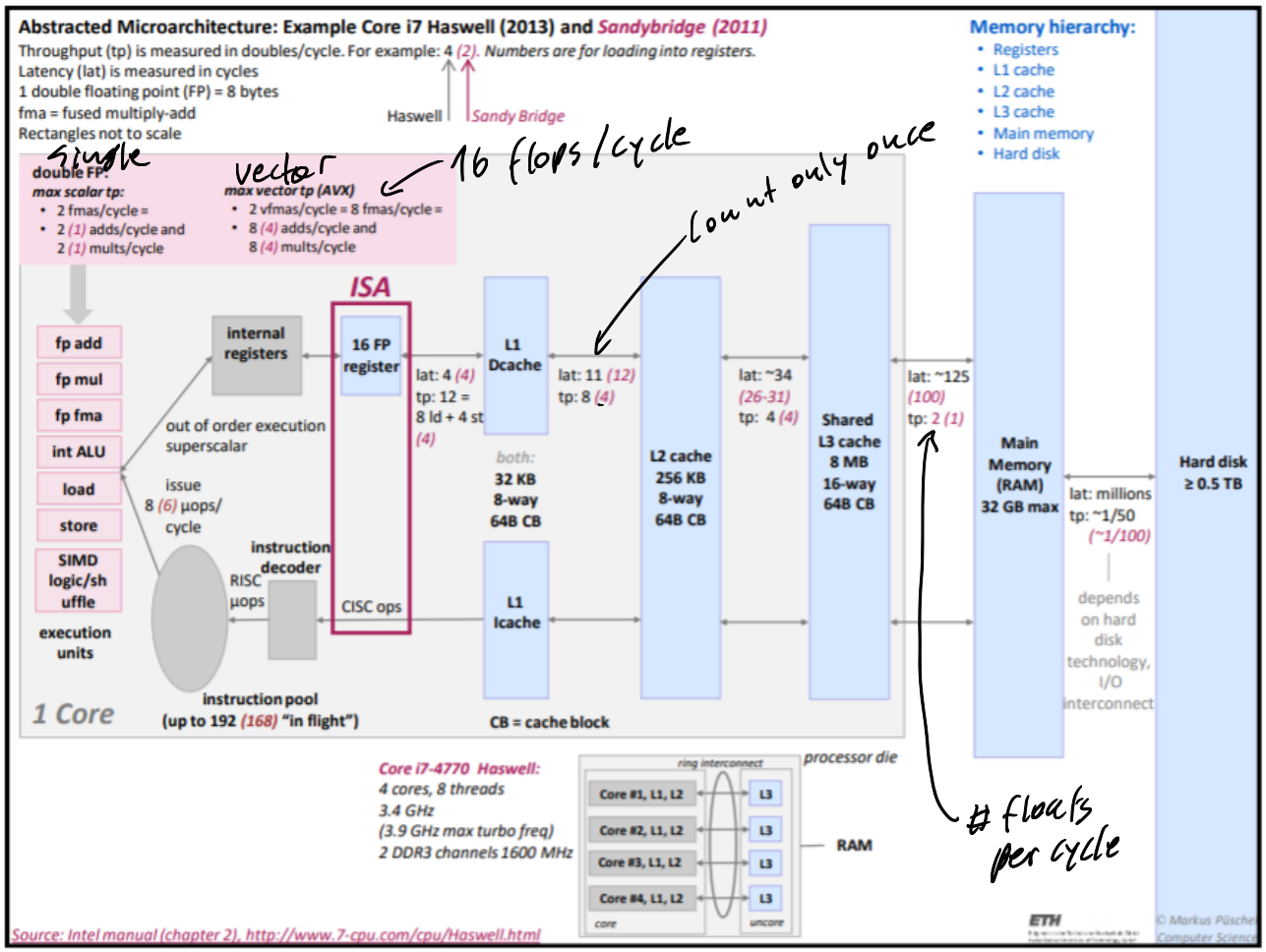
Zusammenfassung Advanced Systems Lab

- $\Theta(f(n)) = \Omega(f(n)) \cap O(f(n))$
  (Average: Θ, Best: Ω, Worst: O)
- $2n + O(\log(n))$ is canonical
- $O(2n) + \log(n)$ isn't canonical because the rest must be smaller $\Rightarrow O(n)$
  (rest: $O(2n)$, precise: $\log(n)$)
- Matrix Matrix Multiplication: · Runtime $O(n^3)$
  · Space $O(n^2)$
- $\Theta(f(n))$ only describes eventual trend of runtime (doesn't tell us tightness, or how start looks)
- Cost measure: $C(n) = N_{add} \cdot C_{add} + N_{mult} \cdot C_{mult}$
- CPUs can do arithmetic (flops) and indexing (ints) at the same time because there are a Arithmetic LU and Integer LU
- Performance: · flops/cycle $\hat{=}$ #flops/#cycles
  · upper bound "peak performance" given by CPU
- Asymptotic Runtime vs. Cost
- Architecture/instruction set architecture: - Assembly
  - Registers
  - e.g. x86
- Microarchitecture: - Implementation of Architecture
  - Cache structures, frequency, ...
  - Not relevant for writing code
- SIMD: · Do vector instructions

- Intel's Tick Tock Model
  - Tick: Shrink of process technology
  - Tock: New Microarchitecture (= new names)
  - Since 2016: process - architecture - optimization
- Cisc: - Complex instruction set computer
  - Intel's architecture
  - Is converted into Risc internally
- FMA: - 1 FMA ≙ 2 flops ≙ 1 fadd + 1 fmult
