 - 1) - 1$$
$$b_1 = 1b_0 - 1 \qquad b_3 = 3b_2 - 1 = 3(2(1b_0 - 1) - 1) - 1$$

You soon figure out that

$$b_n = n! \times \left(b_0 - 1 - \frac{1}{2!} - \frac{1}{3!} - \ldots - \frac{1}{n!} \right) = n! \times \left(b_0 - \sum_{i=1}^{n} \frac{1}{i!} \right)$$

And you recall that (Maclaurin expansion)

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

If $b_0 = e - 1$, then b_n should converge to zero! You can't represent $e - 1$ exactly, error is accumulating like crazy!

Conclusions

Knowing how a computer works allows you to

- recognize simple performance issues and not end up with super slow code
- recognize some potentially dangerous operations that will give you incorrect results

Modern engineering needs strong interdisciplinary skills, and numerical computing is a central tool.
Even if you only write small Matlab prototype programs, computer architecture somewhat affects you.

Also if you are a theoretical person, some computer architecture is needed: you could develop the best and most beautiful algorithm in the world, but if from a practical point of view is not good no one will use it.

– Part 1 –

What we find inside a computer

What is a computer?

- What is a computer in your opinion?
- What does a computer do?

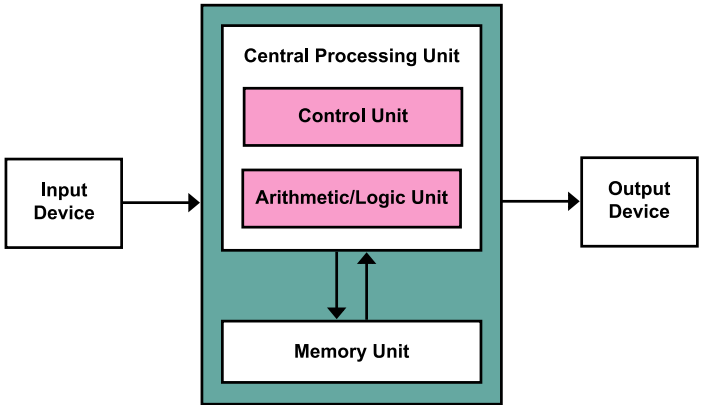
What is a computer?

A computer is a machine capable of executing very simple instructions. Typically they do not do much more than simple arithmetic, comparisons, data movement and jumps.

Factorial in x86 assembly language:

```
fact:                   ;
    mov     eax, 1      ; eax = 1
    cmp     edi, 0      ; if (edi == 0)
    je      fact_end    ;   go to fact_end
fact_loop:              ;
    imul    eax, edi    ; eax = eax * edi
    sub     edi, 1      ; edi = edi - 1
    jnz     fact_loop   ; if (edi != 0)
                        ;   go to fact_loop
fact_end:               ;
    ret                 ; return to caller
```

Von Neumann architecture



In this course we will study:

- Processor
- Memory
- Their interactions

Image from `https://en.wikipedia.org/wiki/Von_Neumann_architecture`

Processor: The data path

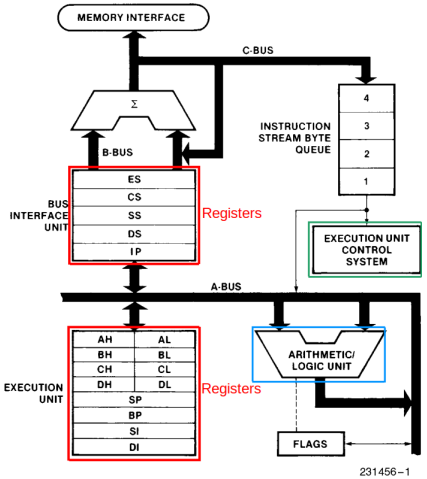


Figure 1. 8088 CPU Functional Block Diagram

Image from the Intel 8088 datasheet, document No. 231456.

- Registers: basic memory units in a computer system. Some are user-visible, some are not
- Arithmetic logic unit: does the (integer) math
- Control unit: makes everything work together
- Flags: Status bits about the result of the last operation (overflow, carry, zero, ...)

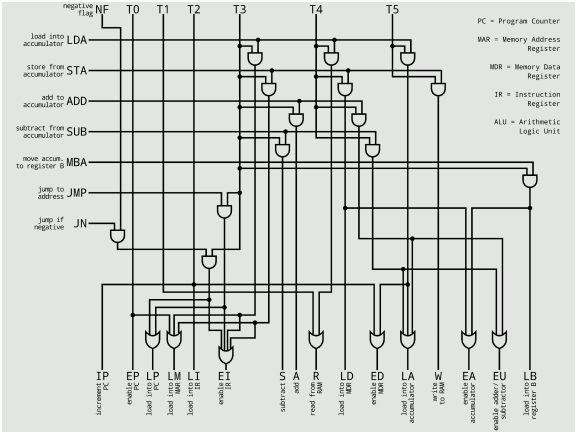
But who drives the flow of data in the data path?

Fetch-decode-execute

CPU runs instructions from a program stored in memory. A register usually called `PC` or `IP` points to the next instruction to execute. From startup to shutdown CPU does what is called **instruction cycle**:

1. **Fetch**: retrieve the next instruction from memory
2. **Decode**: figure out the type of instruction (memory access and if is direct/indirect, arithmetic operation, jump, ...)
3. **Execute**: Activate the appropriate units of the CPU, for example the ALU for an arithmetic instruction. Possibly modify `IP` if it was a jump
4. **Repeat**

Instruction cycle in a toy CPU



- Example instruction cycle (total 6 clock cycles).
- Decode phase not really there: only fetch and exec. All the details in the course repo.
- Only in extremely simple CPUs the CU is hardwired. Tipically they are microcoded.

This toy CPU executes one instruction per instruction cycle or one instruction every 6 clock cycles.

Toy CPU from `https://www.cs.binghamton.edu/~reckert/hardwire3new.html`
Image from `https://en.wikipedia.org/wiki/Control_unit`

Pipelining

The Fetch-Decode-Execute phases involve different hardware:

- when fetching, decode & execute not used
- when decoding, fetch & execute not used
- when executing, fetch & decode not used

It is possible to overlap the three phases (think about car assembly line) → **pipeline**.

Now we can **potentially** execute one instruction per **clock** cycle.

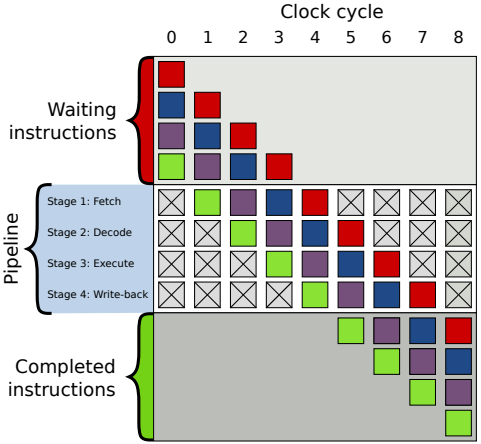


Image from `https://en.wikipedia.org/wiki/Pipeline_stall`

Superscalar architectures

If one pipeline is good, two is better. For example, the Pentium (P5 μarch) processor was superscalar.

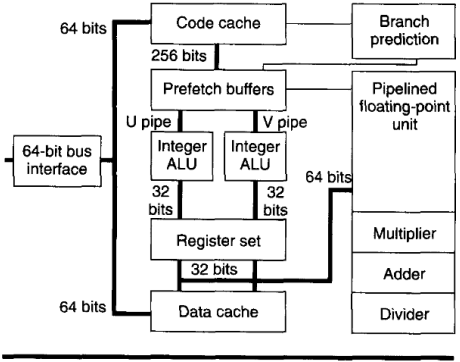


Figure 2. Pentium processor block diagram.

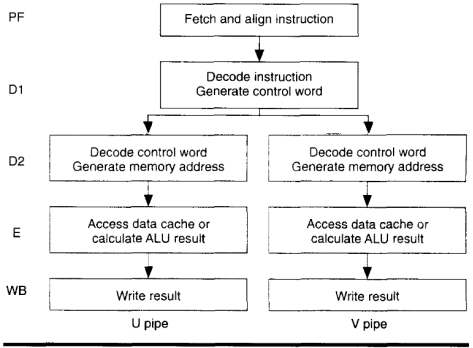


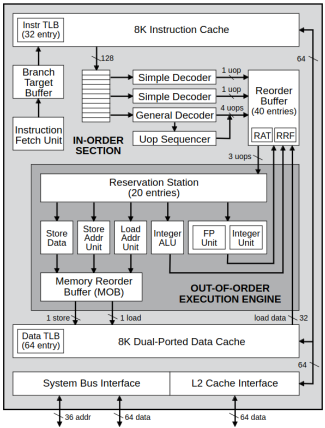
Figure 4. Superscalar execution.

