UNIVERSITÀ
DEGLI STUDI
FIRENZE

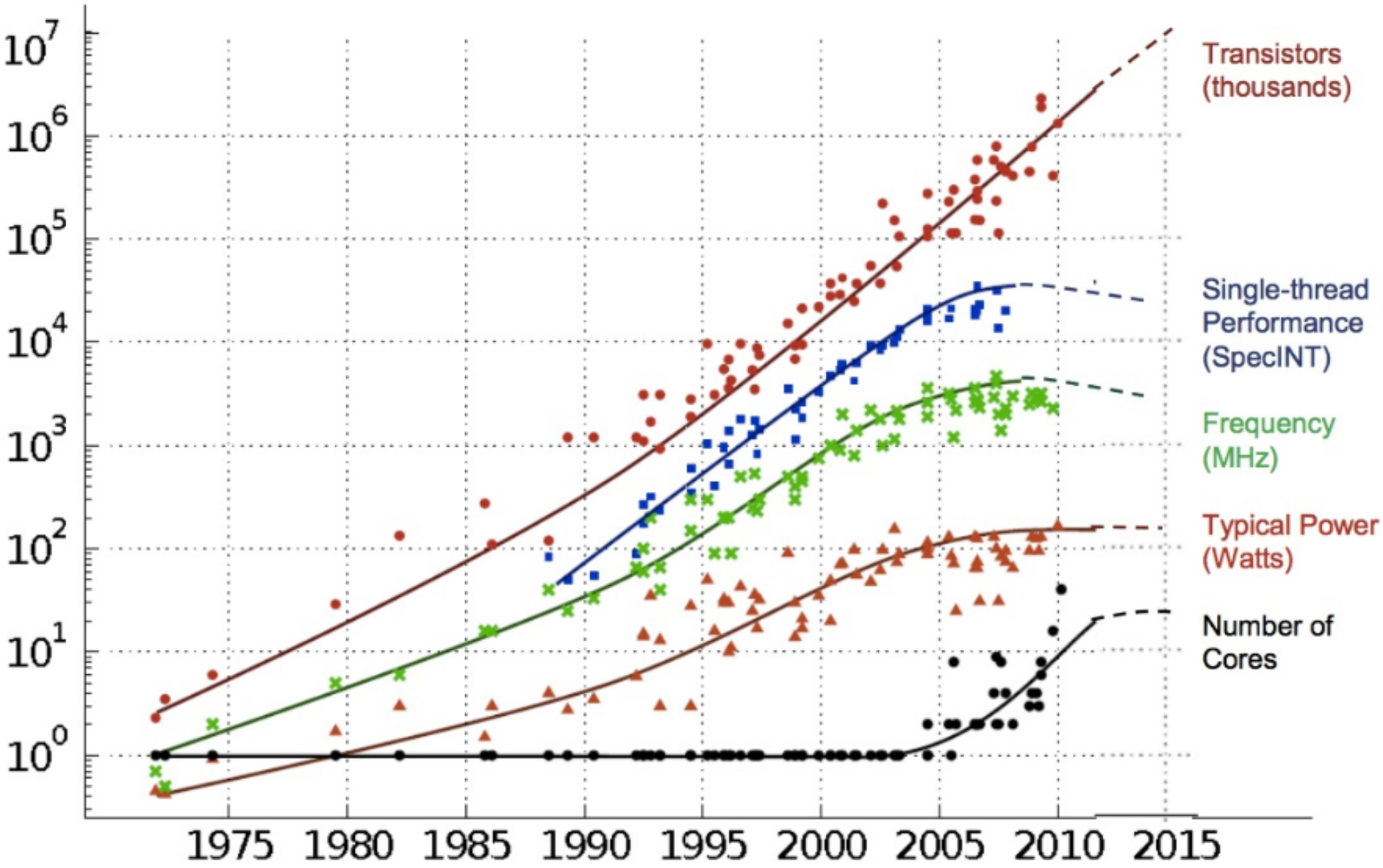
Parallel Programming

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Modern CPUs

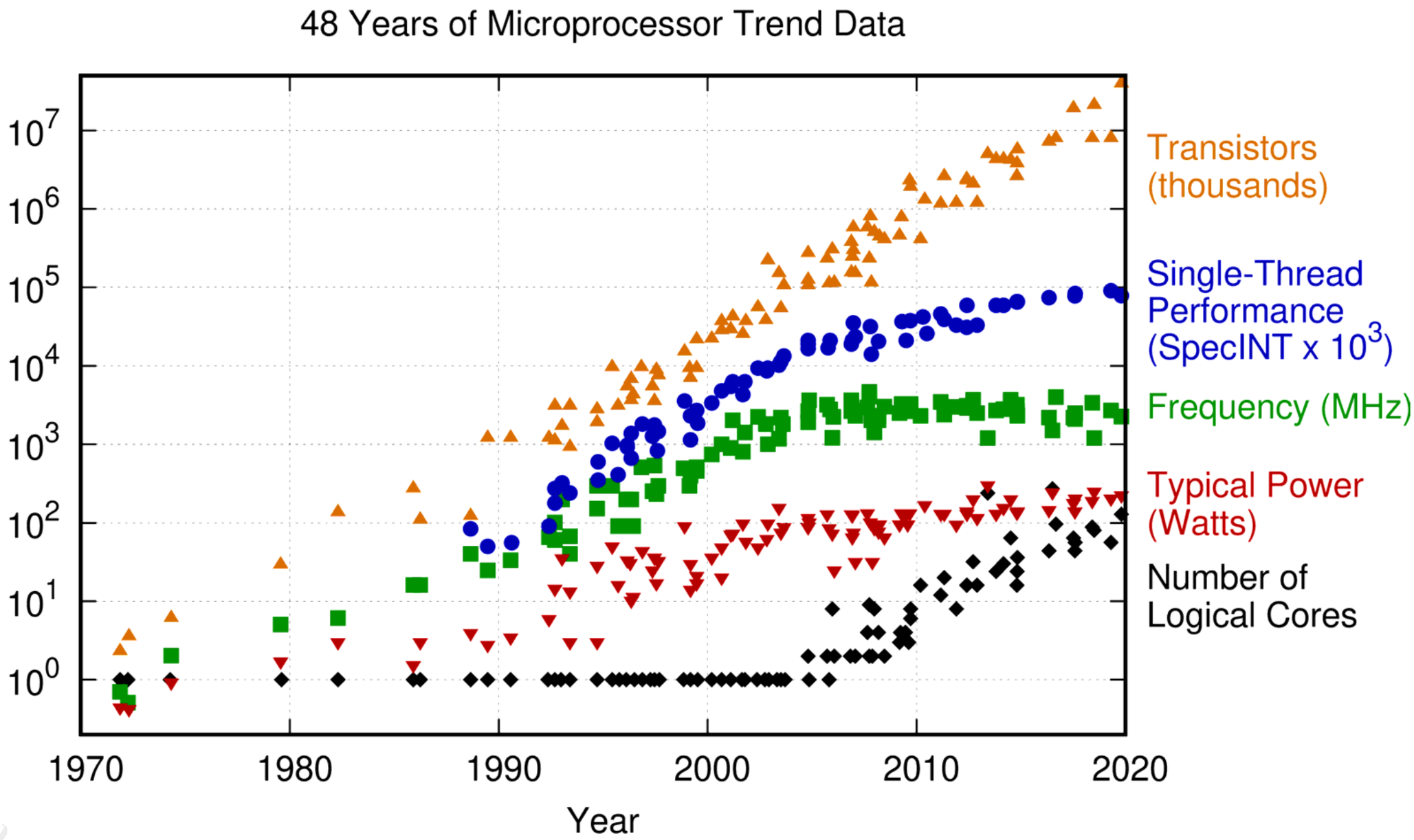
Historical trends in CPU performance



From ‘Data processing in exascale class computer systems’, C. Moore

https://www.lanl.gov/conferences/salishan/salishan2011/3moore.pdf

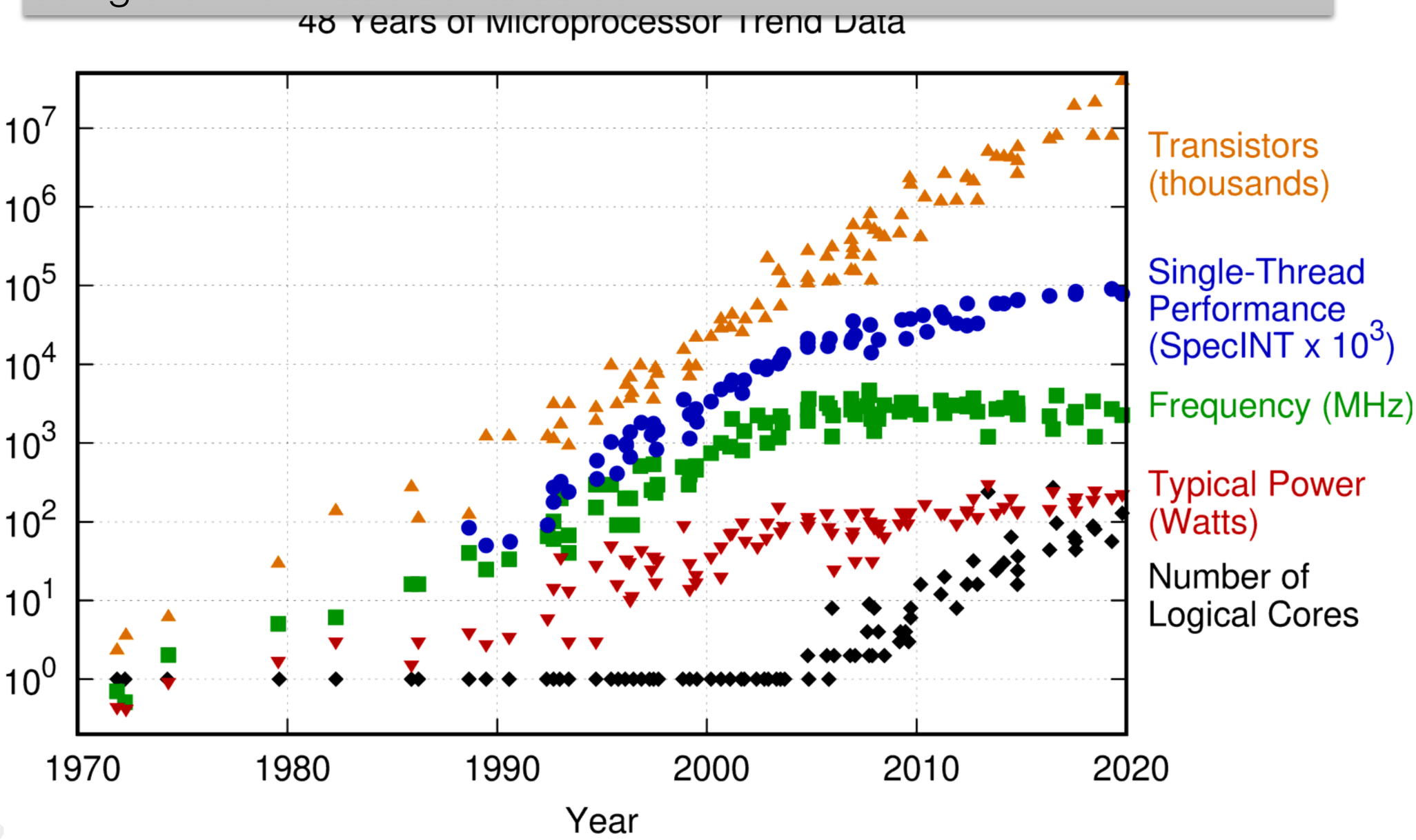
Historical trends in CPU performance: updates



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

https://github.com/karlrupp/microprocessor-trend-data

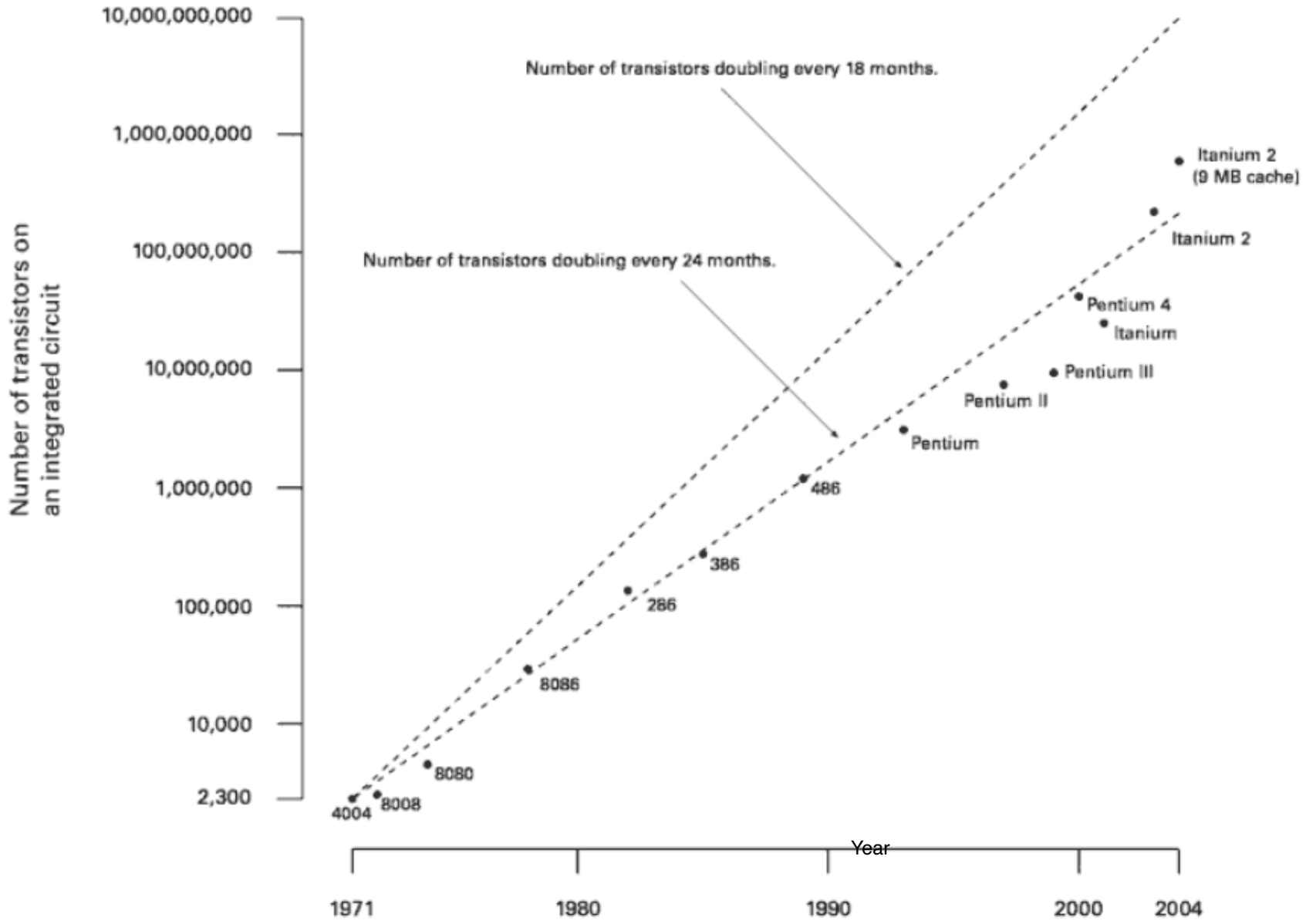
Single-thread performance has kept increasing slightly, reflecting that this is still an important quantity. These increases were achieved with clever power management and dynamic clock frequency adjustments ("turbo").
The number of cores is now increasing with a power law: transistor counts are increasing in accordance to Moore's Law, so one way of using them is in additional cores.



