


Practical aspects of computer architectures for data acquisitions: computing platforms

...

Pawel Szostek, Niko Neufeld
ISOTDAQ, 01.02.2016

CERN openlab



LHCb



Content of this presentation

In this lecture I will talk about:

- key concepts in computer architectures,
- bird's eye view evolution of the silicon technology
- seven performance dimensions of modern computing platforms,
- how fast computers are,
- useful tools for running stuff,

What I will not talk about

- Memory Management Unit,
- Cache associativity,
- PCIe architecture (see Paolo's talk),
- FPGAs (see Hannes' and Manoel's talks),
- ASICs,



Image source: www.diquaedila.it

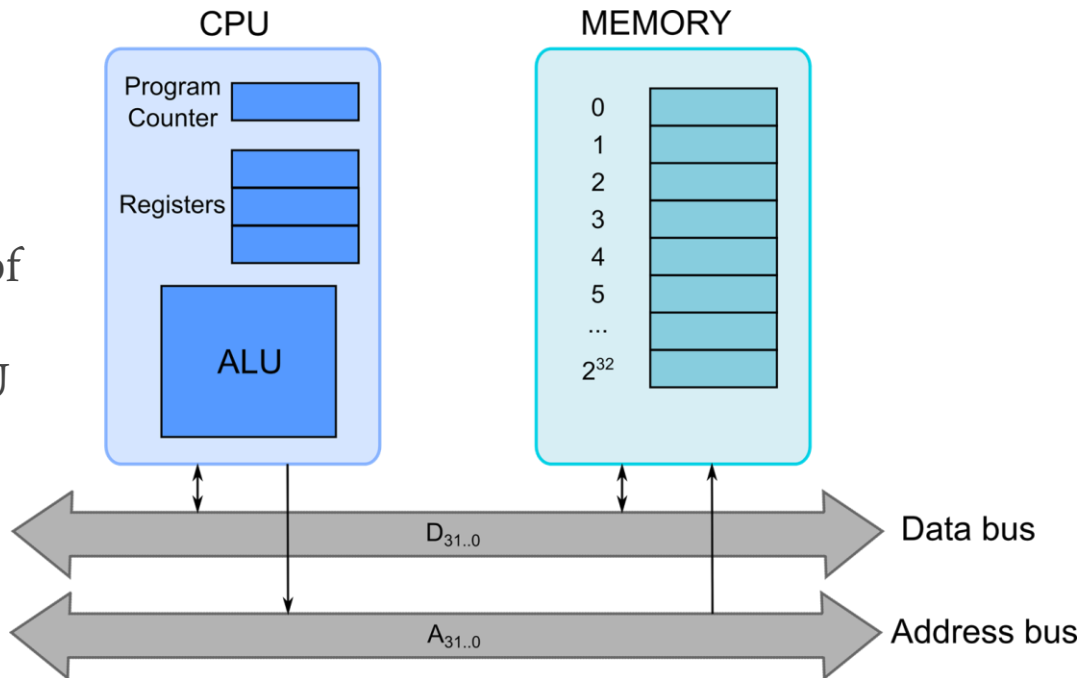
Part 1: Basic concepts in the computer architecture

Computers are already 70 years old...

but it's still von Neumann's idea!*

- there is an execution unit,
- there is a memory,
- they communicate over buses
- program counter keeps tracks of the execution
- registers store operands of ALU operations
- ALU does the proper computation

* with a few improvements

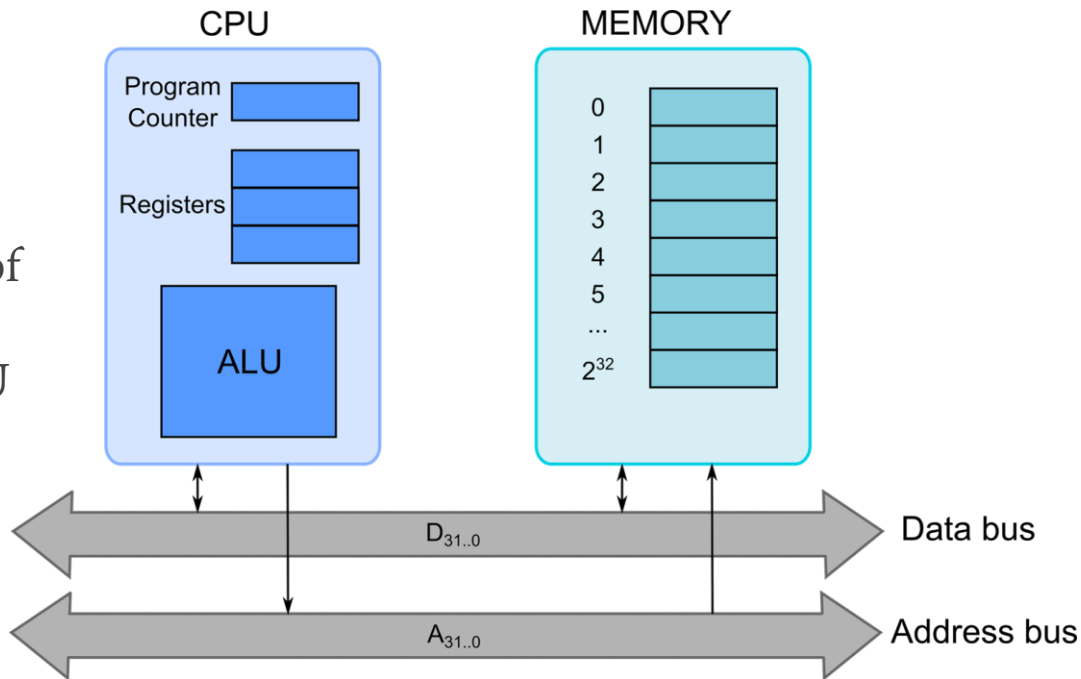


Computers are already 70 years old...

but it's still von Neumann's idea!*

- there is an execution unit,
- there is a memory,
- they communicate over buses
- program counter keeps tracks of the execution
- registers store operands of ALU operations
- ALU does the proper computation