Superscalar = more than scalar but not yet vector. Not a synonym of hyperthreading or multicore!

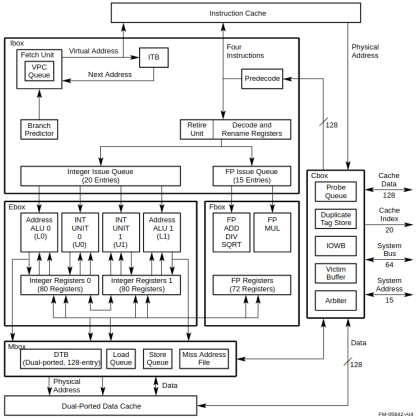
Architecture of the Pentium microprocessor, D. Alpert, D. Avnon, IEEE Micro

Out of order and speculative execution

Intel P6 μarch, from Pentium Pro to Pentium 3



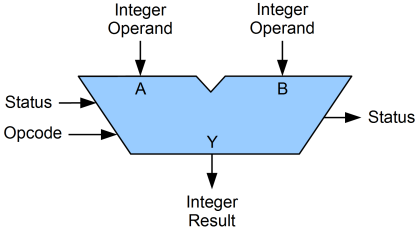
Alpha 21264



All modern high performance CPUs are superscalar, out of order designs.

Vector ALU and FPU

Modern processors feature vectorial Arithmetic Logic Units (ALU) and Floating Point Units (FPU).



- Classical ALUs/FPUs: 1 instruction $\implies$ 1 scalar operation (`add ax, bx`)
- Vector ALUs/FPUs: 1 instruction $\implies$ 1 operation on many scalars (`vaddpd ymm0, ymm1, ymm2`)
- On x86 CPUs: SSE, AVX, AVX2, AVX512. On ARM: NEON. On POWER: Altivec.

Vector units normally support very small vectors, sizes between 2 to 16 depending on the data type.

- Later in this course we will talk about AVX.

Interface to the outside world

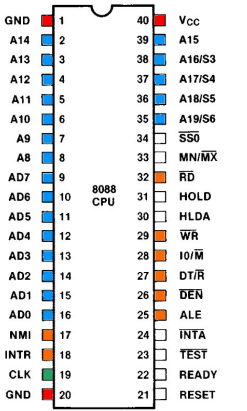


Figure 2. 8088 Pin Configuration

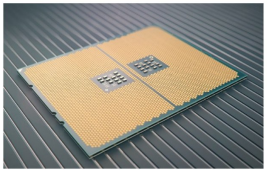
- Power: without power the CPU won't work :)
- Data and address signals: `D0...D7` are the data I/O pins, `A0...A19` are the address pins $\implies$ where I want to read data from
- Control signals: ask the memory to read or write, interrupts...
- Clock: drives all the operations inside the CPU

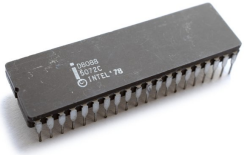
The 8088 is a CPU from 1979. At power-on, modern CPUs appear **exactly** like an 8086/8088: in order to use all their features, the operating system has to switch them into **protected mode** (32 bit) or **long mode** (64 bit).

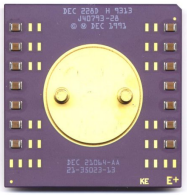
Image from the Intel 8088 datasheet, document No. 231456.

Some CPUs


intel
XEON
4th Gen
Intel Xeon Scalable Processor



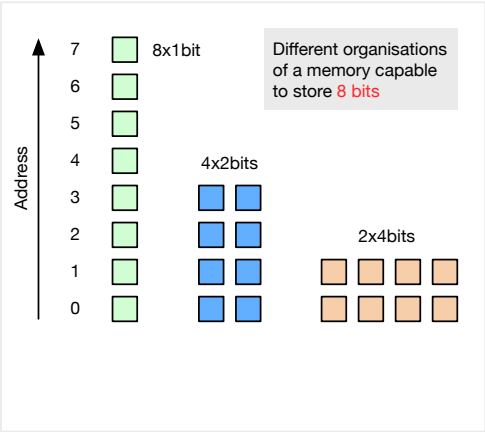

INTEL '78




Sun
Ultra SPARC III
D1219879 0206
PG2.2 980 USA
SME 10528 LGA
7154
-1050
EEC

The memory

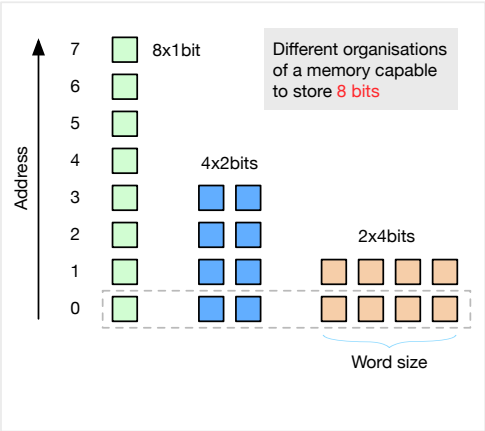
The memory holds 0s and 1s, which we call bits (binary digits). Nothing more.



- You can arrange your bits in various ways however...

The memory

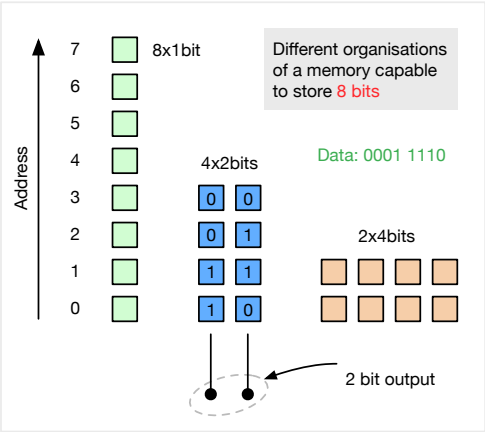
The memory holds 0s and 1s, which we call bits (binary digits). Nothing more.



- You can arrange your bits in various ways however...
- And this determines the **word size** of your memory

The memory

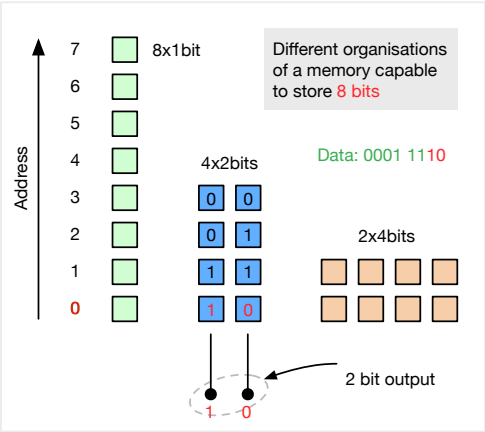
The memory holds 0s and 1s, which we call bits (binary digits). Nothing more.



- We consider for example the 4x2bit layout: memory has 2 byte output.

The memory

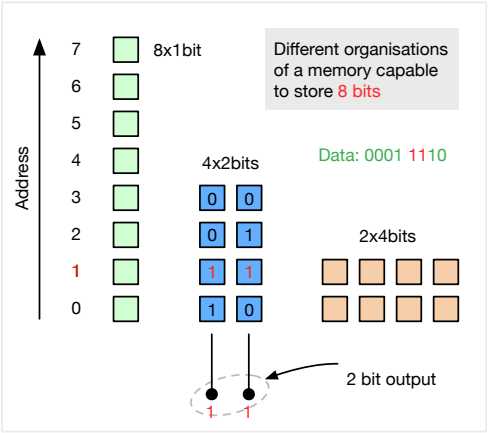
The memory holds 0s and 1s, which we call bits (binary digits). Nothing more.



- We consider for example the 4x2bit layout: memory has 2 byte output.
- The word we're interested in is selected by specifying an address.

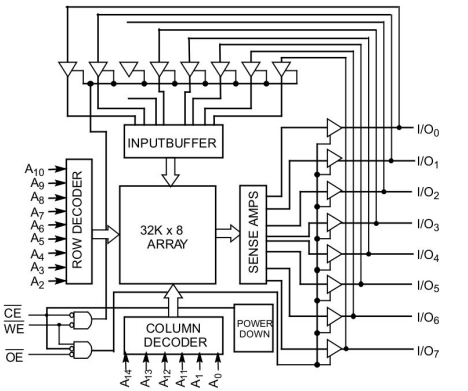
The memory

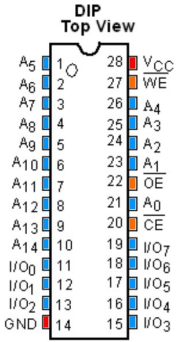
The memory holds 0s and 1s, which we call bits (binary digits). Nothing more.