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

https://github.com/karlrupp/microprocessor-trend-data

Moore’s law

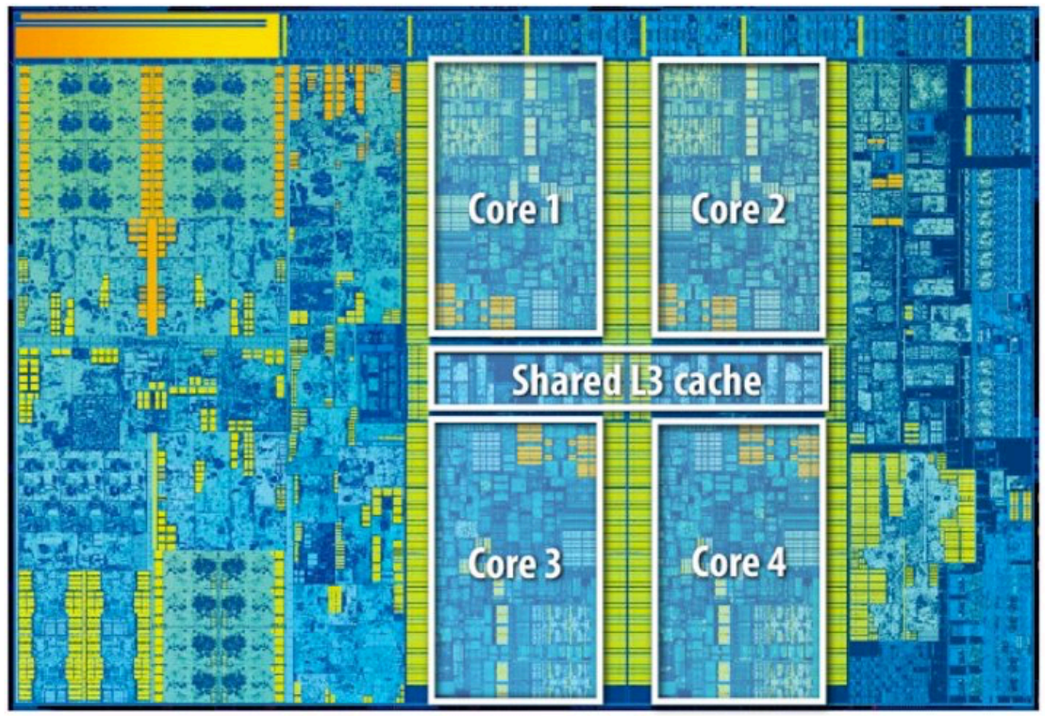


- Still works for # of transistors, not for clock speed.

Multi core era

- Idea #1: use increasing transistor count to add more cores to processors
  - Instead of increasing sophisticated processor logic to accelerate single instruction stream (e.g., out-of-order and speculative ops). Typical of GPU processors.
- Idea #2: add more ALUs, to amortize cost/complexity of managing an instruction stream
  - SIMD: same instruction broadcast to all ALUs executed in parallel on all ALUs

Example: Skylake

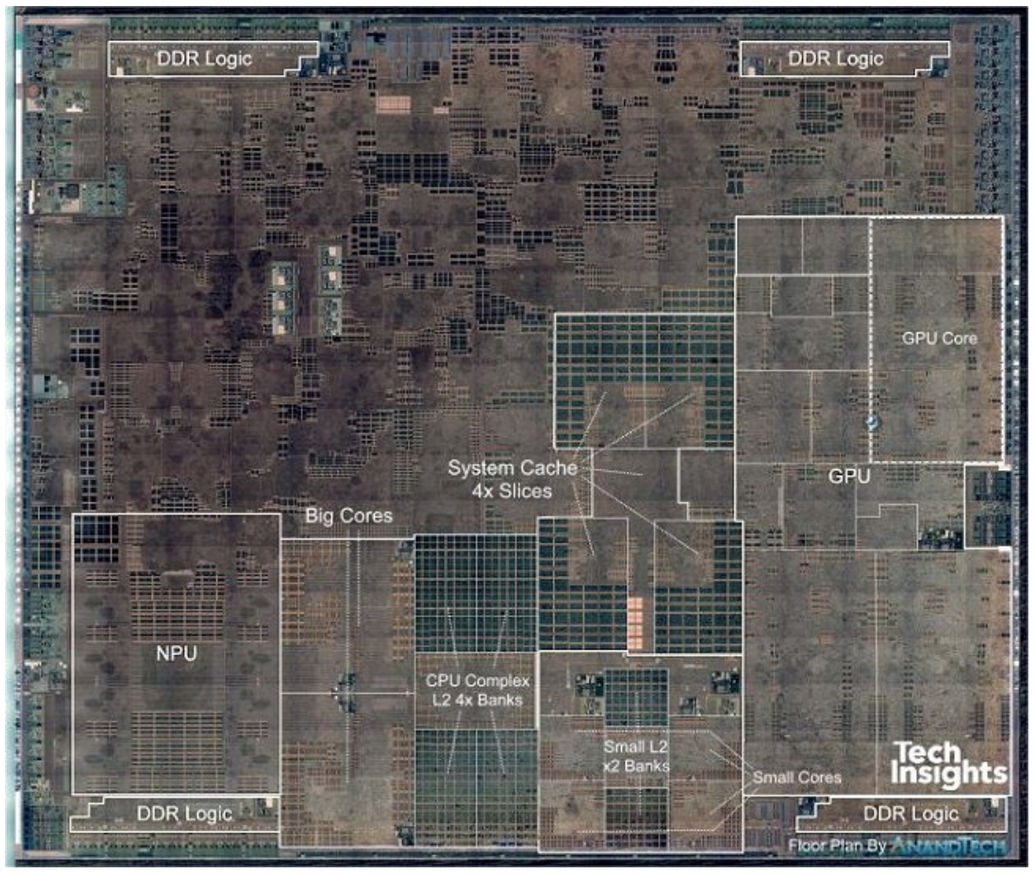


- Core i7 quad-core CPU. 14 nm 3D transistors.
- Core i9 goes up to 18 cores.

Example: Ice Lake

- Increase in L1 data cache size (32KB i/48KB d), larger L2 cache. 10nm.
  - Following processor architectures increase cache sizes.
  - 18% increase in IPC (Instructions Per Clock) over Skylake
  - Six new AVX-512 instructions
  - Intel Deep learning Boost, to accelerate… deep learning. AVX-512 instructions to handle reduced precision int and floats.

Example: Apple A12



- 8 core NPU, 2 big core (w/ 4 L2 banks), 4 small core (w/ 4 L2 banks), 4 GPU cores. 7 nm.

ARM big.LITTLE

- Architecture that combines slower processor cores (LITTLE) with relatively more powerful and power-hungry ones (big)

- The goal of this architecture is to create a multi-core processor that can:

  - adjust better to dynamic computing needs;

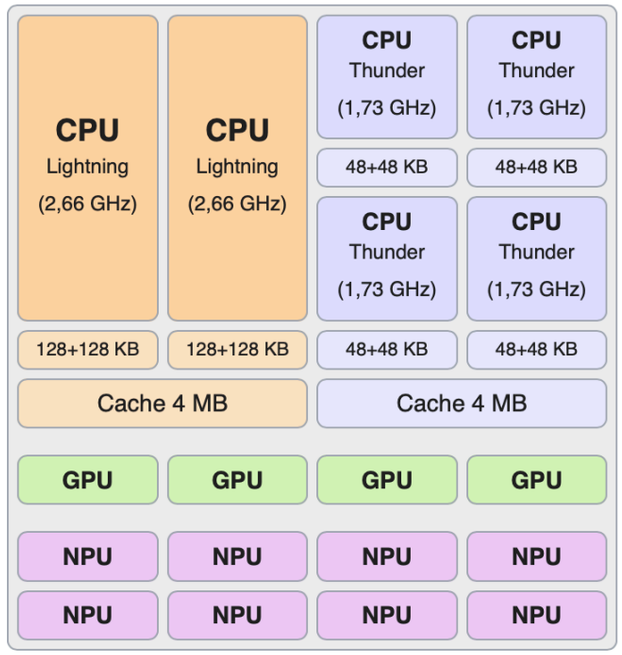
  - use less power than clock dynamic frequency scaling.

- Intel is following this approach in the most recent processors

Example: Apple A13 and A14

- A13 is still a 7nm SOC.
  Larger (1.6B more transistors),
  faster and lower consumption

- A14 is a 5nm SOC.
  3.3B more transistors than A13 !
  Again: 2 fast cores, 4 efficient cores,
  4-core GPU and 16-core NPU



Example: Apple A15

- A15 is a 5nm SOC (like A14).
- 15B transistors, again: 2 fast cores and
  4 efficient cores.
  5-core GPU (1 more than A14) and faster (+40%)
  16-core NPU.
  Cache greatly increased to 32MB

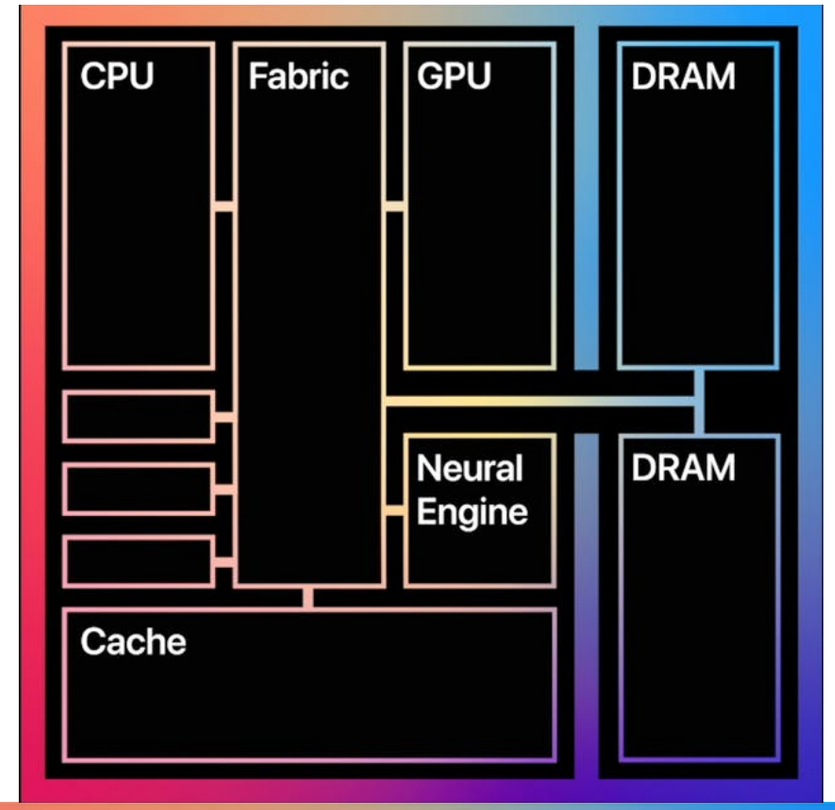
Example: Apple M1

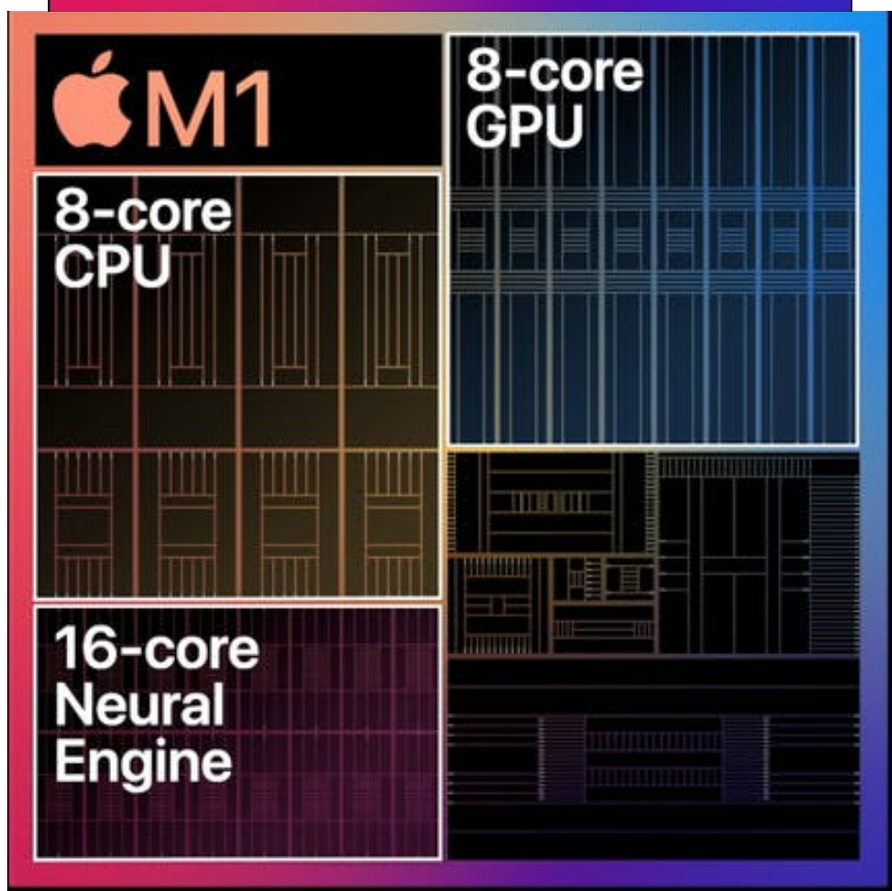
- M1 is a 5nm SOC (like A14).