* with a few improvements

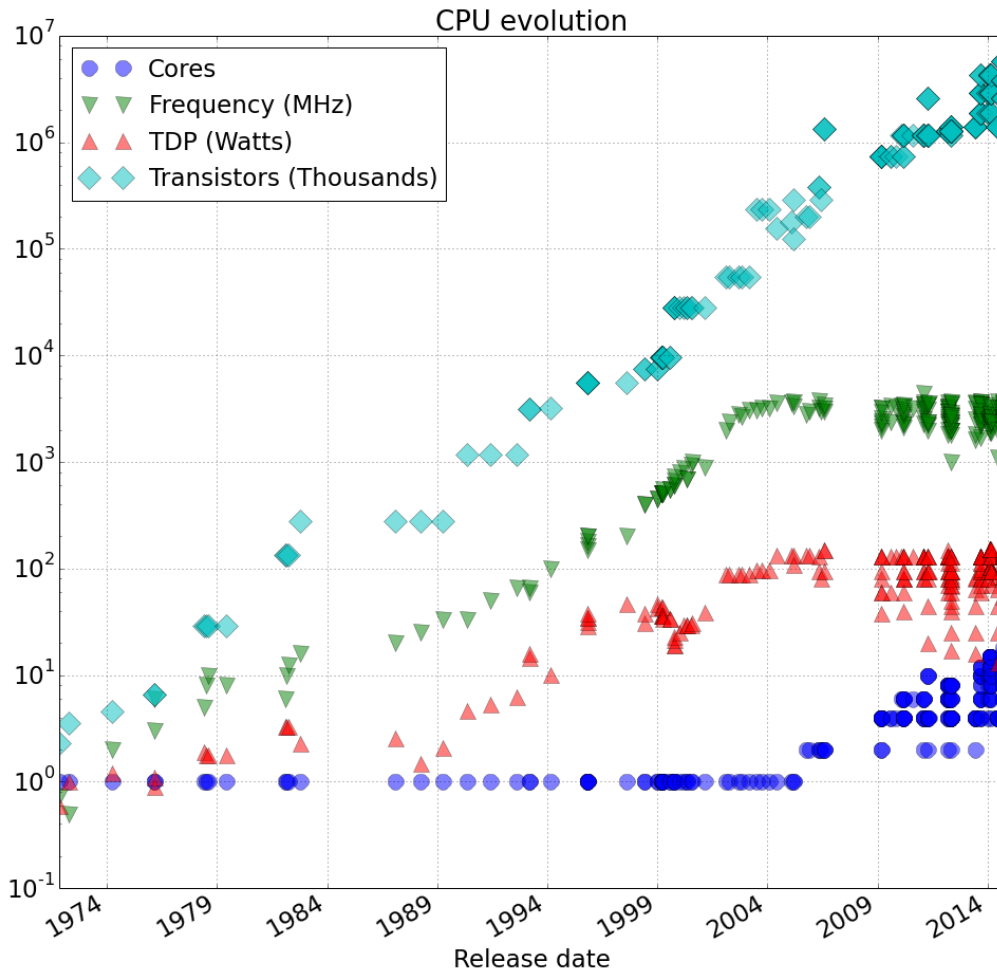


↪ This lecture is mostly about the improvements 5

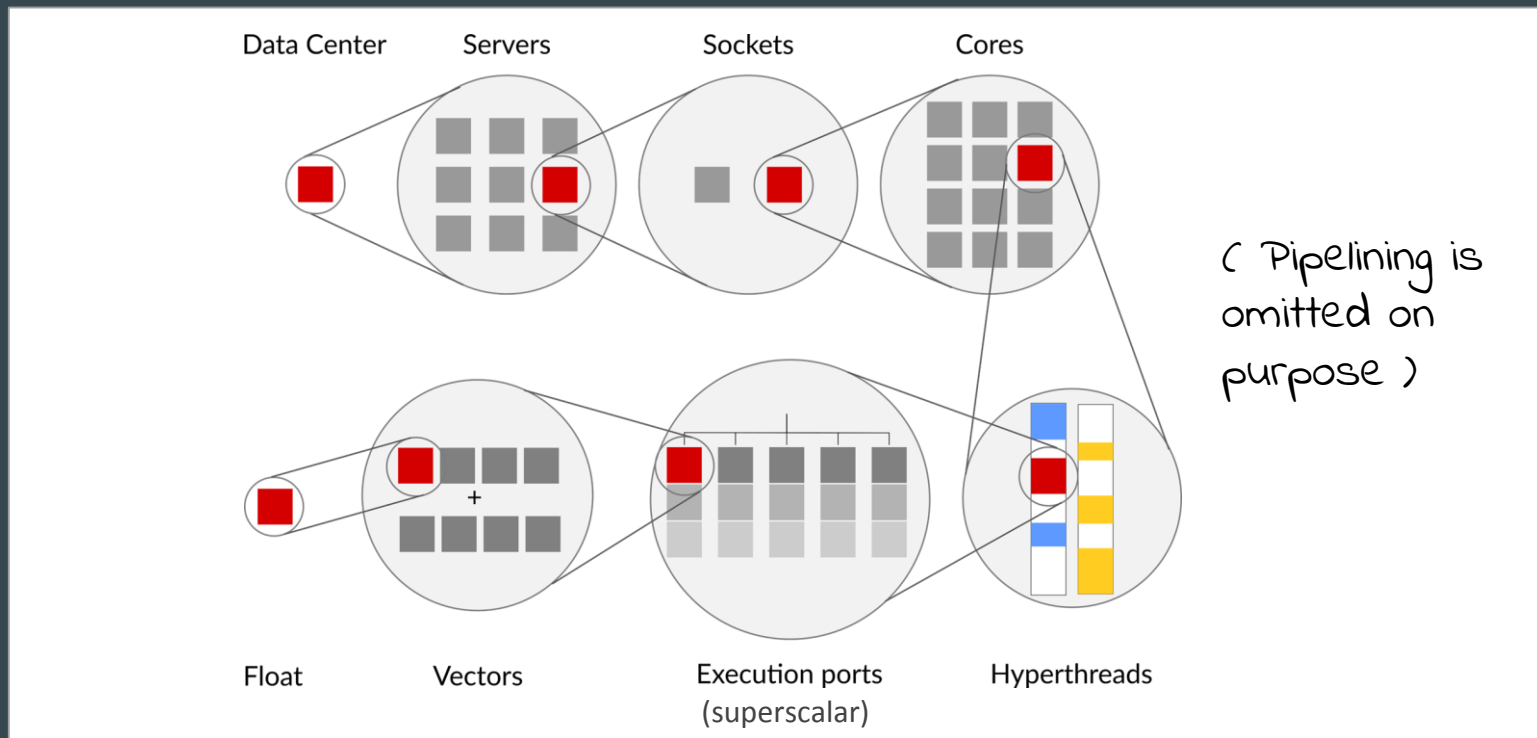
Moore's law

Probably you hear that for the 34th time in the past 7 days, but...

- Number of transistors in CPUs is growing **exponentially**
- Clock frequencies don't grow anymore
- New transistors are invested into more and bigger cores



Multiplicative dimensions of parallelism



Why should we care?



CMS Experiment at the LHC, CERN

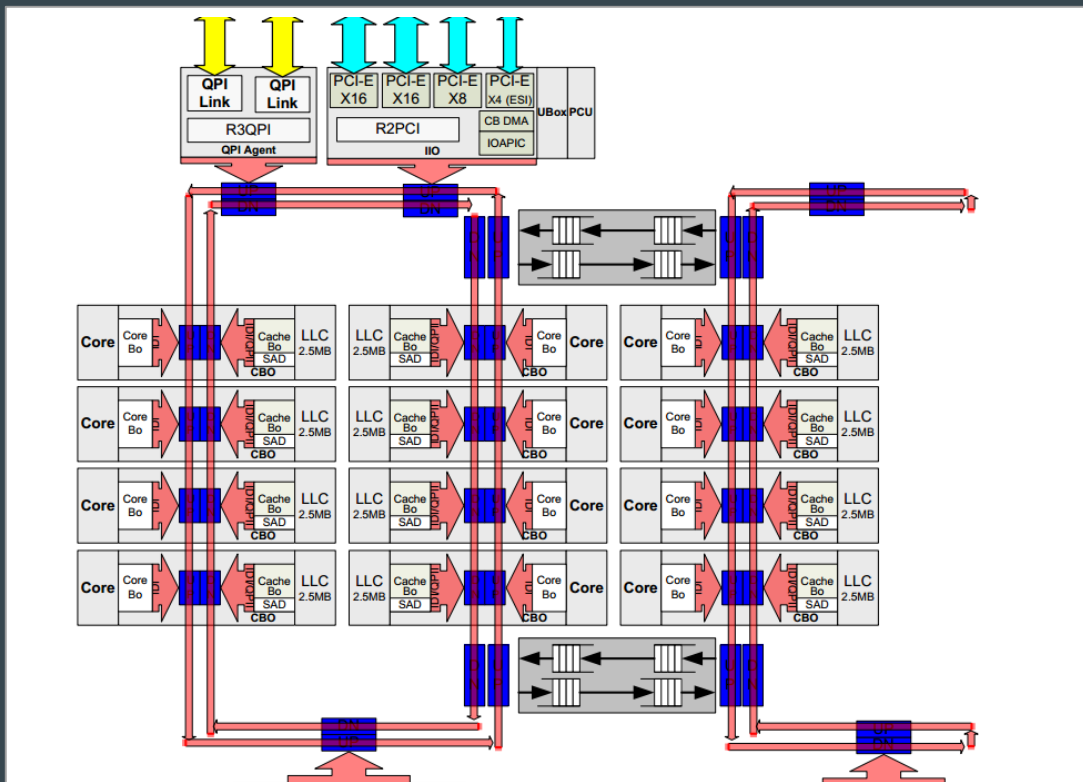
Data recorded: 2010-Jul-09 02:25:58.839811 GMT(04:25:58 CEST)

Run / Event: 139779 / 4994190

That's why. Period.

Era of Pentium 4 is over

- Nowadays processors have more than one core,
- Cores are connected by an interconnect (a.k.a. uncore),
- They share LLC, e.g. one core can use in this case $12 \times 2.5\text{MB}$ of cache,
- Note: QPI, PCIe attached to the cores 0-7