  - Recently implemented, because before Intel supported only 2 inputs
  - FMA: X * y + Z



```
/* x, y are vectors of doubles of length n, alpha is a double */
for (i = 0; i < n; i++)
  x[i] = x[i] + alpha*y[i];
```

FMA

maximal achievable percentage of (vector) peak performance

Number flops?	2n	
Runtime bound no vector ops:	n/2	cycles needed for n length
Runtime bound vector ops:	n/8	
Runtime bound data in L1:	n/4	50
Runtime bound data in L2:	n/4	50
Runtime bound data in L3:	n/2	25
Runtime bound data in main memory:	n	12.5

↳ Bottleneck is memory

```
/* matrix multiplication; A, B, C are n x n matrices of doubles */
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      C[i*n+j] += A[i*n+k]*B[k*n+j];
```

Number flops?	$2n^3$
Runtime bound no vector ops:	$n^3/2$
Runtime bound vector ops:	$n^3/8$
Runtime bound data in L1:	$(3/8)\ n^2$
...	
Runtime bound data in main memory:	$(3/2)\ n^2$

↳ Compute bound: $n^3/2 > \left(\frac{3}{2}\right) n^2$

- Operational Intensity: $I(n) = \frac{W(n)}{Q(n)}$ — # flops with n / # bytes transferred cache ↔ memory

↳ If we look at upper bound, we only take matrix: n^2, vector: n once

- 1 double floating point = 8 bytes
- Low $I(n)$ ≙ memory bottleneck

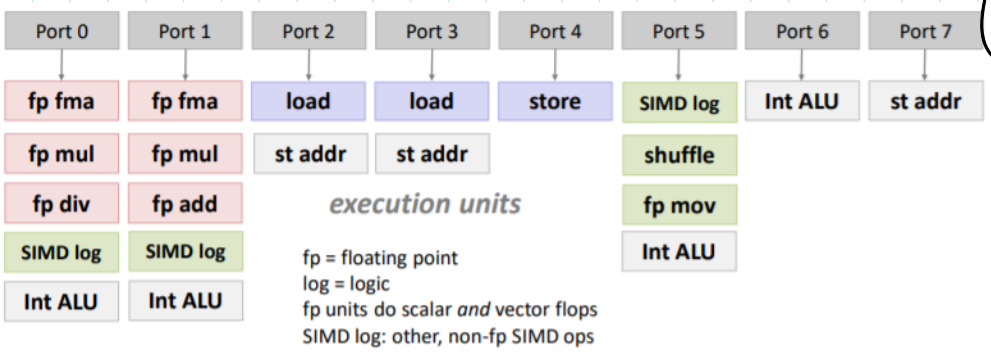
```
/* x, y are vectors of doubles of length n, alpha is a double */
for (i = 0; i < n; i++)
  x[i] = x[i] + alpha*y[i];
```

