Performance and Performance modeling

What is HPC?

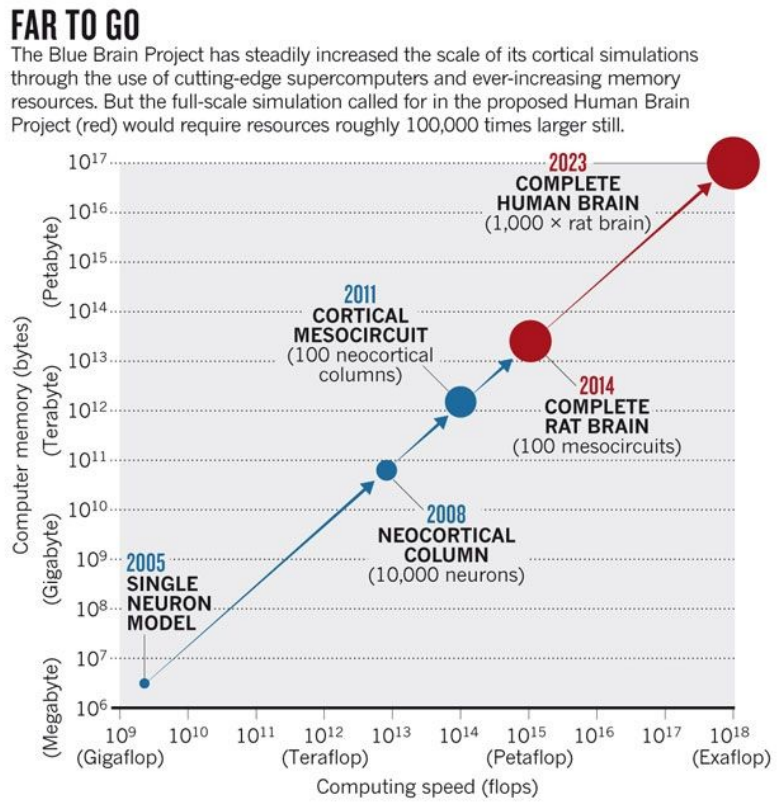
What is HPC?

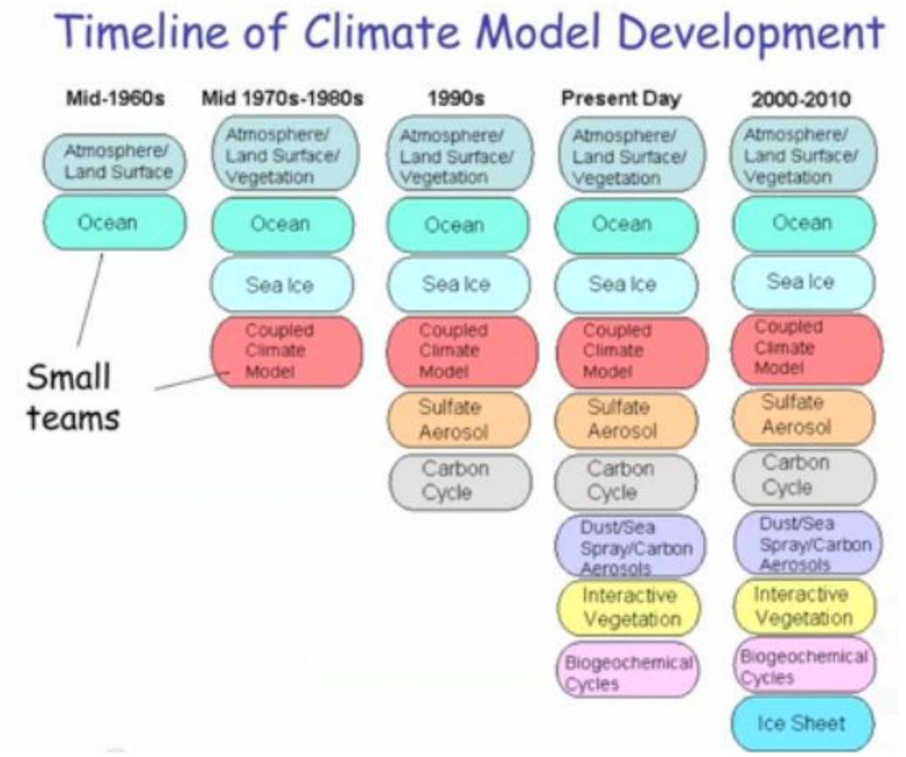
It’s High Performance Computing...

… High Performance means minimize Time To Solution

Why?

Why More Performance?





HPC First Principles: Latency and Throughput

Latency: time to complete a operation
Throughput: how many operations per time

Water hose analogy:

- Latency is how long it takes to water to go through the pipe
- Throughput is the amount of water the pipe is outputting

Remarks:
- Cutting the pipe in half halves latency but throughput remains unchanged
- Adding a pipe does not change the latency but increases the throughput

HPC First Principles: Latency and Throughput

- Easily confused (they refer to speed)
- Often contradict each other
- **Relate to each other with Little's Law**

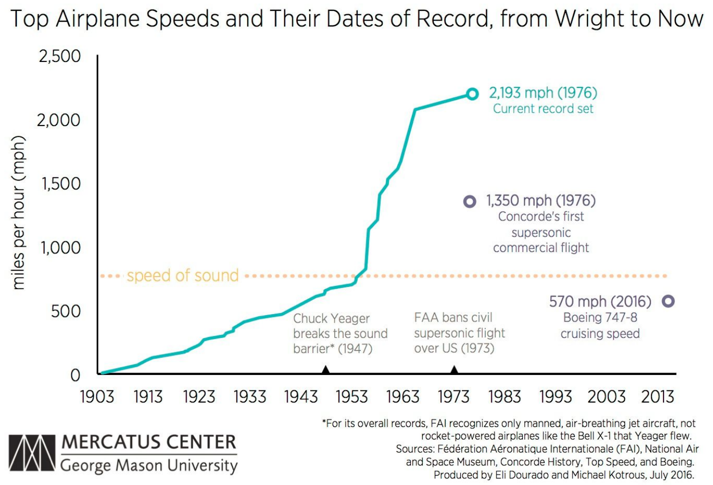
$$L = \wedge W$$

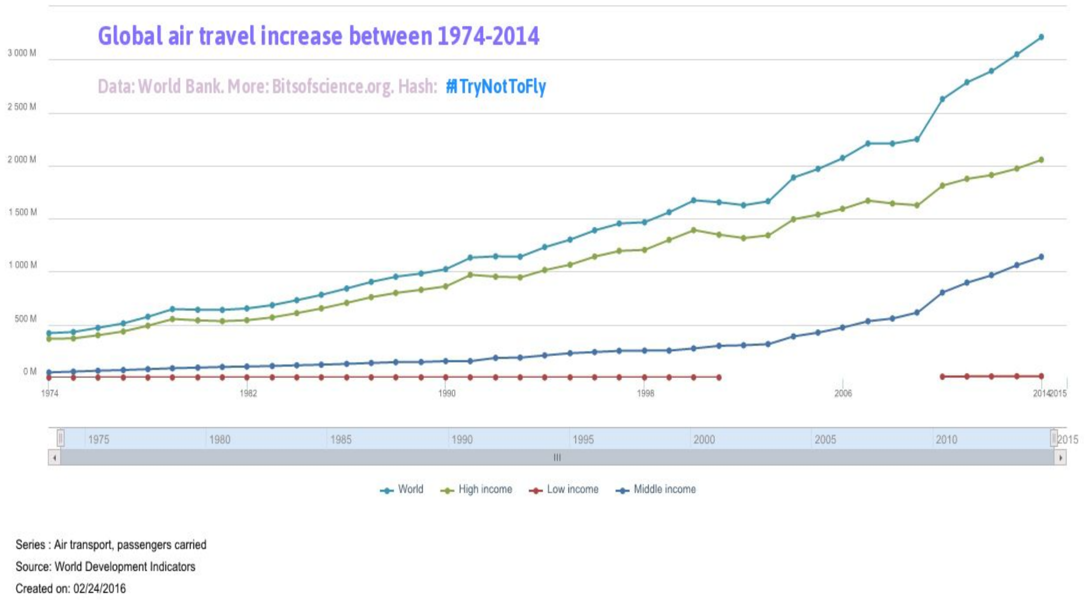
L = average number of items in the queuing system
W = average waiting time in the system for an item
$\wedge$ = average number of items arriving per unit time

Which one is the most important?

HPC First Principles: Latency and Throughput

In general, latency reduction hits physical limits





Planes do not fly faster but there are much more in the sky…

Latency and Throughput for CPUs

Latency (in seconds or cycles):

how long it takes before the next dependent operation can start

dependant operations performance is limited by latency

Throughput (in Instruction Per Cycles or Per Seconds): number of independent operations per time unit

independent operations performance is limited by throughput

Little’s Law

parallelism = latency * throughput

Or

latency = parallelism/throughput

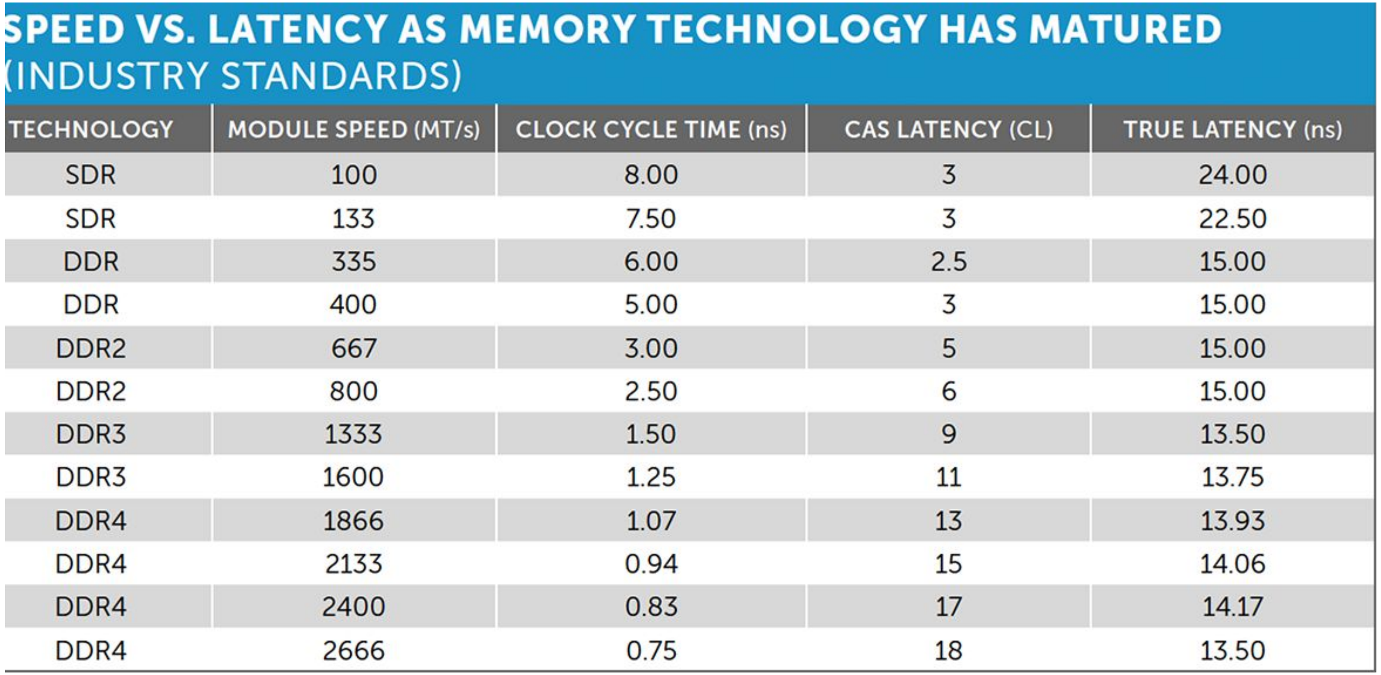
Or

throughput = parallelism/latency

i.e. latency is covered with parallelism

HPC First Principles: Memory Latency and Throughput

Memory bandwidth goes up (nice!) but latency does not go down (not nice).



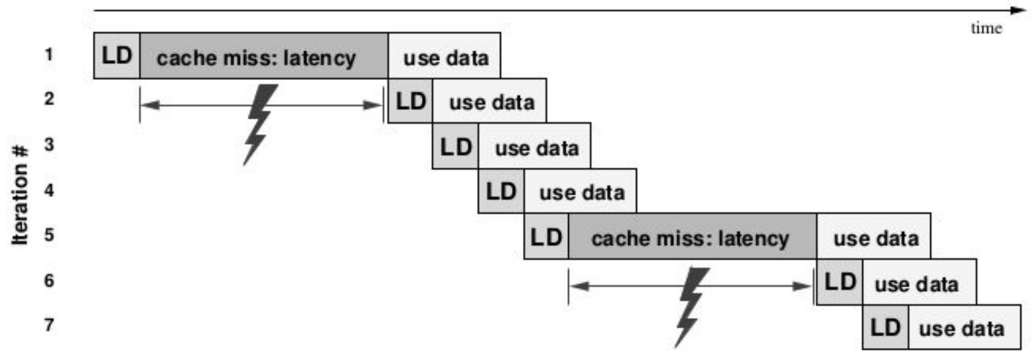
SPEED VS. LATENCY AS MEMORY TECHNOLOGY HAS MATURED
(INDUSTRY STANDARDS)

TECHNOLOGY	MODULE SPEED (MT/s)	CLOCK CYCLE TIME (ns)	CAS LATENCY (CL)	TRUE LATENCY (ns)
SDR	100	8.00	3	24.00
SDR	133	7.50	3	22.50
DDR	335	6.00	2.5	15.00
DDR	400	5.00	3	15.00
DDR2	667	3.00	5	15.00
DDR2	800	2.50	6	15.00
DDR3	1333	1.50	9	13.50
DDR3	1600	1.25	11	13.75
DDR4	1866	1.07	13	13.93
DDR4	2133	0.94	15	14.06
DDR4	2400	0.83	17	14.17
DDR4	2666	0.75	18	13.50