- 4 fast cores and
  4 efficient cores.
  12MB cache for performance cores and 4MB for efficient cores.
  8-core GPU and 16-core NPU

- 128-bit memory interface: 68GB/s memory bandwidth.

- System Level Cache: it serves the whole chip functions





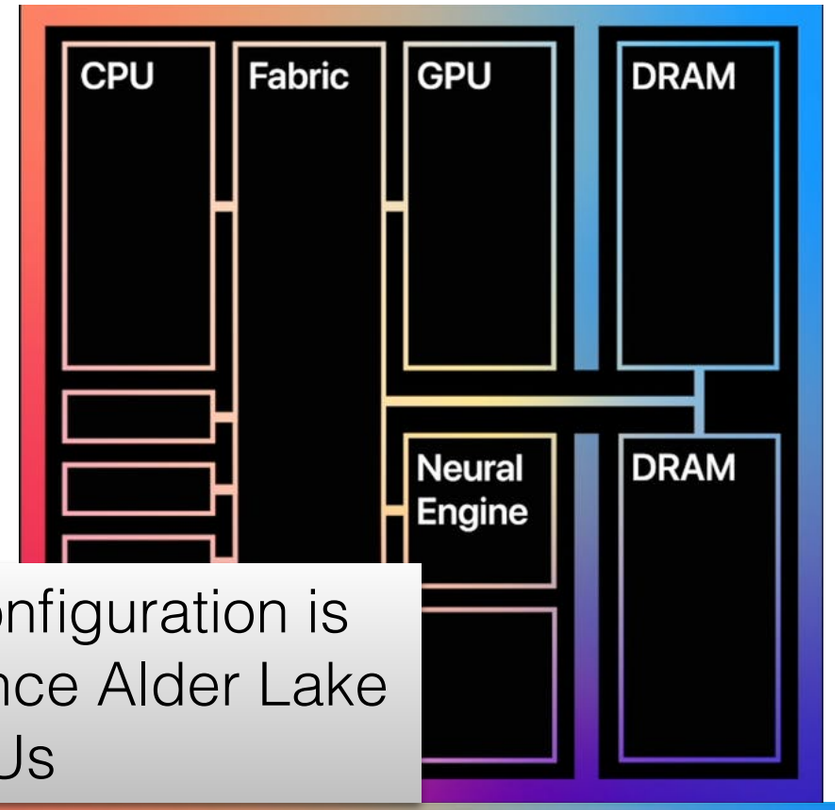
Example: Apple M1

- M1 is a 5nm SOC (like A14).

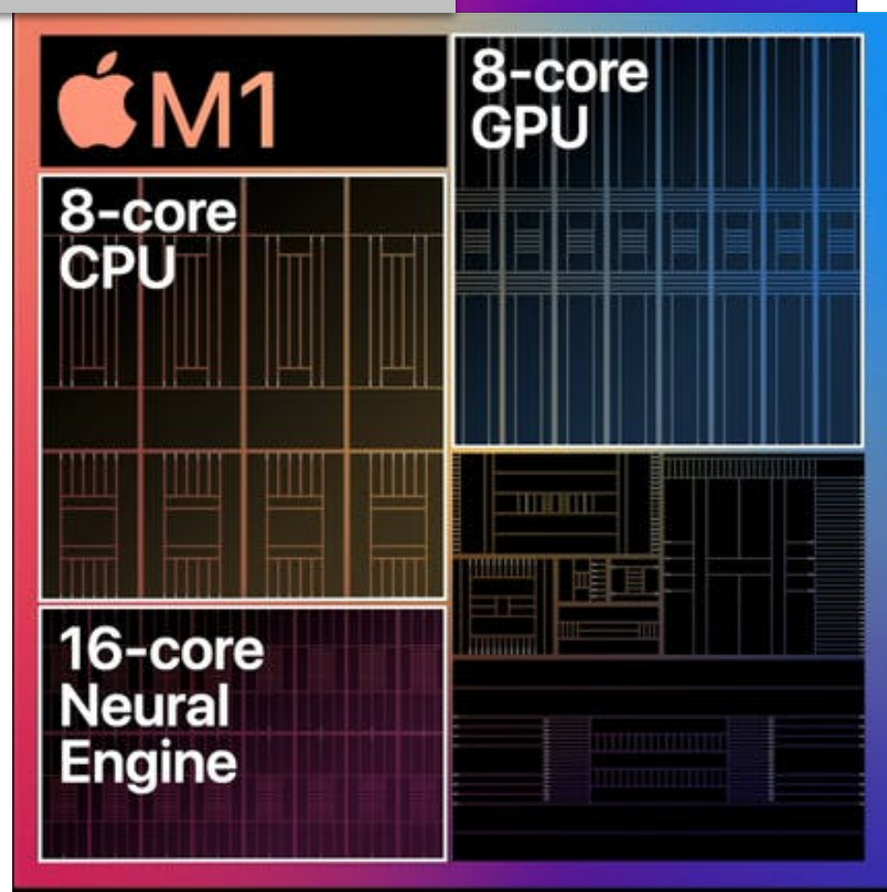
- 4 fast cores and
  4 efficient cores.
  12MB cache for performance
  and 4MB for efficient cores.
  8-core GPU and 16-core NPU

- 128-bit memory interface: 68GB/s memory bandwidth.

- System Level Cache: it serves the whole chip functions

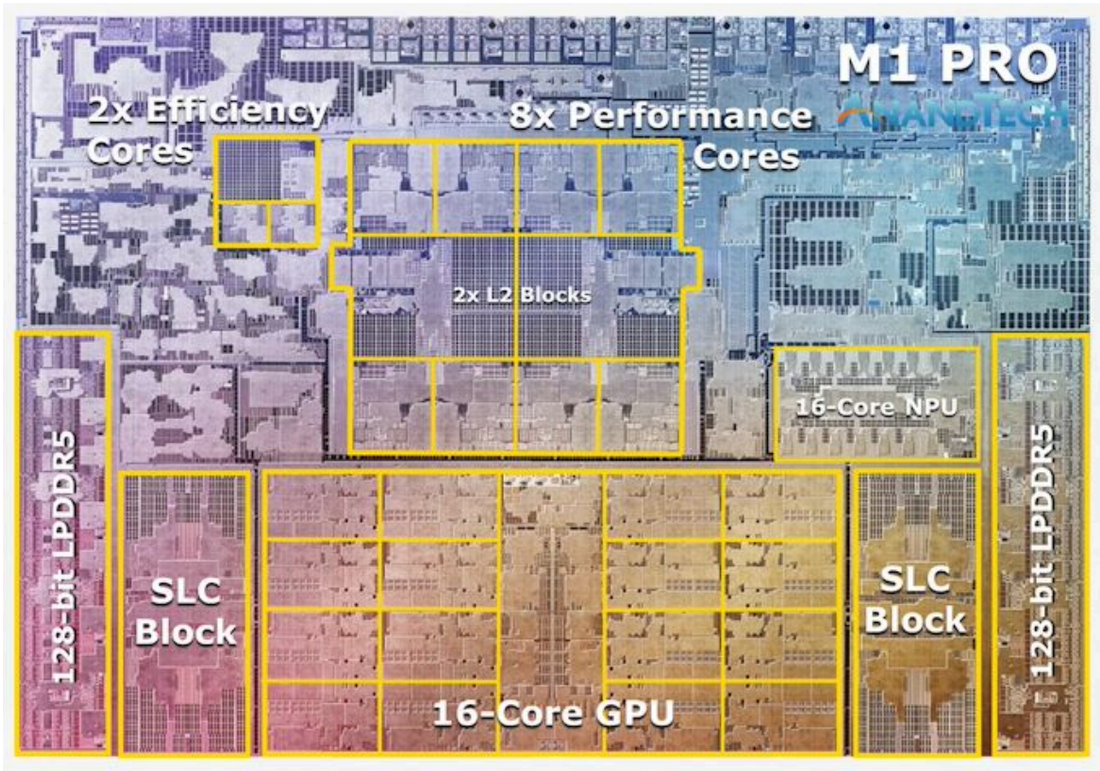


This type of configuration is used by Intel since Alder Lake CPUs



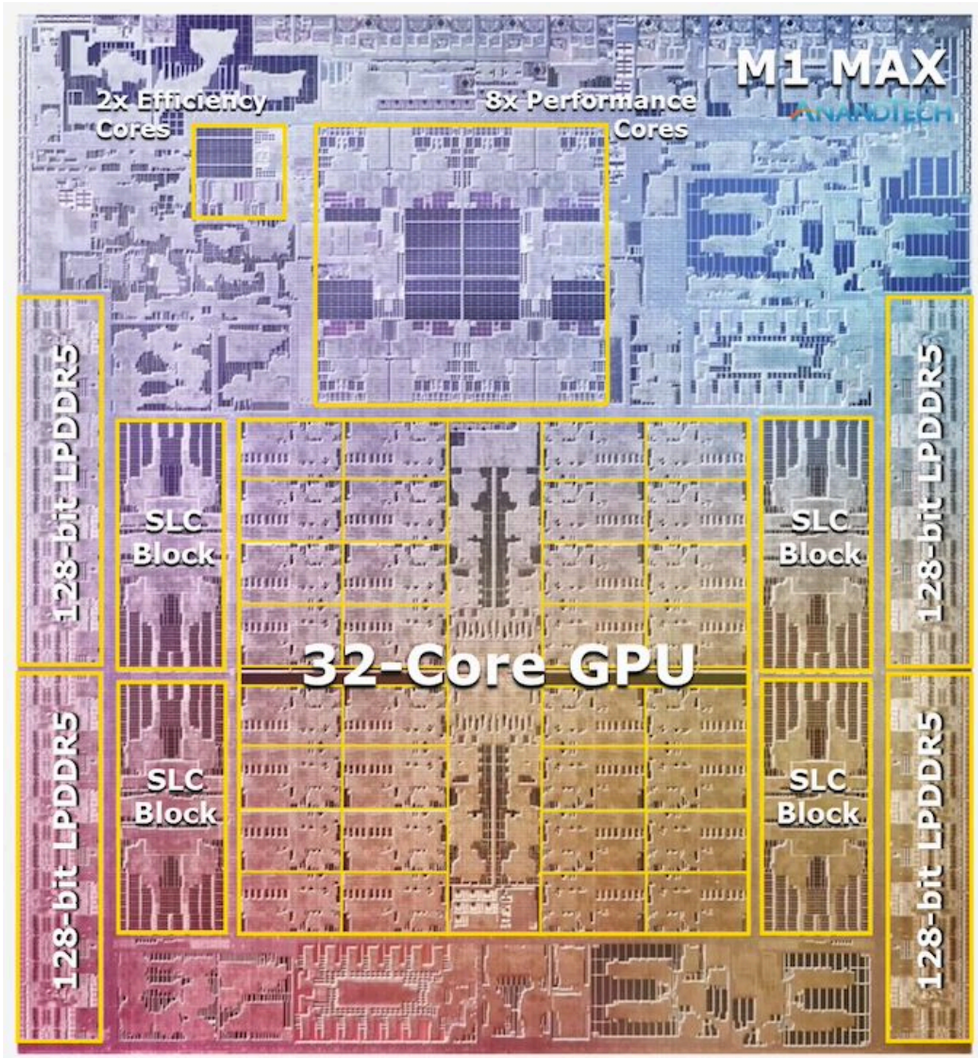
Example: Apple M1 Pro

- 8 Firestorm performance cores + 2 Icestorm efficiency cores: each 4-core efficient cluster has its own 12MB L2 cache, efficient cores with 4MB L2 cache
  - Different clocking depending on core usage
  - 24MB System Level Cache
  - 256-bit LPDDR5 memory @ 6400MT/s (i.e. 204 GB/s) bandwidth
  - Media Engine (H.264 and video encode/decode)
- 16-core GPU
- 37 billion transistors (5nm)