C is in register

- **Operational intensity:**
  - Flops: W(n) = 2n
  - Memory transfers (doubles): ≥ 2n (just from the reads)
  - Reads (bytes): Q(n) ≥ 16n
  - Operational intensity: I(n) = W(n)/Q(n) ≤ 1/8

```
/* matrix multiplication; A, B, C are n x n matrices of doubles */
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      C[i*n+j] += A[i*n+k]*B[k*n+j];
```

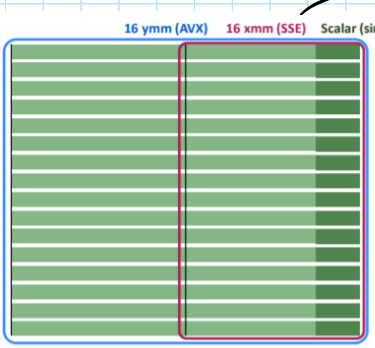
- **Operational intensity:**
  - Flops: W(n) = $2n^3$
  - Memory transfers (doubles): ≥ $3n^2$ (just from the reads)
  - Reads (bytes): Q(n) ≥ $24n^2$
  - Operational intensity: I(n) = W(n)/Q(n) ≤ n/12



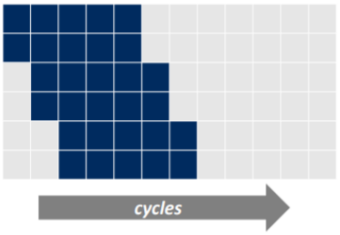
↳ When data is in RAM, written vectors are first read

↳ · Every port can issue one instruction/cycle
· Gap = 1/Throughput (cycles per issue)
· e.g. FMA throughput (ops/cycle): 2
· Same throughput/latency for vector instructions, but then, each instruction has multiple floats

→ 4 doubles
→ 2 doubles



Each register:
256 bits = 4 doubles = 8 singles



Throughput tp = 2/cycle

Gap = 1/tp = 1/2 cycles/issue

How many cycles are at least required (no vector ops)?

- A function with n adds and n mults in the C code — *n/2* → converted to FMA
- A function with n add and n mult instructions in the assembly code — *n*
- A function with n adds in the C code — *n/2* → converted to FMA
- A function with n add instructions in the assembly code — *n*
- A function with n adds and n/2 mults in the C code — *n/2*

- Peak performance of computer:
  - 4 cores
  - 2-way SSE
  - 1 add + 1 mult /cycle
  - 3 GHz
  - ↳ 48 Gflop/s

- Instruction level parallelism (ILP)
  - Can taken advantage of by superscalar computers, because they can execute multiple instructions in one cycle
  - Sequential dependence ⇒ no ILP!
- Loop unrolling ↳ The deeper the pipeline (latency), the more ILP you need
  - Use k accumulators
  - Need to unroll by L, k divides L
  - Large overhead for short lengths
  - Best k = ceil (latency * throughput)
    - ↳ In every cycle you can start two mults, and they need 5 cycles to finish, so you can have 10 floats in the pipeline

- Compiler limitations
  - Associativity law doesn't hold for floats
  - Move code out of loop
  - Replace costly computations with simpler ones
  - Reuse portions of expressions
  - Removing procedure calls
  - Function inlining, if source code is compiled together
  - Compiler usually treats procedure call as black box
  - Memory accessed every iteration => Scalar replacement
  - Memory aliasing: Memory can overlap, this leads to other results, if we use local variable, which doesn't have aliasing
- Single vs double precision
  - Single: 4 bytes ≙ float
  - Double: 8 bytes ≙ double
- Intrinsics
  - Assembly coded in C functions
  - You need to code explicit loads and stores
  - Use aligned loads and stores as much as possible
    - ↳ malloc allocation guarantees aligned allocation
  - Never divide, multiply by inverse instead

```
 0 1 2 3 4 5 6 7
|a|a|b|b|b|b|c|d|  bytes
|       |       |  words
```

The problem is that on some CPU architectures, the instruction to load a 4-byte integer from memory only works on word boundaries. So your program would have to fetch each half of `b` with separate instructions.