Superscalar architecture - Haswell

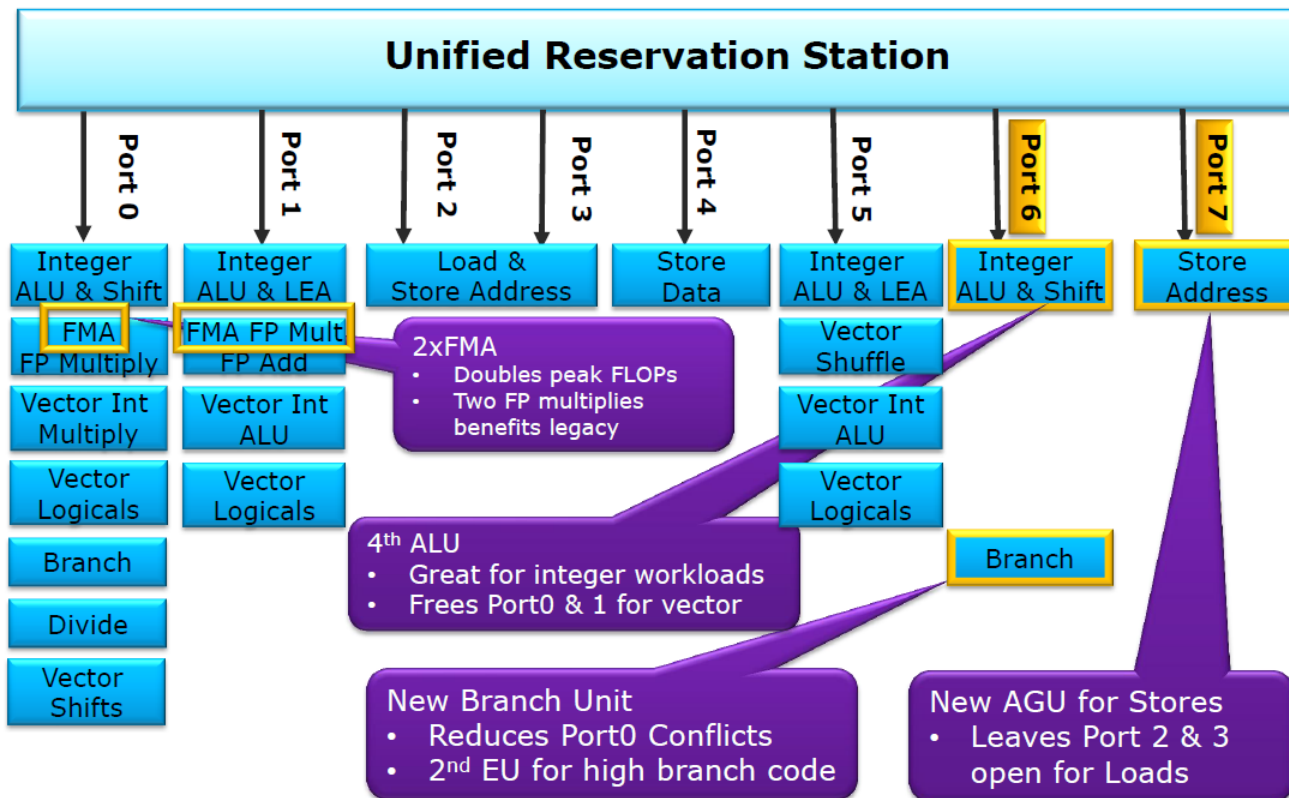
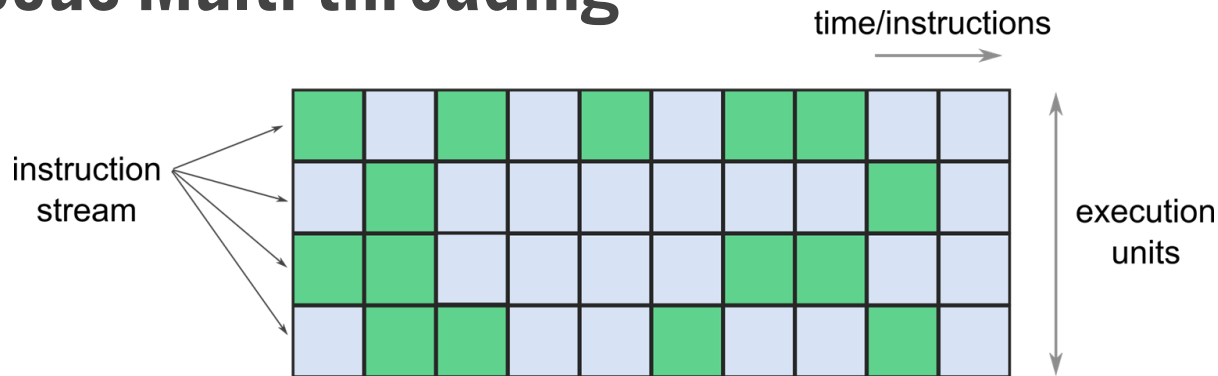