- We consider for example the 4x2bit layout: memory has 2 byte output.
- The word we're interested in is selected by specifying an address.

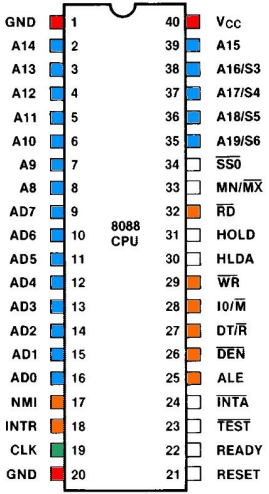
A memory from the '80s: the 62256





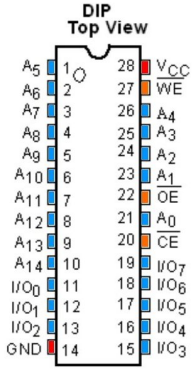
256 kbits, layout is 32k by 8bits: notice that we have exactly 15 address pins (`A0...A14`)

Connecting the memory and the CPU



231456–2

Figure 2. 8088 Pin Configuration



The system bus of the 8088

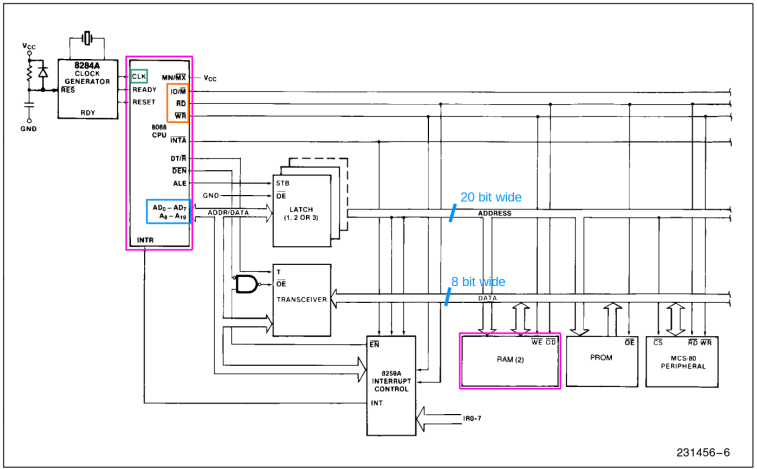
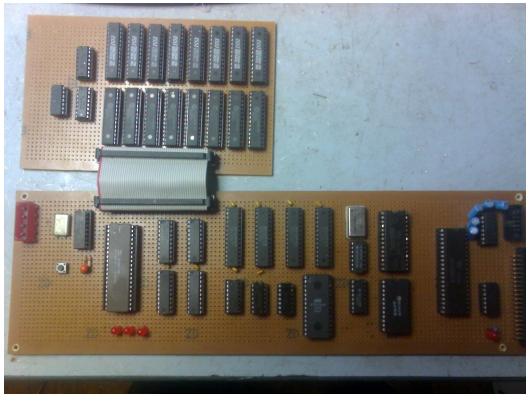


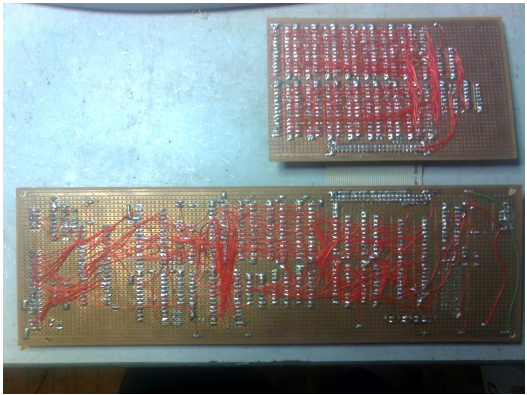
Figure 6. Demultiplexed Bus Configuration

- For each bus operation, one byte gets transferred

The bus of an homemade computer

CPUs, memories and buses are not something abstract, they do actually exist! /s





From: https://kaput.retroarchive.org/8088.html

Double width, double bandwidth: the bus of the 8086

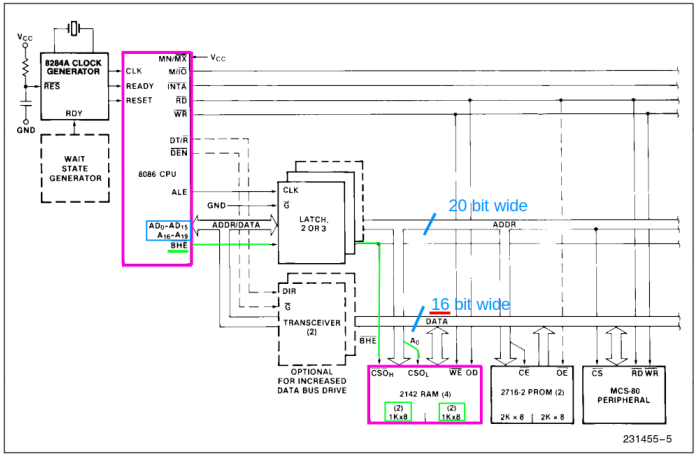
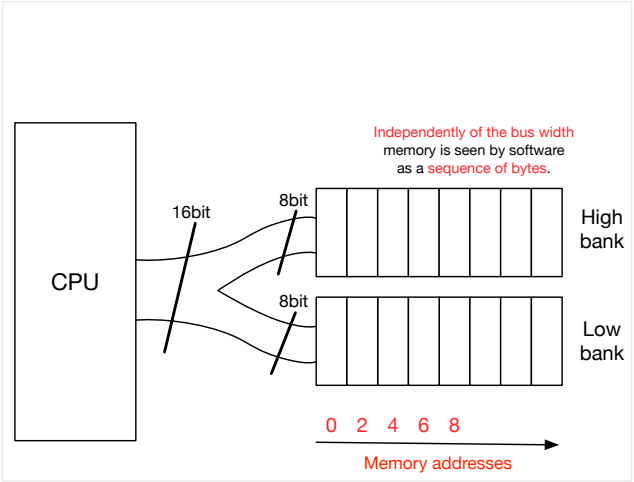


Figure 4a. Minimum Mode 8086 Typical Configuration

- For each bus operation, two bytes gets transferred

Misaligned memory access

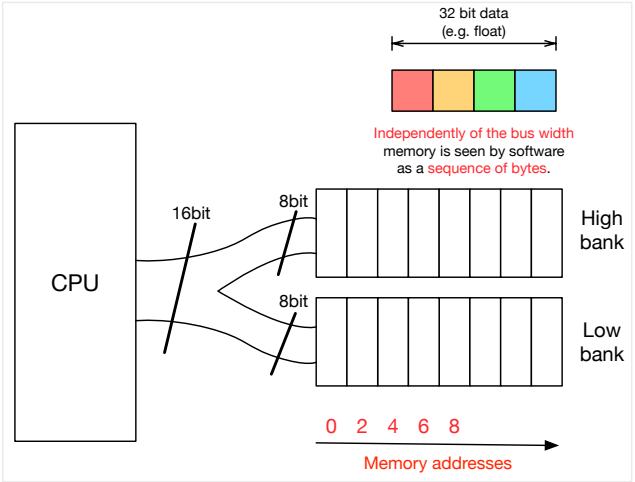


Different HW organization don't change the fact that memory is a sequence of bytes.

In our 8086 system:

- Low bank stores bytes at even addressess
- High bank stores bytes at odd addressess

Misaligned memory access

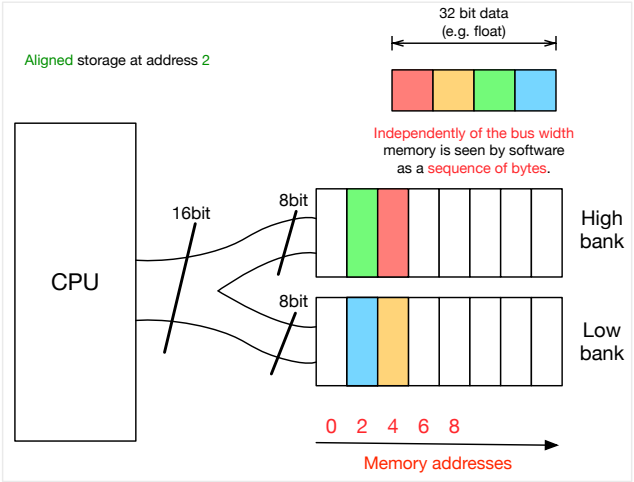


It is OK to store data at any address then?

Imagine we have a single precision floating point number (4 bytes)

- Least Significant Byte is the blue one
- Most Significant Byte is the red one.