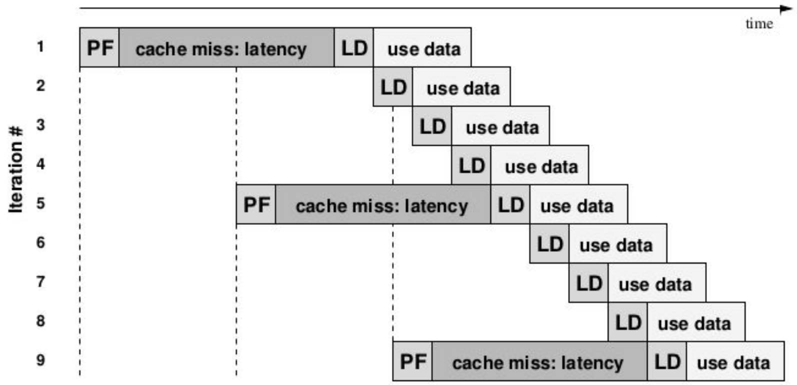
http://www.crucial.com/usa/en/memory-performance-speed-latency

HPC First Principles: Memory Latency and Throughput

Memory Prefetchers: increasing throughput by maximizing locality



Latency bound



Bandwidth bound

CPU needs to know where to fetch data:

- Contiguous accesses will maximize throughput
- Non-contiguous accesses are latency bound

Maths Operations Latency and Throughput

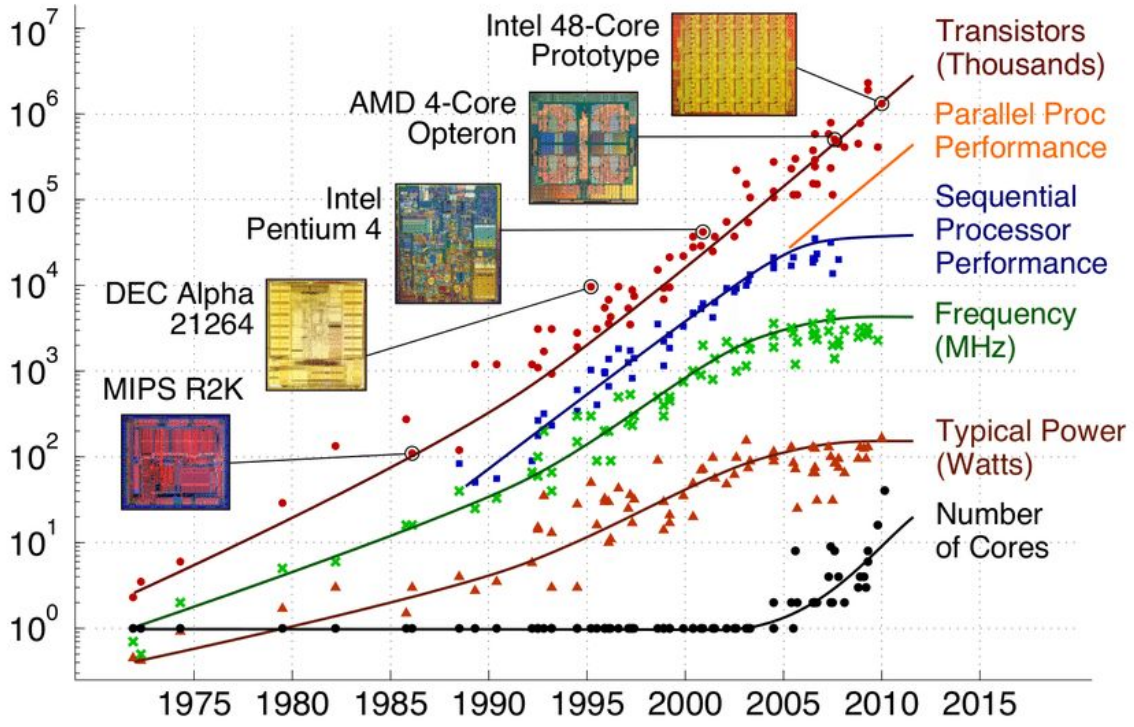
	Broadwell		KNL		Kepler GPU	
	Lat (C)	TP (IPC)	Lat (C)	TP (IPC)	Lat (C)	TP (IPC)
Add	3	1	2	2	9	?
Multiply	3	2	7	2	9	?
Fused MAdd	5	2	6	2	9	32 (?)
Division	10-14	0.05	32	0.031	141	?
Sqrt	10-23	0.2	38	0.063	181	?
SinCos	52-124	?	50-250	0.006	18	?
Atan	97-147	?	125-265	0.001	?	?
Log	92	?	190	0.005	22	?

As a reference, a DRAM memory access is about 200 cycles (memory wall).

CPUs are designed to perform multiply-adds, but that’s about it…

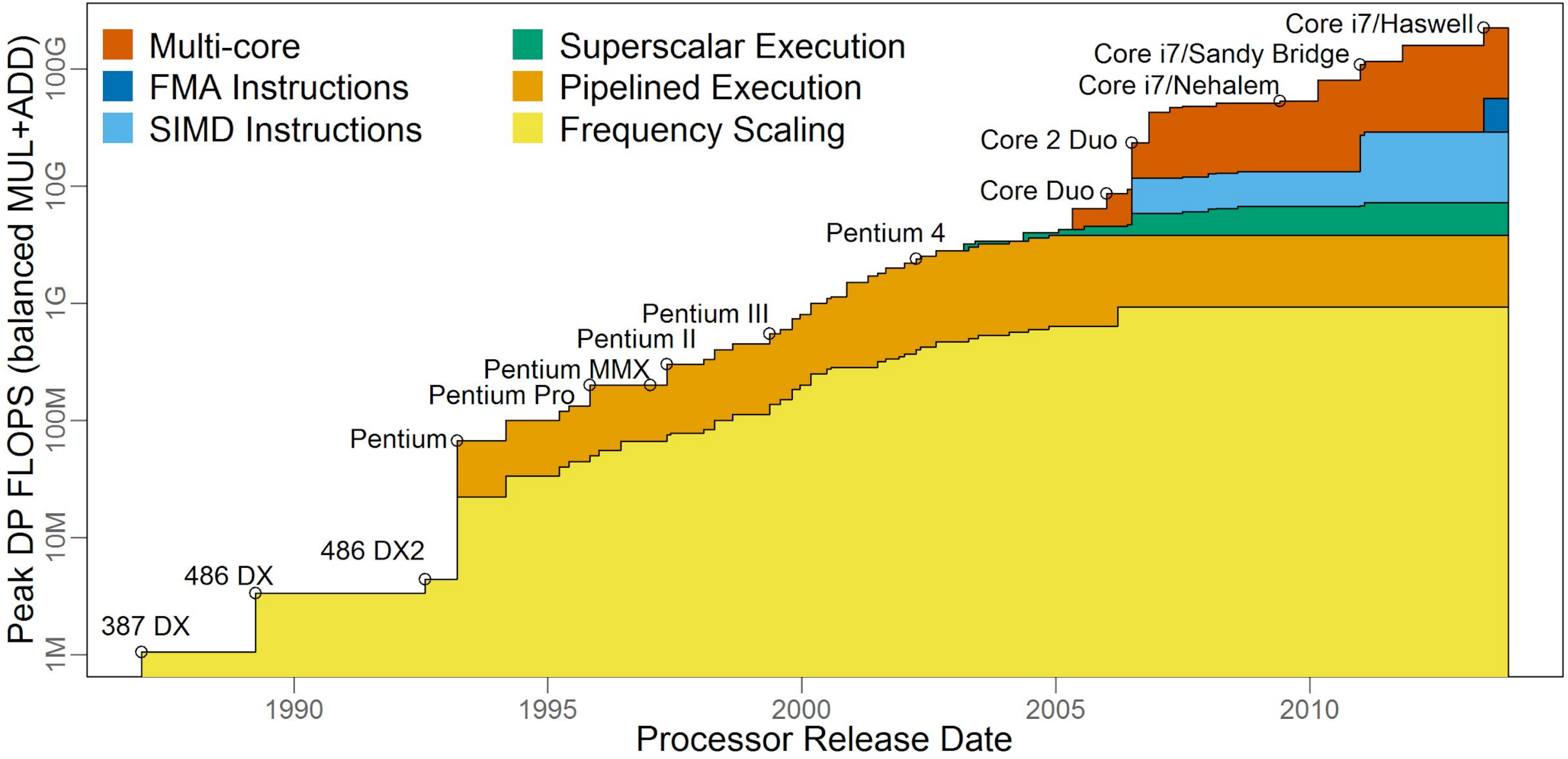
use as many FMAs as possible.

HPC First Principles: CPU Latency and Throughput

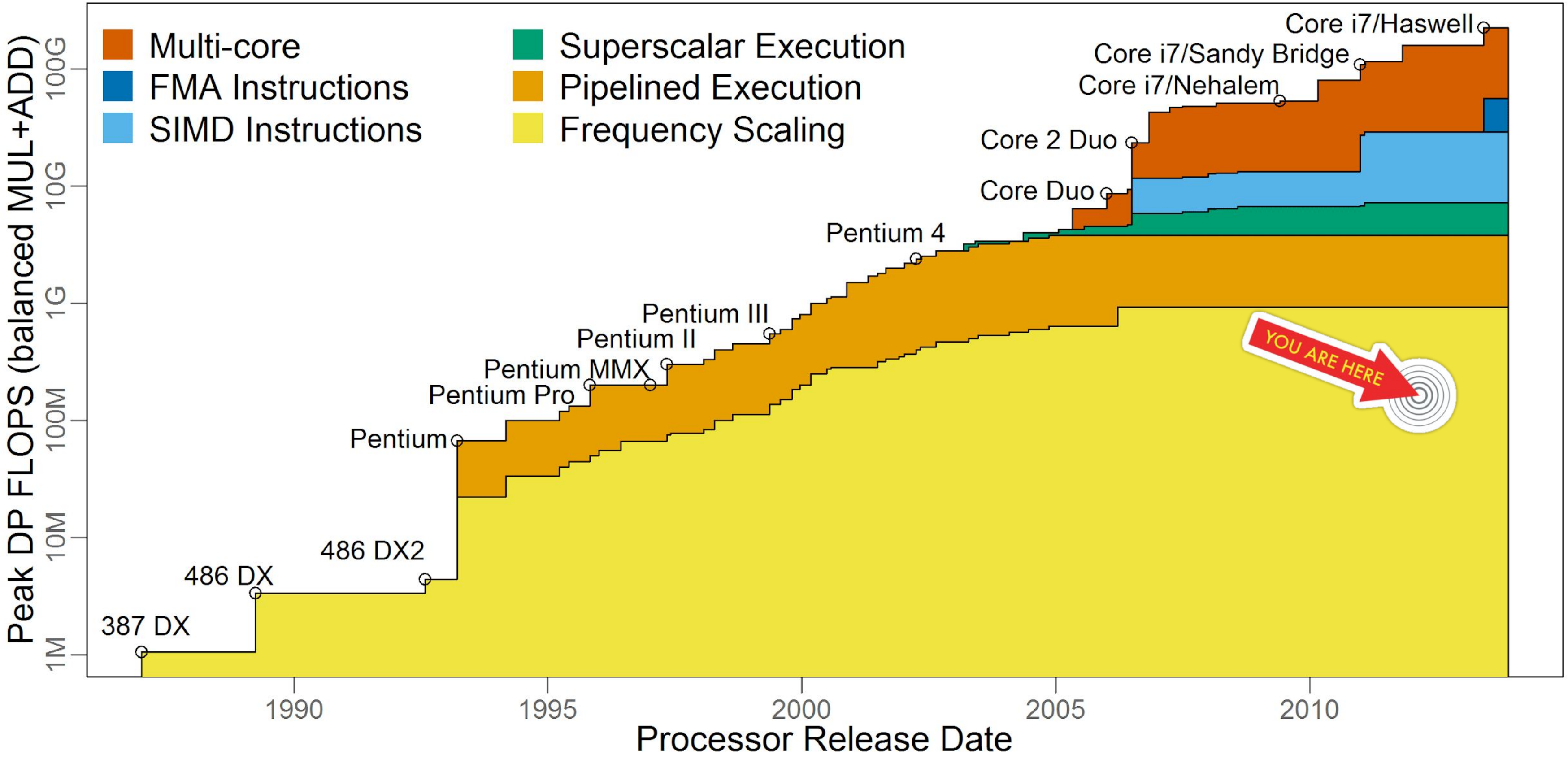


CPUs frequency has hit a limit more than 10 years ago but performance stills goes up thanks to Parallel Performance