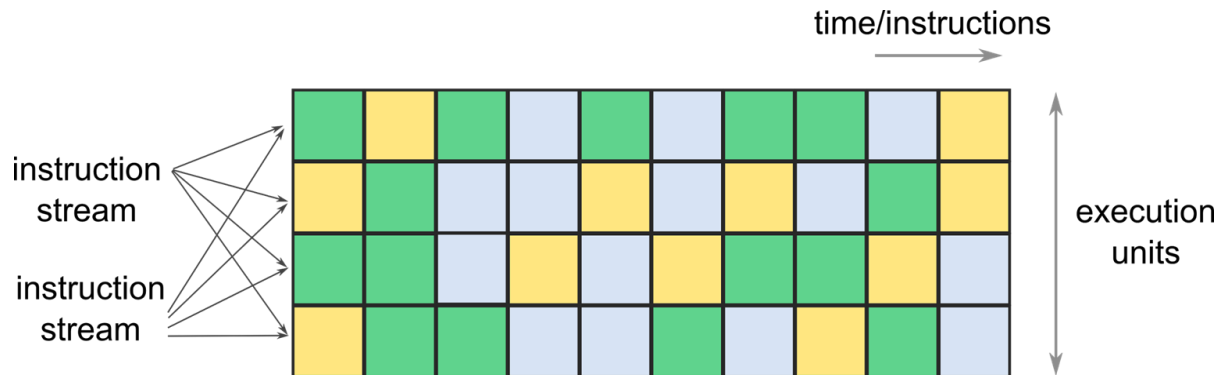
Image source: Intel

Simultaneous Multi-threading

Normal situation:
no SMT, one
instruction stream

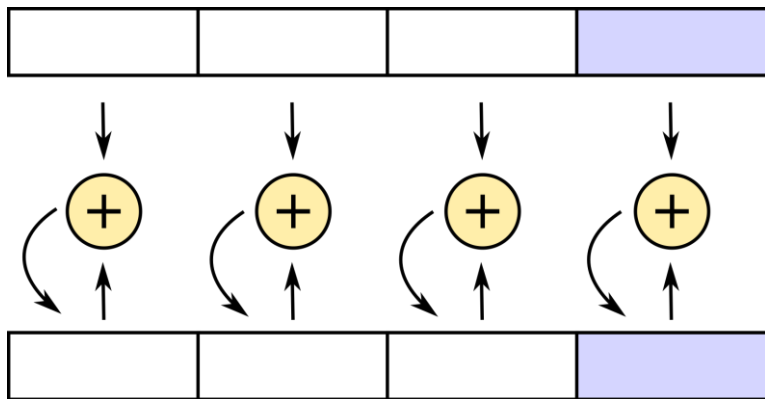


SMT enabled: two
instruction streams
sharing some
hardware resources



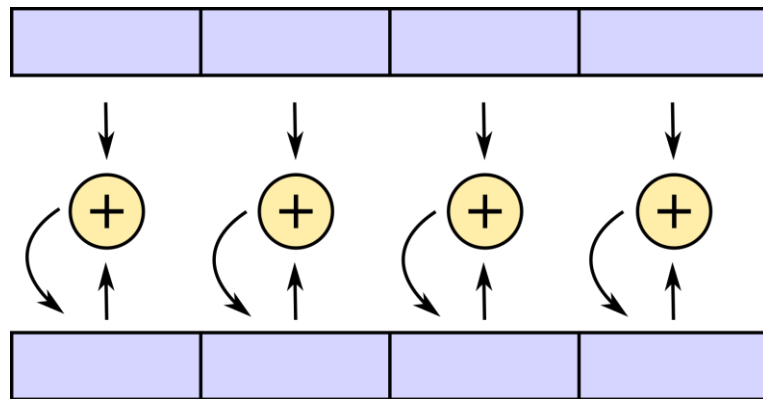
Vector instructions

Scalar addition



→ A long register is involved, but only a fraction of it is used

Vector addition

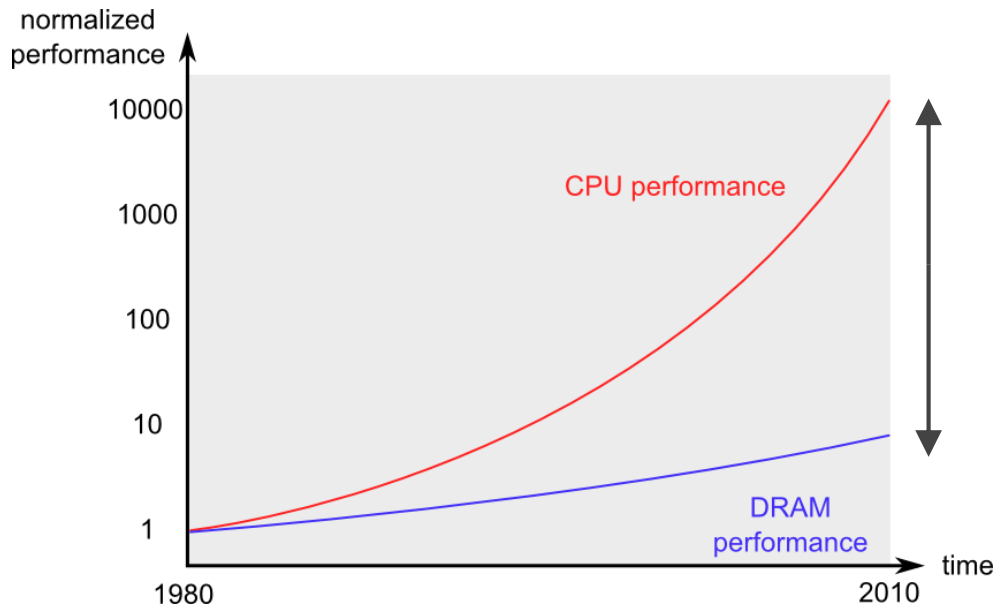


→ same latency as the scalar counterpart
→ 4 times higher throughput



Part 1.1: Memory architecture

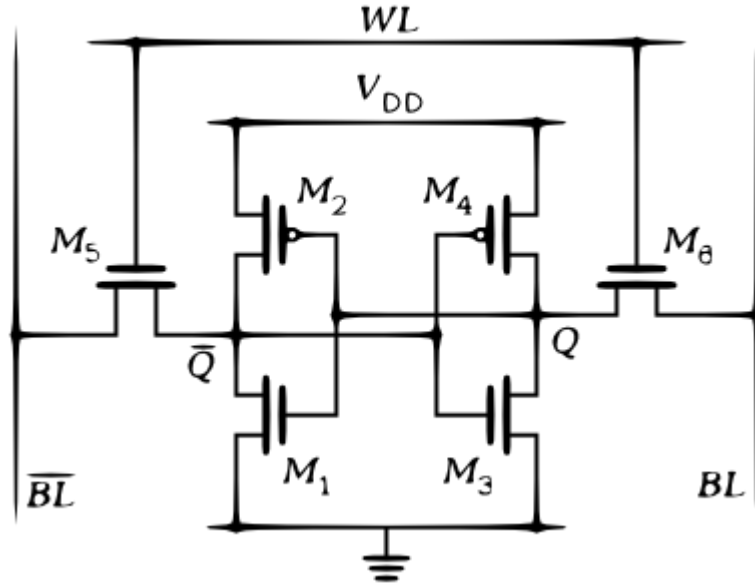
What changed since the 70's?



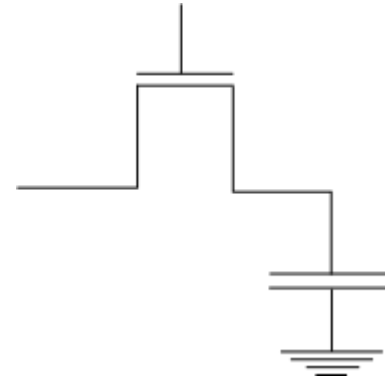
our problem is

growing.

(Short) interlude: memory != memory



SRAM



DRAM

Which technology is faster? And why SRAM?

How to speed up the memory?

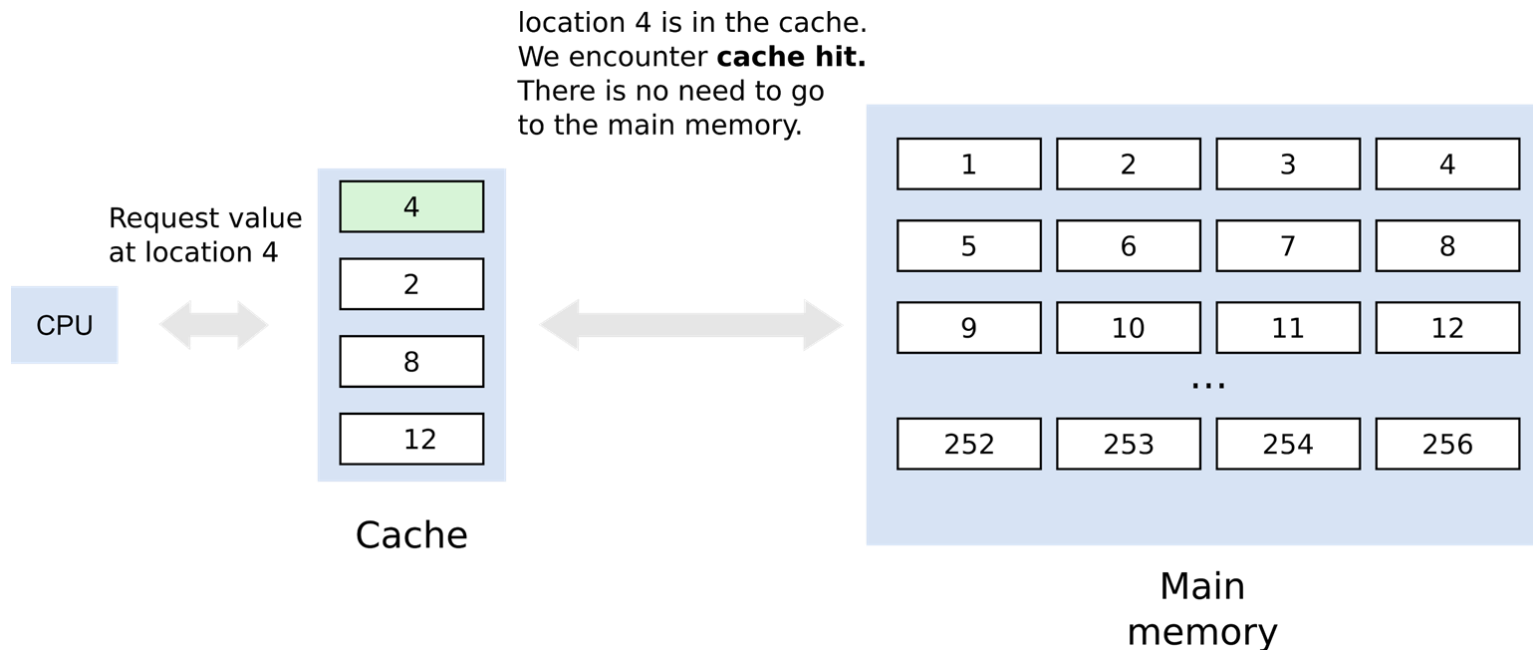
Problem: fast memory is expensive

Solution: introduce memory hierarchy, with a fast memory on the top

and slow on the bottom

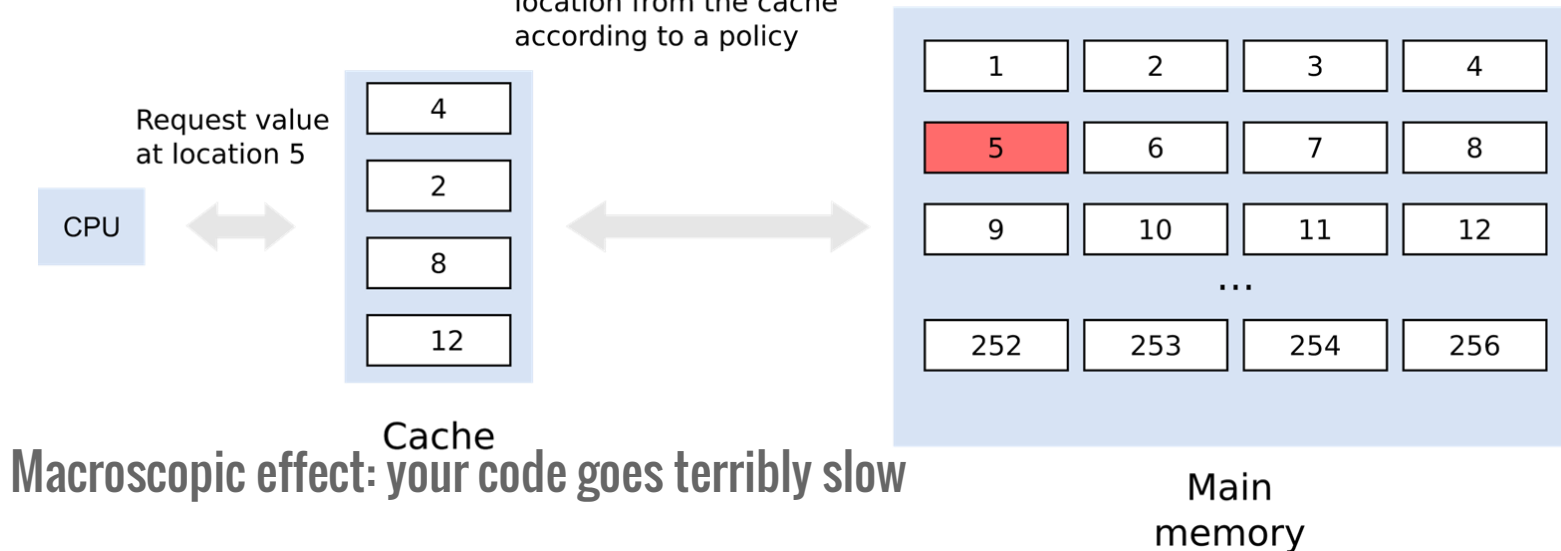
	Technology	Capacity	Latency
Registers	SRAM	bytes	< 1 ns
L1 Cache	SRAM	kilobytes	1 ns
L2 Cache	SRAM	megabytes	< 10 ns
main memory	DRAM	gigabytes	70-100ns