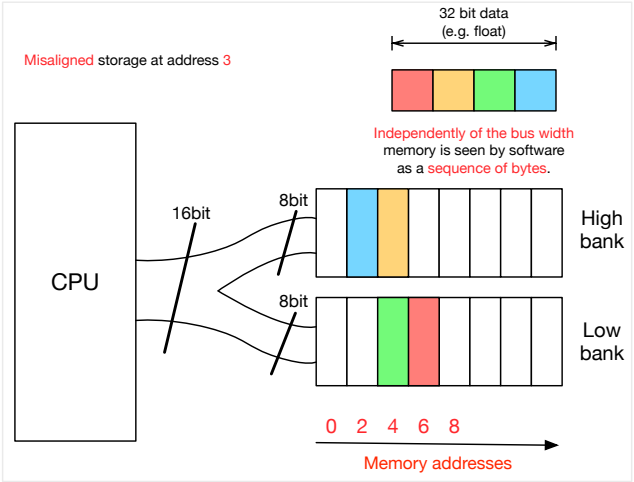
Misaligned memory access



If our float is stored aligned:

- First bus cycle: read byte 0 and byte 1
- Second bus cycle: read byte 2 and byte 3

Misaligned memory access

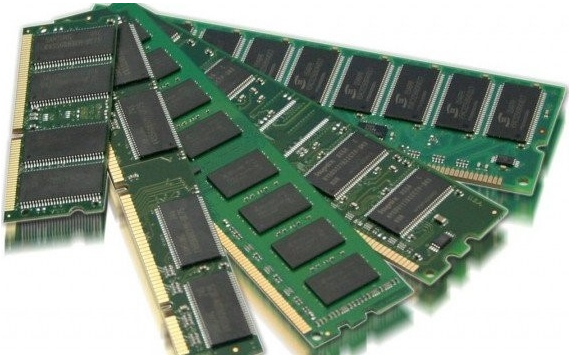


If our float is stored misaligned:

- First bus cycle: read byte 0, throw away half of what you read
- Second bus cycle: read byte 1 and byte 2
- Third bus cycle: read byte 3, throw away half of what you read

On certain architectures misaligned access can be extremely slow or completely disallowed. MIPS for example requires you to use specific instructions if you insist to do unaligned loads/stores.

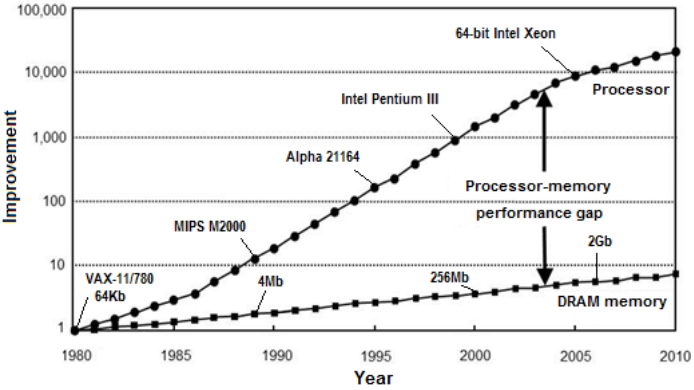
Memory in a modern PC

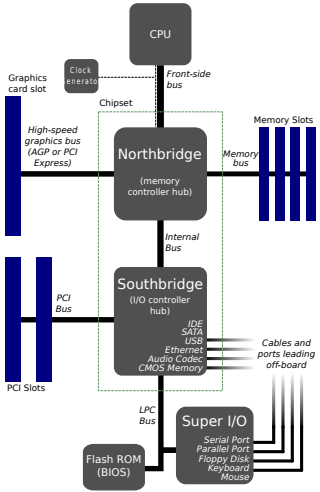


- From the original Pentium onwards, data bus width in PCs is 64 bit (yes, even in 32 bit architectures)
- On the 32 bit machines, address bus is 32 bits
- On 64 bit machines, even if you have 64 bit pointers, physical address bus width is anywhere between 39 and 48

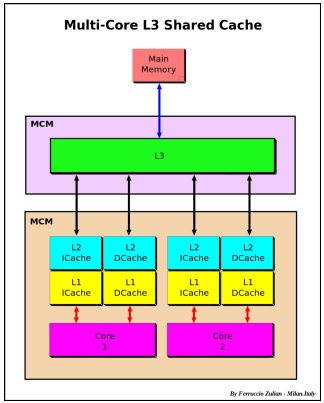
The different evolution of CPUs, memories and other peripheals

CPUs and memories evolved at different speeds. This gave rise to layered architectures.





The need for cache memory



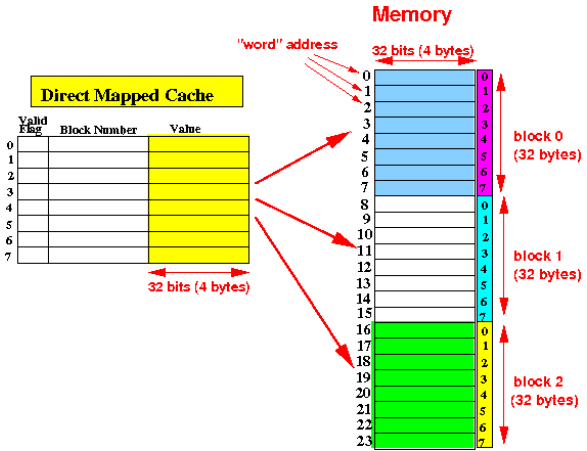
How to handle speed difference between CPU and memory?

- Cache: small but fast memory
- Locality principle suggests to introduce caches
- Algorithms must be cache-aware, otherwise you are going to waste resources
- Knowledge of CPU-memory interactions and caching mechanisms is required to develop efficient algorithms

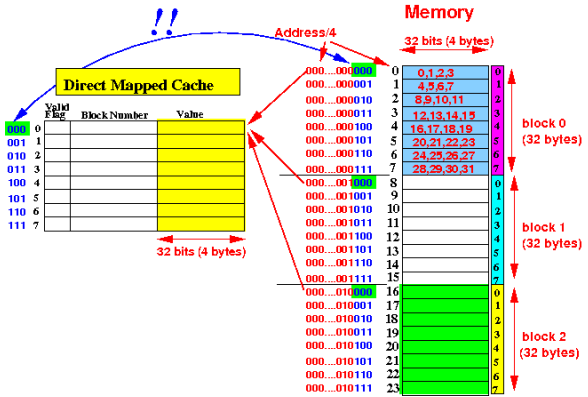
On modern machines:

- L1: 32-128 kB, $\mathcal{O}$(1 TB/s)
- L2: 512 kB, $\mathcal{O}$(1 TB/s)
- L3: 6-128 MB, $\mathcal{O}$(400 GB/s)
- Main memory: 8-128 GB, $\mathcal{O}$(100 GB/s)

A simple, direct-mapped cache

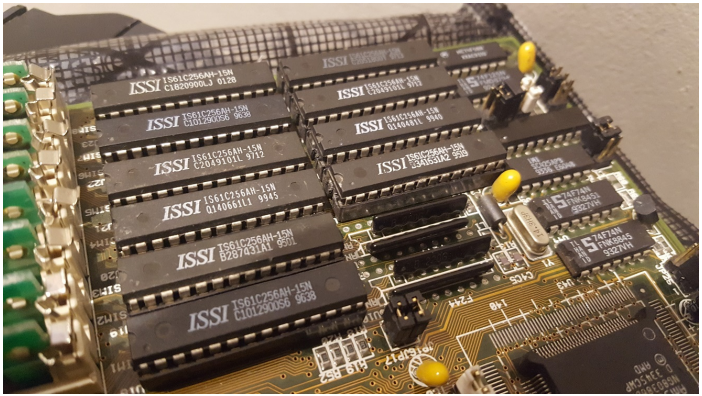


A simple, direct-mapped cache



Physical cache memory

The last machines to have a user-visible cache memory where perhaps the 80486. On modern machines cache is inside the CPU chip.



– Part 2 –

The limits of a computing architecture

Understanding memory specifications

Most common memories from the last 20+ years are called DDR (Double Data Date). We are currently at the 5th generation, DDR5.

- Data is transferred both the rising and falling edge of the clock signal
- Bus width for DDR memories is 64 bit (8 bytes)

There are two differents way to specify speed, the DDR labelling and the PC labelling

- DDRx-yyyy where x is the generation and yyyy the speed in MT/s
- PCx-yyyy where x is the generation and yyyy the speed in MB/s

Example: DDR3-1600 is equivalent to PC3-12800

Determining memory topology in a given machine

Depending on the machine you are on, memory can be connected to CPU in different ways.

- On desktop: tipically 2 or rarely 3 channels, 4 channels on high end i9 and Threadripper
- On servers: $\geq$ 2 channels, up to 12 with newest NUMA topologies (HPSC 2)

How to determine the topology:

- $\Longrightarrow$ Where to find data on Intel & AMD sites
- Demo of the `dmidecode` and `lstopo` utilities.