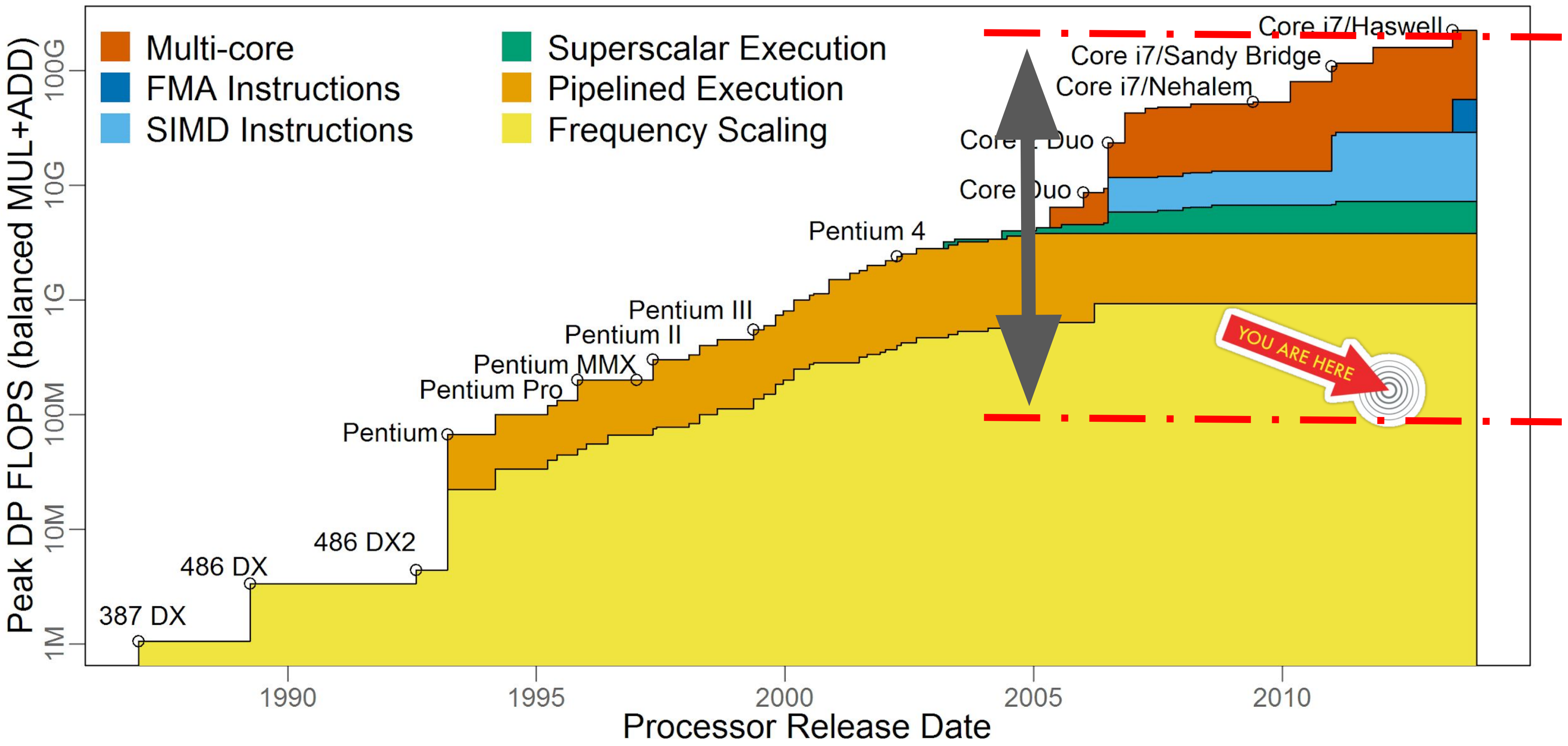
CPU Peak FP Performance



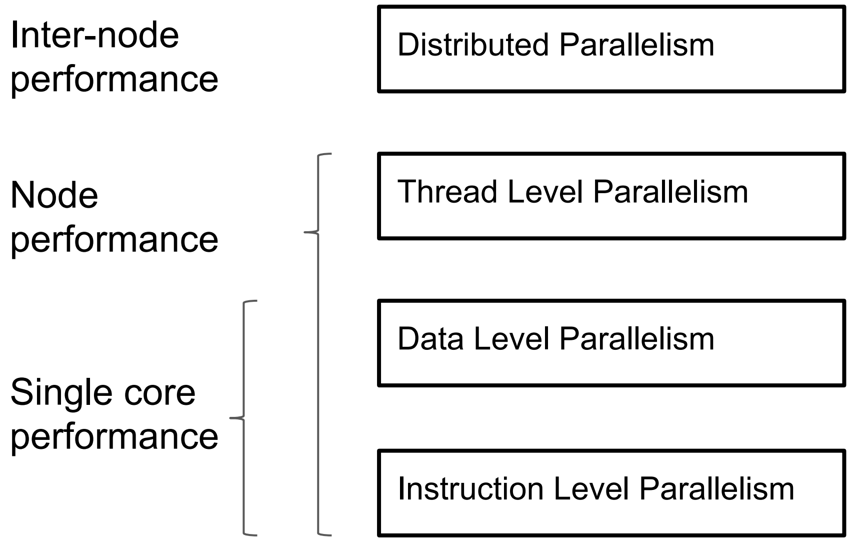
CPU Peak FP Performance - The Ninja Gap



CPU Peak FP Performance - The Ninja Gap



Different levels of parallelism



MPI
Expected speedup: number of nodes

OpenMP, POSIX pThreads
Expected speedup: number of cores per node

SIMD. Ex: SSE, AVX, AVX2, AVX512
Expected speedup: 2x to 8x (AVX512)

Pipelining, Superscalar execution, Out of Order Execution...
Expected speedup: 10s of x

Performance is based on the duplication of resource and is harnessed by parallelism

Instruction Level Parallelism

Pipelining

Superscalar architecture

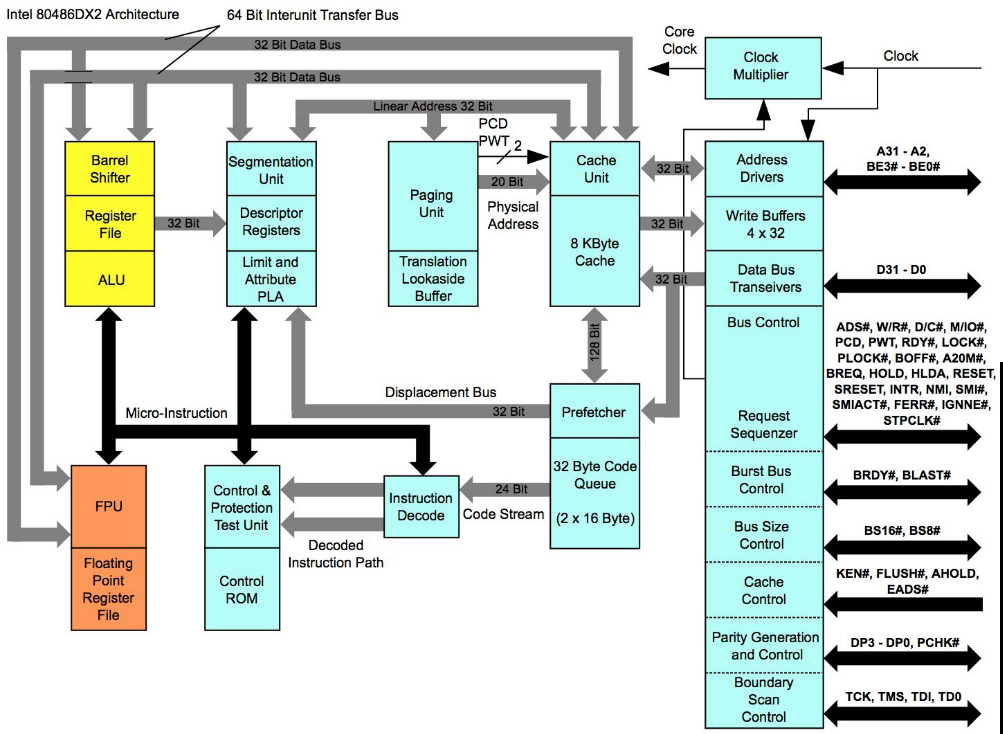
Out-Of-Order Execution

Speculative execution

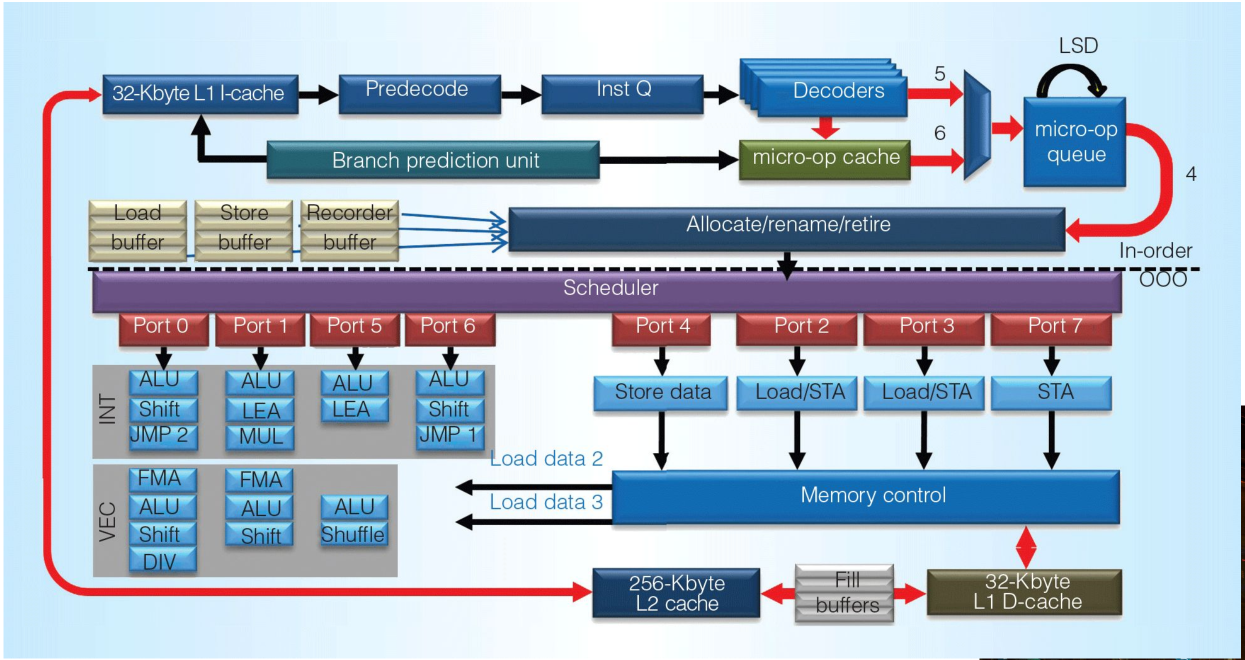
MELTDOWN SPECTRE

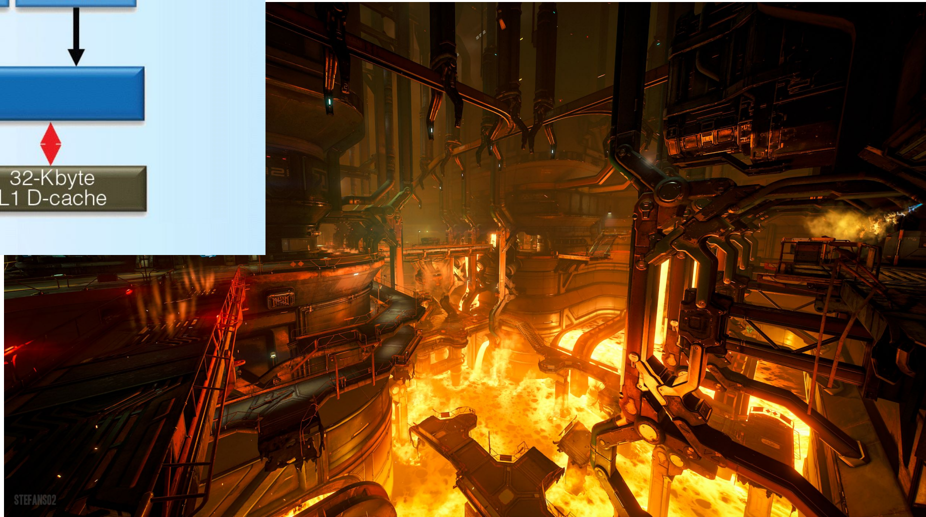
The goal is to maximize CPU utilization and avoid stalls

Vintage CPU Architecture: Intel’s 486 DX2



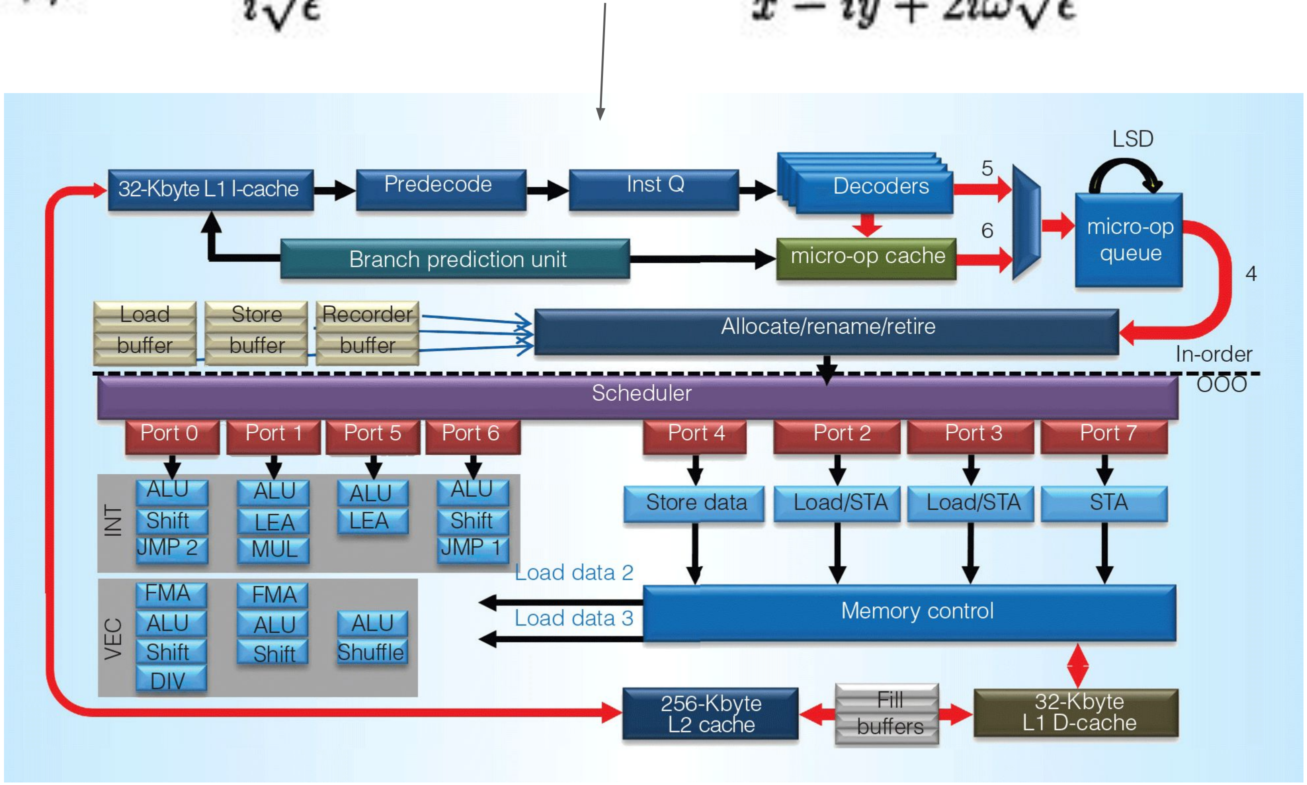
Modern CPU Architecture: Intel's Skylake