Cache loads: hit

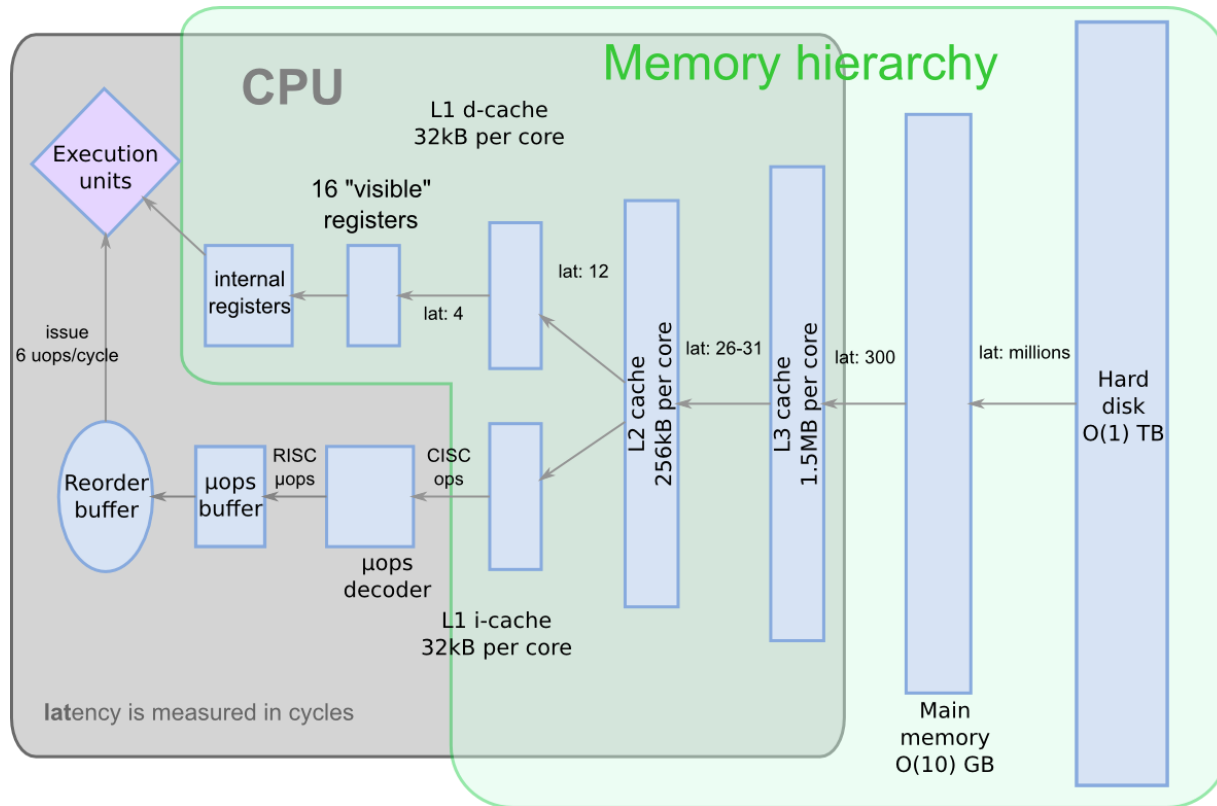


Cache loads: miss

Location 5 is not in the cache.
We encounter a **cache miss**.
There is a need to **fetch**
location 5 from the main
memory and **evict** another
location from the cache
according to a policy



Food for thought: the big picture



Memory bandwidth consequences

(you remember our growing problem?)

→ Theoretical peak memory bandwidth: the maximum amount of data that can be read in a unit of time.

$$\text{bandwidth}_{\text{peak}} = \text{channels} \times \text{bus width} \times \text{frequency}$$

→ therefore for a real memory we get (NVIDIA Tesla K40)

$$\text{bandwidth}_{\text{peak}} = 2 \times 384/8 \text{ (bytes)} \times 3\text{GHz} = 288 \text{ GB/s}$$

288 GB is equivalent to 36G doubles

→ K40's throughput is 1400GFLOPS (double)

→ To achieve peak performance we need $1400/36 = 39$ operations per double



Part 2: Computing landscape - a view from 10,000 meters



Typical machine in your computing farm

- dual socket Intel* platform,
- 6-8 cores per socket, twice as much with hyperthreading,
- 2.4 GHz main clock frequency,
- 256 bits vectors, AVX/AVX2 ISA,
- 5-8 superscalar execution units, 4-way dispatch,
- 64GB of main memory,
- 4 memory channels (DDR3, DDR4),
- 1x SSD or 2x SSD with LVM striping,

* sorry AMD, but this is the truth

High performance vs. low power solutions

- power dissipation is a major problem in the datacenter
- power envelopes of the CPUs available for data centers span from 5W to 140W,
- high-power units usually are delivered with high core counts and wider cores
- HEP software doesn't necessarily profit from all these goodies
- so far no spectacular victories

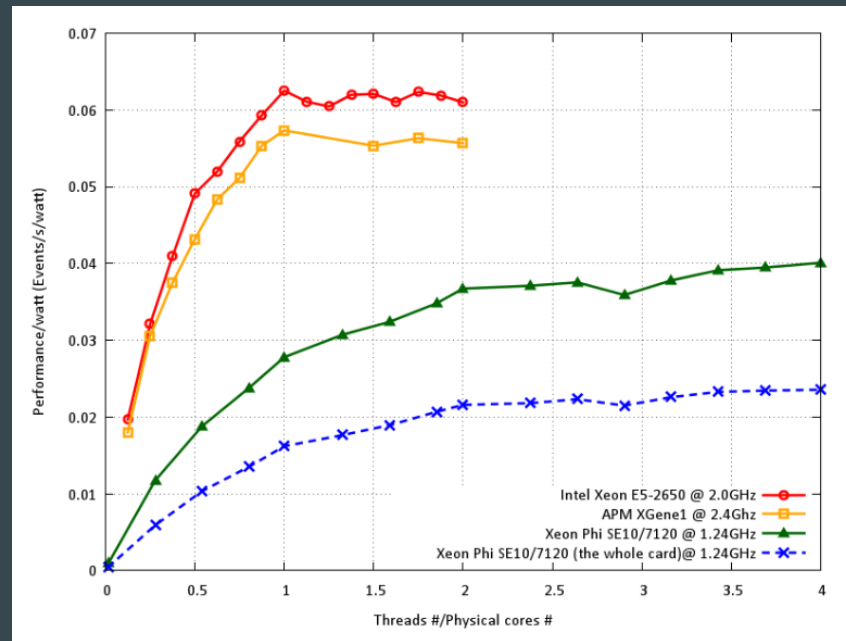


image source: David Abdurachmanov

Coprocessors

- Intel's response to GPGPU
- PCIe card with ~60 lightweight cores on it
- 16GB on-board memory
- both native and off-load execution
- nowadays rather exotic, but the next generation might be a game changer (~1 year from now)