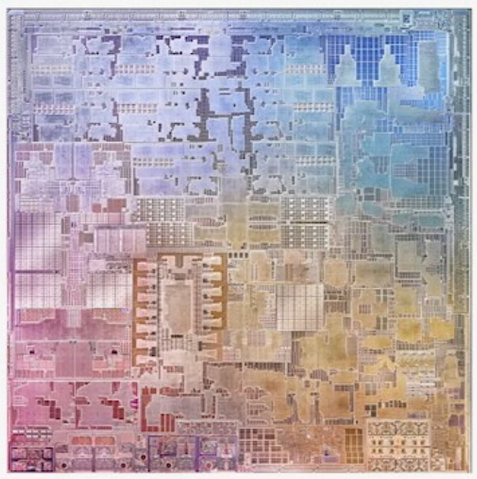
Example: Apple M1 Max

- 57 billion transistors (5nm) !
- More GPU cores @ 1296MHz
- Media engine for encode/ decode functions
- 512-bit LPDDR5 memory: 408GB/s bandwidth
- 48MB System Level Cache



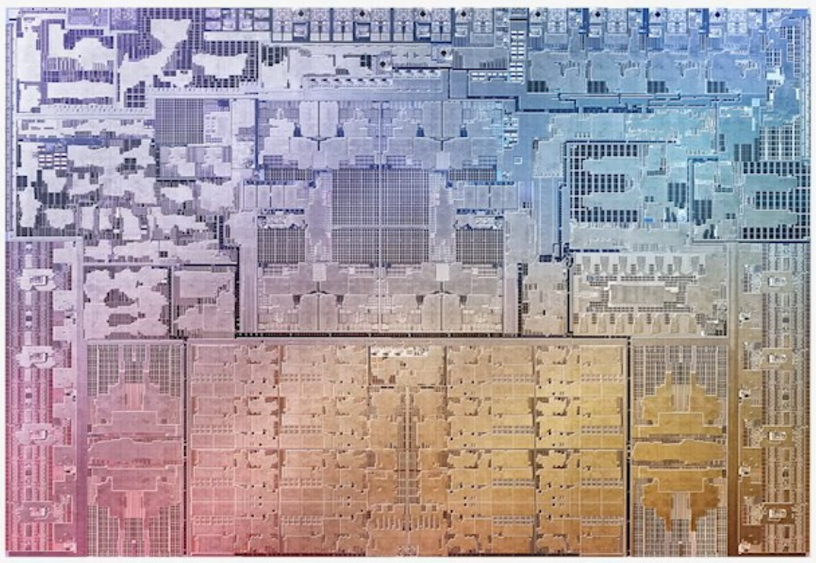
Example: M1 family

10.99 x 10.96mm
~120.4mm²



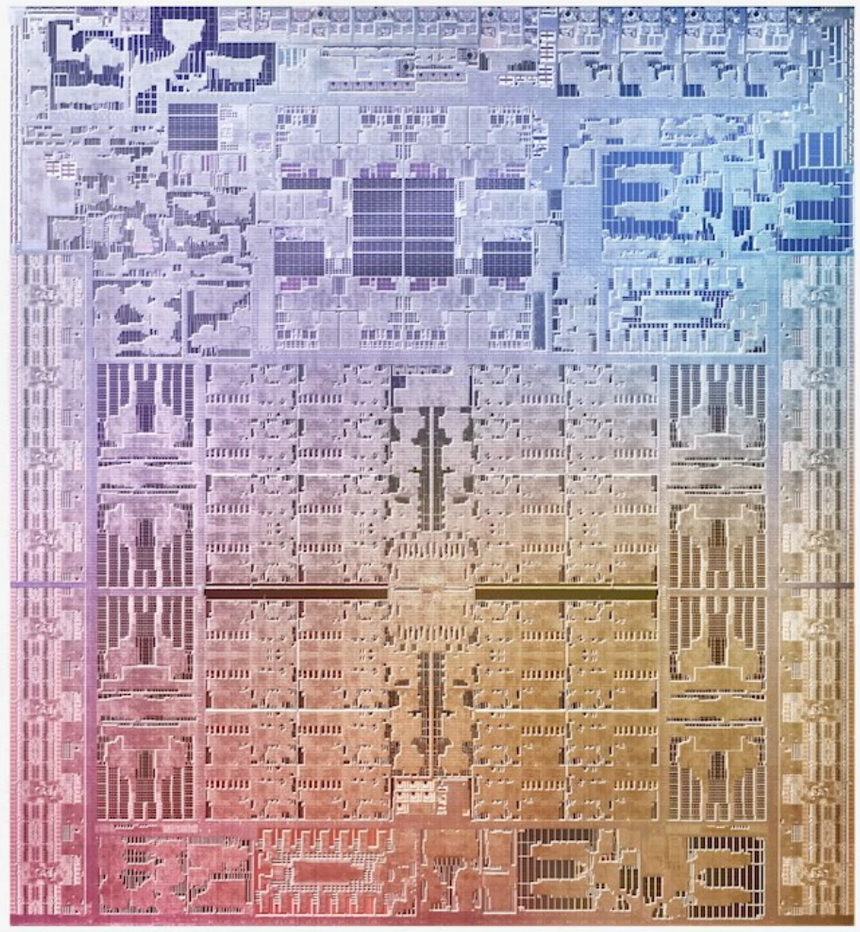
M1

18.95 x 12.98mm
~245.92mm²



M1 Pro

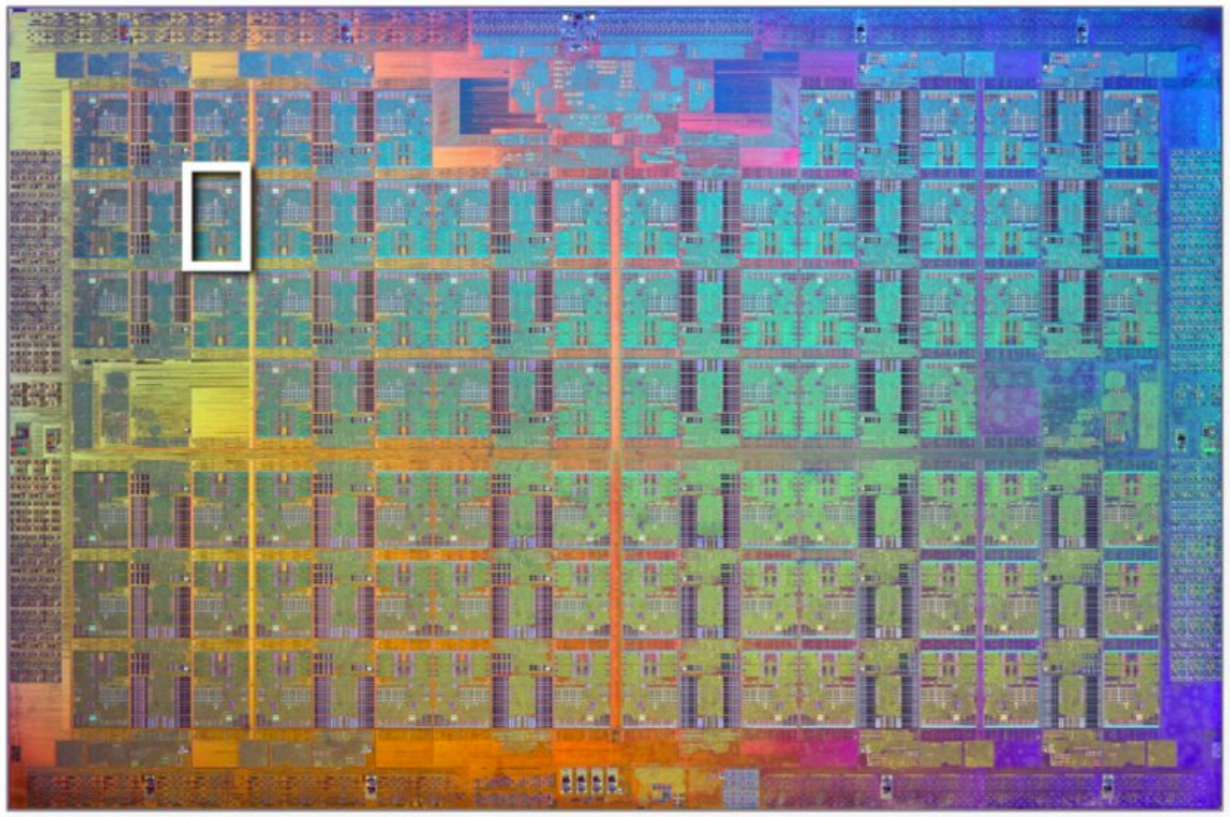
19.96 x 21.66mm
~432.35mm²



M1 Max

ANANDTECH

Example: Xeon Phi



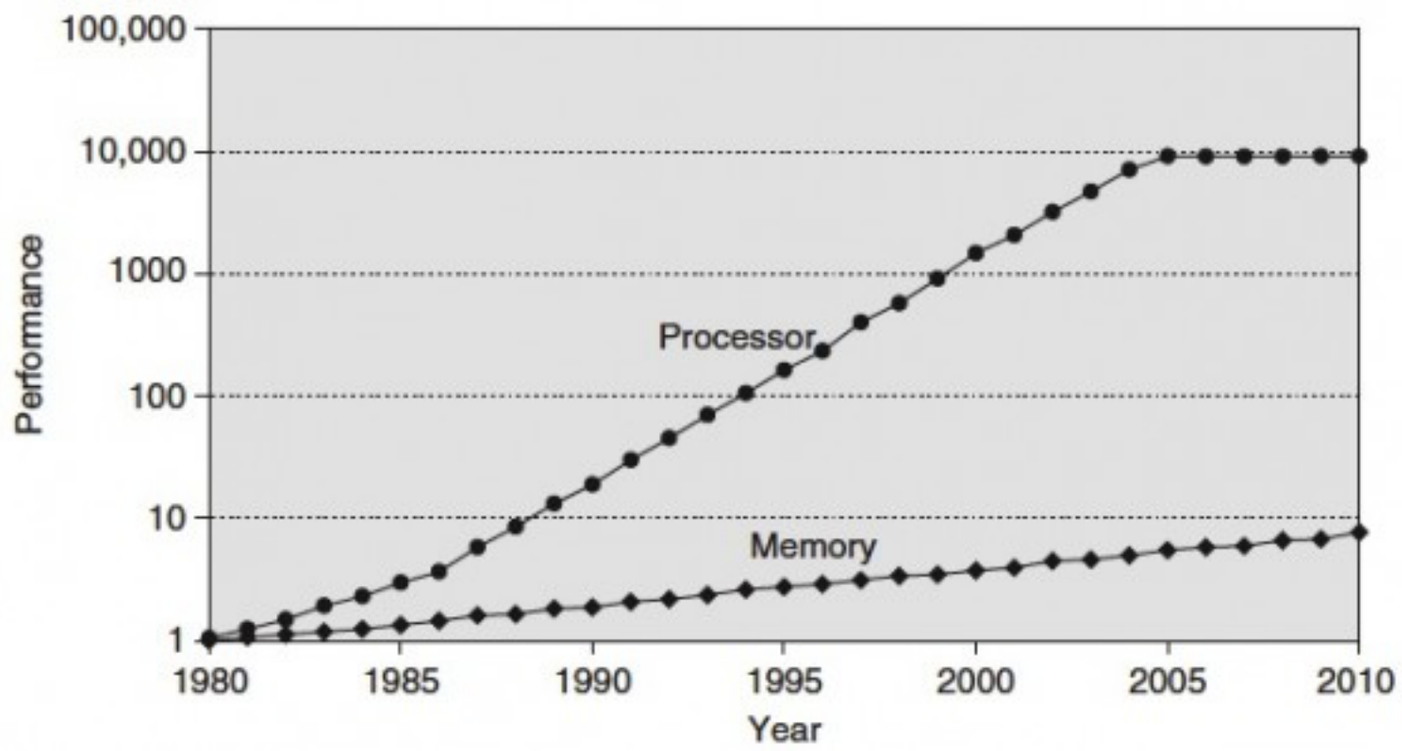
- 72 core CPU. Clock 1.053 - 1.7 GHz. 14 nm
  Support up to AVX-512 SIMD instructions
  Designed to compete with GPUs, discontinued in 2020.

Memory: terminology

- Memory latency: amount of time for a memory request (e.g. load/store) from a processor to be serviced by the memory system
  - e.g. 100 cycles, 100 nsec
    - Memory latency can cause processor stalls, e.g. data does not arrive, and CPU can't process instructions that have a dependency on that data.
- Memory bandwidth: the rate at which the memory system can provide data to a processor
  - E.g. 10 GB/s

Memory architecture

- The importance of the memory architecture has increased with the advances in performance and architecture in CPU.
  - The CPU performance plateau (i.e. evaluating time between processor memory requests - for a single processor or core - and the latency of a DRAM access) is due to the introduction of multi-core architectures.



Source: Computer Architecture, A quantitative Approach by Hennessy and Patterson

Memory bandwidth

- An Intel Core i7-7700K (Q1'17), has about 50GB/s memory bandwidth and 4 cores; if all 4 cores are under equal load, they get about 12.5GB/s each. The clock is ~4.2GHz; assume that with all 4 cores active and hitting memory, so they come in just under 3 bytes per cycle of memory bandwidth.