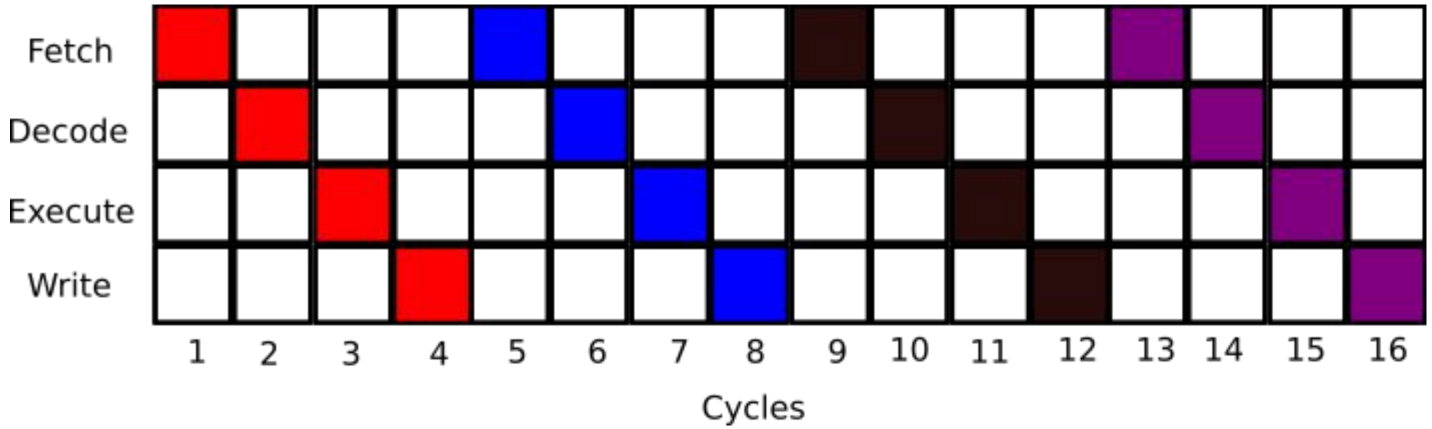


Bottom-up: mapping to the architecture

$$I^*_{\omega,1/2} = \frac{(1-\epsilon^2)\kappa_0\omega}{\imath\sqrt{\epsilon}} \ln \frac{\frac{1-\epsilon}{1+\epsilon}x - \imath\frac{1+\epsilon}{1-\epsilon}y + 2\imath\sqrt{\epsilon}\sqrt{\omega^2 + \frac{x^2}{(1+\epsilon)^2} + \frac{y^2}{(1-\epsilon)^2}}}{x - \imath y + 2\imath\omega\sqrt{\epsilon}}.$$



Non-pipelined Processor



- 16 cycles to execute 4 instructions, **instruction latency is 4 cycles**
- **Throughput is ¼ Instructions Per Cycles**

Pipelining

Pipelining

Ford’s assembly line

Pipelining

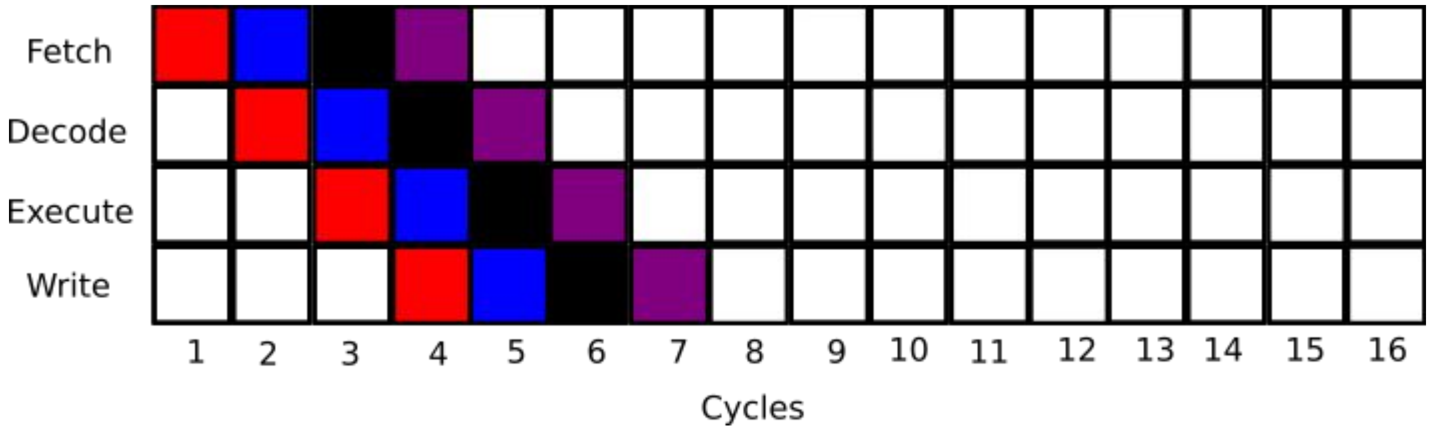
Ford’s assembly line

Ford had been trying to increase his factories’ productivity for years. The workers (...) arranged the parts in a row on the floor, put the under-construction auto on skids and dragged it down the line as they worked. **Ford broke the Model T’s assembly into 84 discrete steps and trained each of his workers to do just one**.

The most significant piece of Ford’s efficiency crusade was the assembly line.

https://www.history.com/this-day-in-history/fords-assembly-line-starts-rolling

Pipelining



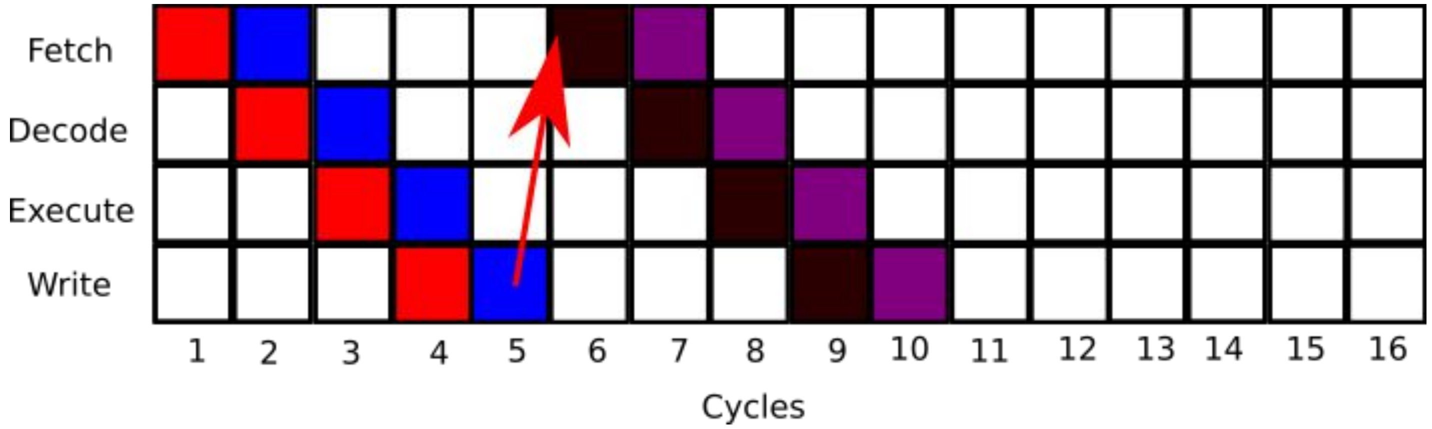
- 16 cycles to execute 4 instructions broken down into 4 “stages”
- Instruction latency is STILL 4 cycles
- Throughput is 4/7 (~ 1/2) IPC, almost 2X compared to non-pipelined

Pipelining

Like the Ford’s assembly line, instructions are broken down in many small steps (stages)

- Increased IPC through increased parallelism
- Smaller stages means increased frequency which unlocked the frequency era

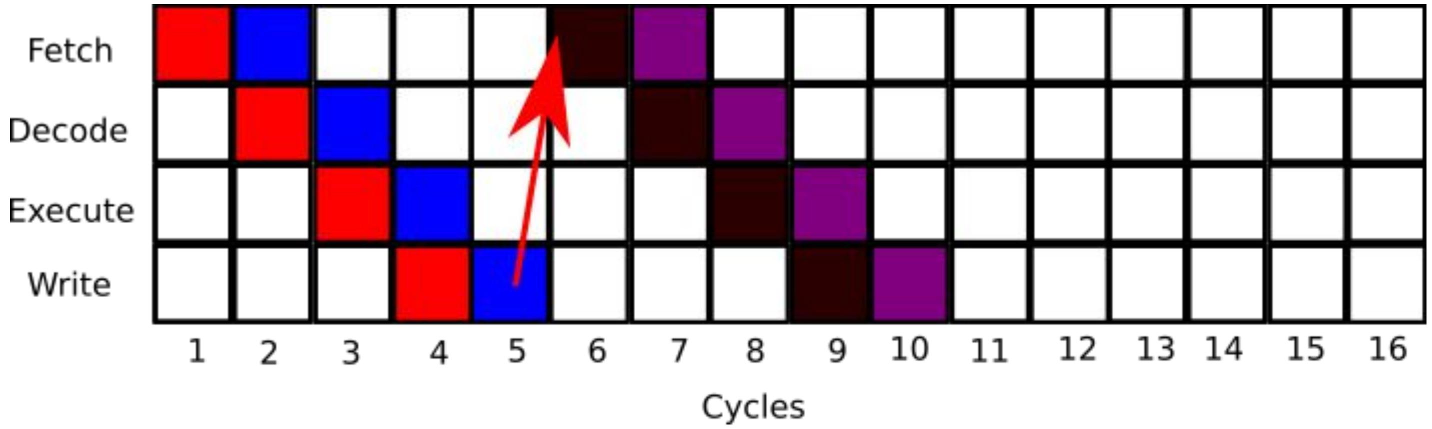
But there’s a price to pay: deep pipelines can easily stall



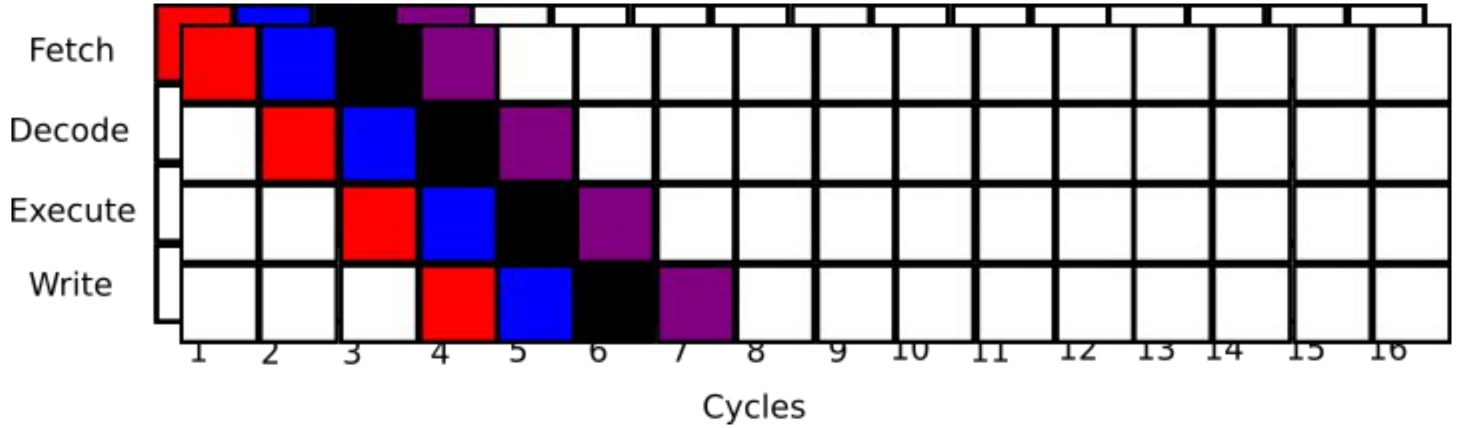
Pipelining

Stalls (bubbles) happen when the pipeline cannot advance properly

- Most probable cause is instruction dependency
- The CPU is waiting for a resource (e.g. read/write in the memory)



Superscalar Architecture



- Multiple pipelines to increase Instructions Per Cycles
- Can be spoiled by data, control, structural hazards and Multi-cycle instructions.