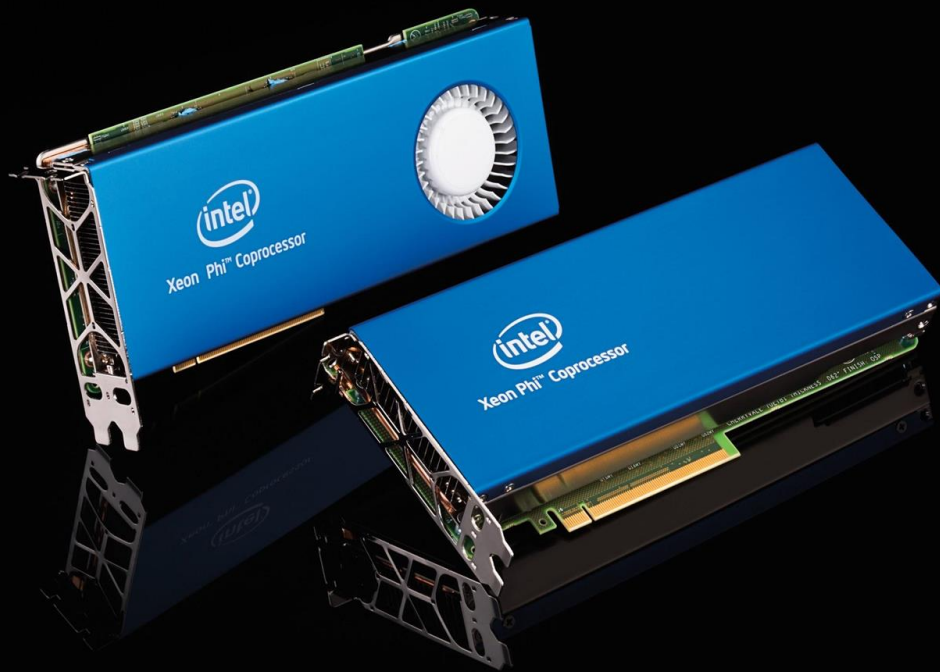
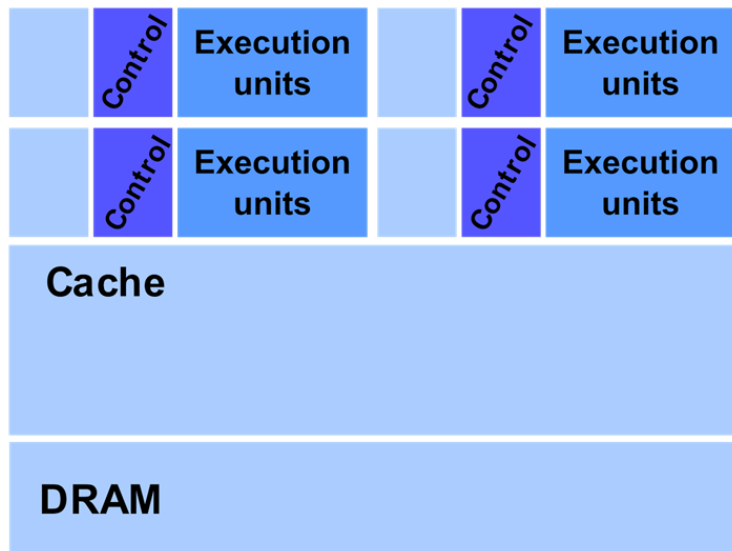
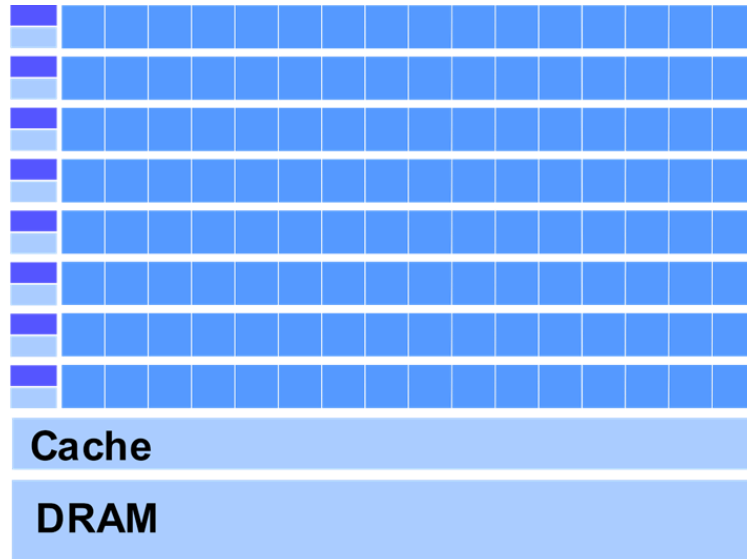


Image source: Intel

GPGPUs



CPU



GPU

GPUs can execute massively parallel code with simple control flow.

Part 3: How fast are computers? ...

Latencies every programmer should (roughly) know

Access type	cycles	nanoseconds
L1 cache reference	4	2
L2 cache reference	12	6
L3 cache reference	44	22
Main memory reference	300	150
Read 1MB sequentially from an SSD	500,000	1,000,000
HDD seek	5,000,000	10,000,000
CERN-SLAC-CERN round-trip	oh well...	150,000,000

Source: own measurements, Intel

Performance optimization checklist

Level	Possible gains	Factor	Means
Algorithm	Huge	10x..1000x and more	Changing complexity, (parallelizing)
Source code	Medium	1x-10x	Data layout, memory accesses, data reuse, vectorization
Compiler	Medium or Low	1.5x	Tweaking compilation flags, (changing the compiler)
Operating system	Low	1.3x	Upgrading the kernel and glibc runtime
Hardware	Medium	10% between two consecutive microarchitectures	Moving to a newer microarchitecture (e.g. Ivy Bridge -> Haswell)

Riddle #1: Simple loop iterations

```
// Number to guess: How many iterations of  
// this loop can we do in one second?  
// gcc -o iter -O2 iter.c
```

```
int main(int argc, char **argv) {  
    int NUMBER, i, s = 0;  
    NUMBER = atoi(argv[1]);  
  
    for (s = i = 0; i < NUMBER; ++i) {  
        s += 1;  
    }  
    return 0;  
}
```

100,000

1,000,000

10,000,000

100,000,000

1,000,000,000

Riddle #1: Simple loop iterations

```
// Number to guess: How many iterations of  
// this loop can we do in one second?  
// gcc -o iter -O2 iter.c
```

```
int main(int argc, char **argv) {  
    int NUMBER, i, s = 0;  
    NUMBER = atoi(argv[1]);  
  
    for (s = i = 0; i < NUMBER; ++i) {  
        s += 1;  
    }  
    return 0;  
}
```

100,000

1,000,000

10,000,000

100,000,000

1,000,000,000

Riddle #1.1: Same thing, but with Python

```
#!/usr/bin/env python

// Number to guess: How many iterations of
// this loop can we do in one second?

def f(NUMBER):
    s = 0
    for _ in xrange(NUMBER):
        s += 1

import sys
f(int(sys.argv[1]))
```

100,000

1,000,000

10,000,000

100,000,000

1,000,000,000

Riddle #1.1: Same thing, but with Python

```
#!/usr/bin/env python

// Number to guess: How many iterations of
// this loop can we do in one second?

def f(NUMBER):
    s = 0
    for _ in xrange(NUMBER):
        s += 1

import sys
f(int(sys.argv[1]))
```

100,000

1,000,000

10,000,000

100,000,000

1,000,000,000

It's actually 80,000,000

Riddle #2: Writing to the main memory

```
// gcc -o iter -O2 iter.c
// includes

static const unsigned int CHUNK_SIZE = 1024*1024;
char chunk[CHUNK_SIZE];

int main(int argc, char **argv) {
    long long int NUMBER, bytes_written = 0;
    char *mem = (char*) malloc(128*sizeof(char)*CHUNK_SIZE);
    NUMBER = std::stol(argv[1]);
    size_t chunks_idx = 0;
    while(bytes_written < NUMBER) {
        memcpy(mem+chunks_idx*CHUNK_SIZE, chunk, CHUNK_SIZE);
        bytes_written += CHUNK_SIZE;
        chunks_idx = (chunks_idx+1)%128;
    }
    printf("%c\n", mem[NUMBER%11]);
}
```

100,000

1,000,000

10,000,000

100,000,000

1,000,000,000

Riddle #2: Writing to the main memory

```
// gcc -o iter -O2 iter.c
// includes

static const unsigned int CHUNK_SIZE = 1024*1024;
char chunk[CHUNK_SIZE];

int main(int argc, char **argv) {
    long long int NUMBER, bytes_written = 0;
    char *mem = (char*) malloc(128*sizeof(char)*CHUNK_SIZE);
    NUMBER = std::stol(argv[1]);
    size_t chunks_idx = 0;
    while(bytes_written < NUMBER) {
        memcpy(mem+chunks_idx*CHUNK_SIZE, chunk, CHUNK_SIZE);
        bytes_written += CHUNK_SIZE;
        chunks_idx = (chunks_idx+1)%128;
    }
    printf("%c\n", mem[NUMBER%11]);
}
```

100,000

1,000,000

10,000,000

100,000,000

1,000,000,000

It's actually 8,000,000,000

Memory writing optimized: STREAM

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	60071.5	0.010935	0.010654	0.012926
Scale:	60645.6	0.010578	0.010553	0.010592
Add:	66335.0	0.014515	0.014472	0.014544
Triad:	67687.6	0.014460	0.014183	0.016421

And the winner is...