Computing theoretical memory bandwidth

To know the maximum theoretical speed of your memory subsystem you use the following formula:

$$\text{DDR rating} \times 8 \times \#\text{ of channels}$$

- For example, a dual-channel system with a DDR3-1600 memory can theoretically achieve

$$8 \text{ byte} * 3200 \text{ MT/s} * 2 \text{ channels} = 25600 \text{ MB/s}$$

- Pay attention on server machines: typically max memory bandwidth is reached only by parallel applications

Measuring actual memory bandwidth

The quick & dirty way to measure the memory speed of your system is to measure how much time `memcpy()` and `memset()` take.

- Buffer size must be chosen appropriately to avoid cache effects
- Be sure to do multiple measurements

Let's analyze a simple benchmark code $\Longrightarrow$ `code/perf/membench.cpp`.

If everything is working ok you should reach 70%/80% of the theoretical bandwidth.

Back to AXPY - I

Recall the AXPY operation we introduced at the beginning:

```
for (size_t i = 0; i < N; i++)
    y[i] = alpha*x[i] + y[i];
```

We measured 1.67 GFLOPS/s. But was it the right performance number?

Let's try to evaluate the memory bandwidth $\implies$ `code/perf/axpy.cpp`.

Back to AXPY - II

We measured around 20 GB/s...

```
for (size_t i = 0; i < N; i++)
    y[i] = alpha*x[i] + y[i];
```

Let's count:

- sizeof(double) is 8 bytes, plus we do 3 memory transfers (2 read + 1 write) per cycle
- We do 2 operations per cycle (+ and *)
- This means that to do the two operations we need to transfer 24 bytes

Back to AXPY - II

We measured around 20 GB/s...

```
for (size_t i = 0; i < N; i++)
    y[i] = alpha*x[i] + y[i];
```

Let's count:

- sizeof(double) is 8 bytes, plus we do 3 memory transfers (2 read + 1 write) per cycle
- We do 2 operations per cycle (+ and *)
- This means that to do the two operations we need to transfer 24 bytes

$$20\ [\text{GB/s}] \times 2/24\ [\text{FLOPS/byte}] = 1.67\ \text{GFLOPS/s}$$

AXPY is **limited by memory bandwidth**. You **can't** do more GFLOPS/s than this. GFLOPS/s is the **wrong** metric to evaluate **this** operation.

Understanding CPU specifications

Good Luck!

The bare minimum you need to know

- `cat /proc/cpuinfo`

The CPU naming almost always follows commercial criteria and has little relation to CPU performance. We are looking for:

- Base frequency
- SIMD width, or if the ALU can work with vectors. Intel CPUs: SSE = 2, AVX = 4, AVX512 = 8
- Number of arithmetic units (typically 2)
- Number of **cores**[1] (not threads!)
- FMA capability: Fused Multiply Add, the capability to compute $a * b + c$ with a *single instruction*. 1 = no, 2 = yes.

Architecture whitepapers usually contain this information.

[1] Hyperthreading is totally useless in scientific computing! Turn it off.

Computing theoretical CPU performance

Assume you have an i7-4790 CPU:

- Base freq: 3.6 GHz
- Total cores: 4
- SIMD width: 4 (AVX2)
- FMA: yes, on Intel (not AMD) cpus AVX2 implies FMA $\rightarrow$ 2
- Number of arith units: 2

Therefore:

$$3.6 \times 4 \times 4 \times 2 \times 2 = 230.4 \text{ GFLOPS/s (57.6 per core)}$$

A partial reference: `https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Core-Processors.pdf`

Measuring actual CPU performance

This measurement is tricky and highly problem-dependent, so there is no unique answer.

For scientific computing needs: take the best GEMM[2] implementation you have for your machine and measure its running time on sufficiently large matrices. Sorry to disappoint you!

$\Longrightarrow$ `code/perf/cpubench.cpp`

[2] GEMM stands for GEneral Matrix Multiply and is one of the Level 3 operations included in BLAS

Measuring actual CPU performance

This measurement is tricky and highly problem-dependent, so there is no unique answer.

For scientific computing needs: take the best GEMM[2] implementation you have for your machine and measure its running time on sufficiently large matrices. Sorry to disappoint you!

$\Longrightarrow$ `code/perf/cpubench.cpp`

We got 41.85 GFLOPS/s, $\approx$ 72.7 % of theoretical max.

[2] GEMM stands for GEneral Matrix Multiply and is one of the Level 3 operations included in BLAS

Back to XNPY

Recall XNPY: we defined something equivalent to

```
for (size_t i = 0; i < N; i++) {
    double xpow = 1.0; double xi = x[i];
    for (size_t p = 0; p < pow; p++)
        xpow *= xi; /* No FMA, half peak perf! */
    y[i] = xpow + y[i];
}
```

For each **outer** cycle:

- 2 reads and 1 write from memory (24 bytes total)
- 1 operation in the outer cycle plus `pow` operations in inner cycle
- Total: (`pow`+1) FLOPS / 24 bytes

$$20 \text{ GB/s} \times (\texttt{pow}+1)/24 \text{ [FLOPS/byte]} = 5(\texttt{pow}+1)/6 \text{ GFLOPS/s}$$

The higher the value of `pow`, the better the performance. $\implies$ `code/perf/xnpy.cpp`

Memory bound and CPU bound

In the simplest model, without hierarchical memory

- Memory bound: the computation is limited by the speed of memory
- CPU bound: the computation is limited by the speed of the CPU

AXPY is memory bound. XNPY memory bound for low `pow` and CPU bound for high `pow`.

In other words:

- If your algorithm is memory bound, it is useless to buy a faster CPU
- If your algorithm is CPU bound, it is useless to buy a faster memory

KNOW YOUR PLATFORM AND YOUR ALGORITHMS!

The roofline model

For XNPY we computed

$$20 \text{ GB/s} \times (\texttt{pow}+1)/24 \text{ [FLOPS/byte]} = 5(\texttt{pow}+1)/6 \text{ GFLOPS/s}$$

- The quantity in red is called arithmetic intensity. We denote it with I.
- The quantity in green is the available memory bandwidth. We denote it with β.
- We also denote with π the peak performance.

The roofline model

For XNPY we computed

$$20 \text{ GB/s} \times (\texttt{pow}+1)/24 \text{ [FLOPS/byte]} = 5(\texttt{pow}+1)/6 \text{ GFLOPS/s}$$

- The quantity in red is called arithmetic intensity. We denote it with I.
- The quantity in green is the available memory bandwidth. We denote it with β.
- We also denote with π the peak performance.

The attainable performance of an algorithm is limited by

$$P = \min \begin{cases} \pi \\ \beta \times I \end{cases}$$

The roofline model

For XNPY we computed

$$20 \text{ GB/s} \times (\texttt{pow}+1)/24 \text{ [FLOPS/byte]} = 5(\texttt{pow}+1)/6 \text{ GFLOPS/s}$$

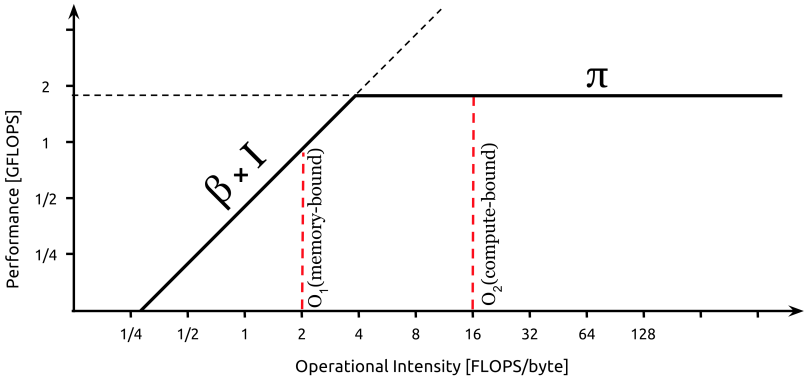
- The quantity in red is called arithmetic intensity. We denote it with I.
- The quantity in green is the available memory bandwidth. We denote it with β.
- We also denote with π the peak performance.