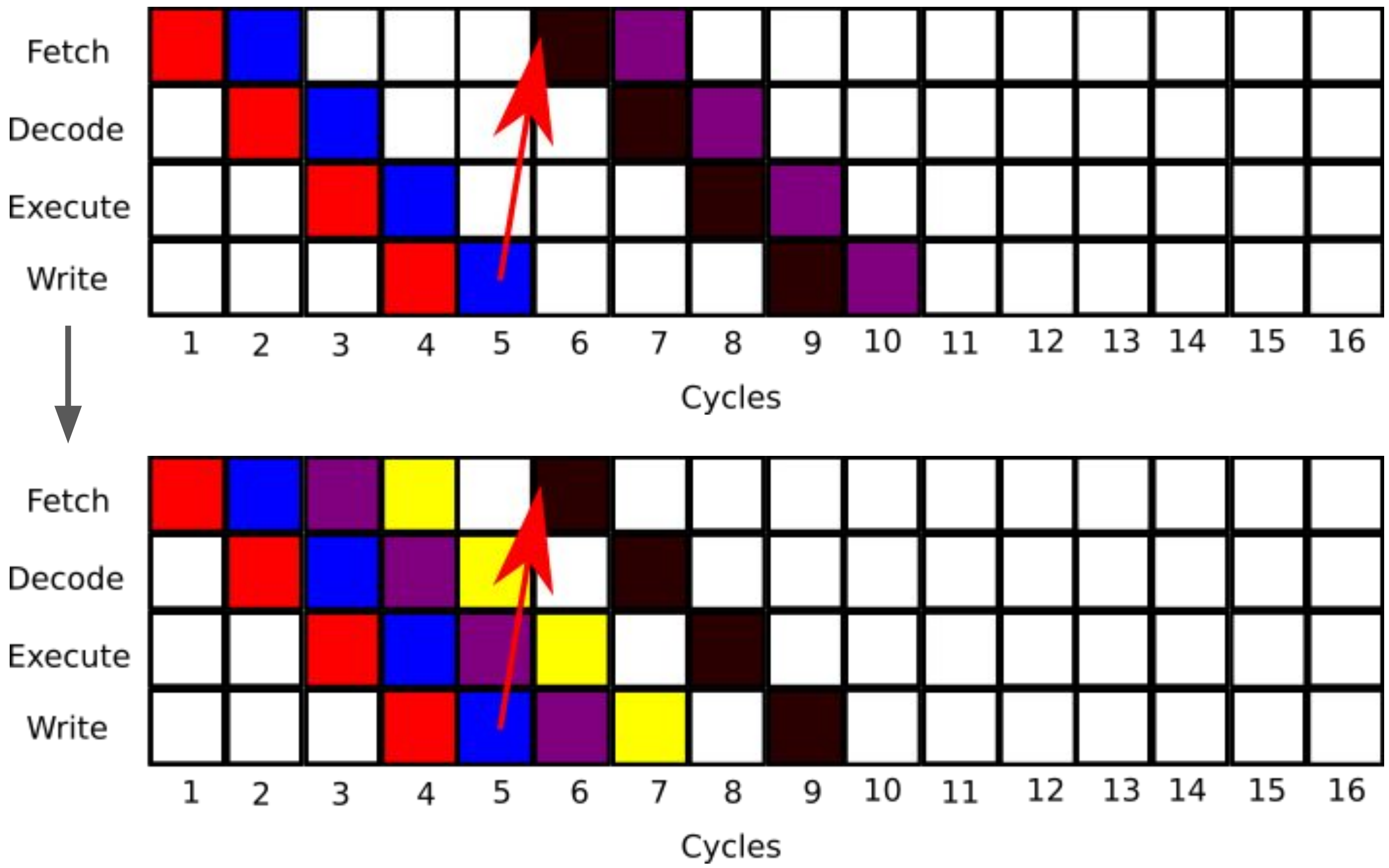
Out-Of-Order Execution

In-Order Execution: first instruction **in** is first instruction executed

Out-Of-Order Execution: first instruction **ready** is first instruction executed

- Operations are reordered
- Operations without dependencies are executed when the execution engines are ready
- Results are immediately available when needed (if the prediction was correct)

Out-Of-Order execution

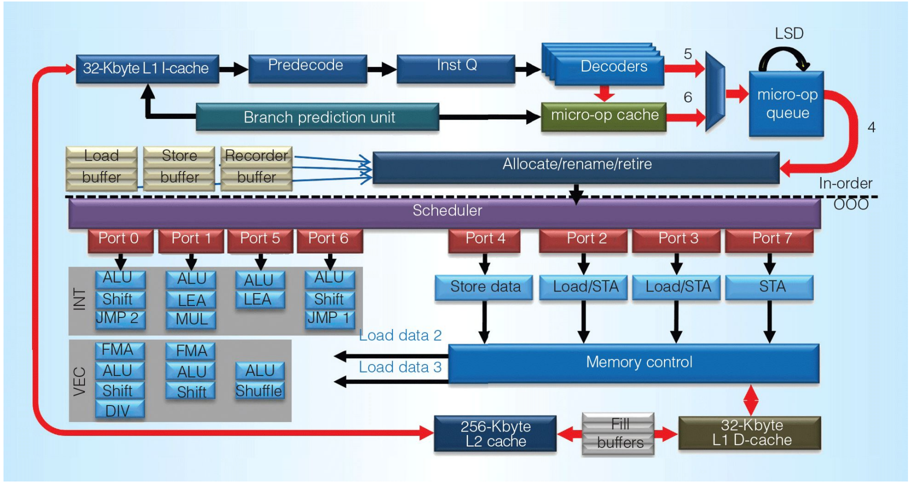


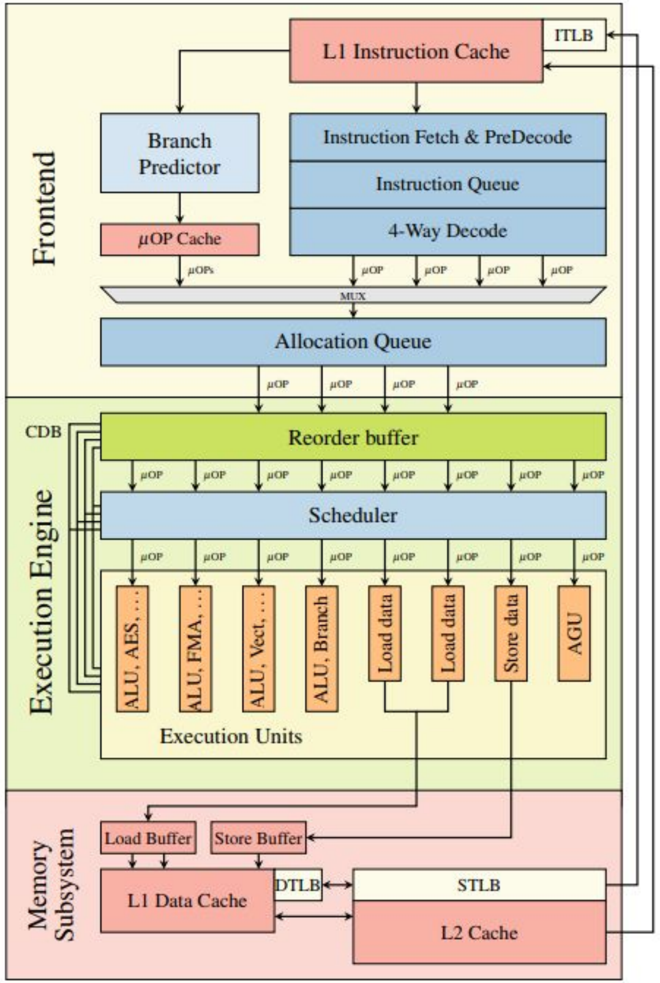
Modern CPU architecture

CPU Execution Engines:

- Pipelined
- Out-Of-Order execution
- Superscalar

ILP keeps the pipelines full if parallelism is extracted





Speculative Execution

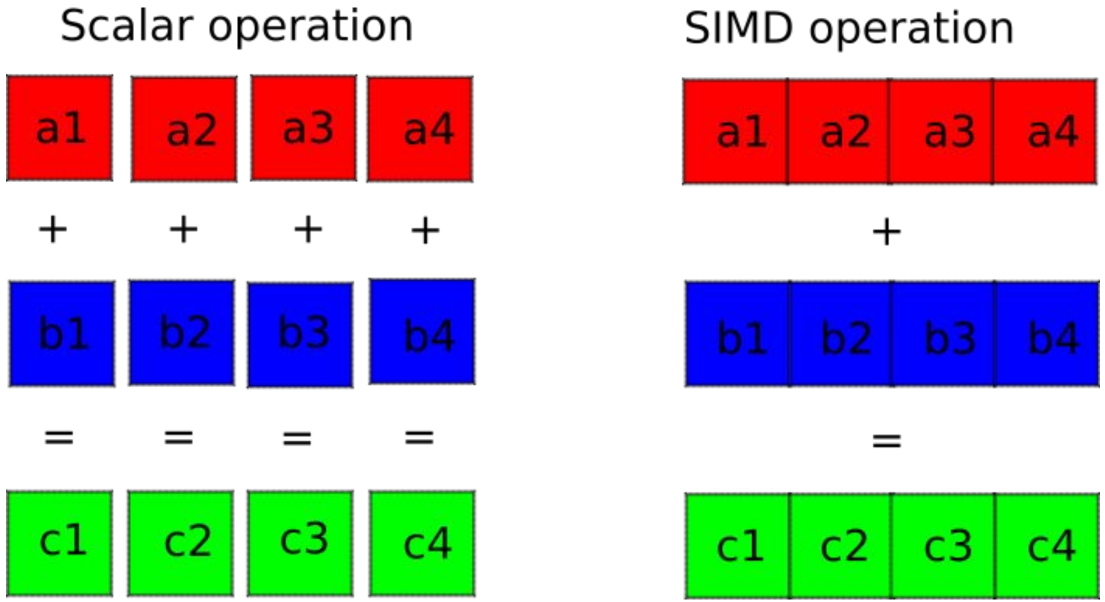
Speculative execution: tentative execution despite dependencies

MELTDOWN SPECTRE

Very interesting read: https://meltdownattack.com/meltdown.pdf

Data Level Parallelism (SIMD)

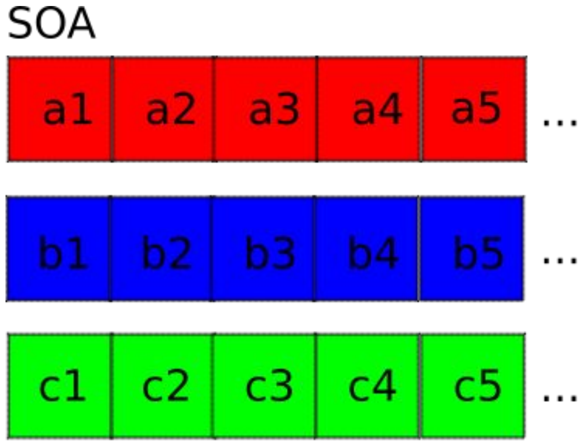
SIMD: Same Instruction Multiple Data



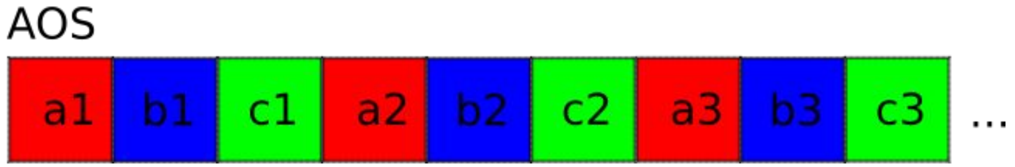
- Throughput is multiplied by the vector size
- AVX: 256 bits, AVX512: 512 bits, GPUs: 2048 bits

HPC Data Structures: SoA vs AoS

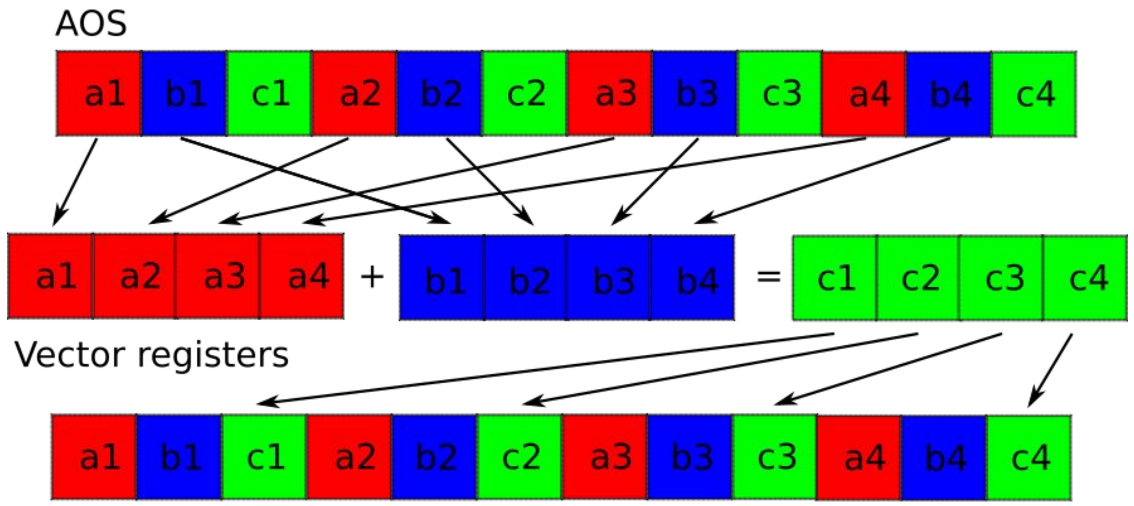
```
struct {
    float a[N];
    float b[N];
    float c[N];
} SoA;
```



```
struct {
    float a, b, c;
} AoS[N];
```

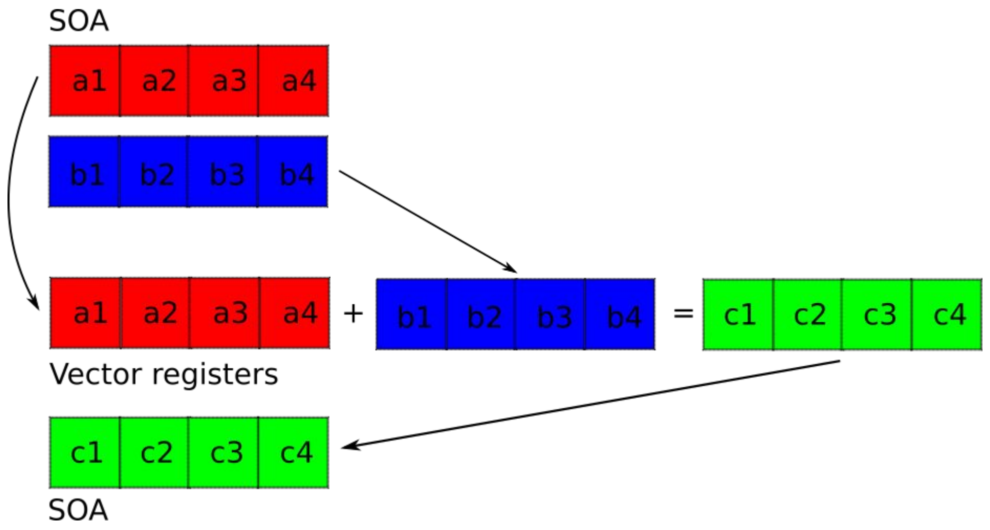


HPC Data Structures: SoA vs AoS



- Information needs to be shuffled to and from the vector registers before and after the vector operations
- Compilers will have a very hard time optimizing/vectorizing the code
- Cache “unfriendly”

HPC Data Structures: SoA vs AoS



- 1-to-1 correspondence between the cache lines and the registers
- No shuffle/gather/scatter needed

DLP: Compilation and Assembly

Compilation flags (for gcc):

Standard Optimization flags:

- -O0: default, no optimization
- -O(1): enables optimization to reduce time to solution (for instance autovectorization/DLP)
- -O2: even more aggressive optimization

Standard Debug flags:

- -g0: no debug information
- -g1: minimal debug information
- -g: default debug information
- -g3: maximal debug information

Special flag: -Ofast

- sets -O3
- Sets --fast-math: extra performance boost you need but use with case
    - breaks strict IEEE-754 (floating point) compliance
    - Reorder code to improve ILP
    - enables fast approximation of transcendental functions (sqrt, div...)