Memory writing optimized: STREAM

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	60071.5	0.010935	0.010654	0.012926
Scale:	60645.6	0.010578	0.010553	0.010592
Add:	66335.0	0.014515	0.014472	0.014544
Triad:	67687.6	0.014460	0.014183	0.016421

And the winner is...

```
#pragma omp parallel for
  for (j=0; j<STREAM_ARRAY_SIZE;
j++)
    c[j] = a[j];

#pragma omp parallel for
  for (j=0; j<STREAM_ARRAY_SIZE;
j++)
    b[j] = scalar*c[j];
```

```
#pragma omp parallel for
  for (j=0; j<STREAM_ARRAY_SIZE;
j++)
    a[j] = b[j]+scalar*c[j];

#pragma omp parallel for
  for (j=0; j<STREAM_ARRAY_SIZE;
j++)
    c[j] = a[j]+b[j];
```

Riddle #3: Writing to a drive

```
// Number to guess: How many bytes can we
// write onto a drive in a second?
static const uint32_t CHUNK_SIZE =
1024*1024;
char s[CHUNK_SIZE];

void cleanup(int fp, char* name) {
    fsync(fp);
    close(fp);
    remove(name);
}
```

```
int main(int argc, char** argv) {
    uint32_t NUMBER, bytes_written = 0;
    memset(s, CHUNK_SIZE, 'a');
    NUMBER = std::stoul(argv[1]);
    int fp = open("./tmp", O_WRONLY | O_CREAT);
    while (bytes_written < NUMBER) {
        write(fp, s, CHUNK_SIZE);
        bytes_written += CHUNK_SIZE;
    }
    cleanup(fp, "./tmp");
}
```

100,000

1,000,000

10,000,000

100,000,000

1,000,000,000

Riddle #3: Writing to a drive

```
// Number to guess: How many bytes can we
// write onto a drive in a second?
static const uint32_t CHUNK_SIZE =
1024*1024;
char s[CHUNK_SIZE];

void cleanup(int fp, char *name) {
    fsync(fp);
    close(fp);
    remove(name);
}
```

```
int main(int argc, char** argv) {
    uint32_t NUMBER, bytes_written = 0;
    memset(s, 'a', CHUNK_SIZE);
    NUMBER = atoi(argv[1]);
    FILE *f = fopen("test.tmp", "w+");
    if (f == NULL) return 1;
    while (NUMBER > 0) {
        bytes_written = fwrite(s, 1, CHUNK_SIZE, f);
        NUMBER--;
    }
    cleanup(f, "test.tmp");
    return 0;
}
```

It depends!

100,000

1,000,000

10,000,000

100,000,000

1,000,000,000

Interlude: SSD vs HDD performance

Sustained sequential reads for the data center-grade parts:

- Intel 400GB SATA3 SSD -> 500MB/s
- Intel 400GB PCIe SSD -> 2000MB/s
- HGST 4TB SATA3 -> 227 MB/s

Drives can be set up with LVM striped partition and various file systems.

How much time would it take to fill up a 400GB SSD with data?

Back to riddle #3

```
// Number to guess: How many bytes can we
// write onto a drive in a second?
static const uint32_t CHUNK_SIZE =
1024*1024;
char s[CHUNK_SIZE];

void cleanup(int fp, char* name) {
    fsync(fp);
    close(fp);
    remove(name);
}
```

```
int main(int argc, char** argv) {
    uint32_t NUMBER, bytes_written = 0;
    memset(s, CHUNK_SIZE, 'a');
    NUMBER = std::stoul(argv[1]);
    int fp = open("./tmp", O_WRONLY | O_CREAT);
    while (bytes_written < NUMBER) {
        write(fp, s, CHUNK_SIZE);
        bytes_written += CHUNK_SIZE;
    }
    cleanup(fp, "./tmp");
}
```

PCIe SSD

350,000,000

SSD

250,000,000

HDD

120,000,000

Riddle #4: What's wrong with this function?

```
// #include this and that
static const size_t DIM = 2048;

// sums up all the numbers in a 3d array
long long unsigned sumup(unsigned char array[DIM][DIM][DIM]) {
    long long unsigned sum = 0LL;
    for(size_t i=0; i<DIM; ++i)
        for(size_t j=0; j<DIM; ++j)
            for(size_t k=0; k<DIM; ++k)
                sum += array[i][k][j];
    return sum;
}
```

Riddle #4: What's wrong with this function?

```
// #include this and that
static const size_t DIM = 2048;

// sums up all the numbers in a 3d array
long long unsigned sumup(unsigned char array[DIM][DIM][DIM]) { //3d array
    long long unsigned sum = 0LL;
    for(size_t i=0; i<DIM; ++i)
        for(size_t j=0; j<DIM; ++j)
            for(size_t k=0; k<DIM; ++k)
                sum += array[i][k][j];
    return sum;
}
```

Execution takes 11 seconds

While we iterate over the array, the consecutive elements are DIM bytes away from each other. This means, that for every element we need to bring new cache line into L3 cache.

```
sum += array[i][j][k];
```


Here it's only 1.96 seconds!!!!

Riddle #4.1: Memory access pattern

```
// No numbers to guess here
```

```
int main(int argc, char **argv) {  
    int NUMBER, i, j = 1;  
    NUMBER = atoi(argv[1]);
```

Here NUMBER = 80M

```
    char* array = malloc(NUMBER);  
    for (i = 0; i < NUMBER; ++i) {  
        j = (j * 2) % NUMBER;  
        array[i] = j;  sequential  
    }  
    printf("%d", array[NUMBER/2]);  
}
```

100,000

1,000,000

10,000,000


100,000,000

1,000,000,000

```
// Number to guess: How many bytes can  
// we traverse randomly in one second
```

```
int main(int argc, char **argv) {  
    int NUMBER, i, j = 1;  
    NUMBER = atoi(argv[1]);
```

Number = ?

```
    char* array = malloc(NUMBER);  
    for (i = 0; i < NUMBER; ++i) {  
        j = (j * 2) % NUMBER;  
        array[j] = j;  random-ish  
    }  
    printf("%d", array[NUMBER/2]);  
}
```

Riddle #4.1: Memory access pattern

```
// No numbers to guess here
```

```
int main(int argc, char **argv) {  
    int NUMBER, i, j = 1;  
    NUMBER = atoi(argv[1]);
```

Here NUMBER = 80M

```
    char* array = malloc(NUMBER);  
    for (i = 0; i < NUMBER; ++i) {  
        j = (j * 2) % NUMBER;  
        array[i] = j;   
    }  
    printf("%d", array[NUMBER/2]);  
}
```

← sequential

```
// Number to guess: How many bytes can  
// we traverse randomly in one second
```

```
int main(int argc, char **argv) {  
    int NUMBER, i, j = 1;  
    NUMBER = atoi(argv[1]);
```

Number = ?

```
    char* array = malloc(NUMBER);  
    for (i = 0; i < NUMBER; ++i) {  
        j = (j * 2) % NUMBER;  
        array[j] = j;   
    }  
    printf("%d", array[NUMBER/2]);  
}
```