The attainable performance of an algorithm is limited by

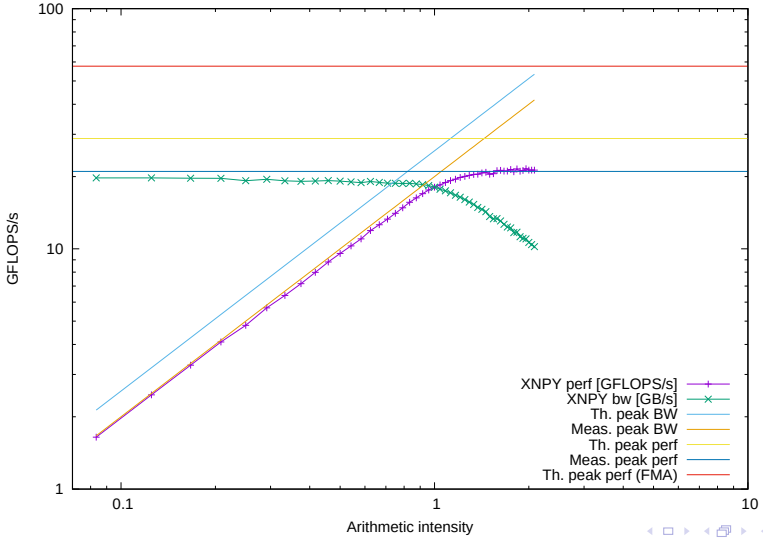
$$P = \min \begin{cases} \pi \\ \beta \times I \end{cases}$$

- $I < \beta/\pi$ memory bound
- $I > \beta/\pi$ CPU bound

The limits of your machine



XNPY performance



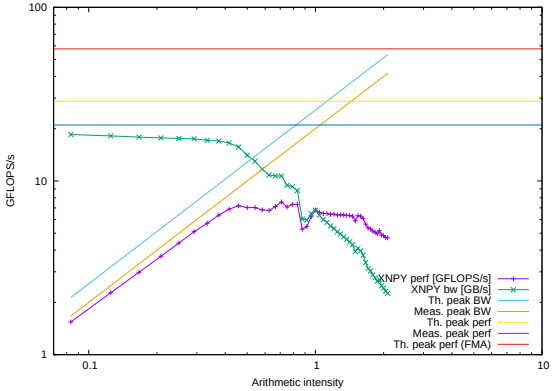
– Part 3 –

Making your code go fast(er): the basics

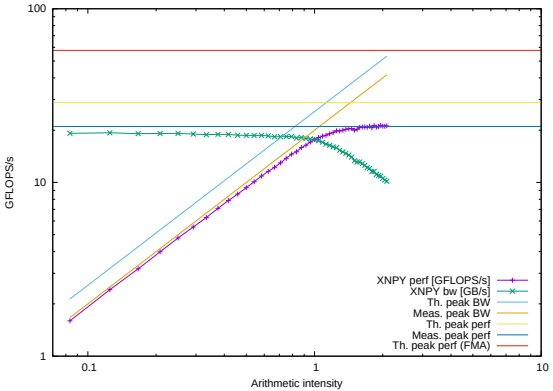
Optimization rules

The need for measurements

XNPY: There is a problem...



XNPY: No optimization needed/possible!



What to measure?

Whatever metric is important to your application. Can be time, memory bandwidth, cache misses...whatever.

Most sophisticated measurements are done with a profiler (later) and sometimes require hardware support.

On simple algorithms, already measuring just time can be a huge help.

Measuring time

The key to performance measurements is to accurately measure time. This is not obvious.

First thing to get right: cpu time vs. wall time. Approximately:

- cpu time is the time spent by a process on **all cpus**
- wall time is the time spent by a process from start to end

Example (simplified). A perfectly parallel algorithm running on 4 CPUs: t seconds wall time $\rightarrow 4t$ seconds CPU time.

Different interfaces available:

- `time()` (wall) and `clock()` (cpu). Available everywhere, not accurate.
- `gettimeofday()` (wall). Available everywhere, accuracy depending on the OS/HW.
- `getrusage()` (cpu). Available everywhere, accuracy depending on the OS/HW.
- `clock_gettime()` (wall and cpu). Not available everywhere, very accurate.
- `std::chrono` in C++ (wall). Beware, spec is quite flexible and you don't know what you get.

$\Longrightarrow$ Let's take a look to code/perf/utils.cpp

I lied to you all the way along...

When I was talking about XNPY I wasn't using this implementation...

```
for (size_t i = 0; i < N; i++) {
    double xpow = 1.0; double xi = x[i];
    for (size_t p = 0; p < pow; p++)
        xpow *= xi;
    y[i] = xpow + y[i];
}
```

I lied to you all the way along...

...but this:

```
/* AVX 4 unroll on aligned memory */
__attribute__((noinline))
static void
xnpy_impl_avx_align_unroll( size_t N, size_t pow, const double * __restrict x,
    double * __restrict__ y)
{
  if (N < 8) {
    /* N is less than 8: vector too small for the AVX loop.
     * compute directly and return. */
    for (size_t i = 0; i < N; i++) {
      double xpow = 1.0;
      for (size_t p = 0; p < pow; p++)
        xpow *= x[i];
      y[i] = xpow + y[i];
    }
    return;
  }

  /* Use AVX on the aligned part */
  const double one = 1.0;

  size_t i = 0;
  for ( ; i < N; i+= 32)
    __m256d ones = _mm256_broadcast_sd(&one);
    __m256d xpow1 = xpow0;
    __m256d xpow2 = xpow0;
    __m256d xpow3 = xpow0;
    __m256d xpow4 = xpow0;
    __m256d xpow5 = xpow0;
    __m256d xpow6 = xpow0;
    __m256d xpow7 = xpow0;

    __m256d x0x3 = _mm256_load_pd(&x[i+0]);
    __m256d x1x4 = _mm256_load_pd(&x[i+4]);
    __m256d x2x5 = _mm256_load_pd(&x[i+8]);
    __m256d x3x6 = _mm256_load_pd(&x[i+12]);
    __m256d x4x7 = _mm256_load_pd(&x[i+16]);
    __m256d x5x8 = _mm256_load_pd(&x[i+20]);
    __m256d x6x9 = _mm256_load_pd(&x[i+24]);
    __m256d x7x10 = _mm256_load_pd(&x[i+28]);

    for (size_t p = 0; p < pow; p++) {
      xpow0 = _mm256_mul_pd(xpow0, x0x3);
      xpow1 = _mm256_mul_pd(xpow1, x1x4);
      xpow2 = _mm256_mul_pd(xpow2, x2x5);
      xpow3 = _mm256_mul_pd(xpow3, x3x6);
      xpow4 = _mm256_mul_pd(xpow4, x4x7);
      xpow5 = _mm256_mul_pd(xpow5, x5x8);
      xpow6 = _mm256_mul_pd(xpow6, x6x9);
      xpow7 = _mm256_mul_pd(xpow7, x7x10);
    }

    x0x3 = _mm256_load_pd(&y[i+0]);
    x1x4 = _mm256_load_pd(&y[i+4]);
    x2x5 = _mm256_load_pd(&y[i+8]);
    x3x6 = _mm256_load_pd(&y[i+12]);
    x4x7 = _mm256_load_pd(&y[i+16]);
    x5x8 = _mm256_load_pd(&y[i+20]);
    x6x9 = _mm256_load_pd(&y[i+24]);
    x7x10 = _mm256_load_pd(&y[i+28]);

    xpow0 = _mm256_add_pd(xpow0, x0x3);
    xpow1 = _mm256_add_pd(xpow1, x1x4);
    xpow2 = _mm256_add_pd(xpow2, x2x5);
    xpow3 = _mm256_add_pd(xpow3, x3x6);
    xpow4 = _mm256_add_pd(xpow4, x4x7);
    xpow5 = _mm256_add_pd(xpow5, x5x8);
    xpow6 = _mm256_add_pd(xpow6, x6x9);
    xpow7 = _mm256_add_pd(xpow7, x7x10);

    _mm256_stream_pd( &y[i+0], xpow0 );
    _mm256_stream_pd( &y[i+4], xpow1 );
    _mm256_stream_pd( &y[i+8], xpow2 );
    _mm256_stream_pd( &y[i+12], xpow3 );
    _mm256_stream_pd( &y[i+16], xpow4 );
    _mm256_stream_pd( &y[i+20], xpow5 );
    _mm256_stream_pd( &y[i+24], xpow6 );
    _mm256_stream_pd( &y[i+28], xpow7 );
  }


  /* and finish the remaining elements */
  for ( ; i < N; i++) {
    double xpow = 1.0;
    for (size_t p = 0; p < pow; p++)
      xpow *= x[i];
    y[i] = xpow + y[i];
  }
}
```

Naive XNPY

In order to understand what is going on, we must peek at the machine code (don't worry!)

```
for (size_t i = 0; i < N; i++) {
    double xpow = 1.0; double xi = x[i];
    for (size_t p = 0; p < pow; p++)
        xpow *= xi;
    y[i] = xpow + y[i];
}
```