DLP: Compilation and Assembly

Check the assembly for vector instructions

- Compilers options (-O)
- Misalignments,
- Standards were not designed for HPC,
- ...

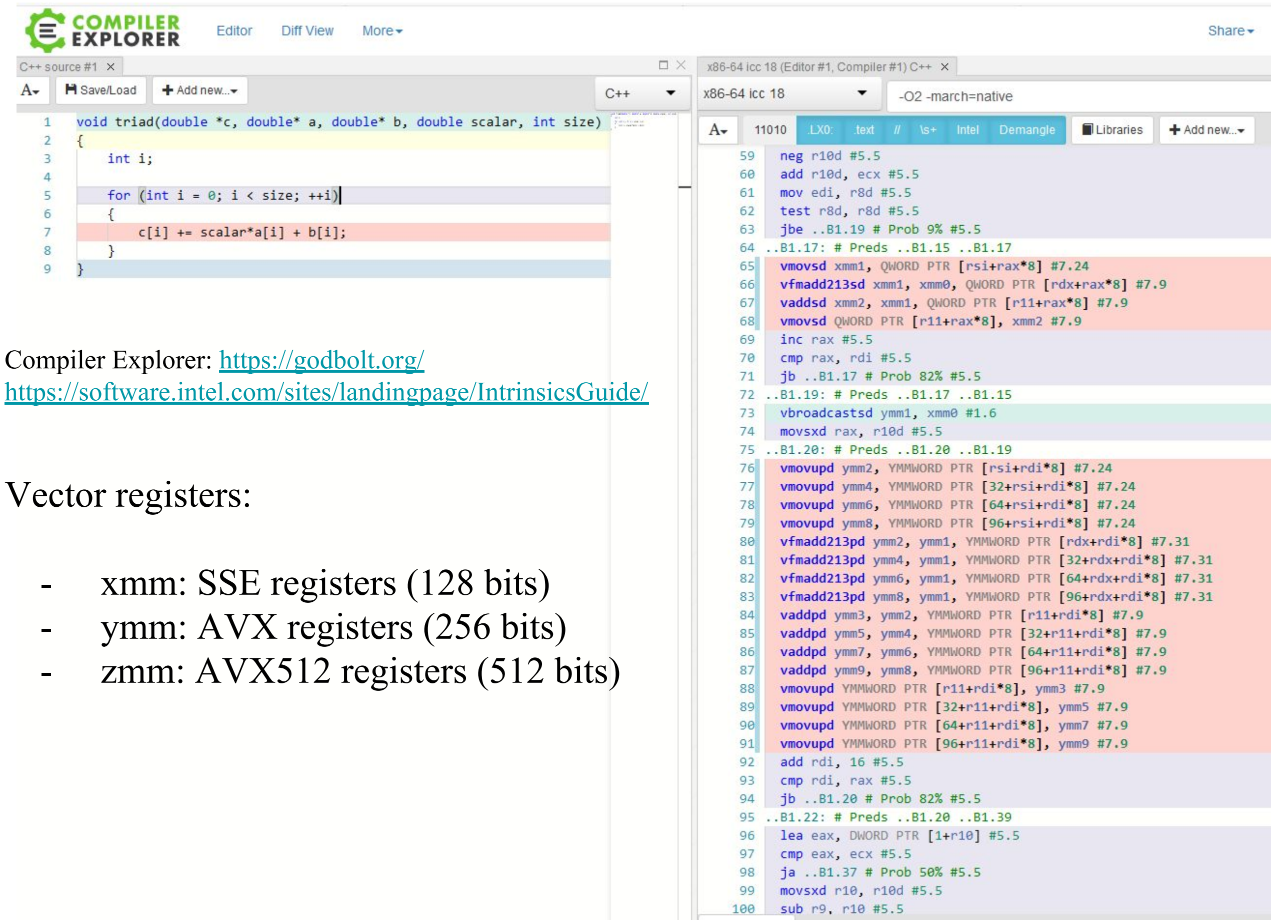
Example:

vfmadd213pd

v: vector instruction
p: packed
d: double

vfmadd213ss

first s: scalar
second s: float (single)



```
-O2 -march=native
```

```
void triad(double *c, double* a, double* b, double scalar, int size)
{
    int i;

    for (int i = 0; i < size; ++i)
    {
        c[i] += scalar*a[i] + b[i];
    }
}
```

```
  neg r10d #5.5
  add r10d, ecx #5.5
  mov edi, r8d #5.5
  test r8d, r8d #5.5
  jbe ..B1.19 # Prob 9% #5.5
..B1.17: # Preds ..B1.15 ..B1.17
  vmovsd xmm1, QWORD PTR [rsi+rax*8] #7.24
  vfmadd213sd xmm1, xmm0, QWORD PTR [rdx+rax*8] #7.9
  vaddsd xmm2, xmm1, QWORD PTR [r11+rax*8] #7.9
  vmovsd QWORD PTR [r11+rax*8], xmm2 #7.9
  inc rax #5.5
  cmp rax, rdi #5.5
  jb ..B1.17 # Prob 82% #5.5
..B1.19: # Preds ..B1.17 ..B1.15
  vbroadcastsd ymm1, xmm0 #1.6
  movsxd rax, r10d #5.5
..B1.20: # Preds ..B1.20 ..B1.19
  vmovupd ymm2, YMMWORD PTR [rsi+rdi*8] #7.24
  vmovupd ymm4, YMMWORD PTR [32+rsi+rdi*8] #7.24
  vmovupd ymm6, YMMWORD PTR [64+rsi+rdi*8] #7.24
  vmovupd ymm8, YMMWORD PTR [96+rsi+rdi*8] #7.24
  vfmadd213pd ymm2, ymm1, YMMWORD PTR [rdx+rdi*8] #7.31
  vfmadd213pd ymm4, ymm1, YMMWORD PTR [32+rdx+rdi*8] #7.31
  vfmadd213pd ymm6, ymm1, YMMWORD PTR [64+rdx+rdi*8] #7.31
  vfmadd213pd ymm8, ymm1, YMMWORD PTR [96+rdx+rdi*8] #7.31
  vaddpd ymm3, ymm2, YMMWORD PTR [r11+rdi*8] #7.9
  vaddpd ymm5, ymm4, YMMWORD PTR [32+r11+rdi*8] #7.9
  vaddpd ymm7, ymm6, YMMWORD PTR [64+r11+rdi*8] #7.9
  vaddpd ymm9, ymm8, YMMWORD PTR [96+r11+rdi*8] #7.9
  vmovupd YMMWORD PTR [r11+rdi*8], ymm3 #7.9
  vmovupd YMMWORD PTR [32+r11+rdi*8], ymm5 #7.9
  vmovupd YMMWORD PTR [64+r11+rdi*8], ymm7 #7.9
  vmovupd YMMWORD PTR [96+r11+rdi*8], ymm9 #7.9
  add rdi, 16 #5.5
  cmp rdi, rax #5.5
  jb ..B1.20 # Prob 82% #5.5
..B1.22: # Preds ..B1.20 ..B1.39
  lea eax, DWORD PTR [1+r10] #5.5
  cmp eax, ecx #5.5
  ja ..B1.37 # Prob 50% #5.5
  movsxd r10, r10d #5.5
  sub r9, r10 #5.5
```

Compiler Explorer: https://godbolt.org/
https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Vector registers:

- xmm: SSE registers (128 bits)
- ymm: AVX registers (256 bits)
- zmm: AVX512 registers (512 bits)

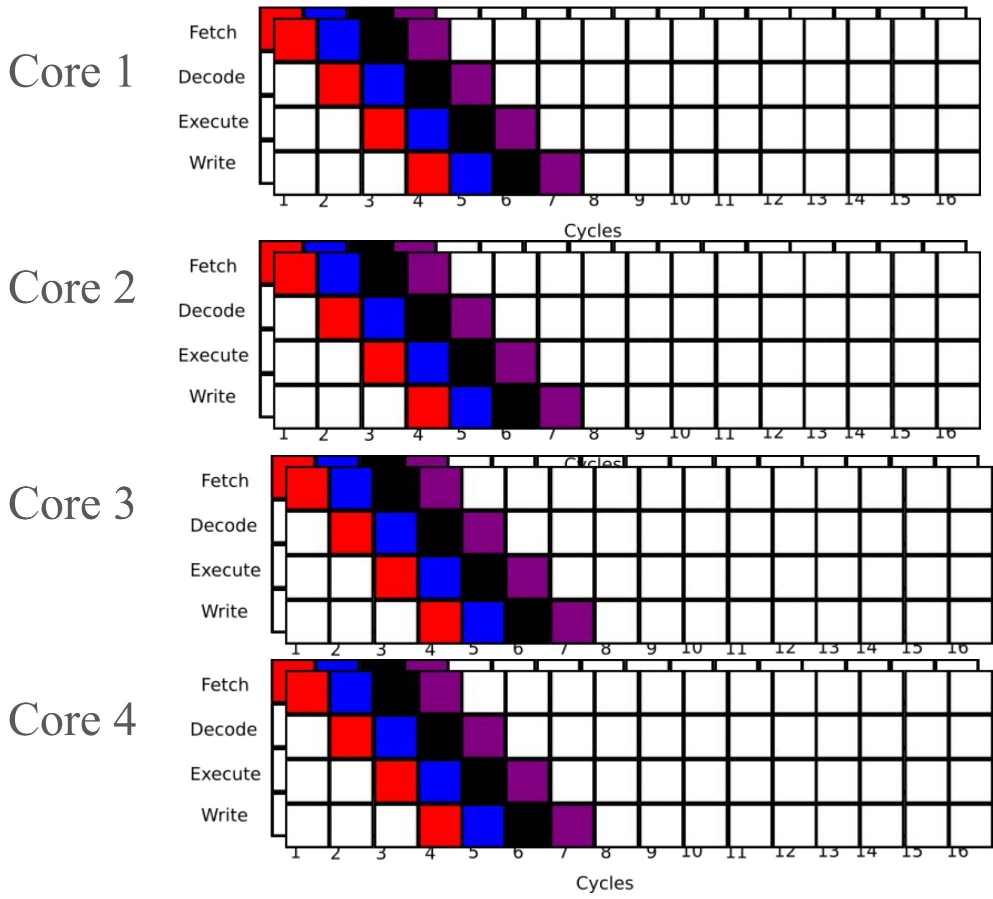
Thread Level Parallelism (multi-many core)

Use of multiple concurrent threads of execution that are inherently parallel

- Increase throughput of multithreaded codes by covering the latencies
- Extremely effective in some cases (GPUs)

However...

- (just) resource (cores) duplication
- Diminishing returns
- Concurrent programming is very difficult
- Current hardware is not really designed to support TLP (e.g. caches...)



HPC Take Home Message - Node Performance

The goal of HPC is to increase mathematical throughput:

- Latency is NOT going down therefore throughput is increased
- Throughput is going up IF parallelism is increased
- Avoid pipeline stalls by having data “close” to the CPU

HPC kernel optimization focus on extracting parallelism and maximizing data locality

Assessing Performance: The Roofline Model

- There are many different types hardware (CPU, GPUs…)
- There are many different codes (algorithms, kernels…)

Is there a unified model to assess software performance?

Answer: the Roofline Model

- Roofline: an insightful visual performance model for multicore architectures, Williams, S. and Waterman, A. and Patterson, D., Communication to ACM, 2009

- A view of the parallel computing landscape, K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, Communication to ACM, 2009

Assessing Performance: The Roofline Model

Software abstraction:

- Kernels can be represented by:
  - Number of mathematical operations (Flops)
  - Number of Data transfers (B)

Hardware abstraction:

- Time To Solution is inversely proportional to:
  - Mathematical throughput (Flops/s)
  - DRAM throughput (B/s)

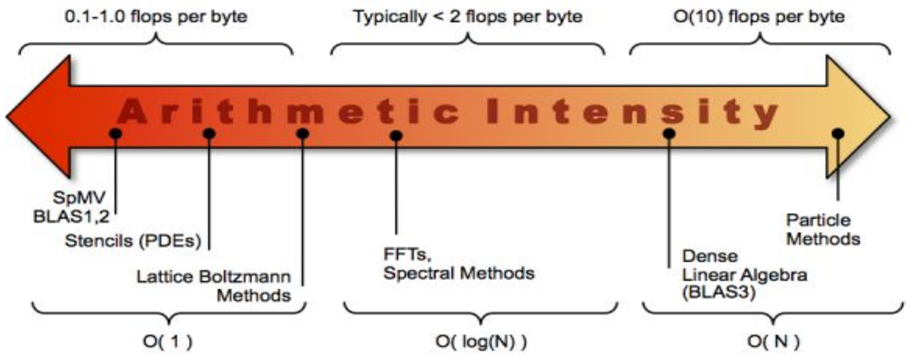
Assessing Performance: The Roofline Model

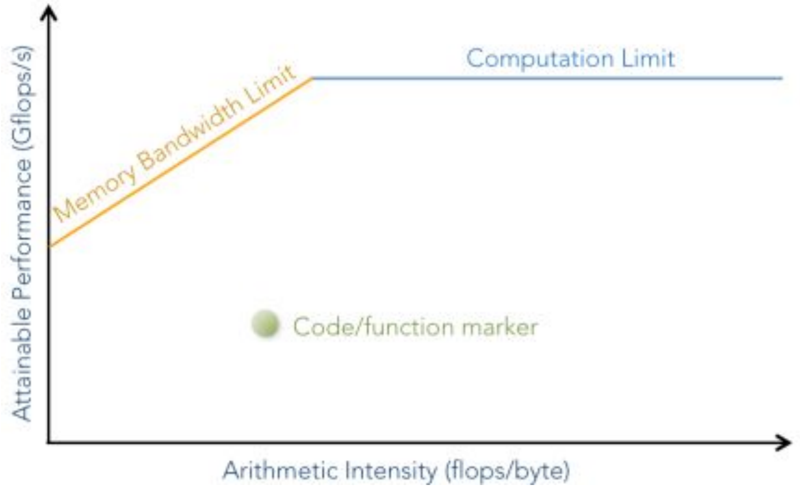
AI = flops/DRAM accesses

$$\text{Perf (flops)} = \min \left\{ \begin{array}{l} \textbf{Peak FP performance} \\ \textbf{Peak BW*AI} \end{array} \right\}$$