← random-ish

100,000

1,000,000

10,000,000

100,000,000

1,000,000,000

It's actually 35,000,000

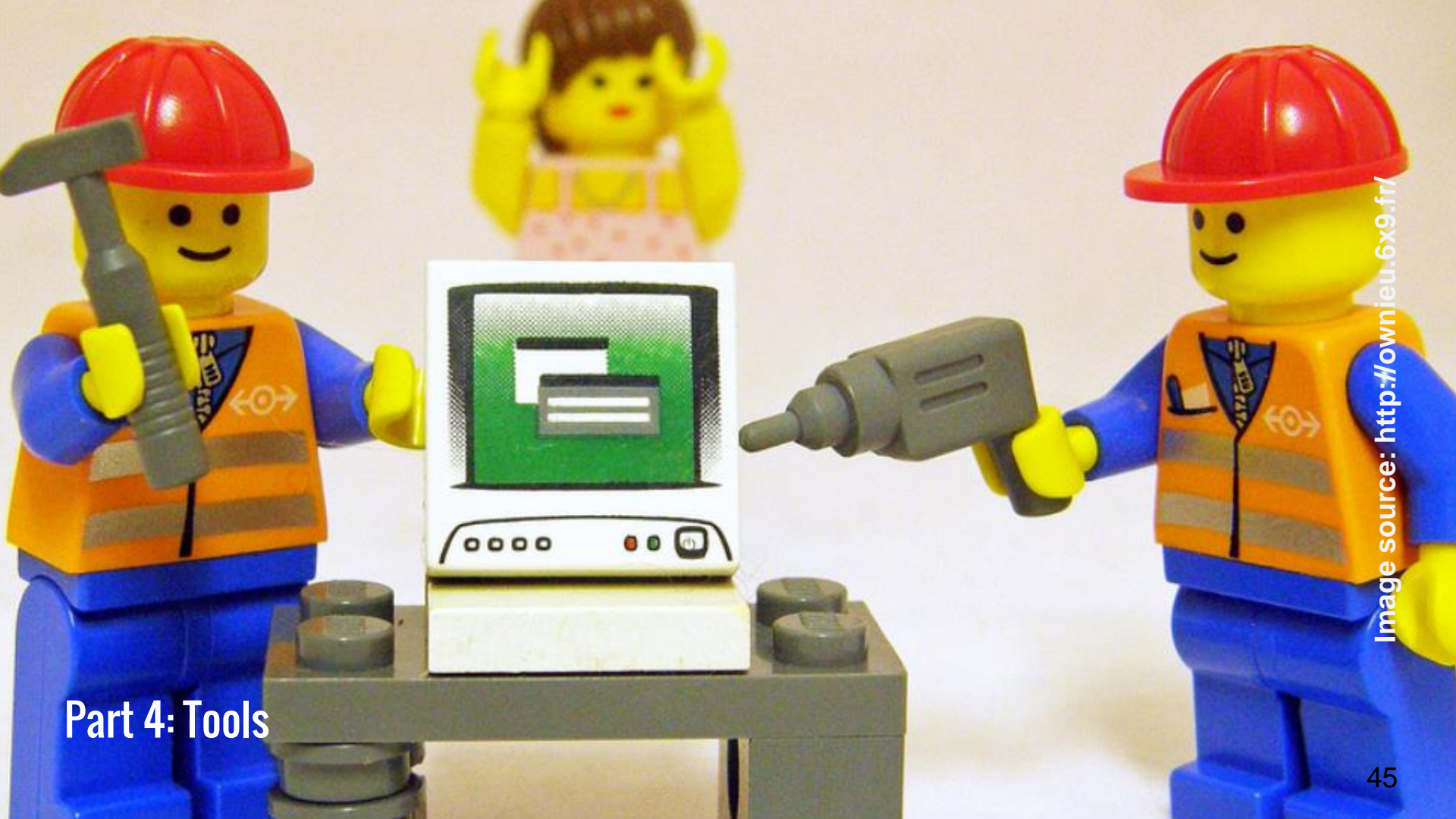


Image source: <http://ownieu.6x9.fr/>

Part 4: Tools

htop

1	[8.6%	15	[0.9%	29	[100.0%	43	[100.0%	
2	[0.0%	16	[0.0%	30	[100.0%	44	[100.0%	
3	[0.0%	17	[0.0%	31	[100.0%	45	[100.0%	
4	[100.0%	18	[0.0%	32	[0.0%	46	[100.0%	
5	[100.0%	19	[0.0%	33	[0.0%	47	[100.0%	
6	[0.0%	20	[0.0%	34	[100.0%	48	[100.0%	
7	[0.0%	21	[0.0%	35	[100.0%	49	[100.0%	
8	[100.0%	22	[0.0%	36	[0.0%	50	[100.0%	
9	[0.9%	23	[0.5%	37	[0.0%	51	[100.0%
10	[0.0%	24	[0.0%	38	[100.0%	52	[100.0%	
11	[92.0%	25	[0.0%	39	[100.0%	53	[100.0%		
12	[0.0%	26	[0.0%	40	[100.0%	54	[100.0%	
13	[0.0%	27	[0.0%	41	[100.0%	55	[100.0%	
14	[0.5%	28	[0.0%	42	[100.0%	56	[100.0%	
Mem	[2642/64159MB	Tasks: 101, 108 thr; 29 running				
Swp	[0/32191MB	Load average: 9.98 2.38 0.83				
						Uptime: 54 days, 04:41:07				

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
33304	paszoste	20	0	2576	1024	344	R	100.	0.0	0:29.91	./cpuburn-in 1
33311	paszoste	20	0	2572	1020	344	R	100.	0.0	0:29.91	./cpuburn-in 1
33301	paszoste	20	0	2572	1020	344	R	100.	0.0	0:29.91	./cpuburn-in 1
33306	paszoste	20	0	2572	1020	344	R	100.	0.0	0:29.91	./cpuburn-in 1
33327	paszoste	20	0	2580	1024	344	R	100.	0.0	0:29.89	./cpuburn-in 1
33318	paszoste	20	0	2580	1024	344	R	100.	0.0	0:29.90	./cpuburn-in 1
33316	paszoste	20	0	2580	1024	344	R	100.	0.0	0:29.90	./cpuburn-in 1
33319	paszoste	20	0	2572	1020	344	R	99.8	0.0	0:29.90	./cpuburn-in 1
33321	paszoste	20	0	2576	1024	344	R	99.8	0.0	0:29.90	./cpuburn-in 1
33324	paszoste	20	0	2576	1024	344	R	99.8	0.0	0:29.90	./cpuburn-in 1
33326	paszoste	20	0	2576	1024	344	R	99.8	0.0	0:29.90	./cpuburn-in 1
33317	paszoste	20	0	2576	1024	344	R	99.8	0.0	0:29.90	./cpuburn-in 1

F1Help F2Setup F3SearchF4FilterF5Tree F6SortByF7Nice -F8Nice +F9Kill F10Quit

taskset / numactl / GOMP_CPU_AFFINITY

- Both tools allow setting CPU affinity
- Usually yields better results than when relying on the OS scheduler
- GOMP_CPU_AFFINITY is an env. var. recognized by OpenMP
- Definitely compulsory when reading from a high bandwidth IO device (PCIe, SATA etc.)

```
$ cat get_cpu.c
#include <stdio.h>
#include <sched.h>

int main() {
    int cpu;
    cpu = sched_getcpu();
    printf("Running on core %d\n", cpu);
}

$ gcc get_cpu.c -o get_cpu
$ taskset -c 42 ./get_cpu
Running on core 42
```

GOMP_CPU_AFFINITY

You remember the STREAM benchmark? (slide #35)

```
$ ./stream
```

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	61167.9	0.010488	0.010463	0.010527
Scale:	60663.4	0.010573	0.010550	0.010599
Add:	67359.2	0.014274	0.014252	0.014306
Triad:	68196.6	0.014345	0.014077	0.016322

```
$ GOMP_CPU_AFFINITY=0-55 ./stream
```

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	65938.5	0.009743	0.009706	0.009803
Scale:	65285.8	0.009850	0.009803	0.010001
Add:	73716.3	0.013090	0.013023	0.013141
Triad:	73994.0	0.013038	0.012974	0.013110

perf / ocperf

→ perf gives insights into Hardware Events from CPU's Performance Monitoring Units. Ocperf is a thin layer on the top of perf adding more human readable names

```
$ python ../pmu-tools/ocperf.py stat -e  
mem_load_uops_retired.l3_miss,uops_executed.stall_cycles ./indices_good  
perf stat -e  
cpu/event=0xd1,umask=0x20,name=mem_load_uops_retired_l3_miss/,cpu/event=  
0xb1,umask=0x1,inv=1,cmask=1,name=uops_executed_stall_cycles/  
./indices_good
```

Performance counters for './indices_good':

14'054	mem_load_uops_retired_l3_miss
29'199'824	uops_executed_stall_cycles
1.969126	seconds time elapsed

Performance for './indices_bad':

127'067	mem_load_uops...
24'792'121'333	uops_executed...
11.33531028	seconds

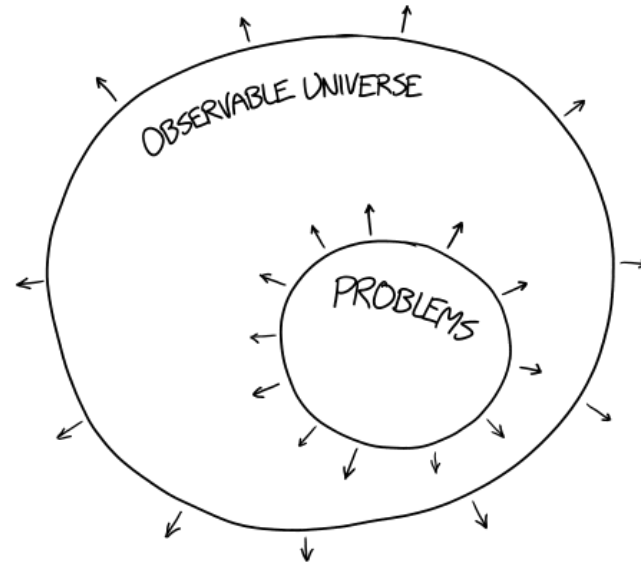
Concluding remarks

→ The aim of the lecture was to:

- ◆ summarize the last decades in evolution of computing hardware,
- ◆ get you to realize that performance is handled not only by the compiler and libraries. The story is far more complicated,
- ◆ give you rough estimates on possible throughputs and performance showstoppers.
- ◆ give you an overview of the computing landscape,

→ When facing a huge data flow, we can't afford using only a fraction of the hardware we have.

→ Performance is complicated business. When in doubt, look to the specification.. or write a test program. Be brave and bold.



Thank you!

Questions?

Catch me at ISOTDAQ until tomorrow morning

or

mail pawel.szostek@cern.ch