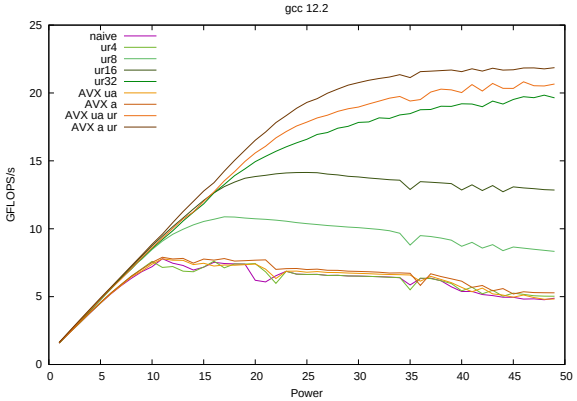
```
56          je      .L18
57  .L16:
58          vmulsd  xmm0, xmm0, xmm1
59          add     eax, 2
60          vmulsd  xmm0, xmm0, xmm1
61          cmp     eax, r9d
62          jne     .L16
63  .L18:
64          vaddsd  xmm0, xmm0, QWORD PTR [rcx+r8*8]
65          vmovsd  QWORD PTR [rcx+r8*8], xmm0
66          add     r8, 1
67          cmp     rdi, r8
68          jne     .L14
69          ret
```

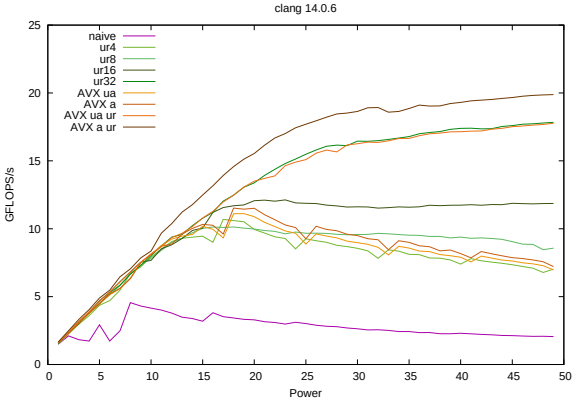
What happens is approximately that the compiler is generating pure scalar code, even if the outer loop iterations are independent.

- We are not taking advantage of the vector units in the processor.
- Let's help the compiler to vectorize: $\Longrightarrow$ demo on loop unrolling

ASM generated by GCC 13.2, Godbolt

Beware of your compiler





Last XNPY slide!

This is the last slide about XNPY! At last!

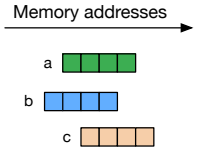
Just some last remarks on

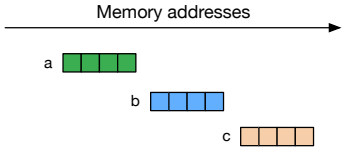
- the vectorized version
- aligned and unaligned memory access

A quick detour: restrict pointers

You may have spotted that in the code I used the `__restrict__` keyword on some function parameters. Consider the following function:

```
void sum(double * c,
    const double * a,
    const double * b) {
    for (int i = 0; i < 4; i++)
        c[i] = a[i] + b[i];
}
```





With `__restrict__` keyword I am promising to the compiler that there is no *pointer aliasing*

$\Longrightarrow$ Demo on godbolt (note: `-O3 -mavx`)

- If you break the promise it's undefined behaviour.
- FORTRAN assumes no aliasing: this is why people thinks it is faster than C++.
- cf. `a.noalias()` in Eigen.

Review of C pointers

Pointer syntax:

- `int *` → pointer to `int`
- `int const *` → pointer to `const int` (pointed data can't change)
- `int * const` → const pointer to `int` (pointer can't change)
- `int const * const` → const pointer to `const int` (neither pointer nor pointed data can change)

To add confusion:

- `int const *` is the same as `const int *`
- `int const * const` is the same as `const int * const`

Matrix multiplication

AXPY and XNPY do only linear accesses to the memory, so the cache was not important. So **let the cache enter the game**!

Let $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{R}^{N \times N}$. The matrix multiplication $\boldsymbol{C} = \boldsymbol{AB}$ is computed as $c_{ij} = a_{ik} b_{kj}$. This is probably the most important operation in scientific computing.

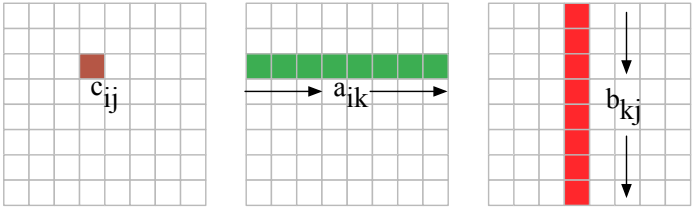
The code for the matrix multiplication looks like this:

```
for (size_t i = 0; i < N; i++)
    for (size_t j = 0; j < N; j++)
      for (size_t k = 0; k < N; k++)
        c[I(i,j)] += a[I(i,k)] * b[I(k,j)];
```

Let's do some measurements!

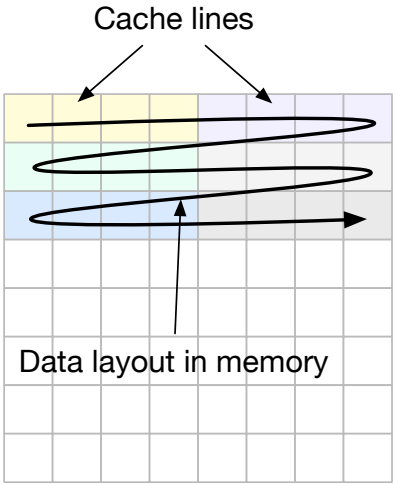
Matrix multiplication memory access patterns

```
for (size_t i = 0; i < N; i++)
    for (size_t j = 0; j < N; j++)
      for (size_t k = 0; k < N; k++)
        c[I(i,j)] += a[I(i,k)] * b[I(k,j)];
```



- Matrix ***A*** is accessed by rows
- Matrix ***B*** is accessed by columns

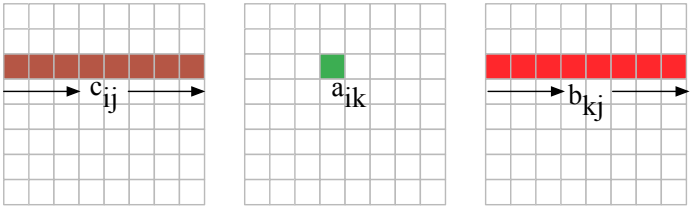
Matrix data layout



- Matrices are laid out row by row
- Accessing by column is not cache friendly
- We should try to access **B** by row

Matrix data layout

```
for (size_t i = 0; i < N; i++)
    for (size_t k = 0; k < N; k++)       // swapped this
        for (size_t j = 0; j < N; j++)   // and this
            c[I(i,j)] += a[I(i,k)] * b[I(k,j)];
```



- Matrix ***C*** is repeatedly accessed by rows, ***A*** is accessed by rows, ***B*** is accessed repeatedly by rows
- Instead of computing a single element of ***C*** in one go, we compute multiple elements iteratively
- Lesson: **Pay attention to your memory access patterns** and try to make cache happy.

More matrix-matrix optimization techniques

- Tiling (or blocking): essentially is loop unrolling in 2D

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

- Multi-level tiling: Tile, but in a way that takes in account all the cache levels
- Packing: reorder the elements in a way that minimizes the memory traffic

– Part 4 –

Making your code go fast(er): the gory stuff

Optimization rules revisited

Profilers

A profiler is a software that can do various measurements on your code

- GProf
- Valgrind (much more than a profiler)
- Intel VTune (I'll show you this)

Profilers are usually not easy to use. I'll show you just the tip of an iceberg.

$\Longrightarrow$ Intel VTune demo on matrix multiplication

Finally some math

We consider the 1D conservation law

$$\frac{\partial u}{\partial t} + \frac{\partial f}{\partial x} = g, \qquad x \in \Omega, \quad + \text{BCs}.$$

- $u(x,t)$ solution
- $f(u)$ flux (for simplicity: $f(u) = au$, $a > 0$)
- $g(x,t)$ forcing function or source

We want to solve this equation numerically.

Solving a PDE

What does "solving a PDE numerically" mean?

The full details of the PDE solution u cannot be represented/handled in a numerical method, therefore one needs to find an approximate solution u_h. But:

- How to represent the numerical approximation u_h of u?
- In which sense does u_h satisfy the given PDE?
  If u_h is approximate and we plug it in the original equation, the residual

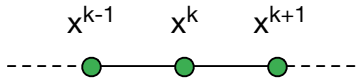
$$\mathcal{R}(x,t) = \frac{\partial u_h}{\partial t} + \frac{\partial f_h}{\partial x} - g(x,t)$$

  will in general not be zero. But how/where?

Different numerical methods result from different choices.

Finite differences

We consider a grid of K points $x^k, k \in \{1, \dots, K\}$ and $h^k = x^{k+1} - x^k$ the local grid size.