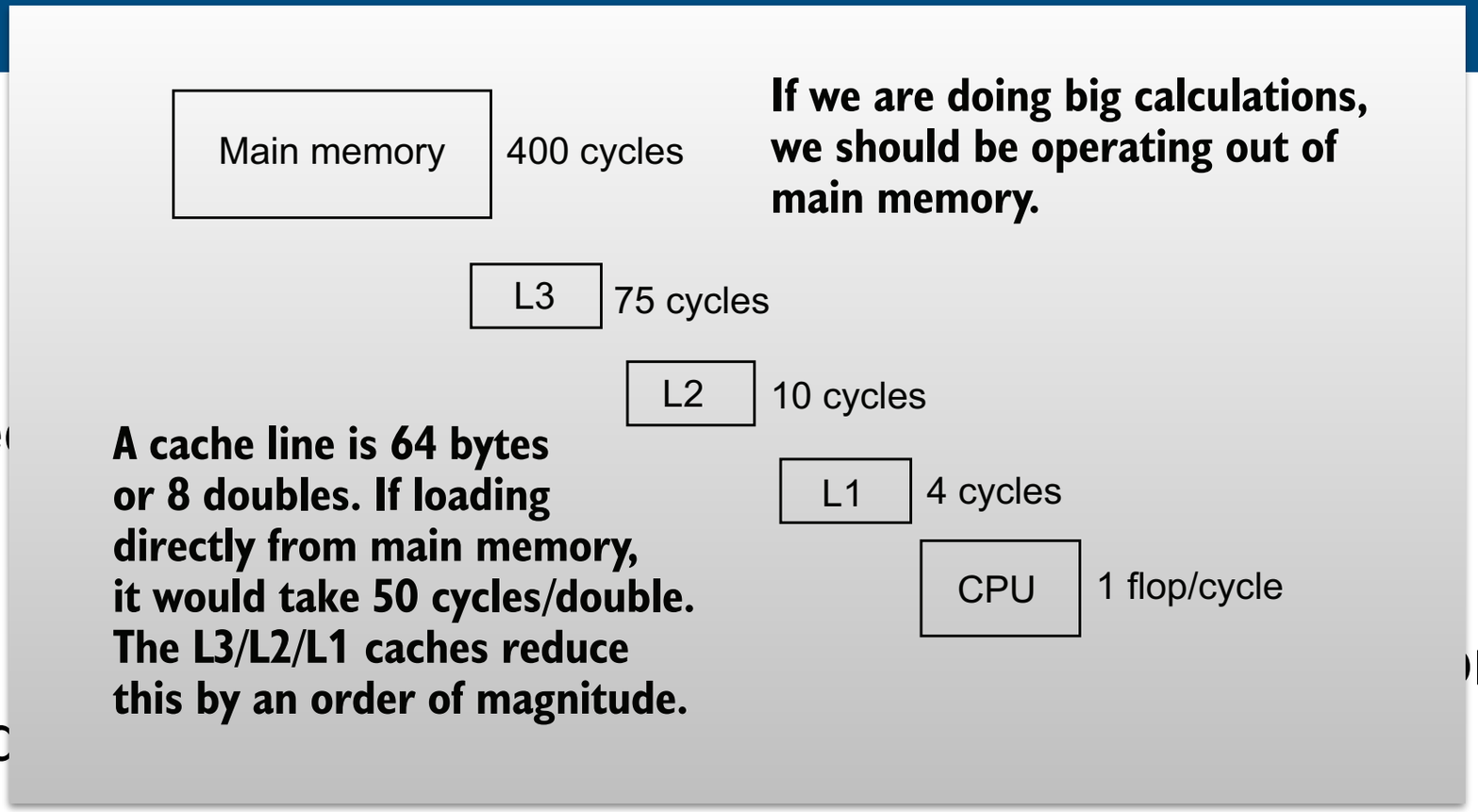
- MOS 6502 (created in '75) had a 4 bytes per cycle of memory bandwidth (1 MHz clock, 1 byte access per clock cycle and 3-5 cycles per instruction)

Memory bandwidth

- Consider a SIMD instruction on Intel Core i7-7700K cores: each core can execute 3 SIMD instructions per clock cycle, operating on 256 bit words;
- let’s consider each 256-bit word as composed by 32 bits vector elements to be computed separately, we get 24 “instructions” executed per cycle and a memory bandwidth of about 0.125 bytes per cycle (or one byte every 8 “instructions”).

Memory / Caches

- There's need of hundreds of CPU cycles to access RAM. To avoid to wait too much for data the solution is to use:
  - Several layers of cache memory: reduce latency (and improves bandwidth too)
  - Prefetch data: even if we write a dumb loop that reads and operates on a large block of 64-bit (8-byte) values, the CPU is smart enough to prefetch the correct data before it's needed. E.g. it may be possible to process at about 22 GB/s on a 3Ghz processor.
    - A calculation that can consume 8 bytes every cycle at 3Ghz only works out to 24GB/s (we're losing just ~8% performance by having to go to main memory)



- There's ne avoid to wait too
  - Several roves bandwic

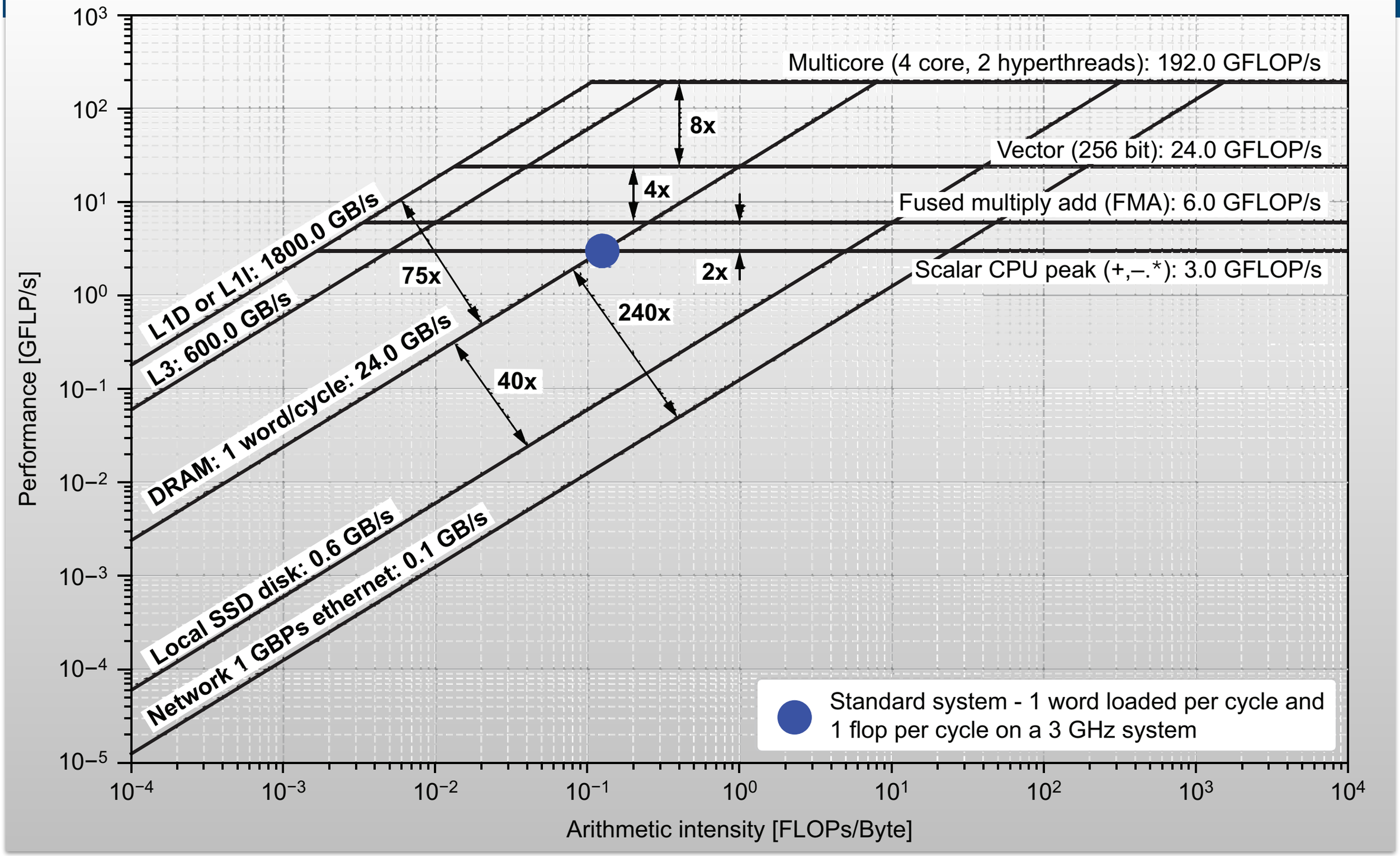
- Prefetch data: even if we write a dumb loop that reads and operates on a large block of 64-bit (8-byte) values, the CPU is smart enough to prefetch the correct data before it's needed. E.g. it may be possible to process at about 22 GB/s on a 3Ghz processor.
  - A calculation that can consume 8 bytes every cycle at 3Ghz only works out to 24GB/s (we're losing just ~8% performance by having to go to main memory)

Memory bandwidth / CPU speed

- Let's consider the 1 word loaded per cycle and 1 flop per cycle as our starting point.
  Most scalar arithmetic operations like addition, subtraction, and multiplication, can be done in 1 cycle. The division operation can take longer at 3–5 cycles. In some arithmetic mixes, 2 flops/cycle are possible with the fused multiply-add instruction. The number of arithmetic operations that can be done increases further with vector units and multi-core processors. Parallelism, greatly increase the flops/cycle.

- Memory performance increase through a deeper hierarchy of caches means that memory accesses can only match the speedup of operations if the data is contained in the L1 cache, typically about 32 KiB.
  But if we only have that much data, we wouldn't be so worried about the time it takes. We really want to operate on large amounts of data that can only be contained in main memory (DRAM) or even on the disk or network.

- Machine balance: the total number of flops that can be executed divided by the memory bandwidth.



- Machine balance: the total number of flops that can be executed divided by the memory bandwidth.

Cache bandwidth

- Data is transported up the memory hierarchy in chunks called cache lines. If memory is not accessed in a contiguous, predictable fashion, the full memory bandwidth is not achieved.
  - E.g.: accessing data in columns for a 2D data structure that is stored in row order will stride across memory by the row length. This can result in as little as one value being used out of each cache line !
- A rough estimate of the memory bandwidth from this data access pattern is 1/8th of the stream bandwidth (1 out of every 8 cache values used).
- This can be generalized for other cases where more cache usage occurs by defining a non-contiguous bandwidth (B_{nc}) in terms of the percentage of cache used (U_{cache}) and the empirical bandwidth (B_E):
- $B_{nc} = U_{cache} \times B_E$ = Average Percentage of Cache Used × Empirical Bandwidth

Theoretical max. FLOPS

- The theoretical maximum flops (F_T) can be calculated with
- $F_T = C_v \times f_c \times I_c$ = Virtual Cores × Clock Rate × Flops/Cycle
- The number of cores includes the effects of hyperthreads that make the physical cores (C_h) appear to be a greater number of virtual or logical cores (C_v) (e.g. hyperthreading).
- The clock rate is the turbo boost rate when all the processors are engaged.
- The flops per cycle, or more generally instructions per cycle (I_c), includes the number of simultaneous operations that can be executed by the vector unit.

Theoretical max. FLOPS

- To determine the number of operations that can be performed, we take the vector width (V_W) and divide by the word size in bits (W_{bits}). We also include the fused multiply-add (FMA) instruction as another factor of two operations per cycle. We refer to this as fused operations (Fops) in the equation. For this specific processor, we get

- $I_c = V_W/W_{bits} \times F_{ops}$
  E.g.: (256-bit Vector Unit/64 bits) × (2 FMA) = 8 Flops/Cycle

- $C_v = C_h \times HT$
  E.g.: (4 Hardware Cores × 2 Hyperthreads)

- Considering a 4 core CPU with HT and 3.7GHz clock:
  F_T = (8 Virtual Cores) × (3.7 GHz) × (8 Flops/Cycle) = 236.8 GFlops/s

Theoretical memory bandwidth

- Memory transfer rate (MTR) is usually given in millions of transfers per sec (MT/s). The double data rate (DDR) memory performs transfers at the top and bottom of the cycle for two transactions per cycle. This means that the memory bus clock rate is half of the transfer rate in MHz. The memory transfer width (T_w) is 64 bits thus 8 bytes are transferred. There are two memory channels (M_c) on most desktop and laptop architectures.
  PCs and laptops usually have only 1 CPU socket (N_s), but HPC servers may have 2 or 4.

- B_T =MTR × M_c × T_w × N_s = DataTransferRate × Memory Channels × Bytes Per Access × Sockets

- E.g. a 16GB MacBook Pro (15-inch, 2018) with Micron 8ATF1G64HZ-2G6E1 chips with MTR os 2666 MT/s on 2 channels and has 1 socket has a 42.7GiB/s memory bandwidth.

Empirical measurement of bandwidth and flops

- The STREAM Benchmark measures the time to read and write a large array. It provides four measures, depending on the operations performed on the data by the CPU as it is being read:
  - the copy: no floating-point work
  - scale and add: one arithmetic operation
  - triad: two arithmetic operation.
- Source: https://github.com/jeffhammond/STREAM.git

Out of Order Execution

- Since several years x86 chips have been able to speculatively execute and re-order execution (to avoid blocking on a single stalled resource).

- But a x86 CPU is required to update externally visible states, like registers and memory, as if everything were executed in order.

- The implementation of this involves making sure that, for any pair of instructions with a dependency, those instructions execute in the correct order with respect to each other.

- This implies also that loads and stores to the same location can’t be reordered with respect to each other.