- Peak BW is measured with stream (https://www.cs.virginia.edu/stream/)
- Peak FP is measured with dgemm (BLAS 3)

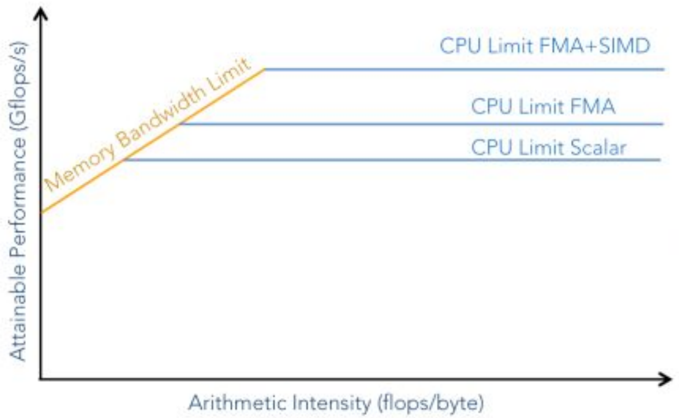
https://picsar.net/performance/roofline-model-application/





Assessing Performance: The Roofline Model

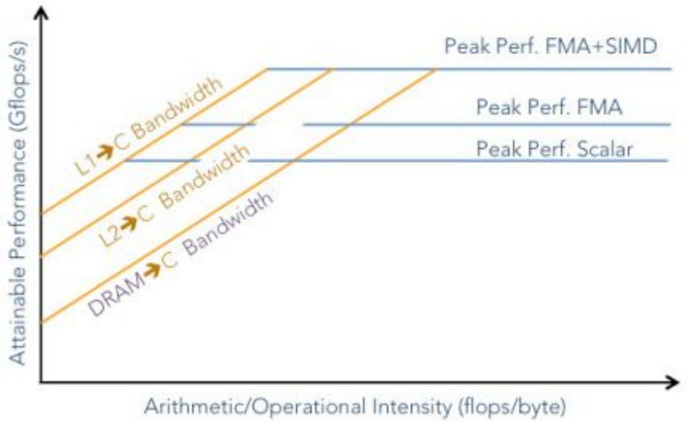
Optimized Roofline:



The attainable system performance is the maximal performance that can be reached by an application:

$$\text{Attainable perf Gflops/s} = \min \begin{cases} \text{Peak performace Gflops/s} \\ \text{Peak Memory Bandwidth} \times \text{Arithmetic Intensity} \end{cases}$$

Cache Aware Roofline:



•Total volume communication across all memory hierarchies relatively to the core:

$$\text{Attainable perf Gflops/s} = \min \begin{cases} \text{Peak performance Gflops/s} \\ \text{Bandwidths to Core} \times \text{Arithmetic Intensity} \end{cases}$$

$$\text{Bandwidths to Core} \begin{cases} \text{Bandwidth from L1 to Core} \\ \text{Bandwidth from L2 to Core} \\ \text{Bandwidth from L3 to Core} \\ \text{Bandwidth from DRAM to Core} \end{cases}$$

CPU Theoretical peak FP performance

(Theoretical) **Peak FP performance** (Gflops/s) **=**

Number of FP ports * (Superscalar architecture)
flops/cycles * (e.g. 2 for FMAs)
Vector size * (DLP)
Frequency * (in Ghz)
Number of cores (TLP)

Example: Skylake 2.5 Ghz, 28 cores (Platinum 8180)

Peak perf = 2 (FMA ports) *2 (FMA flops) * 8 (DP) * 2.5
= 32 flops DP/cycle*2.5 Ghz
= 80 Gflops/s DP per core
= 2240 Gflops/s DP per socket (CPU)

Nina Gap Peak perf = 1*1*1*2.5 = 2.5 Gflops/s, ~1/1000th of the full peak

Assessing Performance: The Roofline Model

Advantages:

- Caps the performance
- Allows visual goals for optimization
- Easily used everywhere

Disadvantages:

- Latencies need to be covered
- Oversimplification
- Node-only
- “Vanilla” version is cache-oblivious
- OI can be hard to compute (e.g. Read For Ownership)

Assessing Performance: examples

Setup:

- Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz
- DDR3 2.133 Ghz

Peak performance (single core): 2*2*4*2.50 = 40 Gflops/s, 34.5 measured
Peak Bandwidth (single core): ~20 GB/s (est.), 13.9 measured

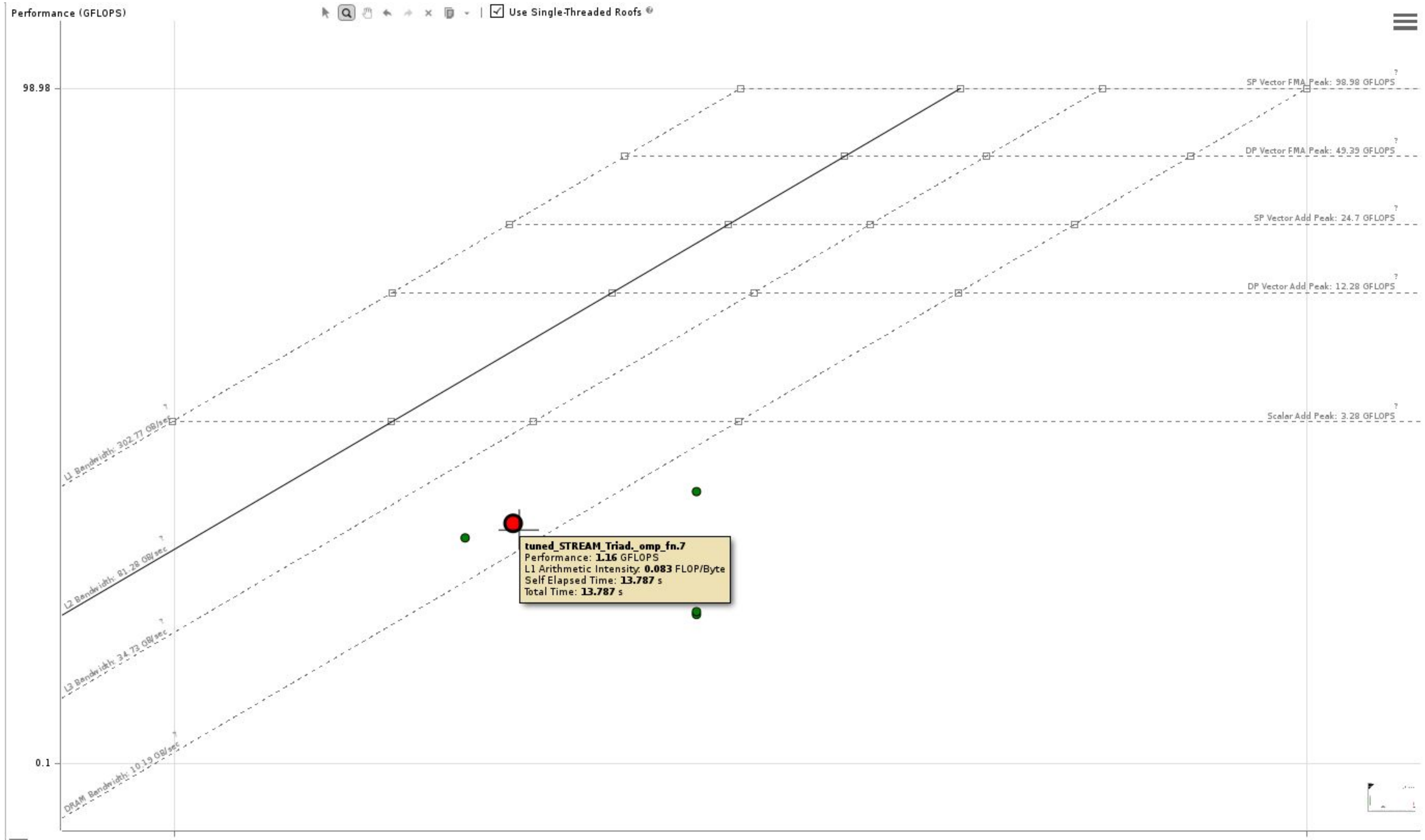
Ridge Point: 2 Flops/B, 2.48 measured

- Triad:

$$c[i] = scalar*a[i] + b[i]$$

Arithmetic Intensity: 2 flops/3*8 Bytes = 2/24 = 12 = 0.083 Flops/B
Performance: 13.9*0.083 = 1.15 Gflops

Assessing Performance: examples



tuned_STREAM_Triad._omp_fn.7
Performance: **1.16** GFLOPS
L1 Arithmetic Intensity: **0.083** FLOP/Byte
Self Elapsed Time: **13.787** s
Total Time: **13.787** s

Assessing Performance: examples

DGEMM: Double precision GEneralized Matrix Multiplication (BLAS 3)

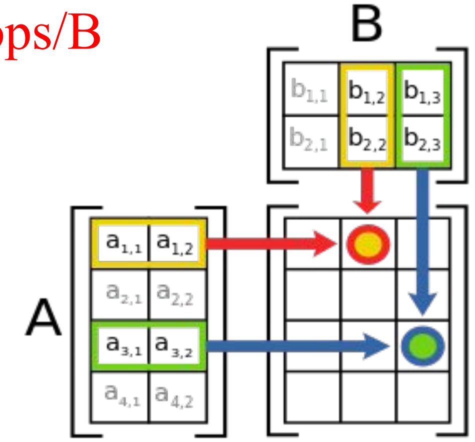
C[i, j] += sum(A[i, k]*B[k, j]) i,j,k = 1..N

Arithmetic Intensity:

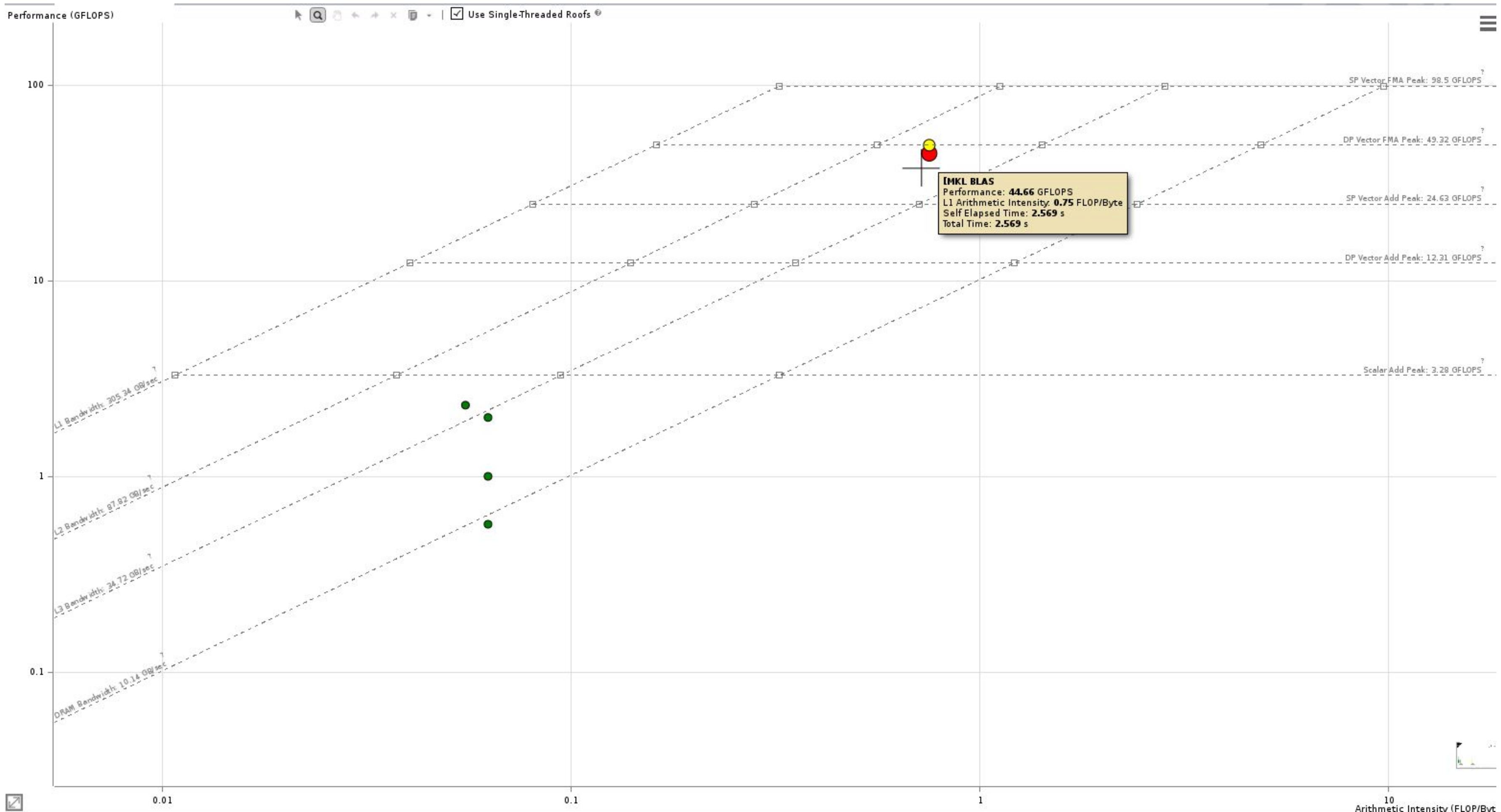
2N^3 Flops, 3N^2*8 DRAM accesses = N/12 flops/B

Performance:

34.5 Gflops if N > 2.35*3



Assessing Performance: examples



[MKL BLAS
Performance: **44.66** GFLOPS
L1 Arithmetic Intensity: **0.75** FLOP/Byte
Self Elapsed Time: **2.569** s
Total Time: **2.569** s

Assessing Performance: Another Go

Flops/cycle can be computed by:

Flops/cycle = number of FMA ports * 2 (FMA) * vec length

Example: Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz

Flops/cycle = 2*2*4 = 16 Flops/cycle Double Precision

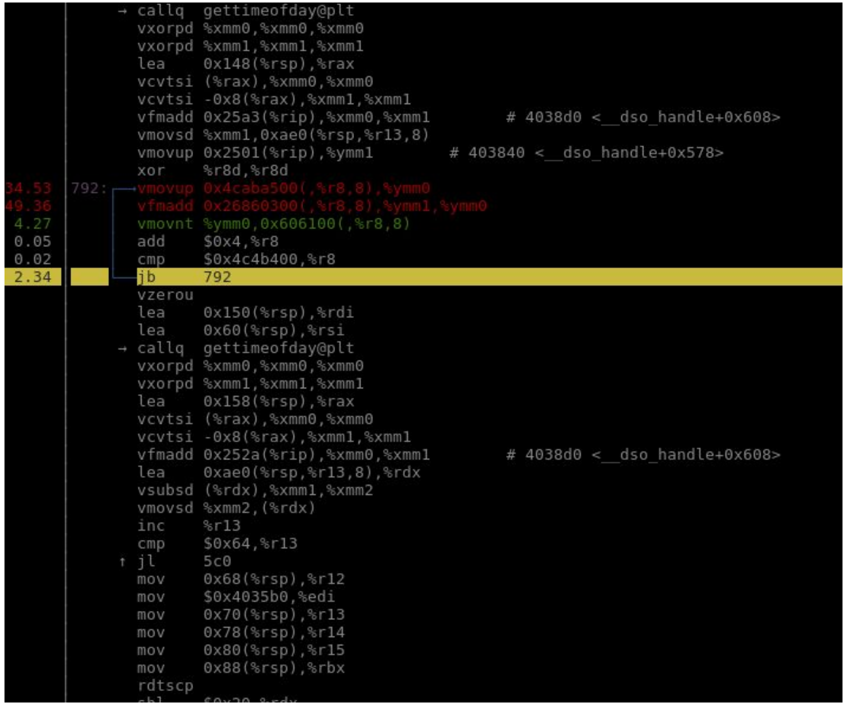
Use RDTSCP on x86_64:

```
static inline unsigned long long cycles()
{
     unsigned long long u;
     asm volatile ("rdtscp; \
               shlq $32, %%rdx; \
               orq %%rdx,%%rax; \
               movq %%rax,%0"\
               :"=q"(u)::"%rax", "%rdx", "rcx");
     return u;
}
```

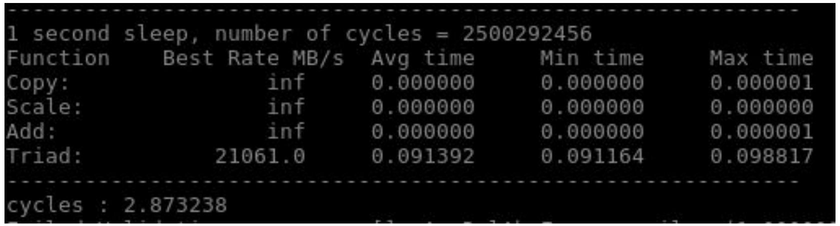
And derive performance by counting flops in the code.

Triad example: 2 (FMA) * 4 (vector size) = 8 flops in 3 cycles

Performance: 8/3 ~ 2.6 Flops/cycle, i.e. 6.5 Gflops/s



```
             → callq   gettimeofday@plt
               vxorpd %xmm0,%xmm0,%xmm0
               vxorpd %xmm1,%xmm1,%xmm1
               lea    0x148(%rsp),%rax
               vcvtsi (%rax),%xmm0,%xmm0
               vcvtsi -0x8(%rax),%xmm1,%xmm1
               vfmadd 0x25a3(%rip),%xmm0,%xmm1        # 4038d0 <__dso_handle+0x608>
               vmovsd %xmm1,0xae0(%rsp,%r13,8)
               vmovup 0x2501(%rip),%ymm1        # 403840 <__dso_handle+0x578>
               xor    %r8d,%r8d
34.53   792:   vmovup 0x4caba500(,%r8,8),%ymm0
49.36          vfmadd 0x26860300(,%r8,8),%ymm1,%ymm0
 4.27          vmovnt %ymm0,0x606100(,%r8,8)
 0.05          add    $0x4,%r8
 0.02          cmp    $0x4c4b400,%r8
 2.34          jb     792
               vzerou
               lea    0x150(%rsp),%rdi
               lea    0x60(%rsp),%rsi
             → callq  gettimeofday@plt
               vxorpd %xmm0,%xmm0,%xmm0
               vxorpd %xmm1,%xmm1,%xmm1
               lea    0x158(%rsp),%rax
               vcvtsi (%rax),%xmm0,%xmm0
               vcvtsi -0x8(%rax),%xmm1,%xmm1
               vfmadd 0x252a(%rip),%xmm0,%xmm1        # 4038d0 <__dso_handle+0x608>
               lea    0xae0(%rsp,%r13,8),%rdx
               vsubsd (%rdx),%xmm1,%xmm2
               vmovsd %xmm2,(%rdx)
               inc    %r13
               cmp    $0x64,%r13
             ↑ jl     5c0
               mov    0x68(%rsp),%r12
               mov    $0x4035b0,%edi
               mov    0x70(%rsp),%r13
               mov    0x78(%rsp),%r14
               mov    0x80(%rsp),%r15
               mov    0x88(%rsp),%rbx
               rdtscp
```



```
-------------------------------------------------------------
1 second sleep, number of cycles = 2500292456
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:                 inf     0.000000     0.000000     0.000001
Scale:                inf     0.000000     0.000000     0.000000
Add:                  inf     0.000000     0.000000     0.000001
Triad:            21061.0     0.091392     0.091164     0.098817
-------------------------------------------------------------
cycles : 2.873238
```

Floating Point Representation

All real numbers can be approximated by floating point numbers represented by

$$(-1)^s\, d.dd\ldots d \times \beta^{e}$$

where:

- $d.dd\ldots d$ is called the mantissa and has p digits,

  $d.dd\ldots d = (d_0 + d_1\beta + d_2\beta^2 + d_3\beta^3 + \ldots d_{p-1}\beta^{p-1})$
- β is the base
- e is the exponent
- s is the sign

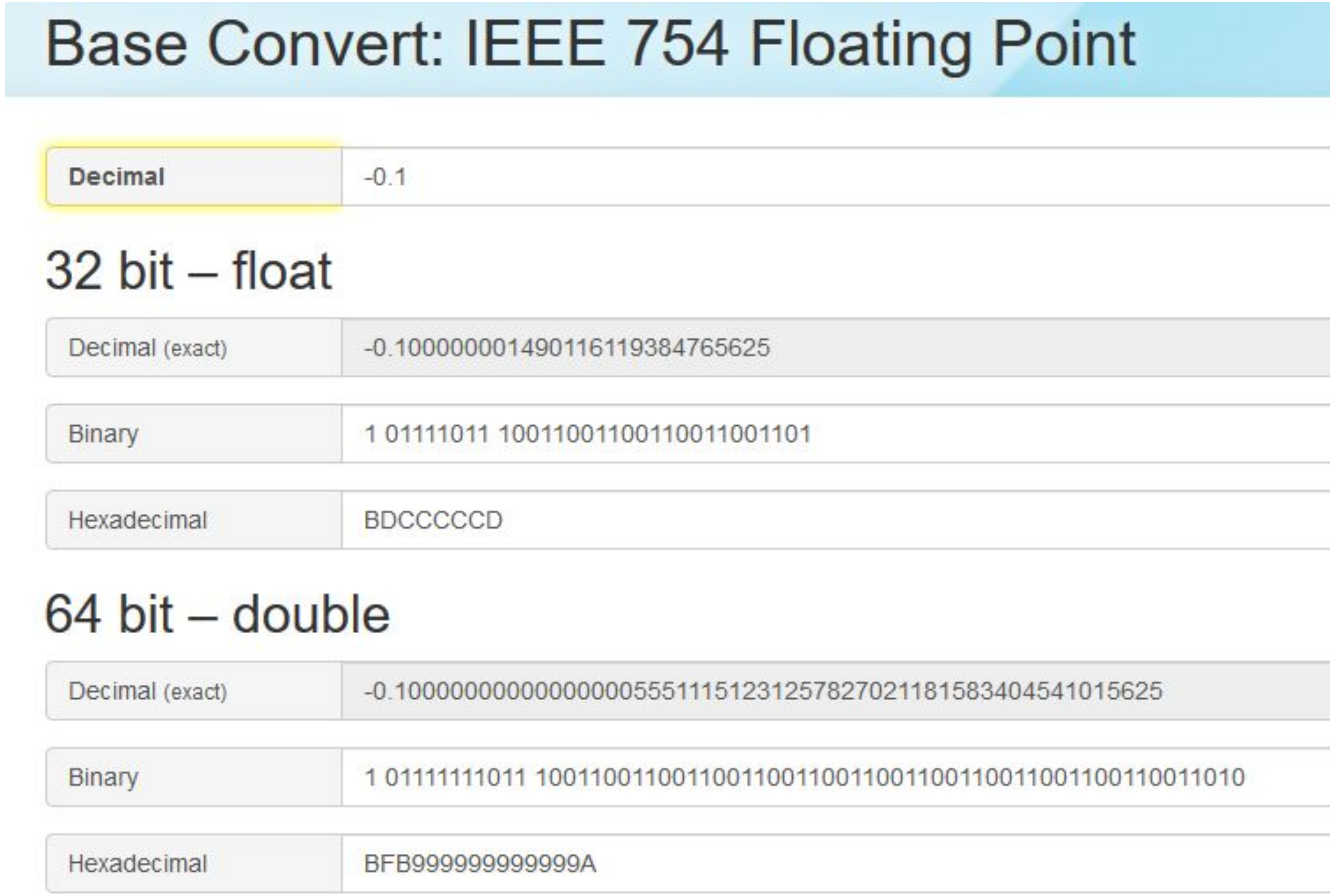
Any real number can be represented by a linear combination of

$0.5, 0.25, 0.125, 0.0625, \ldots, (1/2^{p-1})$

IEEE 754 defines two different floating point representation, single (32bits) and double precisions (64bits) in base two:

Precision	**Sign**	**Exponent**	**Mantissa**
Single precision	1	8	23 (+1)
Double precision	1	11	52 (+1)

Floating Point Representation



Base Convert: IEEE 754 Floating Point

Decimal	-0.1

32 bit – float

Decimal (exact)	-0.100000001490116119384765625
Binary	1 01111011 10011001100110011001101
Hexadecimal	BDCCCCCD

64 bit – double

Decimal (exact)	-0.1000000000000000055511151231257827021181583404541015625
Binary	1 01111111011 1001100110011001100110011001100110011001100110011010
Hexadecimal	BFB999999999999A

Floating Point Representation

Compute:

$$f(x, y) = (333.75 - x^2)y^6 + x^2(11x^2y^2 - 121y^4 - 2) + 5.5y^8 + x/(2y)$$

For

$$x = 77617 \text{ and } y = 33096 \text{ (exactly represented in FP32)}$$

FP32: $f(x, y) = 1.1726$
FP64: $f(x, y) = 1.17260394005318$
FP128: $f(x, y) = 1.172603940053178631$

Floating Point Representation

Compute:

$$f(x, y) = (333.75 - x^2)y^6 + x^2(11x^2 y^2 - 121y^4 - 2) + 5.5y^8 + x/(2y)$$

For

x = 77617 and y = 33096 (exactly represented in FP32)

FP32: $f(x, y) = 1.1726$
FP64: $f(x, y) = 1.17260394005318$
FP128: $f(x, y) = 1.17260394005318631$

The correct answer is:

$f(x, y) = -0.827396059946821368141165095479816 29$

computational precision != result accuracy

Floating Point Arithmetic

$\mathbb{R}$ is a field:

1- Closure:

a+b and a.b are in $\mathbb{R}$

2- Commutative laws:

a + b = b + a, a.b = b.a

3- Associative laws:

(a + b) + c = a + (b + c)

(a.b).c = a.(b.c)

4- distributive laws:

a.(b + c) = a.b + a.c

5- there are real numbers 0 and 1 such that

a + 0 = 0 + a = a and a.1 = 1.a = a

6- for any real number a, there exists -a such that

a + (-a) = (-a) + a

and if a != 0, there exists a^(-1) such that

a.a^(-1) = a^(-1).a = 1

7- cancellation: if a.b = a.c and a != 0 then b = c

It follows that:

8- (a + b) - b = a and a.(b/a) = b

Floating Point Arithmetic

FP is NOT a field:

1- Closure:

a+b and a.b are in FP

2- Commutative laws:

a + b = b + a, a.b = b.a

3- ~~Associative laws:~~

~~(a + b) + c = a + (b + c)~~

~~(a.b).c = a.(b.c)~~

4- ~~distributive laws:~~

~~a.(b + c) = a.b + a.c~~

5- there are real numbers 0 and 1 such that

a + 0 = 0 + a = a and a.1 = 1.a = a

6- for any real number a, there exists -a such that

a + (-a) = (-a) + a

and if a != 0, there exists a^(-1) such that

a.a^(-1) = a^(-1).a = 1

7- ~~cancellation: if a.b = a.c and a != 0 then b = c~~

It follows that:

8- (a + b) - b = a and ~~a.(b/a) = b~~

HPC brutal facts

HPC is about minimizing Time To Solution by **maximizing the throughput using parallelism**, not reducing latency

Without HPC techniques, **your code won’t run faster on supercomputers** than on your workstation

CPUs are very good at doing **fused multiply-adds** but that’s about it

HPC is about knowing **how your software AND the hardware best work together** to get maximum performance

HPC is about **hacking** your way around the language standard

You’ll need to have a **look at the assembly**

Bibliography

Computer Architecture, A Quantitative Approach
Patterson, 2011

Introduction to High Performance Computing for Scientists and Engineers
Georg Hager, Gerhard Wellin, CRC Press

Roofline: An Insightful Visual Performance Model for Multicore Architectures
Samuel Williams, Andrew Waterman, and David Patterson

Floating-Point Computation
Pat H. Sterbenz

Exercices

https://c4science.ch/source/phpc-2019/

Directory: Serie03