But if the memory was laid out as:

```
 0 1 2 3 4 5 6 7 8 9 A B
|a|a| | |b|b|b|b|c|d| | |
|       |       |       |
```

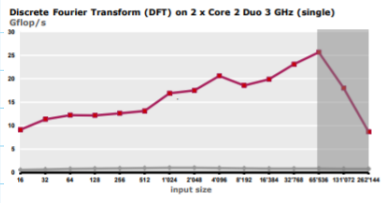
Then access to `b` becomes straightforward. (The disadvantage is that more memory is required, because of the padding bytes.)

Different data types have different alignment requirements. It's common for `char` to be 1-byte aligned, `short` to be 2-byte aligned, and 4-byte types (`int`, `float`, and pointers on 32-bit systems) to be 4-byte aligned.

`malloc` is required by the C standard to return a pointer that's properly aligned for any data type.

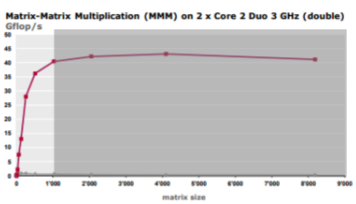
- Temporal locality: Recently used data will likely be used again
  - Naturally supported by cache
- Spatial Locality: Data near by will likely be accessed
  - Supported by transferring data in blocks

FFT: *I(n) = O(log(n))*



Up to 40-50% peak
Performance drop outside last level cache (LLC)
Most time spent transferring data

MMM: *I(n) = O(n)*

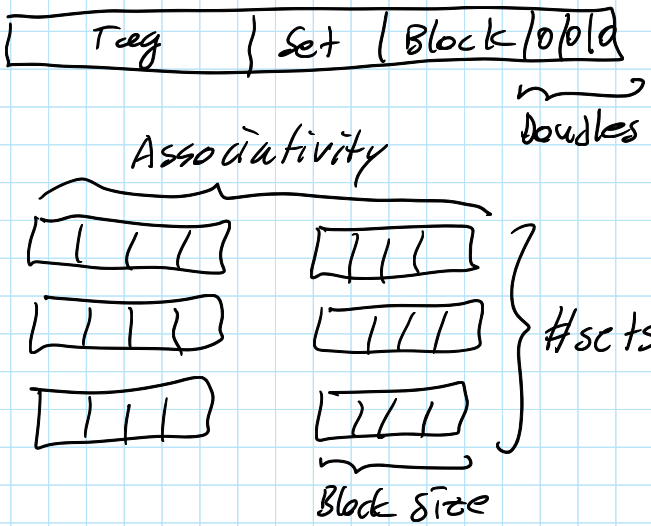


Up to 80-90% peak
Performance can be maintained outside LLC
Cache miss time compensated/hidden by computation

- Asymptotic bounds on operational intensity
  - Vector sum: $y = x + y$ $\Rightarrow O(n/n) = O(1)$ => Not good
  - Matrix-Vector: $y = Ax$ $\Rightarrow O(n^2/n^2) = O(1)$
  - FFT $\Rightarrow O(\log(n))$
  - MMM: $C = AB + C$ $\Rightarrow O(n^3/3n^2) = O(n)$

- Cache misses
  - Compulsatory
  - Capacity
  - Conflict (multiple data blocks are mapped to same location)