Operator forwarding

- Pipeline optimization that reduces stalls when waiting for data, e.g. if an operation needs operands from a previous operation:

- ```
  ADD A B C   #A=B+C
  SUB D C A   #D=C-A
  ```

Without operand forwarding

1	2	3	4	5	6	7	8
Fetch ADD	Decode ADD	Read Operands ADD	Execute ADD	Write result			
	Fetch SUB	Decode SUB	*stall*	*stall*	Read Operands SUB	Execute SUB	Write result

With operand forwarding

1	2	3	4	5	6
Fetch ADD	Decode ADD	Read Operands ADD	Execute ADD	Write result	
	Fetch SUB	Decode SUB	*stall*	Read Operands SUB: use result from previous operation	Execute SUB	Write result
  ```

Dekker's algorithm

- Let us consider a simplified version of the Dekker's algorithm for mutual exclusion between two processors. The following code assumes that `flag1` and `flag2` are both initially to `0`:

Thread 1

```
flag1 = 1;
if(flag2==0) {
// critical section
}
flag1=0;
```

Thread2

```
flag2=1;
if(flag1==0) {
// critical section
}
flag2=0;
```

Dekker's algorithm

OoOE and memory access reordering make this algorithm fail !

- Let us consider a simplified version of the Dekker's algorithm for mutual exclusion between two processors. The following code assumes that `flag1` and `flag2` are both initially to `0`:

Thread 1

```
flag1 = 1;
if(flag2==0) {
// critical section
}
flag1=0;
```

Thread2

```
flag2=1;
if(flag1==0) {
// critical section
}
flag2=0;
```

Memory / Concurrency

- x86 loads and stores have some other restrictions:
  - In particular, for a single CPU, stores are never reordered with other stores, and…
    - …stores are never reordered with earlier loads,
  - regardless of whether or not they're to the same location.

In this example, and all the following, will be used the AT&T assembly style: `OP src, dst`
Intel style uses the `OP dst, src` format.

Memory / Concurrency

- x86 loads and … restrictions:
  - In particula… s are never reordered w…
    - …stores ar… earlier loads,
  - regardless of … the same location.

```
mov 1, [%esp]
mov [%ebx], %eax
```

can be executed as

```
mov [%ebx], %eax
mov 1, [%esp]
```

but not vice-versa.

In this example, and all the following, will be used the AT&T assembly style: `OP src, dst`
Intel style uses the `OP dst, src` format.

Memory / Concurrency

- The fact that loads may be reordered with older stores has an effect in multi-core systems:
  - Let us have x and y in shared memory, both initialized to zero, while r1 and r2 are processor registers.

Thread 0	Thread 1
`mov 1, [_x]`	`mov 1, [_y]`
`mov [_y], r1`	`mov [_x], r2`

- When the two threads execute on different cores, the non-intuitive result `r1 == 0` and `r2 == 0` is allowed.
  - Notice that such result is consistent with the processor executing the loads before the stores (which access different locations).

M ncy

This may even break algorithms like the Peterson lock that is used to perform mutual exclusion for two threads.

The solution is to use appropriate additional x86 instructions that force the ordering of the instructions, i.e. memory barriers (aka fences)

- The fact tl as an effect in m
  - Let us have x and y in shared memory, both initialized to zero, while r1 and r2 are processor registers.

Thread 0	Thread 1
`mov 1, [_x]`	`mov 1, [_y]`
`mov [_y], r1`	`mov [_x], r2`

- When the two threads execute on different cores, the non-intuitive result `r1 == 0` and `r2 == 0` is allowed.
  - Notice that such result is consistent with the processor executing the loads before the stores (which access different locations).

Memory fence

- Suppose a thread prepares data to be read by another thread, signaling that it is ready using a variable:

Thread 1	Thread 2
`x = 1`	`if ready == 1`
`ready = 1`	`R = x`

- Initially the globals x and ready are zero. R is a thread-local variable (register). Can Thread 2 see ready == 1 and x == 0? Yes, for two separate reasons. On a relaxed-memory-model multiprocessor:

  1. writes to memory can be completed out of order and

  2. reads from memory can be satisfied out of order.

- Even if the writes are ordered by a (write) fence, the reads in Thread 2 may still execute out of order. Imagine that Thread 2 asks for the values of ready and x. The request for x arrives first, before any writes by Thread 1 are done. The memory sends back the value 0 for x. Next, the two writes by Thread 1 are committed. Then the first read message (fetch ready) arrives, and the memory responds with the value 1. Thread 2 sees a non-zero value of ready, but a zero (uninitialized) value of x.

- Notice that the read of x is speculative. The processor issues the read request just in case the branch ready == 1 were taken. If it's not, it can always abandon the speculation.

Memory fence

Solution:

Have a write fence and a read fence. Both fences are necessary!

Thread 1	Thread 2
`x = 1`	`if ready == 1`
`write fence`	`read fence`
`ready = 1`	`R = x`

In Java, marking a shared variable `volatile`, tells the compiler to issue memory fences on every access.

In C++ **atomics**, are implemented with all the fencing in place.

In general monitors and locks, have appropriate fences (or their equivalents) built in.

of x.

- Notice that the read of x is speculative. The processor issues the read request just in case the branch ready == 1 were taken. If it's not, it can always abandon the speculation.

Memory / Concurrency

- There's also multi-core ordering. The previous restrictions all apply; if core0 is observing core1, it will see that all of the single core rules apply to core1's loads and stores.
- However, if core0 and core1 interact, there's no guarantee that their interaction is ordered:
- For example, say that core0 and core1 start with `eax` and `edx` set to `0`, `[_foo]` and `[_bar]` in shared memory set to `0`, and core0 executes

```
mov 1, [_foo] ; move 1 to the bytes in memory at address _foo

mov [_foo], %eax ; move the content of the memory addressed by _foo in EAX

mov [_bar], %edx ; move the content of the memory addressed by _bar in EDX
```

- while core1 executes

```
mov 1, [_bar] ; move 1 to the bytes in memory at address _bar

mov [_bar], %eax ; move the content of the memory addressed by _bar in EAX

mov [_foo], %edx ; move the content of the memory addressed by _foo in EDX
```

- For both cores, `eax` has to be 1 because of the within-core dependency between the first instruction and the second instruction. However, it's possible for `edx` to be `0` in both cores because line 3 of core0 can execute before core0 sees anything from core1, and vice-versa.

Memory / Concurrency

- There's also multi-core ordering. The previous restrictions all apply; if core0 is observing core1, it will see that all of the single core rules apply to core1's loads and stores.
- However, if core0 and core1 interact, there's no guarantee that their interaction is ordered:
- For example, say that core0 and core1 start with `eax` and `edx` set to `0`, `[_foo]` and `[_bar]` in shared memory set to `0`, and core0 executes

```
mov 1, [_foo] ; move 1 to the bytes in memory at address _foo

mov [_foo], %eax ; move the content of the memory addressed by _foo in EAX

mov [_bar], %edx ; move the content of the memory addressed by _bar in EDX
```

- while core1 executes