For each point of the grid

$$\frac{du_h(x^k, t)}{dt} + \frac{f(x^{k+1}, t) - f(x^{k-1}, t)}{h^k + h^{k-1}} = g(x^k, t),$$

therefore, if we have K grid points, the scheme will yield K equations. For simplicity we'll take $h^k = \Delta x \ \forall k$

$$u_{n+1}^k = u_n^k - a\Delta t \frac{u_n^{k+1} - u_n^{k-1}}{2\Delta x} + g(x^k, n\Delta t)\Delta t.$$

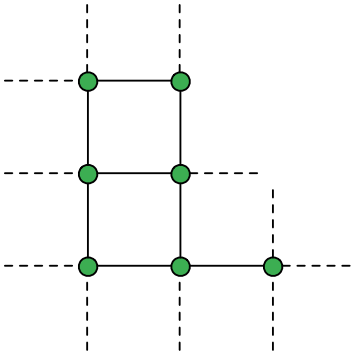
Therefore, in this case, $\mathcal{R}(x, t) = 0$ in each grid point.

Finite differences - Limitations

Finite differences are simple and effective but have drawbacks

$$\frac{d\mathbf{u}_h(\mathbf{x}^{k,\ell},t)}{dt} + \frac{f(\mathbf{x}^{k+1,\ell},t) - f(\mathbf{x}^{k-1,\ell},t)}{2\Delta x} + \frac{f(\mathbf{x}^{k,\ell+1},t) - f(\mathbf{x}^{k,\ell-1},t)}{2\Delta y} = g(\mathbf{x}^{k,\ell},t).$$

- What happens in 2D/3D?
- How do you handle unstructured grids?
- You can not write derivatives in a direction not aligned with one of the cartesian axes...[3]

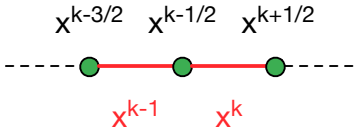


[3] Somewhat you can: see Mimetic Elements/Discrete Geometric Approach/Cell method.

Finite volumes

An alternative approach: instead of working on nodes, let's work on cells.

We consider a grid $\mathcal{M}$ of K *cells*. T^k is now the interval $[x^{k-1/2}, x^{k+1/2}]$.



What is the discrete solution in this case? Simplest possibility $\rightarrow$ cell-by-cell average $\bar{u}^k$

$$\bar{u}^k = \frac{1}{h^k} \int_{T^k} u \, dV, \qquad \frac{d\bar{u}^k}{dt} = \frac{1}{h^k} \int_{T^k} \frac{\partial u}{\partial t} \, dV.$$

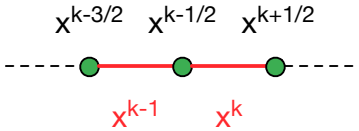
This time we require that the average of the **cellwise** residual goes to zero in each cell

$$\mathcal{R}(x,t) = \frac{\partial \bar{u}^k}{\partial t} + \frac{\partial f(\bar{u}^k)}{\partial x} - g(x,t) = 0.$$

Finite volumes

An alternative approach: instead of working on nodes, let's work on cells.

We consider a grid $\mathcal{M}$ of K *cells*. T^k is now the interval $[x^{k-1/2}, x^{k+1/2}]$.



What is the discrete solution in this case? Simplest possibility $\rightarrow$ cell-by-cell average $\bar{u}^k$

$$\bar{u}^k = \frac{1}{h^k}\int_{T^k} u\, dV, \qquad \frac{d\bar{u}^k}{dt} = \frac{1}{h^k}\int_{T^k} \frac{\partial u}{\partial t}\, dV.$$

This time we require that the average of the **cellwise** residual goes to zero in each cell

$$\mathcal{R}(x,t) = \frac{\partial \bar{u}^k}{\partial t} + \frac{\partial f(\bar{u}^k)}{\partial x} - g(x,t) = 0.$$

Step 1: take the average of everything *elementwise* (remember: $f(u) = au$)

$$\frac{1}{h^k}\int_{T^k} \frac{\partial u}{\partial t} dV + \frac{1}{h^k}\int_{T^k} \frac{\partial f}{\partial x} dV = \frac{1}{h^k}\int_{T^k} g\ dV$$

Finite volumes - Fluxes

Step 2: Apply the divergence theorem to the spatial term

$$\int_{T^k} \nabla \cdot \mathbf{f} dV = \int_{\partial T^k} \mathbf{f} \cdot \hat{\mathbf{n}} dS \quad \xrightarrow{\text{or, in 1D}} \quad \int_{T^k} \frac{\partial f}{\partial x} dl = f(x^{k+1/2}) - f(x^{k-1/2}),$$

which allows us to obtain the element-local statement

$$\frac{d\bar{u}^k}{dt} + \frac{1}{h^k} \int_{T^k} \frac{\partial f}{\partial x} dV = \bar{g}^k \quad \rightarrow \quad \frac{d\bar{u}^k}{dt} + \frac{f(x^{k+1/2}) - f(x^{k-1/2})}{h^k} = \bar{g}^k$$

But... $\bar{u}^k$ is not single valued at the element boundaries, what are the quantities $f(u^{k+1/2})$ and $f(u^{k-1/2})$? (remember: $f(u) = au$)

- Centered fluxes: $f(x^{k+1/2}) := \frac{1}{2}(a\bar{u}^k + a\bar{u}^{k+1})$
- Lax-Friedrichs fluxes: $f(x^{k+1/2}) := \frac{1}{2}(a\bar{u}^k + a\bar{u}^{k+1}) + \frac{C}{2}(\bar{u}^k - \bar{u}^{k+1})\hat{\mathbf{n}}$
- Many other choices are possible

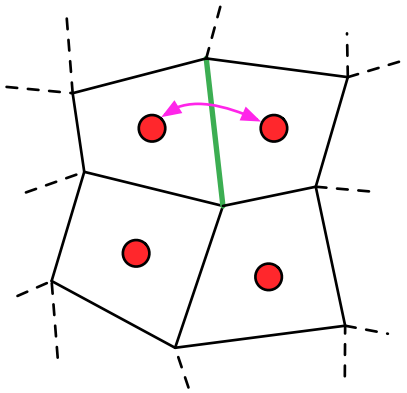
Finite Volumes - Generalizations and limitations

Multi-dimensional generalization is now straightforward:

$$\frac{d\bar{\mathbf{u}}^k}{dt} + \frac{1}{h^k}\int_{\partial T^k} \mathbf{f}\cdot\hat{\mathbf{n}}\, dS = \bar{\mathbf{g}}, \quad \forall k \in \mathcal{M}$$

By asking something (= how the residual goes to zero) on the cells instead of asking something on the nodes we gained geometric flexibility.

Compared to FD, with FV we lose an order of approximation ($\rightarrow$ only one DOF per cell). We can however reconstruct a higher order solution using the averages on the nodes.



Finite volumes for acoustic waves

Acoustic waves can be described by a system of two equations in p (pressure) and $\mathbf{v}$ (velocity)

$$\begin{cases} \frac{\partial p}{\partial t} + \nabla \cdot \mathbf{v} = 0 \\ \frac{\partial \mathbf{v}}{\partial t} + \nabla p = \mathbf{0} \end{cases},$$

or, in conservation law form

$$\frac{\partial \mathbf{q}(\mathbf{x}, t)}{\partial t} = -\nabla \cdot \mathbf{F}(\mathbf{q}), \quad \mathbf{q} := \begin{bmatrix} p \\ v_x \\ v_y \end{bmatrix}, \quad \mathbf{F}(\mathbf{q}) := \begin{bmatrix} v_x & v_y \\ p & 0 \\ 0 & p \end{bmatrix},$$

from which we can obtain a semi-discrete scheme

$$\frac{\partial}{\partial t}(\mathbf{q}_h, \mathbf{v})_{T_k} = -\left(\begin{bmatrix} \{v_{x,h} n_x\} + \{v_{y,h} n_y\} + \frac{C}{2}[\![p_h]\!] \\ \{p_h n_x\} + \frac{C}{2}[\![v_{x,h}]\!] \\ \{p_h n_y\} + \frac{C}{2}[\![v_{y,h}]\!] \end{bmatrix}, \mathbf{v} \right)_{\partial T_k}, \qquad \begin{cases} \{\alpha\} = \frac{\alpha^+ + \alpha^-}{2} \\ [\![\alpha]\!] = \alpha^+ - \alpha^- \end{cases}.$$

Recall that the divergence applied to a matrix operates row by row.

Profiling our FV code

Project assignments

- Detailed analysis of the finite volumes code and optimization proposals (negative results are accepted, provided you show you understand the code)
- Implementing GEMM with the packing technique and measure performance
- Study of the NEON instruction set and comparison with AVX - you need a machine with an ARM cpu if you want to run some code
- Analysis and optimization of (part of) your own research codes