- Cache structure



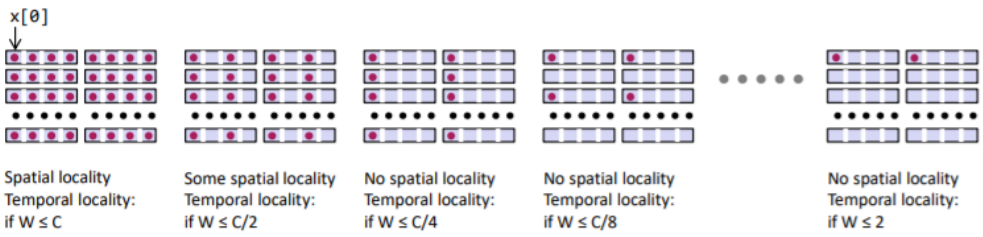
  - Tag: rest of address that is stored in cache for reconstruction of address
  - Direct mapped: - E = 1
    - Every address has unique location in cache
  - Fully associative: - # sets = 1
    - No conflict misses
    - LRU policy for replacement

- If data isn't in cache, it has first be read, after it can be written to

The Killer: Two-Power Strided Working Sets

```
% t = 1,2,4,8,… a 2-power
% size W of working set: W = n/t
for (i = 0; i < n; i += t)
  access(x[i])
for (i = 0; i < n; i += t)
  access(x[i])
```

Cache: E = 2, B = 4 doubles



Working with a two-power-strided working set is like having a smaller cache

↳ Power of to is regular

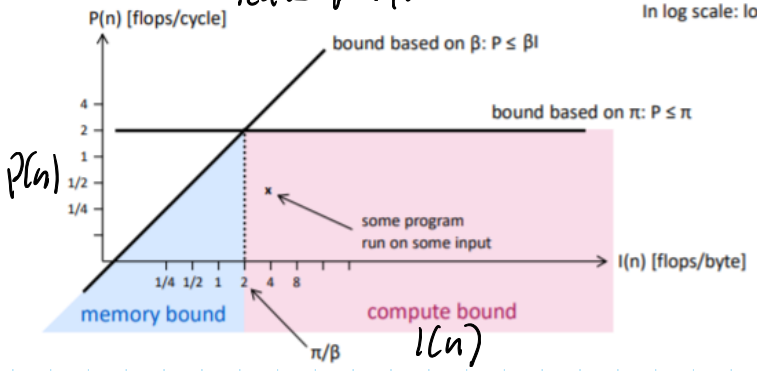
- Roofline Model
  - Operational intensity: $I(n) = \frac{W(n)}{Q(n)}$ $\frac{\#\text{flops}}{\#\text{bytes}}$
  - Performance: $P(n) = \frac{W(n)}{T(n)}$ $\frac{\#\text{flops}}{\#\text{cycles}}$

← Bandwidth bytes/cycle

Example: one core, π = 2, β = 1, no SIMD

↑ Peak performance

Bound based on β:
β ≥ Q/T = (W/T)/(W/Q) = P/I
In log scale: $\log_2(P) \le \log_2(\beta) + \log_2(I)$



- What happens, if we introduce 4-way SIMD?
  - ↑ changes
  - β stays the same
  - More programs become memory bound

- Linear algebra software
  - Blas: Kernel function implemented for each computer (generated by Atlas)
  - Lapack: static higher level function,

Dependencies

- **Read-after-write (RAW) or true dependency**

  W $r_1 = r_3 + r_4$
  R $r_2 = 2r_1$

  nothing can be done
  no ILP

- **Write after read (WAR) or antidependency**

  R $r_1 = r_2 + r_3$
  W $r_2 = r_4 + r_5$

  dependency only by name → rename

  $r_1 = r_2 + r_3$
  $r = r_4 + r_5$

  now ILP

- **Write after write (WAW) or output dependency**

  W $r_1 = r_2 + r_3$
  ...
  W $r_1 = r_4 + r_5$

  dependency only by name → rename

  $r_1 = r_2 + r_3$
  ...
  $r = r_4 + r_5$

  now ILP

} Each variable is assigned exactly once

- Sparse linear algebra

A as matrix

b	c		c
	a		
		b	b
		c	

A in CSR:

values	b	c	c	a	b	b	c	length K
col_idx	0	1	3	1	2	3	2	length K
row_start	0	3	4	6	7			length n+1

- Upper bound on operational intensity
  ↳ $I(n) = 2K / 8(K + 3n)$

} $y = Ax + b$