```
mov 1, [_bar] ; move 1 to the bytes in memory at address _bar

mov [_bar], %eax ; move the content of the memory addressed by _bar in EAX

mov [_foo], %edx ; move the content of the memory addressed by _foo in EDX
```

- For both cores, `eax` has to be 1 because of the within-core dependency between the first instruction and the second instruction. However, it's possible for `edx` to be `0` in both cores because line 3 of core0 can execute before core0 sees anything from core1, and vice-versa.

To solve these issues use locking instead of memory barriers.

Locking

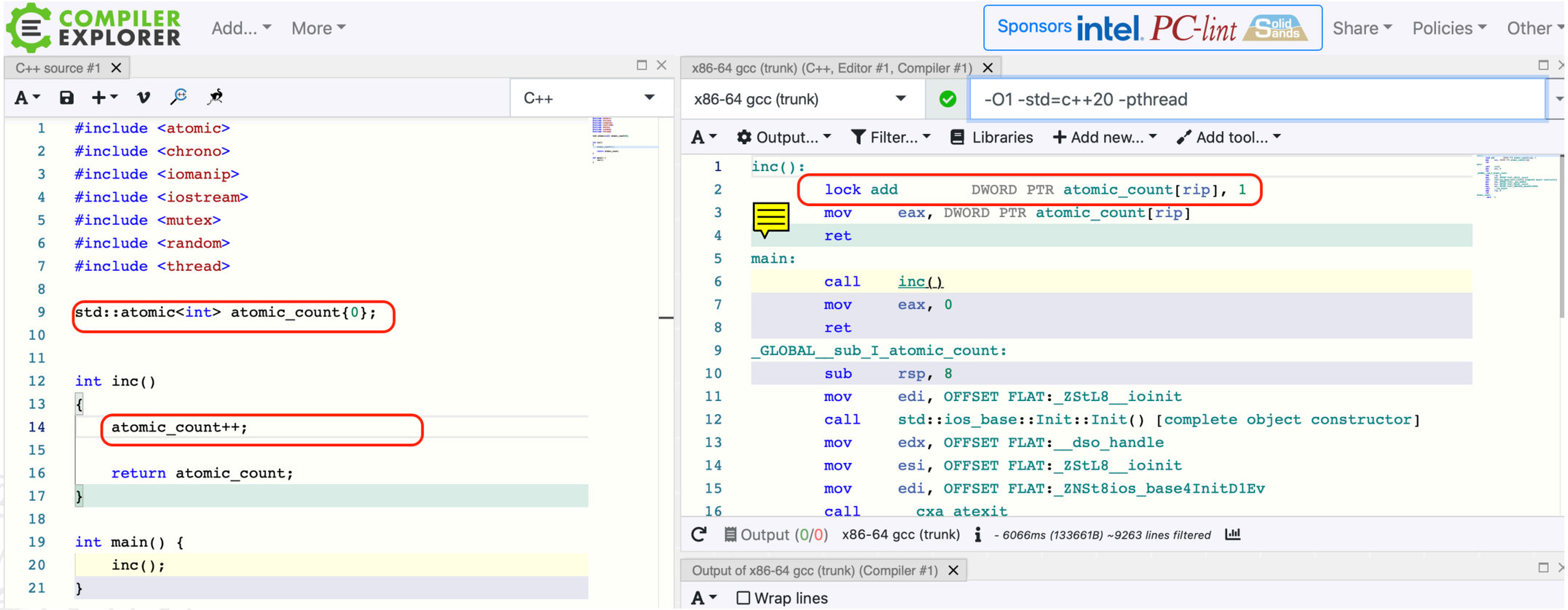
- In modern x86 CPUs locking is cheaper than memory barriers.

- It is possible to lock some instructions using the `lock` prefix

- In addition to making a memory transaction atomic, locks are globally ordered with respect to each other, and loads and stores aren't re-ordered with respect to locks.

Lock and atomics

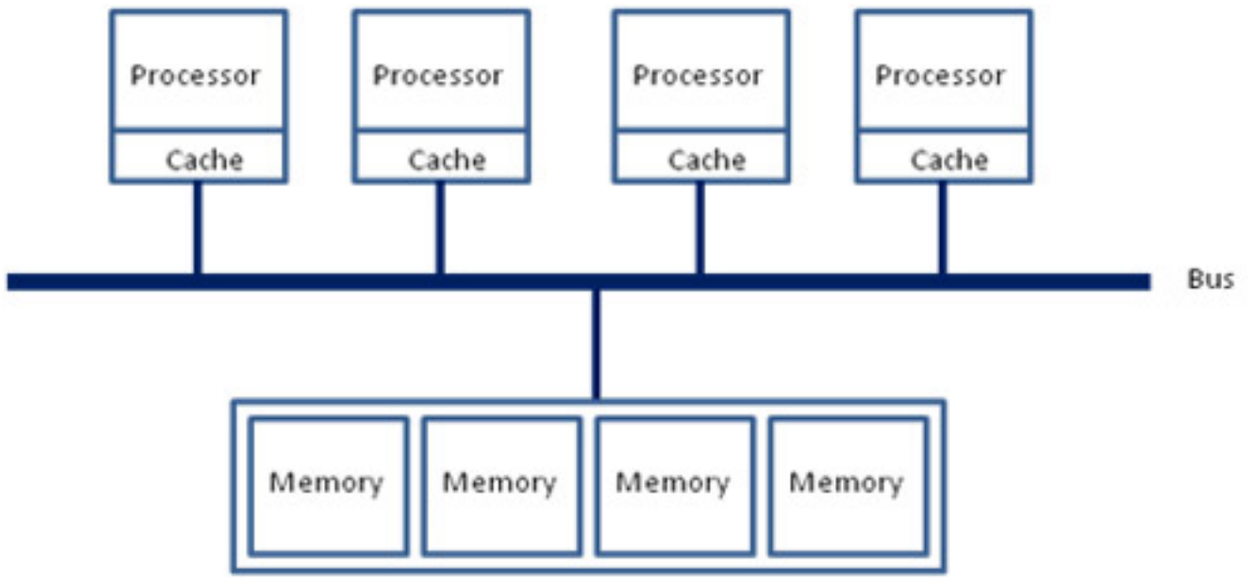
- Locking is used to implement atomic variables and operations. E.g. C++11 atomics.



Memory/ UMA

- Shared Memory Architecture is split up in two types: Uniform Memory Access (UMA), and Non-Uniform Memory Access (NUMA).

- UMA: memory is shared across all CPUs. To access memory, the CPUs have to access a Memory Controller Hub (MCH). This type of architecture is limited on scalability, bandwidth and latency.

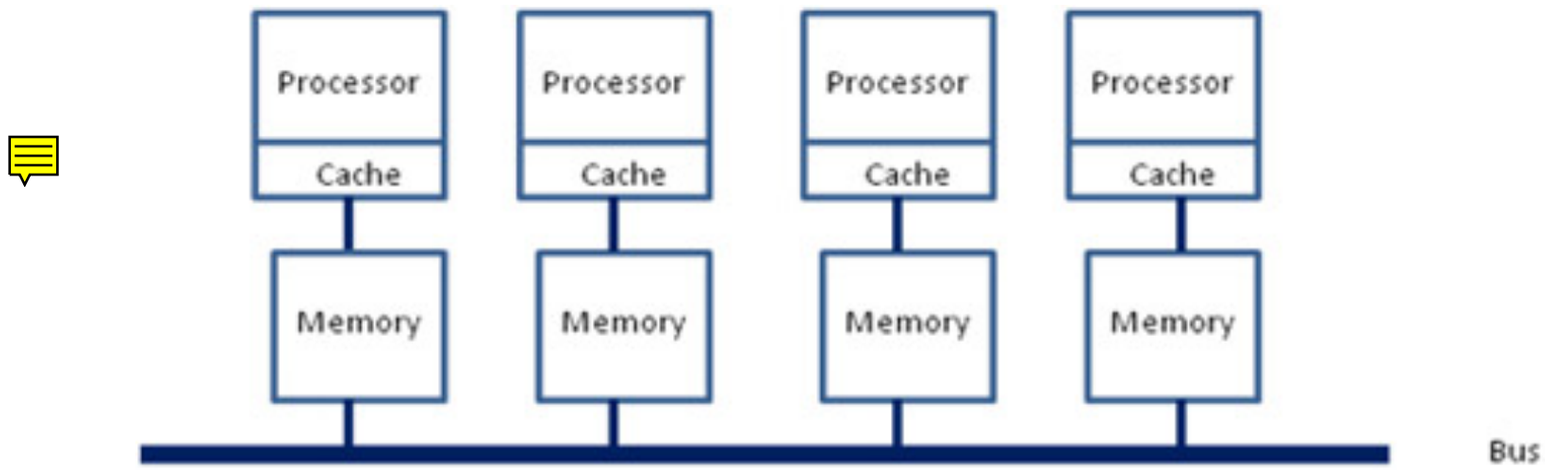


Memory / NUMA

- With NUMA memory is directly attached to the CPU and this is considered to be local.

- Memory connected to another CPU socket is considered to be remote. To access it there is need to traverse the interconnect and connect to the remote memory controller.

- As a result of the different locations memory can exists, this system experiences “non-uniform” memory access time.



Memory / NUMA

- Non-uniform memory access, where memory latencies and bandwidth are different for different processors, is so common that we can we assume it as default.

- Threads that share memory should be on the same socket, and a memory-mapped I/O heavy thread should make sure it's on the socket that's closest to the I/O device it's talking to.

Memory / NUMA

- It’s hard to sync more than 4 CPUs and their caches w.r.t. memory: each one has to warn the others about the memory that it’s going to touch… the bus would saturate.

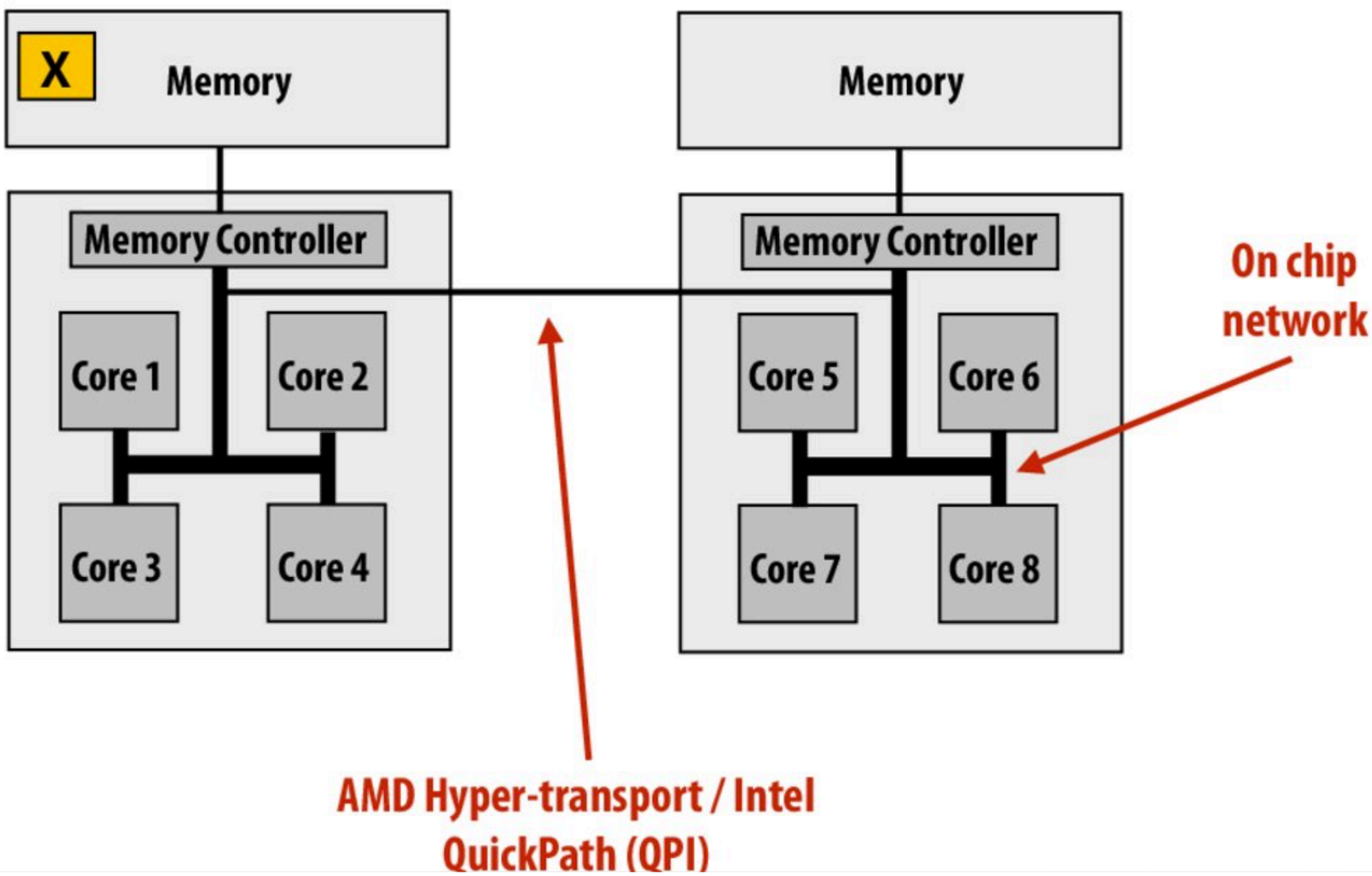
- Multi-core CPUs have a directory to reduce the N-way peer-to-peer broadcast, but the problem with CPUs is still valid.

- The simplest solution is to have each socket to control some region of memory. You pay e penalty performance when trying to access memory assigned to other chips.

- Typically <128 CPUs system have a ring-like bus: you 1) pay the latency/bandwidth penalty for walking through a bunch of extra hops to get to memory, 2) use a finite resource (the ring-like bus) and 3) slow down other cross-socket accesses.

- All this is handled by the O.S.

Memory / NUMA



- All this is handled by the O.S.

Cache coherence

- The cache solution to the memory latency problem becomes complicated when multiple CPUs or cores share a single main memory.
- How do we:
  - Maintain the cache structure to hide latency,
  - While ensuring that cores all see a consistent view of memory?
- Say core 1 reads a memory location X and pulls in its neighbors on a cache line into its L1 cache. Core 2 then writes to memory location X, changing the value in its own cache and main memory.
  - How do we ensure that core 1 does not use the out of date copy of location X?

Cache coherence

- Cache coherence protocols maintain this coherent view of memory for each core
- When memory is accessed (either read or written) caches observe activity on the bus and update the state of the data they contain
  - Invalidating it if it is no longer up to date
  - Marking values that are replicated between caches as shared
  - Marking values as modified
  - Establishing exclusive ownership of a memory location

Cache coherence

- The protocol defines a state machine that tells the cache how to treat data that it contains.
  - Cache reads a value from memory that no other core has read, marks it as exclusively owned.
  - Cache observes another core reading the same memory, transitions the state from exclusive to shared.
  - Cache observes another core write to that memory, transitions the state to invalid to force an update if it is accessed again.
- Many protocols exist, and different ones are used by different CPU manufacturers.

Cache coherence

Examples of protocols

Snooping Cache Coherence:

- The cores share a bus.
- Any signal transmitted on the bus can be “seen” by all cores connected to the bus.
- When core 0 updates the copy of x stored in its cache it also broadcasts this information across the bus.
- If core 1 is “snooping” the bus, it will see that x has been updated and it can mark its copy of x as invalid.

Directory Based Cache Coherence:

- Uses a data structure called a directory that stores the status of each cache line.
- When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable’s cache line in their caches are invalidated.

Cache coherence

- A parallel algorithm should avoid causing the CC protocol to frequently invalidate cached data, which would result in high frequency access to slow main memory.

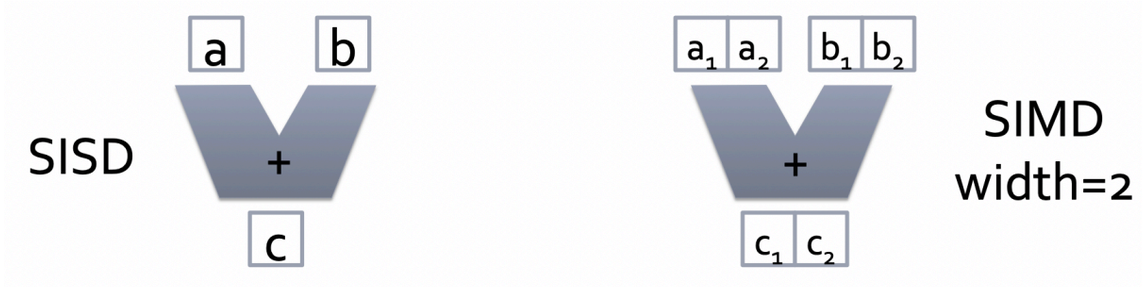
- The issue is called “false sharing”:
  - A cache line contains more than one machine word.
  - When multiple processors access the same cache line, it may look like a potential race condition, even if they access different elements.
  - Can cause coherence traffic.

Context Switches / Syscalls

- A side effect of all the caching that modern cores have is that context switches are expensive. In HPC it is better to:
  - use a thread-per-core rather than thread-per-logical-task
  - use userspace I/O stacks for very high performance I/O.

SIMD

- all modern x86 CPUs support SSE, 128-bit wide vector registers and instructions.
- Since it's common to want to do the same operation multiple times, Intel added instructions that will let you operate on
  - a 128-bit chunk of data as 2 64-bit chunks,
  - 4 32-bit chunks,
  - 8 16-bit chunks,
  - etc.



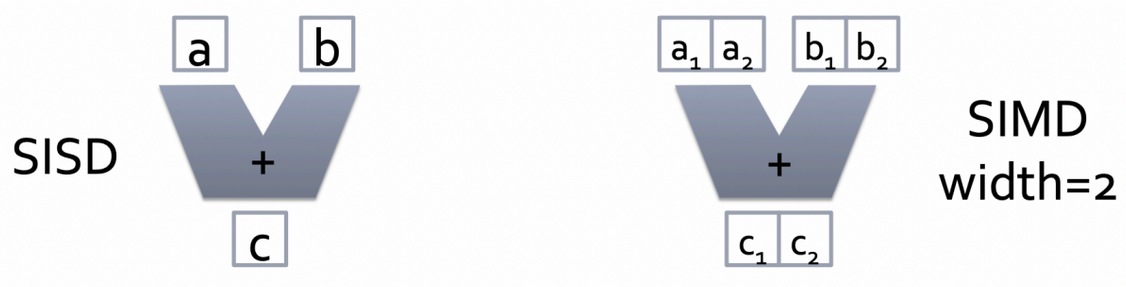
- ARM supports the same thing with a different name (NEON).

SIMD

- all modern x86 CPUs support SSE, 128-bit wide vector registers and instructions.

AVX2 instructions: 256 bit operations: 8x32 bits or 4x64 bits
AVX512 instruction: 512 operations: 16x32 bits…

- Since it's common to want to do the same operation multiple times, Intel added instructions that will let you operate on
  - a 128-bit chunk of data as 2 64-bit chunks,
  - 4 32-bit chunks,
  - 8 16-bit chunks,
  - etc.



- ARM supports the same thing with a different name (NEON).

Example: SIMD

Let us suppose to have v1 and v2 that are 4-dim float vectors, and we want to sum them with SSE instructions:

```
movaps [v1], xmm0 ;xmm0 = v1.w | v1.z | v1.y | v1.x
addps  [v2], xmm0 ;xmm0 = v1.w+v2.w | v1.z+v2.z
                  ;     | v1.y+v2.y | v1.x+v2.x
movaps xmm0, [vec_res]
```

it's faster than performing 4 `fadd` on the single components of the vectors

SIMD

- Compilers are good enough at recognizing common patterns that can be vectorized in simple code.

- E.g. the following code, will automatically use vector instructions with modern compilers:

```
for (int i = 0; i < n; ++i) {

  sum += a[i];

}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

SIMD

Compiler Explorer C++ Editor Diff View More Share

C++ source #1

```
1  int sum(int* a, int num) {
2    int sum = 0;
3    for(int i=0; i<num; i++)
4      sum += a[i];
5    return sum;
6  }
7
```

x86-64 clang 5.0.0 (Editor #1, Compiler #1)

x86-64 clang 5.0.0 -O0 -fno-unroll-loops

11010 .LX0: .text // \s+ Intel Demangle

```
1  sum(int*, int):                              # @sum(int*, int)
2          push    rbp
3          mov     rbp, rsp
4          mov     qword ptr [rbp - 8], rdi
5          mov     dword ptr [rbp - 12], esi
6          mov     dword ptr [rbp - 16], 0
7          mov     dword ptr [rbp - 20], 0
8  .LBB0_1:                                     # =>This Inner Loop Header: Depth=1
9          mov     eax, dword ptr [rbp - 20]
10         cmp     eax, dword ptr [rbp - 12]
11         jge     .LBB0_4
12         mov     rax, qword ptr [rbp - 8]
13         movsxd  rcx, dword ptr [rbp - 20]
14         mov     edx, dword ptr [rax + 4*rcx]
15         add     edx, dword ptr [rbp - 16]
16         mov     dword ptr [rbp - 16], edx
17         mov     eax, dword ptr [rbp - 20]
18         add     eax, 1
19         mov     dword ptr [rbp - 20], eax
20         jmp     .LBB0_1
21 .LBB0_4:
22         mov     eax, dword ptr [rbp - 16]
23         pop     rbp
24         ret
```

UNIVERSITÀ DEGLI STUDI FIRENZE

SIMD

Compiler Explorer C++ Editor Diff View More Share

C++ source #1

```
1  int sum(int* a, int num) {
2    int sum = 0;
3    for(int i=0; i<num; i++)
4      sum += a[i];
5    return sum;
6  }
7
```

x86-64 clang 5.0.0 (Editor #1, Compiler #1)

x86-64 clang 5.0.0 | -O3 -fno-unroll-loops | 11010 | .LX0: | .text | // | \s+ | Intel | Demangle

```
8         xor     eax, eax
9         jmp     .LBB0_4
10 .LBB0_1:
11        xor     eax, eax
12        ret
13 .LBB0_6:
14        and     esi, 3
15        mov     r8, rcx
16        sub     r8, rsi
17        pxor    xmm0, xmm0
18        mov     rax, r8
19        mov     rdx, rdi
20 .LBB0_7:                                # =>This Inner Loop Header: Depth=1
21        movdqu  xmm1, xmmword ptr [rdx]
22        paddd   xmm0, xmm1
23        add     rdx, 16
24        add     rax, -4
25        jne     .LBB0_7
26        pshufd  xmm1, xmm0, 78          # xmm1 = xmm0[2,3,0,1]
27        paddd   xmm1, xmm0
28        pshufd  xmm0, xmm1, 229         # xmm0 = xmm1[1,1,2,3]
29        paddd   xmm0, xmm1
30        movd    eax, xmm0
31        test    esi, esi
32        je      .LBB0_9
33 .LBB0_4:
34        lea     rsi, [rdi + 4*r8]
35        sub     rcx, r8
36 .LBB0_5:                                # =>This Inner Loop Header: Depth=1
37        add     eax, dword ptr [rsi]
38        add     rsi, 4
39        dec     rcx
40        jne     .LBB0_5
41 .LBB0_9:
42        ret
```

UNIVERSITÀ DEGLI STUDI FIRENZE

SIMD

Compiler Explorer C++ Editor Diff View More Share

C++ source #1

```
1  int sum(int* a, int num) {
2    int sum = 0;
3    for(int i=0; i<num; i++)
4      sum += a[i];
5    return sum;
6  }
7
```

x86-64 clang 5.0.0 (Editor #1, Compiler #1)

x86-64 clang 5.0.0 | -O3 -fno-unroll-loops

11010 .LX0: .text // \s+ Intel Demangle

```
8         xor     eax, eax
9         jmp     .LBB0_4
10 .LBB0_1:
11        xor     eax, eax
12        ret
13 .LBB0_6:
14        and     esi, 3
15        mov     r8, rcx
16        sub     r8, rsi
17        pxor    xmm0, xmm0
18        mov     rax, r8
19        mov     rdx, rdi
20 .LBB0_7:                                # =>This Inner Loop Header: Depth=1
21        movdqu  xmm1, xmmword ptr [rdx]
22        paddd   xmm0, xmm1
23        add     rdx, 16
24        add     rax, -4
25        jne     .LBB0_7
26        pshufd  xmm1, xmm0, 78          # xmm1 = xmm0[2,3,0,1]
27        paddd   xmm1, xmm0
28        pshufd  xmm0, xmm1, 229         # xmm0 = xmm1[1,1,2,3]
29        paddd   xmm0, xmm1
30        movd    eax, xmm0
31        test    esi, esi
32        je      .LBB0_9
33 .LBB0_4:
34        lea     rsi, [rdi + 4*r8]
35        sub     rcx, r8
36 .LBB0_5:                                # =>This Inner Loop Header: Depth=1
37        add     eax, dword ptr [rsi]
38        add     rsi, 4
39        dec     rcx
40        jne     .LBB0_5
41 .LBB0_9:
42        ret
```

Read from memory

UNIVERSITÀ DEGLI STUDI FIRENZE

SIMD

Compiler Explorer C++ Editor Diff View More Share

C++ source #1

```
1  int sum(int* a, int num) {
2    int sum = 0;
3    for(int i=0; i<num; i++)
4      sum += a[i];
5    return sum;
6  }
7
```

x86-64 clang 5.0.0 (Editor #1, Compiler #1)

x86-64 clang 5.0.0 | -O3 -fno-unroll-loops | 11010 | .LX0: | .text | // | \s+ | Intel | Demangle

```
8         xor     eax, eax
9         jmp     .LBB0_4
10 .LBB0_1:
11        xor     eax, eax
12        ret
13 .LBB0_6:
14        and     esi, 3
15        mov     r8, rcx
16        sub     r8, rsi
17        pxor    xmm0, xmm0
18        mov     rax, r8
19        mov     rdx, rdi
20 .LBB0_7:                                # =>This Inner Loop Header: Depth=1
21        movdqu  xmm1, xmmword ptr [rdx]
22        paddd   xmm0, xmm1
23        add     rdx, 16
24        add     rax, -4
25        jne     .LBB0_7
26        pshufd  xmm1, xmm0, 78          # xmm1 = xmm0[2,3,0,1]
27        paddd   xmm1, xmm0
          pshufd  xmm0, xmm1, 229         # xmm0 = xmm1[1,1,2,3]
          paddd   xmm0, xmm1
          movd    eax, xmm0
          test    esi, esi
32        je      .LBB0_9
33 .LBB0_4:
34        lea     rsi, [rdi + 4*r8]
35        sub     rcx, r8
36 .LBB0_5:                                # =>This Inner Loop Header: Depth=1
37        add     eax, dword ptr [rsi]
38        add     rsi, 4
39        dec     rcx
40        jne     .LBB0_5
41 .LBB0_9:
42        ret
```

Read from memory

Xmm1 = [0+2, 1+3, 2+0, 3+1]

Xmm0 = [0+2 + 1+3, 1+3 + 1+3, 2+0 + 2+0, 3+1 + 3+1]

Eax = [0+2 + 1+3]

Different compilers may shuffle differently the xmm registers

SIMD

- It is possible to use directly SIMD instructions like SSE in C/C++, using “intrinsic” / “builtin” functions, that are directly mapped to CPU instructions.
- Include the required header and ask the compiler to activate the required SSE level
  - <mmintrin.h> MMX
  - <xmmintrin.h> SSE
  - <emmintrin.h> SSE2
  - <pmmintrin.h> SSE3
  - <tmmintrin.h> SSSE3
  - <smmintrin.h> SSE4.1
  - <nmmintrin.h> SSE4.2
  - <ammintrin.h> SSE4A
  - <wmmintrin.h> AES
  - <immintrin.h> AVX

SIMD

- More recent vector units also implement a gather/ scatter memory load operation where the data loaded into the vector unit does not have to be in contiguous memory locations (gather) and the store from the vector to memory does not have to be contiguous memory locations (scatter).

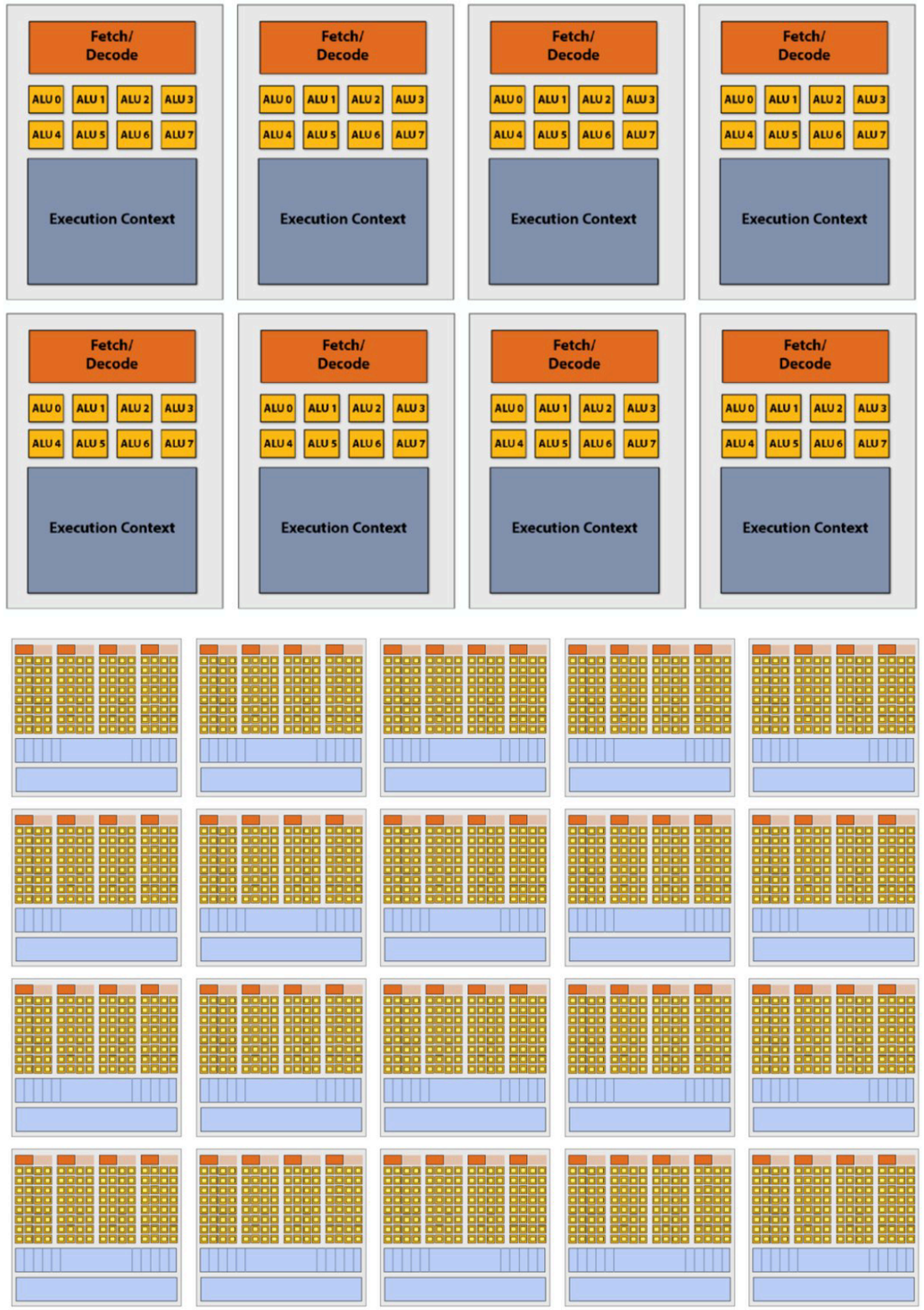
- Useful in HPC (e.g. for sparse linear algebra)

- Implemented in Xeon Phi and AVX-512

Cores and SIMD ALUs

- Example: 8 core Intel Xeon E5-1660v4
  8 cores, 8 SIMD ALUs per core (AVX2 instructions)

- Example: NVIDIA GTX 1080
  20 Streaming Multiprocessors (i.e. cores)
  128 SIMD ALUs per core
  = 20*128 = 2560 CUDA cores



Simultaneous Multi Threading / Hyper-Threading

- Hyper-Threading Technology is a form of simultaneous multithreading.
- Architecturally, a processor with HT consists of two logical processors per core, each of which has its own processor architectural state.
  - Each logical processor can be individually halted, interrupted or directed to execute a specified thread, independently from the other logical processor sharing the same physical core.
- The goal is to improve ILP: if the instructions in a thread have too many dependencies perhaps in the other thread they do not, and it is possible to parallelize them
  - Intel CPUs are superscalar: they can execute many instructions in parallel. HT is an extension of superscalar CPU design.

Simultaneous Multi Threading / Hyper-Threading

- The logical processors in a hyper-threaded core share the execution resources.
- These resources include:
  - the execution engine
  - caches
  - system bus interface;
- the sharing of resources allows two logical processors to work with each other more efficiently.
- This works by duplicating certain sections of the processor (5% more die area) - those that store the architectural state - but not duplicating the main execution resources.
- Xeon Phi processors had up to 4-way HT.

Simultaneous Multi Threading / Hyper-Threading

- Performance improvements are very application-dependent
  - high clock CPUs have long pipelines. The CPU scheduler may be optimistic to fill the pipeline with instructions, but they may not be executable (e.g. because of missing data)
  - a specific part of the CPU (replay system in Pentium4) catches those instructions that can not be executed and reissues them until they execute successfully, occupying the execution units
  - if the execution units sits idly (typically 66% time) it’s not a problem, but in HT this interferes with the execution of the instructions of the other thread.
    - CPUs with a replay queue, that check if the data can be retrieved from the various cache levels or waits for it from memory, suffer less from this.

Intel Turbo-Boost

- It’s dynamic over-clocking of some Intel CPUs
- It is activated when the operating system requests the highest performance state of the processor. These performance states are defined by the open standard Advanced Configuration and Power Interface (ACPI) specification, supported by all major operating systems.
- The increased clock rate is limited by the processor's power, current and thermal limits, as well as the number of cores currently in use and the maximum frequency of the active cores

Credits

- These slides report material from:
  - Dan Luu (Microsoft)
  - Bartosz Milewski (Reliable Software, University of Washington)
  - Kayvon Fatahalian and Randal Bryant (CMU)

Books

- Parallel and High Performance Computing, R. Robey, Y. Zamora, Manning - Chapt. 3